AST builders, Scala & metaprogramming

Optional semester project

Noé De Santo

1 Introduction

1.1 A motivating example

Simple planes [1] is a computer game where you have to create your own planes, which you can then fly.

More interestingly, it is possible to write a program which will control the plane using a dedicated programming language, funky trees [2]. This language is however expressed in XML, doesn't directly support usual control-flow structures and isn't type safe.

This makes this language both difficult to explain to people without programming experience, as well as quite frustrating to people with such experience. This is quite a shame, as the highly visual and concrete aspect of controlling a plane could make a great pedagogical tool to teach computer science.

1.2 Designing a better language

One solution to this issue would be to design a "higher-level" programming language for simple planes, treating XML as its "low-level" assembly.

However, designing such a language and implementing a full compiler for it would quickly become time consuming, and would most likely result in pretty minimal (feature-wise) language.

Another quicker solution would be to use an pre-existing language, and embed our language in it directly, resulting in a domain-specific language (DSL) [3].

Scala, being a feature-rich language, makes a pretty good candidate for such usages; e.g. it allows custom operators definitions and has excellent support for algebraic data types.

Thus, an attempt at writing our DSL could yield something along these lines:

```
val thrust = Variable("thrust")
val roll = Variable("roll")
thrust := 1 andThen
While(roll < 360) {
  roll := roll + 10 andThen
  waitFor(1)
}
```

We can however already notice some downsides, among others:

- The variables are named twice and must be declared before all of the actual code;
- An operator must be used for sequencing (`andThen`).

Overall, the user ends up manipulating the resulting AST a bit too directly, despite the DSL making it a bit more natural. This is especially problematic, as the end-user (the one using the DSL) might be learning programming, in which case manipulating a program (for the plane) using a program (the one in Scala) might be confusing. Even for more seasoned programmers, this can be confusing if they are not used to computer languages representation.

1.3 Metaprogramming

One possible way to alleviate the mentioned issues is to use metaprogramming. In particular, Scala has some young yet powerful metaprogramming facilities [4].

> ***Aside.*** From now on, we will be talking about multiple kinds of AST at the same time: the AST of the Scala program, which will be manipulated through metaprogramming, and the AST which is the results of this program, built using the DSL and thus defined by the user. We will refer to the former as "Scala's AST" and to the later as "user's AST".

In fact, these facilities allow to manipulate the Scala's program AST directly. This would allow to define some function on ASTs such that

$$'\{t_1; t_2\} \longmapsto \textsc{sequence}(t_1, t_2)$$

Where SEQUENCE is some function (or constructor) producing a user AST.

1.4 Goals

The goal of this project, as well as report, is thus to explore how metaprogramming can be used for AST generation. More precisely, starting from the less-funky-tree compiler [5], it aims at identifying elements which could be generalized and abstracted, as to create a metaprogramming-powered DSL library.

These ideas being also kind of reminiscent of F#'s computation expressions [6], this project will also aim at seeing potential common abstractions as well as differences between the two.

2 A toy example

We will now introduce, ROB a tiny toy example, tailored to show multiple aspects of such DSLs.

Rob is a fictional robot living on a toric grid which can be programmed with some assembly, and we will be building a DSL in Scala for it. We won't define said assembly, as the focus of this project is on the DSL definition, and is not concerned with the compilation of the DSL into assembly. We instead define the AST which will be generated by the DSL in fig. 1.

In particular, we would like the DSL to allow to write such a program:

```
// Goal: move rob to the origin
// Also indicates if already centered
// on some coordinate

while robOrientation !== Orientation.N do
  robRotateLeft

if robY === Number(0) then
  robOutput(0)
else
  while robY !== Number(0) do
    robAdvance

robRotateRight
if robX === Number(0) then
  robOutput(1)
else
  while robX !== Number(0) do
    robAdvance
```

We will also assume, for future examples, that a goto function can be implemented, which takes two coordinates and moves Rob to this position.

3 AST builders

The core idea to solve the issue at hand is to use Scala metaprogramming, more precisely its reflection capabilities, which allow to directly manipulate the program's AST.

The problem can then be split into the following sub-problems:

1. Allow the DSL designer to specify how to transform a Scala construct into its counterpart;
2. Make the DSL type-check;
3. Given a Scala AST, traverse it and identify which instances of the different constructs (variable definition, expression sequencing, …) have to be transformed into their user's AST counterpart.

