Speakeasy

privacy-preserving programs,
composed in the real world

ScalaCon,
2021-11-04

Jakob Odersky
Manohar Jonnalagedda

Outline

- Intro
- Part I - What is Speakeasy?
- Part II - What is a PDD?
- Part III - What can one do with a PDD graph?
- Part IV - Speakeasy meets Scala 3
- Conclusion

Secure multi-party computation (MPC)

subfield of cryptography with the goal of creating methods for parties to **jointly compute a function** over their inputs **while keeping those inputs private**.

`f(a, b) = c`

revealed to both

secret to one party

secret to other party

		players		
				
shared zeroes	d	d_1	d_2	d_3
	j	j_1	j_2	j_3
	m	m_1	m_2	m_3
data		A	B	C
shared sum		S_1	S_2	S_3
		$S = S_1 + S_2 + S_3$		

dimitar

jakob

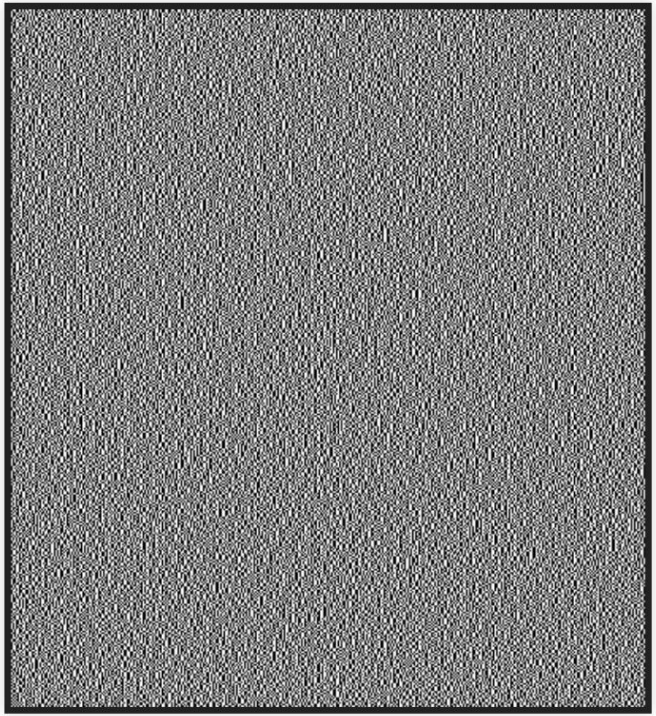
manohar

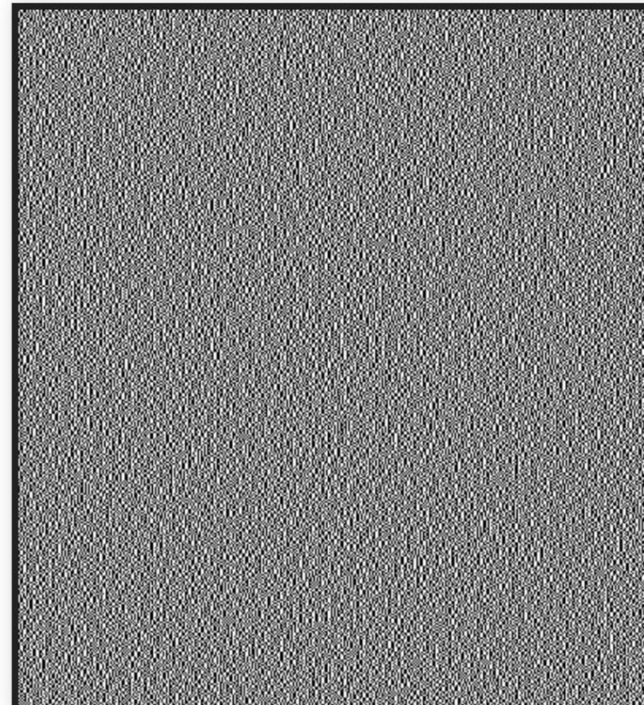
Secret shares for zero are **distributed**

sum

Shared sums are **revealed publicly**

Inputs

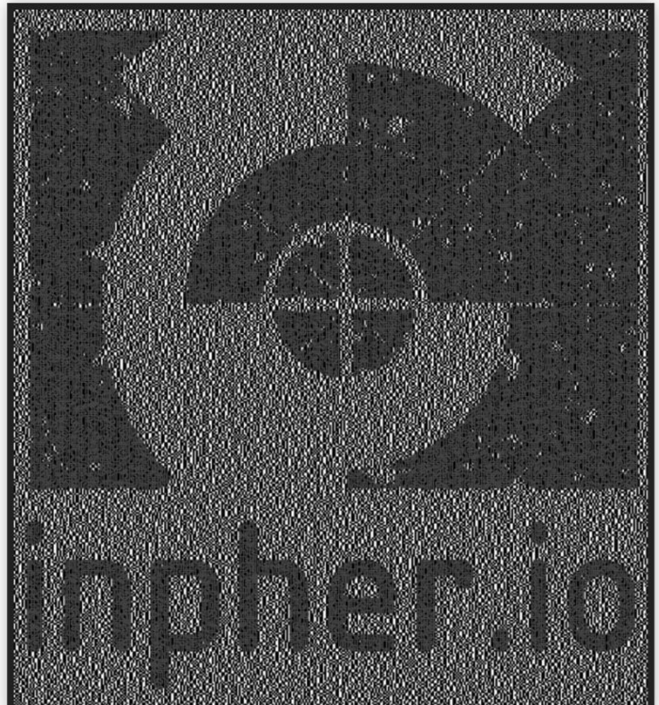




Reveal

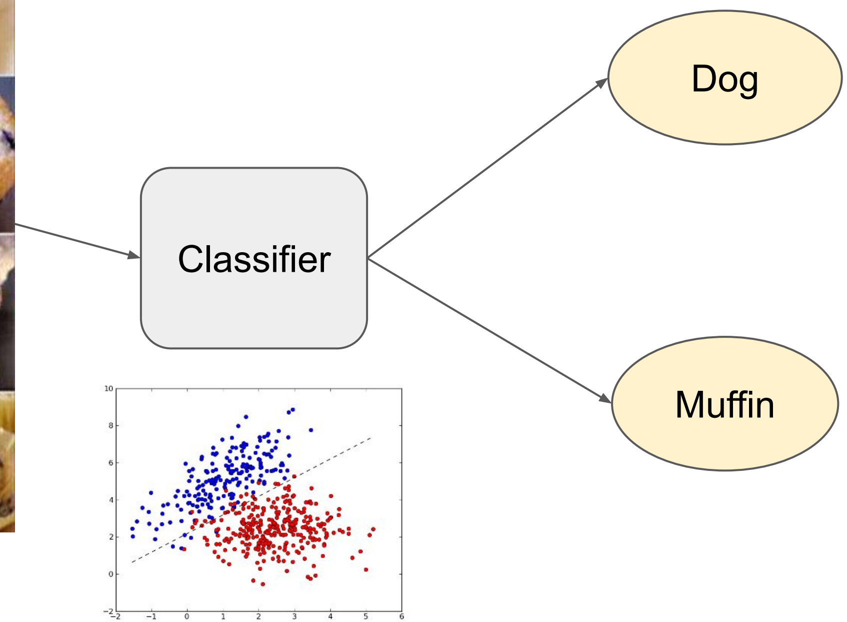
Secret Share

Result



Reveal Secret Share

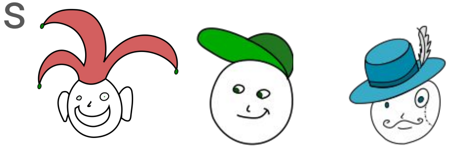
Classification: Chihuahua or Muffin ?



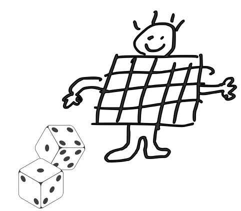
The Setup

- *Data analyst* who wants to perform privacy-preserving computations

- *Data owners* (players) that are honest but curious
  - Faithfully follow what they're told to do (honest)
  - If they can learn something, they will (curious)



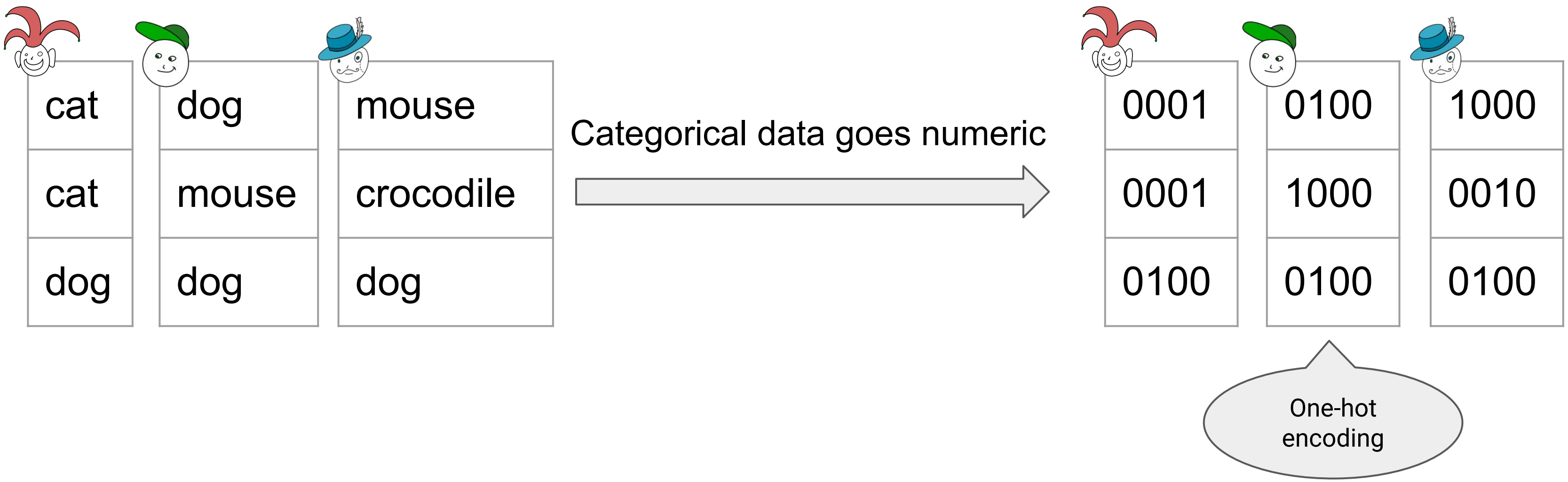
- A *runtime* that processes a computations request by the analyst
  - The runtime knows static metadata (e.g. dimensions)
- A *trusted dealer* for generating randomness for an mpc computation



