Underspecified Interpretations in a Curry-Typed 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

December 2004

Abstract

In previous work we have developed Property Theory with Curry Typing (PTCT), an intensional first-order logic for natural language semantics. PTCT permits fine-grained specifications of meaning. It also supports polymorphic types and separation types.[1] We develop an intensional number theory within PTCT in order to represent proportional generalized quantifiers like "*most*", and we suggest a dynamic type-theoretic approach to anaphora and ellipsis resolution. Here we extend the type system to include product types, and use these to define a permutation function that generates underspecified scope representations within PTCT. We indicate how filters can be added to encode constraints on possible scope readings. Our account offers several important advantages over other current theories of underspecification.

1 Introduction

In a series of recent papers (Fox & Lappin, 2001; Fox, Lappin, & Pollard, 2002a, 2002b; Fox & Lappin, 2003a, 2003b) and a forthcoming monograph (Fox & Lappin, to appear) we present a new model for the computational semantics of natural language, Property Theory with Curry Typing (PTCT). This theory allows us to express fine-grained distinctions of meaning. It also supports a unified dynamic treatment of anaphora and ellipsis.

In this paper we will show that by extending the type system of PTCT to include product types, PTCT provides us with the resources for generating underspecified semantic representations within the representation language, rather than through meta-language devices, as in most current treatments of underspecification (Reyle, 1993;

[*]An earlier version of this paper was presented at the Workshop on Lambda Calculus, Type Theory, and Natural Language in December, 2003 at King's College, London. We are grateful to the participants of this forum for useful discussion. We would also like to thank Christian Ebert and Jan van Eijck for helpful advice on a number of significant issues. The research of the second author has been supported by grant number RES–000–23–0065 from the Economic and Social Research Council of the UK.

[1]Separation types yield a form of *subtype*.

Bos, 1995; Blackburn & Bos, 2003; Copestake, Flickinger, & Sag, 1997). PTCT encodes a property theory within a language of terms (an untyped λ-calculus). We add dynamic Curry typing (Curry & Feys, 1958) and use a first-order logic to specify the truth-conditions of the propositional subpart of the term language. The resulting system gives us a language of semantic representation in which intensions (meanings) are characterized independently of modality and possible worlds. This feature of the framework permits fine-grained distinctions of meaning that cannot be captured by Montague semantics (Montague, 1974) or most of its successors.

Our semantic representation language is first-order, rather than higher-order; we achieve the sort of expressive power previously limited to higher-order theories within a formally more constrained system. This provides an effective procedure for modelling inference in natural language.

We extend PTCT in a straightforward way to express underspecified scope representations as terms within the representation language. The expressive power of the language permits the formulation of restrictions on scope readings that cannot be captured in other theories of underspecification which rely on special purpose extra-linguistic operations and a weak system for constraint specification.

2 PTCT: Syntax

2.1 Syntax of The Basic System

The core language of PTCT consists of the following sub-languages, where x ranges over a set of variables, c ranges over a set of constants, B is a basic type, and Prop characterises the type of propositions:

(1) Terms $t ::= x \mid c \mid l \mid T \mid \lambda x(t) \mid (t)t$
(logical constants) $l ::= \hat{\sim} \mid \hat{\wedge} \mid \hat{\vee} \mid \hat{\rightarrow} \mid \hat{\leftrightarrow} \mid \hat{\perp} \mid \hat{\forall} \mid \hat{\exists} \mid \hat{=}_T \mid \hat{\cong}_T \mid \epsilon T$

(2) Types $T ::= B \mid \mathsf{Prop} \mid T_1 \Longrightarrow T_2 \mid X \mid \{x \in T : \varphi'\} \mid \Pi X.T$
where X ranges over types excluding those of the form $\Pi X.T$.

(3) Wff $\varphi ::= \alpha \mid {\sim}\varphi \mid (\varphi_1 \wedge \varphi_2) \mid (\varphi_1 \vee \varphi_2) \mid (\varphi_1 \rightarrow \varphi_2) \mid (\varphi_1 \leftrightarrow \varphi_2)$
$\mid (\forall x\varphi) \mid (\exists x\varphi) \mid (\forall X\varphi) \mid (\exists X\varphi)$
(atomic wff) $\alpha ::= t =_T s \mid t \in T \mid t \cong_T s \mid {}^{\mathsf{true}}t$

PTCT is a first order logic in which types and propositions are terms over which we can quantify. This allows rich expressiveness whilst restricting the system to first order resources (Fox & Lappin, to appear, Chapter 9).

The language of terms is the untyped λ-calculus, enriched with logical constants. It is used to *represent* the interpretations of natural language expressions. It has no internal logic. With an appropriate proof theory, the simple language of types together with the language of terms can be combined to produce a Curry-typed λ-calculus.

The syntactic rules of PTCT given here are quite relaxed, and allow syntactic expressions that have no intuitively meaningful interpretation. This does not undermine the usefulness of the system. The rules give a minimal characterisation of the syntax, and the proof theory and model theory characterise the proper subset of well-formed PTCT terms that we take to represent meaningful expressions.