(1) can easily be solved using a trait, that the DSL developer will have to implement:

```
trait AstBuilder[
  Tree[_],
  Variable[_] <: Tree[_]
] {
  def initialize(variable: Variable[T],
    value: Tree[T]): Tree[Unit]
  def assign(variable: Variable[T], value:
    Tree[T]): Tree[Unit]
  def sequence[S](first: Tree[S], second:
    Tree[T]): Tree[T]
  def ifThenElse[T](cond: Tree[Boolean],
    thenn: Tree[T], elze: Tree[T]): Tree[T]
}
```

The type parameter `Tree[_]` is the type of the produced AST (i.e. the user's AST), and `Variable[_]` is used to represent variables inside of the user's AST. Its type parameter allows the user to have type-safety inside the DSL, e.g. it allows to add a type-safe + operator to Rob's DSL:

```
extension (lhs: AST[Int])
  def +(rhs: AST[Int]) = Add(lhs, rhs)
```

With the previous solution, the transform part of (3) can be done by simply replacing the Scala construct by a call to the correct method. We can also decide to apply the transformation if and only if its operand have the user's AST type (i.e `AST[_]` in Rob's case). A simple example would be:

```
val i: Int = 0
val t: Variable[Boolean] = Number(1)
```

which should be transformed into:

```
val i: Int = 0
assign(variable("t"), Number(1))
```

Here, `i` is not transformed because it does not have type `Variable[_]`, whereas `t` has.

Finally, a simple solution to (2) is to introduce implicit conversion when needed:

```
var x: Variable[Int] = Number(0)
if x === Number(1) then
  x = Number(1)
else
  x = Number(-1)
```

which should be transformed into:

```
sequence(
  initialize(variable("x"), Number(0)),
  ifThenElse(
    ===(variable("x"), Number(1)),
    assign(variable("x"), Number(1)),
    assign(variable("x"), Number(-1))
  )
)
```

```
enum Orientation:
  case N,S,W,E

sealed trait AST[+T]
class Variable[+T](id: String) extends AST[T]
// Basic constructs
case class Assign[T](vr: Variable[T], vl: AST[T]) extends AST[Unit]
case class Sequence[+T](l: AST[?], r: AST[T]) extends AST[T]
// Arithmetic
case class Number(i: Int) extends AST[Int]
case class Add(l: AST[Int], r: AST[Int]) extends AST[Int]
// Logic & control flow
case class ===[T](l: AST[T], r: AST[T]) extends AST[Boolean] // Test two values for equality
case class !==[T](l: AST[T], r: AST[T]) extends AST[Boolean]
case class Ite[+T](cnd: AST[Boolean],thenn: AST[T],elze: AST[T]) extends AST[T]//If-then-else
case class While[T](cnd: AST[Boolean], body: AST[?]) extends AST[Unit]// While-loop
// IO
case object robOrientation extends AST[Orientation] // Get current orientation
case object robX extends AST[Int]
case object robY extends AST[Int]
case object robInput extends AST[Boolean] // Read 1 bit from (rob's) stdin
case class robOutput(out: AST[Boolean]) extends AST[Unit] // Write 1 bit to (rob's) stdout
// Navigation
case object robRotateLeft extends AST[Orientation] // Rotate and return new orientation
case object robRotateRight extends AST[Orientation]
case object robAdvance extends AST[Unit] // Advance to tile in front
```

Figure 1. Rob's AST.

This code sample won't compile for multiple reasons:

- The condition of a control-flow structure must be a `Boolean`;
- The variable initialization as well as both assignations have type `Tree[Int]` as their right-hand-side, whereas the left-hand-side is a `Variable[Int]`.

Also, the fact that both assignations have type `Unit`, whereas they will have type `Tree[Unit]` after translation, can disallow some code we would like to allow. However, if implicit conversions are added, more precisely

- `Conversion[Tree[Boolean], Boolean]`,
- `Conversion[Tree[T], Variable[T]]` and
- `Conversion[Unit, Tree[Unit]]`

this codes is then accepted by the compiler. These implicit conversions do not "do" anything: they simply indicate to the compiler that such programs should compile, and are erased during transformation, as the resulting user's AST does not require a conversions e.g. from `Tree[Boolean]` to `Boolean]` inside the condition of an if-then-else.

4 More is less

The tricks introduced in the previous section have some serious caveats, that we will now discuss.

The most glaring one is the use of implicit conversions. These are known to be a kind of pandora's box of confusing behaviors; here however, the issue goes a bit deeper. The issue is that those conversions are "contextual", they only make sense in a particular case: the `Boolean` one only makes sense when used in a control-flow structure; the tree to variable one should only be used to assign to a variable; the `Unit` one is only meaningful when applied to an assignation to a user's AST variable.

```
val x: Variable[Boolean] = robInput
val b: Boolean = x // Should not be allowed

def f(u: Tree[Unit]) = ???
f(()) // Should not be allowed
```

It is possible to mitigate this by implementing the conversions such that they throw an exception (at runtime) when called. Since they are erased anyway when used correctly, this will only throw when used incorrectly.

