Generating underspecified interpretations as terms of the representation language

Chris Fox
Dept. of Computer Science
University of Essex
foxcj@essex.ac.uk

Shalom Lappin
Dept. of Computer Science
King’s College London
lappin@dcs.kcl.ac.uk
Overview

- Two traditional assumptions of formal semantics
- Property Theory with Curry Typing (PTCT): An expressive “first-order” system with fine-grained intensionality
- Syntax and proof theory
- Model theory
- Product types (used for weak inhomogeneous lists)
- Quantifiers and arity-reduction of relations
- Underspecification in PTCT
- Constraints on scope readings as filters in PTCT
- Comparison with other theories of underspecification
- Conclusions
Background
Assumption I

First Assumption: Functional Types and Higher-Order Logic

- Usually assumed that higher-order logic and type theory are necessary to achieve the expressive power required for NL semantic.
- In particular, the function types of higher-order systems are taken to be required for generalised quantifiers and modifiers.
- e.g. Montague (1974); Gallin (1975); Barwise and Cooper (1981); Keenan and Stavi (1986).
- If we are interested in a computationally viable theory, then we should be concerned about the formal power of the theory.
- Property Theory with Curry Typing (PTCT) is a first-order theory with function types.
  - ... and separation types (sub-types)
  - ... and polymorphic types
  - ... and product types.
Assumption II

Second Assumption: Intensions and Possible Worlds

- The view that characterizes intensions as functions from possible worlds (situations) to extensions has been influential at least since Carnap (1947).
- It achieves detailed formal expression in Montague (1974).
- This treatment of intensions is not sufficiently fine-grained.
- PTCT follows Bealer (1982), taking intensions to be basic (actually represented by first-order terms).
- Intensions are independent of modality, and identity is not reduced to equivalence.
The General Approach

- In formal semantics, NL is mapped to a formal logic.
The General Approach

- In formal semantics, NL is mapped to a formal logic.
- Such logics typically consist of wff, terms and types.
The General Approach

- In formal semantics, NL is mapped to a formal logic.
- Such logics typically consist of wff, terms and types.
- NL is usually mapped directly to wffs.
The General Approach

- In formal semantics, NL is mapped to a formal logic.
- Such logics typically consist of wff, terms and types.
- NL is usually mapped directly to wffs.
- Intensions (e.g. in IL) are derived from wffs.
The General Approach

- In formal semantics, NL is mapped to a formal logic.
- Such logics typically consist of wff, terms and types.
- NL is usually mapped directly to wffs.
- Intensions (e.g. in IL) are derived from wffs.
- Why not use terms for our notion of intensionality?
The General Approach

- In formal semantics, NL is mapped to a formal logic.
- Such logics typically consist of wff, terms and types.
- NL is usually mapped directly to wffs.
- Intensions (e.g. in IL) are derived from wffs.
- Why not use terms for our notion of intensionality?
- In which case we map NL to terms.
The General Approach

- In formal semantics, NL is mapped to a formal logic.
- Such logics typically consist of wff, terms and types.
- NL is usually mapped directly to wffs.
- Intensions (e.g. in IL) are derived from wffs.
- Why not use terms for our notion of intensionality?
- In which case we map NL to terms.
- The language of terms will have to be more complicated.
The General Approach

- In formal semantics, NL is mapped to a formal logic.
- Such logics typically consist of wff, terms and types.
- NL is usually mapped directly to wffs.
- Intensions (e.g. in IL) are derived from wffs.
- Why not use terms for our notion of intensionality?
- In which case we map NL to terms.
- The language of terms will have to be more complicated.
- Some terms will correspond to propositions.
The General Approach

- In formal semantics, NL is mapped to a formal logic.
- Such logics typically consist of wff, terms and types.
- NL is usually mapped directly to wffs.
- Intensions (e.g. in IL) are derived from wffs.
- Why not use terms for our notion of intensionality?
- In which case we map NL to terms.
- The language of terms will have to be more complicated.
- Some terms will correspond to propositions.
- If the language of wffs and types proves a term represents a proposition...
The General Approach

- In formal semantics, NL is mapped to a formal logic.
- Such logics typically consist of wff, terms and types.
- NL is usually mapped directly to wffs.
- Intensions (e.g. in IL) are derived from wffs.
- Why not use terms for our notion of intensionality?
- In which case we map NL to terms.
- The language of terms will have to be more complicated.
- Some terms will correspond to propositions.
- If the language of wffs and types proves a term represents a proposition... 
- ...then the truth conditions of such terms can be considered.
PTCT: Property Theory with Curry Typing
PTCT: Syntax

The language of PTCT consists of the following sub-languages:

Terms \[ t ::= x \mid c \mid l \mid X \mid T \mid \lambda x(t) \mid (t)t \]

(logical constants) \[ l ::= \hat{\land} \mid \hat{\lor} \mid \hat{\rightarrow} \mid \hat{\leftrightarrow} \mid \hat{\sim} \mid \hat{\forall} \mid \hat{\exists} \mid \hat{\equiv}_T \mid \hat{\equiv}_T \mid \epsilon T \]
PTCT: Syntax

The language of PTCT consists of the following sub-languages:

**Terms** \( t ::= x \mid c \mid l \mid X \mid T \mid \lambda x(t) \mid (t)t \)

(logical constants) \( l ::= \wedge \mid \vee \mid \to \mid \leftrightarrow \mid \sim \mid \exists \mid \forall \mid \exists_T \mid \equiv_T \mid \epsilon T \)

- The language of terms is the untyped \( \lambda \)-calculus, enriched with logical constants. Used to represent the interpretations of natural language expressions. It has no internal logic! The identity criteria are those of the \( \lambda \)-calculus.
PTCT: Syntax

The language of PTCT consists of the following sub-languages:

**Terms** \( t ::= x \mid c \mid l \mid X \mid T \mid \lambda x(t) \mid (t)t \)

(logical constants) \( l ::= \hat{\wedge} \mid \hat{\lor} \mid \hat{\rightarrow} \mid \hat{\leftrightarrow} \mid \hat{\sim} \mid \hat{\forall} \mid \hat{\exists} \mid \hat{=}T \mid \hat{\approx}T \mid \varepsilon T \)

- The language of terms is the untyped \( \lambda \)-calculus, enriched with logical constants. Used to **represent** the interpretations of natural language expressions. It has no internal logic! The identity criteria are those of the \( \lambda \)-calculus.

**Types** \( T ::= B \mid \text{Prop} \mid T \rightarrow S \mid X \mid \{x \in T : \varphi\} \)
PTCT: Syntax

The language of PTCT consists of the following sub-languages:

**Terms** \( t ::= x | c | l | X | T | \lambda x(t) | (t)t \)

(logical constants) \( l ::= \land | \lor | \rightarrow | \leftrightarrow | \neg | \exists | \exists^T | \exists^T \)

- The language of terms is the untyped \( \lambda \)-calculus, enriched with logical constants. Used to represent the interpretations of natural language expressions. It has no internal logic! The identity criteria are those of the \( \lambda \)-calculus.

**Types** \( T ::= B | \text{Prop} | T \rightarrow S | X | \{x \in T : \varphi^' \} \)

- \( \varphi^' \) is a term representable fragment of wff.
**PTCT: Syntax**

The language of PTCT consists of the following sub-languages:

**Terms**  \( t ::= x \mid c \mid l \mid X \mid T \mid \lambda x(t) \mid (t)t \)

(logical constants) \( l ::= \wedge \mid \vee \mid \rightarrow \mid \Leftarrow \mid \sim \mid \exists \mid \exists T \mid \exists T \mid \epsilon T \)

- The language of terms is the untyped \( \lambda \)-calculus, enriched with logical constants.
  Used to represent the interpretations of natural language expressions. It has no internal logic! The identity criteria are those of the \( \lambda \)-calculus.

**Types**  \( T ::= B \mid Prop \mid T \rightarrow S \mid X \mid \{x \in T : \varphi'\} \)

- \( \varphi' \) is a term representable fragment of wff.

