Presupposition projection as a scope phenomenon

Julian Grove

CLASP, University of Gothenburg

Ling. research sem., March 23, 2021

The empirical observation

We have linguistic devices that grammatically encode what we take for granted in making an utterance.

The empirical observation

We have linguistic devices that grammatically encode what we take for granted in making an utterance.

1. Karlos brought <u>his car</u>.

The empirical observation

We have linguistic devices that grammatically encode what we take for granted in making an utterance.

1. Karlos brought his car.
   - Karlos has a car. (presupposition)

The empirical observation

How do we identity presuppositions?

The empirical observation

How do we identity presuppositions?

- family-of-sentence tests (Chierchia and McConnell-Ginet, 2000)

How do we identity presuppositions?

- family-of-sentence tests (Chierchia and McConnell-Ginet, 2000)
- “hey, wait a minute!” test (von Fintel, 2004)

How do we identity presuppositions?

- family-of-sentence tests (Chierchia and McConnell-Ginet, 2000)
- "hey, wait a minute!" test (von Fintel, 2004)

The empirical observation

How do we identity presuppositions?

- family-of-sentence tests (Chierchia and McConnell-Ginet, 2000)
- "hey, wait a minute!" test (von Fintel, 2004)

Major research question: what grammatical properties of an expression give rise to its presuppositions?

The empirical observation

How do we identity presuppositions?

- family-of-sentence tests (Chierchia and McConnell-Ginet, 2000)
- "hey, wait a minute!" test (von Fintel, 2004)

Major research question: what grammatical properties of an expression give rise to its presuppositions?

A compositional account answers two questions:

How do we identity presuppositions?

- family-of-sentence tests (Chierchia and McConnell-Ginet, 2000)
- "hey, wait a minute!" test (von Fintel, 2004)

Major research question: what grammatical properties of an expression give rise to its presuppositions?

A compositional account answers two questions:

- How do we grammatically encode presuppositions in simple expressions (presupposition triggers)?

The empirical observation

How do we identity presuppositions?

- family-of-sentence tests (Chierchia and McConnell-Ginet, 2000)
- "hey, wait a minute!" test (von Fintel, 2004)

Major research question: what grammatical properties of an expression give rise to its presuppositions?

A compositional account answers two questions:

- How do we grammatically encode presuppositions in simple expressions (presupposition triggers)?
- How do presuppositions project in complex expressions?

How do we identity presuppositions?

- family-of-sentence tests (Chierchia and McConnell-Ginet, 2000)
- "hey, wait a minute!" test (von Fintel, 2004)

Major research question: what grammatical properties of an expression give rise to its presuppositions?

A compositional account answers two questions:

- How do we grammatically encode presuppositions in simple expressions (presupposition triggers)?
- How do presuppositions project in complex expressions?
  - The "projection problem" (Langendoen and Savin, 1971)

Today's talk

Outline:

Today's talk

Outline:

- Investigate an influential compositional framework for studying presupposition projection: "satisfaction theory" (Geurts, 1996)

Today's talk

Outline:

- Investigate an influential compositional framework for studying presupposition projection: "satisfaction theory" (Geurts, 1996)
  - Heim 1983

Today's talk

Outline:

- Investigate an influential compositional framework for studying presupposition projection: "satisfaction theory" (Geurts, 1996)
  - Heim 1983
  - compositionally derived conditions on dynamic update $\rightsquigarrow$ presuppositions

Outline:

- Investigate an influential compositional framework for studying presupposition projection: "satisfaction theory" (Geurts, 1996)
  - Heim 1983
  - compositionally derived conditions on dynamic update $\rightsquigarrow$ presuppositions
  - the "proviso" problem

Today's talk

Outline:

- Investigate an influential compositional framework for studying presupposition projection: "satisfaction theory" (Geurts, 1996)
  - Heim 1983
  - compositionally derived conditions on dynamic update
    $\rightsquigarrow$ presuppositions
  - the "proviso" problem
- Reconsider the satisfaction account in light of a scope-taking mechanism

Outline:

- Investigate an influential compositional framework for studying presupposition projection: "satisfaction theory" (Geurts, 1996)
  - Heim 1983
  - compositionally derived conditions on dynamic update $\rightsquigarrow$ presuppositions
  - the "proviso" problem
- Reconsider the satisfaction account in light of a scope-taking mechanism
  - Explore predictions

Outline:

- Investigate an influential compositional framework for studying presupposition projection: "satisfaction theory" (Geurts, 1996)
  - Heim 1983
  - compositionally derived conditions on dynamic update $\rightsquigarrow$ presuppositions
  - the "proviso" problem
- Reconsider the satisfaction account in light of a scope-taking mechanism
  - Explore predictions
- Presupposition triggers in the scopes of propositional attitude verbs

We are here

1. The satisfaction theory

2. A scopal account

Rough sketch of how it works

- Basic ideas come from Heim 1983

Rough sketch of how it works

- Basic ideas come from Heim 1983
- Sentences denote *context change potentials*

Rough sketch of how it works

- Basic ideas come from Heim 1983
- Sentences denote *context change potentials*

$$[\![\Delta \quad ; \quad \textit{it's raining}]\!]$$

Rough sketch of how it works

- Basic ideas come from Heim 1983
- Sentences denote *context change potentials*

$$\begin{aligned} & [\![\Delta \quad ; \quad \textit{it's raining}]\!] \\ = \;& [\![\Delta]\!] + [\![\textit{it's raining}]\!] \end{aligned}$$

Rough sketch of how it works

- Basic ideas come from Heim 1983
- Sentences denote *context change potentials*

$$
\begin{aligned}
& [\![\Delta \;\; ; \;\; \textit{it's raining}]\!] \\
= \; & [\![\Delta]\!] + [\![\textit{it's raining}]\!] \\
\text{e.g.,} \quad = \; & \{w \in \mathcal{W} \mid [\![\Delta]\!]^w = 1\} \; \cap \; \{w \in \mathcal{W} \mid \mathrm{rain}\, w\}
\end{aligned}
$$

Rough sketch of how it works

- Basic ideas come from Heim 1983
- Sentences denote *context change potentials*

$$
\begin{aligned}
& [\![\Delta \;\; ; \;\; \textit{it's raining}]\!] \\
= \;& [\![\Delta]\!] + [\![\textit{it's raining}]\!] \\
\text{e.g.,} \quad = \;& \{w \in \mathcal{W} \mid [\![\Delta]\!]^w = 1\} \;\cap\; \{w \in \mathcal{W} \mid \mathrm{rain}\, w\}
\end{aligned}
$$

- What is +? (Depends on your more specific theory.)

Rough sketch of how it works

- Basic ideas come from Heim 1983
- Sentences denote *context change potentials*

$$
\begin{aligned}
& [\![\Delta \;\; ; \;\; \textit{it's raining}]\!] \\
& = \; [\![\Delta]\!] + [\![\textit{it's raining}]\!] \\
\text{e.g.,} \quad & = \; \{w \in \mathcal{W} \mid [\![\Delta]\!]^{w} = 1\} \;\cap\; \{w \in \mathcal{W} \mid \text{rain}\, w\}
\end{aligned}
$$

- What is +? (Depends on your more specific theory.)
    - Might amount to set intersection (of sets of worlds, assignments, ...)

Rough sketch of how it works

- What if the sentence updating Δ has presuppositions?

Rough sketch of how it works

- What if the sentence updating Δ has presuppositions?

$$[\![\Delta]\!] + [\![\textit{Karlos brought his car}]\!]$$

Rough sketch of how it works

- What if the sentence updating Δ has presuppositions?

$$[\![\Delta]\!] + [\![\textit{Karlos brought his car}]\!]$$

- Δ "admits" *Karlos brought his car* only if Δ entails *Karlos has a car.*

Rough sketch of how it works

- What if the sentence updating Δ has presuppositions?

$$[\![\Delta]\!] + [\![\textit{Karlos brought his car}]\!]$$

- Δ "admits" *Karlos brought his car* only if Δ entails *Karlos has a car.*
  - "Stalnaker's bridge" (von Fintel, 2008)

Rough sketch of how it works

- What if the sentence updating Δ has presuppositions?

$$[\![\Delta]\!] + [\![\textit{Karlos brought his car}]\!]$$

- Δ “admits” *Karlos brought his car* only if Δ entails *Karlos has a car.*
  - “Stalnaker’s bridge” (von Fintel, 2008)
- Foregoing assumptions are meant to provide a way of determining what a sentence’s presuppositions *are*:

Rough sketch of how it works

- What if the sentence updating Δ has presuppositions?

$$[\![\Delta]\!] + [\![\textit{Karlos brought his car}]\!]$$

- Δ "admits" *Karlos brought his car* only if Δ entails *Karlos has a car.*
  - "Stalnaker's bridge" (von Fintel, 2008)
- Foregoing assumptions are meant to provide a way of determining what a sentence's presuppositions *are*:
  - S_1 presupposes S_2 iff every context Δ, such that $[\![\Delta]\!] + [\![S_1]\!]$ is successful, entails S_2.

Rough sketch of how it works

- What if the sentence updating Δ has presuppositions?

$$[\![\Delta]\!] + [\![\textit{Karlos brought his car}]\!]$$

- Δ "admits" *Karlos brought his car* only if Δ entails *Karlos has a car.*
  - "Stalnaker's bridge" (von Fintel, 2008)
- Foregoing assumptions are meant to provide a way of determining what a sentence's presuppositions *are*:
  - S_1 presupposes S_2 iff every context Δ, such that $[\![\Delta]\!] + [\![S_1]\!]$ is successful, entails S_2.
  - *Karlos brought his car* $\rightsquigarrow$ *Karlos has a car*

Rough sketch of how it works

Explaining projection behavior: just a matter of using + in the right way.

