DEPARTMENT OF PHILOSOPHY,
LINGUISTICS AND THEORY OF SCIENCE

IMPLEMENTING PERCEPTUAL SEMANTICS IN TYPE THEORY WITH RECORDS (TTR)

Arild Matsson

Master's Thesis	30 credits
Programme	Master's Programme in Language Technology
Level	Advanced level
Semester and year	Spring, 2018
Supervisors	Simon Dobnik and Staffan Larsson
Examiner	Peter Ljunglöf
Keywords	type theory, image recognition, perceptual semantics, visual question answering, spatial relations, artificial intelligence

Abstract

Type Theory with Records (TTR) provides accounts of a wide range of semantic and linguistic phenomena in a single framework. This work proposes a TTR model of perception and language. Utilizing PyTTR, a Python implementation of TTR, the model is then implemented as an executable script. Over pure Python programming, TTR provides a transparent formal specification. The implementation is evaluated in a basic visual question answering (VQA) use case scenario. The results show that an implementation of a TTR model can account for multi-modal knowledge representation and work in a VQA setting.

Acknowledgements

Huge thanks to my supervisors and to professor Robin Cooper, all of whom have provided significant help through this process.

Contents

1 Introduction

Having computers understand visual input is desirable in several areas. A domestic assistant robot may use a camera to navigate and identify useful objects in a home. Driver-less cars need to be able to read road signs and track other moving vehicles. Web crawlers may extract information from images alongside text on the web.

This kind of understanding involves processing sensory (such as visual) input on a cognitive level. Low-level image processing may include tasks such as prominent color extraction, edge detection and visual pattern recognition. Higher-level processing, however, includes identifying objects, their properties and their relations to each other. This information can then be used for language understanding, reasoning, prediction and other cognitive processes. Making the connection between sensory input and cognitive categories is what concerns the field of *perceptual semantics* (Pustejovsky, 1990).

Humans use language to communicate information. Thus it is useful to add linguistic capacities to a perceptual system. With vision and language connected, a robot can talk about what it sees, and descriptions can be automatically generated for images found on the web. Image caption generation is indeed a popular task for evaluating computer vision systems. Another one is *visual question answering (VQA)* (Antol et al., 2017), where the system is expected to generate answers to natural-language questions in the context of a given image.

The connection between different modes of information, such as vision and language, requires a model of semantic representation. Formal models such as first-order logic (FOL) have long been of choice, but recent developments have seen data-driven approaches excel in some cases. Briefly put, the former kind tends to deliver deep structures of information in narrow domains, while the latter more easily covers wide domains, but with shallow information content (Dobnik & Kelleher, 2017). A recent contribution that combines several branches in formal systems is *Type Theory with Records (TTR)* (Cooper, 2005a, 2016). TTR is implemented in Python as *PyTTR* (Cooper, 2017).

1.1 Contribution of this thesis

The main purpose of this thesis is to extend the basic implementation of TTR (PyTTR, Cooper (2017)) to apply it for the first time in a practical task relating vision and language, in particular VQA.

The questions that this research raises are:

1. To what degree is *(a)* TTR as a theoretical framework and *(b)* its existing practical implementation suited to connect existing vision and language systems?
2. What are the benefits of using TTR this way for *(a)* vision and language systems and *(b)* visual question answering?
3. What can connecting vision and language systems tell us about semantics (and TTR)?

To explore these questions, a model will be formulated in TTR and implemented using PyTTR. The model will benefit from builing on past proposals; especially relevant are Dobnik & Cooper (2013) and Dobnik & Cooper (2017). As a limitation for the VQA task, the language domain is restricted to polar (yes/no) questions.

The theoretical background is summarized in Section 2. In Section 3, the strategies and techniques used for the implementation are described. The implementation is then presented in Section 4. In Section 5, the results are discussed in relation to the questions above. Finally, some conclusions are made in Section 6.

2 Background

This section will highlight some important pieces of the history of past research in relevant fields.

2.1 Image recognition and object detection

In image recognition, visual data is analyzed in order to detect and classify objects. A wide range of models have been developed to solve this task. Some focus only on the detection of objects (Blaschko & Lampert, 2008), some only on classification (e.g. ResNet, He et al., 2016), and others attempt to solve the full problem in an integrated fashion (Redmon et al., 2016; He et al., 2017).

Like in machine learning in general, and other kinds of data processing, significant values from the input data (the image) are collected in a process known as feature extraction. Various types of features exist for image processing. For one, Scale-invariant feature transform (SIFT) is a technique where significant locations of an image are used to extract *keypoints* (Lowe, 1999). Classification can then be performed by comparing the keypoints of a target image to those in a database.

With deep neural networks, however, the need for prior feature extraction is generally eliminated (He et al., 2016, 2017). Neural networks that process image data typically contain convolutional network layers. Color image data is highly dimensional, typically represented as a 2D matrix of pixels, where each pixel itself specifies a quantity of each of three basic colors. Convolutional layers are used to divide the image into smaller, overlapping segments, thus capturing locational aspects of the data. This way, the dimensionality can be reduced to a single, one-dimensional vector.

2.2 Computational semantics

Semantics is the study of meaning. Computational semantics is concerned with how to represent meaning digitally, and use it to perform semantic parsing, inference and other tasks (Blackburn & Bos, 2003).

A well-established and largely capable formalism for expressing and operating on propositions is first-order logic (FOL). In FOL, the phrase "all dogs run" can be expressed as $\forall x[\text{dog}(x) \rightarrow \text{run}(x)]$: for each individual it holds that if it is a dog then it runs. Blackburn & Bos (2003) claimed that FOL is an adequate semantic representation in a majority of cases, but "other approaches are both possible and interesting".

Montague (1974) connected FOL and lambda calculus with syntactic parsing to obtain semantic parsing. Thus, a natural-language utterance can be translated to a logical representation. One method for this is context-free grammar (CFG) with feature structures. In CFG, a set of rules determine how an utterance is parsed into a syntax tree: A determiner followed by a common noun form a noun phrase (NP $\rightarrow$ Det N), a sentence consists of a noun phrase and a verb phrase (S $\rightarrow$ NP VP), and so on. Extending a CFG framework with feature structures allows constituents to carry additional information, such as semantic representations, which can be combined as defined in the grammar rules.

For a simple example, a noun phrase may carry the term $t_{\text{NP}} = \lambda P \exists x P(x)$, and a verb phrase may carry $t_{\text{VP}} = \lambda z\ \text{sleep}(z)$. A sentence rule may combine them as $t_{\text{S}} \rightarrow t_{\text{VP}}(t_{\text{NP}})$. The result is $t_{\text{S}} = (\lambda P \exists x P(x))(\lambda z\ \text{sleep}(z))$ which, after β-reduction, is equal to $t_{\text{S}} = \exists x\ \text{sleep}(x)$. In real applications,

the rules for noun and verb phrases are usually more complex in order to cover more complex grammatical constructions.

With recent advancements in computer science, ambitious computational-semantic theories are now in abundance. As a competitor to formal systems, statistical methods have emerged which do well in various tasks within semantics. They leverage the performance of modern computers and the large amounts of data that are available as a product of our largely digitalized society. These data-driven approaches are easily adapted for wide coverage (assuming enough data is available) but they often produce shallow knowledge. Formal approaches, on the other hand, require more or less precisely crafted rules and formulations, which is time-consuming, but it typically enables the result to be more structured and comprehensive (Dobnik & Kelleher, 2017).

Searle (1980) disputes whether a computer really can *understand* concepts, that is, whether it will be able operate on grounded symbols or just the (arbitrary) symbols themselves. Harnad (1990) names this the *symbol grounding problem*. Steels (2008) describes experiments where a number of artificial agents participate in a language game, where they make up random words for preset concepts and manage to "agree" on which words to use for which concepts. With the success in these experiments, Steels argues that the symbol grounding problem is solved.

2.2.1 Type theory in linguistics

Type theory is a logic system developed by Whitehead & Russell (1910), Church (1940), Martin-Löf & Sambin (1984) among others (Coquand, 2015). The theory revolves around the concept that any object belongs to a type. The judgement that an object a belongs to a type T is written $a : T$. Functions are restricted to certain types, which allows more specificity in how they can be applied. For example, the factorial function $f_!$ may be declared over natural numbers by typing it as $f_! : \mathbb{N} \to \mathbb{N}$.

Ranta (2011) uses type theory to drive a method of syntactic parsing. At a glance, consider how the type-theoretical judgements "the" : *Det*, "door" : *N* and $f_{\text{DetN}} : \textit{Det} \to N \to \textit{NP}$ dictate that f_{DetN}("the", "door") is an object of the type *NP*.

Another example of type theory in natural language processing (NLP) is Kohlhase et al. (1996), which extends Discourse Representation Theory (DRT) with elements from type theory in order to provide compositionality.

2.3 Perceptual semantics

An artificial device perceiving its environment will make internal, symbolic representations of the real world outside (Pustejovsky, 1990). According to Frege (1948), these symbols will have *sense* as well as *reference*. A symbol with the sense "the dog" may have a certain dog in the environment as reference. Later, the same symbol and sense may refer to another dog.

By using terms of spatial relations ("left", "right", "above", etc.), the location of one object is specified in terms of the location and orientation of another. Different terminology have been used to refer to the two roles, but we will use *located object* and *reference object* (Dobnik et al., 2012).

Garnham (1989) explores terms of spatial relations and claims that there are three types of meanings for each term: basic, deictic and intrinsic. The basic meaning is relative to the speaker and holds for a single object only. The deictic and intrinsic meanings hold for relations between two objects. The deictic meaning is relative to the coordinate frame of the speaker, while the intrinsic is relative to that of the reference object. For someone standing near a car and facing its right-hand side, an object said to be "to the left of the car"

could be understood to be either near the car's backside (deictic meaning) or its left side (intrinsic meaning).

Logan & Sadler (1996) propose *spatial templates* for the classification of spatial relations. A spatial template is a field of acceptability ratings for a certain spatial relation term. The center of the field is the location occupied by the reference object and each rating denotes the acceptability of using the term if the located object is at the location of that rating. The ratings in spatial templates are collected through experiments.

Regier & Carlson (2001) instead propose a computational classification model known as *attentional vector-sum (AVS)*. The model considers distance between objects and the fact that they can have different shapes (especially elongated in some direction). This model is compared to three simpler alternatives in seven experiments, and AVS is found to perform best.

Coventry et al. (2001) explore extra-geometric constraints on the meaning of spatial relational terms, especially functional ones. The functional relation between objects is significant for the acceptability of terms of spatial relations. For example, an umbrella may well be said to be *above* a man, but less clearly *over* him, if it does not protect him from rain falling sideways in hard wind (Coventry et al., 2001).

2.4 Type Theory with Records (TTR)

Type Theory with Records (TTR) (Cooper, 2005b) combines several theories from logic, semantics and linguistics in a single type-theoretic framework. It employs *records*, objects which themselves are structured compositions of other objects; and accordingly, *record types* which are structures of other types. More details on the features of TTR are given in the overview in Section 2.4.1.

TTR has primarily been used to power various accounts of NLP, for example syntax (Cooper, 2005a,b, 2012, 2016), dialogue (Larsson & Cooper, 2009; Larsson, 2011; Cooper, 2016), situated agents (Dobnik et al., 2012; Dobnik & Cooper, 2013, 2017) and spoken language (Cooper, 2016).

Dobnik et al. (2012) present how TTR can be used to model a situated conversational agent. The agent moves around in a point-space world. Objects, detected as sub-point-spaces are recognized, as are (geometric and extra-geometric) spatial relations between them. The work is followed up by Dobnik & Cooper (2013) and Dobnik & Cooper (2017), which model similar situated agents.

2.4.1 Overview of TTR

This section is an overview of TTR based on Cooper (2012) and Cooper (2016).

In TTR there are two kinds of entities: types and objects. Each type is associated with a set of objects which are of that type, or in other words are **witnesses** of that type. The judgement that the object a is a witness of the type T is written $a : T$. For example, $34 :$ *Int* means that 34 is of the type *Int*.

A **record type** is a set of fields, each field carrying a label and a type. In the record type $\begin{bmatrix} \text{person} & : & \mathit{Ind} \\ \text{age} & : & \mathit{Int} \end{bmatrix}$ there is a field labeled 'person' of type *Ind*, and one labeled 'age' of the type *Int*.

The witnesses of record types are **records**. A record is also a set of fields with labels, but instead of a type, a label is associated with an object. A record r is a witness of a record type T, *if and only if* every field in the record type has a matching field in the record, that is, the labels are the same and the object in the record field is a witness of the type in the record type. The record $\begin{bmatrix} \text{person} & = & a_{12} \\ \text{age} & = & 28 \end{bmatrix}$ is a witness of the record type just mentioned, provided $a_{12} :$ *Ind* and $28 :$ *Int*.