Even though practical and easy to implement, this idea is terrible for the user, as something which is statically known results in a runtime error, which is dirty.

It is possible to report such misuse during AST transformation, by detecting and reporting unwanted conversions, and then stopping compilation. This however is error-prone and requires to correctly predict every corner-case.

This motivated the abandon of features which relied too heavily on implicit conversions; more precisely, if-then-else, while-loops, variable assignations and user-level variables inference are not supported by the library.

Instead, a special operator is used for user-level variables definition and assignation. To be nice to use, the operator for user-level variables had to be a prefix operator; since scala only allows a handful of prefix operators (`+, -, !, ~`), `!` was chosen since it reminds of F# computation expressions (section 6). The symbol `=!` was chosen for assignation to be coherent with `!`, even though `:=` could have been a better choice (it is both already known and has no risk of being confused with `!=`). The reflection API allows to test for variable mutability, as such assignation (using `=!`) on non-mutable variables can be detected and reported.

Replacements can be imported using a mixin for the control-flow structures, which are simply implemented as functions with judicious names and currying of arguments.

```
var x: Tree[Int] = ! Constant(0)
If( Constant(true) ) {
  x =! Constant(1)
}{
  x =! Constant(-1)
}
```

Figure 2. Snippet of code showing the notations used.

5 Some extra considerations

5.1 Side effects ordering

One might wonder how the following piece of code should be translated:

```
robRotateLeft
println("Hello")
robAdvance
```

We would like its value to be an AST which makes Rob rotate left and then advance, but we would also like `Hello` to be printed when said AST is generated.

The trick to achieve this, and more generally to preserve the order of side effects inside a computation, is to add synthetic binders inside of code blocks:

```
val computation$1 = robRotateLeft
println("Hello")
sequence(computation$1, robAdvance)
```

5.2 Meta variables

One lingering question is "Why rely on an operator to decide when to translate a variable definition? Why not reject anything which cannot be translated?"

This comes from a will to allow manipulation of ASTs as Scala constructs, as well as allowing Scala to be used as some kind of the meta-language for the DSL.

The simplest example is to allow toggling of debug code:

```
val debug: Boolean = ???

val orientation = ! (???)
If (orientation !== Orientation.N) { // (1)
  if debug then // (2)
    robPrint(orientation === Orientation.S)
  else
    robNop
  // Remaing of the code
  ???
}
```

Here, (1) will be executed by Rob, whereas (2) toggles whether something should be printed by Rob at runtime.

In particular, if this whole program is passed for translation, `debug`'s definition should not be changed, staying at Scala-level, whereas `orientation` should be transformed as described before.

5.3 Meta sequences

This idea can be pushed further: it would be practical to declare scala-loops which generate code.

A simple example using the `goto` function we introduced earlier:

```
for i <- 0 until 3 yield
  goto(Constant(i), Constant(2*i))
```

which should be transformed into:

```
sequence(
  goto(Constant(0), Constant(0)),
  sequence(
    goto(Constant(1), Constant(2)),
    goto(Constant(2), Constant(4))
  )
)
```

This can easily be done by adding an additional case when transforming a sequence $t_1; t_2$. Previously, this got translated

if and only if both had type `Tree[_]`; now, it should also be translated when

$$t_1\texttt{: Seq[Tree[\_]]} = s_1 :: s_2 :: \ldots :: s_n$$

in which case it should be translated into

$$(s_1 :: s_2 :: \ldots :: s_n :: t_2)\texttt{.foldRight(sequence(\_, \_))}$$

5.4 Inlining all the things

One thing which can seem inconsequential is whether the translation functions must be **`inline`** or not.

However, making them **`inline`** allows for compile-time features opt-out. Let's assume for a moment that we would like the Rob DSL to not allow variable assignation, to encourage a functional programming style. This could then be done as follows:

```
inline def assign(vr: AST[T], vl: AST[T]) =
  scala.compiletime.error(
    "Be like rob, be functional!"
  )
```

6 Computation expressions

The F# programming language has an interesting feature called "Computation Expressions". The core idea is that the language has some "meaningless" (in the sense that they don't have any semantics) keywords, which are disabled by default. These keywords include, but are not limited to **`let!`**, **`return`** and **`for`**.

