From TY_n to DRT: an implementation

Patrick Blackburn, Sébastien Hinderer

From TY_n to DRT: an implementation

Patrick Blackburn*, **Sebastien Hinderer***

*INRIA Lorraine, UHP Nancy, France
{blackbur, hinderer}@loria.fr

Abstract

Since the early 1990s, the semanticist Reinhard Muskens has advocated the use of the TY_n family of higher-order logics as a general framework for semantic representation; he has backed up his claim by treating a wide variety of semantic phenomena in the TY_n framework. Most interestingly of all, he has also shown that the central ideas of Discourse Representation Theory (DRT) can be modeled in TY_n, thereby allowing a clean Montague-style treatment of discourse level phenomena. In this paper, we assess the significance for computational linguistics of his theoretical work. We do so by implementing the TY_n-based approach to DRT in `Nessie`, a generic framework for semantic construction that is implemented in the functional programming language OCaml.

1. Introduction

Since the early 1990s, the semanticist Reinhard Muskens has advocated the use of the TY_n family of logics for semantic representation purposes; see, for example (Muskens, 1991; Muskens, 1996a). Like the original systems proposed by Richard Montague in his pioneering work on formal semantics (see (Montague, 1974)) the TY_n logics are higher-order (that is, they are built over the simply-typed lambda calculus) but Muskens argues that they improve Montague's systems in at least two respects. First, he argues that they are logically much simpler and better behaved. Second, and for our purposes more importantly, he argues that they are far more flexible. Indeed, a constant theme in Muskens' work is that TY_n offers a uniform framework for virtually any kind of semantic analysis. This is because the n in TY_n is significant: it indicates how many basic sorts of entities the logic works with. Semanticists are free to choose as many basic kinds of entities as are required for the analysis at hand, and the resulting logic handles them in a uniform way. For example, the TY_n family provides a setting in which semantic analyses which require the use of ordinary individuals, belief states, times, and situations can be cleanly captured in a single system.

As is well known, Montague's original semantic investigations were confined to the sentential and sub-sentential levels; they offered no insight into the semantics of discourse, or about such puzzling phenomena as "Donkey anaphora". These shortcoming led to the development of Discourse Representation Theory (DRT), a successful and widely-used framework for discourse level semantics; see (Kamp and Reyle, 1993). However DRT had its own weakness. As it was not based on the simply-typed lambda calculus, its semantic construction process lacked the compositionality of Montague's system. Once again, however, Muskens showed that TY_n could clarify the situation. In a classic paper (Muskens, 1996b) he showed that by letting TY_n work with such basic entities as *registers* and *states*, the key insights of DRT could be captured. That is, TY_n makes possible a clean Montague-style treatment of discourse level phenomena. Moreover, it achieves this in the same way as it handles other semantic phenomena: via a translation into classical logic. This opens up the possibility of combining the TY_n approach to DRT with the TY_n treatment of other semantic problems.

But do these theoretical observations have any relevance for computational linguistics? This is the starting point for the work reported here, which is based around a generic semantic construction system called `Nessie`. This allows the user to specify which basic semantic entities are assumed (that is, which version of TY_n is being used), and, once this has been specified, `Nessie` handles the required semantic construction uniformly. In previous work we have successfully applied `Nessie` to the semantics of aspectual expressions (by making use of a version of TY_n which includes events as one of its basic types of entity) and shown how the logical representations built by `Nessie` can be coupled with logic-based inference services. In the present paper we use `Nessie` to investigate the TY_n-based approach to DRT from a computational perspective. As we shall see, the key ideas of the Muskens approach can indeed be captured. However our computational investigations also reveals an interesting gap which requires attention.

Space restrictions make it impossible to give a detailed introduction to DRT, TY_n, or the link between the two. Accordingly, we advise the reader who wishes to know more about the underlying theory to consult (Muskens, 1996b). In what follows, we shall focus exclusively on computational issues. In particular, we shall focus on the computational architecture of `Nessie`, and on the modification required to capture DRT via TY_n in this style of architecture.

2. Introducing `Nessie`

This section gives a brief overview of `Nessie`, a generic tool developed by the authors for building semantic representations in the TY_n family of logics. We first discuss the kind of inputs `Nessie` needs and the kind of output it can produce. We then give some details about the algorithm used to perform the semantic construction task.

2.1. What `Nessie` does

The ideas at the heart of `Nessie` are those introduced in the pioneering work of Montague: semantic representations of complex expressions are assumed to be typed lambda-terms. They are computed from lambda-terms representing the semantics of words, lexical terms being the

basic ingredients. The way terms are combined to obtain the more complex semantic representations is guided by the syntactic structure of the entity the semantics has to be computed for.

Thus `Nessie` needs two inputs to compute a semantic representation: a lexicon containing words and the lambda-terms they are associated with, and a tree reflecting the syntactic structure of an expression. The leaves of this tree contain words whose representations can be found in the lexicon.