1. Karlos has a car, and he brought his car.

Rough sketch of how it works

Explaining projection behavior: just a matter of using + in the right way.

1. Karlos has a car, and he brought his car.
   - $\Delta + [\![(1)]\!] = (\Delta + [\![\textit{K has a car}]\!]) + [\![\textit{K brought his car}]\!]$

Rough sketch of how it works

Explaining projection behavior: just a matter of using + in the right way.

1. Karlos has a car, and he brought his car.
   - $\Delta + [\![(1)]\!] = (\Delta + [\![\textit{K has a car}]\!]) + [\![\textit{K brought his car}]\!]$
   - Update is successful if each of the individual updates is successful.

Rough sketch of how it works

Explaining projection behavior: just a matter of using + in the right way.

1. Karlos has a car, and he brought his car.
   - $\Delta + [\![(1)]\!] = (\Delta + [\![\textit{K has a car}]\!]) + [\![\textit{K brought his car}]\!]$
   - Update is successful if each of the individual updates is successful.
   - … if Δ entails *if Karlos has a car, then Karlos has a car*

Rough sketch of how it works

Explaining projection behavior: just a matter of using + in the right way.

1. Karlos has a car, and he brought his car.
    - $\Delta + [\![(1)]\!] = (\Delta + [\![\textit{K has a car}]\!]) + [\![\textit{K brought his car}]\!]$
    - Update is successful if each of the individual updates is successful.
    - … if Δ entails *if Karlos has a car, then Karlos has a car*
        - $\rightsquigarrow$ No presuppositions for (1).

Rough sketch of how it works

Explaining projection behavior: just a matter of using + in the right way.

1. Karlos has a car, and he brought his car.
   - $\Delta + [\![(1)]\!] = (\Delta + [\![\textit{K has a car}]\!]) + [\![\textit{K brought his car}]\!]$
   - Update is successful if each of the individual updates is successful.
   - … if Δ entails *if Karlos has a car, then Karlos has a car*
     - $\rightsquigarrow$ No presuppositions for (1).
     - I'll say that the presupposition in (1) is "filtered".

Rough sketch of how it works

Explaining projection behavior: just a matter of using + in the right way.

1. Karlos has a car, and he brought his car.
   - $\Delta + [\![(1)]\!] = (\Delta + [\![\textit{K has a car}]\!]) + [\![\textit{K brought his car}]\!]$
   - Update is successful if each of the individual updates is successful.
   - … if Δ entails *if Karlos has a car, then Karlos has a car*
     - $\rightsquigarrow$ No presuppositions for (1).
     - I'll say that the presupposition in (1) is "filtered".
2. If Karlos has a car, he brought his car.

Explaining projection behavior: just a matter of using + in the right way.

1. Karlos has a car, and he brought his car.
    - $\Delta + [\![(1)]\!] = (\Delta + [\![\textit{K has a car}]\!]) + [\![\textit{K brought his car}]\!]$
    - Update is successful if each of the individual updates is successful.
    - … if Δ entails *if Karlos has a car, then Karlos has a car*
        - $\rightsquigarrow$ No presuppositions for (1).
        - I'll say that the presupposition in (1) is "filtered".
2. If Karlos has a car, he brought his car.
    - $\Delta + [\![(2)]\!]$:

Rough sketch of how it works

Explaining projection behavior: just a matter of using + in the right way.

1. Karlos has a car, and he brought his car.
   - $\Delta + [\![(1)]\!] = (\Delta + [\![\textit{K has a car}]\!]) + [\![\textit{K brought his car}]\!]$
   - Update is successful if each of the individual updates is successful.
   - … if Δ entails *if Karlos has a car, then Karlos has a car*
     - ⋆ $\rightsquigarrow$ No presuppositions for (1).
     - ⋆ I'll say that the presupposition in (1) is "filtered".
2. If Karlos has a car, he brought his car.
   - $\Delta + [\![(2)]\!]$:
     - ⋆ $\Delta_1 = \Delta + [\![\textit{K has c}]\!]$

Explaining projection behavior: just a matter of using + in the right way.

1. Karlos has a car, and he brought his car.
   - $\Delta + [\![(1)]\!] = (\Delta + [\![\textit{K has a car}]\!]) + [\![\textit{K brought his car}]\!]$
   - Update is successful if each of the individual updates is successful.
   - … if Δ entails *if Karlos has a car, then Karlos has a car*
     - $\rightsquigarrow$ No presuppositions for (1).
     - I'll say that the presupposition in (1) is "filtered".
2. If Karlos has a car, he brought his car.
   - $\Delta + [\![(2)]\!]$:
     - $\Delta_1 = \Delta + [\![\textit{K has c}]\!]$
     - $\Delta_2 = \Delta + [\![\textit{K has c}]\!] + [\![\textit{K brought c}]\!]$

Rough sketch of how it works

Explaining projection behavior: just a matter of using + in the right way.

1. Karlos has a car, and he brought his car.
   - $\Delta + [\![(1)]\!] = (\Delta + [\![\textit{K has a car}]\!]) + [\![\textit{K brought his car}]\!]$
   - Update is successful if each of the individual updates is successful.
   - … if Δ entails *if Karlos has a car, then Karlos has a car*
     - $\rightsquigarrow$ No presuppositions for (1).
     - I'll say that the presupposition in (1) is "filtered".
2. If Karlos has a car, he brought his car.
   - $\Delta + [\![(2)]\!]$:
     - $\Delta_1 = \Delta + [\![\textit{K has c}]\!]$
     - $\Delta_2 = \Delta + [\![\textit{K has c}]\!] + [\![\textit{K brought c}]\!]$
     - $= \Delta - (\Delta_1 - \Delta_2)$

Rough sketch of how it works

Explaining projection behavior: just a matter of using + in the right way.

1. Karlos has a car, and he brought his car.
   - $\Delta + [\![(1)]\!] = (\Delta + [\![\textit{K has a car}]\!]) + [\![\textit{K brought his car}]\!]$
   - Update is successful if each of the individual updates is successful.
   - ... if Δ entails *if Karlos has a car, then Karlos has a car*
     - ★ $\rightsquigarrow$ No presuppositions for (1).
     - ★ I'll say that the presupposition in (1) is "filtered".
2. If Karlos has a car, he brought his car.
   - $\Delta + [\![(2)]\!]$:
     - ★ $\Delta_1 = \Delta + [\![\textit{K has c}]\!]$
     - ★ $\Delta_2 = \Delta + [\![\textit{K has c}]\!] + [\![\textit{K brought c}]\!]$
     - ★ $= \Delta - (\Delta_1 - \Delta_2)$
   - Update is successful if each of the individual updates is.

Rough sketch of how it works

Explaining projection behavior: just a matter of using + in the right way.

1. Karlos has a car, and he brought his car.
   - $\Delta + [\![(1)]\!] = (\Delta + [\![\textit{K has a car}]\!]) + [\![\textit{K brought his car}]\!]$
   - Update is successful if each of the individual updates is successful.
   - … if Δ entails *if Karlos has a car, then Karlos has a car*
     - ★ $\rightsquigarrow$ No presuppositions for (1).
     - ★ I'll say that the presupposition in (1) is "filtered".
2. If Karlos has a car, he brought his car.
   - $\Delta + [\![(2)]\!]$:
     - ★ $\Delta_1 = \Delta + [\![\textit{K has c}]\!]$
     - ★ $\Delta_2 = \Delta + [\![\textit{K has c}]\!] + [\![\textit{K brought c}]\!]$
     - ★ $= \Delta - (\Delta_1 - \Delta_2)$
   - Update is successful if each of the individual updates is.
     - ★ $\rightsquigarrow$ No presuppositions for (2).

The proviso problem

Geurts (1996): big problem!

The proviso problem

Geurts (1996): big problem!

1. It's raining, and my car is too far away.

Geurts (1996): big problem!

1. It's raining, and my car is too far away.
   - Update successful if each individual update is.

Geurts (1996): big problem!

1. It's raining, and my car is too far away.
   - Update successful if each individual update is.
     - ★ $\Delta + [\![(1)]\!] = (\Delta + [\![\textit{it's raining}]\!]) + [\![\textit{my car is too far away}]\!]$

Geurts (1996): big problem!

1. It's raining, and my car is too far away.
   - Update successful if each individual update is.
     - $\Delta + [\![(1)]\!] = (\Delta + [\![\textit{it's raining}]\!]) + [\![\textit{my car is too far away}]\!]$
     - ...iff the context entails *if it's raining, I have a car*

The proviso problem

Geurts (1996): big problem!

1. It's raining, and my car is too far away.
   - Update successful if each individual update is.
     - ★ $\Delta + [\![(1)]\!] = (\Delta + [\![\textit{it's raining}]\!]) + [\![\textit{my car is too far away}]\!]$
     - ★ …iff the context entails *if it's raining, I have a car*
     - ★ (1) $\rightsquigarrow$ *if it's raining, I have a car* ☹

The proviso problem

Geurts (1996): big problem!

