User-Defined Control Structures in X10

1 User-Defined Control Structures

KeywordOpDecln ::=
  MethMods `operator` *keywordOp* *TypeParams*$^?$ *Formals* *Guard*$^?$ *Throws*$^?$
  HasResultType$^?$ *MethodBody*
keywordOp ::= `for`|`if`|`try`|`throw`|`async`|`atomic`|`when`|`finish`|`at`
  |`continue`|`break`|`ateach`|`while`|`do`

Similarly to user-defined operators, it is possible to redefine the behavior of some control structures. For example, suppose that we want to define a `if` statement that randomly chooses which branch to execute. In a class `RandomIf`, we define a method named `if` (introduced with the keyword `operator`) that implements this behavior:

```
class RandomIf {
    val random = new Random();
    public operator if(then: ()=>void, else_: ()=>void) {
        if (random.nextBoolean()) {
            then();
        } else {
            else_();
        }
    }
}
```

Then, we can call this method using the syntax of the `if` statement by prefixing the `if` keyword by an object that implements this method:

```
    val random = new RandomIf();
    random.if () {
        Console.OUT.println("true");
    } else {
        Console.OUT.println("false");
    }
```

The blocks that represent the `then` and the `else` branches of the `if` are automatically turned into closures and are given as argument to the `RandomIf.if` method.

To distinguish the use of a user-defined control structure from the use of a built-in one, the first keyword of the control structure must be prefixed with the object that redefines its behavior. The scoping and dispatching rules of user-defined control structures is exactly the same as the one of methods.

User-defined control structures can also be called as standard methods using the `operator` prefix (as for user-defined operators). For example, the previous code is equivalent to:

```
val random = new RandomIf();
random.operator if (() => { Console.OUT.println("true"); },
                    () => { Console.OUT.println("false"); });
```

1.1 User-Defined `for`

$$\begin{array}{l} o\texttt{.for[}\overline{T}\texttt{]}^{?}\texttt{((}\overline{x\texttt{:}t}\texttt{ in)}^{?}\ \overline{e}\texttt{)}\ b \quad \triangleq \\ \qquad o\texttt{.operator for[}\overline{T}\texttt{]}^{?}\texttt{(}\overline{e}\texttt{, (}\overline{x\texttt{:}t}^{?}\texttt{) => }b\texttt{);} \end{array}$$

A `for` loop over a collection may be defined in a container `A` as:

```
operator for[T](c: Iterable[T], body: (T)=>void) = ...
```

The use of such user-defined `for` loop would have the following form:

```
A.for (x: T in c) { ... }
```

and would correspond to the following method call:

```
A.operator for (c, (x: Long) => { ... });
```

The body of the `for` is automatically translated into a closure that takes the iteration variable as parameter. Since there is no type inference for closure parameters, the type of the iteration variable must be given explicitly.

The second argument of a `for` method can be a closure without argument:

```
operator for[T](c: Iterable[T], body: ()=>void) = ...
```

In this case, the method is called using the syntax of a `for` loop without iteration variable:

```
A.for (c) { ... }
```

Example 1 *A naive implementation of a parallel loop can be:*

```
class Parallel {

  public static operator for[T](c: Iterable[T], body: (T)=>void) {
    finish {
      for(x in c) {
        async { body(x); }
      }
    }
  }

  public static def main(Rail[String]) {
    val cpt = new Cell[Long](0);
    Parallel.for(i:Long in 1..10) {
      atomic { cpt() = cpt() + i; }
    }
    Console.OUT.println(cpt());
  }
}
```

Example 2 *We can also use the user-defined* `for` *loops to define iterations over a two dimensional space. Let define a loop that creates an activity for each element of the first dimension.*

```
class Parallel2 {
  public static operator for (space: DenseIterationSpace_2,
                              body: (i:Long, j:Long)=>void) {
    finish {
      for (i in space.min0 .. space.max0) {
        async for (j in space.min1 .. space.max1) {
          body(i, j);
        }
      }
    }
  }
}
```

and it can be used as follows:

```
Parallel2.for (i:Long, j:Long in 1..10 * 1..10) { ... }
```

The list of variables before the `in` *keyword becomes the parameters of the closure whose body is the body of the loop.*

1.2 User-Defined `if`

$$o.\texttt{if}[\overline{T}]^?(\overline{e})\ b_1\ (\texttt{else}\ b_2)^? \triangleq$$
$$o.\texttt{operator if}[\overline{T}]^?(\overline{e}\texttt{, ()=> } b_1(\texttt{, ()=> } b_2)^?\texttt{);}$$

When we use a user-defined `if` statement, the condition is evaluated before calling the `if` method, but the then and else branches are implicitly lifted to closures without argument.

