

MeshCI Workflows

defining and running workflows in Scala

Overview

0. Background

1. what, why, how
2. from direct-style Scala to first-class objects

Part 0: Background

About Me

- used Scala for a long time (intro was in 1.x days, when `Int` was `int`)
- independent developer
- building a software delivery platform, **MeshCI**

MeshCI

- problem statement: “releasing software built by multiple teams is a pain”
 - often, either teams work quickly (in isolation) and integration is rare
 - or integration is frequent and teams are slowed down
 - either way, a lot of time is lost and frustration created
- why does it exist?
 - very large orgs: bespoke solutions and dedicated teams
 - everyone else: CI and releases is no one’s job, not mission critical
- MeshCI is a “convenient and safe” off-the-shelf solution to address this problem
 - improved developer experience and increased release speed for organizations that build software
 - main idea: track dependencies between subprojects, and run tests only affecting dependents
 - be ready to release at all times

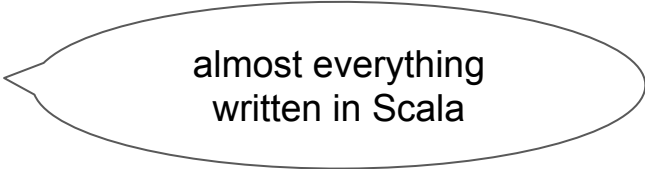
MeshCI: platform

Platform is built using three major subsystems

- dependency tracking
 - how projects depend on one another, works for monorepos, polyrepos and anything in between
- test gathering
 - analyze test results across the whole org
 - identify green builds for cutting releases
 - identify flaky tests
- workflows
 - run tests
 - **underpins most automation of MeshCI**
 - also usable for user-defined tasks, e.g. custom CI and releases

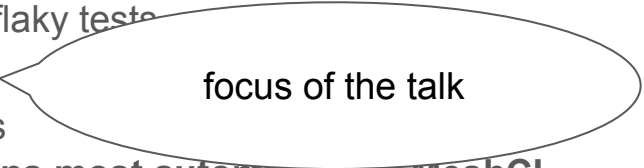
You can use the platform as a whole, or each subsystem independently

MeshCI: platform



almost everything
written in Scala

Platform is built using three major subsystems

- dependency tracking
 - how projects depend on one another, works for monorepos, polyrepos and anything in between
- test gathering
 - analyze test results across the whole org
 - identify green builds for cutting releases
 - identify flaky tests
- **workflows** 
 - run tests
 - **underpins most automation or MeshCI**
 - also usable for user-defined tasks, e.g. releases

You can use the platform as a whole, or each subsystem independently

Part 1: what, why, how

What is a workflow engine?

- Wikipedia: “a software application that manages business processes”
- in context: a system to run a bunch of software-development related tasks on certain events
- continuous integration, delivery and deployment (CI/CD)
- tasks of a workflow have dependencies => they form a directed acyclic graph (DAG)

Examples

- engine examples:
 - GitHub Actions, Jenkins, Apache Airflow, Argo Workflows, Spinnaker, many many more...
- workflow examples:
 - run your tests when a user creates a pull request
 - build and upload binaries of your software when someone pushes a tag to your repository
 - once a night, create a nightly release
 - when someone tags a new container image, push a canary deployment to production, wait for manual approval, and then scale it