However, the language allows the definition of builders, which will give meaning to these keywords. The developer can then opt-in to one of those builders, which will use the definitions inside the builder to give meaning to the keywords.

```
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)
```

6.1 AST builders as computations

This minimalist example already allows to paint some parallels with our AST builders:

- Both have a special kind of value definition, either the user-level variable definition (`!`) or the **`let!`** keyword;
- Both use a builder to define the meaning of the special constructs.

This begs the question as whether AST builders could be expressed as computation expressions. In fact, computation expressions were designed to be extremely abstract, as to fit many different usages; the methods in the builder do not even need to conform to any specific type signature, and allow any types to be used as long as everything type-checks at use-site. It would thus seems natural for AST builders to be expressible in this framework.

It turns out to be possible, but require to step away from the "typical signature" of the `Bind` method. Even though no signature is enforced, `Bind` usually has the type `M<'T> -> ('T -> M<'S>) -> M<'S>` (one could recognize the signature of monads' bind/flatmap). The first argument is the computation being bound, the second argument is the remaining of the computation, which expects to receive the result of the computation getting bound, and the returned value is the result of the overall computation.

In the case of ASTs, `M` would be `Tree[_]`, and "computation" simply becomes a synonym for an AST. The second argument would again be the remaining of the computation, however it doesn't expect the result of the computation. It wouldn't make sense, as a computation is an AST, and its "result" is not known at AST building time, only at its execution time. Instead, something representing that variable, allowing to access its result at execution time, should be passed to the subsequent computation. This yields the following type signature `Tree[T] => (Variable[T] => Tree[S]) => Tree[S]`, whose body would be something among the lines of:

```
def bind[T, S](
  m: Tree[T],
  f: Variable[T] => Tree[S]
): Tree[S] =
  // User provided variable generator
  val v = freshVariable[T]
  sequence(
    // Initialize v with value (of tree) m
    initialize(v, m),
    // Allow f to refer to v
    f(v)
  )
```

6.2 Implementation

Based on the previous observation, it was decided to implement some kind of computation expressions in Scala, and to re-implement the AST builder interface as a computation expression.

6.2.1 Implementing computation expressions.

When implementing computation expressions, some choices had to

be made. In particular, to simplify usage, a type signature was fixed. The fixed signature is the "typical signature" (which allows to express monadic computations), with the tweak mentioned earlier to accommodate the `Variable[_]` type when binding a value.

Also, some of their original features do no make sense in Scala, either because the language already supports some of these (e.g. **`for`**-loops), or because they were deemed to not be a good safety/value tradeoff, such as **`while`**-loops (see section 4 for a discussion on this in the context of AST builders).

On the other hand, assignation is not supported by computation expressions (F# is a functional language after all). Our custom operator `=!` could be implemented as a simple extension method. However, this approach has one downside: it allows values to be reassigned, which is something we would like to prevent, so that the written code still mirrors Scala semantics.

```
val x: Variable[Int] = ! constant(0)
x =! 1 // Should be rejected
```

This can easily be ensured using metaprogramming; thus, the `=!` operator was added as a feature of our implementation of computation expressions. Of course, in case re-assignation doesn't make sense in the context of a particular computation expression, the designer can opt-out of it (as detailed in section 5.4).

The final interface a user has to define is shown in fig. 3. The `init` method is used to implement lazy computations, which we won't discuss in the present report.

A final interesting note is that our "meta sequences" (section 5.3) map almost exactly to the **`for`**-loops in F#'s computation expressions, the only difference being that we allow to combine any construct which has type `Seq[Computation[_]]` to a following `Computation[_]`, whereas F# only allow it if the sequence is generated by a **`for _ yield _`**.

6.2.2 `AstBuilder`'s new implementation.

With all of the previous considerations in mind, defining an AST builder as a computation expression just requires a few lines of code (see fig. 4). Note that it could be made even shorter, but some methods are renamed (e.g. `unit` into `constant`) to provide more meaningful name in the context of ASTs and programing languages.

6.3 Tradeoffs

Implementing the `AstBuilder` as a computation expression of course comes with some tradeoffs.

Among the cons, we can cite the loss of some tiny, yet nice features, which do make sense when constructing ASTs, but don't in the more general context of computations expressions. A concrete example is the naming of variables. When the builder needs to generate a fresh variable, it calls `freshVariable`, which is not aware of the name the variable has in Scala, which prevents it from having the same name.

This could be added by adding the name of the (Scala) variable as an argument to `bind`, which would then pass it to `freshVariable`. However, since this was not a defining feature, nor was it useful to any other computation expression, this feature was dropped when migrating to the computation expressions framework.

On the pros side, the most important one is simply code simplicity. The features of computation expressions being more general and abstract that the ones of AST builders, the code ended up being simpler yet more robust. This is especially great as, from personal experience, such code can have some subtle bugs and be pretty difficult to understand; as such, making the code simpler is a huge win.

