The Fusemate Logic Programming System

Peter Baumgartner

Data61|CSIRO and ANU, Canberra, Australia

Fusemate - Language and Model Computation Overview

Fusemate - Language and Model Computation Overview

Input language: Prolog-like rules

```
R(a,b)
R(X,Y) :- R(Y,X)
R(X,Z) :- R(X,Y), r(Y,Z)
```

Fusemate - Language and Model Computation Overview

Input language: Prolog-like rules

```
R(a,b)
R(X,Y) :- R(Y,X)
R(X,Z) :- R(X,Y), r(Y,Z)
```

Models

```
R(a,b)
R(b,a)
```

← Botton-up model generation
(Hyper tableau, Hyper resolution, SATCHMO, …)

Fusemate - Language and Model Computation Overview

Application:
Situational awareness
= model computation

Input language: Prolog-like rules

```
R(a,b)
R(X,Y) :- R(Y,X)
R(X,Z) :- R(X,Y), r(Y,Z)
```

Models

```
R(a,b)
R(b,a)
```

← Botton-up model generation
(Hyper tableau, Hyper resolution, SATCHMO, …)

Fusemate - Language and Model Computation Overview

Application:
Situational awareness
= model computation

Input language: Prolog-like rules

```
R(a,b)
R(X,Y) :- R(Y,X)
R(X,Z) :- R(X,Y), r(Y,Z)
```

Models

```
R(a,b)
R(b,a)
```

Botton-up model generation
(Hyper tableau, Hyper resolution, SATCHMO, …)

Default negation: stratification “by time”

```
GoodSleep(time) :-
   WakeUp(time),
   GoToBed(t), t <= time - 8,
   not (s < time, t < s, WakeUp(s))
```

Fusemate - Language and Model Computation Overview

Application:
Situational awareness
= model computation

Input language: Prolog-like rules

```
R(a,b)
R(X,Y) :- R(Y,X)
R(X,Z) :- R(X,Y), r(Y,Z)
```

Models

```
R(a,b)
R(b,a)
```

Botton-up model generation
(Hyper tableau, Hyper resolution, SATCHMO, …)

Default negation: stratification "by time"

```
GoodSleep(time) :-
   WakeUp(time),
   GoToBed(t), t <= time - 8,
   not (s < time, t < s, WakeUp(s))
```

"not" subgoals must be strictly earlier "<" than current time

Fusemate - Language and Model Computation Overview

Application:
Situational awareness
= model computation

Input language: Prolog-like rules

```
R(a,b)
R(X,Y) :- R(Y,X)
R(X,Z) :- R(X,Y), r(Y,Z)
```

Models

```
R(a,b)
R(b,a)
```

Botton-up model generation
(Hyper tableau, Hyper resolution, SATCHMO, …)

Default negation: stratification "by time"

```
GoodSleep(time) :-
   WakeUp(time),
   GoToBed(t), t <= time - 8,
   not (s < time, t < s, WakeUp(s))
```

"not" subgoals must be strictly earlier "<" than current time

```
unhappy(time) :- Now(time), not winLottery(time+7)
```

Fusemate - Language and Model Computation Overview

Application:
Situational awareness
= model computation

Input language: Prolog-like rules

```
R(a,b)
R(X,Y) :- R(Y,X)
R(X,Z) :- R(X,Y), r(Y,Z)
```

Models

```
R(a,b)
R(b,a)
```

Botton-up model generation
(Hyper tableau, Hyper resolution, SATCHMO, …)

Default negation: stratification "by time"

```
GoodSleep(time) :-
   WakeUp(time),
   GoToBed(t), t <= time - 8,
   not (s < time, t < s, WakeUp(s))
```

"not" subgoals must be strictly earlier "<" than current time

~~`unhappy(time) :- Now(time), not winLottery(time+7)`~~

Fusemate - Language and Model Computation Overview

Application:
Situational awareness
= model computation

Input language: Prolog-like rules

```
R(a,b)
R(X,Y) :- R(Y,X)
R(X,Z) :- R(X,Y), r(Y,Z)
```

Models

```
R(a,b)
R(b,a)
```

Botton-up model generation
(Hyper tableau, Hyper resolution, SATCHMO, ...)

Default negation: stratification "by time"

```
GoodSleep(time) :-
   WakeUp(time),
   GoToBed(t), t <= time - 8,
   not (s < time, t < s, WakeUp(s))
```

"not" subgoals must be strictly earlier "<" than current time

~~unhappy(time) :- Now(time), **not** winLottery(time+7)~~

Disjunctions: possible model semantics [Sakama 90]

```
Thirsty(time) or Hungry(time) :- GoodSleep(time)
```

Fusemate - Language and Model Computation Overview

Application:
Situational awareness
= model computation

Input language: Prolog-like rules

```
R(a,b)
R(X,Y) :- R(Y,X)
R(X,Z) :- R(X,Y), r(Y,Z)
```

Models

```
R(a,b)
R(b,a)
```

Botton-up model generation
(Hyper tableau, Hyper resolution, SATCHMO, …)

Default negation: stratification "by time"

```
GoodSleep(time) :-
   WakeUp(time),
   GoToBed(t), t <= time - 8,
   not (s < time, t < s, WakeUp(s))
```

"not" subgoals must be strictly earlier "<" than current time

~~unhappy(time) :- Now(time), **not** winLottery(time+7)~~

Disjunctions: possible model semantics [Sakama 90]

```
Thirsty(time) or Hungry(time) :- GoodSleep(time)
```

Inclusive reading of "or"

Models

:	:	:
Thirsty(10)	Hungry(10)	Hungry(10) Thirsty(10)

Application:
Situational awareness
= model computation

Fusemate - Language and Model Computation Overview

Input language: Prolog-like rules

```
R(a,b)
R(X,Y) :- R(Y,X)
R(X,Z) :- R(X,Y), r(Y,Z)
```

Models

```
R(a,b)
R(b,a)
```

Botton-up model generation
(Hyper tableau, Hyper resolution, SATCHMO, ...)

Default negation: stratification "by time"

```
GoodSleep(time) :-
   WakeUp(time),
   GoToBed(t), t <= time - 8,
   not (s < time, t < s, WakeUp(s))
```

"not" subgoals must be strictly earlier "<" than current time

~~unhappy(time) :- Now(time), **not** winLottery(time+7)~~

Disjunctions: possible model semantics [Sakama 90]

```
Thirsty(time) or Hungry(time) :- GoodSleep(time)
```

Inclusive reading of "or"

Models

:	:	:
Thirsty(10)	Hungry(10)	Hungry(10) Thirsty(10)

Belief revision

```
fail(+ GoToBed(time - 8)) :-
   WakeUp(time),
   not (GoToBed(t), t <= time - 8)
```

Fusemate - Language and Model Computation Overview

Application:
Situational awareness
= model computation

Input language: Prolog-like rules

```
R(a,b)
R(X,Y) :- R(Y,X)
R(X,Z) :- R(X,Y), r(Y,Z)
```

Models

```
R(a,b)
R(b,a)
```

Botton-up model generation
(Hyper tableau, Hyper resolution, SATCHMO, …)

Default negation: stratification "by time"

```
GoodSleep(time) :-
   WakeUp(time),
   GoToBed(t), t <= time - 8,
   not (s < time, t < s, WakeUp(s))
```