- Bound type variables are limited in their range to non-polymorphic types in order to avoid impredicative type membership statements.
**PTCT: Syntax**

The language of PTCT consists of the following sub-languages:

**Terms**

\[ t ::= x \mid c \mid l \mid X \mid T \mid \lambda x(t) \mid (t)t \]

(logical constants)

\[ l ::= \land \mid \lor \mid \to \mid \leftrightarrow \mid \sim \mid \Rightarrow \mid \exists \mid \exists T \mid \exists T \mid \epsilon T \]

- The language of terms is the untyped \( \lambda \)-calculus, enriched with logical constants. Used to represent the interpretations of natural language expressions. It has no internal logic! The identity criteria are those of the \( \lambda \)-calculus.

**Types**

\[ T ::= B \mid \text{Prop} \mid T \to S \mid X \mid \{ x \in T : \varphi' \} \]

- \( \varphi' \) is a term representable fragment of wff.

- Bound type variables are limited in their range to non-polymorphic types in order to avoid impredicative type membership statements.

- The languages of types and terms are combined with appropriate rules and axioms to produce a Curry-typed \( \lambda \)-calculus with type variables, separation, and polymorphic types (e.g. Turner 1997).
PTCT: Syntax

\[
\text{Wff} \quad \varphi ::= \alpha \mid (\varphi \land \psi) \mid (\varphi \lor \psi) \mid (\varphi \rightarrow \psi) \mid (\varphi \leftrightarrow \psi) \mid (\forall x \varphi) \mid (\exists x \varphi) \mid \text{true}_t
\]

(atomic wff) \quad \alpha ::= t =_S s \mid \sim t \mid t \in S \mid t \equiv_S s
\[ \text{Wff} \quad \varphi ::= \alpha \mid (\varphi \land \psi) \mid (\varphi \lor \psi) \mid (\varphi \rightarrow \psi) \mid (\varphi \leftrightarrow \psi) \mid (\forall x \varphi) \mid (\exists x \varphi) \mid \text{true} \]

(atomic wff) \[ \alpha ::= t = S s \mid \sim t \mid t \in S \mid t \cong_S s \]

- The first-order language of wffs will be used to formulate type judgements for terms, and truth conditions for those terms judged to be in Prop.
\[ \textbf{...PTCT: Syntax} \]

\[
\text{\textbf{Wff}} \quad \varphi ::= \alpha \mid (\varphi \land \psi) \mid (\varphi \lor \psi) \mid (\varphi \rightarrow \psi) \mid (\varphi \leftrightarrow \psi) \mid (\forall x \varphi) \mid (\exists x \varphi) \mid ^{\text{true}}t \\
\text{(atomic wff)} \quad \alpha ::= t =_S s \mid \sim t \mid t \in S \mid t \equiv_S s
\]

- The first-order language of wffs will be used to formulate type judgements for terms, and truth conditions for those terms judged to be in Prop.
- If a term \( t \) represents a proposition, \( ^{\text{true}}(t) \) is a wff that denotes its truth conditions. The identity criteria of wffs are those of their truth conditions.
PTCT: Syntax

\[ \text{Wff} \quad \varphi ::= \alpha \mid (\varphi \land \psi) \mid (\varphi \lor \psi) \mid (\varphi \rightarrow \psi) \mid (\varphi \leftrightarrow \psi) \mid (\forall x \varphi) \mid (\exists x \varphi) \mid \text{true} \text{ } t \]

(atomic wff) \[ \alpha ::= t =_S s \mid \sim t \mid t \in S \mid t \cong_S s \]

- The first-order language of wffs will be used to formulate type judgements for terms, and truth conditions for those terms judged to be in $\text{Prop}$.
- If a term $t$ represents a proposition, $\text{true}(t)$ is a wff that denotes its truth conditions. The identity criteria of wffs are those of their truth conditions.
- All types are term representable.
**PTCT: Rules and Axioms**

Here we exemplify some of these kinds of rules as they apply to conjunction, both as it appears in the language of wff (\(\land\)), and in the language of terms (\(\hat{\land}\)).

- The basic connectives of the wff

\[
\begin{align*}
\varphi & \quad \psi \\
\hline
\varphi \land \psi & \land i \\
\varphi & \quad \varphi \land \psi \\
\hline
\varphi & \quad \varphi \land \psi \\
\psi & \quad \varphi \land \psi
\end{align*}
\]

\(\land e\)
PTCT: Rules and Axioms

Here we exemplify some of these kinds of rules as they apply to conjunction, both as it appears in the language of wff (\(\land\)), and in the language of terms (\(\hat{\land}\)).

- The basic connectives of the wff

\[
\begin{align*}
\varphi & \quad \psi \\
\hline
\varphi \land \psi & \quad \land i \\
\varphi \land \psi & \quad \land e \\
\psi & \quad \land e
\end{align*}
\]

- Typing rules for \(\lambda\)-terms

\[
t \in \text{Prop} \land t' \in \text{Prop} \rightarrow (t \land t') \in \text{Prop}
\]
**PTCT: Rules and Axioms**

Here we exemplify some of these kinds of rules as they apply to conjunction, both as it appears in the language of wff (\(\land\)), and in the language of terms (\(\hat{\land}\)).

- The basic connectives of the wff

\[
\frac{\varphi \ \psi}{\varphi \land \psi} \ \land \ i \quad \frac{\varphi \land \psi}{\varphi} \ \land \ e \quad \frac{\varphi \land \psi}{\psi} \ \land \ e
\]

- Typing rules for \(\lambda\)-terms

\[
t \in \text{Prop} \land t' \in \text{Prop} \rightarrow (t \hat{\land} t') \in \text{Prop}
\]

- Truth conditions for Propositions

\[
t \in \text{Prop} \land t' \in \text{Prop} \rightarrow (\text{true}(t \hat{\land} t') \leftrightarrow \text{true}t \land \text{true}t')
\]
PTCT: Rules and Axioms

Here we exemplify some of these kinds of rules as they apply to conjunction, both as it appears in the language of wff ($\land$), and in the language of terms ($\hat{\land}$).

- The basic connectives of the wff

$$\frac{\varphi \quad \psi}{\varphi \land \psi} \land i \quad \frac{\varphi \land \psi}{\varphi} \land e \quad \frac{\varphi \land \psi}{\psi} \land e$$

- Typing rules for $\lambda$-terms

$$t \in \text{Prop} \land t' \in \text{Prop} \to (t \hat{\land} t') \in \text{Prop}$$

- Truth conditions for Propositions

$$t \in \text{Prop} \land t' \in \text{Prop} \to \left(\text{true}(t \hat{\land} t') \iff \text{true}t \land \text{true}t'\right)$$

- We have developed a tableau system for PTCT.
PTCT: Rules and Axioms

Here we exemplify some of these kinds of rules as they apply to conjunction, both as it appears in the language of wff ($\land$), and in the language of terms ($\hat{\land}$).

- The basic connectives of the wff

$$
\frac{\varphi \quad \psi}{\varphi \land \psi} \quad \text{i} \quad \frac{\varphi \land \psi}{\varphi} \quad \text{e} \quad \frac{\varphi \land \psi}{\psi} \quad \text{e}
$$

- Typing rules for $\lambda$-terms

$$t \in \text{Prop} \land t' \in \text{Prop} \rightarrow (t \hat{\land} t') \in \text{Prop}$$

- Truth conditions for Propositions

$$t \in \text{Prop} \land t' \in \text{Prop} \rightarrow (\text{true}(t \hat{\land} t') \leftrightarrow \text{true} t \land \text{true} t')$$

- We have developed a tableau system for PTCT.

- In Fox and Lappin forthcoming we prove the soundness and completeness of the basic logic of PTCT.
Equivalence and Identity

There are two equivalence notions in this theory: intensional identity and extensional equivalence.

- $t \cong_T s$ states that the terms $t, s$ are extensionally equivalent in type $T$
  - **Propositions:** in the case where two terms $t, s$ are propositions $(t, s \in \text{Prop})$, then $t \cong_{\text{Prop}} s$ corresponds to $t \leftrightarrow s$
  - **Predicates:** in the case where two predicates of $T$ are extensionally equivalent $t \cong_{(T \rightarrow \text{Prop})} s$ then $t, s$ each hold of the same elements of $T$. Therefore $\forall x (x \in T \rightarrow (\text{true} t(x) \leftrightarrow \text{true} s(x)))$

- $t =_T s$ states that two terms are intensionally identical in type $T$.
  - The rules for intensional identity are essentially those of the $\lambda\alpha\beta\eta$-calculus.
  - We are able to derive $t =_T s \rightarrow t \cong_T s$ for all types inhabited by $t, (s)$, but not $t \cong_T s \rightarrow t =_T s$. 
Polymorphism and Separation

Applications of Polymorphism and Separation in NL

- NL expressions act as if they belong to more than one semantic type: *playing tennis is fun, to play tennis is fun, tennis is fun* (Chierchia 1982; Turner 1997).

