IMPLEMENTATION OF MBRACE FOR LARGE-SCALE CLOUD COMPUTING

Diploma project

Kostas Rontogiannis

July 2015

National Technical University of Athens
School of Electrical and Computer Engineering

WHAT IS MBRACE?

- `MBrace` is a simple programming model for cloud programming.
- Written in F#/.NET.
- Nessos Information Technologies.
- Open source `https://github.com/mbraceproject`.
- Refactored from a monolithic framework to a runtime agnostic model (`MBrace.Core`).

- The `MBrace` programming model.
- Implement a runtime for `MBrace` on top of Microsoft Azure (`MBrace.Azure`).

PROGRAMMING MODEL

- `MBrace` programming model.
- Basic data structures and algorithms.
- Foundations for implementing `MBrace` runtimes.

- `MBrace` is based on F# computation expressions.
- Give custom semantics to F# code.
- Additional grammar rule : $\langle \mathit{expr} \rangle := \langle \mathit{cbuilder} \rangle$ ‘**{**’ $\langle \mathit{cexpr} \rangle$ ‘**}**’

SYNTAX

$\langle \mathit{cexpr} \rangle$:= **‘do!’** < expr >
| **‘let!’** <pat> **‘=’** <expr> **‘in’** <cexpr>
| **‘let’** <pat> **‘=’** <expr> **‘in’** <cexpr>
| **‘return!’** <expr>
| **‘return’** <expr>
| <cexpr> **‘;’** <cexpr>
| **‘if’** <expr> **‘then’** <cexpr> **‘else’** <cexpr>
| **‘match’** <expr> **‘with’** <pat> **‘->’** <cexpr>
| **‘while’** <expr> **‘do’** <cexpr>
| **‘for’** <pat> **‘in’** <expr> **‘do’** <cexpr>
| **‘use’** <val> **‘=’** <expr> **‘in’** <cexpr>
| **‘use!’** <val> **‘=’** <expr> **‘in’** <cexpr>
| **‘try’** <cexpr> **‘with’** <pat> **‘->’** <cexpr>
| **‘try’** <cexpr> **‘finally’** <expr>
| <expr>

Expression		Desugaring
⟦**let** binding **in** *cexpr*⟧	=	**let** binding **in** ⟦*cexpr*⟧
⟦**let!** pattern = expr **in** *cexpr*⟧	=	cbuilder.Bind(expr, (**fun** pattern -> ⟦*cexpr*⟧))
⟦**return** expr⟧	=	cbuilder.Return(expr)
⟦**return!** expr⟧	=	cbuilder.ReturnFrom(expr)
	...	

Defining a custom computation expression (like the cloud workflow) is straightforward by creating a `cbuilder` class and defining certain special methods on the class like `Bind, Return`, etc.

```
let getLength (uri : Uri) : int =
        let client = new System.Net.WebClient()
        let html = client.DownloadString(uri)
        html.Length
```

```
let getLengthAsync (uri : Uri) : Async<int> =
    async {
        let client = new System.Net.WebClient()
        let! html = client.AsyncDownloadString(uri)
        return html.Length
    }
```

- The effect of let! is to enable execution to continue on other threads.
- After the right side of the let! binding returns, the rest of the asynchronous workflow resumes execution.

MONADS

Computation expressions provide convenient syntax for monads.

```
Method       Signature

Bind       : M<'T> * ('T → M<'U>) → M<'U>
Return     : 'T → M<'T>
```

`Cloud<T>` is the basic type for all cloud workflows.

```
let helloWorld : Cloud<int> =
    cloud {
        return 42
    }

let runtime = Runtime.GetHandle(azureConfiguration)
let result = runtime.Run helloWorld

val result : int = 42
```

Example 1: The `MBrace` hello world

Parallelism

```
Cloud.Parallel : Cloud<'T> seq → Cloud<'T []>
```

```
Cloud.Parallel : Cloud<'T> seq → Cloud<'T []>
```

```
let parallelWorkflow : Cloud<int []> =
    cloud {
        let compute i = cloud { return i * i }
        let! results =
            [1..10]
            |> List.map compute
            |> Cloud.Parallel
        return results
    }

// Evaluate workflow
runtime.Run(parallelWorkflow)

val it : int [] = [|1;4;9;16;25;36;49;64;81;100|]
```

```
Cloud.Choice : Cloud<'T option> seq → Cloud<'T option>
```

```
Cloud.Choice : Cloud<'T option> seq → Cloud<'T option>
```

```
let findIndex (xs : 'T []) (item : 'T) : Cloud<int option> =
    cloud {
        return! xs |> Seq.mapi (fun index x → index, x)
                   |> Seq.map (fun (i,x) →
                         cloud {
                             return
                                 if x = item then Some i
                                 else None
                         })
                   |> Cloud.Choice
    }

// Evaluate workflow
runtime.Run(findIndex [|0..100|] 42)
val it : int option = Some 42
```