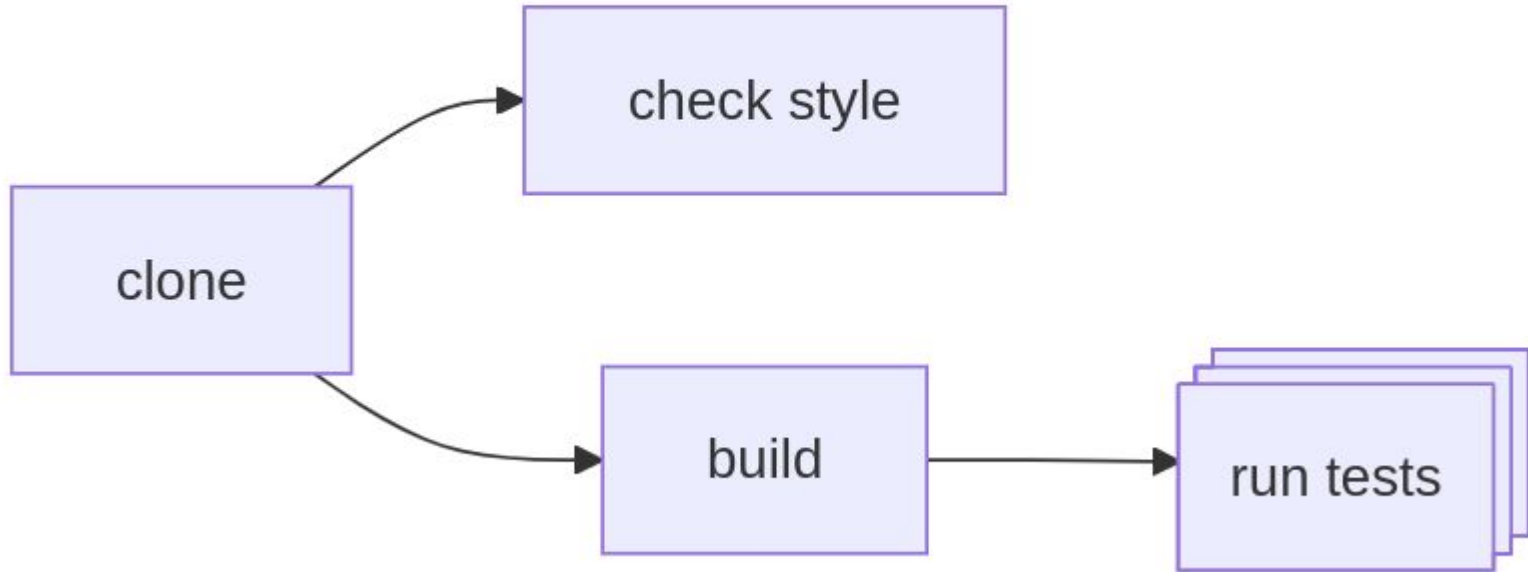
Examples, continued

on push



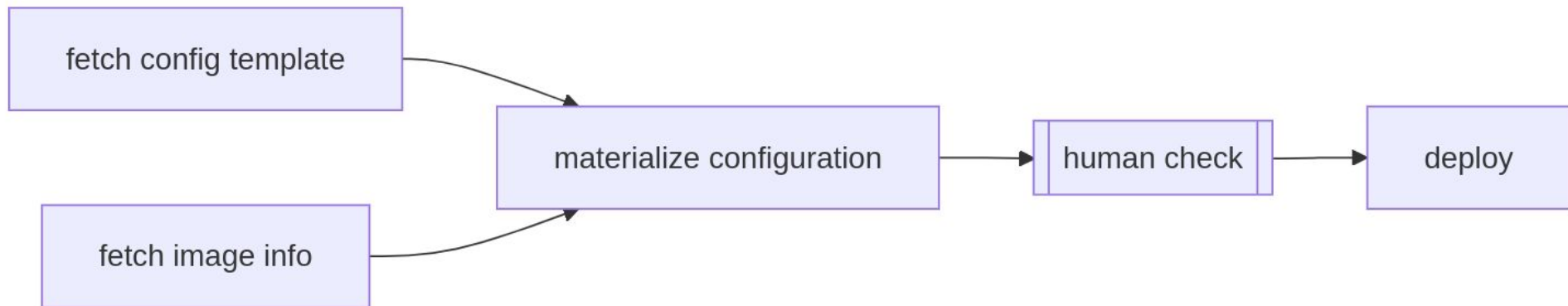
Examples, continued

on pull request



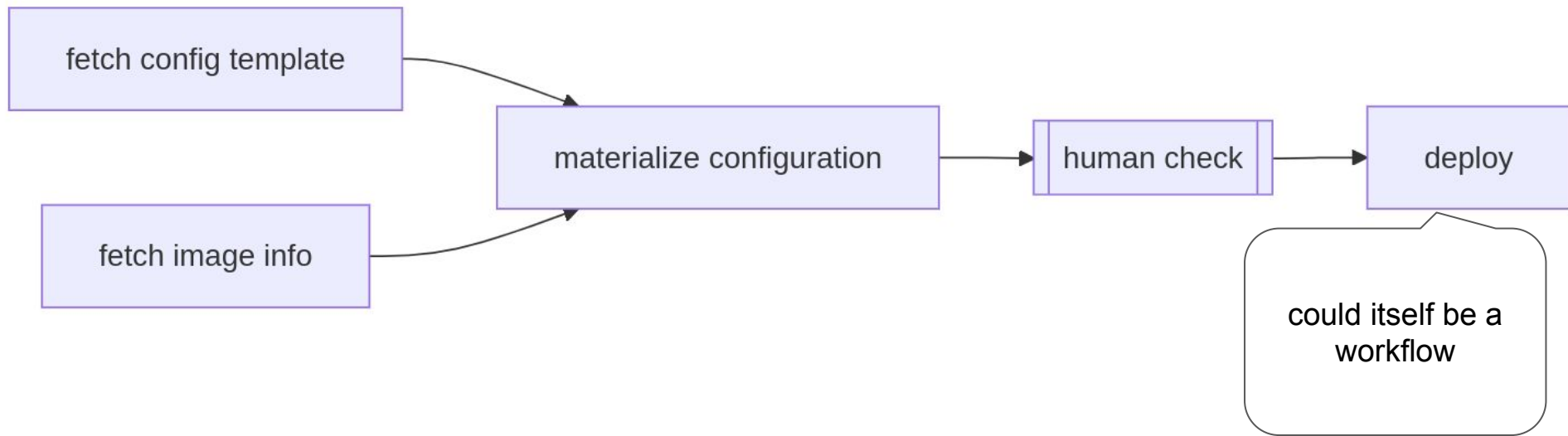
Examples, continued

on template change, or on new image



Examples, continued

on template change, or on new image



Why create a new workflow engine?

Wishlist:

- define tasks in direct-style Scala
 - NOT monadic or DSL-heavy
 - NOT in a UI
 - NOT bash-in-YAML 🤩
- locally runnable
 - most engines are either not locally runnable or require a lot of effort to install
 - imagine you could only debug in prod
- scalable and distributed
 - single machine up to whole cluster
- multiple hosting options
 - piggy-back onto GitHub actions VM, deployable as standalone agents, Kubernetes clusters

Demo

What have we done?

- transform direct style into task graph
- serialize task graph
- schedule and run task graph
- reporting, analysis, ...

What have we done?

- transform direct style into task graph
 - serialize task graph
 - schedule and run task graph
 - reporting, analysis, ...
- } leverage Scala and the JVM
for this
- } many ways to do this, not
really specific to Scala

What have we done?

- transform direct style into task graph
- serialize task graph
- schedule and run task graph
- reporting, analysis, triggering...

next part

Scala and the JVM

for this

many ways to do this, not
really specific to Scala

Technology stack:

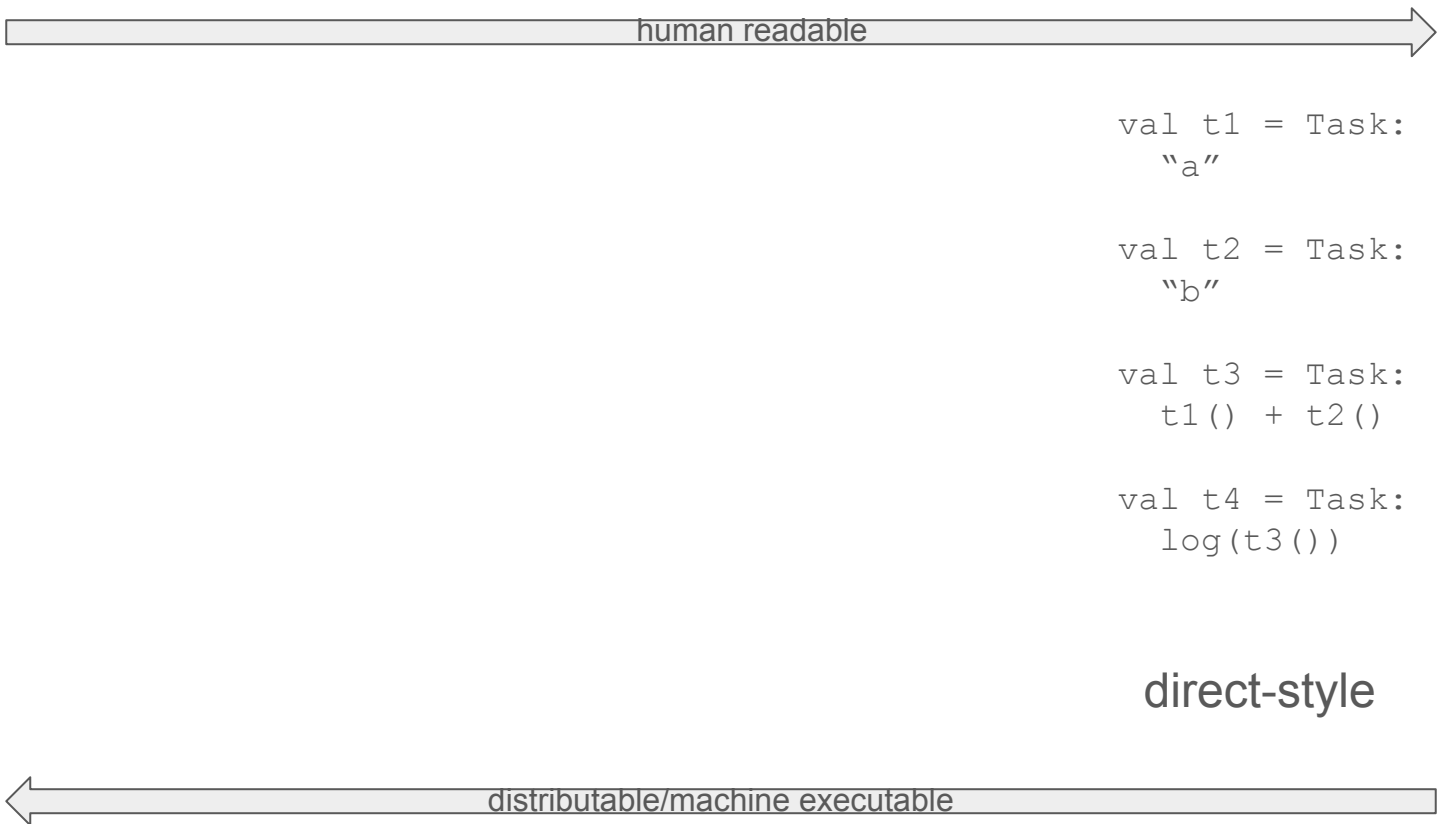
- Scala.js UI with ScalaTags
 - Cask
- ScalaPB, upickle
- Mill to build it all

Part 2: direct-style Scala to first-class objects

Direct-style not sufficient for workflows

- direct-style can be executed almost as-is
 - imagine no `Task`, only functions
- not good enough for workflows
 - need to manage tasks individually
- we need to extract
 - metadata
 - dependencies
 - body
- so that we can serialize, execute, etc

Syntax and structure



Syntax and structure

human readable



```
message TaskDAG {  
  message Node {  
    string name;  
    bytes run;  
    string source;  
  }  
  message Edge {  
    string from;  
    string to;  
  }  
  repeated Node nodes;  
  repeated Edge edges;  
}
```

protobuf

```
val t1 = Task:  
  "a"
```

```
val t2 = Task:  
  "b"
```

```
val t3 = Task:  
  t1() + t2()
```

```
val t4 = Task:  
  log(t3())
```

direct-style

distributable/machine executable



Syntax and structure

human readable

protobuf

```
message TaskDAG {  
  message Node {  
    string name;  
    bytes run;  
    string source;  
  }  
  message Edge {  
    string from;  
    string to;  
  }  
  repeated Node nodes;  
  repeated Edge edges;  
}
```