Note that the condition of a user-defined `if` statement can take an arbitrary number of arguments. This is why we were able to define the `Random.if` that does not take a condition.

1.3 User-Defined `try`

$$o.\texttt{try}[\overline{T}]^?(\overline{e})^?\ b_1\ \texttt{catch}\ (\overline{x\texttt{:}t})\ b_2\ (\texttt{finally}\ b_3)^? \triangleq$$
$$o.\texttt{operator try}[\overline{T}]^?((\overline{e}\texttt{,})^?\ \texttt{()=>}\ b_1\texttt{,}\ (\overline{x\texttt{:}t})\ \texttt{=>}\ b_2(\texttt{,}\ \texttt{()=>}\ b_3)^?)\texttt{;}$$

When we use a user-defined `try` statement, the body of the `try` is lifted to a closure without argument and handler is lifted to a closure that has the parameter of the `catch` as parameter. The `finally` block is also lifted to a closure without argument.

Example 3 *The user-defined* `try` *construct can be used to provide a control structure that automatically removes the nesting of* `MultipleExceptions`*:*

```
class Flatten {

  public static operator try(body:()=>void,
                             handler:(MultipleExceptions)=>void) {
    try { body(); }
    catch (me: MultipleExceptions) {
      val exns = new GrowableRail[CheckedThrowable]();
      flatten(me, exns);
      handler (new MultipleExceptions(exns));
    }
  }

  private static def flatten(me:MultipleExceptions,
                             acc:GrowableRail[CheckedThrowable]) {
    for (e in me.exceptions) {
      if (e instanceof MultipleExceptions) {
        flatten(e as MultipleExceptions, acc);
      } else {
        acc.add(e);
      }
    }
  }
}
```

Used in the following example, the `MultipleExceptions me` *contains the exceptions* `Exception("Exn 1")`*,* `Exception("Exn 2")`*, and* `Exception("Exn 3")` *instead of the exception* `Exception("Exn 1")` *and another* `MultipleExceptions`*.*

```
public static def main(Rail[String]) {
  Flatten.try {
    finish {
      async { throw new Exception("Exn 1"); }
      async finish {
        async { throw new Exception("Exn 2"); }
        async { throw new Exception("Exn 3"); }
      }
    }
  } catch (me: MultipleExceptions) {
    Console.OUT.println(me.exceptions);
  }
}
```

1.4 User-Defined `throw`

$$o\texttt{.throw[}\overline{T}\texttt{]}^{?}\ e^{?}\texttt{;}\ \triangleq\ o\texttt{.operator throw[}\overline{T}\texttt{]}^{?}\texttt{(}e^{?}\texttt{);}$$

The argument of a user-defined `throw` is evaluated before calling the `throw` method.

1.5 User-Defined `async`

$$o\texttt{.async[}\overline{T}\texttt{]}^{?}\texttt{(}\overline{e_1}\texttt{)}^{?}\ \texttt{(clocked (}\overline{e_2}\texttt{))}^{?}\ b\ \triangleq\ o\texttt{.operator async[}\overline{T}\texttt{]}^{?}\texttt{((}\overline{e_1}\texttt{,)}^{?}\ \texttt{(}\overline{e_2}\texttt{,)}^{?}\ \texttt{()=>}\ b\texttt{);}$$

The body of a user-defined `async` is lifted to a closure without argument. The clock arguments are evaluated before the call to the `async` method.

