Report for Type Theory and Natural Language Semantics

Warrick Scott Macmillan

1 Introduction

Since Richard Montague's seminal work investigating the Natural Language (NL) semantics of quantifiers via typed, intentional higher order logic [14], there have been many subsequent iterations. These include:

- How to interface various syntactic grammar formalisms with semantic theories
- How to leverage different logics and type theories to model natural lanaguage semantics
- How to create a systems that one can use to empirically test semantic theories on real data

Montague, a student of Alfred Tarski, was working in the *model-theoretic* tradition of logic. The *proof-theoretic* tradition of logic, beginning with Gentzen [5] and continued by Pragwitz [15], led to the critical developments of Per Martin-Löf's investigations of a constructive foundations of mathematics [12] [13]. Martin-Löf Type Theory (MLTT) was applied to natural language semantics after a discussion between Per and Göran Sundholm about the infamously tricky *donkey anaphora* [18].

Soon thereafter Martin-Löf's student, Aarne Ranta, developed the full theory which applied MLTT to understand natural language semantics in a proof theoretic tradition, very much inspired by but also divergent from Montague [16]. While Ranta's research focus shifted largely from semantics to syntax via his occupation with developing the programming language Grammatical Framework (GF) [17], his original semantic work greatly influenced both linguists and computer scientists. Zhaohui Luo, a type theorist whose early work was an iteration of MLTT [9] , was one of Ranta's primary successors in this endeavor, and along with linguists like Stergios Chatzikyriakidis, there has been much interest elaborating and expanding Ranta's original seed [8] [2]. It has been articulated that both the proof and model theoretic approaches to logic cohere in Modern Type Theories (MTTs) and their application in NL semantics [10].

A central idea in type theoretic semantics in contrast to the Montagovian tradition is the "common nouns as types" maxim [8], whereby the common nouns are actually a universe, instead of functions in the classical logic setting. This not only fits a seemingly more natural intuition where the words denote objects, but also makes it convenient for creating an elaborate subtyping mechanism, namely coercive subtyping as developed by Luo [7] [11].

Despite the obstacles that subtyping presents by disallowing uniqueness of typing, the coercive subtyping approach allows one to retain nice meta-properties about the type theory like canonicity. One may construct an ontological hierarchy that captures semantic nuance and facilitates computation. Computation using coercive types is just one of the many benefits one can leverage from this MTT approach to linguistic semantics.

Proof assistants like Coq and Agda are implementations of different dependent type theories. They allow one to interactively build proofs, or programs, which are implemented according to their specified behaviors, or types. Because the types are identified with theorems by way of the propositions-as-types paradigm, the proof assistants are capable of doing functional programming, advanced program verification, and constructive mathematics. It is possible to shallowly embed semantic encodings from both the Montagovian and MTT traditions in these proof assistants, as was done in Coq [3] [1].

The dependent function type is core to the type theories, and it is possible to prove implications by constructing functions. One can therefore do inference about the semantic encodings. Inference is one important way of empirically testing or observing a semantic theories' success. The FraCas test suite [4] was designed to capture the inferability of various semantic phenomena in a suite of 346 question, each of which has at least one premise from which a native speaker would be able to affirm, deny, or defer the question if an answer is not knowable under the assumptions.

2 A Montagovian Example

We here showcase an example similar to the FraCas test suite to demonstrate the way in which one does inference with Agda. This takes place after having interpreted the syntax and constructed semantic formulas in Montague's logic, which we implement in dependent type theory. The pseudo-FraCas example is as follows :

```
Premise    : Every man loves a woman.
Question 1 : Does John love a woman?
Answer 1 : Yes.
Question 2 : Does some man love a woman?
Answer 2   : Yes.
```

2.0.1 Montague Semantics

We can, given a means of constructing trees out of basic syntactic categories, assign types to the categories and functions to the rules which obey the rules' signatures. One can decide, based on some GF abstract syntax, how these GF functions evaluate to formulas in the logic. This then allows one to derive meaningful sentences based off the abstract syntax trees, by normalizing the lambda terms. The failure of grammatical terms to normalize (enough) is a sign of semantic incoherence. This can either be the result of improper typing assignments for given lexical categories or a failure to give "proper" lambda terms to given rules or lexical constants. It is also a goal of the theory to only admit semantically reasonable ideas, i.e. that there aren't superfluous, meaningless sentences admitted.