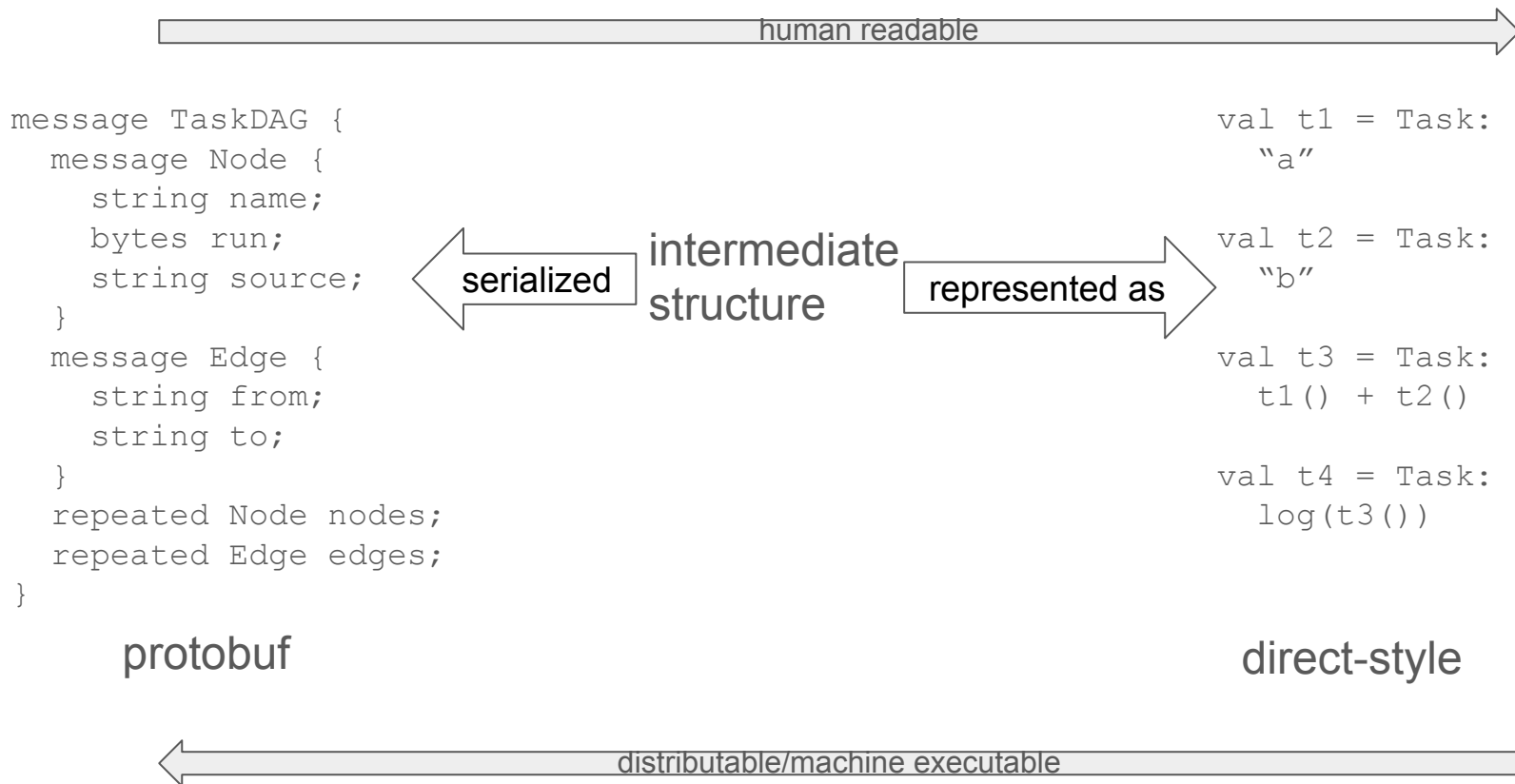
direct-style

```
val t1 = Task:  
  "a"  
  
val t2 = Task:  
  "b"  
  
val t3 = Task:  
  t1() + t2()  
  
val t4 = Task:  
  log(t3())
```

???

distributable/machine executable

Syntax and structure



Syntax and structure

human readable



```
message TaskDAG {  
  message Node {  
    string name;  
    bytes run;  
    string source;  
  }  
  message Edge {  
    string from;  
    string to;  
  }  
  repeated Node nodes;  
  repeated Edge edges;  
}
```

protobuf

```
case class Task[A](  
  name: String,  
  deps: Seq[Task[?]],  
  source: String,  
  run: Env => A  
)  
  
trait Env:  
  def res(t: Task[A]): A  
  def log(msg: Any): Unit
```

first-class objects

```
val t1 = Task:  
  "a"  
  
val t2 = Task:  
  "b"  
  
val t3 = Task:  
  t1() + t2()  
  
val t4 = Task:  
  log(t3())
```

direct-style

distributable/machine executable



Syntax and structure

human readable



```
message TaskDAG {  
  message Node {  
    string name;  
    bytes run;  
    string source;  
  }  
  message Edge {  
    string from;  
    string to;  
  }  
  repeated Node nodes;  
  repeated Edge edges;  
}
```

