CiBO TECHNOLOGIES

ScalaStan

Joe Wingbermuehle

Scala

"Object-Oriented Meets Functional"

A hybrid object-oriented and functional programming language with a strong static type system.

Runs on the Java Virtual Machine (JVM).

2017 RedMonk Ranking:

1 JavaScript
2 Java
3 Python
4 PHP
5 C#
6 C++
7 CSS
8 Ruby
9 C
10 Objective-C
11 Swift
12 Shell
12 Scala
14 R
15 Go
15 Perl

```scala
object HelloWorld extends App {
  println("Hello, World!")
}
```

https://scala-lang.org

Goals of ScalaStan

- **Make it easy and type-safe to get data into Stan.**

Goals of ScalaStan

- Make it easy and type-safe to get data into Stan.
- **Make it easy and type-safe to get results out of Stan.**

Goals of ScalaStan

- Make it easy and type-safe to get data into Stan.
- Make it easy and type-safe to get results out of Stan.
- **Make running Stan simple.**

+

Goals of ScalaStan

- Make it easy and type-safe to get data into Stan.
- Make it easy and type-safe to get results out of Stan.
- Make running Stan simple.
- **Make developing Stan models easier:**
    - **Type safety**

Goals of ScalaStan

- Make it easy and type-safe to get data into Stan.
- Make it easy and type-safe to get results out of Stan.
- Make running Stan simple.
- **Make developing Stan models easier:**
    - Type safety
    - **Model and result caching**

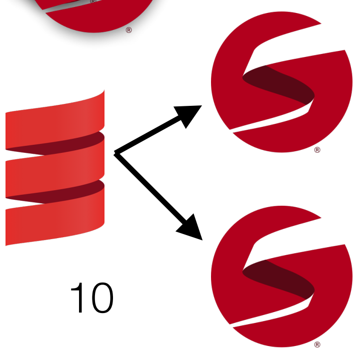
Goals of ScalaStan

- Make it easy and type-safe to get data into Stan.
- Make it easy and type-safe to get results out of Stan.
- Make running Stan simple.
- **Make developing Stan models easier:**
    - Type safety
    - Model and result caching 
    - **Discoverability of Stan functions and distributions** 

Goals of ScalaStan

- Make it easy and type-safe to get data into Stan.
- Make it easy and type-safe to get results out of Stan.
- Make running Stan simple.
- **Make developing Stan models easier:**
  - Type safety
  - Model and result caching 
  - Discoverability of Stan functions and distributions 
  - **Make Stan models composable**

Goals of ScalaStan

- Make it easy and type-safe to get data into Stan.
- Make it easy and type-safe to get results out of Stan.
- Make running Stan simple.
- **Make developing Stan models easier:**
    - Type safety
    - Model and result caching 
    - Discoverability of Stan functions and distributions
    - Make Stan models composable
    - **Make it possible to generate Stan** 

What Is ScalaStan?

- A Stan wrapper in Scala

Like RStan and PyStan

- An embedded domain-specific language (DSL) for Stan
  - Type-checked API in Scala for generating Stan

```scala
val model = new Model {
  sigma ~ stan.cauchy(0, 1)
  y ~ stan.normal(m * x + b, sigma)
}
```

→

```stan
model {
  sigma ~ cauchy(0, 1);
  y ~ normal(m * x + b, sigma);
}
```

Linear Regression Example

Stan Code:

```
data {
  int<lower=0> N;
  vector[N] x;
  vector[N] y;
}
parameters {
  real alpha;
  real beta;
  real<lower=0> sigma;
}
model {
  y ~ normal(alpha + beta * x, sigma);
}
```

Simple linear regression:

$$y_n \sim \text{Normal}(\alpha + \beta X_n, \sigma)$$

Models in ScalaStan

Stan:

```
data {
  int<lower=0> N;
  vector[N] x;
  vector[N] y;
}
```

ScalaStan:

```
val N = data(int(lower = 0))
val y = data(vector(N))
val x = data(vector(N))
```


CiBO TECHNOLOGIES

Models in ScalaStan

Stan:

```
data {
  int<lower=0> N;
  vector[N] x;
  vector[N] y;
}
parameters {
  real alpha;
  real beta;
  real<lower=0> sigma;
}
```

ScalaStan:

```
val N = data(int(lower = 0))
val y = data(vector(N))
val x = data(vector(N))

val alpha = parameter(real())
val beta = parameter(real())
val sigma = parameter(real(lower = 0))
```

CiBO TECHNOLOGIES

Models in ScalaStan

Stan:

```
data {
  int<lower=0> N;
  vector[N] x;
  vector[N] y;
}
parameters {
  real alpha;
  real beta;
  real<lower=0> sigma;
}
model {
  y ~ normal(alpha + beta * x, sigma);
}
```

ScalaStan:

```
val N = data(int(lower = 0))
val y = data(vector(N))
val x = data(vector(N))

val alpha = parameter(real())
val beta = parameter(real())
val sigma = parameter(real(lower = 0))
```

```
val model = new Model {
  y ~ stan.normal(alpha + beta * x, sigma)
}
```

`Model` is a trait in Scala providing the DSL for models.