In a separation type $\{x \in T : \varphi'\}$ is a term representable fragment of a wff, where term representability is defined recursively, as in Fox and Lappin (2003b). This restriction on separation types avoids semantic paradoxes of type membership which could otherwise be generated in the specification of these types.

The values of bound type variables is limited to non-polymorphic types in order to avoid impredicative type membership statements.[2]

The first-order language of wffs is used to formulate type judgements for terms, and truth conditions for those terms judged to be in Prop.

It is important to distinguish between the notion of a proposition itself (in the language of wff), and that of a term that *represents* a proposition (in the language of terms). $^{\mathsf{true}}(t)$ will be a true wff whenever the proposition represented by the term t is true, and a false wff whenever the proposition represented by t is false. The representation of a proposition t ($\in \mathsf{Prop}$) is distinct from its truth conditions ($^{\mathsf{true}}(t)$). The identity criteria for propositions, taken as terms, are those of the λ-calculus with α, β, and η reduction.

It is important to realise that if $t \notin \mathsf{Prop}$, then $^{\mathsf{true}}(t)$ will be false. We enforce a strictly bivalent Boolean evaluation in the proof theory and model theory presented elsewhere (Fox & Lappin, 2004, to appear). However, in principle it is possible to modify this semantics. For example, we could take the truth value of $^{\mathsf{true}}(t)$ to be undefined when $t \notin \mathsf{Prop}$, whilst preserving Boolean negation (with the "law of excluded middle") for propositions. This would complicate our semantics, and we do not pursue this issue here.

2.2 Rules and Axioms for PTCT

The rules and axioms governing the logical behaviour of PTCT can be summarized as follows. The rules for the basic connectives of the wff have standard classical first-order behaviour. The axioms for identity of terms $=_T$ are those of α, β, and η reduction in the untyped λ-calculus. The rules for typing λ-terms are the rules/axioms of the Curry-typed calculus, augmented with rules governing those terms that represent propositions (Prop). Additional rules for the language of wffs govern the truth conditions of terms in Prop, which represent propositions. Finally, the rules for equivalence $\cong_T$ specify it as the relation of extensional equivalence.

We illustrate some of these rules as they apply to conjunction, as it appears in the language of terms ($\hat{\wedge}$), of type judgements, and of wff ($\wedge$).

(4) The basic connectives of the wff

$$\frac{\varphi \quad \psi}{\varphi \wedge \psi} \wedge i \qquad \frac{\varphi \wedge \psi}{\varphi} \wedge e \qquad \frac{\varphi \wedge \psi}{\psi} \wedge e$$

(5) Typing rules for λ-terms

$$t \in \mathsf{Prop} \wedge t' \in \mathsf{Prop} \rightarrow (t \mathbin{\hat{\wedge}} t') \in \mathsf{Prop}$$

[2]Earlier presentations of PTCT avoid impredicative types by using polymorphism *kinds* — rather than polymorphic types — where type variables range over types, but not over kinds. The simplification adopted here, of restricting the type variables directly, was suggested to us by Ray Turner.

(6) Truth conditions for Propositions

$$t \in \mathsf{Prop} \wedge t' \in \mathsf{Prop} \rightarrow ({}^{\mathsf{true}}(t \mathbin{\hat{\wedge}} t') \leftrightarrow {}^{\mathsf{true}}t \wedge {}^{\mathsf{true}}t')$$

We have encoded the proof theory of PTCT in a tableau system, which we present in Fox and Lappin (to appear, Chapter 5), together with proofs of soundness and completeness. A slightly earlier version of the proof theory appears in Fox and Lappin (2004).

2.3 Equivalence and Identity

There are two equivalence relations in this theory, intensional identity and extensional equivalence. $t \cong_T s$ states that the terms t, s are extensionally equivalent in type T. In the case where two terms t, s are propositions ($t, s \in \mathsf{Prop}$), then $t \cong_{\mathsf{Prop}} s$ corresponds to $t \leftrightarrow s$. In the case where two predicates of T are extensionally equivalent $t \cong_{(T \Longrightarrow \mathsf{Prop})} s$ then t, s each hold of all and only the same elements of T. Therefore $\forall x(x \in T \rightarrow ({}^{\mathsf{true}}t(x) \leftrightarrow {}^{\mathsf{true}}s(x)))$.

$t =_T s$ states that two terms are intensionally identical in type T. The proof system for PTCT permits us to derive $t =_T s \rightarrow t \cong_T s$ for all types inhabited by t, (s), but not $t \cong_T s \rightarrow t =_T s$. Therefore, two expressions (terms) can be provably equivalent but intensionally distinct. We have achieved this result without recourse to modal notions.

The fact that we can distinguish between equivalence and intensionality permits to sustain, for example, differences in meaning in natural language that allude other intensional logics. The precise definition of equivalence and identity are given by our proof and model theories (Fox & Lappin, to appear, Chapter 5). See Fox and Lappin (2004) for a slightly earlier version of this.

