From Zero to Three

Building a User Friendly DSL with Scala 3

2021-02-13

Jakob Odersky

Overview

- designing a domain specific language (DSL) for privacy preserving computations
- features used from Scala 3
- experiences as early Dotty adopters

Background

- Inpher Inc (https://inpher.io)
- system to run Privacy Preserving Computations (PPC)
- specifically, Multi-Party Computation (MPC)
  - public = f(private)
  - well-known protocols to evaluate primitives (addition, multiplication, etc)
  - virtual machine implementing these protocols
  - compiler to compose, verify and optimize
- diverse background: some software engineering, cryptography, other disciplines

Past, Present, Future

- Compiling to Preserve Our Privacy
  - Scala Days 2019 https://www.youtube.com/watch?v=44EL11N3tOs
  - an external DSL to build MPC circuits
  - includes a more detailed overview of MPC and the runtime architecture
- From Zero to Three
  - building a user-friendly DSL with Scala 3
- Speakeasy: privacy preserving algorithms made practical
  - Scala Days ~~2020~~ 2021 (?)
  - an embedded DSL for composing privacy-preserving programs
  - will have more content

Why Move from External to Embedded?

- most “interesting/hard” work is done on an intermediate representation
- building user-level API requires a lot of work for marginal gains
  - fast growing complexity of an external DSL is not well suited for startups
- embedding in Scala allows us to skip implementing front-end systems such as parsers

Building the DSL

Goals

- make it simple to read, but also to **write**
- users must be able to **quickly find and fix** things by **themselves**
- **error messages** are very important
  - exceptions can sometime give better messages than type errors
- users don’t necessarily care about how things work
  - **don’t fall in love with your own code**

Approach

- decouple **structure** from **user-level** API
  - staging
  - allows experimentation with different backends
  - in this talk: from user-level to staged representation
- make it easy to **find definitions and implementations**
  - easy understanding and editing by users

```
// import _root_.{ppclib => ppc}
import ppclib as ppc
import ppc.Dataset

@main def run(): Unit =

  val a: Dataset = ppc.readCsv("A.csv").map{x =>
    x * 2
  }

  val b: Dataset = ppc.readCsv("B.csv").map{x =>
    ppc.debug(s"the value is $x")
    x * 2
  }

  val result: Dataset = ppc.multiparty {
    val fp: ppc.mpc.FixedPoint = b()
    ppc.mpc.reveal(a() + fp)
  }

  // do something with the result, typically you would do something more
  // involved than just printing the representation
  println(result)
```

Structure

```
package ppclib

class Dataset(parent: Builtin)

/** This corresponds to the DSL's Abstract Syntax Tree. It's the canonical
 * representation of a user program. */
enum Builtin {
 case ReadCsv(name: String)
 case Map(inputs: Seq[Dataset], fn: (MapContext, Seq[Partition]) => Seq[Partition])
 case Mpc(inputs: Seq[Dataset], fn: (mpc.Context, Seq[mpc.FixedPoint]) => Seq[mpc.FixedPoint])
}
```

Scala 3 Concepts

A Selection of Concepts from Scala 3

- will focus on a subset today
  a. top-level definitions
  b. context functions
  c. new metaprogramming API (quotes & splices, and TASTY reflection)
- out of scope: what we do with the staged representation

Concept 1: Top-Level Definitions

- familiar concept in many other languages
- no indirection
- used to define user-level API

```
1  package ppclib
2
3  def readCsv(name: String): Dataset = Dataset(Builtin.ReadCsv(name))
4
```

Concept 2: Context Functions

- “functions that have implicit parameters”
- https://dotty.epfl.ch/docs/reference/contextual/context-functions.html