Goal: Keep the low-level ScalaStan API as similar to Stan as possible.

CiBO TECHNOLOGIES

Scala IDE Support

Types are checked in Scala

```
val model = new Model {
  y ~ stan.normal(alpha + beta * x, sigma) + sigma
}
```

Scala IDE Support

Types are checked in Scala

```
val model = new Model {
  y ~ stan.normal(alpha + beta * x, sigma) + sigma
}
```

IDE provides discoverability

```
val model = new Model {
  y ~ stan.nor
}
  normal[A <: StanType : C…        StanContinuousDistribution[R, StanReal]
  exp_mod_normal[M <: Stan…        StanContinuousDistribution[R, StanReal]
  multi_normal[M <: Stan…        StanContinuousDistribution[R, StanMatrix]
  multi_normal_cholesky[…        StanContinuousDistribution[R, StanMatrix]
  multi_normal_precision…        StanContinuousDistribution[R, StanMatrix]
  skew_normal[X <: StanTyp…        StanContinuousDistribution[R, StanReal]
  lognormal[M <: StanType …        StanContinuousDistribution[R, StanReal]
Press ^. to choose the selected (or first) suggestion and insert a dot afterwards >>
```

Scala IDE Support

Types are checked in Scala

```scala
val model = new Model {
  y ~ stan.normal(alpha + beta * x, sigma) + sigma
}
```

IDE provides discoverability

```scala
val model = new Model {
  y ~ stan.nor
}
```

```
f normal[A <: StanType : C…        StanContinuousDistribution[R, StanReal]
f exp_mod_normal[M <: Stan…        StanContinuousDistribution[R, StanReal]
f multi_normal[M <: Stan…          StanContinuousDistribution[R, StanMatrix]
f multi_normal_cholesky[…          StanContinuousDistribution[R, StanMatrix]
f multi_normal_precision…          StanContinuousDistribution[R, StanMatrix]
f skew_normal[X <: StanTyp…        StanContinuousDistribution[R, StanReal]
f lognormal[M <: StanType …        StanContinuousDistribution[R, StanReal]
Press ^. to choose the selected (or first) suggestion and insert a dot afterwards >>  π
```

```scala
mu: StanValue[A], sigma: StanValue[B]
val model = new Model {
  y ~ stan.normal(alpha + beta * x, sigma)
}
```

Getting Data Into ScalaStan

Required data values:

```
val N = data(int(lower = 0))
val y = data(vector(N))
val x = data(vector(N))
```


CiBO TECHNOLOGIES

Getting Data Into ScalaStan

Required data values:

```
val N = data(int(lower = 0))
val y = data(vector(N))
val x = data(vector(N))
```

From Scala directly:

```
val xs = Vector(1.0, 3.0, 4.0, 5.0)
val ys = Vector(2.5, 5.1, 6.3, 7.0)
val results = model
  .withData(x, xs)
  .withData(y, ys)
```

```
final def withData[T <: StanType, V](
  decl: StanDataDeclaration[T],
  data: V
)(implicit ev: V <:< T#SCALA_TYPE): CompiledModel
```


CiBO TECHNOLOGIES

Getting Data Into ScalaStan

Required data values:

```
val N = data(int(lower = 0))
val y = data(vector(N))
val x = data(vector(N))
```

From Scala directly:

```
val xs = Vector(1.0, 3.0, 4.0, 5.0)
val ys = Vector(2.5, 5.1, 6.3, 7.0)
val results = model
  .withData(x, xs)
  .withData(y, ys)
```

```
final def withData[T <: StanType, V](
  decl: StanDataDeclaration[T],
  data: V
)(implicit ev: V <:< T#SCALA_TYPE): CompiledModel
```

From CSV:

```
val source = CsvDataSource.fromFile("data.csv")
val results = model
  .withData(source(x, "x"))
  .withData(source(y, "y"))
```

Getting Data Into ScalaStan

Required data values:

```
val N = data(int(lower = 0))
val y = data(vector(N))
val x = data(vector(N))
```

From Scala directly:

```
val xs = Vector(1.0, 3.0, 4.0, 5.0)
val ys = Vector(2.5, 5.1, 6.3, 7.0)
val results = model
  .withData(x, xs)
  .withData(y, ys)
```

```
final def withData[T <: StanType, V](
  decl: StanDataDeclaration[T],
  data: V
)(implicit ev: V <:< T#SCALA_TYPE): CompiledModel
```

From CSV:

```
val source = CsvDataSource.fromFile("data.csv")
val results = model
  .withData(source(x, "x"))
  .withData(source(y, "y"))
```

From R:

```
val rsource = RDataSource.fromFile("data.R")
```


CiBO TECHNOLOGIES

Getting Data Into ScalaStan

Required data values:

```
val N = data(int(lower = 0))
val y = data(vector(N))
val x = data(vector(N))
```

