Of Lambdas and Linguists
Montague Semantics

Haskell Meetup
Singapore (7 August 2019)
Linguistics in a Haskell Meetup?

Natural languages

- weird! wonderful! intuitive
- messy and hard to work with

Logical languages

- easy to reason with
- not how people talk

Typed lambda calculus?
Disclaimer 1

I’m a coder, not a linguist!

linguists, computational linguists, logicians
I love you all
Siri? Alexa?
Google Translate?

😭
sorry, no idea!
Logical languages 🤖
Pieces of a logic

1. Vocabulary
2. Model
3. Language
4. Satisfaction

```
data Vocab =
  V0 Thing
  | V1 Rel1
  | V2 Rel2

data VocabThing =
  Bart
  | Lisa
  | SantaLH -- Santa's Little Helper

data VocabRel1 =
  Student
  | Greyhound
  | Run

data VocabRel2 =
  Sibling
  | Torment
```
Pieces of a logic

Model ▸ Domain

data Entity = E1 | E2 | E3 | E4..
    deriving (Bounded)

Model ▸ Interpretation Function

model9871 = (model9871_t, model9871_u, model9871_b)

model9871_t :: VocabThing → {Entity}
model9871_t Bart          = {E1}
model9871_t Lisa          = {E2}
model9871_t SantaLH       = {E3}

model9871_u :: VocabRel1 → {Entity}
model9871_u Student       = {E1, E2}
model9871_u Greyhound     = {E3}
model9871_u Runs          = {E2, E3}

model9871_b :: VocabRel2 → {(Entity, Entity)}
model9871_b Sibling       = {((E1, E2), (E2, E1))
model9871_b Torment       = {((E1, E2)},

1. Vocabulary
2. Model
3. Language
4. Satisfaction
newtype Variable = Variable { fromVariable :: String }

data Term =
  Const Thing -- Bart
| Var Variable -- x

data Formula =
  UnaryRel Rel1 Term -- Greyhound(x)
| BinaryRel Rel2 Term Term -- Torment(Bart)(Lisa)
| Not Formula -- ¬Torment(Lisa)(Bart)
| And Formula Formula -- Greyhound(x) ∧ Run(x)
| Or Formula Formula -- Greyhound(x) ∨ Student(y)
| Implies Formula Formula -- Student(x) → Torment[x][Lisa]
| Exists Variable Formula -- ∃z.Sibling[z][SantaLH]
| ForAll Variable Formula -- ∀z.¬Torment[Lisa][z]
| Possibly Formula -- ◇Sibling[Lisa][Bart]
| Necessarily Formula -- □Sibling[Bart][Lisa]

Vocabulary
Model
Language
Satisfaction
Pieces of a logic

type Assignment = Variable → Maybe Entity

assign3193 :: Assignment
assign3193 “x” = Just E1
assign3193 “y” = Just E2
assign3193 _ = Nothing

class Interp res v where
  interp :: Model → Assignment → v → {res}

instance Interp Entity Variable where
  interp mdl asg v = maybe emptySet singleton (asg v)

instance Interp Entity VocabThing where
  interp mdl asg t = (fst3 mdl) t

instance Interp (Entity, Entity) VocabRel2 where
  interp mdl asg t = (thd3 mdl) t

instance Interp Entity Term where
  interp mdl asg (Constant c) = interp mdl asg c
  interp mdl asg (Var v)      = interp mdl asg v
Pieces of a logic

satisfies :: Model → Assignment → Formula → Maybe Bool
satisfies model asg form =
  case interp model asg form of
    {} -> Nothing
    xs -> Just (or xs)

instance Interp Bool Formula where
  interp mdl asg f = case f of
    UnaryRel  r t     -> ⟨ int t ∈ int r ⟩
    BinaryRel r t1 t2 -> ⟨(int t1, int t2) ∈ int r ⟩
    Not f       -> ⟨ not (sat f) ⟩
    And f1 f2   -> ⟨ int f1 && int f2 ⟩
    Or f1 f2    -> ⟨ int f1 || int f2 ⟩
    Implies p q -> ⟨ not (int f1) || int f2 ⟩
    Exists x f  -> ⟨ any (\a -> interp mdl a f) (variants asg x) ⟩
    ForAll x f  -> ⟨ all (\a -> interp mdl a f) (variants asg x) ⟩
  where
    int = interp mdl asg

variants :: Assignment → Variable → {Assignment}
variants asg v0 = Set.fromList $
  [ \v -> if v == v0 then Just e else asg v
    | e <- [minBound..maxBound]] -- for all entities
Jobs for a 🤖