MPC is not enough

- Data does not live in a vacuum
- Not all data is numeric
- Goes through other local processing pipelines
- May use other privacy-preserving techniques

MPC is not enough - making it numeric



Part I 

What is Speakeasy

What is Speakeasy

- A library for building, composing and executing privacy preserving programs.
  - based on Private Distributed Datasets (PDDs)
  - domain-specific language (DSL) for MPC and garbled circuit computations
  - high-level API for operations on stacked, distributed data
- Allows for plugging schedulers

Anatomy of a Speakeasy program

Execution amongst data owners

Value returned to the runtime

Loading/sourcing data

```
val a: Pdd[SourceTable] = se.Player("alice").readCsv("data.csv")
val b: Pdd[SourceTable] = se.Player("bob").readCsv("data.csv")
val c: Pdd[SourceTable] = se.Player("carlos").source(
  SourceTable.fromDoubles(
    Seq(Seq(-1.0, -2.0, -3.0),
        Seq(-4.0, -5.0, -6.0)
    )))
```

Local computations

```
val doubled: Pdd[SourceTable] = a.map { table => table.times(2) }
```

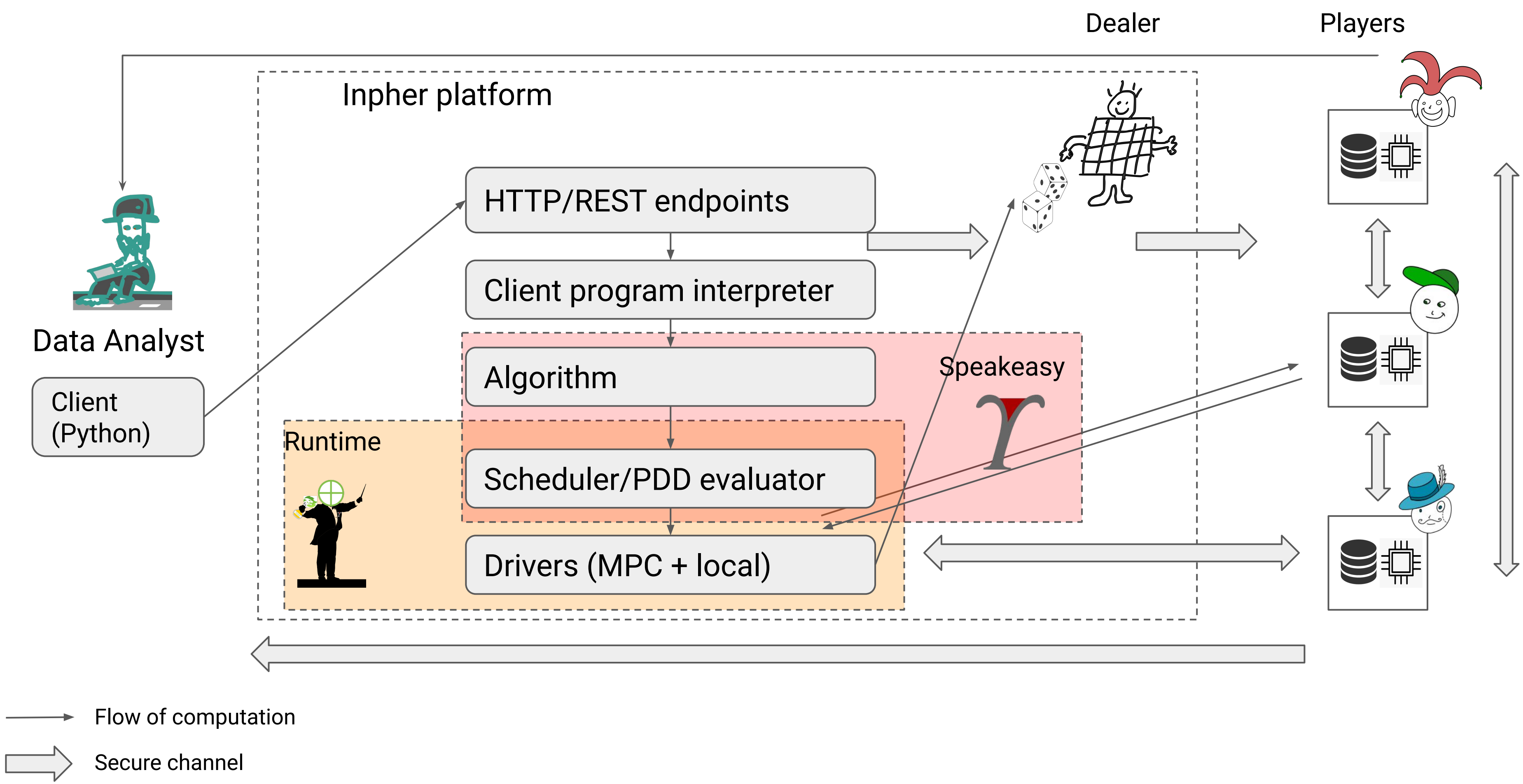
MPC computations

```
val (res1: Pdd[SourceTable], res2: Pdd[SourceTable]) = se.multiparty {
   val m1: mpc.Matrix = se.mpc.reveal(a.stacked() + b.stacked()).asPublic
   val m2: mpc.Matrix = (a.stacked() + doubled.stacked() + c.stacked()).asSecret
   (m1, m2)
 }
```

Evaluation of a result

```
val (result1: SourceTable, result2: SourceTable) = se.eval(res1, res2)
```

The Setup


Dealer
Players
Inpher platform
Data Analyst
Client (Python)
HTTP/REST endpoints
Client program interpreter
Algorithm
Speakeasy
Runtime
Scheduler/PDD evaluator
Drivers (MPC + local)
Flow of computation
Secure channel

Anatomy of a Speakeasy program

- A Speakeasy program is executed by a runtime
- The Pdd[A] world
  - Values of type Pdd[A] known only to their owners
  - Operations on pdds sent to the data owners
  - Operations in `multiparty` context compiled to Inpher's MPC engine
- The A world
  - Values of type A are visible to the runtime
  - Algorithm developers must be mindful of what the runtime can see

Part II 

What is a PDD

“

```
trait Pdd[A] {
 def owners: Set[Player]
 def dep: Operation
}
```

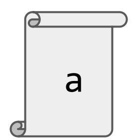
A PDD[A] **represents** some data of type A that may be distributed across multiple partitions.

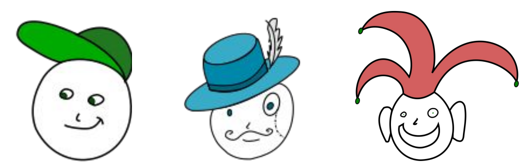
Each partition of a PDD is owned by one party in a privacy-preserving computation, and together they make up one logical dataset.

A PDD is always the result of an **operation**.

”

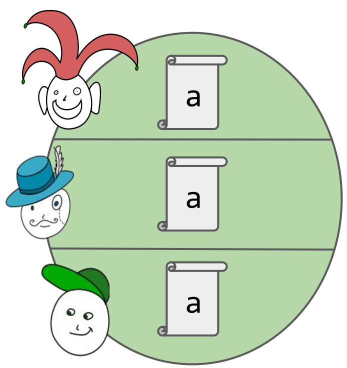
PDD = data + ownership



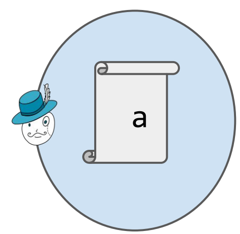


Kinds of PDDs

Uniform



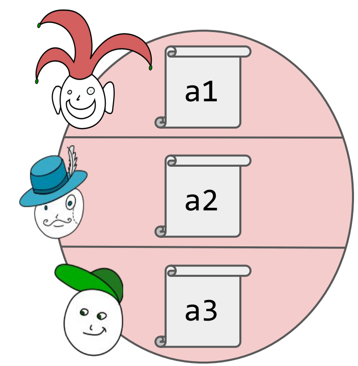
`UniformPdd[A]`



`PrivatePdd[A]`

Just an alias for frequent single-owner case

Secret-shared



`SecretPdd[A]`

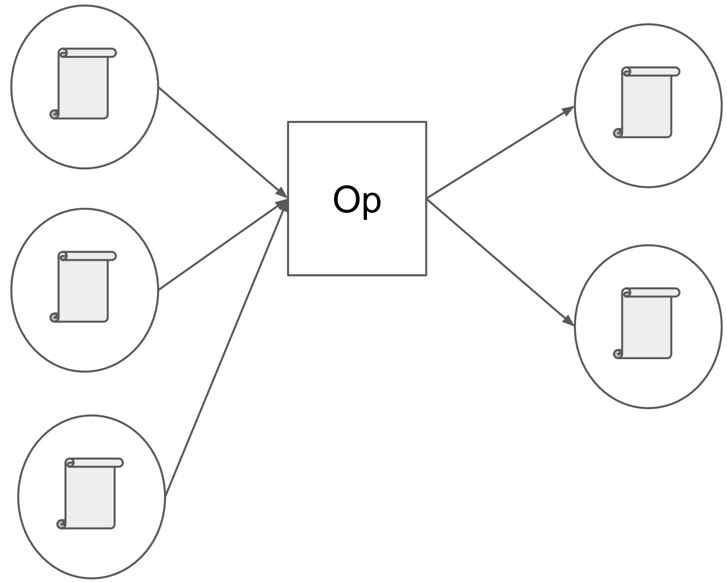
`a=a1+a2+a3`

Additive shares

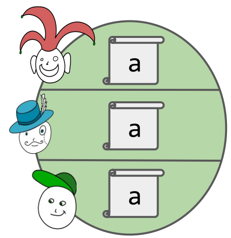
Operations on PDDs

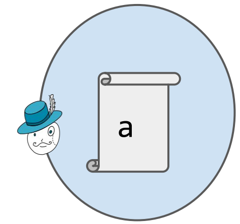
```
trait Pdd[A] {
 def owners: Set[Player]
 def dep: Operation
}
class SecretPdd[A] extends Pdd[A] { … }
class UniformPdd[A] extends Pdd[A] { … }
```