Context Functions: Introductory Example

```
def buildGraph(f: Builder => Unit) = ???
def addEdge(from: String, to: String, builder: Builder): Unit = ???

buildGraph{ builder =>
  addEdge("A", "B", builder)
  addEdge("B", "C", builder)
}
```

```
def buildGraph(f: Builder ?=> Unit) = ???
def addEdge(from: String, to: String)(using builder: Builder): Unit = ???

buildGraph{
  addEdge("A", "B")
  addEdge("B", "C")
}
```

Context Functions: Introductory Example

```
def buildGraph(f: Builder => Unit) = ???
def addEdge(from: String, to: String, builder: Builder): Unit = ???

buildGraph{ builder =>
  addEdge("A", "B", builder)
  addEdge("B", "C", builder)
}
```

```
def buildGraph(f: Builder ?=> Unit) = ???
def addEdge(from: String, to: String)(using builder: Builder): Unit = ???

buildGraph{
  addEdge("A", "B")
  addEdge("B", "C")
}
```

Context Functions: Accessing the Environment

- access contextual information quickly
  - environment within a map
  - without changing other code

```
trait MapContext {
  def debug(message: String): Unit // the runtime will implement this
}
```

Context Functions: Accessing the Environment

```
def map(fn: Partition => Partition): Dataset = ???
map{ part =>
  part * 2
}

def mapWithContext(fn: (MapContext, Partition) => Partition): Dataset = ???
mapWithContext{ (ctx, part) =>
  ctx.debug("message")
  part * 2
}

def debug(message: String)(using ctx: MapContext) = ctx.debug(message)
def map(fn: MapContext ?=> Partition => Partition): Dataset = ???
map{ part =>
  debug("message")
  part * 2
}
```

Context Functions: Composing Abstractions

- our language has constructs to express high-level PPC flows
- it **also** has its own specialized tools for only MPC
  - multiple datasets can be combined in an MPC call
  - within it, partitions are viewed as `FixedPoint` types

Context Functions: Composing Abstractions

```
def multiparty(fn: mpc.Context ?=> mpc.FixedPoint): Dataset = ???

class Dataset {

  def apply()(using mpc.Context): mpc.FixedPoint = ???

}
```

```
val a: Dataset = ???
val b: Dataset = ???

val result: Dataset = multiparty {
  val fp: mpc.FixedPoint = a()
  // do anything in the "MPC world" here
  mpc.solve(fp + b())
}
```

Context Functions: A Note on Error Handling

- Scala 3 suggests where implicits may be found
- `implicitNotFound` annotation is your friend

```
@annotation.implicitNotFound("This operation is only allowed within a map.")
trait MapContext
```

```
def debug(using ctx: ppclib.MapContext): Unit
This operation is only allowed within a map. bloop
Peek Problem (Alt+F8)    No quick fixes available
debug("")
```

Concept 3: (New) Macros

- https://dotty.epfl.ch/docs/reference/metaprogramming/macros.html
- hygienic: whose expansion is guaranteed not to cause the accidental capture of identifiers
- macros can be defined in same project as expansion-sites

Macros

- declare

```
inline def multiparty(fn: mpc.Context ?=> mpc.FixedPoint): Dataset = ${multipartyImpl('fn)}
```

- implement

```
def multipartyImpl(using qctx: Quotes)(expr: Expr[mpc.Context ?=> mpc.FixedPoint]): Expr[Dataset] =
```

Macros

- declare

```
inline def multiparty(fn: mpc.Context ?=> mpc.FixedPoint): Dataset = ${multipartyImpl('fn)}
```

- implement

```
def multipartyImpl(using qctx: Quotes)(expr: Expr[mpc.Context ?=> mpc.FixedPoint]): Expr[Dataset] =
```

Macros: Bridging Border Worlds

- we saw how context functions enable composition of abstractions
- we have a staged DSL
  - `case Mpc(inputs: Seq[Dataset], fn: (mpc.Context, Seq[mpc.FixedPoint]) => Seq[mpc.FixedPoint])`
  - need to know inputs at compile time