While FraCoq used trees generated straight from the GF Resource Grammar Library (RGL) [6], we choose here a simpler syntax taken from [19]. Our implementation differs from Fracoq in a few ways. First, we choose different type assignments for the grammatical categories. Additionally, we use Agda instead of Coq.

In the Montagovian tradition one uses a simple type theory. There are two basic types, entities and formulas, denoted e and t, respectively. Everything else is constructed out of higher order functions ending in t. The entities are meant to denote some objective thing in the world, like John, whereas the formulas, only occurring in the codomain of a function, may represent utterances such as "John walks".

Given a suite with nouns (N), verbs (V), noun phrases (NP), verb phrases (VP), determiners (Det) and sentences (S) we can then give a Montagovian interpretation to accommodate the above FraCas-like example. We assign the grammatical categories, and functions over those categories, as an abstract syntax in GF:

```
cat
  S ; N ; NP ; V; VP ; V ; Det ;
fun
  sentence : NP -> VP -> S ;
  verbp : V -> NP -> VP ;
  ...
```

As was done with Coq in [1], we can embed such a grammar in Agda. First, we assign GF categories to Agda sets.

$$[\![\_]\!] : Cat_{GF} \to Set_{Agda}$$

For the functions in the abstract syntax, we simply map the interpretation functorially with respect to the arrows.

$$\begin{aligned} [\![\_]\!] : fun_{Cat} &\longrightarrow fun_{Agda} \\ A \to B &\mapsto [\![A]\!] \to [\![B]\!] \end{aligned}$$

We then know that the interpretation a given GF function $f{:}X$, must be well typed with the Agda semantics, i.e. $[\![f]\!]{:}[\![X]\!]$. Finally, the way one constructs ASTs by plugging in functions (or leaves) with the correct return type into a given function (or node), and witnessing an evaluation as the application of the function at a node to its leaves in successive order. These become Agda function applications:

$$[\![f(g)]\!] \to [\![f]\!]([\![g]\!])$$

We assign Agda types to the categories, knowing that entities are postulated as a type.

```
t = Set
S = t
NP = (e → t) → t
VP = NP → t
V = NP → NP → t
Det = (e → t) → (e → t) → t
N = e → t
```

Agda's equality symbol indicates our the semantic interpretation, $[\![\_]\!]$, of our grammar's categories, whereby we are here working with a shallow embedding. One could instead elect to define both the GF embedding as a record and Montague's intentional type theory as an inductive dataype, whereby the semantics could then be given explicitly. This degree of formality is not necessary for our simple examples, but doing so would allow one to prove meta-properties about the GF embedding, and should certainly be investigated. The two GF functions for forming sentences and verb phrases can then be given the following Agda interpretations.

```
sentence : NP → VP → S
sentence S V = V S

verbp : V → NP → VP
verbp V O S = V S O
```

We axiomatically include primitive lexical items of `love`, an entity `j` for John, and `man` and `woman` as nouns, so that we can articulate logically interesting facts about them. As our encoding of noun phrases takes an argument, we can define the syntactic notion `John` by applying an argument function to `j`.

```
postulate
  love : e → e → t
  j : e
  man : N
  woman : N
  johnMan : man j

John : NP
John P = P j
```

We define our quantifiers using the Agda's dependent Π and Σ which make up the core of any dependently typed language.

```
Every : Det
Every P Q = (x : e) → P x → Q x

A : Det
A cn vp = Σ[ x ∈ e ] (cn x × vp x)
```

We also notice that we may define the ditransitive verb `loves` two ways, which are obtained commuting the NP arguments.

```
loves loves' : V
loves O S = O (λ x → S λ y → love x y) -- (i)
loves' O S = S (λ x → O λ y → love x y) -- (ii)
```

We can then observe two different semantic interpretations of the phrase "every man loves a woman".

```
everyManLovesAWoman = sentence (Every man) (verbp loves (A woman)) -- (i)
everyManLovesAWoman' = sentence (Every man) (verbp loves' (A woman)) -- (ii)
```

The two interpretations of "love" which differ as to where the respective arguments are applied in the program, manifest in a semantic ambiguity of whether there is one or possibly many women are in the context of consideration. With Agda's help in normalizing these two types, this ambiguity is reflected in whether the outermost quantifier is universal or existential. Note the product type has been desugared to a non-dependent Σ-type.

```
(i) = (x : e) → man x → Σ e (λ x  → Σ (woman x ) (λ x  → love x x ))
(ii) = Σ e (λ x → Σ (woman x) (λ x  → (x  : e) → man x  → love x x ))
```

Let's articulate natural language inference to a hypothetical mathematician.