3 Underspecified Representations

3.1 Adding Product Types to PTCT

We extend the type system of PTCT to include product types $S \otimes T$, which 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.

(7) PROD: $\langle x, y \rangle \in (S \otimes T) \leftrightarrow x \in S \wedge y \in T$

Unlike monomorphic lists, the k-tuples that instantiate product types allow us to express polymorphic relations.

The appropriate notions of pairs and projections required for product types are λ-definable.

(8) $\langle x, y \rangle =_{\mathsf{def}} \lambda z(z(x)(y))$

(9) $\mathsf{fst} =_{\mathsf{def}} \lambda p(p \lambda xy(x))$

(10) $\mathsf{snd} =_{\mathsf{def}} \lambda p(p \lambda xy(y))$

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 $T_1 \otimes (T_2 \otimes (\ldots T_n) \ldots)$. We specify that for any k-tuple $\langle t_1, \ldots, t_k\rangle \in T_1 \otimes \ldots T_k$, the last element of the k-tuple, t_k is a designated object, like 0 or $\perp$. This condition insures that it is possible to recognize the end of a k-tuple and so compute its arity. The designated element of a k-tuple plays the same role as the empty list does as in the tail of every list. 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.

3.2 Generalized Quantifiers as Arity-Reducing Functions

Generalized quantifiers (GQs) represent noun phrases. We follow Keenan (1992) and van Eijck (2003) in taking GQs to be an arity reduction operator that applies to a relation r to yield either a proposition or a relation r' that is produced by effectively saturating one of r's argument with the GQ.[3] On this view, applying the GQ corresponding to "*every student*" ($\mathsf{every\_student}'$ or $\lambda P \lambda Q \forall x(\mathsf{student}' \rightarrow Q(x))$) to the binary relation $\lambda yx(\mathsf{loves}'(x, y))$ gives the one-place relation $\lambda x(\mathsf{every\_student}'(\lambda y.\mathsf{loves}'(x, y)))$. Through β-reduction this gives $\lambda x(\forall y(\mathsf{student}'(y) \rightarrow \mathsf{love}'(x, y)))$, which is the property of loving every student.

GQs are of type $(X \Longrightarrow \mathsf{Prop}) \Longrightarrow \mathsf{Prop}$, which we write Quant^X for clarity (where X is typically B). Core propositional relations, such as verbs, are of type $X_1 \Longrightarrow \ldots \Longrightarrow X_n \Longrightarrow \mathsf{Prop}$. Slightly modifying van Eijck's (2003) Haskell-based treatment of GQs, we define an operator R to "lift" quantifiers to the appropriate level to combine with a relation.

(11) $R \in \mathsf{Quant}^X \Longrightarrow ((X \Longrightarrow T) \Longrightarrow T)$

(12) $Q \in \mathsf{Quant}^X \wedge r \in (X \Longrightarrow \mathsf{Prop}) \rightarrow RQr = Qr$

(13) $Q \in \mathsf{Quant}^X \wedge r \in (X \Longrightarrow T) \wedge (T \notin \mathsf{Prop}) \rightarrow RQr = \lambda x RQ(rx)$

Now we can compose representations of n quantifiers with a relation r using $RQ_1(RQ_2 \ldots (RQ_n r) \ldots)$.

3.3 Indexed Permutations of GQ Scope Sequences

Natural language is ambiguous with respect to the scoping of quantifiers, modifiers, conjunction, and negation. Many of these scopings are purely semantic in nature. So, for example the following sentence allows two alternative scope readings.

(14) Every man loves a woman

(15) $\forall x(\mathsf{man}'(x) \rightarrow \exists y(\mathsf{woman}'(y) \wedge \mathsf{loves}'(x, y)))$

(16) $\exists y(\mathsf{woman}'(y) \wedge \forall x(\mathsf{man}'(x) \rightarrow \mathsf{loves}'(x, y)))$

[3] In Keenan's presentation, some generalised quantifiers can bind more that one of r's arguments, and so reduce its arity by more than 1. These GQ are formed from constituent quantifiers that exhibit relations of mutual dependence. Due to these relations, the GQ which they yield cannot be reduced to a simple functional composition of one quantifier with another. An example of such a GQ is $\langle$*every student, a different book*$\rangle$ in "*Every student read a different book.*".

We want our theory to produce "*underspecified*" representations that subsume all the various readings, and from which the different readings can be generated. We can express computable functions in PTCT, and so we can incorporate the machinery of underspecified semantics directly into the representation language.