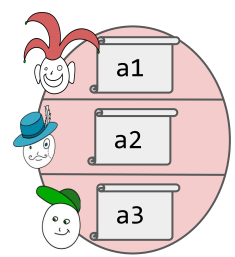
- Change data
- Change ownership
- Evaluate result



Operations on PDDs

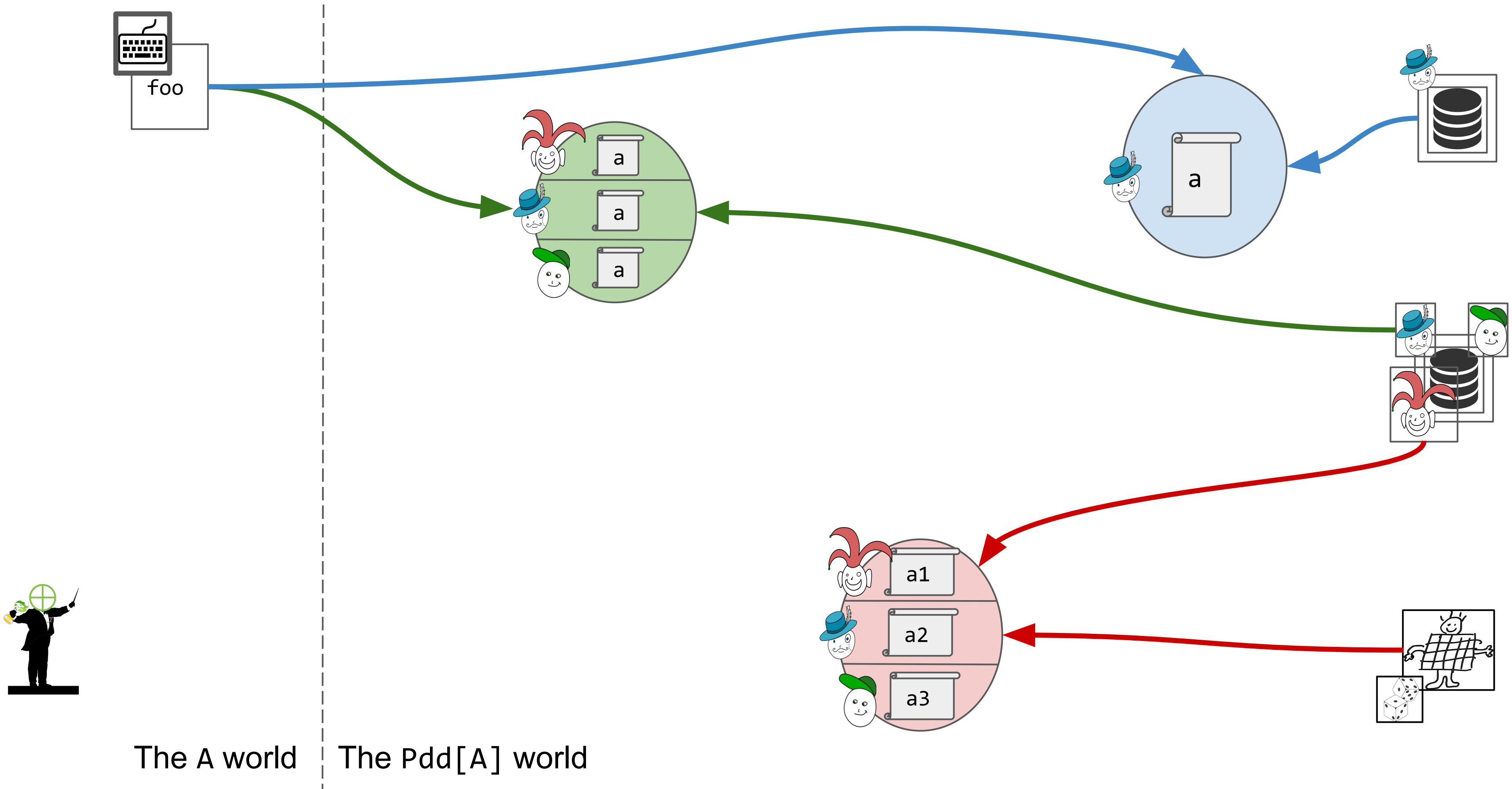






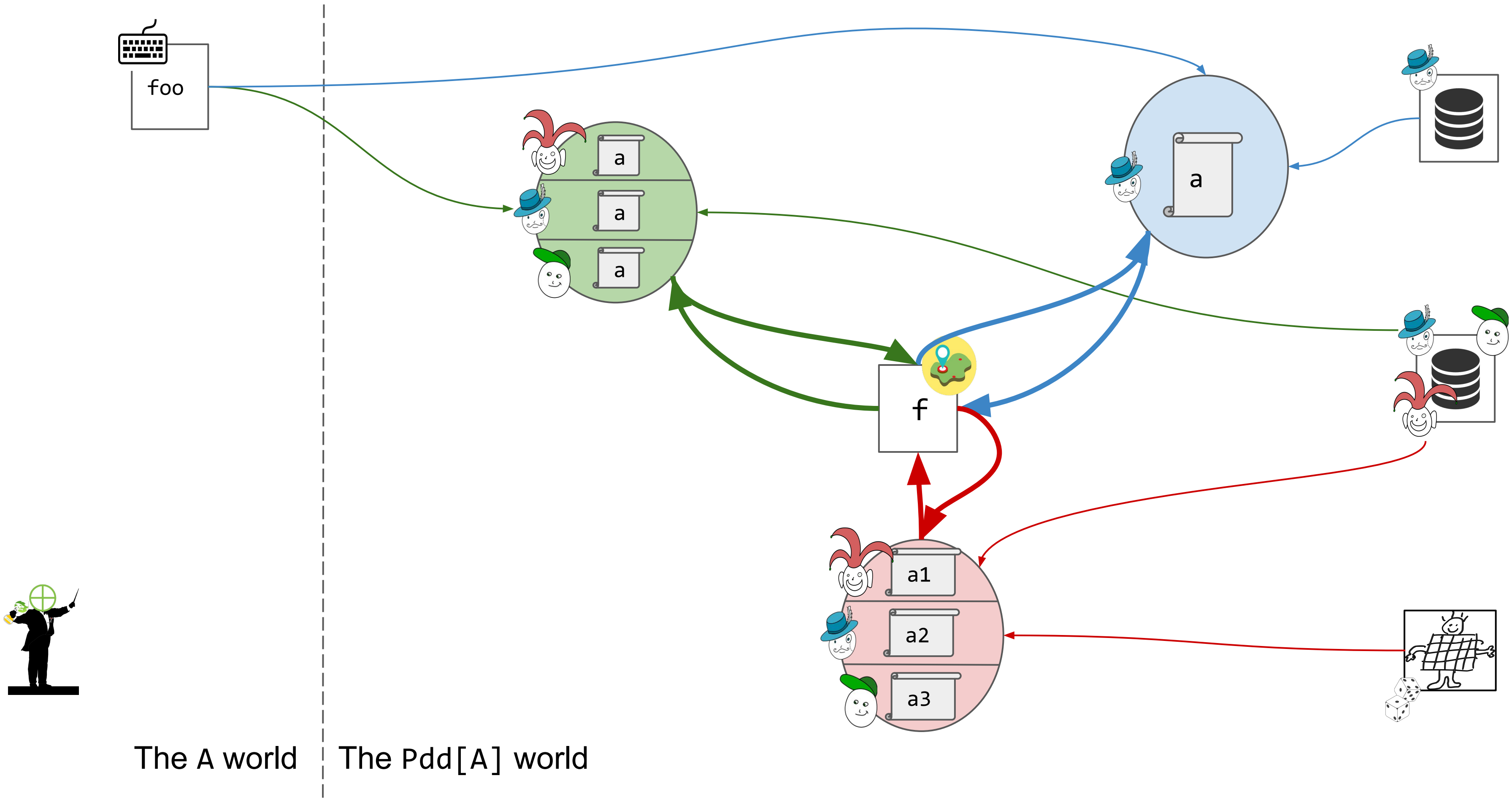
The `A` world | The `Pdd[A]` world

Operations on PDDs

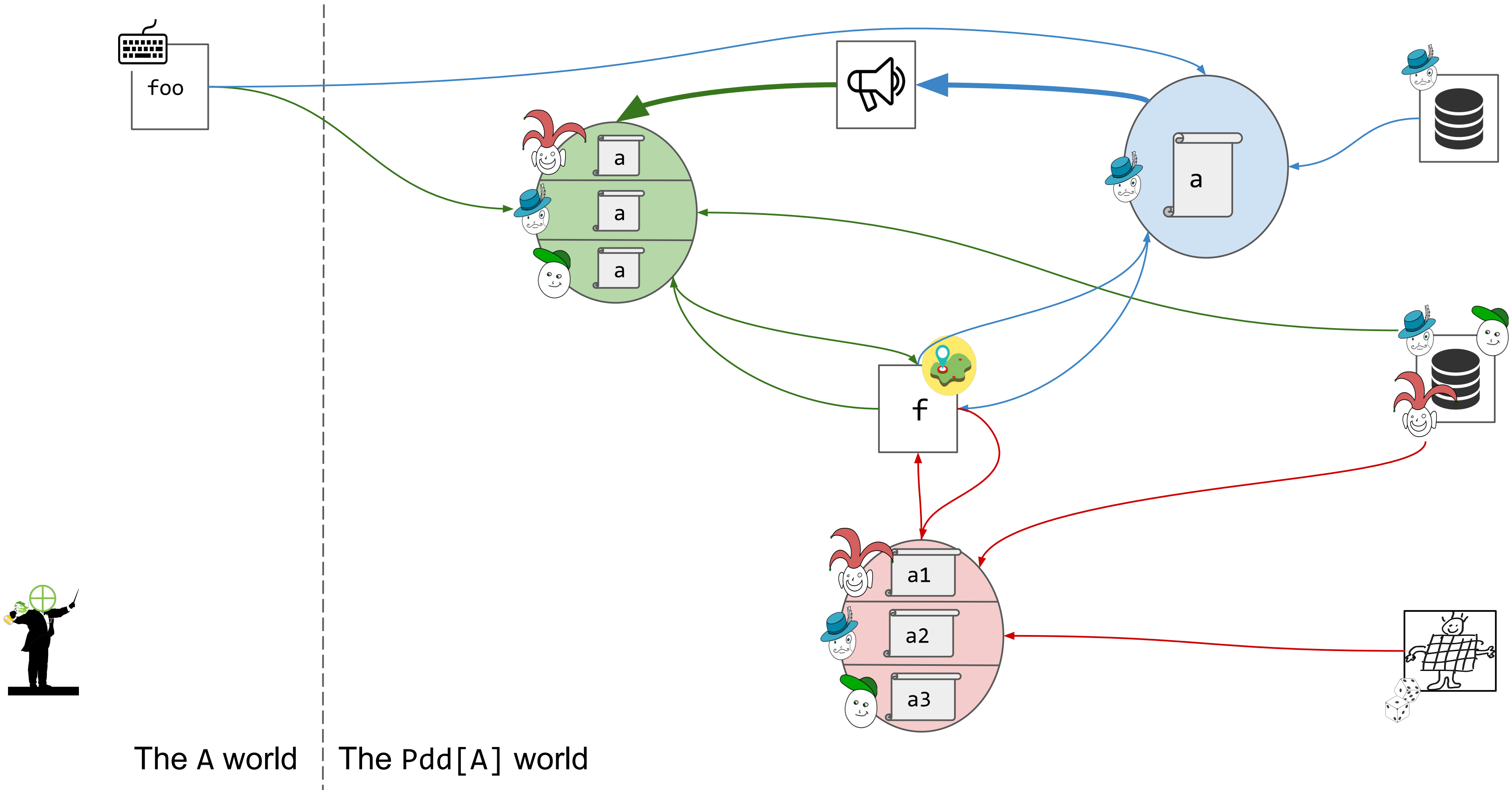


The A world | The Pdd[A] world

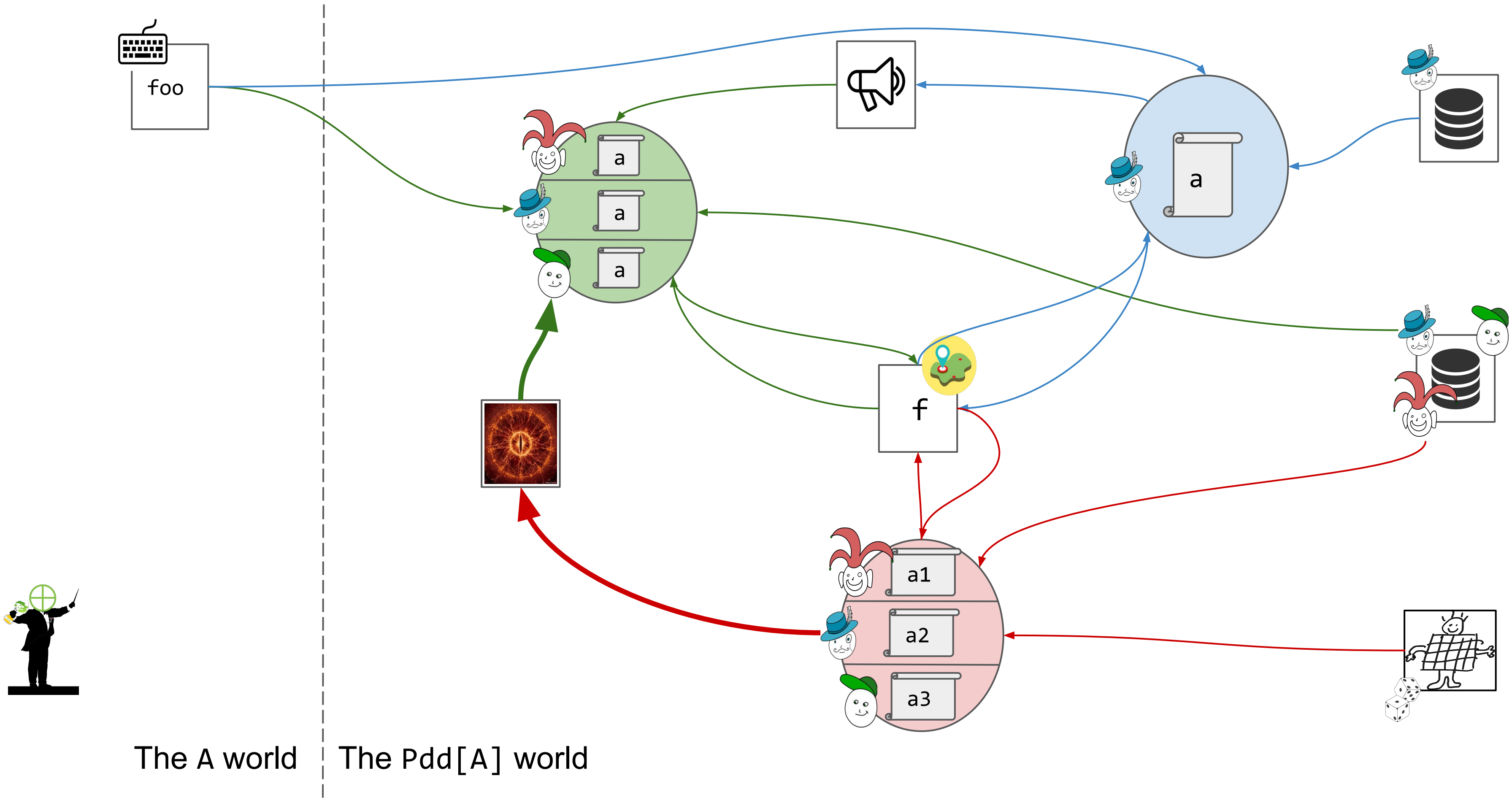
Operations on PDDs



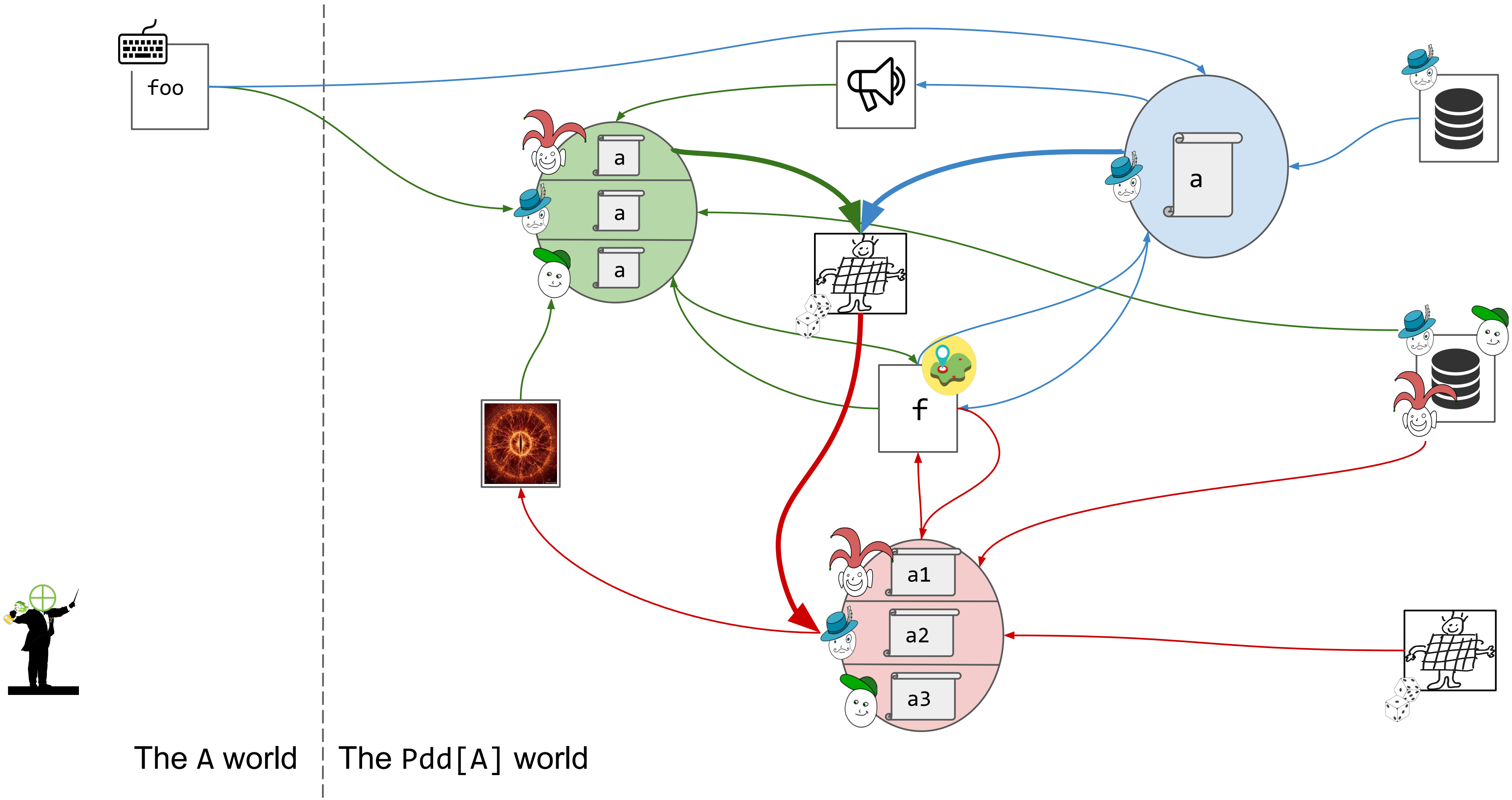
Operations on PDDs



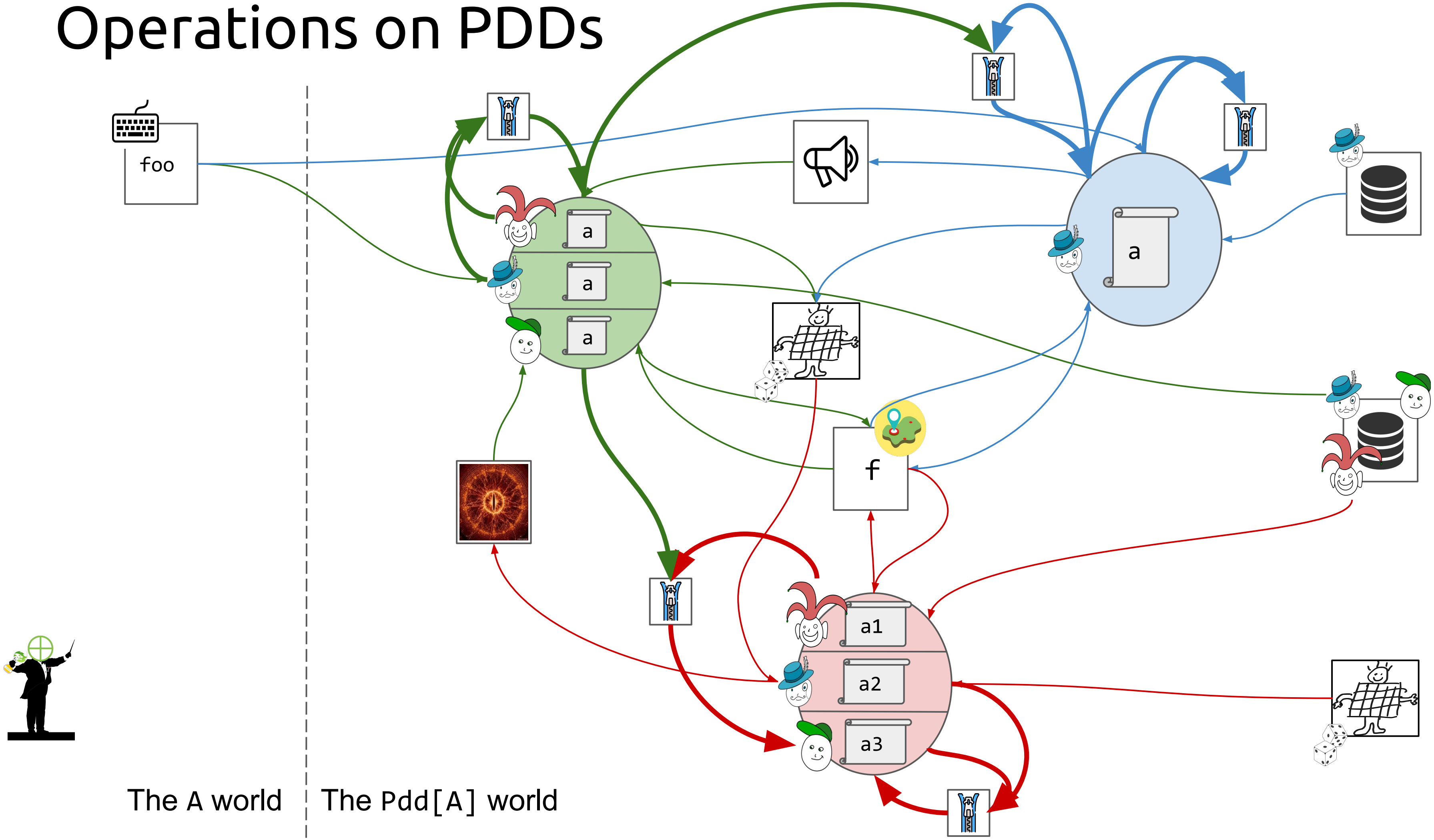
Operations on PDDs



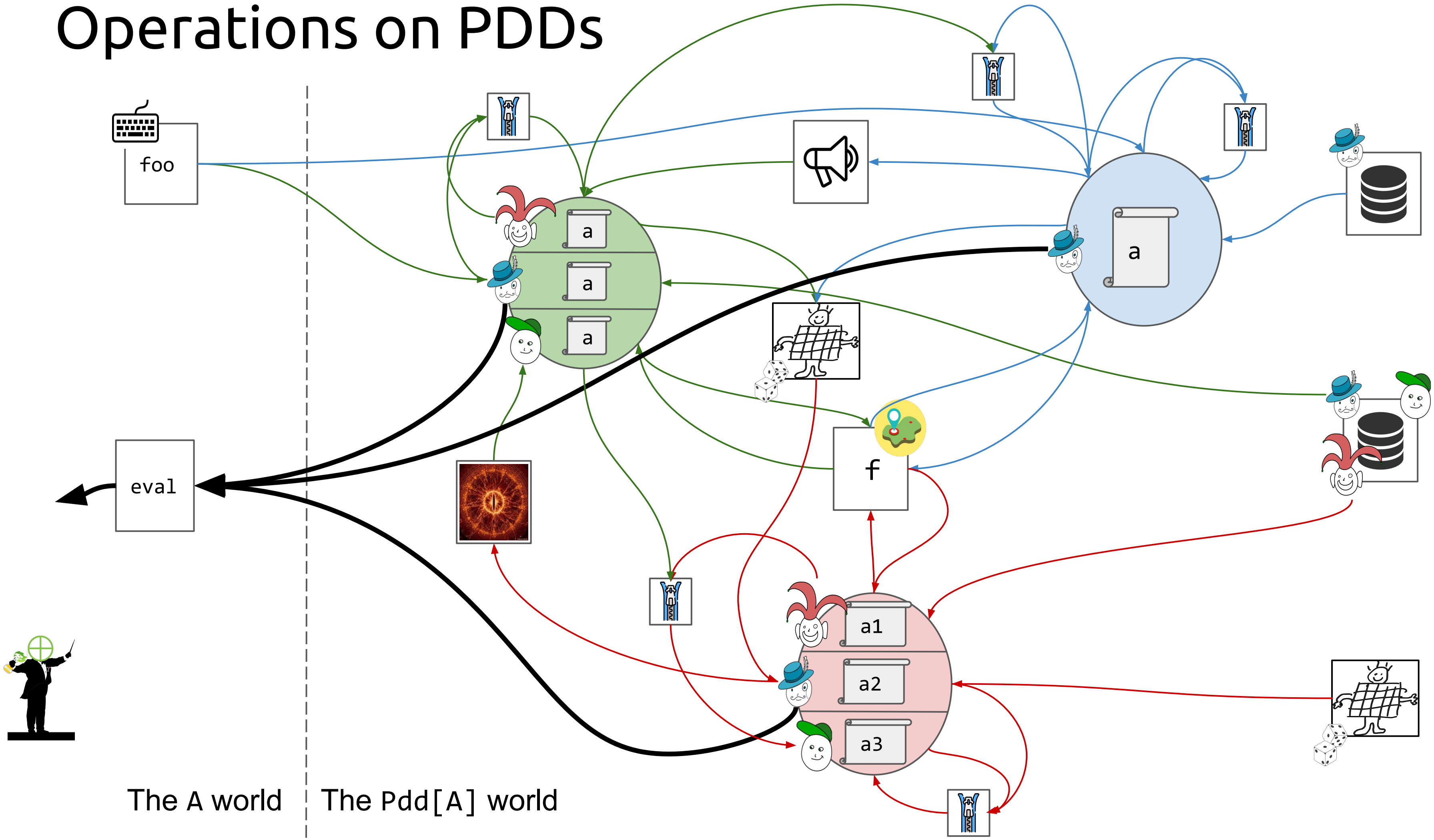
Operations on PDDs



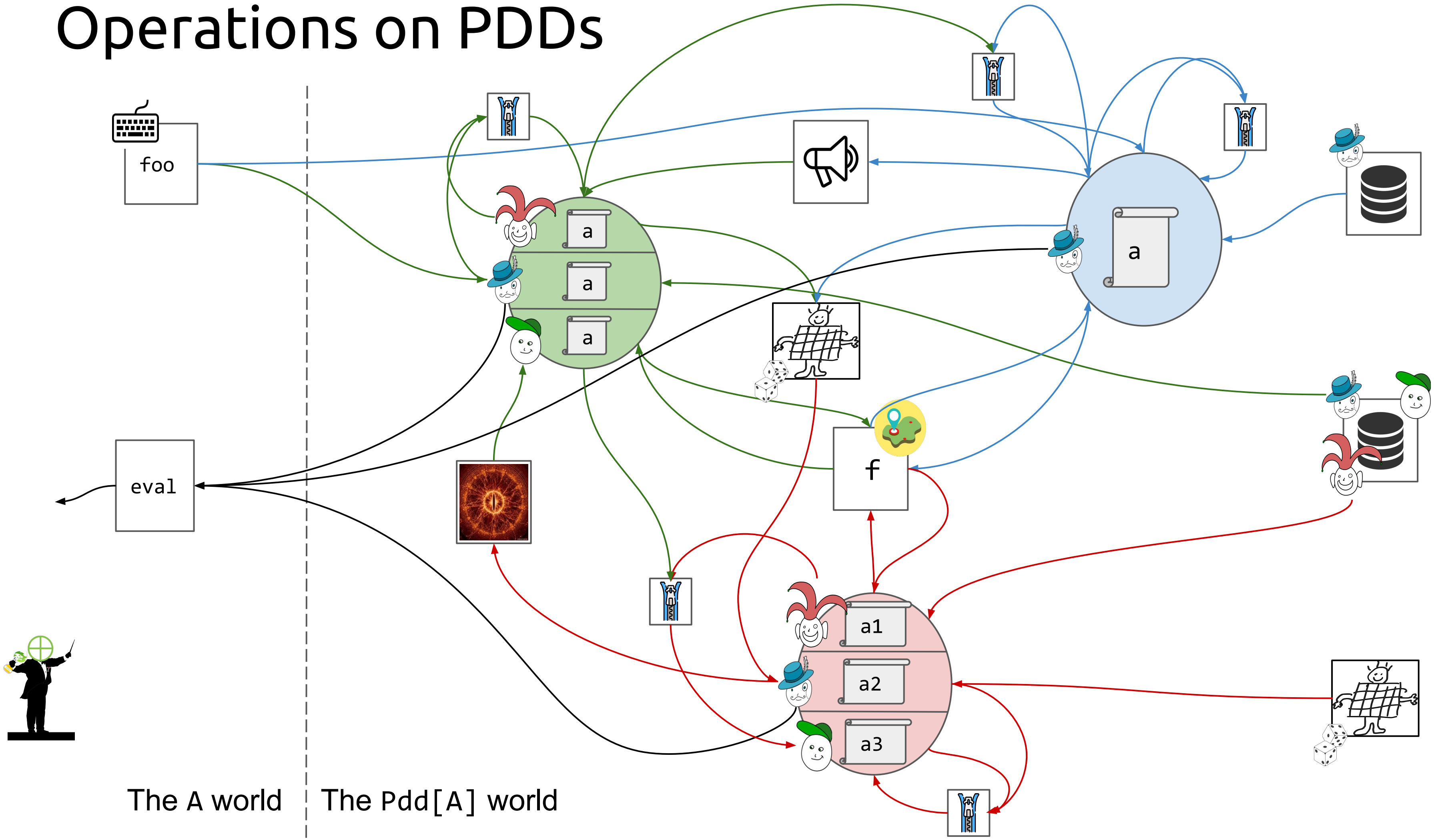
Operations on PDDs



Operations on PDDs



Operations on PDDs

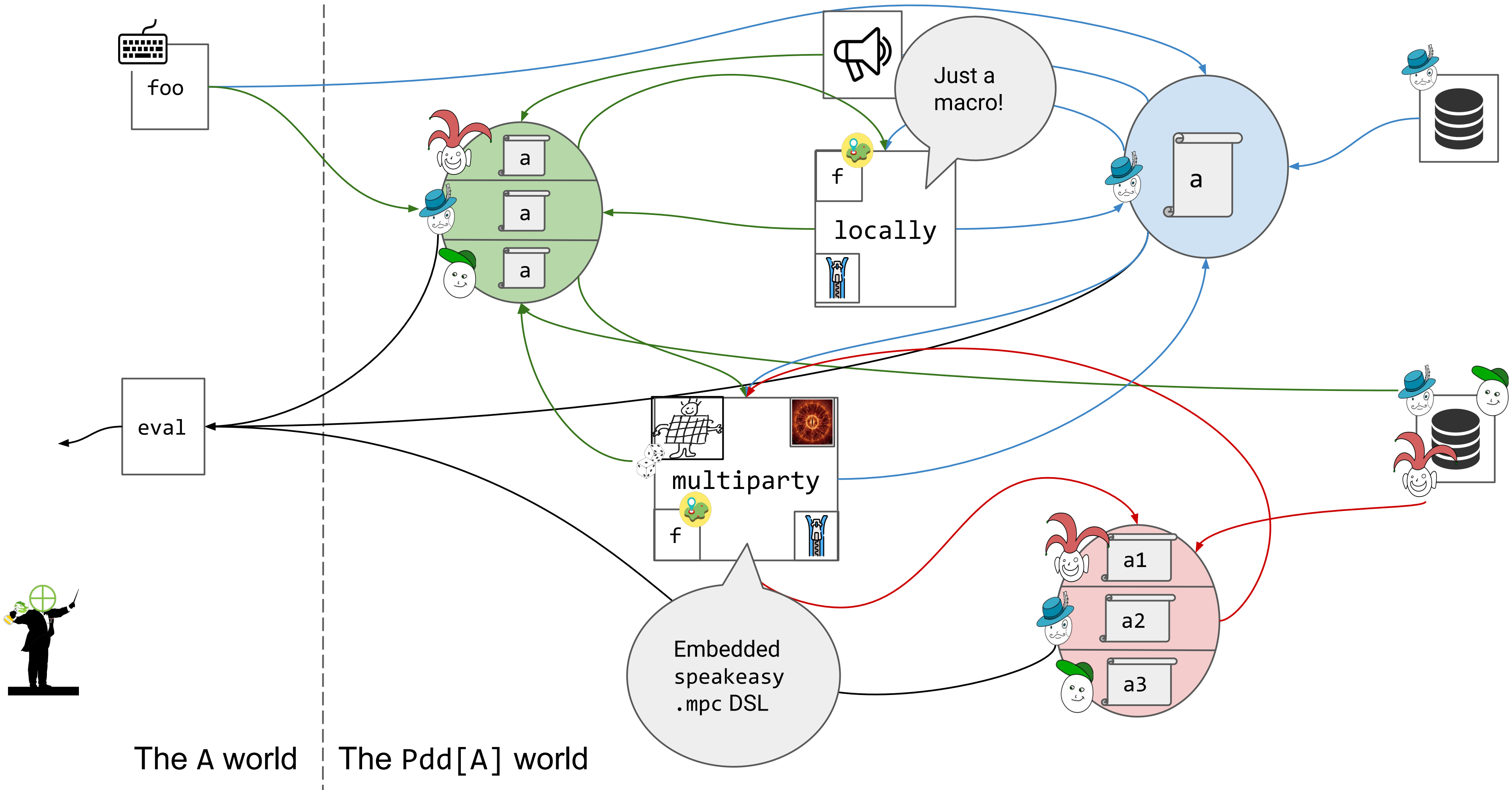


Operations on PDDs

```