1. It's raining, and my car is too far away.
   - Update successful if each individual update is.
     - $\Delta + [\![(1)]\!] = (\Delta + [\![\textit{it's raining}]\!]) + [\![\textit{my car is too far away}]\!]$
     - …iff the context entails *if it's raining, I have a car*
     - (1) $\rightsquigarrow$ *if it's raining, I have a car* ☹
2. If the airport is nearby, I can pick my sister up when she lands.

The proviso problem

Geurts (1996): big problem!

1. It's raining, and my car is too far away.
   - Update successful if each individual update is.
     - $\Delta + [\![(1)]\!] = (\Delta + [\![\textit{it's raining}]\!]) + [\![\textit{my car is too far away}]\!]$
     - …iff the context entails *if it's raining, I have a car*
     - (1) $\rightsquigarrow$ *if it's raining, I have a car* ☹
2. If the airport is nearby, I can pick my sister up when she lands.
   - Individual updates to Δ:

The proviso problem

Geurts (1996): big problem!

1. It's raining, and my car is too far away.
   - Update successful if each individual update is.
     - $\Delta + [\![(1)]\!] = (\Delta + [\![\textit{it's raining}]\!]) + [\![\textit{my car is too far away}]\!]$
     - …iff the context entails *if it's raining, I have a car*
     - (1) $\rightsquigarrow$ *if it's raining, I have a car* ☹
2. If the airport is nearby, I can pick my sister up when she lands.
   - Individual updates to Δ:
     - $\Delta_1 = \Delta + [\![\textit{airport nearby}]\!]$

Geurts (1996): big problem!

1. It's raining, and my car is too far away.
   - Update successful if each individual update is.
     - ★ $\Delta + [\![(1)]\!] = (\Delta + [\![\textit{it's raining}]\!]) + [\![\textit{my car is too far away}]\!]$
     - ★ …iff the context entails *if it's raining, I have a car*
     - ★ (1) $\rightsquigarrow$ *if it's raining, I have a car* ☹
2. If the airport is nearby, I can pick my sister up when she lands.
   - Individual updates to Δ:
     - ★ $\Delta_1 = \Delta + [\![\textit{airport nearby}]\!]$
     - ★ $\Delta_2 = \Delta + [\![\textit{airport nearby}]\!] + [\![\textit{pick sister up}]\!]$

The proviso problem

Geurts (1996): big problem!

1. It's raining, and my car is too far away.
   - Update successful if each individual update is.
     - $\Delta + [\![(1)]\!] = (\Delta + [\![\textit{it's raining}]\!]) + [\![\textit{my car is too far away}]\!]$
     - …iff the context entails *if it's raining, I have a car*
     - (1) $\rightsquigarrow$ *if it's raining, I have a car* ☹
2. If the airport is nearby, I can pick my sister up when she lands.
   - Individual updates to Δ:
     - $\Delta_1 = \Delta + [\![\textit{airport nearby}]\!]$
     - $\Delta_2 = \Delta + [\![\textit{airport nearby}]\!] + [\![\textit{pick sister up}]\!]$
     - $= \Delta - (\Delta_1 - \Delta_2)$

Geurts (1996): big problem!

1. It's raining, and my car is too far away.
   - Update successful if each individual update is.
     - $\Delta + [\![(1)]\!] = (\Delta + [\![\textit{it's raining}]\!]) + [\![\textit{my car is too far away}]\!]$
     - ...iff the context entails *if it's raining, I have a car*
     - (1) $\rightsquigarrow$ *if it's raining, I have a car* ☹
2. If the airport is nearby, I can pick my sister up when she lands.
   - Individual updates to Δ:
     - $\Delta_1 = \Delta + [\![\textit{airport nearby}]\!]$
     - $\Delta_2 = \Delta + [\![\textit{airport nearby}]\!] + [\![\textit{pick sister up}]\!]$
     - $= \Delta - (\Delta_1 - \Delta_2)$
   - (2) $\rightsquigarrow$ *if the airport is nearby, I have a sister* ☹

The proviso problem

- According to the satisfaction theory, filtration is automatic.

The proviso problem

- According to the satisfaction theory, filtration is automatic.
- But sometimes it shouldn’t happen.

We are here

1. The satisfaction theory

2. A scopal account

Compositional semantics

We can encode an expression's presuppositions by allowing meanings to be partial.

Compositional semantics

We can encode an expression's presuppositions by allowing meanings to be partial.

- $[\![\textit{his car}]\!] = \lambda g^{\gamma}.\begin{cases} c & c \text{ is the car of } \texttt{sel}\,g \\ \# & \text{no such car exists} \end{cases}$

Compositional semantics

We can encode an expression's presuppositions by allowing meanings to be partial.

- $[\![\textit{his car}]\!] = \lambda g^{\gamma}.\begin{cases} c & c \text{ is the car of } \texttt{sel}\,g \\ \# & \text{no such car exists} \end{cases}$

Compositional semantics

We can encode an expression's presuppositions by allowing meanings to be partial.

- $[\![\textit{his car}]\!] = \lambda g^{\gamma}.\begin{cases} c & c \text{ is the car of } \texttt{sel}g \\ \# & \text{no such car exists} \end{cases} = \lambda g^{\gamma}.\text{car\_of}(\texttt{sel}g)$

Compositional semantics

We can encode an expression's presuppositions by allowing meanings to be partial.

- $[\![\textit{his car}]\!] = \lambda g^{\gamma}.\begin{cases} c & c \text{ is the car of } \texttt{sel}g \\ \# & \text{no such car exists} \end{cases} = \lambda g^{\gamma}.\text{car\_of}(\texttt{sel}g)$

We can allow this by sophisticating the type system a little bit.

We can encode an expression's presuppositions by allowing meanings to be partial.

- $[\![\textit{his car}]\!] = \lambda g^{\gamma}.\begin{cases} c & c \text{ is the car of } \texttt{sel}\,g \\ \# & \text{no such car exists} \end{cases} = \lambda g^{\gamma}.\text{car\_of}(\texttt{sel}\,g)$

We can allow this by sophisticating the type system a little bit.

- Simple types:

$$\mathcal{T} ::= e \mid \gamma \mid t \mid \mathcal{T} \to \mathcal{T}$$

We can encode an expression's presuppositions by allowing meanings to be partial.

- $[\![\textit{his car}]\!] = \lambda g^{\gamma}.\begin{cases} c & c \text{ is the car of } \mathtt{sel}\, g \\ \# & \text{no such car exists} \end{cases} = \lambda g^{\gamma}.\text{car\_of}(\mathtt{sel}\, g)$

We can allow this by sophisticating the type system a little bit.

- Simple types:

$$\mathcal{T} ::= e \mid \gamma \mid t \mid \mathcal{T} \to \mathcal{T}$$

- "Maybe" types:

$$\mathcal{T} ::= e \mid \gamma \mid t \mid \mathcal{T} \to \mathcal{T} \mid \mathcal{T}_{\#}$$

Compositional semantics

We can encode an expression's presuppositions by allowing meanings to be partial.

- $[\![\textit{his car}]\!] = \lambda g^{\gamma}.\begin{cases} c & c \text{ is the car of } \mathtt{sel}\,g \\ \# & \text{no such car exists} \end{cases} = \lambda g^{\gamma}.\text{car\_of}(\mathtt{sel}\,g)$

We can allow this by sophisticating the type system a little bit.

- Simple types:

$$\mathcal{T} ::= e \mid \gamma \mid t \mid \mathcal{T} \to \mathcal{T}$$

- "Maybe" types:

$$\mathcal{T} ::= e \mid \gamma \mid t \mid \mathcal{T} \to \mathcal{T} \mid \mathcal{T}_{\#}$$

  - E.g., the type $e_{\#}$ is that of something which is either an individual (e.g., Karlos) or undefined (#).

Compositional semantics

We can encode an expression's presuppositions by allowing meanings to be partial.

- $[\![\textit{his car}]\!] = \lambda g^{\gamma}.\begin{cases} c & c \text{ is the car of } \texttt{sel}\,g \\ \# & \text{no such car exists} \end{cases} = \lambda g^{\gamma}.\text{car\_of}(\texttt{sel}\,g)$

We can allow this by sophisticating the type system a little bit.

- Simple types:

$$\mathcal{T} ::= e \mid \gamma \mid t \mid \mathcal{T} \to \mathcal{T}$$

- "Maybe" types:

$$\mathcal{T} ::= e \mid \gamma \mid t \mid \mathcal{T} \to \mathcal{T} \mid \mathcal{T}_{\#}$$

  - E.g., the type $e_{\#}$ is that of something which is either an individual (e.g., Karlos) or undefined (#).
  - This move allows us to treat partial functions as total; e.g., a partial function of type $e \to t$ is now a total function of type $e \to t_{\#}$ that maps the part of its domain on which it is not defined to #.

Compositional semantics

Meanings will often be context-dependent and possibly undefined.

Compositional semantics

Meanings will often be context-dependent and possibly undefined.

- i.e., of type $\gamma \rightarrow \alpha_{\#}$ for some type α

Compositional semantics

Meanings will often be context-dependent and possibly undefined.

- i.e., of type $\gamma \longrightarrow \alpha_{\#}$ for some type α

Compositional semantics

Meanings will often be context-dependent and possibly undefined.

- i.e., of type $\gamma \to \alpha_{\#}$ for some type α

1. Karlos brought his car.

Meanings will often be context-dependent and possibly undefined.