A type T_{sub} is a **subtype** of another type T_{super}, written $T_{sub} \sqsubseteq T_{super}$, if any witness of the subtype is necessarily also a witness of the supertype. For example, *Int* $\sqsubseteq$ *Number*. In the case of record types, this means that a record type T_{sub} is a subtype of a record type T_{super} if and only if every field in T_{super} is present also in T_{sub} (allowing that a type in a T_{sub} field is itself a subtype of the type in the corresponding T_{super} field). For example, $\left[\ \mathrm{x}\ :\ \mathit{Int}\ \right] \sqsubseteq \left[\ \mathrm{x}\ :\ Number\ \right]$, provided *Int* $\sqsubseteq Number$.

Relationships between objects can be modeled using **ptypes**. A ptype is constructed from a predicate and a list of objects. For instance, the type hug(a, b) is constructed from the predicate 'hug' with the objects a in the first place and b in the second. The situation that a is hugging b is true if there exists a witness of hug(a, b). In a record type, a ptype typically uses the labels of other fields for its arguments: In $\left[\begin{array}{lcl} \mathrm{x} & : & \mathit{Ind} \\ \mathrm{c} & : & \mathrm{green(x)} \end{array}\right]$, the field 'c' is *dependent* on the field 'x'.

A type can be restricted to only have a single witness: If T is a type and $a : T$, then T_a is a **singleton type** having a as its only witness. There is an alternative notation for singleton types used in record types: The record type $\left[\ \mathrm{x}\ :\ \mathit{Ind}_a\ \right]$ can also be written $\left[\ \mathrm{x} = a\ :\ \mathit{Ind}\ \right]$, and is restricted to the record $\left[\ \mathrm{x}\ =\ a\ \right]$. A singleton-typed field notated this way is known as a **manifest field**. Any number of fields in a record type may be manifest fields.

A function from T_{dom} to T_{rng} is a witness of the **function type** $T_{dom} \to T_{rng}$. For instance, if $f = \lambda x :$ *Ind* . blue(x), then $f :$ *Ind* $\to$ *Type*.

A list of objects of type T is a witness of the **list type** $[T]$: If L is a list and $\forall a \in L, a : T$, then $L : [T]$. In this thesis, the list containing a, b and c will be written as $[a, b, c]$.

Types themselves may be the witnesses of other types. In order to allow this, types are sorted into **orders**, where types of one order may be witnesses of a type of a higher order. (This technique is known as **stratification**.) The type *Type*$^n, n > 0$ is the type of all types of order $n - 1$. Similarly, *RecType*$^n, n > 0$ is the type of all record types of order $n - 1$. Most of the types in this thesis will be of order 0, so we will skip the order superscript unless necessary, and use *Type* and *RecType* to denote *Type*^{1} and *RecType*^{1}, respectively.

The **relabeling** η of a record type T is a set of tuples where the first element is a label in T and the second is another, new label. T_η is another record type, similar to T but where the first item in each element of η has been replaced with the second item. So, if $T = \left[\ \mathrm{x}\ :\ T'\ \right]$ and $\eta = \{\langle \mathrm{x}, \mathrm{y} \rangle\}$, then $T_\eta = \left[\ \mathrm{y}\ :\ T'\ \right]$.

Flattening transforms a nested record type into an non-nested record type. In a nested record type such as $T = \left[\begin{array}{lcl} x & : & \left[\ y\ :\ T_1\ \right] \\ z & : & T_2 \end{array}\right]$, a **path** of labels from consecutive levels can be used to address a nested field, thus $x.y$ refers to the field with the type T_1. The flattened type $\varphi(T)$ contains every field in the first level, and the paths from the nested type are used as labels: $\varphi(T) = \left[\begin{array}{lcl} x.y & : & T_1 \\ z & : & T_2 \end{array}\right]$

The **meet** of two types $T_1 \wedge T_2$ is a new type whose witnesses are those that are witnesses both of T_1 and T_2 (an intersection of the sets of witnesses). The **join** $T_1 \vee T_2$ is a type whose witnesses are those that are witnesses either of T_1 or of T_2, or both (a union).

The **merge** of two types $T_1 \mathbin{\barwedge} T_2$ is a more complicated operation. If T_1 and T_2 are record types, their fields are added together to a new record type; if any label occurs in both types, so T_1 has $\langle \ell, T_1' \rangle$ and T_2 has $\langle \ell, T_2' \rangle$, a field with that label is added, which has the merge of the two field types, $\langle \ell, T_1' \mathbin{\barwedge} T_2' \rangle$. If any of T_1 and T_2 is not a record type, then $T_1 \mathbin{\barwedge} T_2 = T_1 \wedge T_2$.

2.5 Visual question answering (VQA)

Antol et al. (2017) suggest visual question answering (VQA) as a challenge for multi-modal semantic systems. A VQA system is presented an image and a natural-language question about the image, and is expected to produce a natural-language answer. The initiative includes datasets and a series of annual competitions since 2016.

A neural-network approach to question answering tasks in general is proposed by Andreas et al. (2016), where multiple neural-network modules are assembled like constituents in a syntax tree. For example, for the VQA question "What color is the bird?", a network that locates objects of a given class is connected to one which classifies the color at the indicated location. The method of composing the various modules is trained jointly with the module networks themselves.

3 Method

3.1 PyTTR: Programming with TTR

Cooper (2017) provides a Python implementation of TTR known as PyTTR. It supports the modeling of TTR types and operations such as judgement and type checking. As a Python library it also enables other features and peripheral procedures to be written in Python.

PyTTR allows, in turn, the implementation of TTR models. By implementing a theoretical model as a computer program, it can "come alive" and be tested on real problems and data. When implemented, the model can be evaluated and compared in practical settings to other models.

3.2 Object detection with YOLO

You only look once (YOLO) (Redmon et al., 2016) is a neural network model for image recognition. Given an image, it will detect objects and classify them. Each detection consists of a bounding box in pixel coordinates, a class label and a confidence score between 0 and 1.

YOLO is trained using a loss function which takes detection as well as classification into account. In other words, it simultaneously predicts bounding boxes and classifies the contained objects. Unlike He et al. (2017) and others, it does not contain any recurrent layers. The joint, recurrence-free model makes for a rather small network size and a high evaluation speed, although it does lag behind in accuracy compared to the state of the art (Redmon et al., 2016).

The model exists in a few different network configurations, which have all been trained on the COCO dataset (Lin et al., 2014). Development within this thesis has been using the "YOLOv2" configuration (Redmon, 2018).

YOLO is written in C, using the Darknet neural network library (Redmon, 2013). It can be used in Python with the TensorFlow machine learning library (Abadi et al., 2015) and the Darkflow library (Trieu, 2018) which translates a Darknet model to TensorFlow.

When invoked from Python, the return value is a collection of dictionary objects, each containing a label, coordinates and a confidence score, as exemplified in Listing 1. Results with confidence over a certain threshold are cast into TTR records. In this process, the bounding box coordinates are cast from a top-left and bottom-right tuple $\langle\langle x_1, y_1\rangle, \langle x_2, y_2\rangle\rangle$ to a center-width-height tuple $\langle x_c, y_c, w, h\rangle$ (later defined as the

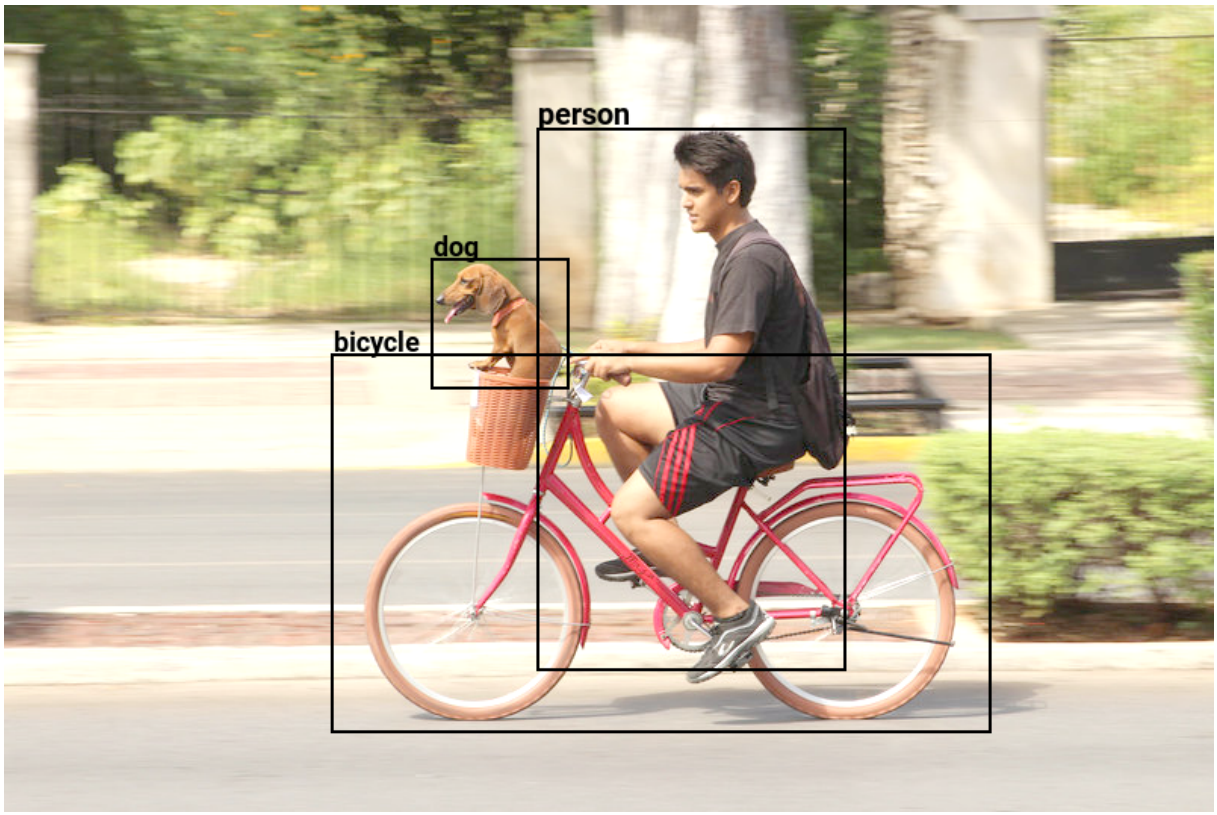

Figure 1: Visualization of the labels and bounding boxes emitted by YOLO when given an image depicting a cyclist with a dog.

Segment type), as the latter is more adequate for the spatial classification used in this project.

3.3 Objects and perception with TTR

The perception of objects in this model is largely based on Dobnik & Cooper (2017, Section 5.1). First, the object detection algorithm returns a set of *perceptual objects*. Each of these is evidence that a certain location is associated with a certain property (such as being a dog). Second, an *individuated object* is generated for each perceptual object. The individuated object additionally refers to a specific individual, and explicitly associates the property and the location with this individual. It is the type of situations where, for example, the individual a_1 is a dog at location l_1.

In Dobnik & Cooper (2017), the world has the form of a 3D point space rather than a 2D image. This necessitates different types for the perceptual input and the locations of perceived objects. In the point space case, the *PointMap* list type (a list of points) is used for the full "world". Any part of the world is also a list of points, thus also a *PointMap*. In our case, *Image* is used for the full image but we use *Segment* to refer to parts of it.

3.4 Spatial relations

Our method of spatial relation classification is inspired by Dobnik & Cooper (2013) but more simplistic. One simplification is that the reference frame is fixed. In the words of Garnham (1989) (as introduced in Section 2.3), this means we only consider the deictic meaning of spatial relation terms, and not the intrinsic. "Left" will mean to the left in the plane of the image, even if the reference object is turned on the side or toward the viewer. Another simplification is the neglection of the *functional* aspects of spatial relations

```
[
    {
        'topleft': {'x': 354, 'y': 86},
        'bottomright': {'x': 551, 'y': 437},
        'label': 'person',
        'confidence': 0.80116189,
    },
    {
        'topleft': {'x': 224, 'y': 234},
        'bottomright': {'x': 646, 'y': 476},
        'label': 'bicycle',
        'confidence': 0.85828924,
    },
    ...
]
```

Listing 1: Example output of YOLO invocation.

(Coventry et al., 2001).

In our model, a spatial classifier κ takes two locations and returns a boolean result. We have implemented spatial classifiers as Python functions. For the purpose of this thesis, no sophisticated spatial classification has been considered. Instead, a naive comparison between centers of bounding boxes was implemented using pre-defined rules. This was done for the four relations "left", "right", "above" and "below".

3.5 Language and VQA

In contrast to full VQA systems, the model presented in this thesis will be restricted to a limited type of questions, namely polar questions on the location of one object in relation to another. Such a question has a corresponding declarative statement: The question "Is there a lamp above a table?" corresponds to the statement "There is a lamp above a table".

Giving the natural-language utterance a representation in the same formal framework as the image allows comparing them to each other. The system will laborate with a *question type* (Q) representing the question, as well as a *scene type* (S) representing the perceived scene.

