Exercise Session 10

Towards Home Exam 2

Oleks Shturmov

<olekss@uio.no> / <oleks@oleks.info>

University of Oslo

IN[59]570: Distributed Objects

April 29, 2019 (Revised on May 6, 2019)

The source code for these slides is maintained here:

https://github.com/emerald/in5570v19/tree/master/exercise-sessions/10

Agenda

1. Parametric Polymorphism (in Emerald)
2. The `PCRType` from Home Exam 2
3. Emulating Unavailability (using Docker)

Polymorphism in Emerald

In a language with **polymorphism**,
we can write code that works for many datatypes,
not just one particular type of data

- Emerald supports **inclusion polymorphism**, due to its *conformance relation*: In place of a particular type, Emerald hapily accepts a value of a different, but conforming type
  - You know about conformance from previous weeks
- Emerald also supports **parametric polymorphism**, where types depend on the actual parameters (e.g., the parameters to a method call)
  - See also Section 7.4 on pages 18-19 of [Raj et al., 1991]

Passing the Type Parameter Implicitly

Showing the contents of implicit.m

```
const main <- object main
  op showType[a : t]                      % Where does t come from?
    forall t
    stdout.putstring[t$name || "\n"]
  end showType
  initially
    self.showType[5]                      % From here!
  end initially
end main
```

```
$ emx implicit.x
integertype
```

- The **forall** clause introduces an *unconstrained type variable*
- We can then (somewhat backwards) use `t` in the signature preceeding the **forall** clause (**op** `showType[a : t]`)
- We must use a **forall** clause, as otherwsie `t` is undefined ☹
- `t` gets the type `ConcreteType`; `t` can be inspected at runtime

There Are No Runtime Costs

All types are determined at compile-time!

- Watch out for “type must be manifest” errors from the Emerald compiler; if you get these, it means that the type of some expression cannot be determined at compile-time

Passing the Type Parameter Explicitly

Showing the contents of explicit.m

```
const main <- object main
  op showType[t : Type, a : t]      % t is an explicit parameter
    stdout.putstring[(typeof a)$name || "\n"]
  end showType
  initially
    self.showType[Integer, 5]       % but still comes from here
  end initially
end main
```

```
$ emx explicit.x
integertype
```

- Recall that, in Emerald, types are also objects
- As another example, recall how you must explicitly pass a type to the **Array**`.of` constructor, to get an **Array** of that type
- Unfortunately, we can't do much with `t` directly (see line 3)
- Values of type `Type` are assumed to only be available at runtime

A Brief Interlude: `typeof` vs. `syntactictypeof`

Quite unlike many popular languages, Emerald provides two ways to ask for the type of an expression — **`typeof`** and **`syntactictypeof`**:

- **`typeof`** gives the actual type at *runtime*
- **`syntactictypeof`** gives the type determined at *compile time*

The Emerald system guarantees that the runtime type of an expression will conform to its compile-time type.

typeof vs. **syntactictypeof** Illustrated

What happens if we ask for **typeof** `t` instead of **typeof** `a` above?

```
$ diff explicit.m typeof.m
3c3
<     stdout.putstring[(typeof a)$name || "\n"]
---
>     stdout.putstring[(typeof t)$name || "\n"]
$ emx typeof.x
pat
```

What about **syntactictypeof** `t`?

```
$ diff explicit.m syntactictypeof.m
3c3
<     stdout.putstring[(typeof a)$name || "\n"]
---
>     stdout.putstring[(syntactictypeof t)$name || "\n"]
$ emx syntactictypeof.x
typetype
```

Constraining Type Variables Such That . . .

Showing lines 1–8 of `replicate.m`

```
const RType <- typeobject RType
  operation replicate[X : t, N : Integer]
    forall t
    suchthat
      t *> typeobject ot
        op clone -> [result : t]
      end ot
end RType
```

- Use a **`suchthat`** clause
- `*>` means *conforms to*, and the expression on the right-hand side can be any type-valued expression

Building Values with Types Such That . . .