- i.e., of type $\gamma \rightarrow \alpha_{\#}$ for some type α

1. Karlos brought his car.

   - *Functional Application* (*FA*)

$$\lambda g^{\gamma}.\begin{cases} mg(ng) & \# \notin \{mg, ng\} \\ \# & \# \in \{mg, ng\} \end{cases}$$

$m \qquad n$

or

m n

m n

Meanings will often be context-dependent and possibly undefined.

- i.e., of type $\gamma \longrightarrow \alpha_{\#}$ for some type α

1. Karlos brought his car.

   - *Functional Application* (*FA*)

$$\lambda g^{\gamma}.\begin{cases} mg(ng) & \# \notin \{mg, ng\} \\ \# & \# \in \{mg, ng\} \end{cases}$$

$m \qquad n$

or

m n

m $\qquad$ n

   - $[\![$*Karlos brought his car*$]\!]$ =

Meanings will often be context-dependent and possibly undefined.

- i.e., of type $\gamma \rightarrow \alpha_{\#}$ for some type α

1. Karlos brought his car.

   - *Functional Application* (*FA*)

     $$\lambda g^{\gamma}.\begin{cases} mg(ng) & \# \notin \{mg, ng\} \\ \# & \# \in \{mg, ng\} \end{cases}$$

     $m \quad n$

     or

     m n

     m n

   - $[\![\textit{Karlos brought his car}]\!] =$
     - $FA(FA[\![\textit{brought}]\!][\![\textit{his car}]\!])[\![\textit{Karlos}]\!] =$

Meanings will often be context-dependent and possibly undefined.

- i.e., of type $\gamma \rightarrow \alpha_{\#}$ for some type α

1. Karlos brought his car.

   - *Functional Application* (*FA*)

$$\lambda g^{\gamma}.\begin{cases} mg(ng) & \# \notin \{mg, ng\} \\ \# & \# \in \{mg, ng\} \end{cases}$$

$$m \qquad n$$

or

m n

m n

   - $[\![\mathit{Karlos\ brought\ his\ car}]\!] =$
     - ★ $FA(FA[\![\mathit{brought}]\!][\![\mathit{his\ car}]\!])[\![\mathit{Karlos}]\!] =$
     - ★ $FA(FA(\lambda g^{\gamma}.\text{brought})[\![\mathit{his\ car}]\!])(\lambda g^{\gamma}.\text{k}) =$

Meanings will often be context-dependent and possibly undefined.

- i.e., of type $\gamma \rightarrow \alpha_{\#}$ for some type α

1. Karlos brought his car.

   - *Functional Application* (*FA*)

$$\lambda g^{\gamma}.\begin{cases} mg(ng) & \# \notin \{mg, ng\} \\ \# & \# \in \{mg, ng\} \end{cases}$$

$m \quad n$

or

m n

m n

   - $[\![$*Karlos brought his car*$]\!]$ =
     - $FA(FA[\![brought]\!][\![his\ car]\!])[\![Karlos]\!]$ =
     - $FA(FA(\lambda g^{\gamma}.\text{brought})[\![his\ car]\!])(\lambda g^{\gamma}.\text{k})$ =
     - $\lambda g^{\gamma}.\begin{cases} \text{brought}ck & \texttt{car\_of}(\texttt{sel}g) = c \\ \# & \texttt{car\_of}(\texttt{sel}g) = \# \end{cases}$

Compositional semantics: an analysis of conditionals

1. If Karlos has a car, Karlos brought his car.

1. If Karlos has a car, Karlos brought his car.

- $$[\![if]\!] = \lambda p^{\gamma \to t_\#}, q^{\gamma \to t_\#}, g^\gamma . \begin{cases} 1 & pg = 0 \\ 1 & pg = 1 \wedge qg = 1 \\ 0 & pg = 1 \wedge qg = 0 \\ \# & pg = \# \\ \# & pg = 1 \wedge qg = \# \end{cases}$$

Compositional semantics: an analysis of conditionals

1. If Karlos has a car, Karlos brought his car.

- $[\![if]\!] = \lambda p^{\gamma \to t_\#}, q^{\gamma \to t_\#}, g^\gamma . \begin{cases} 1 & pg = 0 \\ 1 & pg = 1 \wedge qg = 1 \\ 0 & pg = 1 \wedge qg = 0 \\ \# & pg = \# \\ \# & pg = 1 \wedge qg = \# \end{cases}$
- $[\![$*if Karlos has a car, Karlos brought his$_i$ car*$]\!]$

Compositional semantics: an analysis of conditionals

1. If Karlos has a car, Karlos brought his car.

- $[\![if]\!] = \lambda p^{\gamma \to t_\#}, q^{\gamma \to t_\#}, g^\gamma . \begin{cases} 1 & pg = 0 \\ 1 & pg = 1 \wedge qg = 1 \\ 0 & pg = 1 \wedge qg = 0 \\ \# & pg = \# \\ \# & pg = 1 \wedge qg = \# \end{cases}$

- $[\![$*if Karlos has a car, Karlos brought his$_i$ car*$]\!]$
  - ★ $\lambda g^\gamma . \begin{cases} 1 & [\![K\ has\ a\ car]\!]g = 0 \\ 1 & [\![K\ has\ a\ car]\!]g = 1 \wedge [\![K\ brought\ his\ car]\!]g = 1 \\ 0 & [\![K\ has\ a\ car]\!]g = 1 \wedge [\![K\ brought\ his\ car]\!]g = 0 \\ \# & [\![K\ has\ a\ car]\!]g = \# \\ \# & [\![K\ has\ a\ car]\!]g = 1 \wedge [\![K\ brought\ his\ car]\!]g = \# \end{cases}$

1. If Karlos has a car, Karlos brought his car.

- $$[\![if]\!] = \lambda p^{\gamma \to t_\#}, q^{\gamma \to t_\#}, g^\gamma . \begin{cases} 1 & pg = 0 \\ 1 & pg = 1 \wedge qg = 1 \\ 0 & pg = 1 \wedge qg = 0 \\ \# & pg = \# \\ \# & pg = 1 \wedge qg = \# \end{cases}$$

- $[\![\textit{if Karlos has a car, Karlos brought his}_i \textit{ car}]\!]$
  - $$\lambda g^\gamma . \begin{cases} 1 & [\![\textit{K has a car}]\!]g = 0 \\ 1 & [\![\textit{K has a car}]\!]g = 1 \wedge [\![\textit{K brought his car}]\!]g = 1 \\ 0 & [\![\textit{K has a car}]\!]g = 1 \wedge [\![\textit{K brought his car}]\!]g = 0 \\ \# & [\![\textit{K has a car}]\!]g = \# \\ \# & [\![\textit{K has a car}]\!]g = 1 \wedge [\![\textit{K brought his car}]\!]g = \# \end{cases}$$
  - $\rightsquigarrow$ no presupposition ☺

- $[\![if]\!] = \lambda p^{\gamma \to t_\#}, q^{\gamma \to t_\#}, g^\gamma . \begin{cases} 1 & pg = 0 \\ 1 & pg = 1 \wedge qg = 1 \\ 0 & pg = 1 \wedge qg = 0 \\ \# & pg = \# \\ \# & pg = 1 \wedge qg = \# \end{cases}$

Compositional semantics: an analysis of conditionals

- $[\![if]\!] = \lambda p^{\gamma \to t_{\#}}, q^{\gamma \to t_{\#}}, g^{\gamma}. \begin{cases} 1 & pg = 0 \\ 1 & pg = 1 \wedge qg = 1 \\ 0 & pg = 1 \wedge qg = 0 \\ \# & pg = \# \\ \# & pg = 1 \wedge qg = \# \end{cases}$
- The semantic type of the conditional connective is $(\gamma \to t_{\#}) \to (\gamma \to t_{\#}) \to \gamma \to t_{\#}$.

Compositional semantics: an analysis of conditionals

- $[\![if]\!] = \lambda p^{\gamma \to t_\#}, q^{\gamma \to t_\#}, g^\gamma . \begin{cases} 1 & pg = 0 \\ 1 & pg = 1 \land qg = 1 \\ 0 & pg = 1 \land qg = 0 \\ \# & pg = \# \\ \# & pg = 1 \land qg = \# \end{cases}$
- The semantic type of the conditional connective is $(\gamma \to t_\#) \to (\gamma \to t_\#) \to \gamma \to t_\#$.
  - The high type of the conditional has the Maybe type $t_\#$ occur in a negative position, allowing the conditional to decide the result based on the definedness of its antecedent and its consequent.

A proviso problem crops up in this setting, as well.

A proviso problem crops up in this setting, as well.

1. If the airport is nearby, I'll pick my sister up when she lands.

A proviso problem crops up in this setting, as well.

1. If the airport is nearby, I'll pick my sister up when she lands.
   - $[\![$*airport nearby, pick up sister*$]\!]$

A proviso problem crops up in this setting, as well.

1. If the airport is nearby, I'll pick my sister up when she lands.

   - $[\![\textit{airport nearby, pick up sister}]\!]$
     - $\lambda g^{\gamma}.\begin{cases} 1 & [\![\textit{airport nearby}]\!]g = 0 \\ 1 & [\![\textit{airport nearby}]\!]g = 1 \wedge [\![\textit{pick up sister}]\!]g = 1 \\ 0 & [\![\textit{airport nearby}]\!]g = 1 \wedge [\![\textit{pick up sister}]\!]g = 0 \\ \# & [\![\textit{airport nearby}]\!]g = \# \\ \# & [\![\textit{airport nearby}]\!]g = 1 \wedge [\![\textit{pick up sister}]\!]g = \# \end{cases}$

A proviso problem crops up in this setting, as well.