The situation described by the question type will be true if there exists a witness of that type, $r : Q$ (Barwise & Perry, 1981; Cooper, 2005a). The scene type, on the other hand, is considered true by virtue of being generated by perceptual classification. It follows that the question type is true if it is a supertype of the scene type. Thus, rather than looking for a witness to the question type, we formulate the problem as subtype checking, described in detail in Section 4.4. The question is answered with "yes" or "no" depending on whether the scene type is a subtype of the question type, $S \sqsubseteq Q$.

The existing research on TTR-based approaches to natural-language parsing, overviewed in Section 2.4, might be extensive enough to cover the kind of utterances considered here. However, there is currently no implementation available and ready to use, and parsing is not within the main focus of this thesis. Therefore, the natural-language parsing implemented for this thesis is instead a simplistic one, detailed in Section 4.3

4 Results

The results of this project consists primarily of a TTR model, which connects language to visual perception in a basic VQA setting and uses TTR throughout (Section 4.1). The project also solves a few significant sub-problems, which are not entirely within the TTR model but tightly connected to it. These have been solved as algorithms implemented in Python. They are: efficiently combining multiple belief record types into one (Section 4.2), basic translation from first-order logic to TTR (Section 4.3) and finally a subtype relation insensitive to labels, $\sqsubseteq_{\text{rlb}}$ (Section 4.4).

The code is written in a Jupyter notebook file and released at `https://github.com/arildm/imagettr` under the open-source MIT license. This section contains references to that code (specifically the version tagged 1.1) in the form of notebook cell numbers.

4.1 TTR model

The types in the TTR model are largely based on Dobnik & Cooper (2017), but the *Segment* type is new, the individuation function is improved and the *RelClf* mechanism is more concretely defined. The *Agent* type is also new.

In the code, the TTR model is constructed in notebook cells 9–10, 14–19 and 23–26.

Four basic types exist in the model.

Ind A reference to a single individual object, such as the reader or the Eiffel Tower.

Int An integer, such as 415.

Image A 2-dimensional digital image. It serves as an identifier to a set of extracted information, and its file type and actual data is not important in this thesis.

String A piece of plain text of arbitrary length.

A *Segment* is a record type describing a rectangular bounding box within an (implicit) image (Equation 1). Its fields contain the center coordinates of the box ('cx' and 'cy') and the width ('w') and height ('h') of the box. *Ppty* is the type of functions that can be applied to an individual and return a type (Equation 2). In our account the resulting type will be restricted to a ptype that is dependent on the individual, thus describing a property of it.

$$\mathit{Segment} = \begin{bmatrix} \text{cx} & : & \mathit{Int} \\ \text{cy} & : & \mathit{Int} \\ \text{w} & : & \mathit{Int} \\ \text{h} & : & \mathit{Int} \end{bmatrix} \tag{1}$$

$$\mathit{Ppty} = (\mathit{Ind} \rightarrow \mathit{Type}) \tag{2}$$

4.1.1 Object detection

A *perceptual object* is a record of the record type *Obj* (Equation 3). It contains a bounding box (the 'seg' field) and a property ('pfun'). An example record is given in Equation 4. An object detector is a function

from an image to a set of such perceptual objects, as captured by the *ObjDetector* function type (Equation 5). The YOLO object detector is typed as *ObjDetector* in notebook cell 14.

$$\mathit{Obj} = \begin{bmatrix} \text{seg} & : & \mathit{Segment} \\ \text{pfun} & : & \mathit{Ppty} \end{bmatrix} \tag{3}$$

$$\mathit{obj} = \begin{bmatrix} \text{seg} & = & \begin{bmatrix} \text{cx} & = & 435 \\ \text{w} & = & 422 \\ \text{cy} & = & 355 \\ \text{h} & = & 242 \end{bmatrix} \\ \text{pfun} & = & \lambda v : \mathit{Ind} \,.\, \text{bicycle}(v) \end{bmatrix} : \mathit{Obj} \tag{4}$$

$$\mathit{ObjDetector} = (\mathit{Image} \rightarrow [\mathit{Obj}]) \tag{5}$$

4.1.2 Individuation

The perceptual object couples a property with a location, but it does not explicitly say anything about any individual object. In Dobnik & Cooper (2017), the step from the perceptual to the *conceptual* domain is made by generating a record type that corresponds to a situation, namely the situation that a certain individual has a certain property and is at a certain location. This situation record type is known as an *individuated object*, and is a subtype of *IndObj* (Equation 6).

$$\mathit{IndObj} = \begin{bmatrix} \text{x} & : & \mathit{Ind} \\ \text{cp} & : & \mathit{PType} \\ \text{cl} & : & \text{location}(\text{x}, \text{loc}) \\ \text{loc} & : & \mathit{Segment} \end{bmatrix} \tag{6}$$

Here, 'x' is an individual and 'loc' is a bounding box. The 'cl' field specifies that 'loc' is the location of 'x', and the purpose of 'cp' is to declare a property of 'x'. As all individuated objects are subtypes of *IndObj*, the 'cp' field must have a type that will be a supertype of any ptype; we define *PType* to be this.

Definition 1 *For any ptype* $T = \mathit{pred}(v_1, ..., v_n)$, $T \sqsubseteq \mathit{PType}$.

A function for generating an *IndObj* subtype from an *Obj* record is known from Dobnik & Cooper (2017) as an *individuation function*. It is typed as *IndFun* (Equation 7). Note that it generates a record type, in contrast to *ObjDetector* which generates records.

$$\mathit{IndFun} = (\mathit{Obj} \rightarrow \mathit{RecType}) \tag{7}$$

The individuation function is defined as f_{IndFun} in Equation 8 (notebook cell 15). The record type resulting from applying f_{IndFun} is a subtype of *IndObj*, where the 'x' and 'loc' fields are specified using manifest fields. The 'x' field is specified as a newly instantiated *Ind* object, a_n (where n is a number such that the new instatiation is unique). The 'loc' field is specified as the value of 'seg' in the *Obj* record. Having these fields specified allows us to access the values (the individual and the location) at a later stage, when we are looking at the *IndObj* record types and not the *Obj* records. For the types in the 'cl' and 'cp' fields, it is not important

to know *what* the proof is, as long as there is a proof. Therefore, they do not need to be instantiated and specified.

The instantiation of new individual objects a_n assumes that no two *Obj* records describe the same individual. If more than one object detection model were applied, perhaps in an attempt at wider coverage, then we might end up with generating multiple individual objects where a human observer would detect only one. A merging step could then be added in connection with the individuation procedure, where objects of the same property and similar locations are merged as one. Furthemore, the detection models may return different but similar labels, such as "car" and "truck". Covering these cases would additionally require measuring the similarity of different semantic concepts.

An example application of f_{IndFun} is shown in Equation 9. The output record type describes a situation where an individual identified as a_0 is classified as a bicycle and is occupying a 422×242-sized rectangle with its center at $(435, 355)$.

$$f_{\mathit{IndFun}} = \lambda r : \mathit{Obj} \,.\, \begin{bmatrix} \mathrm{x} = a_n & : & \mathit{Ind} \\ \mathrm{cp} & : & r.\mathrm{pfun}(\mathrm{x}) \\ \mathrm{cl} & : & \mathrm{location}(\mathrm{x}, \mathrm{loc}) \\ \mathrm{loc} = r.\mathrm{seg} & : & \mathit{Segment} \end{bmatrix} \tag{8}$$

$$f_{\mathit{IndFun}}\left(\begin{bmatrix} \mathrm{seg} = \begin{bmatrix} \mathrm{cx} = 435 \\ \mathrm{w} = 422 \\ \mathrm{cy} = 355 \\ \mathrm{h} = 242 \end{bmatrix} \\ \mathrm{pfun} = \lambda v : \mathit{Ind} \,.\, \mathrm{bicycle}(v) \end{bmatrix}\right) = \begin{bmatrix} \mathrm{x} = a_0 & : & \mathit{Ind} \\ \mathrm{cp} & : & \mathrm{bicycle}(\mathrm{x}) \\ \mathrm{cl} & : & \mathrm{location}(\mathrm{x}, \mathrm{loc}) \\ \mathrm{loc} = \begin{bmatrix} \mathrm{cx} = 435 \\ \mathrm{w} = 422 \\ \mathrm{cy} = 355 \\ \mathrm{h} = 242 \end{bmatrix} & : & \mathit{Segment} \end{bmatrix} \tag{9}$$

4.1.3 Spatial relation classification

Relations may hold between pairs of individuated objects. How do we detect and model a certain relation between such a pair?

Since we are interested in the spatial relation between a *reference object* and a *located object*, we will be constructing tuple-like records of the type *LocTup* defined in Equation 10. Records of this type contain instantiations (records) of two *IndObj* record types. In Equation 11, a classifier is modeled as a function from such a record to a new record type which should describe the relation.

$$\mathit{LocTup} = \begin{bmatrix} \mathrm{lo} & : & \mathit{IndObj} \\ \mathrm{refo} & : & \mathit{IndObj} \end{bmatrix} \tag{10}$$

$$\mathit{RelClf} = (\mathit{LocTup} \rightarrow \mathit{RecType}) \tag{11}$$

For instance, a classifier for "left" might look like in Equation 12, where κ_{left} is a non-TTR, boolean function. Of course, the requirement that the individual $r.\mathrm{lo.x}$ is actually located at $r.\mathrm{lo.loc}$ (and same for $r.\mathrm{refo}$) is

implicit from the typing as *IndObj*, where a field typed as location(x, loc) is necessarily present.

$$\lambda r : \mathit{LocTup} \,.\, \begin{cases} \begin{bmatrix} \text{x} & : & r.\text{lo.x} \\ \text{y} & : & r.\text{refo.x} \\ \text{cr} & : & \text{left}(\text{x}, \text{y}) \end{bmatrix}, & \text{if } \kappa_{\text{left}}(r.\text{lo.loc}, r.\text{refo.loc}) \\ [\,], & \text{otherwise} \end{cases} \tag{12}$$

This is implemented in notebook cell 16, where the function `relclf` creates functions like the one in Equation 12. The function `get_relclfs` creates such a function for each of the four predicate-classifier pairs, and `find_all_rels` applies each classifier to each *IndObj* pair. (The latter step is later re-implemented in the agent algorithm.)

4.1.4 Beliefs

The set of individuated objects, added to the set of relation classification results, forms a set of beliefs. Each of these types is a situation held to be true, by virtue of resulting from perception mechanisms. They can be *combined* into one *scene* record type which describes the full scene. The method of such combination is not trivial, and is discussed in Section 4.2.

4.1.5 Language

In order to add the connection to language, any natural-language utterance must be parsed into TTR. As discussed in Section 3.5, TTR-based natural-language parsing has not yet been implemented as a ready-to-use library. Therefore, parsing must be done externally to the TTR model. This is described in Section 4.3. Equation 13 models the question "Is there a lamp above a table?" (equivalent to the statement "There is a lamp above a table").

$$\begin{bmatrix} \text{x} & : & \mathit{Ind} \\ \text{y} & : & \mathit{Ind} \\ \text{c}_\text{x} & : & \text{lamp}(\text{x}) \\ \text{c}_\text{y} & : & \text{table}(\text{y}) \\ \text{c}_\text{r} & : & \text{above}(\text{x}, \text{y}) \end{bmatrix} \tag{13}$$

As mentioned in Section 3.5, answering the question corresponds to checking whether $S \sqsubseteq Q$. An important problem, however, stems from the fact that TTR record types are labeled. In general, fields in the scene type and question type will not share labels in a way that enables simple subtype checking to be useful. The remedy to this is an alternative subtype relation $\sqsubseteq_{\text{rlb}}$ which is insensitive to label names. This new relation is discussed in Section 4.4.

4.1.6 Agent

The perceptual-conceptual pieces described above are now connected in an *agent* record type (Equation 14 and Equation 15) with associated manipulation algorithms. Upon receiving an image, it will carry out object detection, individuation and spatial relation classification, in order to form its beliefs. It may also receive a parsed natural-language utterance, which will then be verified against the beliefs. A construction like this

provides a means to answer to natural-language questions about the image.

$$\mathit{Agent} = \begin{bmatrix} \text{objdetector} & : & \mathit{ObjDetector} \\ \text{indfun} & : & \mathit{IndFun} \\ \text{relclfs} & : & [\mathit{RelClf}] \\ \text{state} & : & \mathit{AgentState} \end{bmatrix} \tag{14}$$

$$\mathit{AgentState} = \begin{bmatrix} \text{img} & : & \mathit{Image} \\ \text{perc} & : & [\mathit{Obj}] \\ \text{bel} & : & [\mathit{RecType}] \\ \text{utt} & : & \mathit{String} \\ \text{que} & : & \mathit{RecType} \end{bmatrix} \tag{15}$$

The fields 'objdetector', 'indfun' and 'relclfs' of *Agent* are to be statically defined for a specific agent. While running, the agent will modify the *AgentState* record in 'state'. The 'perc' field will contain a list of perceptual objects. The 'bel' field will be a list of beliefs modelled as record types: individuated objects and spatial relations between individuals.