"Is fun" can have the type $X: X \rightarrow \text{Prop}$.

"and" can have the type $X: X \rightarrow X \rightarrow X$. 

We use separation types for dynamic analyses of anaphora and ellipsis.
Polymorphism and Separation

Applications of Polymorphism and Separation in NL

- NL expressions act as if they belong to more than one semantic type: *playing tennis is fun, to play tennis is fun, tennis is fun* (Chierchia 1982; Turner 1997).

- Some NL expressions (such as conjunctions) can combine expressions of different types. There are constraints on the types of the combined expressions and the resultant expression.
Polymorphism and Separation

Applications of Polymorphism and Separation in NL

- NL expressions act as if they belong to more than one semantic type: *playing tennis is fun, to play tennis is fun, tennis is fun* (Chierchia 1982; Turner 1997).
- Some NL expressions (such as conjunctions) can combine expressions of different types. There are constraints on the types of the combined expressions and the resultant expression.
- More “natural” treatments of such phenomena are possible if we allow a flexible system of types (such as Curry-style typing), where
Polymorphism and Separation

Applications of Polymorphism and Separation in NL

- NL expressions act as if they belong to more than one semantic type: *playing tennis is fun, to play tennis is fun, tennis is fun* (Chierchia 1982; Turner 1997).
- Some NL expressions (such as conjunctions) can combine expressions of different types. There are constraints on the types of the combined expressions and the resultant expression.
- More “natural” treatments of such phenomena are possible if we allow a flexible system of types (such as Curry-style typing), where
  1. expressions can belong to more than one type;
Polymorphism and Separation

Applications of Polymorphism and Separation in NL

- NL expressions act as if they belong to more than one semantic type: *playing tennis is fun, to play tennis is fun, tennis is fun* (Chierchia 1982; Turner 1997).
- Some NL expressions (such as conjunctions) can combine expressions of different types. There are constraints on the types of the combined expressions and the resultant expression.
- More “natural” treatments of such phenomena are possible if we allow a flexible system of types (such as Curry-style typing), where
  1. expressions can belong to more than one type;
  2. functional types can apply to arguments of several types.
Polymorphism and Separation

Applications of Polymorphism and Separation in NL

- NL expressions act as if they belong to more than one semantic type: *playing tennis is fun, to play tennis is fun, tennis is fun* (Chierchia 1982; Turner 1997).
- Some NL expressions (such as conjunctions) can combine expressions of different types. There are constraints on the types of the combined expressions and the resultant expression.
- More “natural” treatments of such phenomena are possible if we allow a flexible system of types (such as Curry-style typing), where
  1. expressions can belong to more than one type;
  2. functional types can apply to arguments of several types.
- “Is fun” can have the type \( \Pi X.X \rightarrow Prop \).
Polymorphism and Separation

Applications of Polymorphism and Separation in NL

- NL expressions act as if they belong to more than one semantic type: *playing tennis is fun, to play tennis is fun, tennis is fun* (Chierchia 1982; Turner 1997).
- Some NL expressions (such as conjunctions) can combine expressions of different types. There are constraints on the types of the combined expressions and the resultant expression.
- More “natural” treatments of such phenomena are possible if we allow a flexible system of types (such as Curry-style typing), where
  1. expressions can belong to more than one type;
  2. functional types can apply to arguments of several types.
- “Is fun” can have the type $\Pi X. X \Rightarrow Prop$.
- “and” can have the type $\Pi X. X \Rightarrow X \Rightarrow X$. 

We use separation types for dynamic analyses of anaphora and ellipsis.
Polymorphism and Separation

Applications of Polymorphism and Separation in NL

- NL expressions act as if they belong to more than one semantic type: *playing tennis is fun, to play tennis is fun, tennis is fun* (Chierchia 1982; Turner 1997).

- Some NL expressions (such as conjunctions) can combine expressions of different types. There are constraints on the types of the combined expressions and the resultant expression.

- More “natural” treatments of such phenomena are possible if we allow a flexible system of types (such as Curry-style typing), where
  1. expressions can belong to more than one type;
  2. functional types can apply to arguments of several types.

- “Is fun” can have the type $\Pi X. X \Rightarrow Prop$.
- “and” can have the type $\Pi X. X \Rightarrow X \Rightarrow X$.

- We use separation types for dynamic analyses of anaphora and ellipsis.
Model Theory
Model Theory for PTCT

Sketch of a model

- A model of the untyped λ-calculus (e.g. General Functional Models), 
  $\mathcal{D} = \langle D, [D \rightarrow D], \Phi, \Psi \rangle$ where $D$ is isomorphic to $[D \rightarrow D]$
  1. $D$ is a non-empty set,
  2. $[D \rightarrow D]$ is some class of functions from $D$ to $D$,
  3. $\Phi : D \rightarrow [D \rightarrow D]$,
  4. $\Psi : [D \rightarrow D] \rightarrow D$,
  5. $\Psi(\Phi(d)) = d$ for all $d \in D$
  (Meyer 1982).

- Interpret the types as terms in $D$ that correspond to subsets of $D$. 
A Model for PTCT

- A model of PTCT is $\mathcal{M} = \langle \mathcal{D}, \mathcal{T}, \mathcal{P}, \mathcal{B}, \mathcal{B}, \mathcal{T} \rangle$, where
  1. $\mathcal{D}$ is a model of the $\lambda$-calculus.
  2. $\mathcal{T} : \mathcal{D} \rightarrow \{0, 1\}$ models the truth predicate $\text{true}$.
  3. $\mathcal{P} \subseteq \mathcal{D}$ models the class of propositions.
  4. $\mathcal{B} \subseteq \mathcal{D}$ models the class of basic individuals.
  5. $\mathcal{B}(\mathcal{B})$ is a set of sets whose elements partition $\mathcal{B}$ into equivalence classes of individuals.
  6. $\mathcal{T} \subseteq \mathcal{D}$ models the term representation of types.

with sufficient structural constraints on $\mathcal{T}$, $\mathcal{P}$ and $\mathcal{T}$ to validate the rules of PTCT.
Extensions of the Type System
Product Types

- Product types are of the form $S \otimes T$
Product Types

- Product types are of the form $S \otimes T$
- They have elements of the form $\langle s, t \rangle$
Product Types

- Product types are of the form $S \otimes T$
- They have elements of the form $\langle s, t \rangle$
- We add the type $S \otimes T$, and a tableau rule corresponding to the following axiom.
Product Types

- Product types are of the form $S \otimes T$
- They have elements of the form $\langle s, t \rangle$
- We add the type $S \otimes T$, and a tableau rule corresponding to the following axiom.
  \[
  \text{PROD: } \langle x, y \rangle \in (S \otimes T) \leftrightarrow x \in S \land y \in T
  \]
Product Types

- Product types are of the form $S \otimes T$
- They have elements of the form $\langle s, t \rangle$
- We add the type $S \otimes T$, and a tableau rule corresponding to the following axiom.
  \[
  \text{PROD: } \langle x, y \rangle \in (S \otimes T) \iff x \in S \land y \in T
  \]
- We make use of product types in our treatment of underspecification: unlike (monomorphic) lists, they allow us to cope with polymorphic relations.
Product Types

- Product types are of the form $S \otimes T$
- They have elements of the form $\langle s, t \rangle$
- We add the type $S \otimes T$, and a tableau rule corresponding to the following axiom.
  \[
  \text{PROD: } \langle x, y \rangle \in (S \otimes T) \leftrightarrow x \in S \land y \in T
  \]
- We make use of product types in our treatment of underspecification: unlike (monomorphic) lists, they allow us to cope with polymorphic relations.
- The appropriate notion of pairs and projections are $\lambda$-definable:
  \[
  \begin{align*}
  \langle x, y \rangle &= \text{def } \lambda z(z(x)(y)) \\
  \text{fst} &= \text{def } \lambda p(p\lambda xy(x)) \\
  \text{snd} &= \text{def } \lambda p(p\lambda xy(y))
  \end{align*}
  \]
Product Types