1. If the airport is nearby, I'll pick my sister up when she lands.

   - $[\![\textit{airport nearby, pick up sister}]\!]$
     - $\lambda g^{\gamma}.\begin{cases} 1 & [\![\textit{airport nearby}]\!]g = 0 \\ 1 & [\![\textit{airport nearby}]\!]g = 1 \wedge [\![\textit{pick up sister}]\!]g = 1 \\ 0 & [\![\textit{airport nearby}]\!]g = 1 \wedge [\![\textit{pick up sister}]\!]g = 0 \\ \# & [\![\textit{airport nearby}]\!]g = \# \\ \# & [\![\textit{airport nearby}]\!]g = 1 \wedge [\![\textit{pick up sister}]\!]g = \# \end{cases}$
     - $\rightsquigarrow$ *if the airport is nearby, I have a sister* ☹

Summary

- The introduction of Maybe types allows for a semantics of undefinedness.

Summary

- The introduction of Maybe types allows for a semantics of undefinedness.
- The meanings of presupposition filters (e.g., conditionals) can be stated by having them be sensitive to values of such types.

Summary

- The introduction of Maybe types allows for a semantics of undefinedness.
- The meanings of presupposition filters (e.g., conditionals) can be stated by having them be sensitive to values of such types.
- Presupposition projection and Functional Application go hand-in-hand.

Summary

- The introduction of Maybe types allows for a semantics of undefinedness.
- The meanings of presupposition filters (e.g., conditionals) can be stated by having them be sensitive to values of such types.
- Presupposition projection and Functional Application go hand-in-hand.
    - $\rightsquigarrow$ proviso problem

- *Functional Application*

$$\lambda g^{\gamma}.\begin{cases} mg(ng) & \# \notin \{mg, ng\} \\ \# & \# \in \{mg, ng\} \end{cases}$$

$m \qquad n$

or

m n

m n

Note

- *Functional Application*

$$\lambda g^{\gamma}.\begin{cases} mg(ng) & \# \notin \{mg, ng\} \\ \# & \# \in \{mg, ng\} \end{cases}$$

$m \quad n$

or

m n

m n

- This composition strategy is an instance of a certain structure from functional programming (and category theory):

Note

- *Functional Application*

$$\lambda g^{\gamma}.\begin{cases} mg(ng) & \# \notin \{mg, ng\} \\ \# & \# \in \{mg, ng\} \end{cases}$$

$$m \qquad n$$

or

m n

m n

- This composition strategy is an instance of a certain structure from functional programming (and category theory):
  - an Applicative Functor (McBride and Paterson, 2008)

Note

- *Functional Application*

$$\lambda g^{\gamma}.\begin{cases} mg(ng) & \# \notin \{mg, ng\} \\ \# & \# \in \{mg, ng\} \end{cases}$$

$m \quad n$

or

m n

m n

- This composition strategy is an instance of a certain structure from functional programming (and category theory):
  - an Applicative Functor (McBride and Paterson, 2008)
    - a Functor $\mathcal{F} : \mathcal{T} \to \mathcal{T}$

Note

- *Functional Application*

$$\lambda g^{\gamma}.\begin{cases} mg(ng) & \# \notin \{mg, ng\} \\ \# & \# \in \{mg, ng\} \end{cases}$$

m $\quad$ n

or

m n

m $\quad$ n

- This composition strategy is an instance of a certain structure from functional programming (and category theory):
  - an Applicative Functor (McBride and Paterson, 2008)
    - a Functor $\mathcal{F} : \mathcal{T} \to \mathcal{T}$
    - an operator $(\circledast) : \mathcal{F}(\alpha \to \beta) \to \mathcal{F}(\alpha) \to \mathcal{F}(\beta)$ ('apply')

Note

- *Functional Application*

$$\lambda g^{\gamma}.\begin{cases} mg(ng) & \# \notin \{mg, ng\} \\ \# & \# \in \{mg, ng\} \end{cases}$$

m *n*

or

m n

m n

- This composition strategy is an instance of a certain structure from functional programming (and category theory):
  - an Applicative Functor (McBride and Paterson, 2008)
    - ★ a Functor $\mathcal{F} : \mathcal{T} \to \mathcal{T}$
    - ★ an operator $(\circledast) : \mathcal{F}(\alpha \to \beta) \to \mathcal{F}(\alpha) \to \mathcal{F}(\beta)$ ('apply')
    - ★ an injection function $\eta : \alpha \to \mathcal{F}\alpha$ ('return')

Note

- *Functional Application*

$$\lambda g^{\gamma}.\begin{cases} mg(ng) & \# \notin \{mg, ng\} \\ \# & \# \in \{mg, ng\} \end{cases}$$

$m \qquad n$

or

m n

m n

- This composition strategy is an instance of a certain structure from functional programming (and category theory):
  - an Applicative Functor (McBride and Paterson, 2008)
    - a Functor $\mathcal{F} : \mathcal{T} \rightarrow \mathcal{T}$
    - an operator $(\circledast) : \mathcal{F}(\alpha \rightarrow \beta) \rightarrow \mathcal{F}(\alpha) \rightarrow \mathcal{F}(\beta)$ ('apply')
    - an injection function $\eta : \alpha \rightarrow \mathcal{F}\alpha$ ('return')
  - Here:

Note

- *Functional Application*

$$\lambda g^{\gamma}.\begin{cases} mg(ng) & \# \notin \{mg, ng\} \\ \# & \# \in \{mg, ng\} \end{cases}$$

m *n*

or

m n

m n

- This composition strategy is an instance of a certain structure from functional programming (and category theory):
  - an Applicative Functor (McBride and Paterson, 2008)
    - a Functor $\mathcal{F} : \mathcal{T} \to \mathcal{T}$
    - an operator $(\circledast) : \mathcal{F}(\alpha \to \beta) \to \mathcal{F}(\alpha) \to \mathcal{F}(\beta)$ (‘apply’)
    - an injection function $\eta : \alpha \to \mathcal{F}\alpha$ (‘return’)
  - Here:
    - $\mathcal{F}(\alpha) = \gamma \to \alpha_{\#}$

Note

- *Functional Application*

  $$\lambda g^{\gamma}.\begin{cases} mg(ng) & \# \notin \{mg, ng\} \\ \# & \# \in \{mg, ng\} \end{cases}$$

  $m \qquad n$

  or

  m n

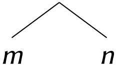
  m n

- This composition strategy is an instance of a certain structure from functional programming (and category theory):
  - an Applicative Functor (McBride and Paterson, 2008)
    - a Functor $\mathcal{F} : \mathcal{T} \rightarrow \mathcal{T}$
    - an operator $(\circledast) : \mathcal{F}(\alpha \rightarrow \beta) \rightarrow \mathcal{F}(\alpha) \rightarrow \mathcal{F}(\beta)$ ('apply')
    - an injection function $\eta : \alpha \rightarrow \mathcal{F}\alpha$ ('return')
  - Here:
    - $\mathcal{F}(\alpha) = \gamma \rightarrow \alpha_{\#}$
    - $m \circledast n = \lambda g^{\gamma}.\begin{cases} mg(ng) & \# \notin \{mg, ng\} \\ \# & \# \in \{mg, ng\} \end{cases}$ (Functional Application)

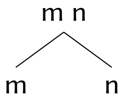
Note

- *Functional Application*

$$\lambda g^{\gamma}.\begin{cases} mg(ng) & \# \notin \{mg, ng\} \\ \# & \# \in \{mg, ng\} \end{cases}$$



or



- This composition strategy is an instance of a certain structure from functional programming (and category theory):
  - an Applicative Functor (McBride and Paterson, 2008)
    - a Functor $\mathcal{F} : \mathcal{T} \to \mathcal{T}$
    - an operator $(\circledast) : \mathcal{F}(\alpha \to \beta) \to \mathcal{F}(\alpha) \to \mathcal{F}(\beta)$ ('apply')
    - an injection function $\eta : \alpha \to \mathcal{F}\alpha$ ('return')
  - Here:
    - $\mathcal{F}(\alpha) = \gamma \to \alpha_{\#}$
    - $m \circledast n = \lambda g^{\gamma}.\begin{cases} mg(ng) & \# \notin \{mg, ng\} \\ \# & \# \in \{mg, ng\} \end{cases}$ (Functional Application)
    - $\eta a = \lambda g^{\gamma}.a$

Note

I'll argue that what limits our system is that its composition strategy is *only* applicative.

Note

I'll argue that what limits our system is that its composition strategy is *only* applicative.

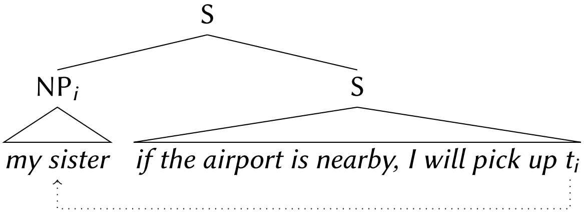
We need something more powerful: a monad.

- What a monad adds to our applicative system is a mechanism for scope-taking.

- What a monad adds to our applicative system is a mechanism for scope-taking.
- The proviso-problem arises when presupposition triggers can't take scope in the appropriate way.