Theorem: If every man loves a woman, then John loves a woman.

Proof: Assume that John is a person who is human. Knowing that every man loves a woman, we can apply this knowledge to John, who is a man, and identity a woman that John loves. Specifically, this woman John loves is a person, evidence that person is a woman, and evidence that John loves that person. QED

We also show this argument in Agda, noting that we comment out the more verbose but informative information about who the woman John loves really is. We explicitly include the data of the proof commented out so-as to see how the overly elaborate natural language argument manifests in code.

```
-- (i)
jlaw : everyManLovesAWoman → johnLovesAWoman
jlaw emlaw = womanJonLoves -- thePerson , thePersonIsWoman , jonLovesThePerson
  where
    womanJonLoves : Σ e (λ z → Σ (woman z) (λ _ → love j z))
    womanJonLoves = emlaw j johnMan
    -- thePerson : e
    -- thePerson = proj womanJonLoves
    -- thePersonIsWoman : woman (thePerson)
    -- thePersonIsWoman = proj (proj womanJonLoves)
    -- jonLovesThePerson : love j thePerson
    -- jonLovesThePerson = proj (proj womanJonLoves)
```

We note this is the first interpretation of `loves`. In the alternative presentation we see that if there is a woman every man loves, that our semantic theory interprets her as a person, the evidence they are a woman, and the evidence that every man loves them. Since every man loves them, John certainly does as well.

```
-- (ii)
jlaw' : everyManLovesAWoman' → johnLovesAWoman'
jlaw' (person , personIsWoman , everyManLovesPerson ) =
  person , personIsWoman , everyManLovesPerson j johnMan
```

Finally, we can know that some man loves a woman by virtue of the fact that John loves a woman and John is a man. We note that the proofs `smlw` and `smlw'` just articulate the data in different order.

```
smlw : johnLovesAWoman → someManLovesAWoman
smlw (w , wWoman , jlovesw ) = j , johnMan , w , wWoman , jlovesw
smlw' : johnLovesAWoman' → someManLovesAWoman'
smlw' (w , wWoman , jlovesw ) = w , wWoman , j , johnMan , jlovesw
```

While the Montagovian approach is historically interesting and still incredibly significant in linguistic semantics, the interpretation of various parts of speech and their means of syntactic combination doesn't always seem to be intuitively reflected in the types. Dependent type theories, or MTTs, which have a more expressive type system, gives us a means of more intuitively encoding the meanings.

This use of a "fancier" type theory for NL semantics can be viewed as analogous to a preference of inductively defined numbers over Von Neumman or Chruch encodings. While the latter constructions are interesting and important historically, they aren't easy to work with and don't match our intuition. Nonetheless, the different encodings of numbers in different foundations can be proven sound and complete with respect to each other, so that one can rest assured that the intuitive notion of number is captured by all of them.

Unlike in formal mathematics, however, to capturing semantics in different foundational theories suggests no way of proving soundness or completeness with respect to the interpretation of different phrases, as the set of semantically fluent utterances are not inductively defined. One may instead take a pragmatic approach and see the difficulties arising when doing inference with respect to different type theories.

3 An MTT Example

We now follow the dependently typed approach to linguistic semantics initiated by Ranta, and do inference on actual FraCas examples in Agda.

Initially, one takes the common nouns as types maxim literally, by saying that the type of common nouns is actually just a universe, which simply gives the universe an alias of `CN` in Agda, $[\![CN]\!] := Set$. Man is common noun, so semantically we just say $[\![Man]\!] : [\![CN]\!]$. And if there is a man John, we simply assert $[\![John]\!] : [\![Man]\!]$.

```
CN = Set

postulate
  man : CN
  john : man
```

In Agda, there is only one sort of predicative universe, `Set`. In Coq there are both impredicative and predicative universes, `Prop` and `Set` respectively, of which `Type` is a superclass. While one defines `CN := Set` in Coq, the type of impredicative propositions are included in both [1] and [3] which is not possible in Agda. It should be possible to make everything predicative in Coq, but the authors' reasons for using impredicativity were not explicated in their work as far as we know. Agda's notion of proposition, `Prop`, is by default proof irrelevant, whereas one must choose to make Coq's propositions proof irrelevant. We don't explore more about the implications of foundations here.

