Types and probability
Implementing probabilistic TTR

Robin Cooper
University of Gothenburg

CLASP Seminar, 14th April 2021

Outline

Outline

Probabilities associated with judgements

- TTR — a type theory with records (Cooper, 2012; Cooper and Ginzburg, 2015; Cooper, in prep)
- a *rich* theory of types taking ideas from Per Martin-Löf and others. Whereas Montague had types like *Entity* and *TruthValue* we can have types like *Dog* and *BoyHugsDogSituation*
- probabilistic TTR (Cooper *et al.*, 2015)

Two kinds of probabilities

- the probability that an object, a, if of type T — $p(a : T)$
- the probability that there is an object of type T — $p(T)$
- Given that types can be used to model propositions, the second probability can be construed as the probability that T is true (cf. the judgement $T\ \mathrm{true}$ in Martin-Löf type theory)

Outline

A Python implementation of TTR

- `https://github.com/GU-CLASP/pyttr`
- an attempt to implement something useful for agents making judgements in a changing environment
- probabilistic TTR (as far as I have got with it) is presented in the notebook `https://github.com/GU-CLASP/pyttr/blob/master/probttr.ipynb`

Dynamic processing

- new types can be created on the fly — don't have to be declared at the beginning of the program
- if $a : T$, we call a a *witness* for T
- witnesses can be added to types on the fly
- **intuition:** useful for agents observing new objects and making judgements about their types
- objects can be judged to be of a type because they meet a *witness condition* associated with that type
- new witness conditions can be associated with a type on the fly
- **intuition:** useful for agents engaging in a learning process and refining the definition of a type

Types as an interface to the world

- witness conditions in `pyttr` can call on arbitrary external modules which need not be implemented in `pyttr`
- for example, the witness condition for a type may call a classifier created by standard machine-learning techniques (Matsson, 2018; Utescher, 2019)

Outline

Witness caches

non-probabilistic pyttr

- objects judged to be witnesses for a type are stored in a witness cache for that type
- leads to *negation as failure* (logic programming)

probabilistic pyttr

- objects stored in a witness cache for the type together with a probability
- means that a negative judgement (zero probability) can be stored in the witness cache

Probability intervals

- Actually, we use probability *intervals* rather than just probabilities
- $[n, m]$ where n is the minimum and m the maximum
- This means that we can represent three truth values:

True	[1,1]
False	[0,0]
Don't know	[0,1]

- . . . plus uncountably many more (actually in the implementation a large finite number using floating point numbers)

Methods

- much of the work is done by *methods*, in the sense of object oriented programming
- means that methods with the same name can be defined differently for different types (organized into Python *classes*)
- often we refer to these as a single method, defined differently for different classes of objects

Judging and querying

- in `pyttr` (both probabilistic and non-probabilistic) there are two methods associated with what is called *judgement* in a type theory
- `judge()` is used to *assert* that an object is of a certain type (with some probability) — *cf. assertion* in logic programming
- `query()` is used to *query* whether (or with what probability) an object is of a certain type — *cf. query* in logic programming

Outline

Witness cache

We declare a new type, T, whose witness cache is initialized to be empty — no judgements have been made.

```
T = Type()
print(T.witness_cache)
```

```
([], [])
```

Making a judgement

We judge *a* to be of type *T* with probability .5.

```
show(T.judge('a',.5))
```

```
'0.5'
```

```
show(T.witness_cache)
```

```
'([a], [0.5])'
```

Rejudging an object for a type

```
show(T.judge('a',.6))
```

```
'0.6'
```

```
show(T.witness_cache)
```

```
'([a], [0.6])'
```

Choicepoint: No history is kept for judgements of a particular object for a particular type.

Probability intervals

Actually, what is stored is a probability interval.

```
show(T.judge('a',.6,1))
```

```
'>=0.6'
```

`judge(a,n,n)` is the same as `judge(a,n)`
`judge(a)` is the same as `judge(a,1)` (which is the same as `judge(a,1,1)`)

Non-specific judging

Making a judgement concerning the probability that there is something of a given type.

```
T_new = Type()
T_new.judge_nonspec(.3,.4)
show(T_new.prob_nonspec)
```

```
'>=0.3&<=0.4'
```

Outline

Querying unconditional probabilities

```
show(T.judge('a'))
```

```
'1.0'
```

```
show(T.query('a'))
```

```
'1.0'
```

`'b'` is not in the witness cache. `[0,1]` ("Don't know") is returned

```
show(T.query('b'))
```

```
'<=1.0'
```

Choicepoint: "Don't know" results are not added to the witness cache.