- Product types are of the form $S \otimes T$
- They have elements of the form $\langle s, t \rangle$
- We add the type $S \otimes T$, and a tableau rule corresponding to the following axiom.
  \[
  \text{PROD: } \langle x, y \rangle \in (S \otimes T) \leftrightarrow x \in S \land y \in T
  \]
- We make use of product types in our treatment of underspecification: unlike (monomorphic) lists, they allow us to cope with polymorphic relations.
- The appropriate notion of pairs and projections are $\lambda$-definable:
  \[
  \langle x, y \rangle = \text{def } \lambda z (z(x)(y))
  \]
  \[
  \text{fst} = \text{def } \lambda p (p \lambda xy(x))
  \]
  \[
  \text{snd} = \text{def } \lambda p (p \lambda xy(y))
  \]
- We write $\langle t_1, t_2, \ldots, t_n \rangle$ for $\langle t_1, \langle t_2, \langle \ldots, t_n \rangle \rangle \ldots \rangle$, and $T_1 \otimes T_2 \otimes \ldots T_n$ for $T_1 \otimes (T_2 \otimes (\ldots T_n) \ldots)$
For inhomogeneous “lists,” we specify that for any $k$-tuple $\langle t_1, \ldots, t_k \rangle \in T_1 \otimes \ldots \otimes T_k$, the last element of the $k$-tuple, $t_k$ is a designated object, like 0 or $\bot$. This condition ensures that it is possible to recognize the end of a $k$-tuple and so compute its arity. It renders the elements of product types equivalent to weak lists with elements of (possibly) distinct types. As in the case of lists, we generally suppress this final designated element when representing a $k$-tuple.
Product Types and Lists

- For inhomogeneous “lists,” we specify that for any $k$-tuple $\langle t_1, \ldots, t_k \rangle \in T_1 \otimes \ldots \otimes T_k$, the last element of the $k$-tuple, $t_k$ is a designated object, like 0 or ⊥.

- This condition ensures that it is possible to recognize the end of a $k$-tuple and so compute its arity.
Product Types and Lists

- For inhomogeneous “lists,” we specify that for any $k$-tuple $\langle t_1, \ldots, t_k \rangle \in T_1 \otimes \ldots \otimes T_k$, the last element of the $k$-tuple, $t_k$, is a designated object, like 0 or $\perp$.

- This condition ensures that it is possible to recognize the end of a $k$-tuple and so compute its arity.

- It renders the elements of product types equivalent to weak lists with elements of (possibly) distinct types.
Product Types and Lists

- For inhomogeneous “lists,” we specify that for any $k$-tuple $\langle t_1, \ldots, t_k \rangle \in T_1 \otimes \ldots \otimes T_k$, the last element of the $k$-tuple, $t_k$, is a designated object, like 0 or $\bot$.
- This condition ensures that it is possible to recognize the end of a $k$-tuple and so compute its arity.
- It renders the elements of product types equivalent to weak lists with elements of (possibly) distinct types.
- As in the case of lists, we generally suppress this final designated element when representing a $k$-tuple.
Quantifiers and Arity Reduction

We take quantifiers to be arity reduction operators (Keenan 1992; van Eijck 2003)

- Quantifiers (representations of noun phrases) are of type $(X \implies \text{Prop}) \implies \text{Prop}$, which we shall write $\text{Quant}^X$ for clarity (where $X$ is typically $B$).

Core propositional relations (such as verbs) are of type $X^1 = \cdots = X^n = \text{Prop}$.

We can define an operator $R$ to "lift" quantifiers to the appropriate level to combine with a relation.

$$R^2 \text{Quant}^X = \cdots = (X = T) = T$$

Rules governing $R$:

$$Q^2 \text{Quant}^X \upuparrows \text{Prop}! R^{Q^r} = Q^{r} R^{Q^r}$$

Now we can compose representations of $n$ quantifiers with a relation $r$ using $R^{Q^1} (R^{Q^2} \cdots (R^{Q^n} \cdots))$. 
Quantifiers and Arity Reduction

We take quantifiers to be arity reduction operators (Keenan 1992; van Eijck 2003)

- Quantifiers (representations of noun phrases) are of type
  
  \((X \rightarrow \text{Prop}) \rightarrow \text{Prop}\), which we shall write \(\text{Quant}^X\) for clarity (where \(X\) is typically \(B\)).

- Core propositional relations (such as verbs) are of type
  
  \(X_1 \rightarrow \ldots \rightarrow X_n \rightarrow \text{Prop}\)

Rules governing \(R\):

\[Q^2 \text{Quant}^X \overset{r}{\rightarrow} T \overset{r}{\rightarrow} T \overset{r}{\rightarrow} \]

Now we can compose representations of \(n\) quantifiers with a relation \(r\) using \(RQ\):

\[RQ_{r_n} \overset{r_{n-1}}{\rightarrow} \ldots \overset{r_1}{\rightarrow} Q_{r_2} \overset{r}{\rightarrow} Q_{r_2} \overset{r}{\rightarrow} \text{Prop}\]
Quantifiers and Arity Reduction

We take quantifiers to be arity reduction operators (Keenan 1992; van Eijck 2003)

- Quantifiers (representations of noun phrases) are of type $(X \rightarrow \text{Prop}) \rightarrow \text{Prop}$, which we shall write $\text{Quant}^X$ for clarity (where $X$ is typically $B$).
- Core propositional relations (such as verbs) are of type $X_1 \rightarrow \ldots \rightarrow X_n \rightarrow \text{Prop}$
- We can define an operator $R$ to “lift” quantifiers to the appropriate level to combine with a relation.

$$R \in \text{Quant}^X \Rightarrow ((X \rightarrow T) \rightarrow T)$$
Quantifiers and Arity Reduction

We take quantifiers to be arity reduction operators (Keenan 1992; van Eijck 2003)

- Quantifiers (representations of noun phrases) are of type
  \((X \implies \text{Prop}) \implies \text{Prop}\), which we shall write \(\text{Quant}^X\) for clarity (where \(X\) is typically \(B\)).

- Core propositional relations (such as verbs) are of type
  \(X_1 \implies \ldots \implies X_n \implies \text{Prop}\)

- We can define an operator \(R\) to “lift” quantifiers to the appropriate level to combine with a relation.

\[
R \in \text{Quant}^X \implies ((X \implies T) \implies T)
\]

- Rules governing \(R\):

\[
Q \in \text{Quant}^X \land r \in (X \implies \text{Prop}) \quad \rightarrow \quad RQr = Qr
\]

\[
Q \in \text{Quant}^X \land r \in (X \implies T) \land r \notin (X \implies \text{Prop}) \quad \rightarrow \quad RQr = \lambda xRQ(rx)
\]
Quantifiers and Arity Reduction

We take quantifiers to be arity reduction operators (Keenan 1992; van Eijck 2003)

- Quantifiers (representations of noun phrases) are of type
  \((X \rightarrow \text{Prop}) \rightarrow \text{Prop}\), which we shall write \(\text{Quant}^X\) for clarity (where \(X\) is typically \(B\)).

- Core propositional relations (such as verbs) are of type
  \(X_1 \rightarrow \ldots \rightarrow X_n \rightarrow \text{Prop}\)

- We can define an operator \(R\) to “lift” quantifiers to the appropriate level to combine with a relation.

  \[ R \in \text{Quant}^X \rightarrow ((X \rightarrow T) \rightarrow T) \]

- Rules governing \(R\):

  \[ Q \in \text{Quant}^X \land r \in (X \rightarrow \text{Prop}) \quad \rightarrow RQr = Qr \]

  \[ Q \in \text{Quant}^X \land r \in (X \rightarrow T) \land r \notin (X \rightarrow \text{Prop}) \quad \rightarrow RQr = \lambda x RQ(rx) \]

- Now we can compose representations of \(n\) quantifiers with a relation \(r\) using

  \[ RQ_1(RQ_2 \ldots (RQ_n r) \ldots) \]
Underspecification
The Problem

- Natural language is ambiguous with respect to the scoping of quantifiers, modifiers, conjunction, and negation.
- Many of these scopings are purely semantic in nature.
- Standard example:

  \[
  \text{Every man loves a woman}
  \]

  is this:

  1. \(\forall x (\text{man}'(x) \rightarrow \exists y (\text{woman}'(y) \land \text{loves}'(x, y)))\) or;
  2. \(\exists y (\text{woman}'(y) \land \forall x (\text{man}'(x) \land \text{loves}'(x, y)))\)?