Once one has a the universe of common nouns, each of which may have many inhabitants, we can ask how they are modified. Intransative Verbs (IVs) like "walk", can be seen as a type restricted by the collection of things which have the ability to walk, such as animals. We can see such verbs as functions taking a specific type of common noun to an arbitrary type : $[\![IV]\!] : ([\![x]\!] : [\![CN]\!]) \rightarrow Set$

```
postulate
  walk : animal -> Set
```

We can then encode the quantifiers, noting that they also return just types. The dependent type `P` below is propositional in Coq. These are more arguably more syntactically pleasing than our Mongtagovian semantics, because they only bind a noun and a property about that noun, rather than rigidly requiring a verb phrase and a noun phrase as arguments.

```
some : (A : CN) → (P : A → Set) → Set
some A P = Σ[ x ∈ A ] P x

all : (A : CN) → (P : A → Set) → Set
all A P = (x : A) → P x
```

Wanting to do inference with these encodings, we hope to show that if John is a man and every man walks, then John walks. The difficulty is that `walk` is a type over animals, not men, and the relation between men and animals are not yet covered by our type theory. The theory of coercive subtyping rectifies this and gives a mechanism of implicity coercing the type of men to the type of animals, as indeed all men are animals. One can form an order from the subtypes, with possible infima and suprema, that may transform some abstract ontological model of the domain into specific ways of using the ontological information to prove inferences.

The coercions in coercive type theory can be approximated by the use of Agda's instance arguments, which are indicated with `{{_}}` below. Nonetheless, Agda doesn't support coercive subtyping as developed by Luo, and therefore has weaknesses relative to Coq when it comes to eliminating "coercion bureaucracy". A coercion is a record type parameterized by two types x and y with one field `coe` which is merely a mapping from x to y. We can then compose and apply functions with arguments for which there exists an coercion.

```
record Coercion {a} (x y : Set a) : Set a where
  constructor ⌞_⌟
  field coe : x → y


_◎_ :   {a} {A B C : Set a} → Coercion A B → Coercion B C → Coercion A C
_◎_ c d = ⌞ (λ x → coe d (coe c x)) ⌟

_$_ :   {a b} {A A  : Set a} {B : Set b} → (A → B) →
        {{c : Coercion A  A }} → A  → B
_$_ f {{c}} a = f (coe c a)

postulate
  ha : human → animal
  mh : man → human
```

The instance arguments, similar to Haskell's type-classes, allow one to introduce the coercion information into a context so that one may compute with these hidden typing relations.

```
instance
  hac = ⌞ ha ⌟
  mhc = ⌞ mh ⌟
  mac = mhc ⊚ hac
```

Once one has defined instances, Agda can infer that `walk` is a property of men, which should be subtypes of animals. We must explicitly declare this in Agda, unfortunately. A type theory with native support for coercive subtyping would save significant hassle, although someone with significant experience using Agda's instance arguments might find a superior way to do this rather than generating all the instances and coercion applications, possibly without resorting to metaprogramming. However, once we have the infrastructure in place, we can not only infer basic facts about men, but also about animals and their relation to men.

```
johnwalk = manwalk john
allmanwalk = all man manwalk
somemanwalk = some man manwalk

thm1 : allmanwalk → johnwalk
thm1  mWalk[m] =  mWalk[m] john

thm2 : johnwalk → somemanwalk
thm2 jw = john , jw

thm3 : somemanwalk → some animal walk
thm3 (m , walk[m]) = ha (mh m) , walk[m]
```

To the best of our knowledge, there is no way of coercing types directly, as in, one cannot simply force the type-checker in `thm3` to accept the man argument `m` without explicitly requiring the programmer to insert the coercions, `ha (mh m)`. Another issue is that `manwalk` and `walk` are explicitly different types, despite the instances allowing Agda to coerce the fact that a man walks, `walk[m]`, to a fact about an animal walking. We may reconcile this with more instance arguments, whereby we create a parameterized record `Walks` with a single data point for the walking capacity. One can then overload walks with all the different entities which can walk, and thereby not have the ugly `manwalks` in the type signature of `thm3'`.

```
record Walks {a} (A : Set a) : Set a where
  field
    walks : A → Set

open Walks {{...}} public

postulate
  animalsWalk : Walks animal

instance
  animalwalks : Walks animal
  animalwalks = animalsWalk

  humanwalks : Walks human
  humanwalks = record { walks = λ h → Walks.walks animalsWalk $ h}

  manwalks : Walks man
  manwalks = record { walks = λ m → Walks.walks animalsWalk $ m}

thm3' : some man walks → some human walks
thm3' (m , walk[m]) = mh m , walk[m]
```