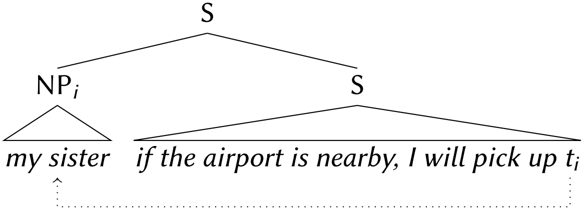
Monadic scope-taking

- What a monad adds to our applicative system is a mechanism for scope-taking.
- The proviso-problem arises when presupposition triggers can't take scope in the appropriate way.
- The "Quantifier Raising" idiom:



Monadic scope-taking

- What a monad adds to our applicative system is a mechanism for scope-taking.
- The proviso-problem arises when presupposition triggers can't take scope in the appropriate way.
- The "Quantifier Raising" idiom:



  - Where the presupposition trigger takes scope determines where its presupposition is evaluated.

- An Applicative Functor (McBride and Paterson, 2008) is

- An Applicative Functor (McBride and Paterson, 2008) is
  - a Functor $\mathcal{F} : \mathcal{T} \to \mathcal{T}$

- An Applicative Functor (McBride and Paterson, 2008) is
  - a Functor $\mathcal{F} : \mathcal{T} \to \mathcal{T}$
  - an operator $(\circledast) : \mathcal{F}(\alpha \to \beta) \to \mathcal{F}(\alpha) \to \mathcal{F}(\beta)$ ('apply')

- An Applicative Functor (McBride and Paterson, 2008) is
  - a Functor $\mathcal{F} : \mathcal{T} \to \mathcal{T}$
  - an operator $(\circledast) : \mathcal{F}(\alpha \to \beta) \to \mathcal{F}(\alpha) \to \mathcal{F}(\beta)$ ('apply')
  - an injection function $\eta : \alpha \to \mathcal{F}\alpha$ ('return')

Monadic scope-taking

- An Applicative Functor (McBride and Paterson, 2008) is
  - a Functor $\mathcal{F} : \mathcal{T} \rightarrow \mathcal{T}$
  - an operator $(\circledast) : \mathcal{F}(\alpha \rightarrow \beta) \rightarrow \mathcal{F}(\alpha) \rightarrow \mathcal{F}(\beta)$ ('apply')
  - an injection function $\eta : \alpha \rightarrow \mathcal{F}\alpha$ ('return')
- A Monad is an applicative functor with one additional operator ('join').

- An Applicative Functor (McBride and Paterson, 2008) is
  - a Functor $\mathcal{F} : \mathcal{T} \to \mathcal{T}$
  - an operator $(\circledast) : \mathcal{F}(\alpha \to \beta) \to \mathcal{F}(\alpha) \to \mathcal{F}(\beta)$ (‘apply’)
  - an injection function $\eta : \alpha \to \mathcal{F}\alpha$ (‘return’)
- A Monad is an applicative functor with one additional operator (‘join’).
  - $\mu : \mathcal{F}(\mathcal{F}(\alpha)) \to \mathcal{F}(\alpha)$

- An Applicative Functor (McBride and Paterson, 2008) is
  - a Functor $\mathcal{F} : \mathcal{T} \to \mathcal{T}$
  - an operator $(\circledast) : \mathcal{F}(\alpha \to \beta) \to \mathcal{F}(\alpha) \to \mathcal{F}(\beta)$ ('apply')
  - an injection function $\eta : \alpha \to \mathcal{F}\alpha$ ('return')
- A Monad is an applicative functor with one additional operator ('join').
  - $\mu : \mathcal{F}(\mathcal{F}(\alpha)) \to \mathcal{F}(\alpha)$
  - In our case:

- An Applicative Functor (McBride and Paterson, 2008) is
  - a Functor $\mathcal{F} : \mathcal{T} \rightarrow \mathcal{T}$
  - an operator $(\circledast) : \mathcal{F}(\alpha \rightarrow \beta) \rightarrow \mathcal{F}(\alpha) \rightarrow \mathcal{F}(\beta)$ ('apply')
  - an injection function $\eta : \alpha \rightarrow \mathcal{F}\alpha$ ('return')
- A Monad is an applicative functor with one additional operator ('join').
  - $\mu : \mathcal{F}(\mathcal{F}(\alpha)) \rightarrow \mathcal{F}(\alpha)$
  - In our case:
    - $\mu : (\gamma \rightarrow (\gamma \rightarrow \alpha_{\#})_{\#}) \rightarrow \gamma \rightarrow \alpha_{\#}$

Monadic scope-taking

- An Applicative Functor (McBride and Paterson, 2008) is
  - a Functor $\mathcal{F} : \mathcal{T} \to \mathcal{T}$
  - an operator $(\circledast) : \mathcal{F}(\alpha \to \beta) \to \mathcal{F}(\alpha) \to \mathcal{F}(\beta)$ ('apply')
  - an injection function $\eta : \alpha \to \mathcal{F}\alpha$ ('return')
- A Monad is an applicative functor with one additional operator ('join').
  - $\mu : \mathcal{F}(\mathcal{F}(\alpha)) \to \mathcal{F}(\alpha)$
  - In our case:
    - $\mu : (\gamma \to (\gamma \to \alpha_{\#})_{\#}) \to \gamma \to \alpha_{\#}$
    - $\mu m = \lambda g^{\gamma}.\begin{cases} ng & mg = n \\ \# & mg = \# \end{cases}$

Monadic scope-taking

- We'll add two compositional operations (defined in terms of the monadic interface):

- We'll add two compositional operations (defined in terms of the monadic interface):
  - Internal lift

Monadic scope-taking

- We'll add two compositional operations (defined in terms of the monadic interface):
  - Internal lift
    - ★ $(\cdot)^{\uparrow} : \mathcal{F}(\alpha) \rightarrow \mathcal{F}(\mathcal{F}(\alpha))$

Monadic scope-taking

- We'll add two compositional operations (defined in terms of the monadic interface):
  - Internal lift
    - $(\cdot)^{\uparrow} : \mathcal{F}(\alpha) \to \mathcal{F}(\mathcal{F}(\alpha))$
    - $m^{\uparrow} = \eta\eta \circledast m$

Monadic scope-taking

- We'll add two compositional operations (defined in terms of the monadic interface):
  - Internal lift
    - $(\cdot)^{\uparrow} : \mathcal{F}(\alpha) \rightarrow \mathcal{F}(\mathcal{F}(\alpha))$
    - $m^{\uparrow} = \eta\eta \circledast m$
    - Frees a presupposition trigger from its local context, so that it can take scope.

Monadic scope-taking

- We'll add two compositional operations (defined in terms of the monadic interface):
  - Internal lift
    - $(\cdot)^{\uparrow} : \mathcal{F}(\alpha) \to \mathcal{F}(\mathcal{F}(\alpha))$
    - $m^{\uparrow} = \eta\eta \circledast m$
    - Frees a presupposition trigger from its local context, so that it can take scope.
  - Join (μ)

Monadic scope-taking

- We'll add two compositional operations (defined in terms of the monadic interface):
  - Internal lift
    - $(\cdot)^{\uparrow} : \mathcal{F}(\alpha) \to \mathcal{F}(\mathcal{F}(\alpha))$
    - $m^{\uparrow} = \eta\eta \circledast m$
    - Frees a presupposition trigger from its local context, so that it can take scope.
  - Join (μ)
    - Applies to an expression with a freed presupposition trigger, in order to fix its scope.

Monadic scope-taking

- As well as make Functional Application a bit more general:

Monadic scope-taking

- As well as make Functional Application a bit more general:
  - $FA = \lambda m, n.mn \quad | \quad \lambda m, n.\eta FA \circledast m \circledast n$

- As well as make Functional Application a bit more general:
  - $FA = \lambda m, n.mn \quad | \quad \lambda m, n.\eta FA \circledast m \circledast n$
  - An example: FA sleep k = sleepk

- As well as make Functional Application a bit more general:
  - $FA = \lambda m, n.mn \quad | \quad \lambda m, n.\eta FA \circledast m \circledast n$
  - An example: FA sleep k = sleepk
  - Another example: $[\![$*Karlos's dog slept*$]\!]$ =

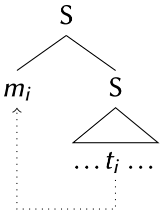
- As well as make Functional Application a bit more general:
  - $FA = \lambda m, n.mn \quad | \quad \lambda m, n.\eta FA \circledast m \circledast n$
  - An example: FA sleep k = sleepk
  - Another example: $[\![\textit{Karlos's dog slept}]\!] =$
    - $FA\, [\![\textit{slept}]\!] [\![\textit{Karlos's dog}]\!] =$

- As well as make Functional Application a bit more general:
  - $FA = \lambda m, n.mn \quad | \quad \lambda m, n.\eta FA \circledast m \circledast n$
  - An example: FA sleep k = sleepk
  - Another example: $[\![\textit{Karlos's dog slept}]\!] =$
    - $FA\ [\![\textit{slept}]\!][\![\textit{Karlos's dog}]\!] =$
    - $FA\ (\eta\text{sleep})(\lambda g^{\gamma}.\text{dog\_ofk})$