class Player {
 def read[A]: Pdd[A]
 def source[A](a: A): UniformPdd[A]
}

def eval[A](pdd: Pdd[A]): A
```

```
trait Pdd[A] {
 def owners: Set[Player]
 def map[B]: Pdd[B]
 def zip[B]: Pdd[(A, B)]
}

class SecretPdd[A] extends Pdd[A] {
 def reveal: UniformPdd[A]
}

class UniformPdd[A] extends Pdd[A] {
 def broadcastTo(ps: Set[Player]): UniformPdd[A]
 def secretShare: SecretPdd[A]
}
```

Operations on PDDs - in practice



Operations on PDDs - in practice

```
// structure
case class ZipMap(
  outs: List[Pdd[_]],
  ins: List[Pdd[_]],
  fn: (Env, List[_]) => List[_]
) extends Operation(ins, outs)

case class Mpc(
  outs: List[Pdd[SourceTable]],
  ins: List[Pdd[SourceTable]],
  compile: (mpc.Context, List[mpc.Matrix]) => List[mpc.Matrix]
) extends Operation(ins), outs)
```

Embedded DSL

Context functions

```
// syntax
transparent inline def multiparty[R](inline body: mpc.Context ?=> R) = // tuple of PDD[?]
transparent inline def locally[R](inline body: Pdd.Env ?=> R) = // tuple of PDD[?]
```

Whitebox macros

Part III 

What one can do with a PDD Graph

Evaluating a PDD

There is **no single way** to evaluate a PDD!

- We can evaluate operations sequentially, in *topological* order
- We can evaluate operations in parallel!
  - “Optimistic” parallelism
  - Work-stealing parallelism
  - Other?
- The architecture of the system may dictate scheduling strategy

Bootleg: an example strategy for evaluating PDDs

Bootleg is a scheduler in which the evaluation of target PDDs is done through the following steps:

1. Computing the transitive closure of the operational dependencies of the target(s).
2. Mapping the closure to runtime tasks, responsible for evaluating operations.
   a. this is known as an *execution plan*
   b. *the exact nature of tasks depends on the architecture*
3. Scheduling the tasks such that:
   a. operational dependencies are evaluated in order
   b. in parallel as much as possible
4. Reconstruction of final result

Evaluating a PDD

- PDDs are *lazy:* they represent a logical dataset
- they're a dataflow modelling tool
- evaluating a (set of) target PDD(s) will run all operations leading to it (them)
- do something useful with the final result
    - e.g. reveal it or save it
- evaluation is pluggable
- **=> description and execution are decoupled**