From Scala directly:

```
val xs = Vector(1.0, 3.0, 4.0, 5.0)
val ys = Vector(2.5, 5.1, 6.3, 7.0)
val results = model
  .withData(x, xs)
  .withData(y, ys)
```

```
final def withData[T <: StanType, V](
  decl: StanDataDeclaration[T],
  data: V
)(implicit ev: V <:< T#SCALA_TYPE): CompiledModel
```

From CSV:

```
val source = CsvDataSource.fromFile("data.csv")
val results = model
  .withData(source(x, "x"))
  .withData(source(y, "y"))
```

From R:

```
val rsource = RDataSource.fromFile("data.R")
```

- Scala ensures the types line up
- The value for N is set automatically

CiBO TECHNOLOGIES

Running ScalaStan Models

```
val results = model
  .withData(x, xs)
  .withData(y, ys)
  .run()
```


CiBO TECHNOLOGIES

Running ScalaStan Models

```scala
val results = model
  .withData(x, xs)
  .withData(y, ys)
  .run()
```

Multiple chains
(to get parallelism and convergence testing):

```scala
.run(chains = 4)
```

Running ScalaStan Models

```
val results = model
  .withData(x, xs)
  .withData(y, ys)
  .run()
```

Multiple chains (to get parallelism and convergence testing):

```
.run(chains = 4)
```

```
final def run(
  chains: Int = 1,
  seed: Int = -1,
  cache: Boolean = true,
  method: RunMethod.Method = RunMethod.Sample()
): StanResults = {
```

Other options:

- **Fix seed**
- **Result caching**
- **Sampler/optimizer parameters**

Getting Results Out of ScalaStan

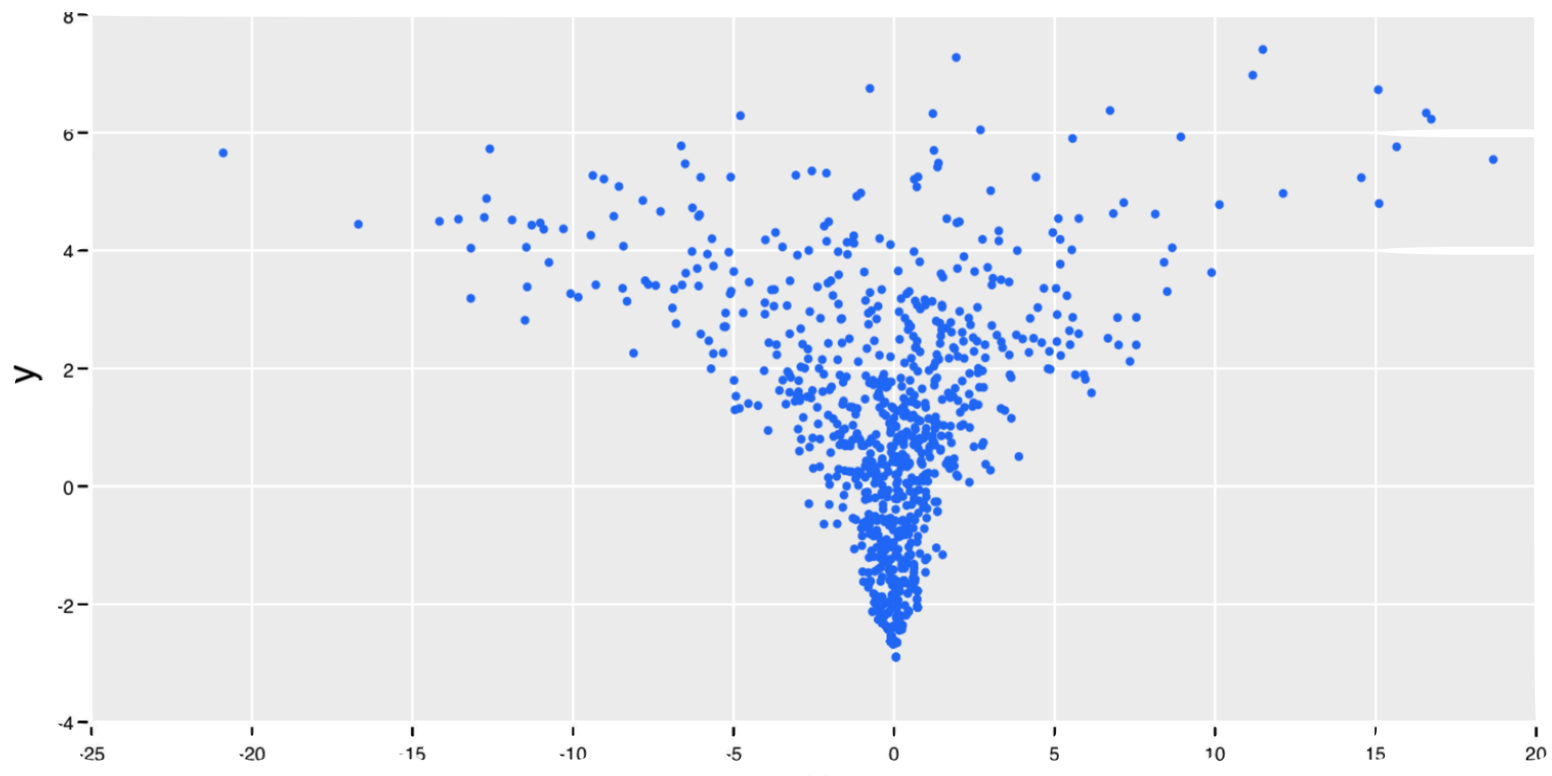
```scala
def mean[T <: StanType](decl: StanParameterDeclaration[T]): T#SUMMARY_TYPE

def sd[T <: StanType](decl: StanParameterDeclaration[T]): T#SUMMARY_TYPE
```

```scala
results.summary(System.out)
println(s"alpha = ${results.mean(alpha)}")
println(s"beta = ${results.mean(beta)}")
```

```
4 chains with 1000 iterations each, 4000 total

Name              Mean     MCSE   StdDev        5%      50%      95%   N_Eff     R_hat
accept_stat__   0.7195  0.03290   0.3413  0.005329   0.8987   0.9999   107.6     1.008
alpha            2.367   0.7259    8.851    -1.166    1.546    5.351   148.7     1.010
beta            0.8654   0.2706    2.748  -0.02792    1.134    1.933   103.1     1.013
divergent__    0.05925 0.008916   0.2361     0.000    0.000    1.000   701.4     1.001
energy__         2.495   0.6340    3.228    -1.479    1.851    8.476   25.93     1.081
lp__           -0.9925   0.7085    2.978    -6.615  -0.2887    2.454   17.67     1.091
n_leapfrog__     18.03    1.661    31.38     3.000    15.00    47.00   357.2     1.001
sigma            2.356    1.016    9.548    0.2312   0.7624    5.971   88.34     1.016
stepsize__     0.07296  0.03059  0.04327   0.04306  0.05132   0.1477   2.001  Infinity
treedepth__      3.215  0.09472    1.259     1.000    3.000    5.000   176.8     1.010

alpha = 2.367276227341493
beta = 0.865363928450001
```