Is it true? (querying, model checking)

alwaysSatisfies model47 (canOpenThePodBayDoors Hal)

\( M \models PodBayDoors(d) \land CanOpen(Hal, d) \)

Does it make sense? (consistency, satisfiability)

\( [\forall x.\text{Greyhound}(x) \rightarrow \text{Runs}(x)] \land \text{Greyhound}(\text{SantasLH}) \land \neg \text{Runs}(\text{SantasLH}) \)

Is this new? (informativity)

Bart and Lisa are students
Bart is a student

very nice, but most people don't speak in logic
Natural languages 🎨
1960s: formal study of syntax

Every student admired Lisa

Context Free Grammar
1960s: formal study of syntax

Every student admired Lisa

Context Free Grammar

Every student admired Lisa
1960s: formal study of syntax

Every student admired Lisa

Context Free Grammar
Every student admired Lisa
Every student admired Lisa
1960s: formal study of syntax

very nice, but what about meaning?

Every student admired Lisa

Context Free Grammar
“There is in my opinion no important theoretical difference between natural languages and the artificial languages of logicians; indeed, I consider it possible to comprehend the syntax and semantics of both kinds of languages within a single natural and mathematically precise theory. On this point I differ from a number of philosophers, but agree, I believe, with Chomsky and his associates.”
“It is clear, however, that no adequate and comprehensive semantical theory has yet been constructed, and arguable that no comprehensive and semantically significant syntactical theory yet exists.”
The rough idea

Sentence in a Natural Language \(\rightarrow\) Formula in a Logical Language

???

Meaning of the Sentence \(\rightarrow\) Meaning of the Formula

✅
The secret weapon?
Lambda Calculus (1930)
1. Lambda-fy your logic

data Term2 =
  Const  Thing
| NVar   Variable
| LVar   LVariable

data Formula2
  | Lambda LVariable NFormula          -- \(\lambda x.\text{Torment}(x)(\text{Lisa})\)
  | Apply  LVariable NFormula Formula2 -- \(\lambda x.\text{Torment}(x)(\text{Lisa}) @ \text{Bart}\)
  | NForm  NFormula

data NFormula =
  UnaryRel  Rel1 Term2          -- \text{Greyhound}(x)
| BinaryRel Rel2 Term2 Term2    -- \text{Torment}(\text{Bart})\{\text{Lisa}\}
| Not NFormula                  -- \neg\text{Torment}(\text{Lisa})(\text{Bart})
| And NFormula NFormula         -- \text{Greyhound}(x) \land \text{Run}(x)
| Or  NFormula NFormula         -- \text{Greyhound}(x) \lor \text{Student}(y)
| Implies NFormula NFormula     -- \text{Student}(x) \rightarrow \text{Torment}(x)(\text{Lisa})
| Exists Variable NFormula      -- \exists z.\text{Sibling}(z,\text{SantaLH})
| ForAll Variable NFormula      -- \forall z.\neg\text{Torment}(\text{Lisa})(z)
2. Pair syntactic rules with semantic fragments

\[
\begin{align*}
\text{NP} & \quad \oplus \quad \text{PN} \\
\downarrow \lambda X.X & \quad \quad \quad \downarrow \text{Lisa} \quad \text{Lisa} \\
\text{PN} & = \quad \text{PN} \\
\lambda X.X & \quad \text{Lisa} \quad \text{Lisa} \\
\end{align*}
\]

\[
\begin{align*}
\text{VP} & \quad \oplus \quad \text{IV} \\
\downarrow \lambda X.X & \quad \text{thinks} \quad \lambda u.\text{Think}[u] \\
\text{IV} & = \quad \text{IV} \\
\downarrow \lambda X.X & \quad \lambda u.\text{Think}[u] \text{ thinks} \\
\lambda u.\text{Think}[u] & \quad \text{think SQ}\lambda R.R@Q \\
\end{align*}
\]
2. Pair syntactic rules with semantic fragments

```
S
  NP ↓  VP ↓
  λQ λR.R@Q
⊕

NP  VP
  ↓  ↓
Lisa  thinks
 λu.Think[u]
```

```
S
  VP  NP
  ↓  ↓
Lisa  λX.X
  think
Think[Lisa]

NP  VP
  ↓  ↓
λQ λR.R@Q
```

```
S
  IV  NP
  ↓  ↓
Lisa  λu.Think[u]
```
2. Pair syntactic rules with semantic fragments