We specify a function $perms\_scope_k$ 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. $perms\_scope_k$ uses a standard algorithm for mapping a sequence $\langle 1, \ldots, k\rangle$ into the indexed sequence of its permutations. We define this algorithm recursively as follows.[4]

Definition 1 (Permutation Algorithm) **Base case** *If $k = 1$, then there is no permutation and the index of the product is* 1*.*

Recursive case *(1) If $k > 1$, then the $k - 1!$ permutation products of $\langle 1, \ldots, k - 1\rangle$ and their indices have been generated.*

(2) The $k!$ permutation products of $\langle 1, \ldots, k\rangle$ are obtained by first appending the new element k to each $\langle 1, \ldots, k - 1\rangle$ permutation product, and then moving it successively left through every possible position in each of these products.

(3) Assign a group index g to all permutation products $\langle 1, \ldots, k\rangle$ in which k occupies the g-th position (where 1 is the rightmost position and k is the leftmost position of a product).

(4) If p is the index of a $\langle 1, \ldots, k - 1\rangle$ permutation product, then the index i of the $\langle 1, \ldots, k\rangle$ permutation product obtained from it is $i = (g.(k - 1)!) + p$.

For our treatment of underspecification, $perms\_scope_k$ needs to take a k-ary product of scope taking elements (in the order in which they appear in the surface syntax) and a k-ary relation representing the core proposition as its arguments. The scope taking elements and the core representation can be combined into a single product, e.g. as a pair consisting of the k-tuples of quantifiers as its first element and the core relation as its second. The permutation function $perms\_scope_k$ produces the $k!$-ary product of scoped readings. When a k-tuple of quantifiers is permuted, the λ-operators that bind the quantified argument positions in the core relation are effectively 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.

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. The ith scope reading is identified by projecting the ith element of the indexed product of propositions that is the output by our $perms\_scope_k$ function. Below we describe a function that performs this projection of a specified scope reading. It is important to recognise that in an implementation of PTCT it is not necessary to compute the full $k!$-ary product of permutations that provides (one of the) arguments for this function. Therefore, the PTCT term consisting of the application of $perms\_scope_k$ to an input pair of a k-tuple of GQs and a core relation provides an underspecified representation of the sentence corresponding to this term. We can

[4]See Campbell (2004) for a recent version of this procedure.

give a uniform type to these representations by defining arbitrary arity product types to cover the type of the k-tuple of GQs that is the first element in the pair to which $perms\_scope_k$ applies and the $k!$-tuple which is its value.

Consider the example "*Every man loves a woman*", with the GQs interpreting the subject and object NPs, the core relation, and PTCT term expressing the underspecified representation of the sentence given, respectively, as follows.

(17) $Q_1 = \lambda P \hat{\forall} x \epsilon B(\mathsf{man}'(x) \hat{\rightarrow} P(x))$

(18) $Q_2 = \lambda Q \hat{\exists} y \epsilon B(\mathsf{woman}'(y) \hat{\wedge} Q(y))$

(19) $\lambda uv.\mathsf{loves}' uv$

(20) $perms\_scope_2(\langle\langle Q_1, Q_2\rangle, \lambda uv.\mathsf{loves}' uv)\rangle$

The permutations of the quantifiers and the core representation that we produce are

(21) $\langle\langle\langle Q_1, Q_2\rangle, \lambda uv.\mathsf{loves}' uv\rangle\langle\langle Q_2, Q_1\rangle, \lambda vu.\mathsf{loves}' uv\rangle\rangle$

Applying relation reduction to computing the final propositions gives us a product containing the two readings.[5]

(22) $perms\_scope_2(\langle\langle Q_1, Q_2\rangle, \lambda uv.\mathsf{loves}' uv)\rangle) =$
$\langle \hat{\forall} x \epsilon B(\mathsf{man}'(x) \hat{\rightarrow} \hat{\exists} y \epsilon B(\mathsf{woman}'(y) \hat{\wedge} \mathsf{loves}'(x, y))),$
$\hat{\exists} y \epsilon B(\mathsf{woman}'(y) \hat{\wedge} \hat{\forall} x \epsilon B(\mathsf{man}'(x) \hat{\wedge} \mathsf{loves}'(x, y)))\rangle$

To obtain resolved scope readings from an underspecified representation, we define a function $project\_scope_k^i$ that computes the ith permutation of a k-ary product of propositions. Specifically, it returns the i-th proposition in the product of scope readings that $perms\_scope_k$ gives as its value. We extend the type system to include the type Num of natural numbers. We can then define $perms\_scope_k$'s type as

$$\langle \mathsf{Prop}_1, \ldots, \mathsf{Prop}_k\rangle \Rightarrow \mathsf{Num} \Rightarrow \mathsf{Prop}$$

where $\mathsf{Num} \leq k$. To ensure that the function is total, we can define $project\_scope_k^i$ so that it projects the (i mod k)th term, for example. Fox and Lappin (to appear, Chapter 6) provide a detailed proposal for the inclusion of natural numbers into PTCT.

