Computation expressions for Scala 3

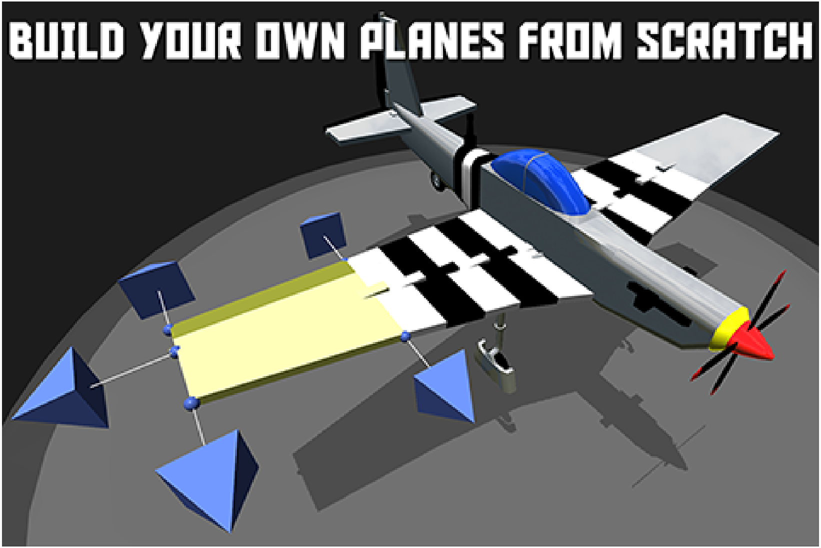
From AST builders to computation expressions

Noé De Santo

17 June 2022

Motivating example

Simple planes...



…are not simple to program

```
<Variables>
  <Setter variable="while_condition_0" function="(Altitude &lt;= 1000.0)"
      activator="(!evaluationFlag_while_condition_0_0)"/>
  <Setter variable="evaluationFlag_while_condition_0_0" function="true" />
  <Setter variable="Pitch" function="20.0" activator="((!evaluationFlag_Pitch_0) &amp;
      while_condition_0)"/>
  <Setter variable="evaluationFlag_Pitch_0" function="true" activator="while_condition_0"/>
  <Setter variable="evaluationFlag_Pitch_0" function="true" activator="(!while_condition_0)"/>
  <Setter variable="memoised_whileBodyEvaluated_0" function="evaluationFlag_Pitch_0" />
  <Setter variable="evaluationFlag_while_condition_0_0" function="false"
      activator="(memoised_whileBodyEvaluated_0 &amp; while_condition_0)"/>
  <Setter variable="evaluationFlag_Pitch_0" function="false"
      activator="(memoised_whileBodyEvaluated_0 &amp; while_condition_0)"/>
</Variables>
```

...are not simple to program

```
<Variables>
  <Setter variable="while_condition_0" function="(Altitude &lt;= 1000.0)"
      activator="(!evaluationFlag_while_condition_0_0)"/>
  <Setter variable="evaluationFlag_while_condition_0_0" function="true" />
  <Setter variable="Pitch" function="20.0" activator="((!evaluationFlag_Pitch_0) &amp;
      while_condition_0)"/>
  <Setter variable="evaluationFlag_Pitch_0" function="true" activator="while_condition_0"/>
  <Setter variable="evaluationFlag_Pitch_0" function="true" activator="(!while_condition_0)"/>
  <Setter variable="memoised_whileBodyEvaluated_0" function="evaluationFlag_Pitch_0" />
  <Setter variable="evaluationFlag_while_condition_0_0" function="false"
      activator="(memoised_whileBodyEvaluated_0 &amp; while_condition_0)"/>
  <Setter variable="evaluationFlag_Pitch_0" function="false"
      activator="(memoised_whileBodyEvaluated_0 &amp; while_condition_0)"/>
</Variables>
```

```
while Altitude <= 1000 do
  Pitch = 20
```

What are we doing?