\[
\begin{align*}
S & \quad \downarrow \quad \text{NP} \quad \downarrow \quad \text{VP} \\
\quad & \quad \downarrow \quad \lambda Q \lambda R. Q @ R \\
\oplus & \\
\text{NP} & \quad \downarrow \quad \text{VP} \\
\text{PN} & \quad \downarrow \quad \text{IV} \\
\text{Lisa} & \quad \text{thinks} \\
\text{Lisa} \quad \lambda u. \text{Think}[u] & \\
\end{align*}
\]

\[
= \quad \begin{array}{c}
S \\
\downarrow \quad \text{NP} \quad \downarrow \quad \text{VP} \\
\downarrow \quad \text{PN} \quad \downarrow \quad \text{IV} \\
\text{Lisa} \quad \text{thinks} \\
\end{array}
\]

\[
\begin{array}{c}
\lambda R. R @ \text{Lisa} \quad \lambda u. \text{Think}[u] \\
\end{array}
\]

\[
= \quad \begin{array}{c}
S \\
\downarrow \quad \text{NP} \quad \downarrow \quad \text{VP} \\
\downarrow \quad \text{PN} \quad \downarrow \quad \text{IV} \\
\text{Lisa} \quad \text{thinks} \\
\end{array}
\]

\[
\lambda Q \lambda R. Q @ R \\
\]

\[
\begin{array}{c}
\{ \lambda R. R @ \text{Lisa} \} @ \{ \lambda u. \text{Think}[u] \} \\
\{ \lambda u. \text{Think}[u] \} @ \text{Lisa} \\
\text{Think}(\text{Lisa}) \\
\end{array}
\]
3. Think real hard

Determiners

\[
\lambda D \lambda R \cdot D @ R
\]

\[
\lambda N \lambda R \cdot \exists x. N @ x \land R @ x
\]

\[
\lambda h. \text{Student}(h)
\]

\[
\lambda R \cdot \exists x. \text{Student}(x) \land R @ x
\]

\[
\lambda R. R @ \text{Lisa}
\]

\[
\lambda u. \text{Think}(u)
\]

\[
\lambda Q \lambda R. Q @ R
\]

\[
\lambda x. x
\]

\[
\lambda x. x
\]

\[
\lambda N \lambda R \cdot \exists x. N @ x \land R @ x
\]

\[
\lambda h. \text{Student}(h)
\]

\[
\lambda R. R @ \text{Lisa}
\]

\[
\lambda u. \text{Think}(u)
\]
3. Think real hard

Transitive verbs

Every teacher inspires a student

\[
\lambda v.\text{Inspires}(\lambda R_2 \exists x.\text{Student}[x] \land R@x)[v]
\]

😊 not well formed
3. Think real hard
Transitive verbs

Every teacher inspires a student

\[ \lambda R_2 \lambda Q.Q@R_2 \]

\[ \forall x. \text{Student}(x) \land \lambda v. \text{Inspires}(x)(v) \]

(ERR) not well formed
This is way easier with types

\[
\text{type } \text{Rel1} = \text{Entity} \rightarrow \text{Bool} \\
[\text{Lisa}] :: \text{Rel1} \rightarrow \text{Bool} \\
[\text{Lisa}] = \lambda R. R@\text{Lisa} \\
[\text{thinks}] :: \text{Rel1} \\
[\text{thinks}] = \lambda u. \text{Think}(u) \\
\]

\[
\text{[[Lisa thinks]]} :: \text{Bool} \\
\text{[[Lisa]]} :: \text{Rel1} \rightarrow \text{Bool} \\
\text{[[thinks]]} :: \text{Rel1} \\
\]

\[
[a] :: \text{Rel1} \rightarrow \text{Rel1} \rightarrow \text{Bool} \\
[a] = \lambda N \lambda R. \exists x. N@x \land R@x \\
[\text{[teacher]}] :: \text{Rel1} \\
[\text{[teacher]}] = \lambda h. \text{Teacher}(h) \\
\]

\[
\text{[[a teacher thinks]]} :: \text{Bool} \\
\text{[[a teacher]]} :: \text{Rel1} \rightarrow \text{Bool} \\
\text{[[thinks]]} :: \text{Rel1} \\
\]
This is way easier with types

type Rel1 = Entity → Bool  
[[lisa]] :: Rel1 → Bool  
[[lisa]] = λR.R@Lisa  
[[thinks]] :: Rel1  
[[thinks]] = λu.Think[u]  