"not" subgoals must be strictly earlier "<" than current time

~~unhappy(time) :- Now(time), **not** winLottery(time+7)~~

Disjunctions: possible model semantics [Sakama 90]

```
Thirsty(time) or Hungry(time) :- GoodSleep(time)
```

Inclusive reading of "or"

Models

:	:	:
Thirsty(10)	Hungry(10)	Hungry(10) Thirsty(10)

Belief revision [IJCAR 2020]

```
fail(+ GoToBed(time - 8)) :-
   WakeUp(time),
   not (GoToBed(t), t <= time - 8)
```

Fusemate - Language and Model Computation Overview

Application:
Situational awareness
= model computation

Input language: Prolog-like rules

```
R(a,b)
R(X,Y) :- R(Y,X)
R(X,Z) :- R(X,Y), r(Y,Z)
```

Models

```
R(a,b)
R(b,a)
```

Botton-up model generation
(Hyper tableau, Hyper resolution, SATCHMO, …)

Default negation: stratification "by time"

```
GoodSleep(time) :-
   WakeUp(time),
   GoToBed(t), t <= time - 8,
   not (s < time, t < s, WakeUp(s))
```

"not" subgoals must be strictly earlier "<" than current time

~~unhappy(time) :- Now(time), **not** winLottery(time+7)~~

Disjunctions: possible model semantics [Sakama 90]

```
Thirsty(time) or Hungry(time) :- GoodSleep(time)
```

Inclusive reading of "or"

Models

:	:	:
Thirsty(10)	Hungry(10)	Hungry(10) Thirsty(10)

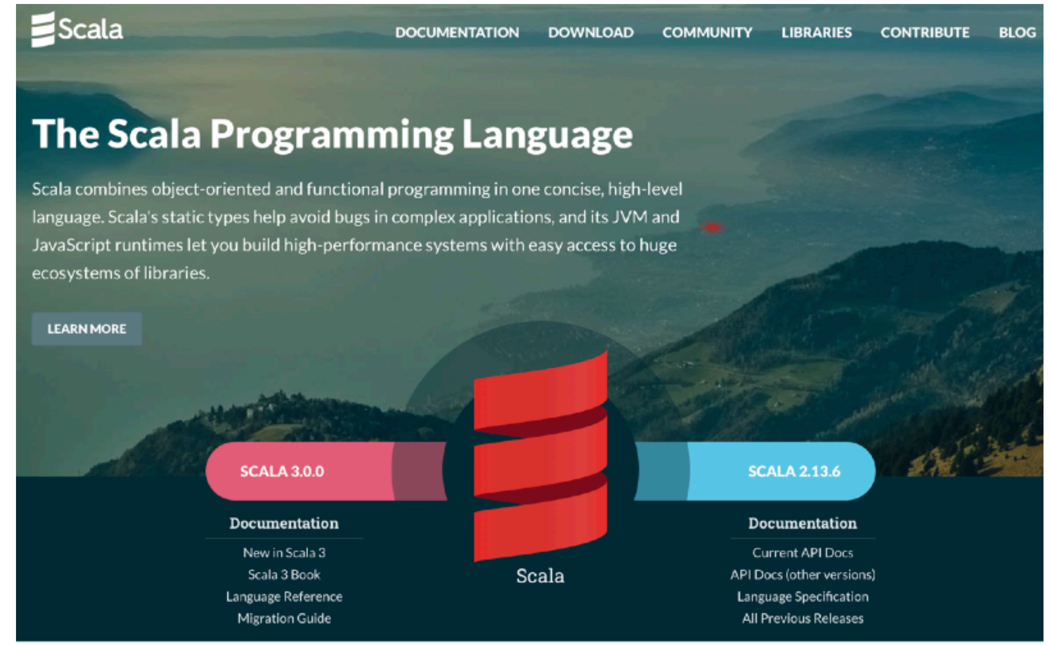
Belief revision [IJCAR 2020]

```
fail(+ GoToBed(time - 8)) :-
   WakeUp(time),
   not (GoToBed(t), t <= time - 8)
```

What's special?

What's new?

What’s Special?



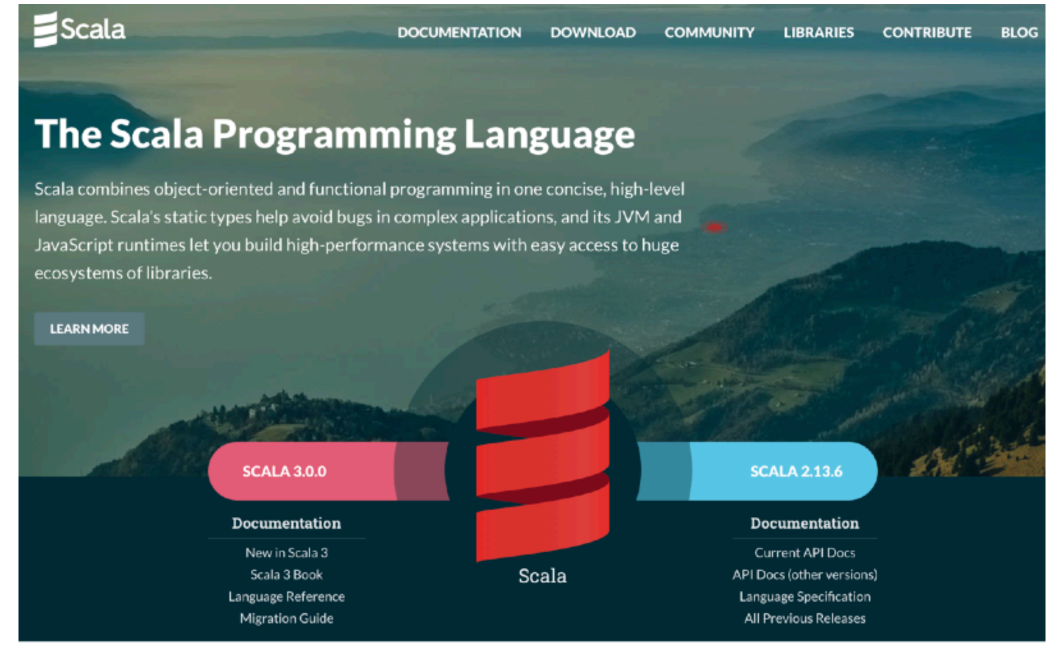
Implementation language: Scala

- Scala combines **object-oriented** and **functional programming**

```
def qsort(l: List[Int]): List[Int] =
  l match {
    case Nil            => Nil
    case pivot :: tail => qsort(tail filter {_ < pivot}) ::: pivot ::
                          qsort(tail filter {_ >= pivot})
  }
```

- Access to **huge ecosystem of libraries**
- Runs on JVM; compiled or in data-analysis style **interactive workbooks** (Jupyter)

What's Special?