Example 4 *An* `async` *that does not execute in the scope in which it is written. The task is created in the scope where the object that defines the* `async` *method is instantiated.*

```
class Escape {
  private var task: ()=>void = null;
  private var stop: Boolean = false;

  public def this() {
    async {
      while (!stop) {
        val t: () => void;
        when (task != null || stop) {
          t = task;
          task = null;
        }
        if (t != null) {
          async { t(); }
        }
      }
    }
  }

  public operator async (body: () => void) {
    when (task == null) {
      task = body;
    }
  }

  public def stop() {
    atomic { stop = true; }
  }
}
```

In the following example, the message `"OK"` *is printed even if the created task never terminates because the task is executed outside of the scope of the* `finish`*.*

```
public static def main(Rail[String]) {
  val toplevel = new Escape();
  finish {
    toplevel.async { when (false){} }
  }
  Console.OUT.println("OK");
}
```

1.6 User-Defined `atomic`

$$o\texttt{.atomic[}\overline{T}\texttt{]}^{?}\texttt{(}\overline{e}\texttt{)}^{?}\ b \quad \triangleq \quad o\texttt{.operator atomic[}\overline{T}\texttt{]}^{?}\texttt{((}\overline{e}\texttt{,)}^{?}\ \texttt{()=>}\ b\texttt{);}$$

The body of a user-defined atomic statement is lifted to a closure without argument.

1.7 User-Defined `when`

$$o\texttt{.when[}\overline{T}\texttt{]}^{?}\texttt{(}\overline{e}\texttt{)}\ b \quad \triangleq \quad o\texttt{.operator when[}\overline{T}\texttt{]}^{?}\texttt{(}\overline{e}\texttt{, ()=>}\ b\texttt{);}$$

The arguments of a user-defined `when` statements are evaluated before the call of the `when` method and the body is lifted to a closure without argument. It means that if the argument of a user-defined `when` is of type `Boolean`, the condition is evaluated once and cannot be changed. To be able to update the condition, it can be an object with mutable field as in the following example or a closure.

Example 5 *We can provide a* `when` *statement whose execution can be canceled while it is waiting:*

```
class CancelableWhen {
  private var stop : Boolean = false;

  public operator when(condition:Cell[Boolean], body:()=>void) {
    when (condition() || stop) {
      if (!stop) { body(); }
    }
  }
  public def cancel() {
    atomic { stop = true; }
  }
}
```

The following example will not print the message "KO" *but will terminate even if the condition* b *of the* when *remains false:*

```
public static def main(Rail[String]) {
  val c = new CancelableWhen();
  val b = new Cell[Boolean](false);
  finish {
    async {
      c.when(b) { Console.OUT.println("KO"); }
    }
    c.cancel();
  }
}
```

1.8 User-Defined finish

$$o\texttt{.finish[}\overline{T}\texttt{]}^{?}\texttt{(}\overline{e}\texttt{)}^{?}\ b \quad\triangleq\quad o\texttt{.operator finish[}\overline{T}\texttt{]}^{?}\texttt{((}\overline{e}\texttt{,)}^{?}\ \texttt{()=>}\ b\texttt{);}$$

The body of a user-defined finish is lifted to a closure.

Example 6 *We define a* finish *that provide the ability to some parallel task to wait for its termination:*

```
class SignalingFinish {
  private var terminated : Boolean = false;
  public operator finish(body: ()=>void) {
    finish {
      body();
    }
    atomic { terminated = true; }
  }
  public def join() {
    when (terminated) {}
  }
}
```

The following example will always print the message "before" *before the message* "after".

```
public static def main(Rail[String]) {
  val t = new SignalingFinish();
  async {
    t.join();
    Console.OUT.println("after");
  }
  t.finish {
    Console.OUT.println("before");
  }
}
```

1.9 User-Defined at

$$o\texttt{.at}[\overline{T}]^{?}(\overline{e})\ b \ \triangleq\ o\texttt{.operator at}[\overline{T}]^{?}(\overline{e}\texttt{, ()=>}\ b\texttt{);}$$

The arguments of the user-defined `at` statement are evaluated before the call of the `at` method and the body of the statement is lifted to a closure without argument.

Example 7 *We define a class* `Ring` *implementing an* `at` *statement without argument. Each call to this user-defined* `at` *statement moves the activity to the next place in the place group given when the object is instantiated.*

```
class Ring {
  val places: PlaceGroup;

  public def this (places: PlaceGroup) {
    this.places = places;
  }

  public operator at(body: ()=>void) {
    at(places.next(here)) { body(); }
  }
}

public static def main(Rail[String]) {
  val r = new Ring(Place.places());
  r.at() {
    Console.OUT.println("Hello from "+here+"!");
    r.at() {
      Console.OUT.println("Hello from "+here+"!");
    }
  }
}
```

1.10 User-Defined ateach

$$o\texttt{.ateach}[\overline{T}]^{?}((\overline{x\texttt{:}t}\ \texttt{in})^{?}\ \overline{e})\ b \ \triangleq$$
$$o\texttt{.operator ateach}[\overline{T}]^{?}(\overline{e}\texttt{, (}\overline{x\texttt{:}t}^{?}\texttt{) =>}\ b\texttt{);}$$

The arguments of the user-defined `ateach` statement are evaluated before the call of the `ateach` method and the body of the statement is lifted to a closure without argument.