Showing lines 10–21 of `replicate.m`

```
const Replicator : RType <- object Replicator
  export operation replicate[X : t, N : Integer]
    forall t
    suchthat
      t *> typeobject ot
        op clone -> [result : t]
    end ot
    for i : Integer <- 0 while i < N by i <- i + 1
      const c <- X.clone[]
    end for
  end replicate
end Replicator
```

- `t` has to be available at compile-time
- Unfortunately, the only way to do so is with a **`forall`** clause
- Parametric polymorphism can be quite verbose in Emerald ☹

Relicating Integers and Strings

Showing lines 23–40 of replicate.m

```
const RInt <- class RInt[value : Integer]
  export operation clone -> [result : RIntType]
    stdout.putstring["Cloning " || value.asstring || "..\n"]
    result <- RInt.create[value]
  end clone
end RInt
const RString <- class RString[value : String]
  export operation clone -> [result : RStringType]
    stdout.putstring["Cloning " || value || "..\n"]
    result <- RString.create[value]
  end clone
end RString
const main <- object main
  initially
    Replicator.replicate[RInt.create[5], 3]
    Replicator.replicate[RString.create["Hello"], 5]
  end initially
end main
```

Constructing Dependent Types

Showing the contents of `replicas.m`

```
const RaType <- typeobject RaType
  operation replicas[X : t] -> [Array.of[rt]]
  forall t
  where
    rt <- typeobject rt
      operation read -> [o : t]
      operation write[o : t]
    end rt
end RaType
```

- Use a **`where`** clause
- The type `rt` *depends on* the given type `t`
- Constructing a value of this particular type however, is even more tricky than for RType; that is, without resorting to type assertions (**`view`** ... **`as`** ...)

The **PCRType** in Home Exam 2

Showing the contents of `typedefs.m`

```
const PCRType <- typeobject PCRType
  operation replicate[X : t, N : Integer]
  forall t
  suchthat
    t *> typeobject ot
      op clone -> [result : t]
    end ot
  operation replicas[X : t] -> [Array.of[rt]]
  forall t
  where
    rt <- typeobject rt
      operation read -> [o : t]
      operation write[o : t]
    end rt
end PCRType
```

Emulating Unavailability (using Docker)

- Docker containers connect to the web via a network "bridge"
- You can connect and disconnect containers from such a bridge
- If a container is not connected to a network bridge,
  for all intents and purposes, it is offline
- This way, we can simulate temporary node unavailability

Creating A (New) Network Bridge

Although a Docker container is by default connected to a default network bridge, you can exert grander control by creating your own network bridge

- To create a network bridge:

```
$ docker network create \
    --subnet=172.18.0.0/24 \
    --ip-range=172.18.0.0/24 \
    --driver=bridge \
    unavail
```

- The subnet and IP range arguments effectively make the following IP address available for containers to use:
  - 172.18.0.2
  - 172.18.0.3
  - ...
  - 172.18.0.254
- This bridge is named `unavail` (see last argument)

Connecting Running Containers to the Bridge

- Start a Docker container
  - Let it have the container ID 85a87446465
- To connect 85a87446465 to `unavail` at address 172.18.0.2:

```
$ docker network connect --ip=172.18.0.2 unavail 85a87446465
```

- To disconnect 85a87446465 from `unavail`:

```
$ docker network disconnect unavail 85a87446465
```

In a similar vein, you can connect up a range of containers, and methodically take them offline one-by-one.

- See attached `monitor.m` for a sample program that monitors the list of available nodes

More Network Operations

As you experiment with Docker and bridge networks, you might find the following useful:

- To inspect the state `unavail` (e.g., see list of connected containers):

```
$ docker network inspect unavail
```

- To remove `unavail`

```
$ docker network rm unavail
```

Further Reading

Raj, Tempero, Levy, Black, Hutchinson, and Jul (1991),
Technical Report: The Emerald Programming Language

https://www.uio.no/studier/emner/matnat/ifi/INF5510/v15/pensum/Report.pdf