Neal's Funnel

```
val x = parameter(real())
val y = parameter(real())

val model = new Model {
  y ~ stan.normal(0, 3)
  x ~ stan.normal(0, stan.exp(y / 2))
}
```

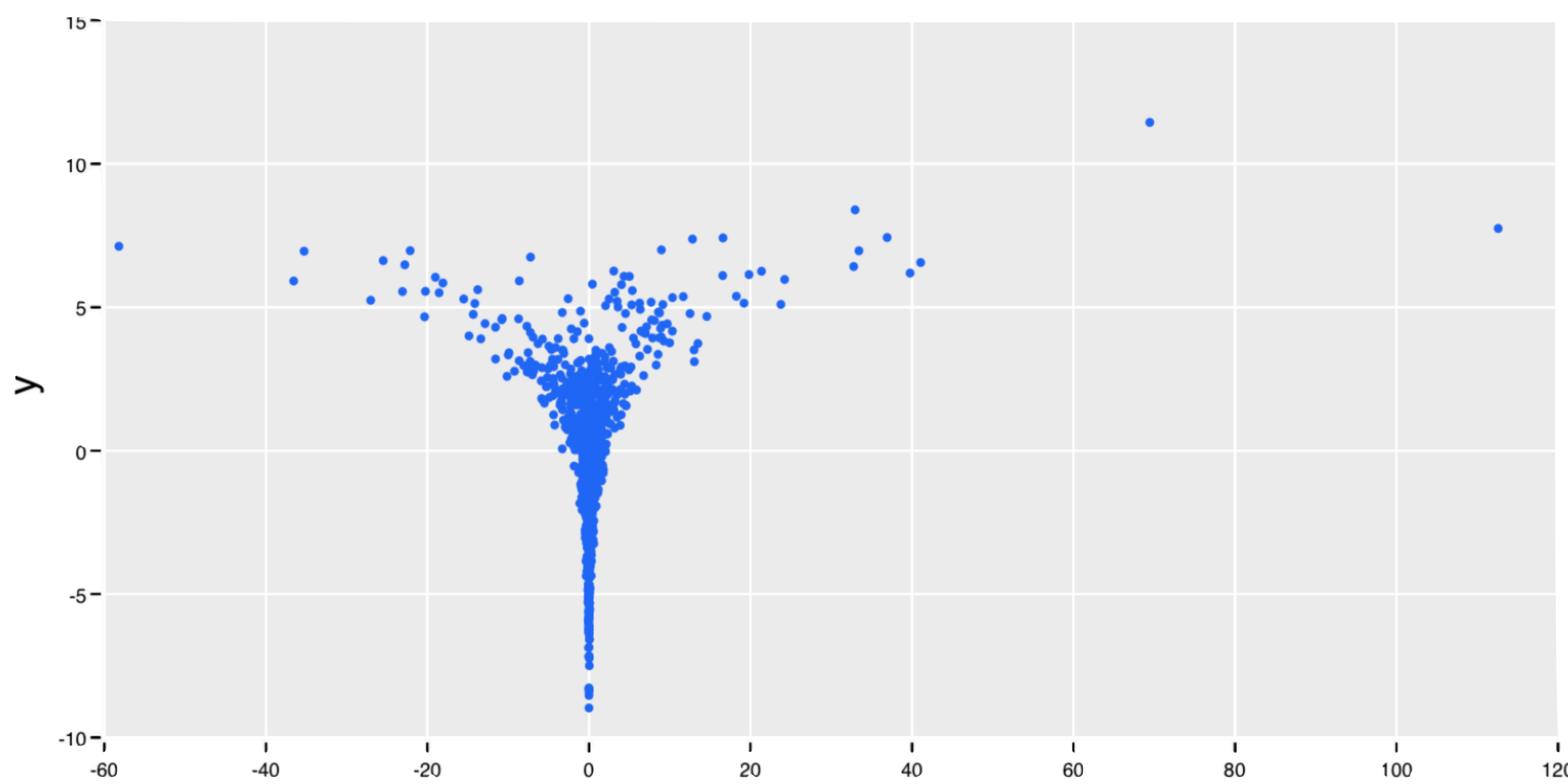


72 of 1000 iterations ended up with a divergence

Non-Centered Parameterization

```
val rawx = parameter(real())
val rawy = parameter(real())

val y = new TransformedParameter(real()) {
  result := 3 * rawy
}

val x = new TransformedParameter(real()) {
  result := stan.exp(y / 2) * rawx
}

val model = new Model {
  y ~ stan.normal(0, 1)
  x ~ stan.normal(0, 1)
}
```

No divergences!



CiBO TECHNOLOGIES

Non-centered Parameterization

```scala
trait TransformedNormalSampler extends Model {
  def sample(
    value: ParameterDeclaration[StanReal],
    mu: StanValue[StanReal],
    sigma: StanValue[StanReal]
  ): ParameterDeclaration[StanReal] = {

    value ~ stan.normal(0, 1)

    new TransformedParameter(real()) {
      result := mu + sigma * value
    }
  }
}
```

```scala
val model = new Model with TransformedNormalSampler {
  val y = sample(rawy, 0.0, 3.0)
  val x = sample(rawx, 0.0, stan.exp(y / 2))
}
```

CiBO TECHNOLOGIES

Composing Models using ScalaStan

Multiple Linear Regression using Least Squares:

Put the model in a trait and implement it with a case class:

```
trait LeastSquaresLike extends Model {
  val x: DataDeclaration[StanMatrix]
  val y: DataDeclaration[StanVector]
  val beta: ParameterDeclaration[StanVector]

  target += -stan.dotSelf(y - x * beta)
}
```

Subtract the squared error from target

```
case class LeastSquaresRegression(
  x: DataDeclaration[StanMatrix],