For an agent record ag : *Agent*, the perception and question-answering procedure is carried out as follows.

Visual perception

1. Visual input in the form of an image is received and assigned to ag.state.img.
2. ag.objdetector is invoked on ag.state.img and creates a collection of perceptual object records that are assigned to ag.state.perc.
3. ag.indfun is, in turn, invoked on each record in ag.state.perc and resulting individuated object record types are added to ag.state.bel.
4. Now, each function in ag.relclfs is applied to each pair of record types in ag.state.bel:
   (a) For each pair T_1 and T_2 in ag.state.bel, a *LocTup* record type is constructed as $\begin{bmatrix} \text{lo} & : & T_1 \\ \text{refo} & : & T_2 \end{bmatrix}$. Note that this will be specified to certain individuals and segments, and is thus more informative than the plain *LocTup* type.
   (b) The specified *LocTup* type is instantiated to a record.
   (c) Each function in ag.relclfs is applied to the created record, and the record type resulting from each application is added to ag.state.bel unless it is empty ([ ]).

   For example, the "left" classifier in Equation 12 is applied to each pair of *IndObj* after combining them into a *LocTup* and instantiating it. Note that the *IndObj* have manifest fields which carry on to the *LocTup* type, so it is more specific than just instantiating *LocTup* itself.

The individuated objects and the spatial relations are contained in the same list, ag.state.bel, which models beliefs of the agent. (Remember that an individuated object is a belief that a certain individual has a certain property and location.) In extension, this list may contain record types of many other shapes, perhaps describing situations where an individual has a certain color or two individuals are involved in an event (as suggested in Section 6.4). Step 4 works here because ag.state.bel is sure to contain only *IndObj* record types

at this point, and because ag.relclfs only contains *RelClf* functions. The general case would necessitate a different formulation (and a new name for the 'relclfs' field), perhaps utilising subtype checking to qualify possible argument combinations.

Language

1. Any language input utterance is assigned to ag.state.utt.
2. The utterance is parsed and the resulting record type (Q) assigned to ag.state.que.
3. The record types in ag.state.bel are combined to one (S). If the resulting record type is a relabel-subtype of ag.state.que, $S \sqsubseteq_{\text{rlb}} Q$ the answer "Yes" is emitted; otherwise "No".

The *Agent* type is implemented in notebook cell 23 and instatiated as a record in cell 24. Its algorithms are implemented in cell 25 as `agent_see` for the *visual perception* part and `agent_hear` for *language*.

The state of an agent is a record of the type *Agent*. An example state ag is shown in Equation 16.

$$
ag = \left[\begin{array}{rcl}
\text{objdetector} & = & \texttt{yolo\_detector} \\
\text{indfun} & = & \texttt{indfun} \\
\text{relclfs} & = & [Clf_{\text{left}}, Clf_{\text{right}}, Clf_{\text{above}}, Clf_{\text{below}}] \\
\text{state} & = & \left[\begin{array}{rcl}
\text{img} & = & \texttt{<Image "dogride.jpg">} \\
\text{perc} & = & [\left[\begin{array}{rcl} \text{seg} & = & \left[\begin{array}{rcl} \text{cx} & = & 452 \\ \text{w} & = & 197 \\ \text{cy} & = & 261 \\ \text{h} & = & 351 \end{array}\right] \\ \text{pfun} & = & \lambda a : \mathit{Ind} \,.\, \text{person}(a) \end{array}\right], \left[\begin{array}{rcl} \text{seg} & = & \left[\begin{array}{rcl} \text{cx} & = & 435 \\ \text{w} & = & 422 \\ \text{cy} & = & 355 \\ \text{h} & = & 242 \end{array}\right] \\ \text{pfun} & = & \lambda a : \mathit{Ind} \,.\, \text{bicycle}(a) \end{array}\right], ...] \\
\text{bel} & = & [\left[\begin{array}{rcl} \text{x} = a_0 & : & \mathit{Ind} \\ \text{cp} & : & \text{person}(x) \\ \text{cl} & : & \text{location}(x, loc) \\ \text{loc} = \left[\begin{array}{rcl} \text{cx} & = & 452 \\ \text{w} & = & 197 \\ \text{cy} & = & 261 \\ \text{h} & = & 351 \end{array}\right] & : & \mathit{Segment} \end{array}\right], \left[\begin{array}{rcl} \text{x} = a_0 & : & \mathit{Ind} \\ \text{y} = a_1 & : & \mathit{Ind} \\ \text{cr} & : & \text{above(x, y)} \end{array}\right], ...] \\
\text{utt} & = & \text{“Is there a dog to the left of a bicycle?”} \\
\text{que} & = & \left[\begin{array}{rcl} \text{x} & : & \mathit{Ind} \\ \text{y} & : & \mathit{Ind} \\ \text{c}_0 & : & \text{dog}(x) \\ \text{c}_1 & : & \text{bicycle}(y) \\ \text{c}_2 & : & \text{left}(x, y) \end{array}\right]
\end{array}\right]
\end{array}\right] \tag{16}
$$

4.2 Combining situation types

The beliefs of the agent are formed by a collection of record types. These are *combined* into one, in order to build the scene type S.

Consider the spatial relation classifiers, which create record types with the fields 'x', 'y', and 'cr'. One of these record types may be specified so that it declares that an individual a_1 is above another individual a_2 (Equation 17). Another record type may declare that a_2 is to the right of a_1 (Equation 18). The *combination* of these beliefs, which declares both these relations, is described by Equation 19. To avoid conflict, some fields have been relabeled, prompting the expectation that combination results have no informative or predictable labels.

$$T_1 = \begin{bmatrix} \text{x} = a_1 & : & \mathit{Ind} \\ \text{y} = a_2 & : & \mathit{Ind} \\ \text{cr} & : & \text{above}(\text{x}, \text{y}) \end{bmatrix} \tag{17}$$

$$T_2 = \begin{bmatrix} \text{x} = a_2 & : & \mathit{Ind} \\ \text{y} = a_1 & : & \mathit{Ind} \\ \text{cr} & : & \text{right}(\text{x}, \text{y}) \end{bmatrix} \tag{18}$$

$$\begin{bmatrix} \text{x} = a_1 & : & \mathit{Ind} \\ \text{y} = a_2 & : & \mathit{Ind} \\ \text{cr}_1 & : & \text{above}(\text{x}, \text{y}) \\ \text{cr}_2 & : & \text{right}(\text{y}, \text{x}) \end{bmatrix} \tag{19}$$

TTR features an *merge* operation ($\wedge$), but a merge will not have the result desired here. Since the same labels occur in both belief types, a merge would result in *meet types*, as seen in Equation 20, in a way which is not useful for this purpose. The meet type of two different singleton types, $\mathit{Ind}_{a_1} \wedge \mathit{Ind}_{a_2}$, can only be true if the two individuals are the same, $a_1 = a_2$. The constraint in the 'cr' field then says that some individual is above and to the right of itself, which is meaningless and certainly not what we are trying to obtain.

$$T_1 \wedge T_2 = \begin{bmatrix} \text{x} & : & \mathit{Ind}_{a_1} \wedge \mathit{Ind}_{a_2} \\ \text{y} & : & \mathit{Ind}_{a_2} \wedge \mathit{Ind}_{a_1} \\ \text{cr} & : & \text{above}(\text{x}, \text{y}) \wedge \text{right}(\text{x}, \text{y}) \end{bmatrix} \tag{20}$$

Cooper (2016) solves this by a method of nesting and flattening (notebook cell 17). Each belief is added as the type of a new field 'prev' in the next belief: $[\text{prev} : T_1] \wedge T_2$ (Equation 21). The result is then flattened to avoid the nesting (Equation 22). The field labeled 'x' in T_1 is now labeled 'prev.x' and does not conflict with the field labeled 'x' from T_2.

$$\begin{bmatrix} \text{prev} & : & \begin{bmatrix} \text{x} = a_1 & : & \mathit{Ind} \\ \text{y} = a_2 & : & \mathit{Ind} \\ \text{cr} & : & \text{above}(\text{x}, \text{y}) \end{bmatrix} \\ \text{x} = a_2 & : & \mathit{Ind} \\ \text{y} = a_1 & : & \mathit{Ind} \\ \text{cr} & : & \text{right}(\text{x}, \text{y}) \end{bmatrix} \tag{21}$$

$$\begin{bmatrix} \text{prev.x} = a_1 & : & \textit{Ind} \\ \text{prev.y} = a_2 & : & \textit{Ind} \\ \text{prev.cr} & : & \text{above}(\text{prev.x}, \text{prev.y}) \\ \text{x} = a_2 & : & \textit{Ind} \\ \text{y} = a_1 & : & \textit{Ind} \\ \text{cr} & : & \text{right}(\text{x}, \text{y}) \end{bmatrix} \tag{22}$$

Another method is used in this project for the purpose of computational speed (notebook cell 18). In this method, each belief record type is relabeled to only have unique labels, and then merged. An example result is shown in Equation 23. Generating unique labels is an operation outside TTR, making this method less purely TTR-powered.

$$\begin{bmatrix} \text{x}_1 = a_1 & : & \textit{Ind} \\ \text{y}_1 = a_2 & : & \textit{Ind} \\ \text{cr}_1 & : & \text{above}(\text{x}_1, \text{y}_1) \\ \text{x}_2 = a_2 & : & \textit{Ind} \\ \text{y}_2 = a_1 & : & \textit{Ind} \\ \text{cr}_2 & : & \text{right}(\text{x}_2, \text{y}_2) \end{bmatrix} \tag{23}$$

Both methods result in duplicate fields: there is no meaningful difference between the fields labeled 'x_1' and 'y_2' above, as both have the same singleton type. Removing these duplicates (also *deduplication*, or *dedupe*) is necessary for the subtype check that will follow. This process first involves finding which fields have the same type as another field. Subsequently, simply removing duplicates is not an option, as there may be other fields that depend on the duplicate field. These dependent fields must also be updated to use the remaining field.

The `combine` and `dedupe` functions are defined in Listing 2 (notebook cells 7 and 18).

In `combine`, a list of record types are reduced to one type by unique relabeling and merging. The `unique_labels` function makes use of `gensym` in the PyTTR library, which generates new, consecutively numbered labels such as `loc_4` (typeset here as loc_4). A label containing an underscore ('_') is assumed to already be uniquely numbered. Thus, labels without undescores are relabeled to new, numbered labels.

Finally, the `dedupe` function removes any duplicated singleton and complex field types by relabeling all occurrences of each such field type with one of their labels. For example, in $\begin{bmatrix} \text{x} = a_1 & : & \textit{Ind} \\ \text{y} = a_1 & : & \textit{Ind} \end{bmatrix}$, 'y' is relabeled to 'x'. (Or possibly vice versa; the order of record type fields is unspecified.) When all duplicates of some field type have been removed, recursion is used to avoid usage of old labels in the outer for-loop.

4.3 Parsing language to TTR

As mentioned in Section 2.4, pure TTR solutions to natural language parsing have been developed but not implemented. Such an implementation is sufficiently complex and outside the scope of this thesis. Furthermore, the language domain is limited for the purpose of this thesis, and a wide-coverage parsing solution is more than what is necessary. Therefore, instead of implementing natural language parsing in TTR, this project uses another standard method to parse natural language to first-order logic (FOL). The FOL expression is subsequently *translated* to TTR in a new algorithm.

```
from functools import reduce

def unique_labels(T):
    """Relabel a RecType so each field label is unique over all RecTypes."""
    rlb = ((l, gensym(l)) for l in T.labels() if '_' not in l)
    return rectype_relabels(T, dict(rlb))

def dedupe(T):
    """Make a copy of a record type without duplicated field values."""
    for l,v in T.fields():
        # Are there more fields with the same value?
        l2s = [l2 for l2,v2 in T.fields() if l2 != l
               # Dedupe singleton types and complex types.
               and (isinstance(v, SingletonType) or not is_basic_type(v))
               # Using show is error-prone, pyttr should have an equals method.
               and show(v) == show(v2)]
        if len(l2s):
            # Relabel all these fields to the same label,
            # overwriting until one remains.
            for l2 in l2s:
                T.Relabel(l2, l)
            # Dependent fields have changed, so start over.
            return dedupe(T)
    # No more duplicates.
    return T

def combine(Ts):
    """Combine a list of belief record types into one."""
    f = lambda a, b: a.merge(unique_labels(b))
    return dedupe(reduce(f, Ts, RecType()))
```

Listing 2: The `combine` and `dedupe` functions.

```
QS[SEM=<?np(?pp)>] -> 'is' 'there' NP[SEM=?np] PP[SEM=?pp]
QS[SEM=<?np(\P.true)>] -> 'is' 'there' NP[SEM=?np]

NP[SEM=<?det(?n)>] -> Det[SEM=?det] N[SEM=?n]
Det[SEM=<\P Q.exists x.(P(x) & Q(x))>] -> 'a' | 'an'

VP[SEM=?pp] -> 'is' PP[SEM=?pp]
PP[SEM=<\x.(?np(\y.?prep(x, y)))>] -> Prep[SEM=?prep] NP[SEM=?np]

Prep[SEM=<left>] -> 'to' 'the' 'left' 'of'

N[SEM=<dog>] -> 'dog'
N[SEM=<person>] -> 'person'
```

Listing 3: A snippet of the FCFG grammar.