Goals:

- Design and implement a library allowing to “program” ASTs inside of Scala;
- Generalize it as much as possible.

What are we doing?

Goals:

- Design and implement a library allowing to “program” ASTs inside of Scala;
- Generalize it as much as possible.

Potential use cases:

- “Little languages”;
- Configuration generation (yaml, kubernetes, …);
- Programmatic drawing (dot, miro, tikz, …).

Foreword:

Today’s toy example

Meet Rob

We will work with an imaginary programmable robot, Rob.

- Lives on a 2D grid;
- Can move forward, rotate on itself;
- Know things about itself (position, orientation);
- Typical features (variables, if-then-else, statement sequencing, …).

AST represented using type `Tree[_]`.

Meet Rob

We will work with an imaginary programmable robot, Rob.

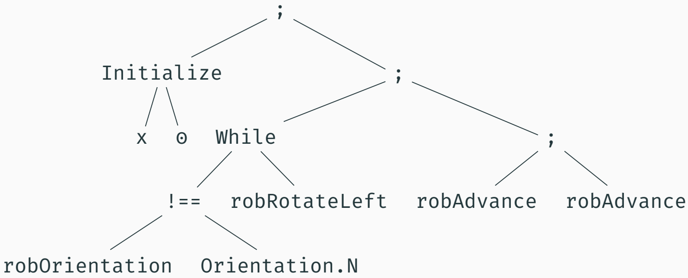
- Lives on a 2D grid;
- Can move forward, rotate on itself;
- Know things about itself (position, orientation);
- Typical features (variables, if-then-else, statement sequencing, …).

AST represented using type `Tree[_]`.

To make it easier to use, we will also assume a
`Conversion[Int, Tree[Int]]`

A Rob example

```
val code = rob{
  val x = 0
  while robOrientation !== Orientation.N do
    robRotateLeft
  robAdvance
  robAdvance
}
```



Implementation

(And why not all of this is a good idea)

Key idea:

- We can leverage Scala's metaprogramming to replace some constructs;

```
robRotateLeft
robRotateLeft
```

$\longmapsto$

```
Sequence(
  robRotateLeft,
  robRotateLeft
)
```

Metaprogramming

Key idea:

- We can leverage Scala's metaprogramming to replace some constructs;

```
robRotateLeft;
robRotateLeft
```

$\longmapsto$

```
Sequence(
  robRotateLeft,
  robRotateLeft
)
```

- Here, we are basically overloading the (implicit) ; operator;
- The same can be done for other constructs (e.g. assignation =).

Variables translation

This requires to correctly dissociate between the Scala-level variable, and the user-language-level one:

```scala
val x = 0
x + 2
```

$\longmapsto$

```scala
val x = Variable("x")
Assign(x, 0);
x + 2
```

Variables translation

This requires to correctly dissociate between the Scala-level variable, and the user-language-level one:

```scala
val x = 0
x + 2
```

$\longmapsto$

```scala
val x = Variable("x")
Assign(x, 0);
x + 2
```

Which, after further transformation and Scala execution, yields:

$\equiv$

```scala
Sequence(
  Assign(Variable("x"), 0),
  Add(Variable("x"), 2)
)
```

Typing issues

We might have some typing issues:

```
var x = 0
if x === 1 then
  x = robX
else
  x = 1
```

Typing issues

We ~~might~~ have ~~some~~ **many** typing issues:

```
var x: Variable[Int] = (0: Int)
if (x === 1): Tree[Boolean] then
  (x: Variable[Int]) = (robX: Tree[Int])
else
  (x = 1): Unit
```

(Note that, after transformation, the code is well-typed, and has type `Tree[Unit]`)

Typing issues

We ~~might~~ have ~~some~~ **many** typing issues:

```
var x: Variable[Int] = (0: Int)
if (x === 1): Tree[Boolean] then
  (x: Variable[Int]) = (robX: Tree[Int])
else
  (x = 1): Unit
```