Witness conditions

Objects can be queried for their probability of belonging to a type even if they are not represented in the witness cache by using a *witness condition*. Witness conditions are functions which map an object to a probability interval.

```
def RealClassifier(n):
    if isinstance(n,float):
        return PConstraint(1)
    else:
        return PConstraint(0)
Real = Type('Real')
Real.learn_witness_condition(RealClassifier)
show(Real.query(.6))
```

```
'1.0'
```

Querying can add to the witness cache

```
show(Real.query('a'))
```

```
'0.0'
```

```
show(Real.witness_cache)
```

```
'([0.6, a], [1.0, 0.0])'
```

cf. memoizing in logic programming

More than one witness condition for a type

- In non-probabilistic `pyttr` an object was a witness for a type if *one* of the witness conditions returned `True` for the object
- **Choicepoint:** In probabilistic `pyttr` we return the maximum of all the probability intervals returned by the witness conditions
- **Choicepoint:** . . . where the maximum is defined as the maximum of all the minima and the maximum of all the maxima
- that is, the result may not be identical with what is returned by any one of the witness conditions

```
T_at = Type('T_at')
def Classifier_a(s):
    if 'a' in s:
        return PConstraint(.8,.9)
    else:
        return PConstraint(.1,.3)
def Classifier_t(s):
    if 't' in s:
        return PConstraint(.2,.95)
    else:
        return PConstraint(.15,.7)
T_at.learn_witness_condition(Classifier_a)
T_at.learn_witness_condition(Classifier_t)
```

```
show(T_at.query('a'))
```

```
>=0.8&<=0.9'
```

```
show(T_at.query('t'))
```

```
'>=0.2&<=0.95'
```

```
show(T_at.query('at'))
```

```
'>=0.8&<=0.95'
```

```
show(T_at.query('b'))
```

```
'>=0.15&<=0.7'
```

Outline

Adding conditions to queries

- Conditions provided as second argument to `query()`
- `T.query(a,[(b,T1),T2])` — queries the probability that `a` is of type `T` given that `b` is of type `T1` and there is some witness for `T2`
- if the probability that `a` is of type `T` does not depend on any of the conditions, then `T.query(a)` is returned
- default assumption that probabilities are independent

Example with no dependence on conditions

```
T1 = Type()
T2 = Type()
T1.judge('a',.6)
show(T1.query('a',[('b',T2)]))
```

```
'0.6'
```

Probabilistic dependence created by subtyping

- Suppose $T_2 \sqsubseteq T_1$
- Then $p(a : T_1 \mid a : T_2) = 1$
- Identity is a case of subtyping: $p(a : T \mid a : T) = 1$

Example: identity case

```
show(T1.query('a',[('a',T1)]))
```

```
'1.0'
```

Example: other subtyping

```
T3 = Type()
T1.learn_witness_condition(lambda x: T3.query(x))
T3.subtype_of(T1)
```

```
True
```

```
show(T1.query('b',[('b',T3)]))
```

```
'1.0'
```

Dependent probabilities and oracles

- dependent probabilities not related to subtyping provided by an *oracle*
- . . . given as a third argument to `query()`
- oracle — a Python function which takes an object, a type and a list of conditions and returns either a probability interval or `None`
- an oracle may call on arbitrary resources external to `pyttr`, e.g. conditional probability tables, Bayesian networks
- **Choicepoint:** `T.query(a,c,o)` returns
  - `T.query(a,c)` if the subtyping condition is met
  - else: `o(a,T,c)` if this returns a probability interval
  - else: `T.query(a)`, i.e. the unconditional probability

A silly oracle

```
def SillyOracle(a,T,c):
    if a is 'a'and T is T1 and ('b',T2) in c:
        return PConstraint(.7,.8)
    else:
        return
```

Using the oracle

```
show(T1.query('a',[('b',T2)],SillyOracle))
```

```
'>=0.7&<=0.8'
```

The oracle is not defined and the result is the unconditional probability

```
show(T1.query('a',[T2],SillyOracle))
```

```
'0.6'
```

The oracle is defined but is ignored because of the subtyping condition

```
show(T1.query('a',[('b',T2),('a',T3)],SillyOracle))
```

```
'1.0'
```

Outline

Computing non-specific probability estimates

- `T.query_nonspec()` asks for the probability that there is something of type `T`, i.e. $p(T)$. The computation of this is a **Choicepoint**.
- if something has been judged to be of type `T` with probability 1, then the non-specific probability is 1.
- if there is nothing in the witness cache, but a judgement has been made with `judge_nonspec()`, then the result of that judgement is returned
- if no non-specific judgement has been made, then the disjunctive probability of the probabilities in the witness cache is returned
- if a non-specific judgement has been made and there is a non-empty witness cache, then the maximum of the non-specific judgement and the disjunctive probability of the probabilities in the witness cache is returned