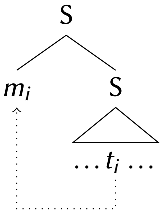
- As well as make Functional Application a bit more general:
  - $FA = \lambda m, n.mn \quad | \quad \lambda m, n.\eta FA \circledast m \circledast n$
  - An example: FA sleep k = sleepk
  - Another example: $[\![\textit{Karlos's dog slept}]\!] =$
    - $FA\ [\![\textit{slept}]\!][\![\textit{Karlos's dog}]\!] =$
    - $FA\ (\eta\text{sleep})(\lambda g^{\gamma}.\text{dog\_ofk})$
    - $\eta FA \circledast (\eta\text{sleep}) \circledast (\lambda g^{\gamma}.\text{dog\_ofk})$

- As well as make Functional Application a bit more general:
  - $FA = \lambda m, n.mn \quad | \quad \lambda m, n.\eta FA \circledast m \circledast n$
  - An example: FA sleep k = sleepk
  - Another example: $[\![\textit{Karlos's dog slept}]\!] =$
    - $FA\ [\![\textit{slept}]\!][\![\textit{Karlos's dog}]\!] =$
    - $FA\ (\eta\text{sleep})(\lambda g^{\gamma}.\text{dog\_ofk})$
    - $\eta FA \circledast (\eta\text{sleep}) \circledast (\lambda g^{\gamma}.\text{dog\_ofk})$
    - $\lambda g^{\gamma}.\begin{cases} FA\ \text{sleep}\ d & \text{dog\_ofk} = d \\ \# & \text{dog\_ofk} = \# \end{cases}$

- In general, we can use “Quantifier Raising” metaphorically:

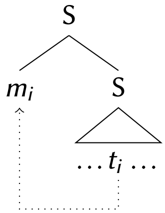


- In general, we can use "Quantifier Raising" metaphorically:



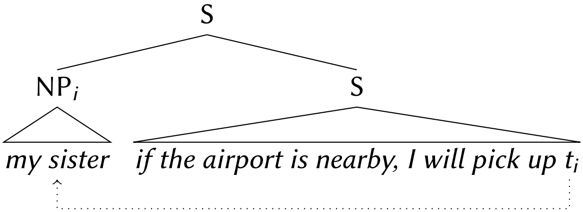
- This means that $\ldots m \ldots$ is interpreted as:

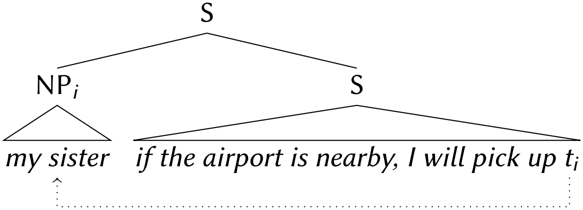
- In general, we can use "Quantifier Raising" metaphorically:



- This means that $\ldots m \ldots$ is interpreted as:
  - $\mu(\ldots [\![m]\!]^{\uparrow} \ldots)$

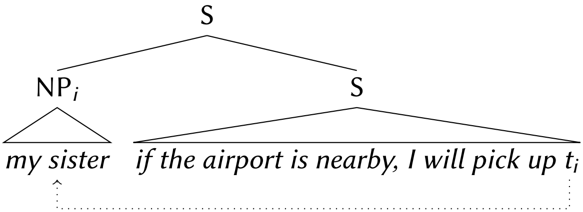
Monadic scope-taking



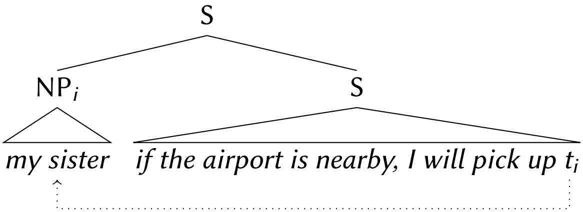


- represents the following interpretation:

Monadic scope-taking

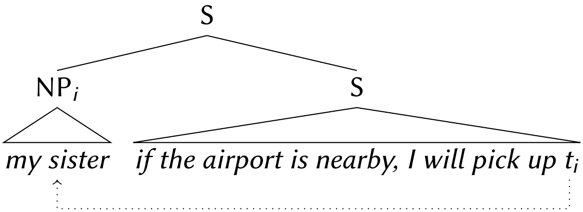


- represents the following interpretation:
  - ⟦*if the airport is nearby I will pick up my sister*⟧ =

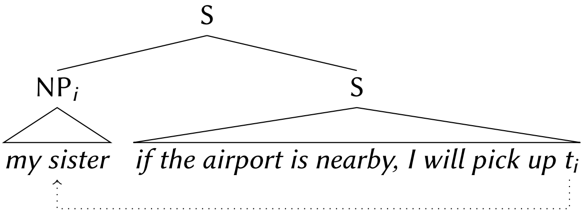


- represents the following interpretation:
  - $[\![\textit{if the airport is nearby I will pick up my sister}]\!] =$
  - $\mu(FA(\eta[\![\textit{if the airport is nearby}]\!])$
    $(FA(FA(\eta[\![\textit{pick up}]\!])(\eta[\![I]\!]))[\![\textit{my sister}]\!]^{\uparrow})) =$

Monadic scope-taking



- represents the following interpretation:
  - $[\![\textit{if the airport is nearby I will pick up my sister}]\!] =$
  - $\mu(FA(\eta[\![\textit{if the airport is nearby}]\!])$ $(FA(FA(\eta[\![\textit{pick up}]\!])(\eta[\![I]\!]))[\![\textit{my sister}]\!]^{\uparrow})) =$
  - $\lambda g^{\gamma}.\begin{cases}[\![\textit{airport nearby}]\!]g \rightarrow \text{pick\_up}sme & [\![\textit{my sister}]\!]g = s \\ \# & [\![\textit{my sister}]\!]g = \# \end{cases}$



- represents the following interpretation:
  - $[\![\textit{if the airport is nearby I will pick up my sister}]\!] =$
  - $\mu(FA(\eta[\![\textit{if the airport is nearby}]\!])$
    $(FA(FA(\eta[\![\textit{pick up}]\!])(\eta[\![\textit{I}]\!]))[\![\textit{my sister}]\!]^{\uparrow})) =$
  - $\lambda g^{\gamma}.\begin{cases} [\![\textit{airport nearby}]\!]g \rightarrow \text{pick\_up}sme & [\![\textit{my sister}]\!]g = s \\ \# & [\![\textit{my sister}]\!]g = \# \end{cases}$
  - $\rightsquigarrow$ *I have a sister* ☺

Filtering…

For examples like

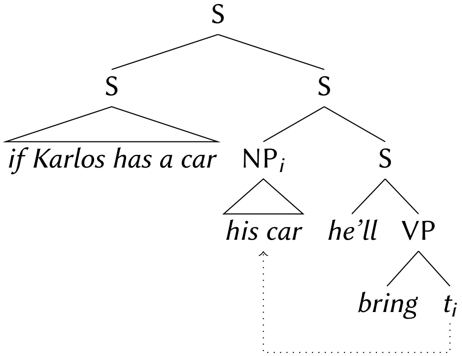
1. If Karlos has a car, he brought his car.

we simply don't scope the consequent clause above the filter, allowing presupposition satisfaction to go through.

For examples like

1. If Karlos has a car, he brought his car.

we simply don't scope the consequent clause above the filter, allowing presupposition satisfaction to go through.



Scope?

- What does the notion of scope-taking invoked for presupposition projection have to do with the usual notion invoked in the analysis of quantifiers?

Scope?

- What does the notion of scope-taking invoked for presupposition projection have to do with the usual notion invoked in the analysis of quantifiers?
  1. Every dog laughed.

Scope?

- What does the notion of scope-taking invoked for presupposition projection have to do with the usual notion invoked in the analysis of quantifiers?
  1. Every dog laughed.
     - ★ $[\![\textit{every dog}]\!] : (e \to t) \to t$

Scope?

- What does the notion of scope-taking invoked for presupposition projection have to do with the usual notion invoked in the analysis of quantifiers?
  1. Every dog laughed.
     - ★ $[\![\textit{every dog}]\!] : (e \rightarrow t) \rightarrow t$
- A common presentation of monads:

Scope?

- What does the notion of scope-taking invoked for presupposition projection have to do with the usual notion invoked in the analysis of quantifiers?
  1. Every dog laughed.
     - ★ $[\![every\ dog]\!] : (e \rightarrow t) \rightarrow t$
- A common presentation of monads:
  - $\eta : \alpha \rightarrow \mathcal{F}(\alpha)$ ('return')

Scope?

- What does the notion of scope-taking invoked for presupposition projection have to do with the usual notion invoked in the analysis of quantifiers?
  1. Every dog laughed.
     - ★ $[\![\textit{every dog}]\!] : (e \to t) \to t$
- A common presentation of monads:
  - $\eta : \alpha \to \mathcal{F}(\alpha)$ ('return')
  - $(\cdot)^{\ggg} : \mathcal{F}(\alpha) \to (\alpha \to \mathcal{F}(\beta)) \to \mathcal{F}(\beta)$ ('bind')

Scope?

- What does the notion of scope-taking invoked for presupposition projection have to do with the usual notion invoked in the analysis of quantifiers?

  1. Every dog laughed.
     - ★ $[\![\textit{every dog}]\!] : (e \rightarrow t) \rightarrow t$

- A common presentation of monads:
  - $\eta : \alpha \rightarrow \mathcal{F}(\alpha)$ ('return')
  - $(\cdot)^{\gg=} : \mathcal{F}(\alpha) \rightarrow (\alpha \rightarrow \mathcal{F}(\beta)) \rightarrow \mathcal{F}(\beta)$ ('bind')
    - ★ $m^{\gg=} = \lambda k.\mu(\eta k \circledast m)$

Scope?