3.1 Irish Delegate Example

We elaborate the following FraCas example, which includes the ditransitive verb "finished", the adjective "Irish", and adverb "on time", and the determiner "the". We include a common noun for `object`, of which `survey` and `animal` should be subtypes.

```
Premise  : Some Irish delegates finished the survey on time.
Question : Did any delegates finish the survey on time?
Answer   : Yes.
```

Semantically, Ditransitive Verbs (DVs) are similar to IVs, they are just binary functions instead of unary.

$$[\![DV]\!] : ([\![x]\!] : [\![CN]\!]) \to ([\![y]\!] : [\![CN]\!]) \to Set$$

The quality of being on time, which modifies a verb, is interpreted as a function which takes a common noun cn, a type indexed by cn (the verb), and returns a type which is itself dependent on cn. The intuition that one can continue to modify a verb phrase with more adverbs is immediately obvious based of the type signature, because it returns the same type as a verb after taking a verb as an arguement.

$$[\![ADV]\!] : (\Pi\ x : [\![CN]\!]) \to ([\![x]\!] \ \to Set) \to ([\![x]\!] \ \to Set)$$

The determiner "the" is simply a way of extracting a member from a given CN.

$$[\![the]\!] : (\Pi\ x : [\![CN]\!]) \to x)$$

Finally, the MTT interpretation of adjectives is definitionally equal to IVs, $[\![ADJ]\!] : ([\![x]\!] : [\![CN]\!]) \to Set$. This does not mean they are semantically at all similar. Verbs describe what an individual does, whereas adjectives describe some property of the individual. To apply an adjective a to a member n of some CN gives a sentence whose meaning is "a is n", whereby the syntactic "is" is implicit in the semantics.

```
postulate
  finish : object → human → Set
  ontime : (A : CN) → (A → Set) → (A → Set)
  the : (A : CN) → A
  irish : object → Set
```

Adjectives are generally not meant to return sentences, but other common nouns. Therefore, we can leverage the dependent product type or records more generally to describe modified common nouns, whereby the first element c is a member of some CN and the second member is a proof that c has the property the adjective expresses. We can therefore see the example of `irishdelegate` as such in Agda:

```
record irishdelegate : CN where
  constructor
    mkIrishdelegate
  field
    c : delegate
    ic : irish $ c
```

We can follow the same methodology as before, coercing Irish delegates to delegates axiomatically, and then applying the semantic interpretations of the words such that the types align correctly. This follows from an intuitive, albeit primitive, syntactic presentation of the sentences.

```
finishTheSurvey : human → Set
finishTheSurvey = finish $ (the survey)

finishedTheSurveyOnTime : delegate → Set
finishedTheSurveyOnTime x = ontime human finishTheSurvey $ x

someDelegateFinishedTheSurveyOnTime : Set
someDelegateFinishedTheSurveyOnTime = some delegate finishedTheSurveyOnTime
```

Once one builds a parallel infrastructure for `irishdelegate`, one can then proceed with the inference. We note that the work has to be doubled because `finishedTheSurveyOnTime` and `someDelegateFinishedTheSurveyOnTime` need to be refactored, renaming `delegate` to `irishdelegate`. Again, this inference is just the identity function modulo an explicit `idd` coercion, and implicit coercions allowing `finishedOnTime` to be cast to its most general formulation where it is parameterized `human`.

```
fc55 :
  someIrishDelegateFinishedTheSurveyOnTime → someDelegateFinishedTheSurveyOnTime
fc55 (irishDelegate , finishedOnTime) = (idd irishDelegate) , finishedOnTime
```