(Note that, after transformation, the code is well-typed, and has type `Tree[Unit]`) Solutions:

1. Do it differently;
2. Trick the type system into accepting this.

hippity hoppity accept my code

We can (ab-)use implicit conversions. We would need:

$$
\begin{aligned}
\texttt{Tree[T]} &\longmapsto \texttt{Variable[T]} \\
\texttt{Tree[Boolean]} &\longmapsto \texttt{Boolean} \\
\texttt{Unit} &\longmapsto \texttt{Tree[Unit]}
\end{aligned}
$$

hippity hoppity accept my code

We can (ab-)use implicit conversions. We would need:

$$
\begin{aligned}
\texttt{Tree[T]} &\longmapsto \texttt{Variable[T]} \\
\texttt{Tree[Boolean]} &\longmapsto \texttt{Boolean} \\
\texttt{Unit} &\longmapsto \texttt{Tree[Unit]}
\end{aligned}
$$

Issues:

- Can be inserted in unexpected (and unwanted) places;
- Highly contextual;
- Technically, do not convert anything: they will get erased.

Think different

1. Use DSL elements to replace some constructs:
    - Step away from Scala notations;
    - No typing-trickery.
2. Use an operator to signify user-level variables.

Think different

1. Use DSL elements to replace some constructs:
   - Step away from Scala notations;
   - No typing-trickery.
2. Use an operator to signify user-level variables.

`val x = value`	$\longmapsto$	`val x = ! (value)`
`if cond then thenn else elze`	$\longmapsto$	`If(cond){ thenn }{ elze }`
`variable = value`	$\longmapsto$	`variable =! value`

Think different

1. Use DSL elements to replace some constructs:
    - Step away from Scala notations;
    - No typing-trickery.
2. Use an operator to signify user-level variables.

```
var x = ! 0
If(x === 1){
  x =! robX
}{
  x =! 1
}
```

Generalization:

toward computation expressions

More than Rob

The previous examples were tied to our robot. Solution:

- Let the user define a `myBuilder`;

```
trait ASTBuilder[Tree[_]] {
  def sequence[S, T](l: Tree[S], r: Tree[T]): Tree[T]
  // Other methods for other constructs
  ...
}

object myBuilder[MyTree] extends ASTBuilder {...}
```

More than Rob

The previous examples were tied to our robot. Solution:

- Let the user define a `myBuilder`;
- Call the methods of the builder.

`l ; r`	$\longmapsto$	`myBuilder.sequence(l, r)`

Computation expresions: Qu’es aquò?

To keep it short and simple:

- Feature of F# (functional language; .NET based);
- Some keywords “do nothing” (e.g. `let!`);
- A function can be implemented to give meaning to them.

Call it maybe

```
type MaybeBuilder() =
  // Gives meaning to let!
  member x.Bind(opt, f) = Option.bind f opt
  // Gives meaning to return
  member x.Return(v) = Some(v)

let maybe = new MaybeBuilder()

let m = maybe{
  let! v = Some(0)
  return v + 1
}
// m == Some(1)
```

The anatomy of `Bind`

`Bind`/**`let`**`!` takes two arguments: the value being bound, and the "follow".

$\longmapsto$

```
let! v = Some(0)
return v + 1
```

```
maybe.Bind(
    Some(0),
    fun (v: int) ->
        return v + 1
)
```

The anatomy of `Bind`

`Bind`/`let!` takes two arguments: the value being bound, and the "follow".