4.3.1 Parsing to FOL

Parsing is done using the NLTK library (Bird et al., 2009), which contains a semantically augmented CFG framework just like the method introduced in Section 2.2. It uses feature structures to associate each constituent with a FOL term possibly using lambda calculus. A snippet of the grammar is given in Listing 3 (the full grammar is given in notebook cell 20). The grammar can be used to generate the sample parse tree in Figure 2. Above the word-tokenized sentence, every node in the tree represents a constituent. Underneath the abbreviation, each constituent features first the expression given in the grammar specification (with some typographical modification), and second, the FOL term resulting from substitution and β-reduction.

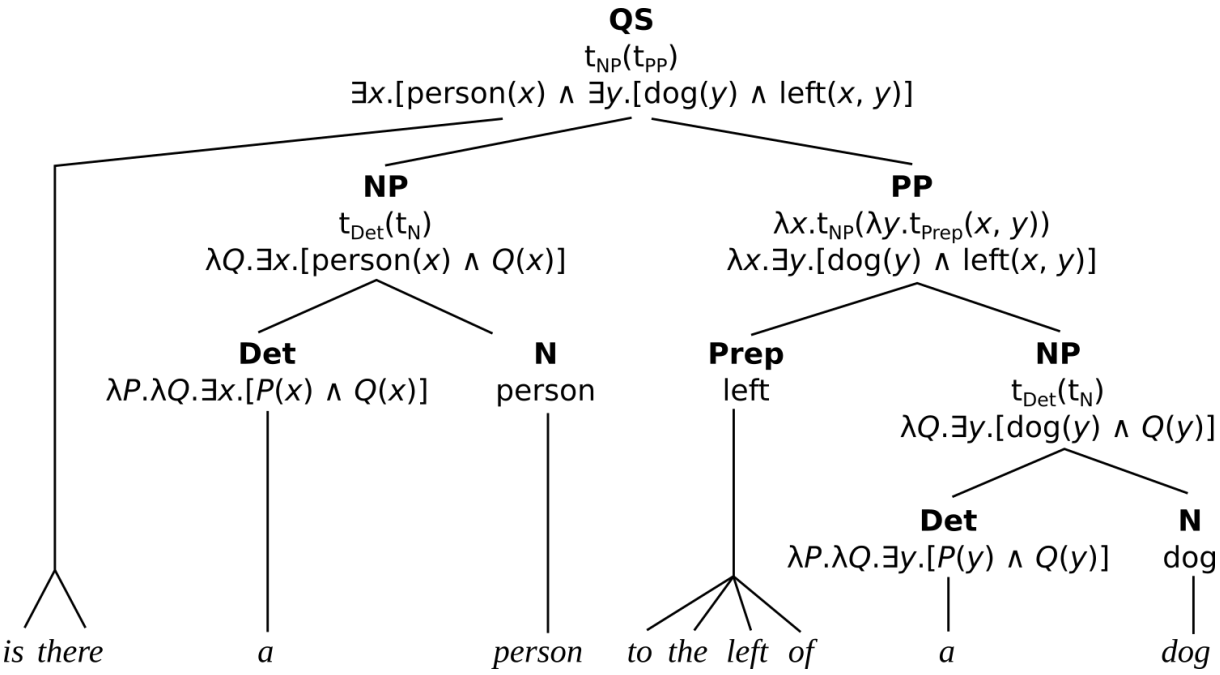

Figure 2: Example syntactic-semantic parsing of an utterance into first-order logic.

4.3.2 Translating FOL to TTR

The result of the CFG parsing is a Python object that encodes the FOL expression. A custom Python function `fol_to_pyttr`, given in Listing 4 (notebook cell 20), traverses this object recursively and builds a PyTTR type.

```
import nltk
from nltk.sem.logic import ApplicationExpression, AndExpression, ExistsExpression,
    ConstantExpression

def fol_to_pyttr(expr, T=RecType()):
    """Turns a FOL object into a RecType."""
    # Existential quantifier -> Ind field.
    if isinstance(expr, ExistsExpression):
        T.addfield(str(expr.variable), Ind)
        return fol_to_pyttr(expr.term, T)

    # Application -> ptype, e.g. left(x, y)
    if isinstance(expr, ApplicationExpression):
        pred, args = expr.uncurry()
        # Create a PType function, e.g. lambda x:Ind . dog(x)
        fun = create_fun(str(pred), 'abcd'[:len(args)])
        T.addfield(gensym('c'), (fun, [str(a) for a in args]))
        return T

    # For and-expressions, interpret each term.
    if isinstance(expr, AndExpression):
        fol_to_pyttr(expr.first, T)
        fol_to_pyttr(expr.second, T)
        return T

    # A constant function in the "is there an X" rule trivially gives "true".
    if isinstance(expr, ConstantExpression) and str(expr.variable) == 'true':
        return T

    raise ValueError('Unknown expression: ' + str(type(expr)) + ' ' + str(expr))

def eng_to_pyttr(text):
    # Tokenize.
    sent = text.lower().strip('.?!').split()
    # NLTK-parse to syntax tree.
    trees = parser.parse(sent)
    # Extract semantic representation for the tree.
    sem = nltk.sem.root_semrep(list(trees)[0])
    # Interpret to TTR record type.
    T = fol_to_pyttr(sem, RecType())
    return T
```

Listing 4: Translation from FOL to TTR.

- For an “Exists” expression, an *Ind* field is added to the type.
- For an “Application” expression, a ptype field is added, copying the predicate and variable names.
- An “And” expression simply triggers recursion into each of the two terms.
- The constant ‘true’ is added to allow simple existential questions like “Is there an aeroplane?”

A wrapper function `eng_to_pyttr` combines simple word tokenization, CFG parsing and FOL-to-TTR translation.

4.4 The relabel-subtype relation

Perceptual mechanisms and the combination of belief types have produced a scene type S. Separately, natural language parsing of a speaker utterance has provided a question type Q. Now, in order to answer

the question, we are interested in whether $S \sqsubseteq Q$.

However, the fact that TTR record types are labeled prevents direct usage of the subtype relation. Field labels in the scene type will generally not agree with those in the question type. This prompts for a more advanced variant of subtype checking, allowing *relabeling*.

Definition 2 *A record type S is a **relabel-subtype** of the record type Q, $S \sqsubseteq_{rlb} Q$, if there is a relabeling η of Q such that $S \sqsubseteq Q_\eta$.*

The number of relabelings in one record type, to the labels of another, can be quite large: If S has 20 fields and Q has five, then there are $\frac{20!}{(20-5)!} = 1\,860\,480$ relabelings of Q (the number of 5-permutations of 20). It is practically impossible to perform all relabelings and check whether subtypeness holds. An alternative algorithm is presented below for the purpose of this project, where fast computation is enabled by making a few assumptions about the input record types.

4.4.1 Subtype relabeling algorithm

This algorithm handles non-dependent ("basic") and dependent fields separately.

First, when considering relabelings of Q, only the basic fields are included. (In this project, those fields are associated with singleton types of either *Ind* or *Segment*.) This drastically limits the number of relabelings to try: If S has eight basic fields and Q has two, there are only $\frac{8!}{(8-2)!} = 56$ relabelings.

Then, for each relabeling being tried, the remaining (dependent) fields are subtype-checked individually, in order to avoid more relabeling. This means checking $\text{dog(x)} \sqsubseteq \text{dog(x)}$ (true) instead of $\left[\ \text{c}_1\ :\ \text{dog(x)}\ \right] \sqsubseteq \left[\ \text{c}_2\ :\ \text{dog(x)}\ \right]$ (false). If there is some field in Q_η that does not have a subtype field in S, then subtypeness cannot hold, and the rest of the complex fields are skipped in favor of trying the next basic-field relabeling.

For an illustration, consider the following simple example. A relabel-subtype check is being performed on the record types S (Equation 24) and Q (Equation 25).

$$S = \begin{bmatrix} \text{x} & : & \textit{Ind} \\ \text{y} & : & \textit{Ind} \\ \text{z} & : & \textit{Ind} \\ \text{c} & : & \text{right(x, y)} \\ \text{d} & : & \text{left(x, z)} \end{bmatrix} \tag{24}$$

$$Q = \begin{bmatrix} \text{p} & : & \textit{Ind} \\ \text{q} & : & \textit{Ind} \\ \text{e} & : & \text{left(p, q)} \end{bmatrix} \tag{25}$$

1. The first basic-field relabeling to try is $\eta_1 = \{\langle \text{p, x}\rangle, \langle \text{q, y}\rangle\}$, yielding $Q_{\eta_1} = \begin{bmatrix} \text{x} & : & \textit{Ind} \\ \text{y} & : & \textit{Ind} \\ \text{e} & : & \text{left(x, y)} \end{bmatrix}$.
2. However, neither right(x, y) or left(x, z), the dependent fields in S, is a subtype of left(x, y)

3. The next relabeling to try is $\eta_2 = \{\langle \mathrm{p}, \mathrm{x}\rangle, \langle \mathrm{q}, \mathrm{z}\rangle\}$, yielding $Q_{\eta_2} = \left[\begin{array}{lcl} \mathrm{x} & : & \mathit{Ind} \\ \mathrm{z} & : & \mathit{Ind} \\ \mathrm{e} & : & \mathrm{left}(\mathrm{x}, \mathrm{z}) \end{array}\right]$. Similar to before, $Q_{\eta_2}.\mathrm{x} \sqsubseteq S.\mathrm{x}$ and $Q_{\eta_2}.\mathrm{z} \sqsubseteq S.\mathrm{z}$
4. Now, $\mathrm{left}(\mathrm{x}, \mathrm{z}) \sqsubseteq \mathrm{left}(\mathrm{x}, \mathrm{z})$
5. Conclusively, $S \sqsubseteq Q_{\eta_2}$ and thus $S \sqsubseteq_{\mathrm{rlb}} Q$

4.4.2 Restrictions of the algorithm

As a prerequisite, any dependent fields must depend only on basic (non-dependent) fields. For example, the algorithm will not correctly handle $\left[\begin{array}{lcl} \mathrm{x} & : & \mathit{Ind} \\ \mathrm{c}_1 & : & \mathrm{great}(\mathrm{x}) \\ \mathrm{c}_2 & : & \mathrm{believe}(\mathrm{x}, \mathrm{c}_1) \end{array}\right]$.

The algorithm does not recurse into nested record types. This restraint could be eliminated by using flattening, but it is not needed in this scope.

4.4.3 Implementation

A Python implementation is given in Listing 5 (notebook cell 21).

4.5 Additions to PyTTR

Some extensions to PyTTR, listed below, were necessary for the implementation to be possible. Of these, some were added directly to the PyTTR library, because simpler version of the operations of functions were already there. Others were defined in the custom application; in these cases, a notebook cell reference is supplied in the list below.

Python-body functions A TTR function is modelled by the PyTTR `Fun` class, where the function body is made up by another PyTTR object. Application of the function is implemented as substituting the argument in the body object. Some operations here have required more advanced operations. I created a `LambdaFun` subclass of `Fun` to allow any Python code as its body (notebook cell 8).

Copying a record type This facilitates the creation of new record types based on an existing one, without altering the original.

Relabeling multiple fields (notebook cell 4) PyTTR originally only contains a method for relabeling a single field.

Relabel fix When a relabeled field occurs in a sibling dependent field value, the dependent field value must be updated to use the new label. For instance, if 'x' is relabeled to 'y' in $\left[\begin{array}{lcl} \mathrm{x} & : & \mathit{Ind} \\ \mathrm{c} & : & \mathrm{p}(\mathrm{x}) \end{array}\right]$, then $\mathrm{p}(\mathrm{x})$ must be updated to $\mathrm{p}(\mathrm{y})$.

A fix for LazyObject LazyObject is a class in the PyTTR library used for making references between fields. Prior to the fix, it could only be used for paths longer than one item, for instance $r.\mathrm{x}$ but not r.