As an example, suppose one wants to build a simple Montague-style representation for the sentence "John walks". A `Nessie` lexicon for such a task might look like this:

```
type e;

family pn {
  type = e;
  pattern = lam P:e->t. (P(_))
};

lemma john : pn;

family iv {
  type = e->t;
  pattern = lam x : e. (_(x))
};

lemma walk : iv;
```

The concepts of *family* and *lemma* that appear in the lexicon have been introduced to reflect the fact that in many cases the objects that belong to a given syntactic category have similar representations that are built from a common pattern and differ from each other only in the constants they use. Thus, since JOHN is a lemma of the PN (Proper Name) family, its representation will have the form that is characteristic of this family, namely $\lambda P : e \rightarrow t.(P(\_))$, where _ is a place-holder. This place-holder will be replaced by a lemma (here john), yielding the expected representation: $\lambda P : e \rightarrow t.(P(john))$. An analogous relation holds between the lemma WALK and the family IV (Intransitive Verb) it belongs to.

Another important point should be made about the lexicon's content. Since the semantic representations we build are (as in Montague's pioneering work) *typed*, it is necessary to be able to perform type-checking. This is why the previous lexicon mentioned the `type` directives. As we remarked when discussing the representation of JOHN, lemmas can introduce constants that are used in their representations. However in order to make the type checking possible, a type also has to be associated with each constant. This is what the `type = T` lines do. They specify that for each lemma of the family they appear in, a constant of the same name as the lemma and of type `T` should be added to the *typing environment*, which we will call *environment* in the rest of this paper. The toy lexicon we introduced previously would hence give rise to the following environment:

```
john : e
walk : e -> t
```

The reader may wonder why the type `t` (representing truth values) that appears in this environment has not been declared in the lexicon, like `e` was. This is because `t` is present in every version of TY_n, so that it does not have to be explicitly declared.

In addition, `Nessie` needs to be given syntactic information. This information is given in the form of syntax trees, which are inductively defined as follows. A syntax tree can be:

1. A leaf containing a family and a lemma;
2. An n-ary tree whose n daughter nodes are syntactic trees ($n >= 1$).

It is worth emphasising that the trees we have in mind are *abstract*. That is, the left-to-right order of the subtrees of a given node does *not* have to correspond with the order of the expressions in the text. To put it another way, the `Nessie` user is in principle free to use any syntactic formalism. The only requirement imposed is that the trees produced by syntactic analysis then be translated into the abstract schema required by `Nessie`. Also, for historical reasons, `Nessie` has an explicit notion of unary and binary syntax trees. Although these trees are particular cases of item 2 in the previous definition, they have been kept for backward compatibility and because they are useful in practice.

Let's return to our little example. A specification for a syntax tree corresponding to the previous text could look like this:

```
binary(s,
        unary(np,leaf(john,pn)),
        unary(vp,leaf(walk,iv))
      )
```

When this tree and the lexicon discussed above are given to `Nessie`, it produces the following representation: $walk(john)$, which is a well-typed term of type t in the previously shown typing environment.

2.2. How `Nessie` works

The `Nessie` system computes a semantic representation, starting from a lexicon and a syntax tree, as follows. First, a typing environment is extracted from the lexicon in the way described above. Then, the semantic representation is computed inductively using the information in the syntax tree as follows:

1. The representation of a leaf is extracted from the lexicon;
2. The representation of a unary node is the representation of its unique child;
3. The representation of a binary node is obtained by applying the representation of its left child (the functor) to the one of its right child (the argument);
4. If a node has n children whose semantic representations are $M_1, \ldots, M_n$, then a term M_0 of the form $\lambda x_1 . \ldots . \lambda x_n . K$ must be provided. The semantic representation of the whole node will then be $M_0(M_1, \ldots, M_n)$.

Finally, the semantic representation provided by this algorithm is type-checked and, if the term is well-typed, it is beta-reduced. Summing up, the whole process is (at least in spirit) close to the pioneering ideas of Montague. The key changes are that the underlying logic must belong to the TY_n family, and that `Nessie` works with an abstract notion of syntax tree to enable it to be independent of the underlying grammatical formalism. Finally, we remark the fact that `Nessie` is implemented in OCaml[1] (a functional programming language that supports object-oriented programming). Thus, it benefits from all the facilities provided by this language, like type inference, polymorphic types, modules and functors, *etc.* Also, the development in OCaml gives access to tools like AlphaCaml (see `http://pauillac.inria.fr/~fpottier/alphaCaml/`), which makes it easy to manipulate inductive types containing bound variables modulo alpha-equivalence. Such structures are in fact at the heart of our lambda-term manipulation library.

3. Adapting `Nessie` to compositional DRS construction

We turn now to the proposal from (Muskens, 1996b) to treat Discourse Representation Structures (DRSs) as TY_n lambda terms. If every DRS really can be represented by such a lambda-term, then it should be possible to use `Nessie` to build DRSs. This section discusses the main problem encountered while trying to compute DRSs using `Nessie` and the solution used to overcome it.