$\longmapsto$

```
let! v = Some(0)
return v + 1
```

```
maybe.Bind(
    Some(0),
    fun (v: int) ->
        return v + 1
)
```

Reminds me a lot of:

Encoding let

$$\text{let } x = t_1;\ t_2 \qquad \equiv \qquad (\lambda x.\, t_2)(t_1)$$

AST builders as computation expressions

In the context of AST builders, we can use this to get back our original transformation.

```
val x = ! 0
x + 2
```

$\longmapsto$

```
val x = Variable("x")
Sequence(
  Assign(x, 0),
  x + 2
)
```

AST builders as computation expressions

In the context of AST builders, we can use this to get back our original transformation.

```
val x = ! 0
x + 2
```

$\longmapsto$

```
val x = Variable("x")
Sequence(
  Assign(x, 0),
  x + 2
)
```

$\equiv$

```
val variable = Variable("x")
Sequence(
  Assign(variable, 0)
  (x => x + 2)(variable)
)
```

AST builders as computation expressions

In the context of AST builders, we can use this to get back our original transformation.

```
def bind[T, S](
  value: Tree[T],
  follow: Variable[T] => Tree[S]
): Tree[S] =
  val variable = freshVariable
  sequence(
    assign(variable, value),
    follow(variable)
  )
```

Wrapping up

What exactly was done?

- Some (restricted) form of computation expressions were implemented;
- AST builders were defined as some particular kind of computation expressions;
- DSL was privileged over implicit conversions.

Does this generalize?

- Computation expressions are (by design) general, yet restricted in our case;
- AST builders seem general enough in our examples.

3-sort

```
While(True){
    var fst: Variable[Int] = ! inbox
    var snd: Variable[Int] = ! inbox
    var trd: Variable[Int] = ! inbox

    If(snd < fst){
      val tmp = ! fst
      fst =! snd
      snd =! tmp
    }
    If(trd < snd){
      val tmp = ! trd
      trd =! snd
      snd =! tmp
      If(snd < fst){
        val tmp = ! fst
        fst =! snd
        snd =! tmp
      }
    }
    outbox =! fst
    outbox =! snd
    outbox =! trd
  }
}
```

Krakabloa race

```
val flightpath: List[(Double, Double)] = List(
  (57514, 18117),
  (55786, 16899),
  (53461, 16991),
  (52678, 17726),
  (52442, 21872),
  (53521, 22993),
  (54681, 21772),
  (55473, 23612),
  (55989, 22328),
  (56800, 21871),
  (56581, 19000),
)

thrust =! 1
setLevelPitch | waitFor(5)
for case (lat, lon) <- flightpath yield
  goToCoordinates(lat, lon)
While(Altitude < 1400){
  elevators =! PID(30, PitchAngle, 0.1, 0, 0.05)
}
setLevelPitch
```

The End (?)

References

- The result: `https://github.com/Ef55/scala-expression-processor`
- Computation expressions: `https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/computation-expressions`
- CE usage: `https://link.springer.com/chapter/10.1007/978-3-319-04132-2_3`

What do you mean “restricted”?

- F#’s CE do not enforce one particular (type-)signature;
- We fixed one for practical purposes.

```
(Computation[T], B[T] => Computation[S]) => Computation[S]
```

where `B` is a type function.

Why this weird signature in particular?

```
object AstBuilder extends ComputationBuilder {
  type W[T] = Expr[T]
  type B[T] = Variable[T]

  def bind[S,T](bound: Expr[T], f: Variable[T] => Expr[S]) = {
    val v = new Variable()
    sequence(
      initialize(v, bound)
      f(v)
    )
  }
}

object maybe extends ComputationBuilder {
  type W[T] = Option[T]
  type B[T] = T

  def bind[S,T](bound: Option[T], f: T => Option[S]) = {
    bound.flatMap(f)
  }
}
```

What if "!" is ambiguous?

Our "binding" `!` is defined on `Tree/Computation`:

```
val x = ! o      ⟼      val x = ! conv(o)
```

```
val b = ! true   ⟼      val x = false
```

If `!` is defined on the type aliased by `Tree/Computation`, there is indeed an issue.

Why so parametric?

`https://github.com/lampepfl/dotty/issues/15176`

tldr; bug in type equality in presence of aliasing/path-dependent types.