- We need a semantic theory that can account for these different scopings.
- This can be achieved by producing *underspecified* representations that subsume all the various readings, and from which the different readings can be generated.
Underspecification in PTCT
An Observation

• We can express computable functions in PTCT, therefore we can incorporate the machinery of underspecified semantics directly into the representation language.
Indexed Permutations

- \textit{perms} is a function that generates all $k!$ permutations of a $k$-ary product term $\langle t_1, \ldots, t_k \rangle$. (Campbell 2004)
Indexed Permutations

- *perms* is a function that generates all \( k! \) permutations of a \( k \)-ary product term \( \langle t_1, \ldots, t_k \rangle \). (Campbell 2004)
- Campbell encodes in his function a deterministic algorithm that maps a \( k \)-element list \( L \) to an indexed \( k! \) list \( \text{perms}(L) \), where each element \( e_i \in \text{perms}(L) \) is the \( i \)-th permutation in \( L \) produced by the algorithm.
Indexed Permutations

- \textit{perms} is a function that generates all $k!$ permutations of a $k$-ary product term $\langle t_1, \ldots, t_k \rangle$. (Campbell 2004)

- Campbell encodes in his function a deterministic algorithm that maps a $k$-element list $L$ to an indexed $k!$ list \textit{perms}(\textit{L}), where each element $e_i \in \textit{perms}(\textit{L})$ is the $i$-th permutation in $L$ produced by the algorithm.

- Building on this work we specify a function \textit{perms\_scope} that generates all $k!$ indexed permutation products of a $k$-ary indexed product term $\langle t_1, \ldots, t_k \rangle$ as part of the procedure for generating the set of possible scope readings of a sentence.
Adapting Indexed Permutations

- For our treatment of underspecification, \( perms_{scope} \) needs to take
Adapting Indexed Permutations

- For our treatment of underspecification, \( perms_{scope} \) needs to take
  1. a \( k \)-ary product of scope taking elements (in the order they appear in the surface syntax)
Adapting Indexed Permutations

• For our treatment of underspecification, \( perms\_scope \) needs to take
  1. a \( k \)-ary product of scope taking elements (in the order they appear in the surface syntax)
  2. a \( k \)-ary relation (representing the core proposition),
Adapting Indexed Permutations

For our treatment of underspecification, \textit{perms\_scope} needs to take

1. a \textit{k}-ary product of scope taking elements (in the order they appear in the
   surface syntax)
2. a \textit{k}-ary relation (representing the core proposition),

and produces the \textit{k!}-ary product of scoped readings—using the relation reduction
approach for combining the quantifiers with the core proposition.
Adapting Indexed Permutations

- For our treatment of underspecification, \textit{perms\textsubscript{scope}} needs to take
  1. a \(k\)-ary product of scope taking elements (in the order they appear in the surface syntax)
  2. a \(k\)-ary relation (representing the core proposition),

and produces the \(k!\)-ary product of scoped readings—using the relation reduction approach for combining the quantifiers with the core proposition.

- When a \(k\)-tuple of quantifiers is permuted, the \(\lambda\)-operators that bind the quantified argument positions in the core relation are permuted in the same order as the quantifiers in the \(k\)-tuple.
Adapting Indexed Permutations

- For our treatment of underspecification, \(perms_{scope}\) needs to take
  1. a \(k\)-ary product of scope taking elements (in the order they appear in the surface syntax)
  2. a \(k\)-ary relation (representing the core proposition),
and produces the \(k!\)-ary product of scoped readings—using the relation reduction approach for combining the quantifiers with the core proposition.

- When a \(k\)-tuple of quantifiers is permuted, the \(\lambda\)-operators that bind the quantified argument positions in the core relation are permuted in the same order as the quantifiers in the \(k\)-tuple.

- This correspondence is necessary to preserve the connection between each GQ and its argument position in the core relation across scope permutations.
Adapting Indexed Permutations

- A scope reading is generated by applying the elements of the $k$-tuple of quantifiers in sequence to the core proposition, reducing its arity with each such operation until a proposition results.
Adapting Indexed Permutations

- A scope reading is generated by applying the elements of the $k$-tuple of quantifiers in sequence to the core proposition, reducing its arity with each such operation until a proposition results.
- This approach builds on the work of van Eijck (2003).
Adapting Indexed Permutations

- A scope reading is generated by applying the elements of the $k$-tuple of quantifiers in sequence to the core proposition, reducing its arity with each such operation until a proposition results.
- This approach builds on the work of van Eijck (2003).
- The scope taking elements and the core representation can be combined into a single product, e.g. as a pair consisting of the quantifiers and the core representation, which provides an argument for $perms_{scope}$. 
Selecting a Reading and Lazy Evaluation

Projecting a reading from the underspecified representation

- The $i$th scope reading is produced by projecting the $i$th element of the product of propositions that is the output of our $\text{perms\_scope}$ function.

- In an implementation of PTCT it is not necessary to compute the full $k!$-ary product of permutations. We can adopt lazy evaluation of our scope reading function to identify the $i$th permutation, as in the application of a Haskell function to a list.

- Therefore, the PTCT term consisting of the application of $\text{perms\_scope}$ to an input pair of a $k$-tuple of GQs and a core relation (that is, $\text{perms\_scope}(Q_1, \ldots, Q_n, r_n)$) provides an underspecified representation of the sentence.
An Example

- Taking our example “Every man loves a woman”, the scope taking elements are represented by the quantifiers

\[ Q_1 = \lambda P \forall x \epsilon B(\text{man}'(x) \rightarrow P(x)) \]
\[ Q_2 = \lambda Q \exists y \epsilon B(\text{woman}'(y) \land Q(y)) \]

and the core representation by

\[ \lambda uv.\text{loves}'uv \]
An Example

- The permutations of the quantifiers and the core representation that
  \( perms_{\text{scope}}(\langle\langle Q_1, Q_2 \rangle, \lambda uv.\text{loves}'uv \rangle) \) produces is

  \[
  \langle\langle Q_1, Q_2 \rangle, \lambda uv.\text{loves}'uv \rangle \\
  \langle\langle Q_2, Q_1 \rangle, \lambda vu.\text{loves}'uv \rangle
  \]
An Example

- The permutations of the quantifiers and the core representation that \( \text{perms}_{\text{scope}}(\langle Q_1, Q_2 \rangle, \lambda uv.\text{loves}^\prime uv) \) produces is

  \[
  \langle \langle Q_1, Q_2 \rangle, \lambda uv.\text{loves}^\prime uv \rangle \\
  \langle \langle Q_2, Q_1 \rangle, \lambda vu.\text{loves}^\prime uv \rangle 
  \]

- Applying the relation reduction approach to computing the final propositions gives us a product containing the two readings:

  \[
  \langle \forall x \in B(\text{man}^\prime(x) \Rightarrow \exists y \in B(\text{woman}^\prime(y) \land \text{loves}^\prime(x, y))), \\
  \exists y \in B(\text{woman}^\prime(y) \land \forall x \in B(\text{man}^\prime(x) \land \text{loves}^\prime(x, y))) \rangle 
  \]
An Alternative View

Combining permutation and selection

As an alternative to permutation followed by projection, we can define $\text{permute}_k^i$ that directly computes the $i$th permutation of a $k$-ary product.
An Alternative View

Combining permutation and selection

- As an alternative to permutation followed by projection, we can define $\text{permute}_k^i$ that directly computes the $i$th permutation of a $k$-ary product.