We note that one could have instead included an extensionality clause for adjectives and adverbs, whereby one gives additional information so that the argument and return types, dependent on some noun A, behave coherently with respect to arbitrary arguements of A. One can then derive the adverb by forgetting the extensionality clause. The inference works out the same.

```
postulate
  ADV : (A : CN) (v : A → Set) → Σ[ p ∈ (A → Set) ] ((x : A) → p x → v x)

on_time : (A : CN) (v : A → Set) → A → Set
on_time A v = proj (ADV A v)


fc55' :
  2someIrishDelegateFinishedTheSurveyOnTime → 2someDelegateFinishedTheSurveyOnTime
fc55' (irishDelegate , finishedOnTime) = (idd irishDelegate) , finishedOnTime
```

We now investigate the possiblity of gereralizing a notion of Irishness, as well as integrating the adjectival semantics with our previous work generating instance arguements for “walks”.

Unlike walking, which was assumed to apply to all animals, being Irish is a restriction on the set of objects of some given domain. Therefore we can’t just define the record parametrically for all common nouns, but rather must include an instance argument for the coercion. Note this would break the semantic model if we were to include the type of common noun “Swede” with a coercion to humans, because one would be able to make an Irish Swede. Such a notion may or may not be semantically feasible depending on one’s interpretation.

```
record irishThing (A : CN) {{c : Coercion A object}} : CN where
  constructor
    mkIrish
  field
    thing : A
    isIrish : irish $ thing
```

Once can now declare Irish entities using the record for humans, delegates, and animals, where one can include the coercion arguments explicitly, even though they are inferrable. Thereafter, we can overload walks even more. Although a lot of this code is boilerplate, the instance declarations must be nullary, and basic code generation techniques would be needed to scale this to a larger corpus. The point is, once we know that animals walk, anything subsumed under that category is straightforward to make "walkable".

```
IrishDelegate = irishThing delegate {{doc}}
IrishHuman = irishThing human {{hoc}}
IrishAnimal = irishThing animal {{aoc}}

instance
  irishAnimalWalks : Walks IrishAnimal
  irishAnimalWalks = record { walks = helper }
    where
      helper : irishThing animal → Set
      helper (mkIrish a isIrish ) = Walks.walks animalsWalk a

  irishHumanWalks : Walks IrishHuman
  irishHumanWalks = record { walks = helper }
    where
      helper : irishThing human → Set
      helper (mkIrish a isIrish ) = Walks.walks animalsWalk $ a
```

We can now prove analagous theorems to what we showed earlier, with the adjectival modification showing as extra data in both the input and output. One can always forsake the Irish detail and prove a weaker conclusion, as in `thm5`.

```
thm4 : some IrishHuman walks → some IrishAnimal walks
thm4 (mkIrish hum isIrish[hum] , snd) = (mkIrish (ha hum) isIrish[hum]) , snd

thm5 : some IrishHuman walks → some animal walks
thm5 (mkIrish hum isIrish[hum] , snd) = (ha hum) , snd
```

If we decide to now assume some anonymous `irishHuman` exists, and we prove that human is an animal, `irishAnimal`, we can see the fruits of our labor insofar as the identity function works in `thm6` despite the function being between different types. In `thm7` we can also then use our anonymous human as a witness for the existential claim that "some Irish animal walks".

```
postulate
  irishHuman : irishThing human {{hoc}}

instance
  irishAnimal : irishThing animal
  irishAnimal = mkIrish (ha (irishThing.thing irishHuman)) (irishThing.isIrish irishHuman)

thm6 : walks irishAnimal → walks irishHuman
thm6 = id

thm7 : walks irishHuman → some IrishAnimal walks
thm7 x =
  mkIrish (ha (irishThing.thing irishHuman)) ((irishThing.isIrish irishHuman)) , x
```

One might try to prove something even sillier : that an Irish animal is an Irish object. Problematically, for the instance checker to be happy we need to reflexively coerce an object due to the constraint that a coercion to an object must exist to build an `irishThing`.

```
postulate
  oo : object → object
instance
  ooc = ⌞ oo ⌟

postulate
  irishHumanisIrishThing : irishThing animal → irishThing object
```

If one tries to prove this though, it's impossible to complete the program.

```
irishHumanisIrishThing (mkIrish thing isIrish) = mkIrish ((ao (thing))) {!!}
```

Agda computes with the reflexive coercion instance, and therefore we come to the irredeemable goal :

```
Goal: irish $ ao thing
Have: irish $ thing
```

One might think to just add an extra instance to appease Agda :

```
instance
  aooc = ⌞ ao ⌟ ⊚ ooc
```

However, if we were to add an additional instance allowing an animal to be coerced to an object, this would break the necessary uniqueness of instance arguments, consistent with the uniqueness of coercions property in type theories supporting coercive subtyping. This example highlights the limitations of working with a make-believe subtyping mechanism. While instances give the Agda programmer the benefits of ad-hoc polymorphism, they are is still not a substitute for a type theory with coercive subtyping built in, especially when it comes to MTT semantics.