3.4 An Alternative Perspective

Note that we can define an operation $permute_k^i$ that *directly* computes the ith permutation of a k-ary product. Given an integer i, its type will be

$$\Pi X_1 \ldots \Pi X_k(\langle X_1 \otimes \ldots X_k\rangle) \Rightarrow ((X_1 \Rightarrow \ldots X_k \Rightarrow \mathsf{Prop}) \Rightarrow \mathsf{Prop})$$

Using this approach, once we have supplied $permute_k^i$ with k quantifiers and a k-ary relation, and abstract i, the result can be taken to be an underspecified representation

[5]To simplify the exposition, the syntactic distinction between intensional term representations of propositions and extensional wffs is dropped in some of the following examples.

that denotes a (partial) function from integers to propositions. If we make the function complete, then we can define the type of these representations as

$$\mathsf{Num} \Rightarrow \mathsf{Prop}$$

As before, one way in which the function can be made complete is if $permute^i_k$ selects the (i mod k)th permutation.

3.5 Polymorphism in Core Relations and Generalized Quantifiers

Consider the following sentences.

(23) Every art dealer introduced a critic to two artists.

(24) Someone believes everything that Mary believes.

(25) The algorithm assigns an integer to each theorem in the logic.

In these examples "*introduced*" denotes a relation among three (basic) individuals, "*believes*" a relation between an individual and a proposition, and "*assigns*" a relation that holds among an abstract individual (an algorithm), a number, and a proposition. To avoid proliferating $perms\_scope_k$ functions to match each of the possible types of the core relation and its GQ arguments, it is necessary to represent these arguments as polymorphic, and to allow for a polymorphic relational type. By using product types rather than monomorphic lists as the basis for computing scope permutations we are able to do this. k-tuples can be specified for elements of distinct types.

The internal polymorphism of GQs is a somewhat more complex issue. We have been assuming that there is some way of encoding polymorphism in the type of the quantifiers representing noun phrases, so we have quantifiers with the types $(B \Longrightarrow \mathsf{Prop}) \Longrightarrow \mathsf{Prop}$ and $(\mathsf{Prop} \Longrightarrow \mathsf{Prop}) \Longrightarrow \mathsf{Prop}$, for example. Usually the type of the determiner function is fixed independently of the common noun property to which it applies. However, we could allow the type of the common noun property argument of a quantifier to determine the type of the quantifier. As it stands, this option is not directly expressible in PTCT, which adopts implicit polymorphism. In future work we will consider extending our type system to allow for explicit or parametric polymorphic typing in order to allow for this kind of type dependency. The type of a determiner would then be $K \Longrightarrow \mathsf{Prop}$, where K is understood as a meta-variable for some property type. The determiner's representation would require a type parameter, or constraint, that forces K to be instantiated by the same type $(A \Longrightarrow \mathsf{Prop})$ as the common noun property with which it combines.

3.6 Constraints on Scope Readings

There are various kinds of constraints that limit the set of possible scope readings for a particular sentence to a proper subset of the set of $k!$ orderings of the k scope taking elements which appear in it. A common condition on relative scope is the strong preference for wide scope assignment to certain quantifiers by virtue of their lexical semantic properties, like "*a certain N′*".

A second kind of condition depends upon the syntactic domain in which a GQ appears. So, for example, a quantified NP within a relative clause cannot take scope over a quantified NP in which the relative clause is embedded. The following two examples illustrate these constraints.

The strongly preferred reading of (26) is the one on which "*a certain book*" takes wide scope relative to "*every critic*".

(26) Every critic reviewed a certain book.

In (27) "*every assignment*" can only take narrow scope relative to "*a student who completed every assignment*".

(27) A student who completed every assignment came first in the class.

Scope constraints of these kinds can be formulated as filters on the $k!$-tuple of permutations $\langle\langle Qtuple_1, Rel_1\rangle, \ldots, \langle Qtuple_{k!}, Rel_{k!}\rangle\rangle$ that $perms\_scope_k$ generates for an argument pair $\langle Qtuple_1, Rel_1\rangle$. Each such filter is Boolean property function that imposes a condition on the elements of the $k!$-tuple.[6]

Let $\langle Quants, Rel\rangle$ be a variable ranging over pairs in which *Quant* is a k-tuple and *Rel* is a k-ary relation. We take *a_certain* to be a PTCT property that is true of all and only GQs that represent "*a certain N′*", and is false of anything else. As the k-tuples are indexed, there is a one-to-one correspondence between the elements of a k-tuple and their respective indices. Let $tuple\_element(i, Quants) = Q_i$ if Q_i is a member of *Quants* and the distinguished term ω otherwise. i and j are variables ranging over integers (they are of the type Num). We can specify the lexical scope constraint illustrated in (26) as the filter in (28).

(28) $$\lambda\langle Quants, Rel\rangle[\hat{\sim}(\hat{\exists} i\epsilon\mathsf{Num}\hat{\exists} j\epsilon\mathsf{Num}(a\_certain(tuple\_element(i, Quants))\ \hat{\wedge}\ \hat{\sim}\, a\_certain(tuple\_element(j, Quants))\ \hat{\wedge}\ j < i))]$$

This condition requires that no element of a $k!$-tuple of scope readings contains a k-tuple of GQs in which the index of a *a_certain* GQ is higher than that of a non-*a_certain* GQ (and so outscoped by it). Notice that we have only quantified over integers (elements of the type Num) in this filter. 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.