- Given an integer $i$, its type will be

$$\prod X_1 \ldots \prod X_k(\langle \text{Quant}^{X_1} \otimes \ldots \text{Quant}^{X_k} \rangle \Rightarrow (X_1 \Rightarrow \ldots X_k \Rightarrow \text{Prop}) \Rightarrow \text{Prop})$$
Typing the Representations

Typing underspecified representations

- If we adopt the first approach to underspecified representations using \(\text{perms\_scope}\), we could give them a uniform type by defining arbitrary-arity product types.
Typing the Representations

Typing underspecified representations

- If we adopt the first approach to underspecified representations using $\text{perms\_scope}$, we could give them a uniform type by defining arbitrary-arity product types.

- Underspecified representations on the alternative account with $\text{permute}_k^i$—where we compute the $i$th permutation directly—can be taken to be (partial) functions from integers to propositions.
Typing the Representations

Typing underspecified representations

- If we adopt the first approach to underspecified representations using \( \textit{perms\_scope} \), we could give them a uniform type by defining arbitrary-arity product types.

- Underspecified representations on the alternative account with \( \textit{permute}_i \) —where we compute the \( i \)th permutation directly—can be taken to be (partial) functions from integers to propositions.

- If we make the function complete, then we can define the type of these representations as

\[
N \Rightarrow \text{Prop}
\]
Typing the Representations

Typing underspecified representations

- If we adopt the first approach to underspecified representations using $perms_{scope}$, we could give them a uniform type by defining arbitrary-arity product types.

- Underspecified representations on the alternative account with $permute^i_k$ —where we compute the $i$th permutation directly—can be taken to be (partial) functions from integers to propositions.

- If we make the function complete, then we can define the type of these representations as

  $$N \Rightarrow \text{Prop}$$

- One way in which the function can be made complete is if $permute^i_k$ selects the $(i \mod k)$th permutation.
**Polymorphism**

*Why polymorphism is useful in this problem*

- By using product types rather than monomorphic lists as the basis for computing permutations we are able to deal with cases where the core representation is a relation over arguments of different types.

- This is exemplified by sentences involving relations between individuals and propositions,

  Someone believes everything that Mary believes.

  and sentences asserting relations between abstract objects and numbers

  The algorithm assigns an integer to each theorem in the logic.
A More Complex Case of Polymorphism

Polymorphic quantifiers: implicit v. explicit polymorphism

- We are assuming that there is some way of encoding polymorphism in the type of the quantifiers representing noun phrases, so we have $(B \rightarrow \text{Prop}) \rightarrow \text{Prop}$ and $(\text{Prop} \rightarrow \text{Prop}) \rightarrow \text{Prop}$, for example.
A More Complex Case of Polymorphism

Polymorphic quantifiers: implicit v. explicit polymorphism

- We are assuming that there is some way of encoding polymorphism in the type of the quantifiers representing noun phrases, so we have \((B \leftrightarrow \text{Prop}) \rightarrow \text{Prop}\) and \((\text{Prop} \leftrightarrow \text{Prop}) \rightarrow \text{Prop}\), for example.
- Usually the type is fixed by the determiner, independent of the common noun.
A More Complex Case of Polymorphism

Polymorphic quantifiers: implicit v. explicit polymorphism

- We are assuming that there is some way of encoding polymorphism in the type of the quantifiers representing noun phrases, so we have \((B \rightarrow \text{Prop}) \rightarrow \text{Prop}\) and \((\text{Prop} \leftrightarrow \text{Prop}) \rightarrow \text{Prop}\), for example.

- Usually the type is fixed by the determiner, independent of the common noun.

- We can allow the type of the common noun (property) argument of a quantifier to determine the type of the quantifier.
A More Complex Case of Polymorphism

Polymorphic quantifiers: implicit v. explicit polymorphism

- We are assuming that there is some way of encoding polymorphism in the type of the quantifiers representing noun phrases, so we have \((B \rightarrow \text{Prop}) \rightarrow \text{Prop}\) and \((\text{Prop} \rightarrow \text{Prop}) \rightarrow \text{Prop}\), for example.
- Usually the type is fixed by the determiner, independent of the common noun.
- We can allow the type of the common noun (property) argument of a quantifier to determine the type of the quantifier.
- As it stands, this is not directly expressible in PTCT, which adopts implicit polymorphism.
A More Complex Case of Polymorphism

Polymorphic quantifiers: implicit v. explicit polymorphism

- We are assuming that there is some way of encoding polymorphism in the type of the quantifiers representing noun phrases, so we have \((B \rightarrow \text{Prop}) \rightarrow \text{Prop}\) and \((\text{Prop} \leftrightarrow \text{Prop}) \rightarrow \text{Prop}\), for example.

- Usually the type is fixed by the determiner, independent of the common noun.

- We can allow the type of the common noun (property) argument of a quantifier to determine the type of the quantifier.

- As it stands, this is not directly expressible in PTCT, which adopts implicit polymorphism.

- We will consider extending our type system to allow for explicit or parametric polymorphic typing.
Polymorphic quantifiers: implicit v. explicit polymorphism

- We are assuming that there is some way of encoding polymorphism in the type of the quantifiers representing noun phrases, so we have \((B \rightarrow Prop) \rightarrow Prop\) and \((Prop \leftrightarrow Prop) \rightarrow Prop\), for example.
- Usually the type is fixed by the determiner, independent of the common noun.
- We can allow the type of the common noun (property) argument of a quantifier to determine the type of the quantifier.
- As it stands, this is not directly expressible in PTCT, which adopts implicit polymorphism.
- We will consider extending our type system to allow for explicit or parametric polymorphic typing.
- The type of a determiner would then be \(Y \rightarrow Prop\), where \(Y\) is intended to be a place-holder for some property type.
A More Complex Case of Polymorphism

Polymorphic quantifiers: implicit v. explicit polymorphism

- We are assuming that there is some way of encoding polymorphism in the type of the quantifiers representing noun phrases, so we have \((B \rightarrow \text{Prop}) \rightarrow \text{Prop}\) and \((\text{Prop} \leftrightarrow \text{Prop}) \rightarrow \text{Prop}\), for example.
- Usually the type is fixed by the determiner, independent of the common noun.
- We can allow the type of the common noun (property) argument of a quantifier to determine the type of the quantifier.
- As it stands, this is not directly expressible in PTCT, which adopts implicit polymorphism.
- We will consider extending our type system to allow for explicit or parametric polymorphic typing.
- The type of a determiner would then be \(Y \rightarrow \text{Prop}\), where \(Y\) is intended to be a place-holder for some property type.
- The determiner’s representation would require a type parameter, or constraint, that fixes \(Y\) to be the type \((A \rightarrow \text{Prop})\) of the common noun with which it combines.
Constraints on Scope Readings

- Lexically based scope restrictions
Constraints on Scope Readings

- Lexically based scope restrictions
  - The strongly preferred reading of *Every critic reviewed a certain book* is the one on which *a certain book* takes wide scope relative to *every critic.*
Constraints on Scope Readings

- Lexically based scope restrictions
  - The strongly preferred reading of *Every critic reviewed a certain book* is the one on which *a certain book* takes wide scope relative to *every critic*.

- Syntactic scope restrictions:
Constraints on Scope Readings

- Lexically based scope restrictions
  - The strongly preferred reading of *Every critic reviewed a certain book* is the one on which *a certain book* takes wide scope relative to *every critic*.

- Syntactic scope restrictions:
  - *every assignment* can only take narrow scope relative to *a student who completed every assignment* in *A student who completed every assignment came first in the class*. 
Constraints on Scope Readings

Restrictions on scope readings can be stated as filters on the product values of \( \text{perms}_\text{scope} \).

- Let \( \langle \text{Quants}, \text{Rel} \rangle \) be a variable ranging over pairs in which \( \text{Quant} \) is a \( k \)-tuple and \( \text{Rel} \) is a \( k \)-ary relation.
Constraints on Scope Readings

Restrictions on scope readings can be stated as filters on the product values of \( \text{perms}_{-}\text{scope} \).

- Let \( \langle \text{Quants}, \text{Rel} \rangle \) be a variable ranging over pairs in which \( \text{Quant} \) is a \( k \)-tuple and \( \text{Rel} \) is a \( k \)-ary relation.
- \( a_{\text{certain}} \) is a PTCT property that is true of all and only GQs that represent an item \( a_{\text{certain}} N' \), and is false of anything else.
Constraints on Scope Readings

Restrictions on scope readings can be stated as filters on the product values of \textit{perms\_scope}.