Flatten for record types The flatten operation was previously implemented for records but not for record types. I added it to the record type class in order to implement Cooper's merge-and-flatten method described in Section 4.2.

```
from itertools import permutations, combinations

def my_subtype_of(sub, sup):
    """Is T a subtype of U? Accept dependent rectype fields (= (fun, args) tuples)
       with simplistic equality test."""
    try:
        return sub.subtype_of(sup)
    except AttributeError:
        return isinstance(sub, tuple) and isinstance(sup, tuple) and show(sub) ==
            show(sup)

def find_subtype_relabeling(S, Q):
    '''Could record type S be a sub type of record type Q if relabeling in Q is
       allowed?'''
    # For each relabeling of basic-type fields
    for sls in permutations(basic_fields(S), len(basic_fields(Q))):
        # Try the basic-fields relabeling of Q
        rlb_basic = dict(zip(basic_fields(Q), sls))
        Q2 = rectype_relabels(Q, rlb_basic)

        # For each Q field, find a S field that is a subtype
        rlb_dep = dict()
        for ql in nonbasic_fields(Q2):
            for sl in nonbasic_fields(S):
                # The new labels must be unique.
                if sl in rlb_dep.values(): continue
                if my_subtype_of(S.field(sl), Q2.field(ql)):
                    rlb_dep[ql] = sl
                    break
            if ql not in rlb_dep:
                break

        # Successful if all non-basic fields match.
        if len(rlb_dep) == len(nonbasic_fields(Q2)):
            return dict(**rlb_basic, **rlb_dep)
    return None
```

Listing 5: The `find_subtype_relabeling` function.

4.6 Demonstration

With the purpose of testing the integrity of the model, and for the sake of illustration, the model has been run on a handful of images from the VQA dataset (Antol et al., 2017) (notebook cell 26). The results are presented in Table 1. Each question is followed by the answer returned by the model as well as a subjective intuition on whether the answer is correct or incorrect.

Note that the accuracy of the answers returned by the model are not a priority in this project. The performance is dependent on what external modules for classification are integrated. This is discussed further in Section 5.

5 Discussion

5.1 Subtype check

Once the perceived scene and the parsed question had been modeled as two situation types, the task of finding an answer to the question was reduced to a subtype check. Under the current restriction to polar questions, type theory has thus proven itself useful and straightforward as a means for the VQA problem (and, in extension, question answering in general).

TTR in particular posed a problem for the subtype check due to the dependence on field labels. This feature necessitated an extra algorithmic layer to allow a label-insensitive comparison in the proposed relabel-subtype check. As a variant of the subtype check, this solves the problem within the scope of this project, but if the model of perception is extended, the computation time grows quickly. In particular, generating more basic-type fields would drastically increase the number of relabelings.

5.2 PyTTR

The PyTTR programming library provided the ability to work with TTR types, objects and operations. Some extensions were needed in order to realise the present project. Some of these were quite simple, providing more or less basic operations through only a few lines of code (such as copying a record type). These could be implemented directly in the PyTTR library. Others provided operations that were quite specific to the use case at hand, such as "combining" record types with label conflict resolving and deduplicated field types. As such, they are less suited for direct inclusion in PyTTR, and should remain in the project-specific source code. (As both PyTTR and the source code for this project are released open-source, all parts of the implementation are open for anyone to reuse.)

5.3 Inference-first

The algorithm of perception described in Section 4.1.6 performs classification of spatial relations on all pairs of individuated objects. In other words, all of the agent's beliefs are inferred at once. Later, when attempting to answer the given question, the beliefs can be queried directly in the subtype check.

In this application, where the goal is to answer a given question, that means spending more effort than necessary. So far as this project goes, this is no significant impediment, as the number of inferences to make (namely, classification of spatial relations) is rather small. In extension, however, the computation time will grow with the amount of inferences (as well as the amount of detected objects) and this approach does not scale.

1.1 Is there an aeroplane?
Yes (Correct)
1.2 Is there a person to the right of a car?
No (Incorrect?)

3.1 Is there a snowboard above a person?
Yes (Incorrect)
3.2 Is there a snowboard below a person?
Yes (Correct)

2.1 Is there a tent?
No (Incorrect)
2.2 Is there a kite above a person?
Yes (Correct)

4.1 Is there a giraffe below a tree?
No (Incorrect)

Table 1: Demonstration of model performance.

A more viable alternative is to first parse the question and then perform inference as needed to arrive to an answer. If the question is about the spatial relation between a dog and a person, it will probably be enough to see that there is a dog and a person in the scene, and that the spatial relation between them matches the one expressed in the question.

6 Conclusions

Within this project, the foundations of visual question answering (VQA) have been implemented in Type Theory with Records (TTR). The result is an executable application powered by PyTTR.

6.1 Suitability

This project has shown that TTR can indeed be used to connect existing vision and language systems. It enables a detailed model of multi-modal perception and semantics. TTR is the single framework that serves to operate all parts of the pipeline: perception, language and grounding.

This is one of the few applications of the recently developed PyTTR library, which has enabled executing this model on actual data. Extensions to PyTTR were made where needed.

6.2 Benefits

The formal-semantic framework behind the application provides transparency and reversibility. This can be contrasted to neural-network-based models, where the path from input to results is encoded in impermable statistical data. The framework also enables relatively simple implementation of operations (such as verifying a proposition).

6.3 Can it tell us something about semantics?

The problem of answering polar questions has been modeled as a subtype check between two situation types: a grounded scene type and a hypothetical question type. This is different than the commonly established modeling of questions as sets of propositions (Hamblin, 1973). The question could be cast into a set as $\{T \mid T \sqsubseteq_{\text{rlb}} Q\}$, but it still relies on the subtype check.

6.4 Future work

The agent structure is a rather simplistic model, tailored for the use case at hand. It accepts as input an image followed by a number of questions. There is no semantic connection between the current image and question, other than the sequence in which they are input. There is also nothing like a dialogue system; the background to any answer from the agent is the latest image and the latest question only. Changing these things could be important for further some extensions.

Spatial classification and language parsing were achieved using minimal and simplistic implementations. Substituting them with sophisticated systems would make for wider question coverage and higher question answering scores. For instance, Logan & Sadler (1996) proposes spatial templates, regions of acceptability and compound relations (like “above to the right of”).

As discussed in Section 3.4, the present treatment of spatial relations excludes the assignment of the frame of reference, as well as functional aspects, in favor of model simplicity and easy implementation. Both features

are treated in terms of TTR by Dobnik & Cooper (2013). The former requires a notion of the orientation of reference objects. Assuming an object classifier with this capability were available, implementing intrinsic spatial relations would not be a large step from the present model. Such a classifier might detect that a car is facing left in the image; an object to the right in the image could then be classified as being "behind" the car. Support for the functional aspect of spatial relations requires two things. Firstly, classifiers for functional relations (such as protects(o_3.a, o_1.a, o_2.a) in Dobnik & Cooper (2013), for an umbrella protecting a man from rain). Secondly, prediction from a set of functional relations to spatial classifiers that are sensitive to those functional relations. The second is needed to activate the appropriate spatial relation term depending on which functional relations are true according to the classifiers in the first. That is, if the relation between the umbrella and the man is classified as "protects", then the spatial classifier selected for "over" should be one that includes this condition.

Extending the language domain should be an interesting topic for further research. Keeping within the problem domain of geometric spatial relations, allowing other question types than polar questions is one direction to explore. Dobnik (2009, p. 156) lists four basic question types: "Where is the chair?", "Is the table to the left of the chair?" (this is the focus of this project), "What is to the left of the chair?" and "What is the chair to the left of?" Another is to widen the problem domain and add more properties and relations than a primary object class ("car") and spatial relations. For instance, attribute classifiers could recognise color, size, facial expressions and positions, in order to authorise questions such as "Is the girl sitting down?" and "Where is the red flower?". Action event classifiers could identify actions such as "riding" and "talking to".

The use of formal frameworks for question-answering tasks especially invites techniques for logical inference. Consider an image of a person wearing glasses, and the question "Does this person have 20/20 vision?" It is reasonable to assume that a person is wearing glasses because they do not have perfect eyesight, to which "20/20 vision" is synonymous. Logical inference could help to achieve the synonymity as well as the relationship between eyesight and wearing glasses.

The relabel-subtype implementation in the `find_subtype_relabeling` Python function makes some assumptions about the record types being checked (Section 4.4). More specifically, a field type may not be dependent on another dependent field, and record types may not be nested. More complex applications could be achieved if the implementation were extended to handle these cases and eliminate the assumptions.

Furthermore, as discussed in Section 5.1, the implementation does not scale to handle some of the extensions mentioned here. One measure to improve the performance is to introduce type-sensitivity when finding feasible basic-field relabelings. For example, do not try to relabel a field with type *Ind* to one with type *Int*, since *Int* $\not\sqsubseteq$ *Ind*. In the general case, however, the relabel-subtype algorithm may need to be replaced with one that utilises theorem proving in order to find a subtype-compatible relabeling.

As mentioned in Section 5.3, this algorithm does not scale well, as it performs classification of spatial relations (and, in extension, any other kind of inference) before parsing the question. This problem could be overcome by first extracting only direct information from the image, namely the result of the *ObjDetector*; then, when the question has been parsed, finding out what inference steps are necessary to perform. The process of finding required inferences may need to be integrated with the amended subtype strategy just mentioned. This is a necessary condition for most extensions of the language domain and inference.

From a software engineering perspective, the code implemented in this project could be better structured. The code was implemented in a single Jupyter Notebook file, for the sake of easy experimenting. It could be rewritten in standard Python files to make it easier to run on different systems.

The PyTTR codebase deserves some additional development. Its documentation is currently in the form of

Jupyter Notebook files, but it would be helpful to also add in-code comments that explain how the library can be used. It would also help to package the code properly so it could be released and imported to projects like this one more easily.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., & Zheng, X. (2015). TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. Software available from tensorflow.org.

Andreas, J., Rohrbach, M., Darrell, T., & Klein, D. (2016). Learning to Compose Neural Networks for Question Answering. In *HLT-NAACL*.

Antol, S., Agrawal, A., Lu, J., Mitchell, M., Batra, D., Zitnick, C. L., & Parikh, D. (2017). VQA: Visual Question Answering. *International Journal of Computer Vision*, 123(1), 4–31.

Barwise, J. & Perry, J. (1981). Situations and Attitudes. *Journal of Philosophy*, 78(11), 668–691.

Bird, S., Klein, E., & Loper, E. (2009). *Natural Language Processing with Python*. O'Reilly Media, Inc., 1st edition.

Blackburn, P. & Bos, J. (2003). Computational Semantics. *Theoria: An International Journal for Theory, History and Foundations of Science*, (pp. 27–45).

Blaschko, M. B. & Lampert, C. H. (2008). Learning to Localize Objects with Structured Output Regression. In D. Forsyth, P. Torr, & A. Zisserman (Eds.), *Computer Vision – ECCV 2008*, volume 5302 (pp. 2–15). Berlin, Heidelberg: Springer Berlin Heidelberg.

Church, A. (1940). A Formulation of the Simple Theory of Types. *The Journal of Symbolic Logic*, 5(2), 56–68.

Cooper, R. (2005a). Austinian truth, attitudes and type theory. *Research On Language And Computation, Vol. 3, 2005, pp. 333-.362*, (pp. 333–362).

Cooper, R. (2005b). Records and Record Types in Semantic Theory. *Journal of Logic and Computation*, 15(2), 99–112.

Cooper, R. (2012). Type Theory and Semantics in Flux. *Handbook of the Philosophy of Science*, 14, 271–323.

Cooper, R. (2016). Type Theory and Language: From Perception to Linguistic Communication. *Draft of book chapters available from https://sites.google.com/site/typetheorywithrecords/drafts (accessed on 2018-01-17)*.

Cooper, R. (2017). PyTTR. https://github.com/GU-CLASP/pyttr.

Coquand, T. (2015). Type Theory. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, summer 2015 edition.

Coventry, K., Prat-Sala, M., & Richards, L. (2001). The Interplay between Geometry and Function in the Comprehension of Over, Under, Above, and Below. *Journal of Memory and Language*, 44, 376–398.

Dobnik, S. (2009). *Teaching Mobile Robots to Use Spatial Words*. PhD thesis, The Queen's College, University of Oxford.

Dobnik, S. & Cooper, R. (2013). Spatial Descriptions in Type Theory with Records. In *Proceedings of IWCS 2013 Workshop on Computational Models of Spatial Language Interpretation and Generation (CoSLI-3)* (pp. 1–6). Potsdam, Germany: Association for Computational Linguistics.

Dobnik, S. & Cooper, R. (2017). Interfacing Language, Spatial Perception and Cognition in Type Theory with Records. *Journal of Language Modelling*, 5(2), 273–301.

Dobnik, S., Cooper, R., & Larsson, S. (2012). Modelling Language, Action, and Perception in Type Theory with Records. In *International Workshop on Constraint Solving and Language Processing* (pp. 70–91).: Springer.

Dobnik, S. & Kelleher, J. D. (2017). Modular Mechanistic Networks: On Bridging Mechanistic and Phenomenological Models with Deep Neural Networks in Natural Language Processing. In S. Dobnik & S. Lappin (Eds.), *Proceedings of the Conference on Logic and Machine Learning in Natural Language (LaML 2017), Gothenburg, 12–13 June 2017*, volume 1 of *CLASP Papers in Computational Linguistics* (pp. 1–11). Gothenburg, Sweden: Department of Philosophy, Linguistics and Theory of Science (FLOV), University of Gothenburg CLASP, Centre for Language and Studies in Probability.

Frege, G. (1948). Sense and Reference. *The Philosophical Review*, 57(3), 209–230.