  y: DataDeclaration[StanVector],
  beta: ParameterDeclaration[StanVector]
) extends LeastSquaresLike
```

Composing Models using ScalaStan

Multiple Linear Regression using Least Squares:

Put the model in a trait and implement it with a case class:

```
trait LeastSquaresLike extends Model {
  val x: DataDeclaration[StanMatrix]
  val y: DataDeclaration[StanVector]
  val beta: ParameterDeclaration[StanVector]

  target += -stan.dotSelf(y - x * beta)
}
```

```
case class LeastSquaresRegression(
  x: DataDeclaration[StanMatrix],
  y: DataDeclaration[StanVector],
  beta: ParameterDeclaration[StanVector]
) extends LeastSquaresLike
```

Subtract the squared error from target

Now we can create the parameters/data values and instantiate the case class:

```
val n = data(int(lower = 0))
val p = data(int(lower = 0))
val y = data(vector(n))
val x = data(matrix(n, p))

val beta = parameter(vector(p))

val model = LeastSquaresRegression(x, y, beta)
```

CiBO TECHNOLOGIES

Composing Models using ScalaStan

Multiple Linear Regression with the Ridge Penalty:

Implement the ridge penalty in a trait by updating the log probability (target):

```
trait RidgePenaltyLike extends Model {
  val ridgeLambda: DataDeclaration[StanReal]
  val beta: ParameterDeclaration[StanVector]

  target += -ridgeLambda * stan.dotSelf(beta)
}
```

Minimize the Euclidean length of the coefficients

```
case class RidgeRegression(
  x: DataDeclaration[StanMatrix],
  y: DataDeclaration[StanVector],
  beta: ParameterDeclaration[StanVector],
  ridgeLambda: DataDeclaration[StanReal]
) extends LeastSquaresLike with RidgePenaltyLike
```