Macros: Quote and Splice

```scala
def multipartyImpl
  (using qctx: quoted.Quotes)
  (expr: Expr[mpc.Context ?=> mpc.FixedPoint]): Expr[Dataset] = {
  import qctx.reflect._

  val inputExprs: List[Expr[Dataset]] =
    finder.foldTree(Nil, expr.asTerm)(Symbol.spliceOwner)

  '{
    val inputs: List[Dataset] = ${Expr.ofList(inputExprs)}
    val builtin = Builtin.Mpc(
      inputs,
      (ctx, ins) => Seq($expr(using ctx)) // [1]
    )
    Dataset(builtin)
  }
}

// [1]: the real implementation would also need to lift inputs
```

Macros: TASTy Reflection

- typed abstract syntax trees
- stable API
- traverse and transform trees
- mixes seamlessly with expressions

```
// find all calls to dataset.apply()
object finder extends TreeAccumulator[List[Expr[Dataset]]] {
  def foldTree(acc: List[Expr[Dataset]], tree: Tree)(owner: Symbol) = {
    tree match {
      case Apply(Select(term, "apply"), _) if term.tpe <:< TypeRepr.of[Dataset] =>
        term.asExprOf[Dataset] :: acc
      case _ =>
        foldOverTree(acc, tree)(owner)
    }
  }
}
```

Macros: Going Further

- currently limited to M inputs, 1 output dependency modelling
- whitebox macros: return type dependent on input expression
  - allows returning more than one element
  - M-N graph modeling
  - very powerful, use with caution

```
transparent inline def multiparty[A](inline expr: mpc.Context ?=> A) = ...

val a: Dataset = ...
val b: Dataset = ...

val (c: Dataset, d: Dataset) = multiparty {
  (a(), b())
}
```

Misc: Naming is Hard

- import renames

```
import com.company.ppclib.{Dataset, readCsv}
import com.company.ppclib.mpc.{FixedPoint, reveal}

val a: Dataset = multiparty{
  val f: FixedPoint = readCsv("a")()
  reveal(f)
}
```

```
import com.company.{ppclib => ppc}
import ppc.mpc

val a: ppc.Dataset = ppc.multiparty{
  val f: mpc.FixedPoint = ppc.readCsv("a")()
  ppc.reveal(f)
}
```

- union types

```
def visibilityChecker: Builtin => Error | List[Visibility]

def eval(boolExpr: Xor | And)
```

Misc: Other

- enums
- extension methods in context parameters
- no wildcard imports!

Putting It All Together

- https://github.com/jodersky/ppclib

Alternatives

- top-level decls + context-functions => cake pattern
  - inheritance introduces implicit indirection
- code and structure decoupling => interpreter, free-monad style
  - macros enable a less cumbersome syntax

- work well if people already know Scala

Experiences

Experiences

- mill as build-tool, pytest for non-unit tests
- VSCode with Metals was our editor of choice
  - Intellij probably also good, but VSCode has great support for multi-language repos
  - (historical: original language had a custom syntax highlighter)
  - a few times there were incompatibilities with our build tool and/or Metals version
  - no work-blockers
- couple of minor Dotty bugs related to top-level functions
  - quickly fixed, never a blocker
- ecosystem lagging behind
  - to be expected
  - binary compatibility is great, except for macros
  - spent some time porting libraries
  - specifically the “Singaporean Stack” (projects by Li Haoyi)

Experiences (continued)

- newcomers needed some handholding to get familiar with the language
- lack of docs for 3 not a major problem
- some syntax needed explanation
  - nothing too surprising
  - most discussions we have regarding syntax are related to naming, not Scala syntax
  - scalafmt not quite there yet
- we work with non Scala experts; need to make sure the frontend is solid before release

Merci Beaucoup!

- snippets available: https://github.com/jodersky/ppclib
- more to come at Scala Days
- special thanks:
  - Manohar Jonnalagedda