protobuf

```
case class Task[A](  
  name: String,  
  deps: Seq[Task[?]],  
  source: String,  
  run: Env => A  
)  
  
trait Env:  
  def res(t: Task[A]): A  
  def log(msg: Any): Unit
```

first-class objects

```
val t1 = Task:  
  "a"  
  
val t2 = Task:  
  "b"  
  
val t3 = Task:  
  t1() + t2()  
  
val t4 = Task:  
  log(t3())
```

direct-style

distributable/machine executable



Syntax and structure

human readable

```
message TaskDAG {  
  message Node {  
    string name;  
    bytes run;  
    string source;  
  }  
  message Edge {  
    string from;  
    string to;  
  }  
  repeated Node nodes;  
  repeated Edge edges;  
}
```

protobuf

```
case class Task[A](  
  name: String,  
  deps: Seq[Task[?]],  
  source: String,  
  run: Env => A  
)  
  
trait Env:  
  def res(t: Task[A]): A  
  def log(msg: Any): Unit
```

first-class objects

```
val t1 = Task:  
  "a"
```

```
val t2 = Task:  
  "b"
```

```
val t3 = Task:  
  t1() + t2()
```

```
val t4 = Task:  
  log(t3())
```

direct-style

distributable/machine executable

Syntax and structure

human readable



```
message TaskDAG {  
  message Node {  
    string name;  
    bytes run;  
    string source;  
  }  
  message Edge {  
    string from;  
    string to;  
  }  
  repeated Node nodes;  
  repeated Edge edges;  
}
```

protobuf

```
case class Task[A](  
  name: String,  
  deps: Seq[Task[?]],  
  source: String,  
  run: Env => A  
)  
  
trait Env:  
  def res(t: Task[A]): A  
  def log(msg: Any): Unit
```

first-class objects

```
val t1 = Task:  
  "a"  
  
val t2 = Task:  
  "b"  
  
val t3 = Task:  
  t1() + t2()  
  
val t4 = Task:  
  log(t3())
```

direct-style

distributable/machine executable



Syntax and structure

human readable



```
message TaskDAG {  
  message Node {  
    string name;  
    bytes run;  
    string source;  
  }  
  message Edge {  
    string from;  
    string to;  
  }  
  repeated Node nodes;  
  repeated Edge edges;  
}
```

protobuf

```
case class Task[A](  
  name: String,  
  deps: Seq[Task[?]],  
  source: String,  
  run: Env => A  
)  
  
trait Env:  
  def res(t: Task[A]): A  
  def log(msg: Any): Unit
```

first-class objects

```
val t1 = Task:  
  "a"  
  
val t2 = Task:  
  "b"  
  
val t3 = Task:  
  t1() + t2()  
  
val t4 = Task:  
  log(t3())
```

direct-style

distributable/machine executable



Syntax and structure

human readable

```
message TaskDAG {  
  message Node {  
    string name;  
    bytes run;  
    string source;  
  }  
  message Edge {  
    string from;  
    string to;  
  }  
  repeated Node nodes;  
  repeated Edge edges;  
}
```

protobuf

```
case class Task[A](  
  name: String,  
  deps: Seq[Task[?]],  
  source: String,  
  run: Env => A  
)
```