Implementation language: Scala

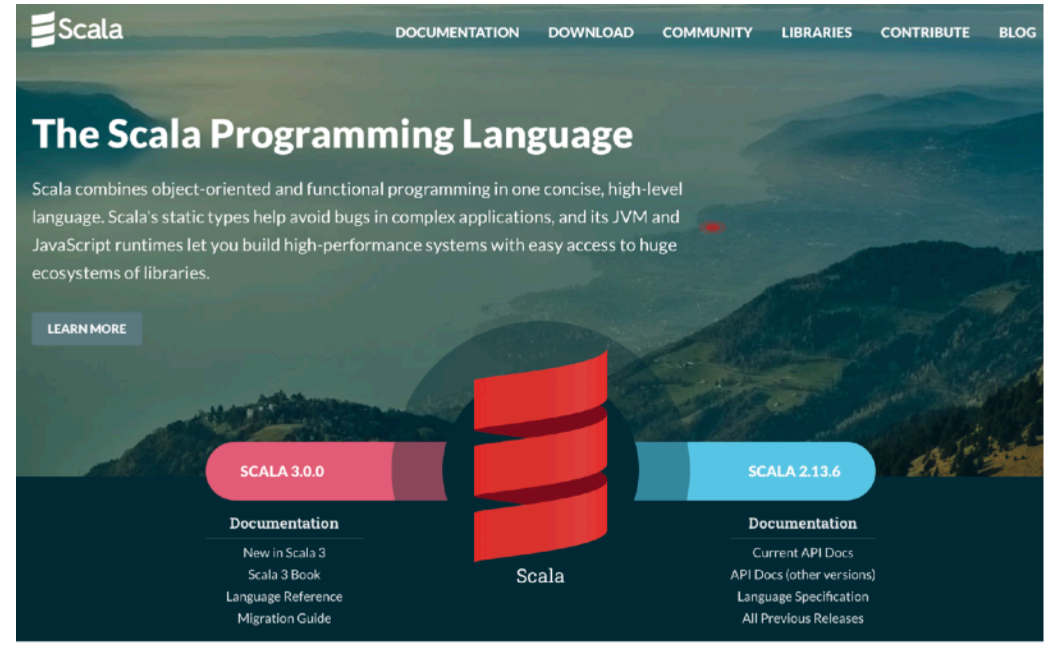
- Scala combines **object-oriented** and **functional programming**

```
def qsort(l: List[Int]): List[Int] =
  l match {
    case Nil          => Nil
    case pivot :: tail => qsort(tail filter {_ < pivot}) ::: pivot ::
                          qsort(tail filter {_ >= pivot})
  }
```

Pattern matching

- Access to **huge ecosystem of libraries**
- Runs on JVM; compiled or in data-analysis style **interactive workbooks** (Jupyter)

What’s Special?



Implementation language: Scala

- Scala combines **object-oriented** and **functional programming**

```
def qsort(l: List[Int]): List[Int] =
  l match {
    case Nil           => Nil
    case pivot :: tail => qsort(tail filter {_ < pivot}) ::: pivot ::
                          qsort(tail filter {_ >= pivot})
  }
```

Pattern matching

- Access to **huge ecosystem of libraries**
- Runs on JVM; compiled or in data-analysis style **interactive workbooks** (Jupyter)

Implementation technique: shallow embedding

- Logic program **translated into** Scala program that is executed for model computation
- AFAIK Fusemate is the only logic programming system implemented that way
- ***Q: what are the advantages/disadvantages of this approach?***

  E.g. in terms of capitalizing on / integrating the **above features** of Scala

Shallow Embedding Into Scala

- User writes Scala program with rules embedded into it

```scala
type Time = Int

case class GoodSleep(time: Time) extends Atom

@rules
    …
    GoodSleep(time) :-
        WakeUp(time),
        GoToBed(t), t <= time - 8,
        not (s < time, t < s, WakeUp(s))
```

Shallow Embedding Into Scala

- User writes Scala program with rules embedded into it

```
type Time = Int
case class GoodSleep(time: Time) extends Atom
@rules
    …
    GoodSleep(time) :-
        WakeUp(time),
        GoToBed(t), t <= time - 8,
        not (s < time, t < s, WakeUp(s))
```

Shallow Embedding Into Scala

- User writes Scala program with rules embedded into it

Σ

```
type Time = Int

case class GoodSleep(time: Time) extends Atom
```

Rules

```
@rules
    …
    GoodSleep(time) :-
        WakeUp(time),
        GoTo Bed(t), t <= time - 8,
        not (s < time, t < s, WakeUp(s))
```

Shallow Embedding Into Scala

- User writes Scala program with rules embedded into it

Σ

```
type Time = Int
case class GoodSleep(time: Time) extends Atom
@rules
    …
    GoodSleep(time) :-
        WakeUp(time),
        GoToBed(t), t <= time - 8,
        not (s < time, t < s, WakeUp(s))
```

Rules

- The rules are macro expanded into Scala curried partial functions

```
(I: Interpretation) => {
    case WakeUp(time) => {
        case GoToBed(t) if t <= time - 8 && I.failsOn("body of not") => GoodSleep(time)
        }
    }
```

Shallow Embedding Into Scala

- User writes Scala program with rules embedded into it

Σ

```
type Time = Int
case class GoodSleep(time: Time) extends Atom
@rules
    …
    GoodSleep(time) :-
        WakeUp(time),
        GoTo Bed(t), t <= time - 8,
        not (s < time, t < s, WakeUp(s))
```

Rules

- The rules are macro expanded into Scala curried partial functions

```
(I: Interpretation) => {
    case WakeUp(time) => {
        case GoToBed(t) if t <= time - 8 && I.failsOn("body of not") => GoodSleep(time)
        }
    }
```

Function application mimics rule evaluation

Shallow Embedding Into Scala

- User writes Scala program with rules embedded into it

Σ

```
type Time = Int
case class GoodSleep(time: Time) extends Atom
@rules
    …
    GoodSleep(time) :-
        WakeUp(time),
        GoToBed(t), t <= time - 8,
        not (s < time, t < s, WakeUp(s))
```

Rules

- The rules are macro expanded into Scala curried partial functions

```
(I: Interpretation) => {
    case WakeUp(time) => {
        case GoToBed(t) if t <= time - 8 && I.failsOn("body of not") => GoodSleep(time)
        }
    }
```

Function application mimics rule evaluation

Recursive call of model computation

Shallow Embedding Into Scala

- User writes Scala program with rules embedded into it

Σ

```
type Time = Int
case class GoodSleep(time: Time) extends Atom
@rules
    …
    GoodSleep(time) :-
        WakeUp(time),
        GoToBed(t), t <= time - 8,
        not (s < time, t < s, WakeUp(s))
```

Rules

- The rules are macro expanded into Scala curried partial functions

```
(I: Interpretation) => {
    case WakeUp(time) => {
        case GoToBed(t) if t <= time - 8 && I.failsOn("body of not") => GoodSleep(time)
    }
}
```

Function application mimics rule evaluation

Recursive call of model computation

- Given-clause loop operating on rules-as-partial-functions and interpretations (tableaux)

Shallow Embedding Into Scala

Logic	Scala
Pred/Fun signature	Class declaration
Atom/Term	Class instance
Interpretation	Set of class instances
Variable	Variable
Rule	Partial function
Matching subst	Pattern matching

- User writes Scala program with rules embedded into it

Σ

```
type Time = Int
case class GoodSleep(time: Time) extends Atom
@rules
    …
    GoodSleep(time) :-
        WakeUp(time),
        GoTo Bed(t), t <= time - 8,
        not (s < time, t < s, WakeUp(s))
```

Rules

All logic notions are Scala

- "Interpretation" available as term
- Trivial interface to/from Scala
- Type checking/inference for free

Every Scala term is a term of the logic

- The rules are macro expanded into Scala curried partial functions

```
(I: Interpretation) => {
    case WakeUp(time) => {
        case GoToBed(t) if t <= time - 8 && I.failsOn("body of not") => GoodSleep(time)
        }
    }
```