STARTASTASK

$\boxed{\texttt{Cloud.StartAsTask : Cloud<'T>} \rightarrow \texttt{Cloud<ICloudTask<'T>>}}$

```
Cloud.StartAsTask : Cloud<'T> → Cloud<ICloudTask<'T>>
```

```
let workflow : Cloud<int * TimeSpan> =
    cloud {
        // Spawn a CloudTask
        let! ctask =
            Cloud.StartAsTask(
                cloud {
                    do! Cloud.Sleep 20000
                    return 42
                })

        let watch = Stopwatch.StartNew()
        // Block until ctask completes
        let! value = ctask.AwaitResult()
        watch.Stop()

        return value, watch.Elapsed
    } // (42, 00:00:22.4323827)
```

mbrace.flow

`MBrace.Flow` is Spark-like library, that provides and easy way to describe data parallel streaming computations, with caching and in-memory computation capabilities.

`MBrace.Flow` is Spark-like library, that provides and easy way to describe data parallel streaming computations, with caching and in-memory computation capabilities.

```
open MBrace.Flow

let workflow : Cloud<(string * int64) []> =
    CloudFlow.OfCloudFileByLine "foobar.txt"
    |> CloudFlow.collect(fun line → line.Split(' '))
    |> CloudFlow.filter(fun word → word.Length > 3)
    |> CloudFlow.countBy id
    |> CloudFlow.toArray
```

Cancellation Cooperative cancellation like .NET; using cancellation tokens.

Exception handling Exceptions can be raised, handled and transferred between machines.

Job scheduling Force execution of jobs in specific workers.

Constraining execution Suppress any distribution effects, force computation evaluation in-memory, etc.

PITFALLS

```
let workflow : Cloud<int> =
    cloud {
        let cell = ref 0
        do! [1..10]
            |> List.map (fun _ →
                 cloud { cell := !cell + 1 })
            |> Cloud.Parallel
            |> Cloud.Ignore
        return !cell
    }

runtime.Run workflow
val it : int = 0
```

Example 3: Value mutation and race conditions

```
let workflow =
    cloud {
      let largeObj = [|1..10000000|]
      return! [1..10]
              |> List.map (fun _ →
                cloud { return doSomething(largeObj) })
              |> Cloud.Parallel
    }
```

Example 4: Capturing large objects

`MBrace` offers a plethora of abstractions for managing data in a global, machine-wide scope.

- Lightweight reference to persisted data.
- Immutable, can be either initialized or dereferenced.
- Can be cached in-memory.

```
let workflow =
    cloud {
        let! cvalue = CloudValue.New [|1..10000000|]
        return! [1..10]
                |> List.map (fun _ → cloud {
                    // Dereference and compute
                    let! value = cvalue.Value
                    return doSomething(value) })
                |> Cloud.Parallel
    }
```

Example 5: Using a `CloudValue`

CloudFile reference to untyped binary files.

CloudValue immutable, typed data.

CloudSequence immutable, collection of items, on-demand fetch.

CloudChannel send/receive messages.

CloudAtom mutable data with concurrency control (atomic or non-atomic updates).

CloudDictionary key-value store.

How to handle hardware (worker node) or unrecoverable runtime failure?

How to handle hardware (worker node) or unrecoverable runtime failure?
The high-level strategy is :

- Define your retry strategy and policy.
- Try the operation that could result in a transient fault.
- If transient fault occurs, invoke the retry policy.
- If all retries fail, raise a `FaultException`.

```
type FaultPolicy =
    { Policy : int → exn → TimeSpan option }

Cloud.FaultPolicy: Cloud<FaultPolicy>
Cloud.WithFaultPolicy:
    FaultPolicy → Cloud<'T> → Cloud<'T>
Cloud.StartAsTask:
    (Cloud<'T> * FaultPolicy) → Cloud<ICloudTask<'T>>
```

```
let result =
    runtime.Run(
        [1..4]
        |> List.map (fun i →
            cloud { return i <> 2 || exit 1 })
        |> Cloud.Parallel
    , faultPolicy = FaultPolicy.NoRetry)

MBrace.Core.FaultException:
 Fault exception when running job
    '071b96a23264423890cb6577184c6fa6', faultCount '1'
at Cloud.Parallel(seq<Cloud<Boolean» computations)
```

IMPLEMENTATION

`Cloud<T>` is the basic type describing any cloud computation.

`Cloud<T>` is the basic type describing any cloud computation. By performing a CPS transformation on cloud expressions we can view a `Cloud<T>` as continuation based distributed computation.

```
type Continuation<'T> = {
    SCont : 'T  → unit
    ECont : exn → unit
    CCont : OperationCancelledException → unit
}
```

How to implement a non-trivial operator like `Cloud.Parallel`?