Part IV

Speakeasy meets ~~Dotty~~ Scala III

Evolution of Speakeasy

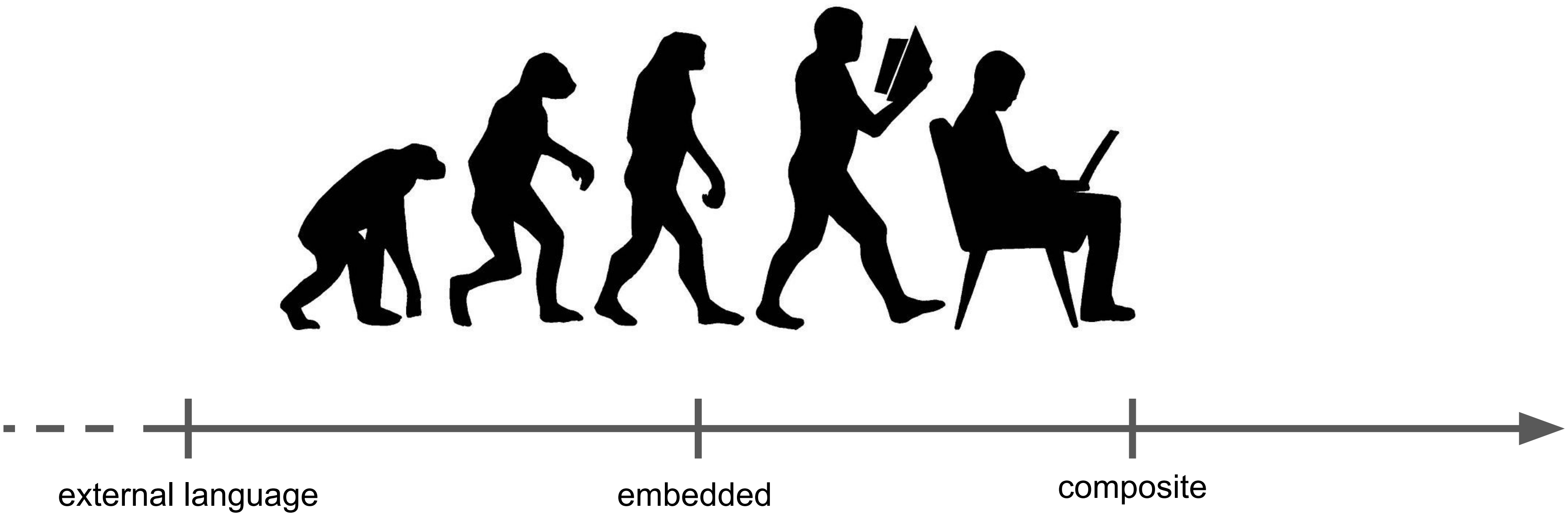


illustration: Johanna Pung, Wikimedia Deutschland, CC-SA

1. External Language

```
def solve(A: Matrix, b: Vector): Vector {
  var nrows: Int = xor.rows(A);
  var ncols: Int = xor.cols(A);

  var P: Matrix = xor.orthrand(nrows, ncols, -6);
  var Q: Matrix = xor.orthrand(nrows, ncols, -6);

  var PAQ: Matrix = P * A * Q;
  var Pb: Vector = P * b;

  xor.reveal(PAQ);
  xor.reveal(Pb);
  var r: Vector = xor.publicSolve(PAQ, Pb);
  return Q * r;
}
```

```
def linreg(y: Vector, X: Matrix): Vector {
  var A: Matrix = xor.transpose(X) * X;
  var b: Vector = xor.transpose(X) * y;
  return solve(A, b);
}

def main() {
  var X: Matrix = xor.input("X");
  var y: Vector = xor.input("y");
  var theta: Vector = linreg(y, X);
  xor.output(theta, "thetas");
}
```

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

2. Embedded Language

```
import speakeasy as se

def solve(A: se.Matrix, b: se.Matrix)(using se.Context): se.Matrix = {
  val nrows: Int = A.nrows
  val ncols: Int = A.ncols

  val P: se.Matrix = se.randOrth(nrows, nrows)
  val Q: se.Matrix = se.randOrth(ncols, ncols)

  val AQ: se.Matrix = A * Q
  val PtAQ: se.Matrix = P.t * AQ
  val Ptb: se.Matrix = P.t * b

  val r: se.Matrix = se.publicSolve(se.reveal(PtAQ), se.reveal(Ptb))
  Q * r
}
```

Scala Features Used in the Embedded DSL

- syntactic:
  - top-level functions
  - import renames

  => similarity to Python-based numeric libraries
  => easy to read and navigate

- structural:
  - builder passed as a "given context"

  => allows decoupling user syntax from program structure

  (alternative is mixin-based module system, aka "cake pattern", which is arguably less discoverable)

Builder Context

```
import speakeasy as se

def solve(A: se.Matrix, b: se.Matrix)(using se.Context): se.Matrix = {
  val nrows: Int = A.nrows
  val ncols: Int = A.ncols

  val P: se.Matrix = se.randOrth(nrows, nrows) (using se.Context)
  val Q: se.Matrix = se.randOrth(ncols, ncols) (using se.Context)

  val AQ: se.Matrix = A * Q (using se.Context)
  val PtAQ: se.Matrix = P.t * AQ (using se.Context)
  val Ptb: se.Matrix = P.t * b (using se.Context)

  val r: se.Matrix = se.publicSolve(se.reveal(PtAQ), se.reveal(Ptb)) (using se.Context)
  Q * r
}
```

Builder Context

```
val AQ: se.Matrix = (A * Q)(using Context)
```

calls

```
class Context {
  def mul(a: Matrix, b: Matrix): Matrix = {
    ... // check and infer some properties, modify some state
    node(a, b) // return an intrinsic node
  }
}
```

Builder Context, End-to-End Flexibility

```
// 1. create a context
val context: Context = initialize()

// 2. "compile" the user code
main(using context)

// 3. do something with the result

// generate code for our proprietary MPC VM
codegen(context)
// or, generate python code for comparison with numeric libraries
codegenPython(context)
// or, interpret for debugging
interpret(context)
```

Builder Context, End-to-End Restrictions

- guard rails
- can't call MPC functions outside of an MPC context
- @implicitNotFound gives friendly error messages

Embedded to Composite

- need way to express higher-level dataflows
  - MPC does not live in a vacuum
  - compiler's output must be more general than a specialized VM's instruction set
- want to offer one environment for algorithm developers
- want to keep the specialized MPC DSL for MPC dataflows

=> need a way to compose languages

```
val a = se.multiparty { ... }
val b = se.locally { ... }
val c = se.multiparty { ... }
```

Scala 3 Features Used in the Composite DSL

- context functions
  - allows to inject a given builder
  - effective in delimiting boundaries between languages
  - "plain old functional abstraction with nice syntax"
- macros
  - enable a light syntax
  - c.f. Future operations vs async/await

Macros and Context Functions

User Code

```
val a: Pdd[SourceTable] = ...
val b: Pdd[SourceTable] = ...

val res: UniformPdd[SourceTable] = se.multiparty{
  val input1: se.mpc.Matrix = a.stacked()
  val input2: se.mpc.Matrix = b.stacked()

  se.mpc.reveal(input1 + input2).asPublic
}
```

Macros and Context Functions

User Code

```
val a: Pdd[SourceTable] = ...
val b: Pdd[SourceTable] = ...

val res: UniformPdd[SourceTable] = se.multiparty{
  val input1: se.mpc.Matrix = a.stacked()
  val input2: se.mpc.Matrix = b.stacked()

  se.mpc.reveal(input1 + input2).asPublic
}
```

Structure

```
val dep = Mpc(
  outs = <synthetic>
  ins = List(a, b),
  compile = (
    ctx: mpc.Context,
    inputs: List[mpc.Matrix]
  ) => <synthetic>
)
UniformPdd(dep)
```

Macros and Context Functions

User Code

```
val a: Pdd[SourceTable] = ...
val b: Pdd[SourceTable] = ...

val res: UniformPdd[SourceTable] = se.multiparty{
  val input1: se.mpc.Matrix = a.stacked()
  val input2: se.mpc.Matrix = b.stacked()

  se.mpc.reveal(input1 + input2).asPublic
}
```

Structure

```
val dep = Mpc(
  outs = <synthetic>
  ins = List(a, b),
  compile = (
    ctx: mpc.Context,
    inputs: List[mpc.Matrix]
  ) => <synthetic>
)
UniformPdd(dep)
```

API

```
transparent inline def multiparty[R](inline body: mpc.Context ?=> R) = // tuple of PDD[SourceTable]
```

Macros and Context Functions

User Code

```scala
val a: Pdd[SourceTable] = ...
val b: Pdd[SourceTable] = ...

val res: UniformPdd[SourceTable] = se.multiparty{
  val input1: se.mpc.Matrix = a.stacked()
  val input2: se.mpc.Matrix = b.stacked()

  se.mpc.reveal(input1 + input2).asPublic
}
```

Structure

```scala
val dep = Mpc(
  outs = <synthetic>
  ins = List(a, b),
  compile = (
    ctx: mpc.Context,
    inputs: List[mpc.Matrix]
  ) => <synthetic>
)
UniformPdd(dep)
```

API

```scala
transparent inline def multiparty[R](inline body: mpc.Context ?=> R) = // tuple of PDD[SourceTable]
```

Macros and Context Functions

User Code

```scala
val a: Pdd[SourceTable] = ...
val b: Pdd[SourceTable] = ...

val res: UniformPdd[SourceTable] = se.multiparty{
  val input1: se.mpc.Matrix = a.stacked()
  val input2: se.mpc.Matrix = b.stacked()

  se.mpc.reveal(input1 + input2).asPublic
}
```

Structure

```scala
val dep = Mpc(
  outs = <synthetic>
  ins = List(a, b),
  compile = (
    ctx: mpc.Context,
    inputs: List[mpc.Matrix]
  ) => <synthetic>
)
UniformPdd(dep)
```

API

```scala
transparent inline def multiparty[R](inline body: mpc.Context ?=> R) = // tuple of PDD[SourceTable]
```

Macros and Context Functions

User Code

```
val a: Pdd[SourceTable] = ...
val b: Pdd[SourceTable] = ...

val res: UniformPdd[SourceTable] = se.multiparty{
  val input1: se.mpc.Matrix = a.stacked()
  val input2: se.mpc.Matrix = b.stacked()

  se.mpc.reveal(input1 + input2).asPublic
}
```