Function application mimics rule evaluation

Recursive call of model computation

- Given-clause loop operating on rules-as-partial-functions and interpretations (tableaux)

What's New? (1)

What’s New? (1)

General aggregation operator

What's New? (1)

General aggregation operator

- Many LP systems (DLV, IDP, Gringo, ...) support aggregation ops `#count #sum #times #min #max`

```
2 < #count { time : GoodSleep(time) }
```

What's New? (1)

General aggregation operator

- Many LP systems (DLV, IDP, Gringo, …) support aggregation ops `#count #sum #times #min #max`

```
2 < #count { time : GoodSleep(time) }
```

- Fusemate implements a more general stratified `collect` operator

```
collect(gsTimes, time sth GoodSleep(time))
```

Semantics: `gsTimes = { time | I ⊨ GoodSleep(time) }`

What's New? (1)

General aggregation operator

- Many LP systems (DLV, IDP, Gringo, …) support aggregation ops `#count #sum #times #min #max`

```
2 < #count { time : GoodSleep(time) }
```

- Fusemate implements a more general stratified `collect` operator **"Logic term = Scala term"**

```
collect(gsTimes, time sth GoodSleep(time))
```

Semantics: `gsTimes = { time | I ⊨ GoodSleep(time) }`

What's New? (1)

General aggregation operator

- Many LP systems (DLV, IDP, Gringo, …) support aggregation ops `#count #sum #times #min #max`

```
2 < #count { time : GoodSleep(time) }
```

- Fusemate implements a more general stratified `collect` operator **"Logic term = Scala term"**

```
collect(gsTimes, time sth GoodSleep(time))
```

Semantics: `gsTimes = { time | I ⊨ GoodSleep(time) }`

- Recover standard aggregation functionality #… with Scala operator

```
2 < gsTimes.size
```

What's New? (1)

General aggregation operator

- Many LP systems (DLV, IDP, Gringo, …) support aggregation ops `#count #sum #times #min #max`

```
2 < #count { time : GoodSleep(time) }
```

- Fusemate implements a more general stratified `collect` operator **"Logic term = Scala term"**

```
collect(gsTimes, time sth GoodSleep(time))
```

Semantics: $gsTimes = \{ time \mid I \models GoodSleep(time) \}$

- Recover standard aggregation functionality #… with Scala operator

```
2 < gsTimes.size
```

- But is more expressive

```
(gsTimes map { _ % 24 } foldLeft(0) { _ + _ }) / gsTimes.size
```

What's New? (1)

General aggregation operator

- Many LP systems (DLV, IDP, Gringo, …) support aggregation ops `#count #sum #times #min #max`

```
2 < #count { time : GoodSleep(time) }
```

- Fusemate implements a more general stratified `collect` operator   **"Logic term = Scala term"**

```
collect(gsTimes, time sth GoodSleep(time))
```

Semantics: `gsTimes = { time | I ⊨ GoodSleep(time) }`

- Recover standard aggregation functionality **#…** with Scala operator

```
2 < gsTimes.size
```

- But is more expressive

```
(gsTimes map { _ % 24 } foldLeft(0) { _ + _ }) / gsTimes.size
```

Comprehension operator

```
choose(t < time sth GoodSleep(t))
```

"The most recent `t` before `time` such that `GoodSleep(t)`"

- Useful for analysing "current state" in situational awareness application

What's New? (1)

General aggregation operator

- Many LP systems (DLV, IDP, Gringo, …) support aggregation ops `#count #sum #times #min #max`

```
2 < #count { time : GoodSleep(time) }
```

- Fusemate implements a more general stratified `collect` operator   "Logic term = Scala term"

```
collect(gsTimes, time sth GoodSleep(time))
```

Semantics: `gsTimes = { time | I ⊨ GoodSleep(time) }`

- Recover standard aggregation functionality #… with Scala operator

```
2 < gsTimes.size
```

- But is more expressive

```
(gsTimes map { _ % 24 } foldLeft(0) { _ + _ }) / gsTimes.size
```

Comprehension operator

These operators are user-definable

```
choose(t < time sth GoodSleep(t))
```

"The most recent `t` before `time` such that `GoodSleep(t)`"

- Useful for analysing "current state" in situational awareness application

What's New? (2)

Stratification by predicates and by time (SBTP)

- Stratification disallows definitorial loop through "**not** *<body>*" literal
- Stratification renders "**not** *<body>*" evaluation monotonic

```
p(time) :- q(time), not r(time-1)
r(time) :- p(time)
```

✗ Stratified by predicates

✓ Stratified by time

✓ **SBTP**

```
q(time) :- p(time), not s(time)
```

✓ Stratified by predicates

✗ Stratified by time

✓ **SBTP**

SBTP = lexicographic combination of "by time" and "by predicates"

What's New (1) - (2) Showcase - Fusemate as Description Logic Reasoner

Description logic ALCIF

$$\begin{aligned} \mathsf{Person} &\sqsubseteq \mathsf{Rich} \sqcup \mathsf{Poor} \\ \mathsf{Person} &\sqsubseteq \exists \mathsf{father}.\mathsf{Person} \\ \mathsf{Rich} &\sqsubseteq \forall \mathsf{father}^{-1}.\mathsf{Rich} \\ \mathsf{Rich} \sqcap \mathsf{Poor} &\sqsubseteq \bot \end{aligned}$$

father is functional

$$\begin{aligned} \mathsf{Anne} &: \mathsf{Person} \sqcap \mathsf{Poor} \\ (\mathsf{Anne}, \mathsf{Fred}) &: \mathsf{father} \\ \mathsf{Bob} &: \mathsf{Person} \\ (\mathsf{Bob}, \mathsf{Fred}) &: \mathsf{father} \end{aligned}$$

What's New (1) - (2) Showcase - Fusemate as Description Logic Reasoner

Description logic ALCIF

$$\mathsf{Person} \sqsubseteq \mathsf{Rich} \sqcup \mathsf{Poor}$$
$$\mathsf{Person} \sqsubseteq \exists \mathsf{father}.\mathsf{Person}$$
$$\mathsf{Rich} \sqsubseteq \forall \mathsf{father}^{-1}.\mathsf{Rich}$$
$$\mathsf{Rich} \sqcap \mathsf{Poor} \sqsubseteq \bot$$

father is functional

$$\mathsf{Anne} : \mathsf{Person} \sqcap \mathsf{Poor}$$
$$(\mathsf{Anne}, \mathsf{Fred}) : \mathsf{father}$$
$$\mathsf{Bob} : \mathsf{Person}$$
$$(\mathsf{Bob}, \mathsf{Fred}) : \mathsf{father}$$

Iterative algorithm

Uses SBTP

Uses aggregation

Paper has details

What's New (1) - (2) Showcase - Fusemate as Description Logic Reasoner

Description logic ALCIF

$$\text{Person} \sqsubseteq \text{Rich} \sqcup \text{Poor} \qquad \text{Anne} : \text{Person} \sqcap \text{Poor}$$
$$\text{Person} \sqsubseteq \exists \text{father}.\text{Person} \qquad (\text{Anne}, \text{Fred}) : \text{father}$$
$$\text{Rich} \sqsubseteq \forall \text{father}^{-1}.\text{Rich} \qquad \text{Bob} : \text{Person}$$
$$\text{Rich} \sqcap \text{Poor} \sqsubseteq \bot \qquad (\text{Bob}, \text{Fred}) : \text{father}$$

father is functional

Iterative algorithm

Uses SBTP

Uses aggregation

Paper has details

As a logic program

```
IsA(x, Exists(RN("father"), CN("Person")), time) :-
  IsA(x, CN("Person"), time)
```