3.2 Addendum on Inductive Types

The above method of introducing members of a specific CN was to just axiomatically include them. One can instead define the common nouns specifically as inductive types, whereby the programmer selectively includes relevant entities as constructors. In this view coercions are functions into other inductive "supertypes".

```
data Men : CN where
  Steve Dave : Men

data Women : CN where
  Suzy : Women

data Human : CN where
  WomenHuman : Women → Human
  MenHuman : Men → Human
```

One can then define walking as a function returning a type indicating the truthiness of whether a given human walks. This gives us more granularity with which to view things, because `allmenWalk` is no longer just assumed, but proven based off the assumptions we've made about our domain.

```
Walk : Human → Set
Walk (MenHuman Steve) = ⊤
Walk (MenHuman Dave) = ⊤
```

```
  Walk (WomenHuman Suze) = ⊤

  allmenWalk : all Men λ x → Walk (MenHuman x)
  allmenWalk Steve = tt
  allmenWalk Dave = tt
```

If one is in a room where `Dave` isn't able to walk, this can be encoded in the type of `Walk'`. One can prove that some man walks by just exhibiting `Steve` as a witness, and therefore we don't need to do inferences in the same way as before, because all our information (at least for this example) is derivable based off how we built `Men`, `Human`, and `Walk`.

```
  Walk' : Human → Set
  Walk' (MenHuman Steve) = ⊤
  Walk' (MenHuman Dave) = ⊥
  Walk' (WomenHuman Suze) = ⊤

  someManWalks' : some Men λ x → Walk' (MenHuman x)
  proj₁ someManWalks' = Steve
  proj₂ someManWalks' = tt
```

We are stuck when we try to prove that all men walk, and this is by design.

```
allmenDontWalk : all Men λ x → Walk' (MenHuman x)
allmenDontWalk Steve = tt
allmenDontWalk Dave = {!!}
```

We can use our previous definition of `Walks` as before, and introduce instance arguments so that we can build similar proofs as before and omit certain coercions cluttering our code - `thm3''` is homologous to `thm3'`.

```
  instance
    humansWhoWalk : Walks Human
    Walks.walks humansWhoWalk = Walk

    Menwalks : Walks Men
    Menwalks = record { walks = helper }
      where
        helper : Men → Set
        helper x = Walks.walks humansWhoWalk (MenHuman x)

  thm3'' : some Men walks → some Human walks
  thm3'' (fst , snd) = MenHuman fst , snd
```

A potential issue arising is that we may introduce inconsistencies into our universe, if we chose to define `Menwalks` with Dave not walking.

```
  instance
    Menwalks' : Walks Men
    Menwalks' = record { walks = helper }
      where
        helper : Men → Set
        helper Steve = ⊤
        helper Dave = ⊥
```

While contradicting `humansWhoWalk`, this isn't ruled out by Agda. The programmer must exert caution when making decisions using the "inductive approach". This approach should be tested on a bigger corpus with many more syntactic and semantic phenomena to access its viability relative to the other approaches considered.

4 Conclusion

We have seen that Agda is fully capable of encoding natural language semantics both in the Montagovian and MTT traditions. While the examples explored here only showcase a tiny fraction of semantically interesting phenomena, they nonetheless suggest that many generalizations are possible which may cover a larger corpus and more phenomena.

In contrast to previous work with Coq, the lack of predicativaty didn't yield any immediate roadblocks, but this may not be the case with other examples. Additionally, Coq's support of coercions give it an advantage over Agda. The use of tactics in Coq also allows for quicker automation, as well as the possibility of scaling inference to much larger data sets. Nonetheless, Agda is extremely experimental - incorporating tactics is one of many goals of the community at large. Additionally, we have found the Agda presentation gives a much more intuitive reading of the inferences and proofs than Coq's tactics - and this is debatably superior from a pedagogical perspective even if less efficient. It would certainly be interesting to try to build a FrAgda library and explicitly compare it to FraCoq.