[[Lisa thinks]] :: Bool  
[[Lisa]] :: Rel1 → Bool  
[[thinks]] :: Rel1

[[Lisa inspires a student]] :: Bool  
[[Lisa]] :: Rel1 → Bool  
[[inspires a student]] :: Rel1
This is way easier with types

```haskell
type Rel1 = Entity → Bool

⟦Lisa inspires a student⟧ :: Bool
⟦Lisa⟧ :: Rel1 → Bool
⟦inspires a student⟧ :: Rel1

type Rel2 = Entity → Rel1

⟦inspires a student⟧ :: Rel1
⟦a student⟧ :: Rel1 → Bool
⟦inspires⟧ :: Entity → Entity → Bool
⟦inspires⟧ :: Entity → Rel1

vpTrans :: (Entity → Rel1) → (Rel1 → Bool) → Rel1
vpTrans = ???
```
This is way easier with types

```
type Rel1 = Entity \rightarrow \text{Bool}

\begin{align*}
[\text{inspires a student}] & : \text{Rel1} \\
[\text{a student}] & : \text{Rel1} \rightarrow \text{Bool} \\
[\text{inspires}] & : \text{Entity} \rightarrow \text{Entity} \rightarrow \text{Bool} \\
[\text{inspires}] & : \text{Entity} \rightarrow \text{Rel1}
\end{align*}
```

```
vpTrans2 :: (Entity \rightarrow \text{Rel1}) \rightarrow (\text{Rel1} \rightarrow \text{Bool}) \rightarrow \text{Rel1}
vpTrans2 \ r2 \ q = q \ r2 -- 😞 want \text{Rel1} got (Entity \rightarrow \text{Rel1})
```

```
vpTrans3 :: (Entity \rightarrow \text{Rel1}) \rightarrow (\text{Rel1} \rightarrow \text{Bool}) \rightarrow \text{Rel1}
vpTrans3 \ r2 \ q = \lambda a \rightarrow q \ (r2 \ a)
vpTrans3 \ r2 \ q = \lambda a \rightarrow q \ (\lambda b \rightarrow r2 \ a \ b) -- \text{eta expanded}
```
3. Think real hard

Transitive verbs

Every teacher inspires a student

\[
\lambda R_2 \lambda Q. \lambda a.(Q@\lambda b.[R_2@a@b])
\]

\[
\lambda u \lambda v. \text{Inspire}(u)(v)
\]

\[
\lambda R. \exists x. \text{Student}(x) \land R@x
\]

\[
\lambda \text{every Student}(x) \rightarrow R@x
\]

\[
\lambda R. \forall x. N@x \\
\land R@x
\]
3. Think real hard

Transitive verbs

Every teacher inspires a student

\[ \lambda R_2 \lambda Q. \lambda a. (Q@b. [R_2@a@b]) \]

\[ \lambda u. \lambda v. \text{Inspire}[u][v] \]

\[ \lambda R. \exists x. \text{Student}[x] \wedge \text{Inspire}[a][x] \]

\[ \forall x. \text{Student}[x] \Rightarrow R@x \]

\[ \lambda N \lambda R. \forall x. N@x \rightarrow R@x \]

\[ \lambda N \lambda R. \exists x. N@x \wedge R@x \]
Every teacher inspires a student

∀y.Teacher(y) → [ ∃x.Student(x) ∧ Inspire(y)[x] ]

∀y.Teacher(y) → [ ∃x.Student(x) ∧ Inspire(y)[x] ]
Also covered in Montague 1973 😳

1. Pronouns
2. Tricky small words
   - meaning of “the”
   - meaning of “is”
3. Tense
4. Modal expressions
   - “possibly”
   - “necessarily”
5. **Scope ambiguities**
   Every teacher inspires a student
   ...His name is Bart

6. **Intension vs extension**
   The Borg assimilated the captain of the Enterprise

7. **Intensional verbs**
   John finds a unicorn
   John seeks a unicorn
   John tries to find a unicorn

Also covered in Montague 1973 😢
Things to read

Representation and Inference for Natural Language
A First Course in Computational Semantics
Patrick Blackburn and Johan Bos

Teaching material

- Montague Semantics (Symbolic Systems 100 course)
  Christopher Potts
- Formal Semantics Teaching Material: https://people.umass.edu/partee/Teaching.htm
  Barbara Partee

Montague Papers

- English as a formal language (1970)
- Universal grammar (1970)
- The proper treatment of quantification in ordinary English (1973)
Things to read

Haskell + Natural Language

• Continuations: natural language meaning as computation
  Chris Barker and Chung-chieh Shan
• Grammatical Framework
  Aarne Ranta
• Haskell Wiki on Linguistics
  https://wiki.haskell.org/Applications_and_libraries/Linguistics