What's New (1) - (2) Showcase - Fusemate as Description Logic Reasoner

Description logic ALCIF

$$\begin{aligned} \text{Person} &\sqsubseteq \text{Rich} \sqcup \text{Poor} \\ \text{Person} &\sqsubseteq \exists \text{father}.\text{Person} \\ \text{Rich} &\sqsubseteq \forall \text{father}^{-1}.\text{Rich} \\ \text{Rich} \sqcap \text{Poor} &\sqsubseteq \bot \end{aligned}$$

father is functional

$$\begin{aligned} \text{Anne} &: \text{Person} \sqcap \text{Poor} \\ (\text{Anne}, \text{Fred}) &: \text{father} \\ \text{Bob} &: \text{Person} \\ (\text{Bob}, \text{Fred}) &: \text{father} \end{aligned}$$

Iterative algorithm

Uses SBTP

Uses aggregation

Paper has details

As a logic program

```
IsA(x, Exists(RN("father"), CN("Person")), time) :-
  IsA(x, CN("Person"), time)
```

ALCIF satisfiability = LP satisfiability"

- LP encodes standard tableau construction [Baader et al 2017]
  - "Time" is quantifier expansion depth
  - TBox -> rules, ABox -> facts
  - Some general library rules
- Requires model inspection for "double blocking"

What's New (1) - (2) Showcase - Fusemate as Description Logic Reasoner

Description logic ALCIF

$$\begin{aligned} \mathsf{Person} &\sqsubseteq \mathsf{Rich} \sqcup \mathsf{Poor} \\ \mathsf{Person} &\sqsubseteq \exists \mathsf{father}.\mathsf{Person} \\ \mathsf{Rich} &\sqsubseteq \forall \mathsf{father}^{-1}.\mathsf{Rich} \\ \mathsf{Rich} \sqcap \mathsf{Poor} &\sqsubseteq \bot \end{aligned}$$

father is functional

$$\begin{aligned} \mathsf{Anne} &: \mathsf{Person} \sqcap \mathsf{Poor} \\ (\mathsf{Anne}, \mathsf{Fred}) &: \mathsf{father} \\ \mathsf{Bob} &: \mathsf{Person} \\ (\mathsf{Bob}, \mathsf{Fred}) &: \mathsf{father} \end{aligned}$$

Iterative algorithm

Uses SBTP

Uses aggregation

Paper has details

As a logic program

```
IsA(x, Exists(RN("father"), CN("Person")), time) :-
  IsA(x, CN("Person"), time)
```

ALCIF satisfiability = LP satisfiability"

- LP encodes standard tableau construction [Baader et al 2017]
  - "Time" is quantifier expansion depth
  - TBox -> rules, ABox -> facts
  - Some general library rules
- Requires model inspection for "double blocking"

```
Label(x, cs, time) :-
  IsA(x, _, time),
  COLLECT(cs, c STH IsA(x, c, time))

// Pairwise blocking
// y is blocked by x if ...
Blocked(y, x, time) :-
  // ... x is an ancestor of y,
  Anc(x, y, time),
  // ... the labels of y and x are the same
  Label(y, yIsAs, time),
  Label(x, xIsAs, time),
  yIsAs ≡ xIsAs,
  // ... y and x are r-successors of some y1 and x1, for s
  HasA(y1, r, y, time),
  HasA(x1, r, x, time),
  // ... the labels of y1 and x1 are the same
  Label(y1, y1IsAs, time),
  Label(x1, x1IsAs, time),
  y1IsAs ≡ x1IsAs
```

What's New (1) - (2) Showcase - Fusemate as Description Logic Reasoner

Description logic ALCIF

$$\mathsf{Person} \sqsubseteq \mathsf{Rich} \sqcup \mathsf{Poor}$$
$$\mathsf{Person} \sqsubseteq \exists \mathsf{father}.\mathsf{Person}$$
$$\mathsf{Rich} \sqsubseteq \forall \mathsf{father}^{-1}.\mathsf{Rich}$$
$$\mathsf{Rich} \sqcap \mathsf{Poor} \sqsubseteq \bot$$

father is functional

$$\mathsf{Anne} : \mathsf{Person} \sqcap \mathsf{Poor}$$
$$(\mathsf{Anne}, \mathsf{Fred}) : \mathsf{father}$$
$$\mathsf{Bob} : \mathsf{Person}$$
$$(\mathsf{Bob}, \mathsf{Fred}) : \mathsf{father}$$

Iterative algorithm

Uses SBTP

Uses aggregation

Paper has details

As a logic program

```
IsA(x, Exists(RN("father"), CN("Person")), time) :-
  IsA(x, CN("Person"), time)
```

ALCIF satisfiability = LP satisfiability"

- LP encodes standard tableau construction [Baader et al 2017]
  - "Time" is quantifier expansion depth
  - TBox -> rules, ABox -> facts
  - Some general library rules
- Requires model inspection for "double blocking"

```
Label(x, cs, time) :-
  IsA(x, _, time),
  COLLECT(cs, c STH IsA(x, c, time))

// Pairwise blocking
// y is blocked by x if ...
Blocked(y, x, time) :-
  // ... x is an ancestor of y,
  Anc(x, y, time),
  // ... the labels of y and x are the same
  Label(y, yIsAs, time),
  Label(x, xIsAs, time),
  yIsAs ≡ xIsAs,
  // ... y and x are r-successors of some y1 and x1, for s
  HasA(y1, r, y, time),
  HasA(x1, r, x, time),
  // ... the labels of y1 and x1 are the same
  Label(y1, y1IsAs, time),
  Label(x1, x1IsAs, time),
  y1IsAs ≡ x1IsAs
```

Textbook 1-to-1

What's New (3) - Usability and Workflow

Case study for combined Scala / logic programming workflow

2 Million taxi rides in New York City

`Ride(taxi,license,from,to,start,end,fare)`

Ride

Gap (between rides)

Pickup/dropoff clusters

(1) Rules for gaps, pickup/dropoff clustering and concave hull

(2) Rules for anomaly detection

```
=====
driver license-3568
=====
taxi-3568 license-3568 2013-01-01T22:10 2013-01-01T22:38        28m       5.7km
pickup anomaly from: hotspot-15
 hour:          0     1     2     3     4     5     6     7     8     9    10    11    12    13    14    15    16    17    18    19    20    21    22    23
 pickups:      16    34    35    30    26    20     7    20     8     5     9    25    36    36    31    55    50    44    24    64    69    38  |109|   21
 dropoffs:  (  16    40    70    73    48    22    33    17    22    28    44    43   116    76    76    83    57    74    70    76    36    13  | 34|   18  )
```