- `Continuation<T>` declares the semantics (cancel on first exception, etc).
- We need a handle to runtime resources in order to enqueue jobs, etc.

`ExecutionContext` contains all types necessary for a computation to interact with the runtime.

```
type ExecutionContext = {
    Token                  : ICloudCancellationToken
    Runtime                : IDistributionProvider
     …
    CloudFileProvider    : CloudFileProvider
    CloudChannelProvider: CloudChannelProvider
     …
    CustomResources      : seq<obj>
}
```

Is `Continuation<T>` enough?

- `Cloud<T>`/`Continuation<T>` are transferred across the cluster (serializable).
- But `ExecutionContext` is has a machine/local scope: worker state, local cache, custom resources, etc.

- Read values and pass state from a shared environment.
- `ExecutionContext` is the machine-local environment.
- The continuations are constructed by supplying the worker's `ExecutionContext`.

```
type Continuation<'T> = {
  SCont : ExecutionContext → 'T  → unit
  ECont : ExecutionContext → exn → unit
  CCont : ExecutionContext → OperationCancelledException → unit
}
```

```
type Continuation<'T> = {
  SCont : ExecutionContext → 'T  → unit
  ECont : ExecutionContext → exn → unit
  CCont : ExecutionContext → OperationCancelledException → unit
}
```

```
Cloud.FromContinuations<'T> :
    (ExecutionContext → Continuation<'T> → unit) → Cloud<'T>
Cloud.GetResource<'T> : unit → 'T
```

A cloud workflow `Cloud<T>` is a function accepting three continuations, `scont`, `econt`, `ccont` and a context `ectx`.

A cloud workflow `Cloud<T>` is a function accepting three continuations, `scont`, `econt`, `ccont` and a context `ectx`.

```
type Cloud<'T> = ExecutionContext → Continuation<'T> → unit
```

A cloud workflow `Cloud<T>` is a function accepting three continuations, `scont`, `econt`, `ccont` and a context `ectx`.

```
type Cloud<'T> = ExecutionContext → Continuation<'T> → unit
```

The `cloud monad` forms a *continuation over reader* monad.

`MBrace.Azure` is based on Microsoft Azure services.

`MBrace.Azure` is based on Microsoft Azure services.

Service Bus is a cloud-based messaging system for connecting distributed applications.

Azure Storage provides the capability to store large amounts of binary data in Azure Blobs, as well as structured NoSQL based records with Azure Tables.

Cloud Services `MBrace.Azure` is hosted in a Worker Role (Platform as a Service).

`mbrace.azure` FEATURES

- Complete implementation of `MBrace.Core`.
- Hosted as a Azure worker role / standalone executable for local testing.
- Worker monitoring : cpu, memory, network utilization, etc.
- Execute computations as 'processes': statistics about execution time, current status, number of jobs spawned, etc.
- Job tracking : execution times, number of retries, serialized job size, current status, assigned worker, etc.
- Logging.

`mbrace.azure` FEATURES

- Fault tolerance, built on top of Service Bus Lease/Lock mechanism.
- All storage primitives:
  - *CloudFiles* are implemented as Azure Blobs.
  - *CloudAtoms*, *CloudDictionaries* are implemented on top of the Azure Table storage.
  - *CloudChannels* are implemented using Service Bus queues.
- Cloud scripting from Visual Studio; automatic assembly upload.

CONCLUSION

- `MBrace.Core` offers a simple, but powerfull model for describing distributed cloud computations.
- `MBrace.Azure` offers a complete cloud scripting experience.

- Keep up with `MBrace.Core`, stabilize API, `MBrace.Core` *1.0*.

GOALS

- Keep up with `MBrace.Core`, stabilize API, `MBrace.Core` *1.0*.
- Provide a common foundation for implementing `MBrace` runtimes (`MBrace.Runtime.Core`).

- Keep up with `MBrace.Core`, stabilize API, `MBrace.Core` *1.0*.
- Provide a common foundation for implementing `MBrace` runtimes (`MBrace.Runtime.Core`).
- Make it easier for someone to contribute.

GOALS

- Keep up with `MBrace.Core`, stabilize API, `MBrace.Core` *1.0*.
- Provide a common foundation for implementing `MBrace` runtimes (`MBrace.Runtime.Core`).
- Make it easier for someone to contribute.
- I hope that `MBrace` becomes the ‘go-to’ framework for cloud computing in .NET.

thanks! :-)

- http://www.m-brace.net/
- http://github.com/mbraceproject/MBrace.Azure
- krontogiannis / MBrace.Azure @ e688c87