Garnham, A. (1989). A Unified Theory of the Meaning of Some Spatial Relational Terms. *Cognition*, 31(1), 45–60.

Hamblin, C. L. (1973). Questions in Montague English. *Foundations of Language*, 10(1), 41–53.

Harnad, S. (1990). The Symbol Grounding Problem. *Physica D: Nonlinear Phenomena*, 42(1-3), 335–346.

He, K., Gkioxari, G., Dollár, P., & Girshick, R. B. (2017). Mask R-CNN. *2017 IEEE International Conference on Computer Vision (ICCV)*, (pp. 2980–2988).

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 770–778). Las Vegas, NV, USA: IEEE.

Kohlhase, M., Kuschert, S., & Pinkal, M. (1996). A Type-Theoretic Semantics for λ-DRT. In P. Dekker & M. Stokhof (Eds.), *Proceedings of the 10th Amsterdam Colloquium* (pp. 479–498). Amsterdam.

Larsson, S. (2011). Do Dialogues Have Content? In *Sylvain Pogodalla And Jean-Philippe Prost, Eds: Proceedings Of Logical Aspects Of Computational Linguistics (Lacl 2011), Springer Lecture Notes In Computer Science.*, volume 6736.

Larsson, S. & Cooper, R. (2009). Towards a Formal View of Corrective Feedback. In *Proceedings of the EACL 2009 Workshop on Cognitive Aspects of Computational Language Acquisition* (pp. 1–9).: Association for Computational Linguistics.

Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., & Zitnick, C. L. (2014). Microsoft COCO: Common Objects in Context. In D. Fleet, T. Pajdla, B. Schiele, & T. Tuytelaars (Eds.), *Computer Vision – ECCV 2014*, volume 8693 (pp. 740–755). Cham: Springer International Publishing.

Logan, G. D. & Sadler, D. D. (1996). A computational analysis of the apprehension of spatial relations. In *Language and Space.*, Language, speech, and communication. (pp. 493–529). Cambridge, MA, US: The MIT Press.

Lowe, D. G. (1999). Object Recognition from Local Scale-Invariant Features. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, volume 2 (pp. 1150–1157 vol.2).

Martin-Löf, P. & Sambin, G. (1984). *Intuitionistic Type Theory*, volume 9. Bibliopolis Napoli.

Montague, R. (1974). *Formal Philosophy; Selected Papers of Richard Montague*. New Haven: Yale University Press.

Pustejovsky, J. (1990). Perceptual Semantics: The Construction of Meaning in Artificial Devices. In *Proceedings. 5th IEEE International Symposium on Intelligent Control 1990* (pp. 86–91 vol.1).

Ranta, A. (2011). *Grammatical Framework: Programming with Multilingual Grammars*. Stanford: CSLI Publications. ISBN-10: 1-57586-626-9 (Paper), 1-57586-627-7 (Cloth).

Redmon, J. (2013). Darknet: Open Source Neural Networks in C. https://pjreddie.com/darknet/ (accessed on 2018-09-21).

Redmon, J. (2018). YOLO: Real-Time Object Detection. https://pjreddie.com/darknet/yolov2/ (accessed on 2018-09-21).

Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You Only Look Once: Unified, Real-Time Object Detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 779–788). Las Vegas, NV, USA: IEEE.

Regier, T. & Carlson, L. A. (2001). Grounding Spatial Language in Perception: An Empirical and Computational Investigation. *Journal of Experimental Psychology: General*, 130(2), 273–298.

Searle, J. R. (1980). Minds, Brains, and Programs. *Behavioral and Brain Sciences*, 3(03), 417.

Steels, L. (2008). The symbol grounding problem has been solved, so what's next? In *Symbols and Embodiment*. Oxford University Press.

Trieu, T. H. (2018). Darkflow. https://github.com/thtrieu/darkflow (accessed on 2018-09-21).

Whitehead, A. N. & Russell, B. (1910). *Principia Mathematica*. Cambridge [usw.]: Cambridge Univ. Pr, 2. ed., reprint edition. OCLC: 252383138.

Appendix A Code

Cell 1

```
import sys
sys.path.append('pyttr')
from pyttr.ttrtypes import *
from pyttr.utils import *
import PIL.Image
```

Cell 2

```
Ind = BType('Ind')
```

Cell 3

```
# Can be called with multiple args *at the end* of a code block to illustrate PyTTR types and objects.
def latex(*objs, code=False):
    # code = True # Uncomment to always get code.
    texcode = '\n\n'.join(to_ipython_latex(obj) for obj in objs)
    if code: print(texcode)
    return Latex(texcode)

# Redefine Image.show() to work with Rec.show().
def image_show(self):
    return str(self)
PIL.Image.Image.show = image_show
```

Cell 4

```
def rectype_relabels(T, rlbs):
    """Relabel multiple fields, given a dict of from-to pairs."""
    T = T.copy()
    # Add a temporary label to each pair, to avoid overwriting.
    # This is not a guard against {a:c, b:c}, but against {a:b, b:c}.
    rlbs = [(l1, l2, gensym('rlb')) for l1, l2 in rlbs.items()]
    # First relabel all fields to tmp labels, then all to desired labels.
    for l1, l2, l_tmp in rlbs:
        T.Relabel(l1, l_tmp)
    for l1, l2, l_tmp in rlbs:
        T.Relabel(l_tmp, l2)
    return T

def is_basic_type(T):
    """Whether a type is a "basic field", i.e. a BType or a SingletonType of a BType."""
    return (isinstance(T, SingletonType) and is_basic_type(T.comps.base_type)) or isinstance(T, BType)

def basic_fields(T):
    """The labels of basic fields in a RecType."""
    return [k for k, v in T.fields() if is_basic_type(v)]

def nonbasic_fields(T):
    """The labels of non-basic fields in a RecType."""
    return [k for k, v in T.fields() if not is_basic_type(v)]
```

Cell 5

```
T = RecType({'a': 1, 'b': 2, 'c': 3})
# Valid relabeling; is handled.
```

```
print(show(rectype_relabels(T, {'a': 'b', 'b':'d'})))
# Invalid relabeling; fails. Fix your usage.
print(show(rectype_relabels(T, {'a': 'c', 'b':'c'})))
```

Cell 6

```
# mkptype saves the created PType objects. If called a second time with the same pred and vars,
# the existing object is returned, instead of instantiating PType every time.
ptypes = dict()
def mkptype(sym, types=[Ind], vars=['v']):
    """Make preds and ptypes identifiable by their predicate names."""
    id = '/'.join([sym, ','.join(show(type) for type in types), ','.join(vars)])
    if id not in ptypes:
        ptypes[id] = PType(Pred(sym, types), vars)
    return ptypes[id]

def create_fun(pred_name, vars=['a']):
    """Create a function of a given number of Inds (length of vars).

    Example: create_fun('give', 'abc') -> \a. \b. \c. give(a, b, c)
    """
    fun = mkptype(pred_name, types=[Ind]*len(vars), vars=vars)
    for v in reversed(vars):
        fun = Fun(v, Ind, fun)
    return fun

latex(create_fun('give', 'abc'))
```

Cell 7

```
def dedupe(T):
    """Make a copy of a record type without duplicated field values."""
    for l,v in T.fields():
        # Are there more fields with the same value?
        l2s = [l2 for l2,v2 in T.fields() if l2 != l
               # Dedupe singleton types and complex types.
               and (isinstance(v, SingletonType) or not is_basic_type(v))
               # Using show is error-prone, pyttr should have an equals method.
               and show(v) == show(v2)]
        if len(l2s):
            # Relabel all these fields to the same label, overwriting until one remains.
            for l2 in l2s:
                T.Relabel(l2, l)
            # Dependent fields have changed, so start over.
            return dedupe(T)
    # No more duplicates.
    return T

T = RecType({
    'x': SingletonType(Ind, 'a'),
    'y': SingletonType(Ind, 'a'),
    'c': (create_fun('foo'), ['x']),
    'd': (create_fun('foo'), ['y']),
})
latex(T, dedupe(T.copy()))
```

Cell 8

```
class LambdaFun(Fun):
    """Models a unary Python function as a TTR function.

    Type/subtype checking on a Fun will app() on a HypObj and use the result.
    In a LambdaFun, the body function may require a real object,
    and would have to be rewritten to be able to handle a HypObj for this purpose.
    To avoid this, an example argument can be specified in __init__,
```

```
    in which case that will be used in place of a HypObj.
    """
    def __init__(self,dom,body, example=None):
        self.__setattr__('domain_type',dom)
        self.__setattr__('body',body)
        self.__setattr__('example', example)
    def show(self):
        return 'lambda r' + ':' + self.domain_type.show() + ' . ' + self.body.__name__ + '(r)'
    def to_latex(self):
        return '\\lambda r' + ':' + self.domain_type.to_latex() + '\\ .\\ ' + self.body.__name__ + '(r)'
    def validate(self):
        return isinstance(self.domain_type,Type)

    def app(self,arg):
        if self.example is not None and LambdaFun.is_hypobj(arg):
            arg = self.example

        res = self.body(arg)
        if 'eval' in dir(res):
            return res.eval()
        else:
            return res

    def is_hypobj(r):
        return isinstance(r, HypObj) or \
            (isinstance(r, Rec) and forsome([v for l,v in r.fields()], LambdaFun.is_hypobj))

    def subst(self,v,a):
        return self

P = FunType(Ind, Ty)
T = RecType({'f': P})
def on_a(r):
    return RecType({'c': r.f.app('z')})
fun = LambdaFun(T, on_a, Rec({'f': create_fun('ex')}))
print(show(fun.app(Rec({'f': create_fun('dog')}))))

# With hypobj.
F = FunType(T, RecTy)
print(F.query(fun))
```

Cell 9

```
# Basic types.

# Ind has already been defined.

Int = BType('Int')
Int.learn_witness_condition(lambda x: isinstance(x, int))
print(Int.query(365))

String = BType('String')
String.witness_conditions.append(lambda s: isinstance(s, str))
print(String.query('Hello World!'))

Image = BType('Image')
Image.learn_witness_condition(lambda x: isinstance(x, PIL.Image.Image) or x is None)
img = PIL.Image.open('res/dogride.jpg') # https://www.flickr.com/photos/hickatee/34017375600
print(Image.query(img))

# Segment type: a rectangular area of a given image.

Segment = RecType({'cx': Int, 'cy': Int, 'w': Int, 'h': Int})
print(Segment.query(Rec({'cx': 100, 'cy': 150, 'w': 40, 'h': 20})))
```

Cell 10

```
PTy = Type('PType')
```

```
PTy.learn_witness_condition(lambda p: isinstance(p, PType) or (
    isinstance(p, HypObj) and forsome(p.types, lambda t: isinstance(t, PType))))

Ppty = FunType(Ind, PTy)
Obj = RecType({'seg': Segment, 'pfun': Ppty})
Objs = ListType(Obj)
ObjDetector = FunType(Image, Objs)

latex(ObjDetector)
```

Cell 11

```
# Instantiate YOLO.

from darkflow.net.build import TFNet

tfnet = TFNet({"model": "yolo/yolo.cfg", "load": "yolo/yolo.weights",
    'config': 'yolo', "threshold": 0.2})
```

Cell 12

```
# Function to apply YOLO to a given image.

import numpy as np

yolo_out = dict()
def yolo(img):
    """Invokes YOLO on a PIL image, caches and returns the result."""
    if str(img) not in yolo_out:
        res = tfnet.return_predict(np.array(img))
        # Save the most confident detections.
        res.sort(key=lambda o: -o['confidence'])
        yolo_out[str(img)] = res[:7]
    return yolo_out[str(img)]

def yolo_coords(o):
    """Extract the coordinates from a YOLO output item as ((x0,y0), (x1,y1))."""
    return (o['topleft']['x'], o['topleft']['y']), (o['bottomright']['x'], o['bottomright']['y'])

def xy1xy2_to_cwh(x1, y1, x2, y2):
    "'Transform to center, width and height."'
    return {'cx': int(x1/2 + x2/2), 'cy': int(y1/2 + y2/2), 'w': x2 - x1, 'h': y2 - y1}

def yolo_reformat(o):
    """Discards the confidence item and reformats the coordinates."""
    return {'label': o['label'],
        'loc': xy1xy2_to_cwh(*sum(yolo_coords(o), ()))}
```

Cell 13

```
from PIL import ImageFont, ImageDraw
from IPython.display import display

# Generate distinguishable colors.
phi = 2 / (1 + 5 ** .5)
colors = ('hsl({}, 90%, 70%)'.format(int(x * 360)) for x in count(0, phi))

def yolo_annotate(img):
    """Displays the image with YOLO results annotated."""
    img_annotated = img.copy()
    draw = ImageDraw.Draw(img_annotated)
    for o in yolo(img):
        color = next(colors)
        draw.rectangle(yolo_coords(o), outline=color)
        draw.text(yolo_coords(o)[0], o['label'], fill=color)
```

```
    display(img_annotated)

yolo_annotate(img)
# Modified image under licence CC-by-nc-sa: https://creativecommons.org/licenses/by-nc-sa/2.0/
```