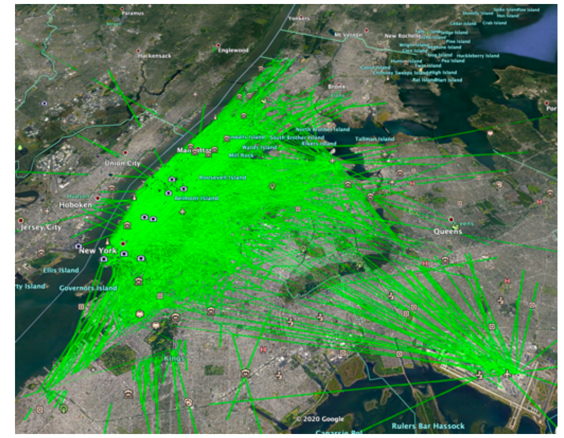
What’s New (3) - Usability an Workflow

From Scala to logic program and back

Scala is both extension language and scripting language

```
val gaps42 = rides filter {
    _.license ≡ "42"
  } saturateFirst {
    Gap(taxi, license, prevEnd, start, prevTo, from) :- (
      Ride(taxi, license, start, end,      _, _, from, _,        _, _),
      Ride(taxi, license, _,      prevEnd, _, _, _,     prevTo, _, _),
      start isAfter prevEnd,
      NOT (
        Ride(taxi, license, otherStart, otherEnd, _, _, _, _, _, _),
        (start isAfter otherStart) ∧ (otherStart isAfter prevEnd)
      )
    ) } collect {
    case g:Gap ⇒ g
  }
```

What's New (3) - Usability an Workflow

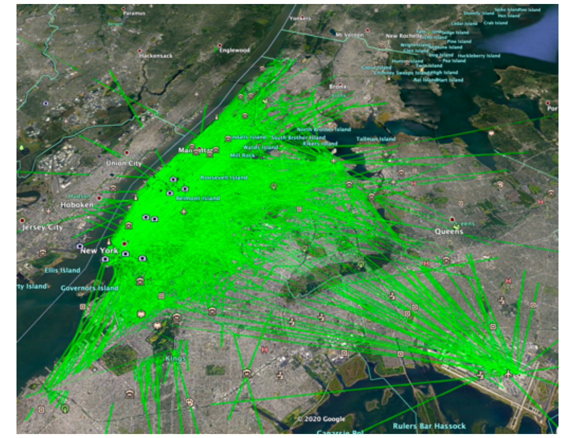


From Scala to logic program and back

Scala is both extension language and scripting language

```
val gaps42 = rides filter {
    _.license ≡ "42"
  } saturateFirst {
    Gap(taxi, license, prevEnd, start, prevTo, from) :- (
      Ride(taxi, license, start, end,      _, _, from, _,       _, _),
      Ride(taxi, license, _,      prevEnd, _, _, _,     prevTo, _, _),
      start isAfter prevEnd,
      NOT (
        Ride(taxi, license, otherStart, otherEnd, _, _, _, _, _, _),
        (start isAfter otherStart) ∧ (otherStart isAfter prevEnd)
      )
    ) } collect {
    case g:Gap ⇒ g
  }
```

What's New (3) - Usability an Workflow

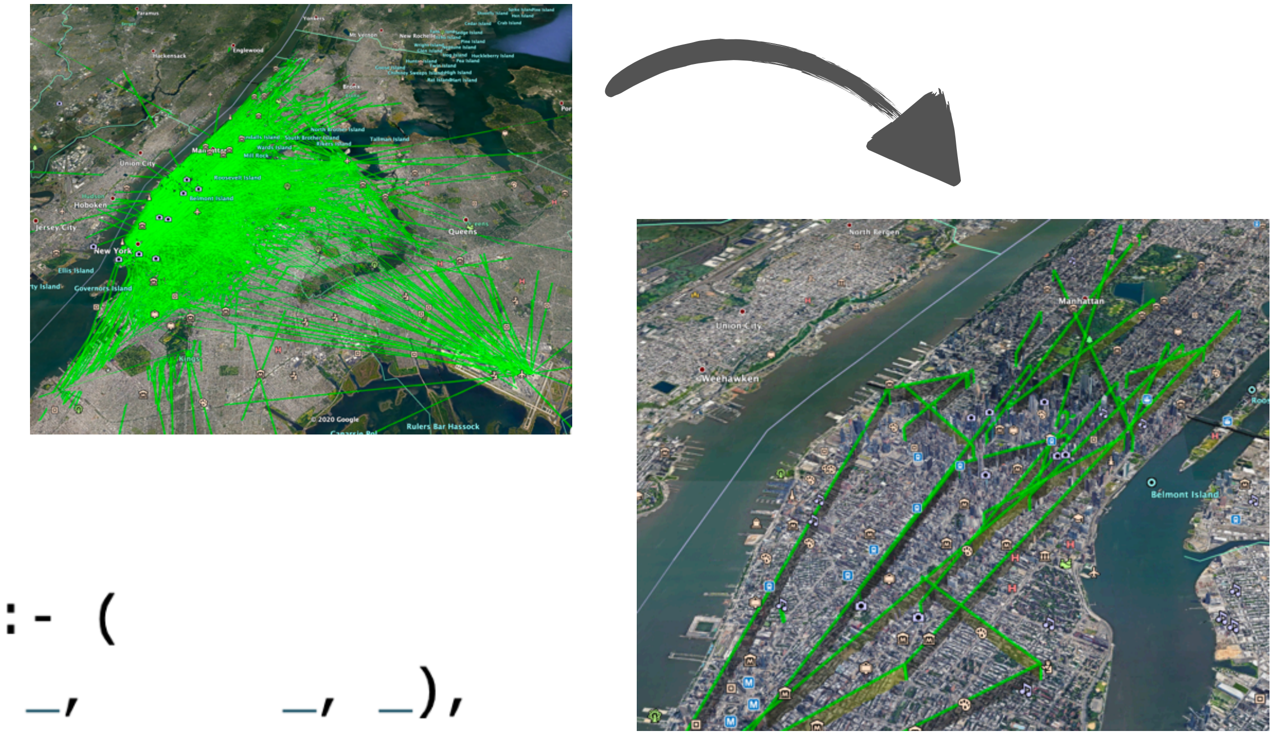


From Scala to logic program and back

Scala is both extension language and scripting language

```
val gaps42 = rides filter {
    _.license ≡ "42"
  } saturateFirst {
    Gap(taxi, license, prevEnd, start, prevTo, from) :- (
      Ride(taxi, license, start, end,     _, _, from, _,       _, _),
      Ride(taxi, license, _,      prevEnd, _, _, _,    prevTo, _, _),
      start isAfter prevEnd,
      NOT (
        Ride(taxi, license, otherStart, otherEnd, _, _, _, _, _, _),
        (start isAfter otherStart) ∧ (otherStart isAfter prevEnd)
      )
    ) } collect {
    case g:Gap ⇒ g
  }
```

What's New (3) - Usability an Workflow



From Scala to logic program and back

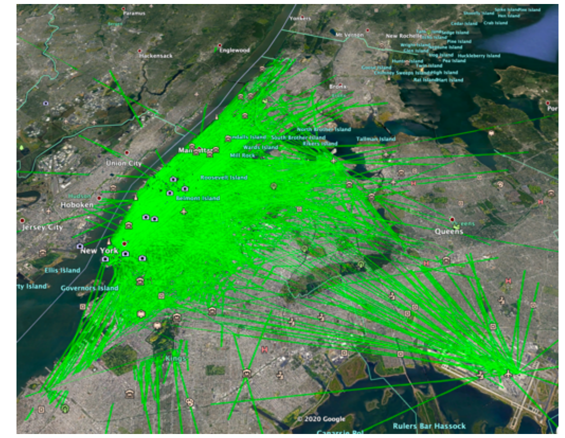
Scala is both extension language and scripting language

```
val gaps42 = rides filter {
    _.license ≡ "42"
  } saturateFirst {
    Gap(taxi, license, prevEnd, start, prevTo, from) :- (
      Ride(taxi, license, start, end,      _, _, from, _,        _, _),
      Ride(taxi, license, _,     prevEnd, _, _, _,     prevTo, _, _),
      start isAfter prevEnd,
      NOT (
        Ride(taxi, license, otherStart, otherEnd, _, _, _, _, _, _),
        (start isAfter otherStart) ∧ (otherStart isAfter prevEnd)
      )
    ) } collect {
    case g:Gap ⇒ g
  }
```