Example 8 *An* `ateach` *control structure that has the same behavior as the built-in* `ateach`*, except that the activities are executed in sequence instead of being executed in parallel.*

```
class Sequential {
  public static operator ateach (d: Dist, body:(Point)=>void) {
    for (place in d.places()) {
      at(place) {
        for (p in d|here) { body(p); }
      }
    }
  }
}
```

1.11 User-Defined `while` and `do`

$$o.\texttt{while}[\overline{T}]^{?}(\overline{e})\ b \triangleq o.\texttt{operator while}[\overline{T}]^{?}(\overline{e}\texttt{, ()=> }b\texttt{);}$$
$$o.\texttt{do}[\overline{T}]^{?}\ b\ \texttt{while (}\overline{e}\texttt{);} \triangleq o.\texttt{operator do}[\overline{T}]^{?}\texttt{(()=> }b\texttt{, }\overline{e}\texttt{);}$$

The arguments of the user-defined `while` (resp. `do`) are evaluated before the call of the `while` (resp. `do`) method and the body of the loop is lifted to a closure without argument. Note that compared to usual loop, the condition is evaluated once before the call of the method that implements the behavior of the loop.

Example 9 *A loop that iterates during at least a given number of milliseconds:*

```
class Timeout {
  public static operator while(ms: Long, body: ()=>void) {
    val deadline = System.currentTimeMillis() + ms;
    while (System.currentTimeMillis() < deadline) {
      body();
    }
  }
}
```

Here, we increment a counter during about 10 milliseconds:

```
public static def main(Rail[String]) {
  val cpt = new Cell[Long](0);
  Timeout.while(10) {
    atomic { cpt() = cpt() + 1; }
  }
  Console.OUT.println(cpt());
}
```

1.12 User-Defined `continue`

$$o\texttt{.continue[}\overline{T}\texttt{]}^{?}\ e^{?}\texttt{;} \quad \triangleq \quad o\texttt{.operator continue[}\overline{T}\texttt{]}^{?}\texttt{(}e^{?}\texttt{);}$$

The argument of a user-defined `continue` is evaluated before calling the corresponding method.

Example 10 *The following code provides a parallel* `for` *loop and a* `continue` *statement that allows to skip an iteration.*

```
class Par {
  private static class Continue extends Exception {}

  public static operator continue () {
    throw new Continue();
  }

  public static operator for[T](c: Iterable[T], body:(T)=>void) {
    finish {
      for(x in c) async {
          try {
            body(x);
          } catch (Continue) {}
        }
    }
  }
}
```

The following example skip every iteration where the loop index is even.

```
public static def main(Rail[String]) {
  val cpt = new Cell[Long](0);
  Par.for(i:Long in 1..10) {
    if (i%2 == 0) { Par.continue; }
    atomic { cpt() = cpt() + 1; }
  }
  Console.OUT.println(cpt());
}
```

1.13 User-Defined `break`

$$o\texttt{.break[}\overline{T}\texttt{]}^{?}\ e^{?}\texttt{;} \quad \triangleq \quad o\texttt{.operator break[}\overline{T}\texttt{]}^{?}\texttt{(}e^{?}\texttt{);}$$

The argument of a user-defined `break` is evaluated before calling the corresponding method.

Example 11 *To break out of a user-defined loop, it is necessary to also define the* `break` *statement:*

```
class Infinite {
  private static class Break extends Exception {}

  public static operator break () {
    throw new Break();
  }

  public static operator while (body:()=>void) {
    try {
      while(true) {
        body();
      }
    } catch (Break) {}
  }

  public static def main(Rail[String]) {
    Infinite.while() {
      Infinite.break;
    }
    Console.OUT.println("OK");
  }
}
```