serialize Task[A] to protobuf

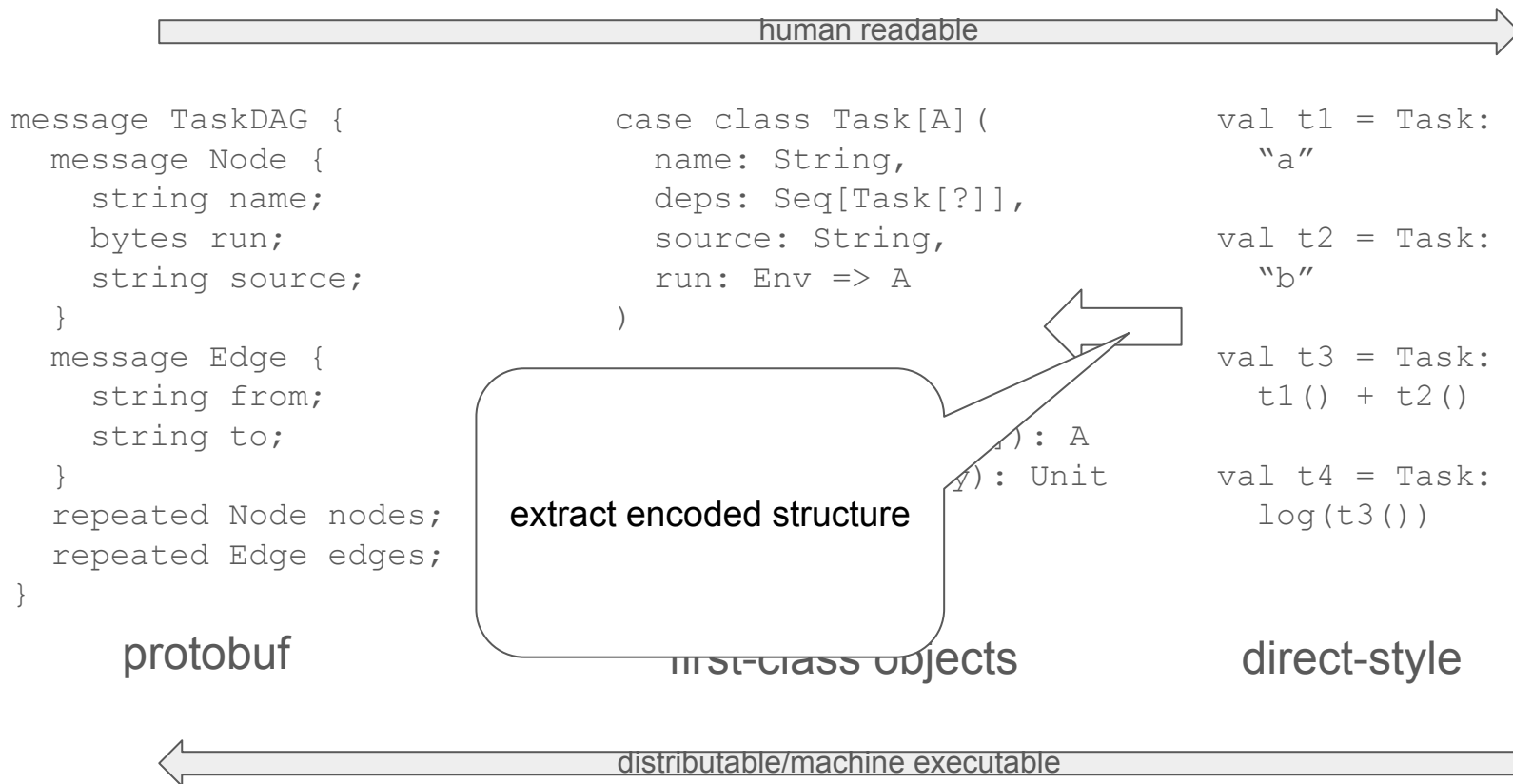
- most brittle is lambda
- see how Apache Spark does it
- Spores? see Jonas' and Philipp's talk

```
val t1 = Task:  
  "a"  
  
val t2 = Task:  
  "b"  
  
val t3 = Task:  
  t1() + t2()  
  
val t4 = Task:  
  log(t3())
```

direct-style

distributable/machine executable

Syntax and structure



Kinds of structure

- syntax sugar

- where is “Env”?
- **context functions**

```
val t1 = Task:  
  "a"
```

```
val t2 = Task:  
  "b"
```

- static structure

- actual code
- **metaprogramming (macros)**

```
val t3 = Task:  
  t1() + t2()
```

```
val t4 = Task:  
  log(t3())
```

- see related talk on the subject: <https://jakob.odersky.com/talks/zero-to-three/>

Context functions

```
def withEnv1(run: Env => Unit) = ...
```

```
withEnv1: env =>  
  log("hello, world") (using env)
```

```
def withEnv2(run: Env ?=> Unit) = ...
```

```
withEnv2:  
  log("hello, world")
```

allow declaring functions which take an implicit parameter

Macros

```
import scala.quoted.Quotes
import scala.quoted.Expr

def show1(x: Int): String = x.toString
inline def show2(inline x: Int): String = ${show2Impl('x)}

def show2Impl(using Quotes)(x: Expr[Int]): Expr[String] =
  Expr(x.show)

val x = 1
println(show1(x + 1)) // 2
println(show2(x + 1)) // x.(1)
```

allow inspecting and manipulating the Scala AST itself

Macros

```
import scala.quoted.Quotes
import scala.quoted.Expr

def show1(x: Int): String = x.toString
inline def show2(inline x: Int): String = ${show2Impl('x)}

def show2Impl(using Quotes)(x: Expr[Int]): Expr[String] =
  Expr(x.show)

val x = 1
println(show1(x + 1)) // 2
println(show2(x + 1)) // x.+(1)
```