Outline

Judging something to be of a meet type with probability 1

If we judge that the probability of a being of type `MeetType(T1,T2)` is 1, then we also judge the probability of a being of `T1` and the probability of a being of `T2` to be 1.

```
Tleft = Type()
Tright = Type()
Tm = MeetType(Tleft,Tright)
Tm.judge('a')
print(show(Tleft.query('a')))
print(show(Tright.query('a')))
```

```
1.0
1.0
```

Judging something to be of a meet type with probability other than 1

Choicepoint: Otherwise, we do not currently draw any conclusions about the probabilities for the component types.

```
Tleft1 = Type()
Tright1 = Type()
Tm1 = MeetType(Tleft1,Tright1)
Tm1.judge('a',.6,.8)
print(show(Tleft1.query('a')))
print(show(Tright1.query('a')))
print(show(Tm1.query('a')))
```

```
<=1.0
<=1.0
>=0.6&<=0.8
```

Non-specific judgements for meet types

```
Tleft2 = Type()
Tright2 = Type()
Tm2 = MeetType(Tleft2,Tright2)
Tm2.judge_nonspec()
print(show(Tleft2.query_nonspec()))
print(show(Tright2.query_nonspec()))
```

```
1.0
1.0
```

```
Tleft3 = Type()
Tright3 = Type()
Tm3 = MeetType(Tleft3,Tright3)
Tm3.judge_nonspec(.6,.8)
print(show(Tleft3.query_nonspec()))
print(show(Tright3.query_nonspec()))
print(show(Tm3.query_nonspec()))
```

```
<=1.0
<=1.0
>=0.6&<=0.8
```

Querying a meet type with an empty witness cache

If an object is not in the witness cache of a meet type then the conjunctive probability of the values returned for the two components is returned by the query() method.

```
Tleft3.judge('a',.6)
print(show(Tright3.query('a')))
show(Tm3.query('a'))
```

```
<=1.0
'<=0.6'
```

Forgetting

If an object is in the witness cache then the probability stored there will be returned, even though there may be conflicting evidence in the two components. In order to get the new value we need to `forget()` (*cf. retract* in logic programming).

```
Tright3.judge('a',.3)
print(show(Tm3.query('a')))
Tm3.forget('a')
print(show(Tm3.query('a')))
```

```
<=0.6
0.18
```

Meet types and oracles

The computation of conjunctive probability uses an adaptation of the Kolmogorov formula for conjunction: $p(a : T_1 \wedge T_2) = p(a : T_1)p(a : T_2 \mid a : T_1)$ following Cooper *et al.* (2015). Therefore an oracle can make a difference when querying a meet type.

```
def Oracle1(a,T,c):
    if a is'a'and T is Tright3 and ('a',Tleft3) in c:
        return PConstraint(.7,.8)
def Oracle2(a,T,c):
    if a is'a'and T is Tright3 and ('a',Tleft3) in c:
        return PConstraint(0)
Tm3.forget('a')
print(show(Tm3.query('a',oracle=Oracle1)))
Tm3.forget('a')
print(show(Tm3.query('a',oracle=Oracle2)))
```

```
>=0.42&<=0.48
0.0
```

Conditional probabilities for meet types

```
print(show(Tm3.query('a',[('a',Tleft3),('a',Tright3)])))
```

```
1.0
```

Passing arguments into witness conditions

- witness condition for meet types (schematically):

```
lambda a,c,oracle: ConjProb([(a,<left>),(a,<right>)],c,oracle)
```

- meet type (like other "logical types") cannot learn witness conditions
- witness conditions in `probttr` have 1–3 arguments
  - object queried
  - object queried, conditions
  - object queried, conditions, oracle

Join types

Treated similarly to meet types.

```
Tleftd = Type()
Trightd = Type()
Tmd = JoinType(Tleftd,Trightd)
Tmd.judge('a',0)
print(show(Tleftd.query('a')))
print(show(Trightd.query('a')))
```

```
0.0
0.0
```

Outline

Record types: basic strategy

- $p(r : T)$ where r is a record and T a record type, is the conjunctive probability that the objects in the fields or r are of the types in the correspondingly labelled types of T (Cooper *et al.*, 2015)
- if there is a label in T which is not in r, then $p(r : T) = 0$
- if r is not a record then $p(r : T) = 0$

A non-dependent record type

```
Tf1 = Type()
Tf2 = Type()
Tr1 = RecType({'l1':Tf1,'l2':Tf2})
Tf1.judge('a',.3)
Tf2.judge('b',.2)
r1 = Rec({'l1':'a','l2':'b'})
print(show(Tr1.query(r1)))
```

```
0.06
```