- What does the notion of scope-taking invoked for presupposition projection have to do with the usual notion invoked in the analysis of quantifiers?
  1. Every dog laughed.
     - ★ $[\![\textit{every dog}]\!] : (e \to t) \to t$
- A common presentation of monads:
  - $\eta : \alpha \to \mathcal{F}(\alpha)$ ('return')
  - $(\cdot)^{\gg=} : \mathcal{F}(\alpha) \to (\alpha \to \mathcal{F}(\beta)) \to \mathcal{F}(\beta)$ ('bind')
    - ★ $m^{\gg=} = \lambda k.\mu(\eta k \circledast m)$
- Applied to, e.g., $[\![\textit{his car}]\!]$, we have:

Scope?

- What does the notion of scope-taking invoked for presupposition projection have to do with the usual notion invoked in the analysis of quantifiers?
  1. Every dog laughed.
     - $[\![\textit{every dog}]\!] : (e \rightarrow t) \rightarrow t$
- A common presentation of monads:
  - $\eta : \alpha \rightarrow \mathcal{F}(\alpha)$ ('return')
  - $(\cdot)^{\ggcurly} : \mathcal{F}(\alpha) \rightarrow (\alpha \rightarrow \mathcal{F}(\beta)) \rightarrow \mathcal{F}(\beta)$ ('bind')
    - $m^{\ggcurly} = \lambda k.\mu(\eta k \circledast m)$
- Applied to, e.g., $[\![\textit{his car}]\!]$, we have:
  - $(\lambda g^{\gamma}.\text{car\_of}(\texttt{sel}g))^{\ggcurly} =$

Scope?

- What does the notion of scope-taking invoked for presupposition projection have to do with the usual notion invoked in the analysis of quantifiers?
  1. Every dog laughed.
     - ★ $[\![\textit{every dog}]\!] : (e \rightarrow t) \rightarrow t$
- A common presentation of monads:
  - $\eta : \alpha \rightarrow \mathcal{F}(\alpha)$ ('return')
  - $(\cdot)^{\ggg} : \mathcal{F}(\alpha) \rightarrow (\alpha \rightarrow \mathcal{F}(\beta)) \rightarrow \mathcal{F}(\beta)$ ('bind')
    - ★ $m^{\ggg} = \lambda k.\mu(\eta k \circledast m)$
- Applied to, e.g., $[\![\textit{his car}]\!]$, we have:
  - $(\lambda g^{\gamma}.\mathtt{car\_of}(\mathtt{sel}g))^{\ggg} =$
  - $\lambda k^{e \rightarrow \gamma \rightarrow t_{\#}}, g^{\gamma}. \begin{cases} kc & \mathtt{car\_of}(\mathtt{sel}g) = c \\ \# & \mathtt{car\_of}(\mathtt{sel}g) = \# \end{cases}$

Scope?

- What does the notion of scope-taking invoked for presupposition projection have to do with the usual notion invoked in the analysis of quantifiers?
  1. Every dog laughed.
     - ★ $[\![\textit{every dog}]\!] : (e \to t) \to t$
- A common presentation of monads:
  - $\eta : \alpha \to \mathcal{F}(\alpha)$ ('return')
  - $(\cdot)^{\ggcurly} : \mathcal{F}(\alpha) \to (\alpha \to \mathcal{F}(\beta)) \to \mathcal{F}(\beta)$ ('bind')
    - ★ $m^{\ggcurly} = \lambda k.\mu(\eta k \circledast m)$
- Applied to, e.g., $[\![\textit{his car}]\!]$, we have:
  - $(\lambda g^{\gamma}.\mathtt{car\_of}(\mathtt{sel} g))^{\ggcurly} =$
  - $\lambda k^{e \to \gamma \to t_{\#}}, g^{\gamma}. \begin{cases} kc & \mathtt{car\_of}(\mathtt{sel} g) = c \\ \# & \mathtt{car\_of}(\mathtt{sel} g) = \# \end{cases}$
  - Allows the presupposition trigger to act like a quantifier, i.e.,

Scope?

- What does the notion of scope-taking invoked for presupposition projection have to do with the usual notion invoked in the analysis of quantifiers?
  1. Every dog laughed.
     - ★ $[\![\textit{every dog}]\!] : (e \rightarrow t) \rightarrow t$
- A common presentation of monads:
  - $\eta : \alpha \rightarrow \mathcal{F}(\alpha)$ ('return')
  - $(\cdot)^{\ggg} : \mathcal{F}(\alpha) \rightarrow (\alpha \rightarrow \mathcal{F}(\beta)) \rightarrow \mathcal{F}(\beta)$ ('bind')
    - ★ $m^{\ggg} = \lambda k.\mu(\eta k \circledast m)$
- Applied to, e.g., $[\![\textit{his car}]\!]$, we have:
  - $(\lambda g^{\gamma}.\mathtt{car\_of}(\mathtt{sel} g))^{\ggg} =$
  - $\lambda k^{e \rightarrow \gamma \rightarrow t_{\#}}, g^{\gamma}. \begin{cases} kc & \mathtt{car\_of}(\mathtt{sel} g) = c \\ \# & \mathtt{car\_of}(\mathtt{sel} g) = \# \end{cases}$
  - Allows the presupposition trigger to act like a quantifier, i.e.,
    - ★ by taking scope syntactically (Heim and Kratzer, 1998)

Scope?

- What does the notion of scope-taking invoked for presupposition projection have to do with the usual notion invoked in the analysis of quantifiers?
  1. Every dog laughed.
     - ★ $[\![\textit{every dog}]\!] : (e \to t) \to t$
- A common presentation of monads:
  - $\eta : \alpha \to \mathcal{F}(\alpha)$ ('return')
  - $(\cdot)^{\gg\!=} : \mathcal{F}(\alpha) \to (\alpha \to \mathcal{F}(\beta)) \to \mathcal{F}(\beta)$ ('bind')
    - ★ $m^{\gg\!=} = \lambda k.\mu(\eta k \circledast m)$
- Applied to, e.g., $[\![\textit{his car}]\!]$, we have:
  - $(\lambda g^{\gamma}.\mathtt{car\_of}(\mathtt{sel}\,g))^{\gg\!=} =$
  - $\lambda k^{e \to \gamma \to t_{\#}}, g^{\gamma}. \begin{cases} kc & \mathtt{car\_of}(\mathtt{sel}\,g) = c \\ \# & \mathtt{car\_of}(\mathtt{sel}\,g) = \# \end{cases}$
  - Allows the presupposition trigger to act like a quantifier, i.e.,
    - ★ by taking scope syntactically (Heim and Kratzer, 1998)
    - ★ by composing via continuations (Barker, 2002; Barker and Shan, 2014)

Summary

- An interpretation scheme that relies on applicative composition with Maybe types has presuppositions project by default and filters apply automatically.

Summary

- An interpretation scheme that relies on applicative composition with Maybe types has presuppositions project by default and filters apply automatically.
- Allowing a presupposition trigger to take scope past a filter means allowing the evaluation of its presuppositions to be delayed until *after* the evaluation of the filter.

Summary

- An interpretation scheme that relies on applicative composition with Maybe types has presuppositions project by default and filters apply automatically.
- Allowing a presupposition trigger to take scope past a filter means allowing the evaluation of its presuppositions to be delayed until *after* the evaluation of the filter.
  - Accomplished in terms of $(\cdot)^{\uparrow}$, μ, and Functional Application.

References

Barker, Chris. 2002. Continuations and the Nature of Quantification. *Natural Language Semantics* 10:211–242. https://doi.org/10.1023/A:1022183511876.

Barker, Chris, and Chung-chieh Shan. 2014. *Continuations and natural language*, volume 53. Oxford studies in theoretical linguistics.

Chierchia, Gennaro, and Sally McConnell-Ginet. 2000. *Meaning and Grammar*. Cambridge: MIT Press, second edition.

von Fintel, Kai. 2004. Would you Believe It? The King of France is Back! (Presuppositions and Truth-Value Intuitions). In *Descriptions and Beyond*, ed. Marga Reimer and Anne Bezuidenhout. New York: Oxford University Press.

von Fintel, Kai. 2008. What is presupposition accommodation, again? *Philosophical Perspectives* 22:137–170. https://doi.org/10.1111/j.1520-8583.2008.00144.x.

References

Geurts, Bart. 1996. Local Satisfaction Guaranteed: A Presupposition Theory and Its Problems. *Linguistics and Philosophy* 19:259–294. https://doi.org/10.1007/BF00628201.

Heim, Irene. 1983. On the Projection Problem for Presuppositions. In *Proceedings of the 2nd West Coast Conference on Formal Linguistics*, ed. Michael D. Barlow, Daniel P. Flickinger, and Michael Westcoat, 114–125. Stanford: Stanford University Press.

Heim, Irene, and Angelika Kratzer. 1998. *Semantics in Generative Grammar*. Malden: Blackwell.

Langendoen, Donald T., and Harris B. Savin. 1971. The projection problem for presuppositions. In *Studies in Linguistic Semantics*, ed. Charles J. Fillmore and Donald T. Langendoen, 54–60. Holt, Rinehart and Winston.

McBride, Conor, and Ross Paterson. 2008. Applicative Programming with Effects. *Journal of Functional Programming* 18:1–13. https://doi.org/10.1017/S0956796807006326.