In order to formulate the condition illustrated in (27) we must introduce syntactic relations. 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. We can formulate the constraint as in (29).

(29) $$\lambda\langle Quants, Rel\rangle[\hat{\sim}(\hat{\exists} i\epsilon\mathsf{Num}\exists j\epsilon\mathsf{Num}(relcl\_embed(tuple\_element(i, Quants)\ tuple\_element(j, Quants))\ \hat{\wedge}\ j < i))]$$

This filter prevents a GQ that interprets an NP in a relative clause from having scope over a GQ that interprets an NP in which the relative clause is embedded.

[6]See van Eijck (2003) for examples of filters on lists specified as Boolean functions on the elements of a list.

4 Comparison with Other Theories

4.1 Storage

Cooper (1983) and Pereira (1990) define a procedure of quantifier storage. In the course of computing the compositional interpretation of a clause, the rules of semantic derivation apply storage to a quantified NP by substituting a variable for the GQ interpretation of the NP and adding the GQ to the storage set. A GQ is stored as the first element of a pair whose second element is the variable that is left in its argument position. The representation produced for the clause consists of the core propositional relation and a set of stored GQ pairs. When a GQ is discharged from storage, it is applied to the core relation, binding the variable in its original position. As the elements of the storage set are unordered, they can be discharged in any sequence, where each sequence yields a possible scope reading. So, for example, when storage is applied to the subject and object GQs in (30), the resulting representation (using a version of Pereira's (1990) notation) is (31).

(30) Most students read two articles.

(31) $\{\langle \mathsf{most\_students}', x\rangle, \langle \mathsf{two\_articles}', y\rangle\} \vdash \mathsf{read}'(x, y)$

The two possible orders for discharging the quantifiers from storage give the scope readings in (32) and (33).

(32) $\mathsf{most\_students}'(\lambda x.\mathsf{two\_articles}'(\lambda y.\mathsf{read}'(x, y)))$

(33) $\mathsf{two\_articles}'(\lambda y.\mathsf{most\_students}'(\lambda x.\mathsf{read}'(x, y)))$

Storage provides an elegant and straightforward way of generating underspecified scope representations for a sentence. However, there are (at least) three difficulties with this approach. First, storage is an additional mechanism defined outside of the semantic representation language as such. The expressions that it produces are not themselves part of this language (a typed λ-calculus) but stages in the derivation of well-formed terms of the representation language. While storage is easily implemented in a declarative fashion, as in Pereira (1990) and Blackburn and Bos (2003), it remains an essentially procedural device that is added to a compositional semantic theory as a means of obtaining scope ambiguity without attaching alternative scope readings to distinct syntactic structures, as in Montague (1974).

By contrast, in our account underspecified representations are themselves terms of PTCT, the representation language. Therefore, this issue does not arise.

It is worth pointing out that we are able to represent 'higher order' quantifiers like "*most*" in PTCT without compromising the first order nature of the theory. This involves constructing a recursive definition of cardinality for separation types and properties (Fox & Lappin, to appear, Chapter 6 and 9).

Second, as the storage set is unordered, it is not obvious how to project a particular scope reading without computing all $k!$ readings from the set. This problem is partially solved when storage is actually implemented, as the storage set becomes an ordered list. However, in order to allow for the projection of a specified scope reading without constructing all permutations of a list l, it is necessary to develop additional procedures for representing the $k!$ factorial list L of l's permutations, and projecting the i-th element of L without actually computing all the elements of L. While this is possible, these procedures have to be defined as additional mechanisms outside of the representation language (say a typed λ-calculus).

This problem does not arise on our approach. Underspecified representations consist in the application of a scope permutation function to an indexed k-tuple, which yields a term of PTCT. The $k!$-tuple that is the value of the function at an argument need not be fully computed to project a specified scope reading. A projection function can identify the i-th element of this product without computing other scope readings. Moreover, it is possible to construct underspecified representations using $perms\_scope_k$ without evaluating the arguments of this function when they first appear. In the course of building representations of the discourse, filters can be applied to the $perms\_scope_k$ function, progressively reducing the search space of possible scope readings . Product types, permutation functions on k-tuples, and projection functions on indexed k-tuples are all defined within the resources of PTCT.

Finally, because storage is a mechanism constructed outside of the representation language, it is necessary to specify an additional constraint language for stating the Boolean conditions required to restrict the set of possible scope readings derived from the storage set.[7]

Again, this problem does not arise on our treatment of underspecification. The filters that express constraints on scope readings are λ-terms of PTCT, and so the resources required for the formulation of these constraints are available within the representation language.