A dependent record type

```
dog = Pred('dog',[Ind])
a_dog = RecType({'x':Ind,
                 'e':(Fun('v', Ind, PType(dog,['v'])),['x'])})
show_latex(a_dog)
```

$$\begin{bmatrix} \mathrm{x} & : & \mathit{Ind} \\ \mathrm{e} & : & \langle \lambda v : \mathit{Ind}\ .\ \mathrm{dog}(v), \langle \mathrm{x} \rangle \rangle \end{bmatrix}$$

```
Ind.judge('d')
PType(dog,['d']).judge('s1',.7)
r2 = Rec({'x':'d','e':'s1','z':'other_stuff'})
show(a_dog.query(r2))
```

```
'0.7'
```

A slightly more complex record type

```
bark = Pred('bark',[Ind])
a_dog_bark = RecType({'x':a_dog,
                      'e':(Fun('v',Ind,PType(bark,['v'])),
                           ['x.x'])})
show_latex(a_dog_bark)
```

$$\left[\begin{array}{lll} \mathrm{x} & : & \left[\begin{array}{lll} \mathrm{x} & : & \mathit{Ind} \\ \mathrm{e} & : & \langle \lambda v : \mathit{Ind}\ .\ \mathrm{dog}(v), \langle \mathrm{x} \rangle\rangle \end{array}\right] \\ \mathrm{e} & : & \langle \lambda v : \mathit{Ind}\ .\ \mathrm{bark}(v), \langle \mathrm{x.x} \rangle\rangle \end{array}\right]$$

```
PType(bark,['d']).judge('s2',.3)
r4 = Rec({'x':r2,'e':'s2'})
show_latex(r4)
```

$$\left[\begin{array}{lll} \mathsf{x} & = & \left[\begin{array}{lll} \mathsf{x} & = & \mathsf{d} \\ \mathsf{e} & = & \mathsf{s}_1 \\ \mathsf{z} & = & \mathsf{other}_{\mathsf{stuff}} \end{array}\right] \\ \mathsf{e} & = & \mathsf{s}_2 \end{array}\right]$$

```
show(a_dog_bark.query(r4))
```

```
'0.21'
```

Conditional probabilites for record types

```
show(a_dog_bark.query(r4,[('s2',PType(bark,['d']))]))
```

```
'0.7'
```

```
show(a_dog_bark.query(r4,[('s1',PType(dog,['d'])),
                          ('s2',PType(bark,['d']))]))
```

```
'1.0'
```

Conclusion

- a language for talking about probabilistic type judgements
- in a dynamic setting where
  - new types can be created
  - witness conditions can be learned
  - probability estimates can change over time
- the type theoretical apparatus can interact with external modules
  - for the definition of witness conditions
  - for the association of oracles with conditional probability judgements
- in the hope that this might at some point contribute to the integration of rule-based and statistical approaches to computational linguistics

Bibliography I

References to work on TTR are available on `https://sites.google.com/site/typetheorywithrecords`

Cooper, Robin (2012) Type Theory and Semantics in Flux, in R. Kempson, N. Asher and T. Fernando (eds.), *Handbook of the Philosophy of Science*, Vol. 14: Philosophy of Linguistics, pp. 271–323, Elsevier BV. General editors: Dov M. Gabbay, Paul Thagard and John Woods.

Cooper, Robin (in prep) From perception to communication: An analysis of meaning and action using a theory of types with records (TTR). Draft available from `https://sites.google.com/site/typetheorywithrecords/drafts`.

Bibliography II

Cooper, Robin, Simon Dobnik, Shalom Lappin and Staffan Larsson (2015) Probabilistic Type Theory and Natural Language Semantics, *Linguistic Issues in Language Technology*, Vol. 10, No. 4, pp. 1–45.

Cooper, Robin and Jonathan Ginzburg (2015) Type Theory with Records for Natural Language Semantics, in S. Lappin and C. Fox (eds.), *The Handbook of Contemporary Semantic Theory*, second edition, pp. 375–407, Wiley-Blackwell.

Matsson, Arild (2018) Implementing perceptual semantics in Type Theory with Records (TTR). Master's thesis, University of Gothenburg. Masters in Language Technology.

Bibliography III

Utescher, Ronja (2019) Visual TTR - Modelling Visual Question Answering in Type Theory with Records, in *Proceedings of the 13th International Conference on Computational Semantics - Student Papers*, pp. 9–14, Association for Computational Linguistics, Gothenburg, Sweden.