Linear regression trait from before

CiBO TECHNOLOGIES

Composing Models using ScalaStan

Multiple Linear Regression with the Lasso penalty:

```
trait LassoPenaltyLike extends Model {
  val lassoLambda: DataDeclaration[StanReal]
  val beta: ParameterDeclaration[StanVector]

  for (i ← beta.range) {
    target += -lassoLambda * stan.fabs(beta(i))
  }
}
```

Minimize the sum of the absolute values of the coefficients

```
case class LassoRegression(
  x: DataDeclaration[StanMatrix],
  y: DataDeclaration[StanVector],
  beta: ParameterDeclaration[StanVector],
  lassoLambda: DataDeclaration[StanReal]
) extends LeastSquaresLike with LassoPenaltyLike
```

Composing Models using ScalaStan

Multiple Linear Regression with the Elastic Net:

Both penalties

```
case class ElasticNetRegression(
  x: DataDeclaration[StanMatrix],
  y: DataDeclaration[StanVector],
  beta: ParameterDeclaration[StanVector],
  ridgeLambda: DataDeclaration[StanReal],
  lassoLambda: DataDeclaration[StanReal]
) extends LeastSquaresLike with RidgePenaltyLike with LassoPenaltyLike {
  val betaElasticNet = new GeneratedQuantity(vector(stan.cols(x))) {
    result := (1 + ridgeLambda) * beta
  }
}
```

```
val model = ElasticNetRegression(x, y, beta, lambda1, lambda2)
```


CiBO TECHNOLOGIES

Other Features

Locals and Assignment:

```
val x = local(real())
x := 10
```

Other Features

Locals and Assignment:

```
val x = local(real())
x := 10
```

Conditionals:

```
when(x > 10) {
  // ...
} otherwise {
  // ...
}
```

CiBO TECHNOLOGIES

Other Features

Locals and Assignment:

```
val x = local(real())
x := 10
```

Conditionals:

```
when(x > 10) {
  // ...
} otherwise {
  // ...
}
```

"while" loops:

```
loop(x < 10) {
  // ...
}
```

Other Features

Locals and Assignment:

```
val x = local(real())
x := 10
```

Parameter (and Data) Transforms:

```
val beta2 = new TransformedParameter(real()) {
  result := beta * 2
}
```

Conditionals:

```
when(x > 10) {
  // ...
} otherwise {
  // ...
}
```

"while" loops:

```
loop(x < 10) {
  // ...
}
```

Other Features

Locals and Assignment:

```
val x = local(real())
x := 10
```

Parameter (and Data) Transforms:

```
val beta2 = new TransformedParameter(real()) {
  result := beta * 2
}
```

Conditionals:

```
when(x > 10) {
  // ...
} otherwise {
  // ...
}
```

Functions:

```
val func = new Function(real()) {
  val x = input(real())
  output(x + 1)
}
```

Return type

Return value

Parameter declaration

"while" loops:

```
loop(x < 10) {
  // ...
}
```

Model/Result Caching

- Avoid re-compiling/building the model when the model doesn't change
- Avoid re-running the model when the model and data don't change
  - Make post-processing changes faster to test.

Stan Model → **SHA1 Hash** → **Compiled Model**

```
val model = new Model {
  y ~ stan.normal(alpha + beta * x, sigma)
}
```

ff9e066b63dd6856a58fef9a8e5667358060e34c


```
00003cc0  74 0c 48 8b bd 30 fa ff  ff e8 2c c6 0b 00 f6 85  |t.H..0....,.....|
00003cd0  b0 fa ff ff 01 74 0c 48  8b bd c0 fa ff ff e8 17  |.....t.H........|
00003ce0  c6 0b 00 48 c7 85 c0 fa  ff ff 00 00 00 00 48 c7  |...H..........H.|
00003cf0  85 b8 fa ff ff 00 00 00  00 c6 85 b0 fa ff ff 0c  |................|
00003d00  66 c7 85 b5 fa ff ff 6f  64 c7 85 b1 fa ff ff 6d  |f......od......m|
00003d10  65 74 68 c6 85 b7 fa ff  ff 00 48 8d bd 40 f9 ff  |eth.......H..@..|
00003d20  ff 48 8d b5 b0 fa ff ff  e8 23 3f 02 00 48 8b 08  |.H.......#?..H..|
00003d30  48 8b 49 38 66 0f 57 c0  66 0f 29 85 20 fa ff ff  |H.I8f.W.f.). ...|
```