- Let \( \langle Quants, Rel \rangle \) be a variable ranging over pairs in which \textit{Quant} is a \( k \)-tuple and \textit{Rel} is a \( k \)-ary relation.
- \textit{a\_certain} is a \textit{PTCT} property that is true of all and only \textit{GQs} that represent a certain \( N' \), and is false of anything else.
- \textit{tuple\_element}(i, Quants) = \( Q_i \) if \( Q_i \) is a member of \textit{Quants}, and the distinguished term \( \omega \) otherwise.
Constraints on Scope Readings

Restrictions on scope readings can be stated as filters on the product values of `perms_scope`.

- Let \( \langle \text{Quants}, \text{Rel} \rangle \) be a variable ranging over pairs in which \( \text{Quant} \) is a \( k \)-tuple and \( \text{Rel} \) is a \( k \)-ary relation.
- \( a_{\text{certain}} \) is a PTCT property that is true of all and only GQs that represent \( a \ \text{certain} \ N' \), and is false of anything else.
- \( \text{tuple}_\text{element}(i, \text{Quants}) = Q_i \) if \( Q_i \) is a member of \( \text{Quants} \), and the distinguished term \( \omega \) otherwise.
- \( i \) and \( j \) are variables ranging over integers (they are of the type Num).
Constraints on Scope Readings

Restrictions on scope readings can be stated as filters on the product values of \( \text{perms\_scope} \).

- Let \( \langle \text{Quants}, \text{Rel} \rangle \) be a variable ranging over pairs in which \( \text{Quant} \) is a \( k \)-tuple and \( \text{Rel} \) is a \( k \)-ary relation.
- \( a\_\text{certain} \) is a PTCT property that is true of all and only GQs that represent a certain \( N' \), and is false of anything else.
- \( \text{tuple\_element}(i, \text{Quants}) = Q_i \) if \( Q_i \) is a member of \( \text{Quants} \), and the distinguished term \( \omega \) otherwise.
- \( i \) and \( j \) are variables ranging over integers (they are of the type Num).
- The constraint can be expressed by the filter:

\[
\lambda \langle \text{Quants}, \text{Rel} \rangle . \sim (\exists i \exists j (a\_\text{certain}(\text{tuple\_element}(i, \text{Quants})) \land \\
\sim a\_\text{certain}(\text{tuple\_element}(j, \text{Quants})) \land j < i))
\]
Constraints on Scope Readings

A syntactic filter

- Let $relcl_{\text{embed}}(Q_1, Q_2)$ hold iff the NP corresponding to $Q_2$ appears in a relative clause contained in the NP corresponding to $Q_1$. 

In both filters we have only quantified over integers (elements of the type $\text{Num}$). We have taken advantage of the isomorphism between $k$-tuples of integers and $k$-tuples of indexed GQs to avoid quantifying over GQ expressions. Therefore, we have remained within the first-order expressive resources of PTCT.
Constraints on Scope Readings

A syntactic filter

- Let $\text{relcl} \_ \text{embed}(Q_1, Q_2)$ hold iff the NP corresponding to $Q_2$ appears in a relative clause contained in the NP corresponding to $Q_1$.

- The constraint can be expressed by the filter:

$$\lambda\langle\text{Quants}, \text{Rel}\rangle.
\neg(\exists i \exists j (\text{relcl} \_ \text{embed}(\text{tuple} \_ \text{element}(i, \text{Quants}), \text{tuple} \_ \text{element}(j, \text{Quants}))
\land j < i))$$
Constraints on Scope Readings

A syntactic filter

- Let $relcl_{-}embed(Q_1, Q_2)$ hold iff the NP corresponding to $Q_2$ appears in a relative clause contained in the NP corresponding to $Q_1$.

- The constraint can be expressed by the filter:

  $$\lambda\langle Quants, Rel \rangle.
  
  \sim (\exists i \exists j (relcl_{-}embed(tuple_{-}element(i, Quants), tuple_{-}element(j, Quants)) \land j < i))$$

- In both filters we have only quantified over integers (elements of the type Num).
Constraints on Scope Readings

A syntactic filter

- Let $relcl\_embed(Q_1, Q_2)$ hold iff the NP corresponding to $Q_2$ appears in a relative clause contained in the NP corresponding to $Q_1$.

- The constraint can be expressed by the filter:

  $$\lambda \langle Quants, Rel \rangle. \sim (\exists i \exists j (relcl\_embed(tuple\_element(i, Quants), tuple\_element(j, Quants)) \land j < i))$$

- In both filters we have only quantified over integers (elements of the type Num).
- We have taken advantage of the isomorphism between $k$-tuples of integers and $k$-tuples of indexed GQs to avoid quantifying over GQ expressions.
Comparison with Other Theories

Storage

- Cooper Storage (1983)
Comparison with Other Theories

Storage

- Cooper Storage (1983)
  - Quantificational arguments are placed in a set (a store), and associated with indexed variables in a propositional core.
Comparison with Other Theories

Storage

- Cooper Storage (1983)
  - Quantificational arguments are placed in a set (a store), and associated with indexed variables in a propositional core.
  - The ordered pair of the store set and the propositional core is an underspecified representation of the sentence.
Comparison with Other Theories

Storage

- Cooper Storage (1983)
  - Quantificational arguments are placed in a set (a store), and associated with indexed variables in a propositional core.
  - The ordered pair of the store set and the propositional core is an underspecified representation of the sentence.
  - The elements of the store are retrieved and applied to the propositional core in any order which yields proper binding of their associated variables.
Comparison with Other Theories

Storage

- Most students read two articles.
Comparison with Other Theories

Storage

- Most students read two articles.
  - \{\langle \text{most\_students'}, x \rangle, \langle \text{two\_articles'}, y \rangle \} \vdash \text{read'}(x, y)
Comparison with Other Theories

Storage

- Most students read two articles.
  - \{⟨most_students’, x⟩, ⟨two_articles’, y⟩\} ⊨ read’(x, y)
  - most_students’(λx.two_articles’(λy.read’(x, y)))
Comparison with Other Theories

Storage

- Most students read two articles.
  - \{\langle \text{most\_students'}, x \rangle, \langle \text{two\_articles'}, y \rangle \} \vdash \text{read'}(x, y)
  - \text{most\_students'}(\lambda x.\text{two\_articles'}(\lambda y.\text{read'}(x, y)))
  - \text{two\_articles'}(\lambda y.\text{most\_students'}(\lambda x.\text{read'}(x, y)))
Comparison with Other Theories

Storage

- *Most students read two articles.*
  - \( \{ \langle \text{most\_students}', x \rangle, \langle \text{two\_articles}', y \rangle \} \vdash \text{read'}(x, y) \)
  - \( \text{most\_students}'(\lambda x.\text{two\_articles}'(\lambda y.\text{read'}(x, y))) \)
  - \( \text{two\_articles}'(\lambda y.\text{most\_students}'(\lambda x.\text{read'}(x, y))) \)
- Keller Storage (1988)
Comparison with Other Theories

Storage

- **Most students read two articles.**
  - \{\langle\text{most\_students}', x\rangle, \langle\text{two\_articles}', y\rangle\} \vdash \text{read}'(x, y)
  - \text{most\_students}'(\lambda x.\text{two\_articles}'(\lambda y.\text{read}'(x, y)))
  - \text{two\_articles}'(\lambda y.\text{most\_students}'(\lambda x.\text{read}'(x, y)))

- Keller Storage (1988)
  - Creates nested stores to encode scope constraints.
Comparison with Other Theories

Hole Semantics

Comparison with Other Theories

Hole Semantics

- Scope-taking elements and the variables they bind in FOL formulae are replaced by indexed meta-variables, to yield an underspecified propositional core.
Comparison with Other Theories

Hole Semantics

  - Scope-taking elements and the variables they bind in FOL formulae are replaced by indexed meta-variables, to yield an underspecified propositional core.
  - A partial order is imposed on these elements which entails constraints on the sequences in which they can be realised within the formalae.
Comparison with Other Theories

Hole Semantics

- Every student wrote a program.
Comparison with Other Theories

Hole Semantics

- Every student wrote a program.