Structure

```
val dep = Mpc(
  outs = <synthetic>
  ins = List(a, b),
  compile = (
    ctx: mpc.Context,
    inputs: List[mpc.Matrix]
  ) => <synthetic>
)
UniformPdd(dep)
```

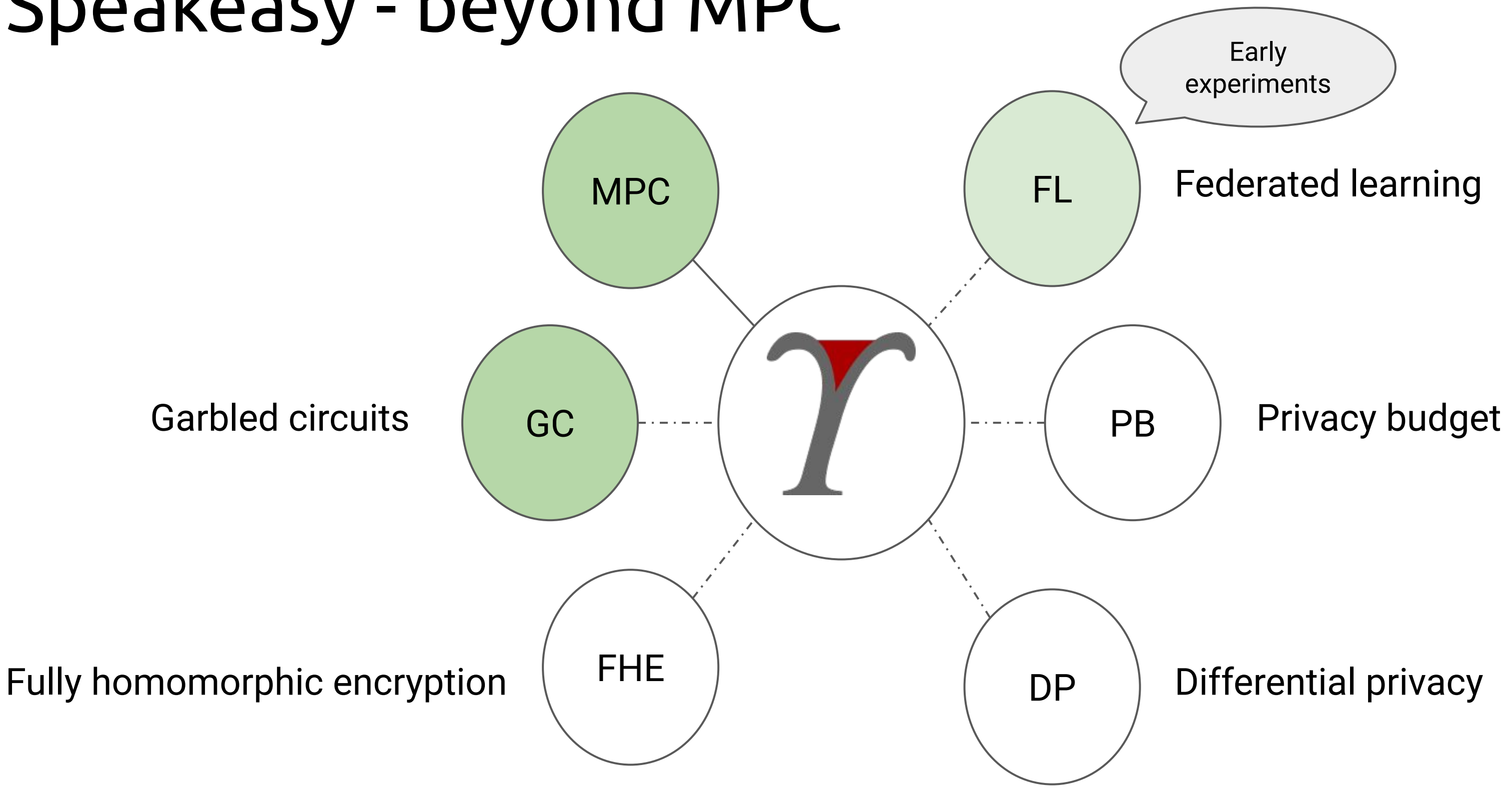
API

```
transparent inline def multiparty[R](inline body: mpc.Context ?=> R) = // tuple of PDD[SourceTable]
```

Key Insights

- Scala 3's contextual abstractions and new macro system are powerful DSL enablers
- **Abstracting structure from syntax is key**
  - manages perceived complexity
  - provides an escape hatch if needed
  - enables introspection and analysis
- Watch talk "From Zero to Three", given at Scala Love in the City 2021 for some more details
  - https://www.youtube.com/watch?v=wi-Pa0K1waI

Speakeasy - beyond MPC



For Your Consideration

Speakeasy Hosted Environment

`https://scalacon.tryit.xor.inpher.io/`

or

`tinyurl.com/wyt2kw`

Team

_	Name	Handle	Salutation
?	You	@CouldBe	The One

Thank You ScalaCenter

special thanks Vincenzo Bazzucchi

jakob@inpher.io
manohar@inpher.io

Merci Beaucoup!

https://scalacon.tryit.xor.inpher.io/

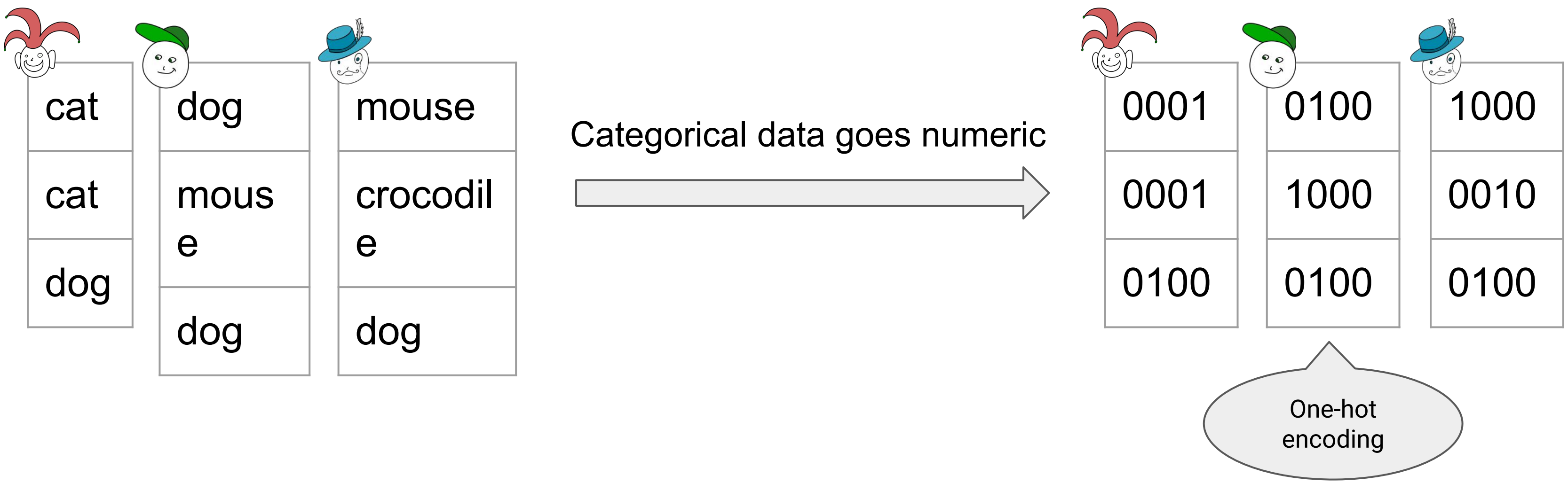
Languages

- Scala 54.5%
- Python 32.8%
- Jupyter Notebook 11.1%
- Shell 0.7%
- C++ 0.5%
- Makefile 0.3%
- Dockerfile 0.1%

Macros and Context Functions

```
@annotation.compileTimeOnly("() may only be called within speakeasy.multiparty")
/** Get a stacked PDD's representation in the context of an MPC computation.
  *
  * This is only available in a `speakeasy.multiparty` block.
  */
def apply(): mpc.Matrix = ???
```

Example - one-hot encoding



- Step 1: union of all pets -> “cat, crocodile, dog, mouse”
- Step 2: one-hot encoding of union -> “0001, 0010, 0100, 1000”
- Step 3: local conversion of categorical data

```scala
def onehotEncode(elems: Set[String]):
  Map[String, List[Int]] = ???

def onehotEncodeGlobally(
    allPets: List[PrivatePdd[List[String]]]):
    (UniformPdd[Map[String, List[Int]]],
     List[PrivatePdd[List[List[Int]]]]) = {
  val allOwners = allPets.map(_.owner)

  val elect = allOwners.head
  val electsPets :: othersPets =
    for (pets <- allPets) yield pets.map(_.toSet)

  val petsCommunicated =
    for (pets <- othersPets) yield
      pets.broadcastTo(elect)
```

```scala
val unionOfPets: PrivatePdd[Set[String]] = {
  val unionOfOthersPets =
    zipAll(petsCommunicated).map {
      allPets => allPets.foldLeft(Set.empty[String])(
        (acc, elems) => acc union elems)
    }
  electsPets.zip(unionOfOthersPets).map {
    case (elects, others) => elects union others
  }
}
val encodingMap = unionOfPets.map(onehotEncode)
val commonMap: UniformPdd[Map[String, List[Int]]] =
  encodingMap.broadcastTo(allOwners: _*)

val res = for (pets <- allPets) yield {
  pets.zip(commonMap).map { case (ps, encoding) =>
    ps.map(encoding)
  }
(commonMap, res) }
```

Outline - temp slide

- 00-03 Intro
- 03-13 Part I - What is Speakeasy?
- 13-23 Part II - What is a PDD?
- 23-27 Part III - What do we do with a PDD graph?
- 27-40 Part IV - Speakeasy meets Scala 3
- 40-45 Buffer