4.2 Holes, Plugs, and Glue Languages

Bos (1995) and Blackburn and Bos (2003) develop a constraint-based system for underspecified representation for first-order logic that they refer to as *Predicate Logic Unplugged* (PLU). This system is a generalization of the *hole semantics* approach to underspecification which Reyle (1993) first developed within the framework of Underspecified Discourse Representation Theory. Copestake et al.'s (1997) Minimal Recursion Semantics is an application of hole semantics within a typed feature structure grammar (HPSG).

An underspecified representation of a quantified first-order formula in PLU is an ordered tripe $\langle LH, F, R\rangle$. LH is a set of labels for formulas and of holes, which are (essentially) metavariables that take formulas as values. F is a set of labelled formulas, which may contain holes for subformulas. R is a set of scope constraints expressed as partial order relations on labels and holes. The PLU representation of (34) is (35).

(34) Every student wrote a program.

(35) $$\langle\{l_1, l_2, l_3, h_0, h_1, h_2\},\ \{l_1 : \forall x(\mathsf{student}'(x) \rightarrow h_1), l_2 : \exists y(\mathsf{program}'(y) \wedge h_2), l_3 : \mathsf{wrote}'(x, y)\},\ \{l_1 \leq h_0, l_2 \leq h_0, l_3 \leq h_1, l_3 \leq h_2\}\rangle$$

[7] Keller (1988) defines a type of storage that encodes relations of syntactic nesting within the stored GQ corresponding to an NP that contains another quantified NP. Although these nested stores avoid certain problems of variable binding encountered with Cooper storage, they do not, in themselves, impose constraints on possible scope readings of the sort that we have discussed in the previous section. See Blackburn and Bos (2003) for a discussion and an implementation of Keller stores.

The partial ordering constraints in (35) define a bounded lattice with h_0 as $\top$, the propositional core of the formula, l_3 as $\bot$, and l_1 and l_2 as midpoints of the lattice between $\top$ and $\bot$. As l_1 and l_2 are not ordered with respect to each other, either formula can be substituted for the hole in the other formula. l_3 must be substituted last in the remaining hole. If l_1 is taken as the value of h_0, l_2 is substituted for h_1, and then l_3 is substituted for h_2, the result is a wide scope reading of the universal quantifier, as in (36). Alternatively, if l_2 is taken as the value of h_0, l_1 is assigned to h_2, and l_3 to h_1, we obtain (37).

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

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

These are the only two scope resolutions that satisfy the partial order conditions in (35).

Hole semantics provides a more expressive and flexible system for constructing underspecified representations than storage. It generalizes naturally to scope elements other than GQs, like negation and modifiers. It is possible to identify a particular scope reading that satisfies the constraints of an underspecified hole semantic representation by imposing a particular order of substitution of labels for holes in a schematic formula set. However, it does suffer from the first and third difficulties which we raised against storage. Underspecified representations are constructed out of metavariables, schematic formulas, and partial ordering statements in a metalanguage that is distinct from the semantic representation language. The substitutions of labelled formulas for holes that generate the well-formed formulas of the representation language which correspond to scope readings are also metalinguistic operations added to the representation language.

More seriously Ebert (2003) shows that PLU and other hole semantic systems are expressively incomplete because their constraint languages do not permit the formulation of Boolean conditions on scope like those given in (28) and (29). As in the case of storage, it is possible to add a constraint language with sufficient expressive power required to state conditions of this kind.[8] But this requires further enrichment and complication of the theory. As we have seen, these problems do not arise on our account.

Dalrymple, Lamping, Pereira, and Saraswat (1999) and Crouch and van Genabith (1999) suggest a theory on which representations of GQs and core relations are expressed as premises in an underspecified semantic glue language. These premises are combined by the natural deduction rules of linear logic in order to yield a formula that represents the scope reading of sentence. The rules can apply to premises in different orders of derivation to generate alternative scope readings. Unlike PLU the glue language can be higher-order. The formal properties of glue language semantics are different than hole semantics, but it is closely related to this approach in the general view of underspecification that it adopts. It would seem that in order achieve expressive completeness in the sense of Ebert (2003), glue language semantics must add a system for stating constraints on the linear logic proof theory which it employs to derive specified interpretations.

[8]We are grateful to Ian Pratt-Hartman for helpful discussion of this point.

4.3 Relation Reduction

Van Eijck developed an approach to underspecified representations, in the functional programming language Haskell, which uses relation reduction and arbitrary arity relations (van Eijck, 2003). This inspired our approach, which we have developed in a more restrictive formal theory.

We give a fully general account and generalise van Eijck's approach in certain respects. In particular, we introduce a function for selecting a specific scope readings, and we make explicit the mechanisms for constraining scope readings using filters. Our approach to underspecification is also polymorphic, which leaves open the possibility of dealing with core relations whose arguments are of different types.

We developed PTCT to have a rich system of types, broadly comparable to that of Haskell, but within a language that we have shown to be of more restricted formal power.

5 Conclusion

We have presented a treatment of underspecified representation within PTCT which uses product types rather than lists to represent scope sequences. These types permit us to accommodate polymorphism in the core relation arguments.