\[
\{l_1, l_2, l_3, h_0, h_1, h_2\},
\]

\[
\{l_1 : \forall x(\text{student}'(x) \rightarrow h_1), l_2 : \exists y(\text{program}'(y) \land h_2), l_3 : \text{wrote}'(x, y)\},
\]

\[
\{l_1 \leq h_0, l_2 \leq h_0, l_3 \leq h_1, l_3 \leq h_2\}\]
Comparison with Other Theories

Hole Semantics

• Every student wrote a program.
  • 
  \[
  \langle\{l_1, l_2, l_3, h_0, h_1, h_2\},
  \{l_1 : \forall x (\text{student}'(x) \rightarrow h_1), l_2 : \exists y (\text{program}'(y) \land h_2), l_3 : \text{wrote}'(x, y)\},
  \{l_1 \leq h_0, l_2 \leq h_0, l_3 \leq h_1, l_3 \leq h_2\}\rangle
  \]
  • \forall x (\text{student}'(x) \rightarrow \exists y (\text{program}'(y) \land \text{wrote}'(x, y)))
Comparison with Other Theories

Hole Semantics

- Every student wrote a program.

\[
\begin{align*}
\langle \{l_1, l_2, l_3, h_0, h_1, h_2\}, \\
\{l_1 \trianglerighteq \forall x (\text{student'}(x) \rightarrow h_1), l_2 \trianglerighteq \exists y (\text{program'}(y) \wedge h_2), l_3 \trianglerighteq \text{wrote'}(x, y)\}, \\
\{l_1 \leq h_0, l_2 \leq h_0, l_3 \leq h_1, l_3 \leq h_2\}\rangle
\end{align*}
\]

- \(\forall x (\text{student'}(x) \rightarrow \exists y (\text{program'}(y) \wedge \text{wrote'}(x, y)))\)

- \(\exists y (\text{program'}(y) \wedge \forall x (\text{student'}(x) \rightarrow \text{wrote'}(x, y)))\)

Copestake et al. (1997) implement Bos and Reyle strategy in a typed-feature structure grammar.
Comparison with Other Theories

Hole Semantics

- Every student wrote a program.

\[
\langle\{l_1, l_2, l_3, h_0, h_1, h_2\},
\{l_1 : \forall x (\text{student}'(x) \rightarrow h_1), l_2 : \exists y (\text{program}'(y) \land h_2), l_3 : \text{wrote}'(x, y)\},
\{l_1 \leq h_0, l_2 \leq h_0, l_3 \leq h_1, l_3 \leq h_2\}\rangle
\]

- \(\forall x (\text{student}'(x) \rightarrow \exists y (\text{program}'(y) \land \text{wrote}'(x, y)))\)

- \(\exists y (\text{program}'(y) \land \forall x (\text{student}'(x) \rightarrow \text{wrote}'(x, y)))\)

- Copestake et al. (1997) implement Bos and Reyle strategy in a typed-feature structure grammar.
Comparison with Other Theories

Glue Languages

- Dalrymple et. al (1999) and Crouch and van Genabith (1999) suggest
Comparison with Other Theories

Glue Languages

- Dalrymple et. al (1999) and Crouch and van Genabith (1999) suggest a theory in which representations of GQs and core relations are expressed as premises in an underspecified semantic glue language, and
Comparison with Other Theories

Glue Languages

- Dalrymple et. al (1999) and Crouch and van Genabith (1999) suggest
  - a theory in which representations of GQs and core relations are expressed as premises in an underspecified semantic glue language, and
  - the natural deduction rules of linear logic can apply to premises in different orders of derivation to generate alternative scope readings.
Comparison with Other Theories

Limitations of these Theories

- The underspecified representations that these theories posit are objects in a metalanguage rather than expression of the semantic representation language.
Comparison with Other Theories

Limitations of these Theories

- The underspecified representations that these theories posit are objects in a metalanguage rather than expression of the semantic representation language.
- They require powerful metalinguistic operations to generate specified scope readings.
Comparison with Other Theories

Limitations of these Theories

• The underspecified representations that these theories posit are objects in a metalanguage rather than expression of the semantic representation language.

• They require powerful metalinguistic operations to generate specified scope readings.

• They are generally expressively incomplete in the sense of Ebert 2003 in that they cannot exclude certain scope readings due the absence of Boolean conditions and quantification in their constraint languages.
Comparison with Other Theories

Limitations of these Theories

- The underspecified representations that these theories posit are objects in a metalanguage rather than expression of the semantic representation language.
- They require powerful metalinguistic operations to generate specified scope readings.
- They are generally expressively incomplete in the sense of Ebert 2003 in that they cannot exclude certain scope readings due to the absence of Boolean conditions and quantification in their constraint languages.
- PTCT avoids these problems.
Comparison with Other Theories

*PTCT avoids these problems.*

- Underspecified representations are terms of the representation language.
Comparison with Other Theories

*PTCT avoids these problems.*

- Underspecified representations are terms of the representation language.
- No metalinguistic operations are required to project a defined scope reading from an underspecified representation.
Comparison with Other Theories

\textit{PTCT avoids these problems.}

- Underspecified representations are terms of the representation language.
- No metalinguistic operations are required to project a defined scope reading from an underspecified representation.
- As constraints on possible scope readings are encoded in filters which are $\lambda$-terms of PTCT, the full expressive power of the representation language is available for stating these conditions.
Conclusions
Conclusions

- We have presented a treatment of underspecified representation within PTCT which:
Conclusions

- We have presented a treatment of underspecified representation within PTCT which:
  - uses product types rather than lists to represent scope sequences;
Conclusions

- We have presented a treatment of underspecified representation within PTCT which:
  - uses product types rather than lists to represent scope sequences;
  - invokes a function from the initial scope sequence of a sentence and its core relation to an indexed $k!$-product of possible scope sequences to provide underspecified representations within the language, rather than by means of meta-linguistic operations.
Conclusions

- We have presented a treatment of underspecified representation within PTCT which:
  - uses product types rather than lists to represent scope sequences;
  - invokes a function from the initial scope sequence of a sentence and its core relation to an indexed $k!$-product of possible scope sequences to provide underspecified representations within the language, rather than by means of meta-linguistic operations.

- We have also:
Conclusions

- We have presented a treatment of underspecified representation within PTCT which:
  - uses product types rather than lists to represent scope sequences;
  - invokes a function from the initial scope sequence of a sentence and its core relation to an indexed $k!$-product of possible scope sequences to provide underspecified representations within the language, rather than by means of meta-linguistic operations.

- We have also:
  - provided a version of our underspecified semantics where the underspecified representation is of a uniform type $N \Rightarrow \text{Prop}$;
Conclusions

- We have presented a treatment of underspecified representation within PTCT which:
  - uses product types rather than lists to represent scope sequences;
  - invokes a function from the initial scope sequence of a sentence and its core relation to an indexed $k!$-product of possible scope sequences to provide underspecified representations within the language, rather than by means of meta-linguistic operations.

- We have also:
  - provided a version of our underspecified semantics where the underspecified representation is of a uniform type $N \Rightarrow \text{Prop}$;
  - accommodated polymorphic quantifier arguments of the propositional core;
Conclusions

- We have presented a treatment of underspecified representation within PTCT which:
  - uses product types rather than lists to represent scope sequences;
  - invokes a function from the initial scope sequence of a sentence and its core relation to an indexed $k!$-product of possible scope sequences to provide underspecified representations within the language, rather than by means of meta-linguistic operations.

- We have also:
  - provided a version of our underspecified semantics where the underspecified representation is of a uniform type $N \Rightarrow \text{Prop}$;
  - accommodated polymorphic quantifier arguments of the propositional core;
  - indicated how we might deal with polymorphism of quantificational determiners;
Conclusions

- We have presented a treatment of underspecified representation within PTCT which:
  - uses product types rather than lists to represent scope sequences;
  - invokes a function from the initial scope sequence of a sentence and its core relation to an indexed $k!$-product of possible scope sequences to provide underspecified representations within the language, rather than by means of meta-linguistic operations.
  - We have also:
    - provided a version of our underspecified semantics where the underspecified representation is of a uniform type $N \Rightarrow \text{Prop}$;
    - accommodated polymorphic quantifier arguments of the propositional core;
    - indicated how we might deal with polymorphism of quantificational determiners;
    - encoded scope constraints as filters on the $k!$ permutation product of possible scope readings.
The End