What’s New (3) - Usability an Workflow

From Scala to logic program and back

Scala is both extension language and scripting language



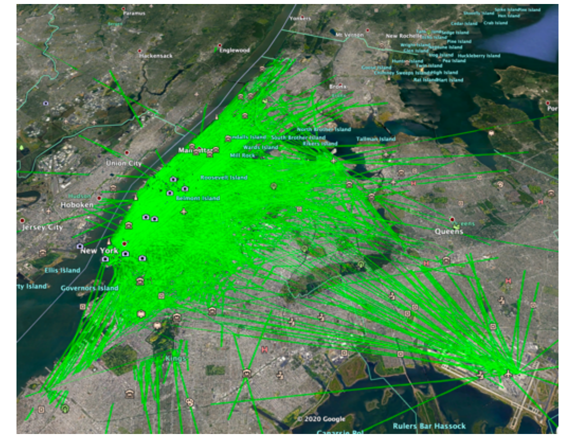
```
val gaps42 = rides filter {
    _.license ≡ "42"
  } saturateFirst {
    Gap(taxi, license, prevEnd, start, prevTo, from) :- (
      Ride(taxi, license, start, end,      _, _, from, _,       _, _),
      Ride(taxi, license, _,      prevEnd, _, _, _,     prevTo, _, _),
      start isAfter prevEnd,
      NOT (
        Ride(taxi, license, otherStart, otherEnd, _, _, _, _, _, _),
        (start isAfter otherStart) ∧ (otherStart isAfter prevEnd)
      )
    ) } collect {
    case g:Gap ⇒ g
  }
```

What’s New (3) - Usability an Workflow

From Scala to logic program and back

Scala is both extension language and scripting language

```
val gaps42 = rides filter {
    _.license ≡ "42"
  } saturateFirst {
    Gap(taxi, license, prevEnd, start, prevTo, from) :- (
      Ride(taxi, license, start, end,      _, _, from, _,        _, _),
      Ride(taxi, license, _,      prevEnd, _, _, _,    prevTo, _, _),
      start isAfter prevEnd,
      NOT (
        Ride(taxi, license, otherStart, otherEnd, _, _, _, _, _, _),
        (start isAfter otherStart) ∧ (otherStart isAfter prevEnd)
      )
    ) } collect {
    case g:Gap ⇒ g
  }
```



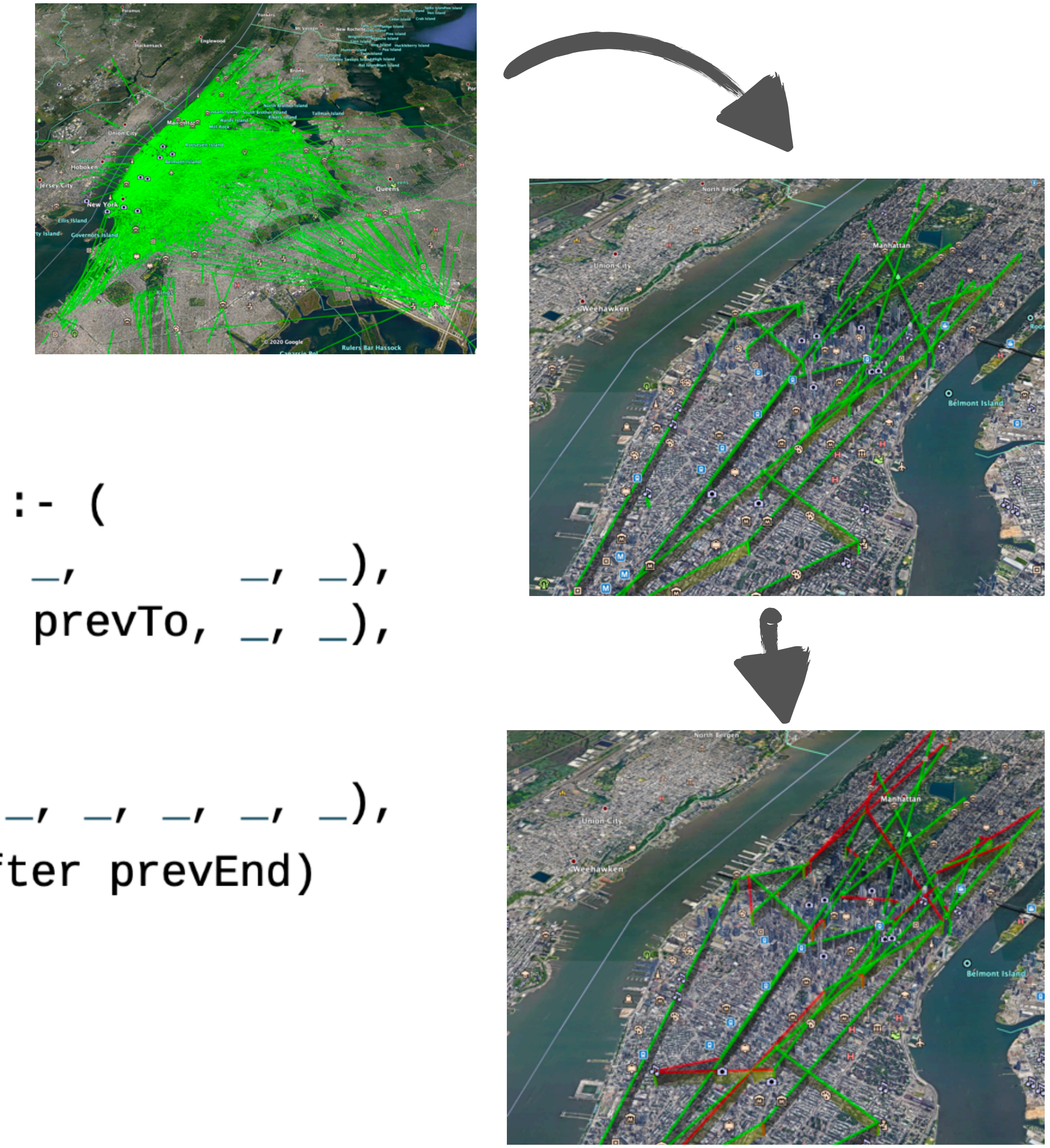
What’s New (3) - Usability an Workflow

From Scala to logic program and back

Scala is both extension language and scripting language

```
val gaps42 = rides filter {
    _.license ≡ "42"
  } saturateFirst {
    Gap(taxi, license, prevEnd, start, prevTo, from) :- (
      Ride(taxi, license, start, end,      _, _, from, _,        _, _),
      Ride(taxi, license, _,      prevEnd, _, _, _,     prevTo, _, _),
      start isAfter prevEnd,
      NOT (
        Ride(taxi, license, otherStart, otherEnd, _, _, _, _, _, _),
        (start isAfter otherStart) ∧ (otherStart isAfter prevEnd)
      )
    ) } collect {
    case g:Gap ⇒ g
  }
```