Using logics and type theories to understand natural language meaning is exciting and important. The use of Agda to do inference provides evidence that proof assistants are an important tool in the semanticist's tool-belt.

References

[1] Jean-Philippe Bernardy and Stergios Chatzikyriakidis. "A Type-Theoretical system for the FraCaS test suite: Grammatical Framework meets Coq". In: *IWCS 2017 - 12th International Conference on Computational Semantics - Long papers*. 2017.

[2] Stergios Chatzikyriakidis and Zhaohui Luo. "An Account of Natural Language Coordination in Type Theory with Coercive Subtyping". In: *Constraint Solving and Language Processing - 7th International Workshop, CSLP 2012, Orléans, France, September 13-14, 2012, Revised Selected Papers*. Ed. by Denys Duchier and Yannick Parmentier. Vol. 8114. Lecture Notes in Computer Science. Springer, 2012, pp. 31–51.

[3] Stergios Chatzikyriakidis and Zhaohui Luo. "Natural Language Inference in Coq". In: *J. Log. Lang. Inf.* 23.4 (2014), pp. 441–480.

[4] Robin Cooper et al. *Using the framework*. Tech. rep. 1996.

[5] G. Gentzen. "Untersuchungen über das logische Schließen I". In: *Mathematische Zeitschrift* 39 (1935), pp. 176–210.

[6] Peter Ljunglöf and Magdalena Siverbo. "A bilingual treebank for the FraCaS test suite". In: *SLTC 2012* (2012), p. 53.

[7] Zhaohui Luo. "Coercions in a polymorphic type system". In: *Math. Struct. Comput. Sci.* 18.4 (2008), pp. 729–751.

[8] Zhaohui Luo. "Common Nouns as Types". In: *Logical Aspects of Computational Linguistics - 7th International Conference, LACL 2012, Nantes, France, July 2-4, 2012. Proceedings.* Ed. by Denis Béchet and Alexander Ja. Dikovsky. Vol. 7351. Lecture Notes in Computer Science. Springer, 2012, pp. 173–185.

[9] Zhaohui Luo. *Computation and reasoning - a type theory for computer science*. Vol. 11. International series of monographs on computer science. Oxford University Press, 1994.

[10] Zhaohui Luo. "Formal Semantics in Modern Type Theories: Is It Model-Theoretic, Proof-Theoretic, or Both?" In: June 2014.

[11] Zhaohui Luo, Sergei Soloviev, and Tao Xue. "Coercive subtyping: Theory and implementation". In: *Inf. Comput.* 223 (2013), pp. 18–42.

[12] Per Martin-Löf. “Constructive Mathematics and Computer Programming”. In: *Logic, Methodology and Philosophy of Science VI*. Ed. by L. Jonathan Cohen et al. Vol. 104. Studies in Logic and the Foundations of Mathematics. Elsevier, 1982, pp. 153–175.

[13] Per Martin-Löf. *Intuitionistic type theory*. Vol. 1. Studies in Proof Theory. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980. Bibliopolis, 1984, pp. iv+91.

[14] Richard Montague. “The Proper Treatment of Quantification in Ordinary English”. In: *Approaches to Natural Language: Proceedings of the 1970 Stanford Workshop on Grammar and Semantics*. Ed. by K. J. J. Hintikka, J. M. E. Moravcsik, and P. Suppes. Dordrecht: Springer Netherlands, 1973, pp. 221–242.

[15] Dag Prawitz. *Natural deduction: A proof-theoretical study*. Courier Dover Publications, 2006.

[16] A. Ranta. *Type Theoretical Grammar*. Oxford University Press, 1994.

[17] Aarne Ranta. “Grammatical Framework”. In: *J. Funct. Program.* 14.2 (Mar. 2004), pp. 145–189.

[18] Göran Sundholm. “Proof Theory and Meaning”. In: *Handbook of Philosophical Logic: Volume III: Alternatives in Classical Logic*. Ed. by D. Gabbay and F. Guenthner. Dordrecht: Springer Netherlands, 1986, pp. 471–506.

[19] Unknown. *Course 5: Montague’s approach to semantics*. https://pdfs.semanticscholar.org/f94b/268c1c91dd1de22cf978a7ea03f8860cbe9d.pdf. 2014.