Data

```
val xs = Seq(1.0, 3.0, 4.0, 5.0)
val ys = Seq(2.5, 5.1, 6.3, 7.0)
```

Configuration → **SHA1 Hash** → **Results**

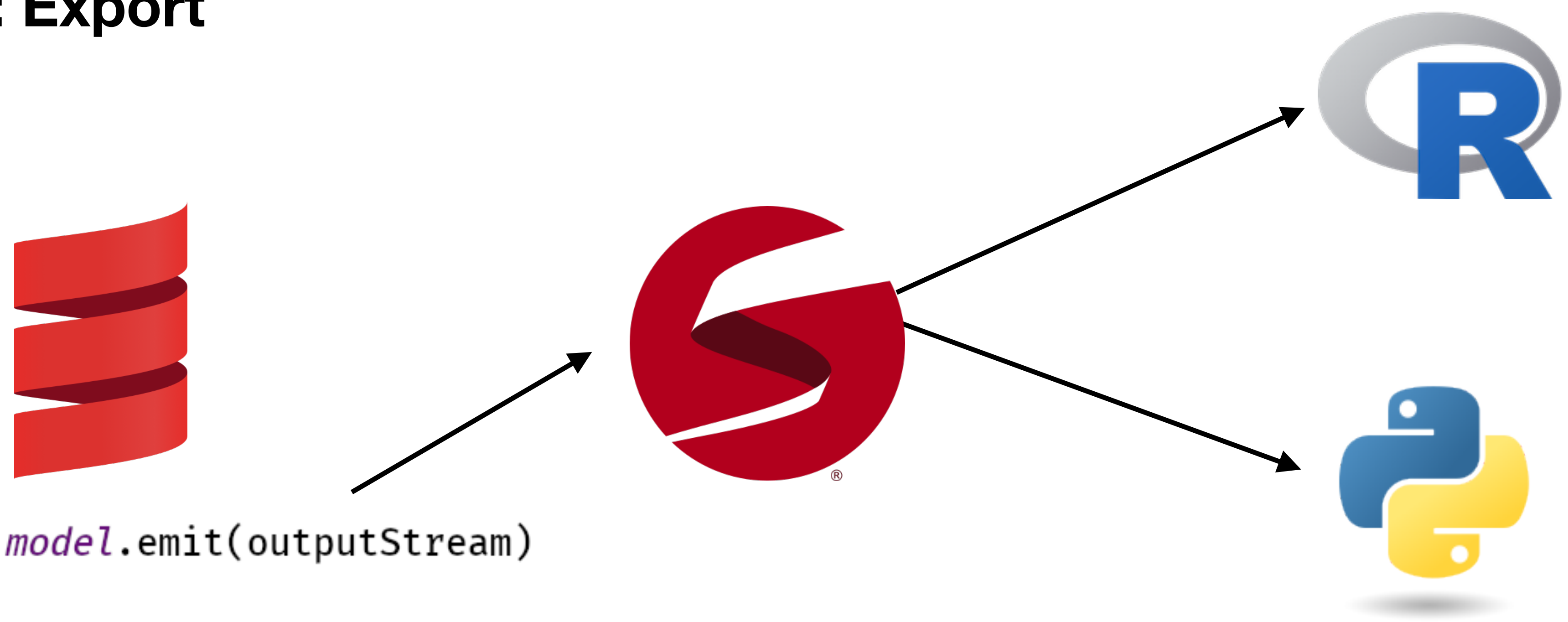
```
.run()
```

915384acba76e5e0cdbd88f280d1342b21d8cef9

Name	Mean	MCSE	StdDev	5%	50%	95%	N_Eff	R_hat
accept_stat__	0.7195	0.03290	0.3413	0.005329	0.8987	0.9999	107.6	1.008
alpha	2.367	0.7259	8.851	-1.166	1.546	5.351	148.7	1.010
beta	0.8654	0.2706	2.748	-0.02792	1.134	1.933	103.1	1.013
divergent__	0.05925	0.008916	0.2361	0.000	0.000	1.000	701.4	1.001
energy__	2.495	0.6340	3.228	-1.479	1.851	8.476	25.93	1.081
lp__	-0.9925	0.7085	2.978	-6.615	-0.2887	2.454	17.67	1.091
n_leapfrog__	18.03	1.661	31.38	3.000	15.00	47.00	357.2	1.001
sigma	2.356	1.016	9.548	0.2312	0.7624	5.971	88.34	1.016
stepsize__	0.07296	0.03059	0.04327	0.04306	0.05132	0.1477	2.001	Infinity
treedepth__	3.215	0.09472	1.259	1.000	3.000	5.000	176.8	1.010