Fusemate invocation



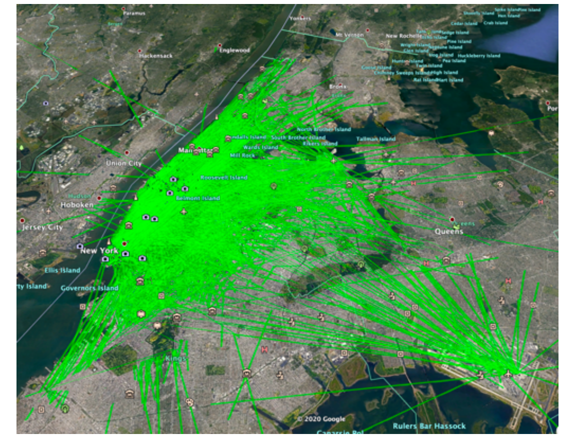
What's New (3) - Usability an Workflow

From Scala to logic program and back

Scala is both extension language and scripting language

```
val gaps42 = rides filter {
    _.license ≡ "42"
  } saturateFirst {
    Gap(taxi, license, prevEnd, start, prevTo, from) :- (
      Ride(taxi, license, start, end,     _, _, from, _,        _, _),
      Ride(taxi, license, _,      prevEnd, _, _, _,    prevTo, _, _),
      start isAfter prevEnd,
      NOT (
        Ride(taxi, license, otherStart, otherEnd, _, _, _, _, _, _),
        (start isAfter otherStart) ∧ (otherStart isAfter prevEnd)
      )
    ) } collect {
    case g:Gap ⇒ g
  }
```

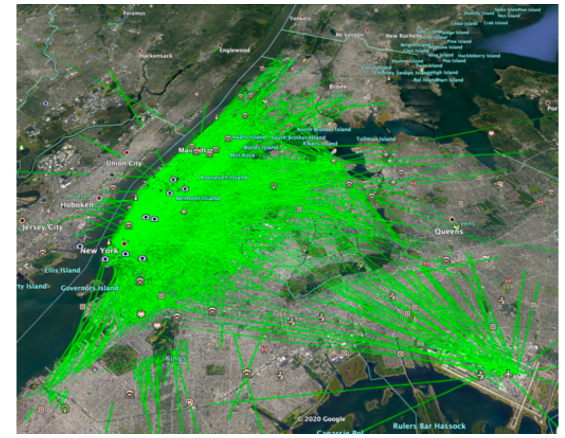
Fusemate invocation



What's New (3) - Usability an Workflow

From Scala to logic program and back

Scala is both extension language and scripting language



```
val gaps42 = rides filter {
    _.license ≡ "42"
  } saturateFirst {
    Gap(taxi, license, prevEnd, start, prevTo, from) :- (
      Ride(taxi, license, start, end,      _, _, from, _,        _, _),
      Ride(taxi, license, _,      prevEnd, _, _, _,     prevTo, _, _),
      start isAfter prevEnd,
      NOT (
        Ride(taxi, license, otherStart, otherEnd, _, _, _, _, _, _),
        (start isAfter otherStart) ∧ (otherStart isAfter prevEnd)
      )
    ) } collect {
    case g:Gap ⇒ g
  }
```

Fusemate invocation

What's New (3) - Usability an Workflow

From Scala to logic program and back

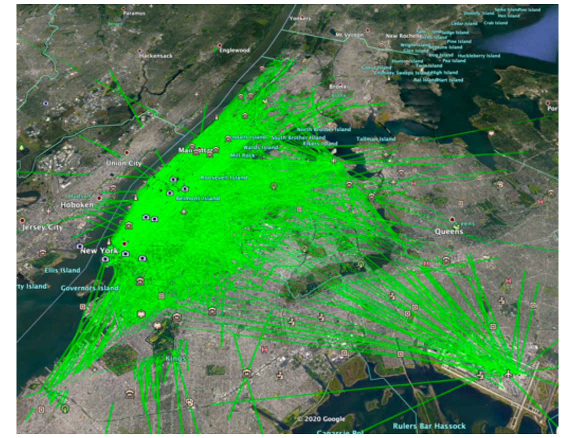
Scala is both extension language and scripting language

```
val gaps42 = rides filter {
    _.license ≡ "42"
  } saturateFirst {
    Gap(taxi, license, prevEnd, start, prevTo, from) :- (
      Ride(taxi, license, start, end,      _, _, from, _,        _, _),
      Ride(taxi, license, _,      prevEnd, _, _, _,    prevTo, _, _),
      start isAfter prevEnd,
      NOT (
        Ride(taxi, license, otherStart, otherEnd, _, _, _, _, _, _),
        (start isAfter otherStart) ∧ (otherStart isAfter prevEnd)
      )
    ) } collect {
    case g:Gap ⇒ g
  }
```

Fusemate invocation

Functional + Logic programming

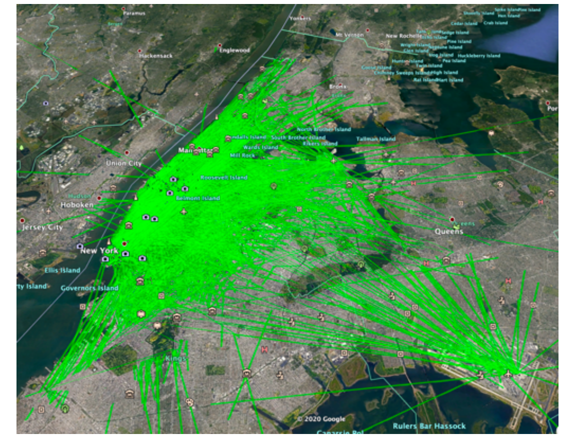
(in a new way?)



What's New (3) - Usability an Workflow

From Scala to logic program and back

Scala is both extension language and scripting language



```
val gaps42 = rides filter {
    _.license ≡ "42"
  } saturateFirst {
    Gap(taxi, license, prevEnd, start, prevTo, from) :- (
      Ride(taxi, license, start, end,     _, _, from, _,       _, _),
      Ride(taxi, license, _,     prevEnd, _, _, _,    prevTo, _, _),
      start isAfter prevEnd,
      NOT (
        Ride(taxi, license, otherStart, otherEnd, _, _, _, _, _, _),
        (start isAfter otherStart) ∧ (otherStart isAfter prevEnd)
      )
    ) } collect {
    case g:Gap ⇒ g
  }
```

Fusemate invocation

Defined as a Scala function

Functional + Logic programming
(in a new way?)

Conclusions

Fusemate is implemented by shallow embedding into Scala

- New operators for aggregation and comprehension
- Atoms and interpretations are first-class citizens
- Light-weight interface logic programming <-> Scala

  Workflow: logic programming = operator on collections of objects (case classes)

Efficiency

- SAT problem for propositional possible models of stratified DLPs is NP-complete
- Atoms indexed by time then indexed by predicate symbols

  Helps a lot, in particular “comprehension”
- OK for slow-running processes

  Bigger data sets currently need combined workflow (taxi example)

Availability

`https://bitbucket.csiro.au/users/bau050/repos/fusemate/`