Expr[T] allows inspecting the structure of T at compile time

returned Expr is spliced back into the call-site
=> it will become the body of whatever apply is assigned to

allow inspecting and manipulating the Scala AST itself

Combining macros and ctx functions

```
val t1: Task:  
  "hello"
```

```
val t2 = Task:  
  val msg = t1() + ", world"  
  log(msg)  
  msg
```



```
val t1 = ...
```

```
val t2 = Task[String](  
  name = "t2",  
  deps = List(t1),  
  source = "...",  
  run = (env: Env) =>  
    val msg = env.get(t1) + ", world"  
    log(msg) (using env)  
    msg  
)
```

Combining macros and ctx functions

```
object Task:
  inline def apply[A](inline run: Env => A): Task[A] = ${TaskMacros.applyImpl('run')}

def log(message: Any)(using env: Env): Unit = env.log(message)
```

```
case class Task[A] (
  name: String,
  deps: Seq[Task[?]],
  source: String,
  run: Env => A
)
```

Macro implementation

```
inline def apply[A](inline run: Env => A): Task[A] = ${applyImpl('run)}

def applyImpl[A: Type](using Quotes)(run: Expr[Env => A]): Expr[Task[A]] =
  import scala.quoted.quotes.reflect.*

  val name: String = Symbol.spliceOwner.owner.name
  val dependencies: Seq[Expr[Task[?]]] = findDependencies(run)
  val code: String = Symbol.spliceOwner.owner.tree.pos.sourceCode.get

  '{
    Task[A] (
      ${Expr(name)},
      ${Expr.ofSeq(dependencies)},
      ${Expr(code)},
      (env: Env) => $run(using env),
    )
  }
```

Tree traversal

```
def findDependencies[A: Type](using Quotes)(tree: Expr[A]): (Seq[Expr[Task[?]]], Expr[A]) =
  import scala.quoted.quotes.reflect.*

  val deps = collection.mutable.ListBuffer.empty[Expr[Task[?]]]

  class Transformer(ctx: Expr[Env]) extends TreeMap:
    override def transformTerm(term: Term)(owner: Symbol): Term =
      term match
        case t@Apply(Select(task, "apply"), _) if task.tpe <:< TypeRepr.of[Task[?]] =>
          t.tpe.asType match
            case '[tt] =>
              deps += task.asExprOf[Task[tt]]
              '{$ctx.args($ {Expr(deps.size - 1)})}.asInstanceOf[tt]}.asTerm
        case _ =>
          super.transformTerm(term)(owner)

  ...
```


Going further

- Find all tasks in the workflow
 - typeclass derivation
 - TASTy inspection
 - runtime initialization

Takeaways

- macros and context functions allow to write first-class objects in a direct-style
 - allows us to separate code and structure
- first-class objects are then used for the rest of the system
 - serialization
 - analysis
 - execution
 - etc
- approach which allows us to have a **direct surface API**, but a **robust structure** for everything below it

Wrapping up

How can I use it?

- it started as and is still an internal tool of MeshCI
- planning to make the runner and UI open source
- needs cleanup, disentangle from monorepo
- future work:
 - integration with platforms (e.g. GitHub)
 - globally hosted UI
 - advanced triggers, prompting user input, custom signals
 - more built-in tasks (e.g. cloning, secrets)

Follow this repo <https://github.com/meshci/workflows> for updates

Isn't this a build tool?

- “isn't everything [with a DAG] a build tool?” – Guillaume Martres
- see: [Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. 2018. Build Systems à la Carte.](#)
 - Excel is also a build tool!
- many modern workflow engines/CI engines/build tools are converging
- different tradeoffs in caching, retrying, triggering and interaction with environment
- the syntax and structure is inspired by the [Mill build tool](#), from Li Haoyi

Merci beaucoup!