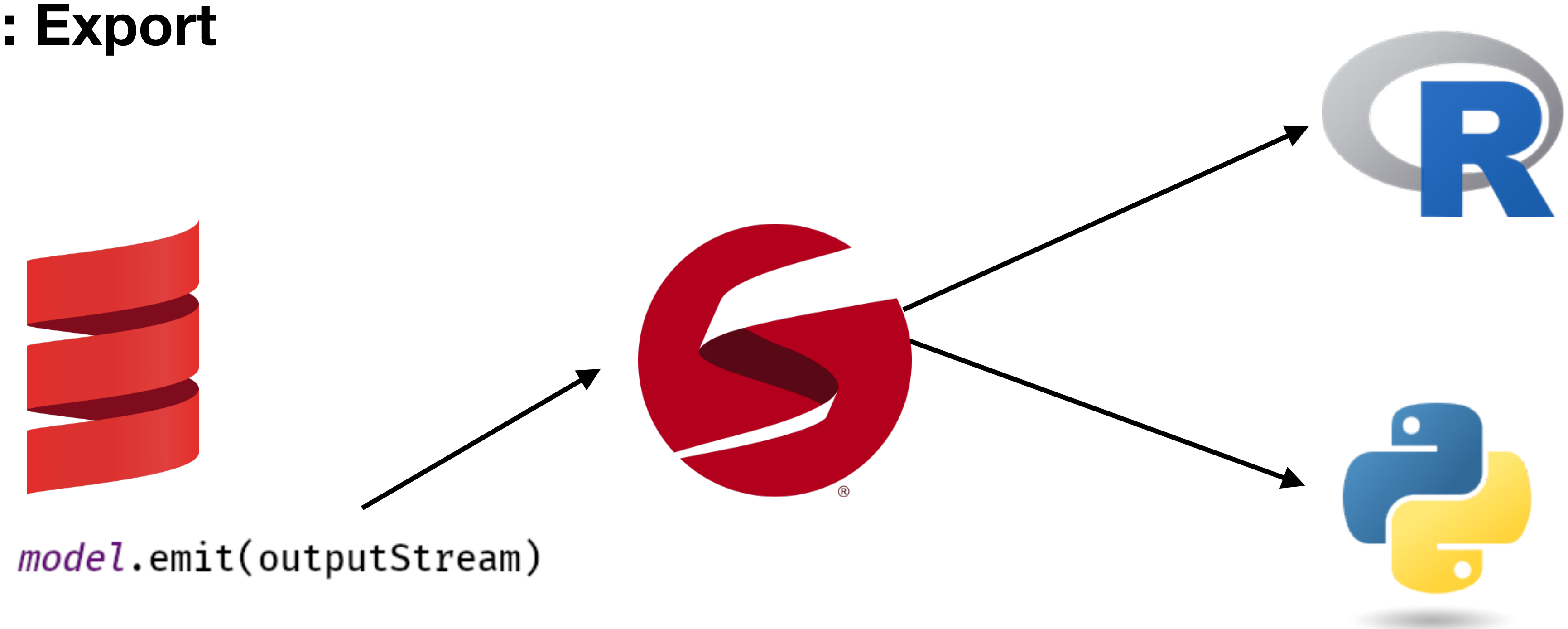
Integration into the Stan Ecosystem

Model Re-use: Export



Integration into the Stan Ecosystem

Model Re-use: Export



Model Re-use: Import

```
val x = data(…)
// …

val sigma = parameter(…)
// …

val model = Model.loadFromFile("model.stan")
```

Future Directions

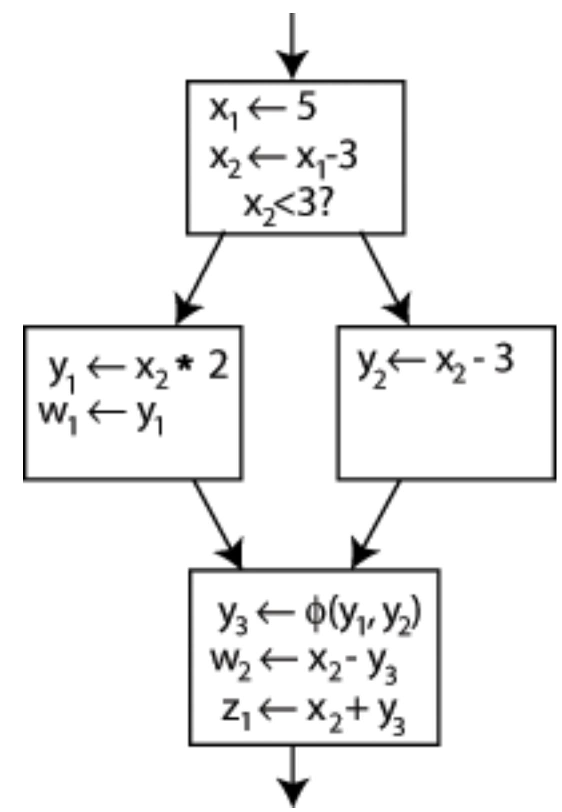
Higher-level API

- **Library of parameterizable models**

Stan code analysis and transformations

- **Automatic re-parameterization**
- **Automatic extraction of transformations**
- **Traditional compiler passes applied to Stan**

  Common subexpression elimination
  Constant Folding
  Strength Reduction
  Dead code elimination



CiBO TECHNOLOGIES

Questions?

ScalaStan GitHub Repository:
https://github.com/cibotech/ScalaStan

We're Hiring Engineers and Data Scientists of All Levels and Backgrounds!
https://www.cibotechnologies.com/about/