6.4 A tiny shadow on the wall

One might wonder why the computation type, `Computation[_]` is a type parameter instead of a (path-)dependent type, as this would be more fitting.

This relates to a bug in Dotty (issue #15176), which prevents proper type testing when mixing dependent types and metaprogramming.

This made it impossible to consistently detect computations (as this is done using type equality) if the said type is dependent, instead of passed as a parameter.

This is also the reason why computation expressions' for-loops are not part of the presented interface in fig. 3, even though we mentioned that such loops are supported: properly abstracting them would require a dependent type, which yields the issue mentioned above. As such, this feature has a fixed (i.e. defined by the library for all computations, not by the computation developer) implementation, as we considered that making this feature available at all was more important than allowing the user to customize its behavior.

7 Discussion

7.1 ExProc library

The result of everything we mentioned in this report is a library to express computation expressions and AST builders in Scala, nicknamed ExProc (**Ex**pression **Proc**essor)[1].

Two fully functional compilers for two video game languages were also built on top of this library, both available on Github:

- https://github.com/Ef55/hrm-scala-compiler;
- https://github.com/Ef55/less-funky-trees (which is an adaptation of [5]).

[1] https://github.com/Ef55/scala-expression-processor

7.2 Future work

Richer AST builders. The features of the AST builders were highly dictated by the example compiler which were meant to be built on top of it. Even though great efforts were put into making the available features as general as possible, this also means that features which were not needed at all were not implemented.

One could explore some of those missing features, such as support for functions definition at the user-language level.

More general computation expressions. As mentioned, the original computation expressions from F# do not enforce any typing restrictions, which is not the case of our implementation.

One could try to lift this restriction. The current implementation could in fact already support this (all of the builder's method are accessed through reflection), but some care might be needed to make the potential errors understandable to both the DSL developer and the end user.

References

[1] Jundroo. *Simple Planes*. URL: https://www.simpleplanes.com/.

[2] "SnoWFLakE0s". *Funky Trees - A Guide*. URL: https://snowflake0s.github.io/funkyguide/.

[3] Peter H. Salus. *Handbook of Programming Languages. Little Languages and Tools*. Macmillan Technical Publishing, 1998. ISBN: 1578700108.

[4] Nicolas Stucki, Aggelos Biboudis, and Martin Odersky. "A Practical Unification of Multi-Stage Programming and Macros". In: 53.9 (2018), pp. 14–27. ISSN: 0362-1340. DOI: 10.1145/3393934.3278139.

[5] Anatolii Kmetiuk. *Funky Trees Compiler*. URL: https://github.com/anatoliykmetyuk/less-funky-trees.

[6] Tomas Petricek and Don Syme. "The F# Computation Expression Zoo". In: *Practical Aspects of Declarative Languages*. Ed. by Matthew Flatt and Hai-Feng Guo. Springer International Publishing, 2014, pp. 33–48. ISBN: 978-3-319-04132-2.

```
trait ComputationBuilder[Computation[_]] {
  type Bound[T]

  // Bind a value val x = ! m; f(x)
  def bind[T, S](m: Computation[T], f: Bound[T] => Computation[S]): Computation[S]
  // Combine two computations l; r
  def combine[T, S](l: Computation[T], r: Computation[S]): Computation[S]
  // Create a computation
  def unit[T](t: => T): Computation[T]
  // Init the computation
  def init[T](c: () => Computation[T]): Computation[T]
  // Reassign a value b =! v
  def assign[T](b: Bound[T], v: Computation[T]): Computation[Unit]
}
```

Figure 3. The interface of a computation expression builder.

```
trait AstBuilder[Tree[_]]
extends ComputationBuilder[Tree] {

  // User must define
  type Variable[T] <: Tree[T]
  def freshVariable[T]: Variable[T]
  def initialize[T](va: Variable[T], init: Tree[T]): Tree[Unit]
  def constant[T](t: T): Tree[T]
  def combine[T, S](l: Tree[T], r: Tree[S]): Tree[S]
  def assign[T](b: Variable[T], v: Tree[T]): Tree[Unit]

  // Computation expression implementation
  type Bound[T] = Variable[T]
  def bind[T, S](m: Tree[T], f: Variable[T] => Tree[S]): Tree[S] =
    val v = freshVariable[T]
    combine(initialize(v, m), f(v))
  def unit[T](t: => T): Tree[T] =
    constant[T](t)
  def init[T](c: () => Tree[T]): Tree[T] =
    c()
}
```

Figure 4. AST builder as a computation expression.