We have characterized an underspecified representation as a PTCT term in which a function $perms\_scope_k$ applies to a pair containing an initial sequence of scope taking elements and a core relation. It returns as its value an indexed $k!$-product of possible scope readings. $project\_scope(perms\_scope_k(\langle Qs, R\rangle), i)$ projects the i-th scope reading in the $k!$-tuple of scope readings $perms\_scope_k(\langle Qs, R\rangle)$.

We have formulated constraints on scope readings as filters on the $k!$-tuples that $perms\_scope_k$ produces. These filters are PTCT property terms which encode Boolean conditions and quantification over the integers of indexed k-tuples.

Underspecified representations, the projection of a particular scope reading, and constraints on possible scope readings are all specified by appropriately typed λ-terms within the semantic representation language, PTCT, rather than through operations on schematic metalinguistic objects.

References

Blackburn, P., & Bos, J. (2003). *Representation and inference for natural language.* Stanford: CSLI.

Bos, J. (1995). Predicate logic unplugged. In *Proceedings of the tenth amsterdam colloquium.* Amsterdam, Holland.

Campbell, W. H. (2004). Indexing permutations. *Journal of Computing in Small Colleges*, *19*, 296–300.

Cooper, R. (1983). *Quantification and syntactic theory.* Dordrecht: D. Reidel.

Copestake, A., Flickinger, D., & Sag, I. A. (1997). *Minimal recursion semantics.* Stanford University, Stanford, CA: unpublished ms.

Crouch, D., & van Genabith, J. (1999). Context change, underspecifi cation, and structure of glue language derivations. In M.Dalrymple (Ed.), *Semantics and syntax in lexical functional grammar* (pp. 117–189). Cambridge, MA: MIT.

Curry, H. B., & Feys, R. (1958). *Combinatory logic* (Vol. 1). North Holland.

Dalrymple, M., Lamping, J., Pereira, F., & Saraswat, V. (1999). Quantifi ers, anaphora and intensionality. In M.Dalrymple (Ed.), *Semantics and syntax in lexical functional grammar* (pp. 39–89). Cambridge, MA: MIT.

Ebert, C. (2003). The expressive completeness of underspecifi ed representations. In *Proceedings of the fourteenth amsterdam colloquium.* Amsterdam, Holland.

Fox, C., & Lappin, S. (2001). A framework for the hyperintensional semantics of natural language with two implementations. In P. de Groote, G. Morrill, & C. Retore (Eds.), *Logical aspects of computational linguistics* (pp. 175–192). Berlin and New York: Springer-Verlag.

Fox, C., & Lappin, S. (2003a). Doing natural language semantics in an expressive fi rst-order logic with flexible typing. In G. Jaeger, P. Monachesi, G. Penn, & S. Wintner (Eds.), *Proceedings of formal grammar 2003* (pp. 89–102). Vienna: Technical University of Vienna.

Fox, C., & Lappin, S. (2003b). A type-theoretic approach to anaphora and ellipsis. In G. Angelova, K. Bontcheva, R. Mitkov, N. Nicolov, & N. Nikolov (Eds.), *Proceedings of recent advances in natural language processing* (pp. 1–10). Borovets, Bulgaria: Bulgarian Academy of Science.

Fox, C., & Lappin, S. (2004). An expressive fi rst-order logic with flexible typing for natural language semantics. *Logic Journal of the Interest Group in Pure and Applied Logics, 12*(2), 135–168.

Fox, C., & Lappin, S. (to appear). *Formal foundations of intensional semantics.* Oxford: Blackwell. (July 2005)

Fox, C., Lappin, S., & Pollard, C. (2002a). First-order curry-typed logic for natural language semantics. In S. Winter (Ed.), *Proceedings of the 7th international workshop on natural language understanding and logic programming* (pp. 175–192). Copenhagen: University of Copenhagen.

Fox, C., Lappin, S., & Pollard, C. (2002b). Intensional fi rst-order logic with types. In G. Alberti, K. Balough, & P. Dekker (Eds.), *Proceedings of the seventh symposium for logic and language* (pp. 47–56). Pecs, Hungary: University of Pecs.

Keenan, E. (1992). Beyond the fregean boundary. *Linguistics and Philosophy, 15,* 199–221.

Keller, W. (1988). Nested cooper storage: The proper treatment of quantifi cation in ordinary noun phrases. In U. Reyle & C. Rohrer (Eds.), *Natural language parsing and linguistic theories.* Dordrecht: Reidel.

Montague, R. (1974). *Formal philosophy: Selected papers of Richard Montague.* New Haven/London: Yale University Press. (Edited with an introduction by R.H. Thomason)

Pereira, F. (1990). Categorial semantics and scoping. *Computational Linguistics*, *16*, 1–10.

Reyle, U. (1993). Dealing with ambiguities by underspecifi cation: Construction, representation and deduction. *Journal of Semantics*, *10*, 123–179.

van Eijck, J. (2003). *Computational semantics and type theory.* CWI, Amsterdam: unpublished ms.