Cell 14

```
# Representing detected objects in TTR.

def yolo_detector(i):
    """Creates IndObj records for YOLO results."""
    for o in yolo(i):
        o = yolo_reformat(o)
        yield Rec({
            'seg': Rec(o['loc']),
            'pfun': create_fun(o['label'].replace(' ', '_')),
        })
ObjDetector.witness_cache.append(yolo_detector)

objs = list(yolo_detector(img))

print(ObjDetector.query(yolo_detector))
print(Objs.query(objs))
print(Obj.query(objs[0]))
print(Ppty.query(objs[0].pfun))
print(Segment.query(objs[0].seg))

latex(objs)
```

Cell 15

```
location_ptype = mkptype('location', [Ind, Segment], ['k', 'l'])
cl_fun = Fun('k', Ind, Fun('l', Segment, location_ptype))

IndObj = RecType({
    'x' : Ind,
    'loc' : Segment,
    'cp' : PTy,
    'cl' : (cl_fun, ['x', 'loc']),
})
IndFun = FunType(Obj, RecTy)

def indfun(r):
    if not Obj.query(r):
        raise ValueError('Input must be Obj')
    indobj = RecType({
        'x': SingletonType(Ind, Ind.create()),
        'cp': (r.pfun, ['x']),
        'loc': SingletonType(Segment, r.seg),
        'cl': (cl_fun, ['x', 'loc']),
    })
    if not indobj.subtype_of(IndObj):
        raise ValueError('The result is not a subtype of IndObj')
    return indobj
IndFun.witness_cache.append(indfun)

indobjs = [indfun(r) for r in objs]

print(Obj.query(objs[1]))
print(RecTy.query(indobjs[1]))
print(indobjs[1].subtype_of(IndObj))
print(IndObj.query(indobjs[1].create_hypobj()))
latex(indobjs)
```

Cell 16

```
from itertools import product
```

```
LocTup = RecType({'lo': IndObj, 'refo': IndObj})
RelClf = FunType(LocTup, RecTy)

location_relation_classifiers = {
    # predicate: classifier
    'left': lambda a, b: a.cx < b.cx,
    'right': lambda a, b: a.cx > b.cx,
    'above': lambda a, b: a.cy < b.cy,
    'below': lambda a, b: a.cy > b.cy,
}

def relclf(r, pred, f):
    # Support type checking of this function, which uses HypObj.
    if isinstance(r.lo.loc.cx, HypObj):
        return RecType()
    # @TODO Why not? IndObj.query(r.lo) and IndObj.query(r.refo)
    if f(r.lo.loc, r.refo.loc):
        c = create_fun(pred, 'ab')
        return RecType({
            'x': SingletonType(Ind, r.lo.x),
            'y': SingletonType(Ind, r.refo.x),
            'cr': (c, ['x', 'y']),
        })
    return RecType()

def get_relclfs():
    relclfs = []
    for pred, f in location_relation_classifiers.items():
        relclfs.append(lambda r, pred=pred, f=f: relclf(r, pred, f))
    return relclfs

relclfs = [LambdaFun(LocTup, relclf) for relclf in get_relclfs()]

loctups = []
def find_all_rels(indobjs):
    """Find all relations between IndObj records."""
    for relclf in relclfs:
        for loT, refoT in product(indobjs, indobjs):
            loctup = Rec({'lo': loT.create(), 'refo': refoT.create()})
            loctups.append(loctup)
            yield relclf.app(loctup)

rels = list(rel for rel in find_all_rels(indobjs) if len(rel.labels()) > 0)

latex(rels)
```

Cell 17

```
# The classical method using nesting and flattening.

from functools import reduce

Prev = LambdaFun(RecTy, lambda old:
         LambdaFun(RecTy, lambda new:
           RecType({'prev': old}).merge(new).flatten()))

def combine_prev(Ts):
    f = lambda old, new: Prev.app(old).app(new)
    return dedupe(reduce(f, Ts, RecType()))
```

Cell 18

```
# Custom method with unique labels before merging.

def unique_labels(T):
    """Relabel a RecType so each field label is unique over all RecTypes."""
    rlb = ((l, gensym(l)) for l in T.labels() if '_' not in l)
```

```
    return rectype_relabels(T, dict(rlb))

def combine(Ts):
    """Combine a list of belief record types into one."""
    f = lambda a, b: a.merge(unique_labels(b))
    return dedupe(reduce(f, Ts, RecType()))
```

Cell 19

```
bel = indobjs + rels

# combine = combine_prev
bel_comb = combine(bel)

latex(bel_comb)
```

Cell 20

```
import nltk
from nltk.sem.logic import ApplicationExpression, AndExpression, ExistsExpression, ConstantExpression

# Parsing to PyTTR cannot really be done directly. NLTK feature grammars support output in the form
# of strings or FOL, where variable substitution is only allowed in FOL. Here we produce a FOL
# expression and later translate it to a PyTTR record type.

grammar = nltk.grammar.FeatureGrammar.fromstring(r"'
%start S
S[SEM=<?np(?vp)>] -> NP[SEM=?np] VP[SEM=?vp]
S[SEM=?q] -> QS[SEM=?q]
QS[SEM=<?np(?pp)>] -> 'is' 'there' NP[SEM=?np] PP[SEM=?pp]
QS[SEM=<?np(\P.true)>] -> 'is' 'there' NP[SEM=?np]

NP[SEM=<?det(?n)>] -> Det[SEM=?det] N[SEM=?n]
Det[SEM=<\P Q.exists x.(P(x) & Q(x))>] -> 'a' | 'an'

VP[SEM=?pp] -> 'is' PP[SEM=?pp]
PP[SEM=<\x.(?np(\y.?prep(x, y)))>] -> Prep[SEM=?prep] NP[SEM=?np]

Prep[SEM=<left>] -> 'to' 'the' 'left' 'of'
Prep[SEM=<right>] -> 'to' 'the' 'right' 'of'
Prep[SEM=<above>] -> 'above'
Prep[SEM=<below>] -> 'below'
N[SEM=<aeroplane>] -> 'aeroplane'
N[SEM=<backpack>] -> 'backpack'
N[SEM=<bicycle>] -> 'bicycle'
N[SEM=<car>] -> 'car'
N[SEM=<dog>] -> 'dog'
N[SEM=<giraffe>] -> 'giraffe'
N[SEM=<kite>] -> 'kite'
N[SEM=<person>] -> 'person'
N[SEM=<snowboard>] -> 'snowboard'
N[SEM=<tent>] -> 'tent'
N[SEM=<tree>] -> 'tree'
"')
parser = nltk.FeatureChartParser(grammar)

texts = [
    'A dog is to the left of a bicycle',
    'Is there a dog to the left of a bicycle?',
    'Is there a person?',
]

def fol_to_pyttr(expr, T=RecType()):
    """Turns a FOL object into a RecType."""
    # Existential quantifier -> Ind field.
    if isinstance(expr, ExistsExpression):
        T.addfield(str(expr.variable), Ind)
        return fol_to_pyttr(expr.term, T)
```

```
    # Application -> ptype, e.g. left(x, y)
    if isinstance(expr, ApplicationExpression):
        pred, args = expr.uncurry()
        # Create a PType function, e.g. lambda x:Ind . dog(x)
        fun = create_fun(str(pred), 'abcd'[:len(args)])
        T.addfield(gensym('c'), (fun, [str(a) for a in args]))
        return T

    # For and-expressions, interpret each term.
    if isinstance(expr, AndExpression):
        fol_to_pyttr(expr.first, T)
        fol_to_pyttr(expr.second, T)
        return T

    # A constant function in the "is there an X" rule trivially gives "true".
    if isinstance(expr, ConstantExpression) and str(expr.variable) == 'true':
        return T

    raise ValueError('Unknown expression: ' + str(type(expr)) + ' ' + str(expr))

def eng_to_pyttr(text):
    # Tokenize.
    sent = text.lower().strip('.?!').split()
    # NLTK-parse to syntax tree.
    try:
        trees = parser.parse(sent)
    except ValueError:
        return RecType()
    # Extract semantic representation for the tree.
    sem = nltk.sem.root_semrep(list(trees)[0])
    # Interpret to TTR record type.
    T = fol_to_pyttr(sem, RecType())
    return T

print(texts[1])
q = eng_to_pyttr(texts[1])
latex(q)
```

Cell 21

```
from itertools import permutations, combinations

def my_subtype_of(sub, sup):
    """Is T a subtype of U? Accept dependent rectype fields with simplistic equality test."""
    try:
        return sub.subtype_of(sup)
    except AttributeError:
        return isinstance(sub, tuple) and isinstance(sup, tuple) and show(sub) == show(sup)

def find_subtype_relabeling(S, Q):
    "'Could record type S be a sub type of record type Q if relabeling in Q is allowed?"'
    # For each relabeling of basic-type fields
    for sls in permutations(basic_fields(S), len(basic_fields(Q))):
        # Try the basic-fields relabeling of Q
        rlb_basic = dict(zip(basic_fields(Q), sls))
        Q2 = rectype_relabels(Q, rlb_basic)

        # For each Q field, find a S field that is a subtype
        rlb_dep = dict()
        for ql in nonbasic_fields(Q2):
            for sl in nonbasic_fields(S):
                # The new labels must be unique.
                if sl in rlb_dep.values(): continue
                if my_subtype_of(S.field(sl), Q2.field(ql)):
                    rlb_dep[ql] = sl
                    break
            if ql not in rlb_dep:
                break

        # Successful if all non-basic fields match.
```

```
        if len(rlb_dep) == len(nonbasic_fields(Q2)):
            return dict(**rlb_basic, **rlb_dep)
    return None
```

Cell 22

```
utts_rlb = []
for text in texts:
    q = eng_to_pyttr(text)
    rlb = find_subtype_relabeling(bel_comb, q)
    print(text + ' - ' + str(bool(rlb)))
    if rlb:
        utt_rlb = rectype_relabels(q, rlb)
        utts_rlb.append(utt_rlb)
latex(utts_rlb)
```

Cell 23

```
AgentState = RecType({
    'img': Image,
    'perc': Objs,
    'bel': ListType(RecTy),
    'utt': String,
    'que': RecTy
})
Agent = RecType({
    'objdetector': ObjDetector,
    'indfun': IndFun,
    'relclfs': ListType(RelClf),
    'state': AgentState,
})
latex(Agent)
```

Cell 24

```
st = Rec({
    'perc': [],
    'bel': [],
    'img': None,
    'utt': '',
    'que': RecType(),
})
ag = Rec({
    'objdetector': yolo_detector,
    'indfun': indfun,
    'relclfs': relclfs,
    'state': st,
})

print(AgentState.query(st))
print(Agent.query(ag))
```

Cell 25

```
# The combined beliefs are "cached".
bel = None

def agent_see(ag, img):
    if not Agent.query(ag): raise ValueError
    ag.state.img = img
    ag.state.perc = list(ag.objdetector(ag.state.img))
    ag.state.bel = [indfun(r) for r in ag.state.perc]
```

```
    for Ta, Tb in product(ag.state.bel, ag.state.bel):
        if Ta == Tb: continue
        loctup = RecType({'lo': Ta, 'refo': Tb})
        for relclf in ag.relclfs:
            new_bel = relclf.app(loctup.create_hypobj())
            if new_bel:
                ag.state.bel.append(new_bel)

    global bel
    bel = combine(ag.state.bel)
    if not Agent.query(ag): raise ValueError

def agent_hear(ag, text):
    if not Agent.query(ag): raise ValueError
    ag.state.utt = text
    ag.state.que = eng_to_pyttr(text)
    if not Agent.query(ag): raise ValueError

def agent_answer(ag):
    rlb = find_subtype_relabeling(bel, ag.state.que)
    return bool(rlb)

q = 'is there a person above a bicycle'
print('Q:', q)
agent_see(ag, img)
agent_hear(ag, q)
a = agent_answer(ag)
print('A:', a)
```

Cell 26

```
vqa_items = [
    {'imgpath': 'res/vqa1.jpg', 'questions': [
        'Is there an aeroplane?',
        'Is there a person to the right of a car?',
    ]},
    {'imgpath': 'res/vqa2.jpg', 'questions': [
        'Is there a tent?',
        'Is there a kite above a person?',
    ]},
    {'imgpath': 'res/vqa3.jpg', 'questions': [
        'Is there a snowboard above a person?',
        'Is there a snowboard below a person?',
    ]},
    {'imgpath': 'res/vqa4.jpg', 'questions': [
        'Is there a giraffe below a tree?',
    ]},
]

def vqa_eval(imgpath, questions):
    img = PIL.Image.open(imgpath)
    yolo_annotate(img)
    agent_see(ag, img)
    for question in questions:
        agent_hear(ag, question)
        ans = 'Yes' if agent_answer(ag) else 'No'
        print(question + ' - ' + ans)

for item in vqa_items:
    vqa_eval(**item)
```