Consider the sentence "A woman owns a donkey". This sentence contains two indefinite determiners. In Muskens' encoding of DRSs as lambda-terms, each determiner is assumed to introduce a new, unique discourse referent; each discourse referent is modelled using a TY_n constant. Once a discourse referent has been introduced (for example, by an indefinite determiner) and associated with an object, this association holds for the rest of the DRS construction process (that is, discourse referents have global scope). Coming back to our example, the first indefinite determiner may introduce a discourse referent u_1 while the second introduces the discourse referent u_2. The only requirement imposed is that u_1 and u_2 be *distinct* constants — but simple though this requirement is, it is crucial to Muskens' approach.

How exactly does Muskens associate a lambda-term with an indefinite determiner? He first assumes that each occurrence of an indefinite determiner has been associated with a unique number which is written as a superscript. If we annotate our example Muskens-style, we obtain: "a^1 woman owns a^2 donkey". Then Muskens associates a distinct lambda-term to each determiner with the help of the following rule:

$$[a^n] \quad := \quad \lambda P.\lambda Q.\lambda i.\lambda j.\exists k_1 \exists k_2 \\ (i[u_n]k_1 \wedge Q(u_n, k_1, k_2) \wedge P(u_n, k_2, j)),$$

where u_n is a new constant to be added to the language. The numerical values of n are simply read off the numerical superscripts that annotate the input sentence (so our previous example would use discourse referents u_1 and u_2).

And now comes the difficulty: how do we incorporate the required distinctness information into `Nessie`? That is, how should we ensure that it is *always* a *new* constant which is added to the environment? If we simply try to translate the previous rule into `Nessie` notation, we notice that it is not possible. `Nessie` can't express the fact that a leaf introduces a new constant. And this is not just a superficial problem due to a poorly designed lexicon. In fact, it is a deeper problem which could be explained as follows: for the moment, `Nessie` provides no way of differentiating two occurrences of a leaf containing the same lemma and family. And viewed from the traditional perspective of Montague semantics, *it shouldn't have to*: all that should matter is the link between the abstract syntax tree and the lambda terms. But the approach adopted by Muskens goes beyond this.[2] His numerical annotations in effect bring a 'dynamic' component to the syntax-semantics interface. He is requiring the input text to supply additional information (namely, information concerning the distinctness of discourse referents). So we need a way to cope with this in `Nessie` in a generic and extensible way.

Theoretically, what needs to be done to solve this problem is relatively simple: the semantic construction algorithm should let the leaves modify the typing environment. More precisely, each leaf of the syntax tree should be allowed to add constants to the environment, and these constants should be usable in the subsequent semantic representation process. Thus the process of building the semantic representation for a leaf should not only produce a lambda-term representing the leaf, it should also update the typing environment, which should then become an input for the rest of the semantic construction process.

But how should this be implemented at the practical level? The most obvious strategy is simply to directly mimic what Muskens does: that is, to index each leaf by its number in a syntax tree. The constants could then make use of this index, which would be unique for each leaf in a syntax tree. However, this strategy has two drawbacks. First, although the constants it gives rise to will indeed be unique inside a parse tree, this approach will necessarily lead to name clashes as soon as one tries to combine several parse trees. Second, and perhaps more fundamentally, choosing this strategy has a deeper consequence: it would mean that it is up to `Nessie` to decide how two occurrences of a leaf should be distinguished from each other. This means that `Nessie` would become more complex, and yet no flexibility would be provided in the way similar leaves could be distinguished from each other. As `Nessie` is conceived of as a stand-alone semantic construction tool, this approach seems misguided.

Instead we introduce a more generic and flexible ap-

[1] See `http://caml.inria.fr`.

[2] The fact that extra information is required to ensure distinctness of constants is a subtle point, intimately related to the fact that Muskens models discourse referents as constants. He does not discuss this point in detail, but it swiftly becomes visible if an implementation is attempted. Indeed, this is how we became aware of it.

proach. The basic idea is to move from static environment extraction to dynamic environment extraction. The idea is to allow a leaf to contain additional information (to be provided by the syntactic component). This information takes the form of so-called *arguments*, which are in fact identifiers that can be used as names for constants to be added to the typing environment by the leaf.

The types of the constants to be added to the environment will be stored in the lexicon. As an example, here is a lexicon entry `Nessie` could use to handle indefinite determiners:

```
lemma a {
  family = det;
  const $1 : pi;
  term = lam P, Q : pi->drs.
         lam i, j : s. exists k1, k2 : s.
  (
    diff(i,$1,k1) && Q($1, k1, k2) &&
                     P($1,k2,j)
  )
};
```

The crucial point the reader should pay attention to is the `$1` appearing several times in the lexical entry. This refers to the first argument of a leaf. The first occurrence (right after the `const` keyword) specifies that a constant of type `pi`, and whose name is the first argument of the considered leaf, should be added to the typing environment.

Thus, when such a lexicon is loaded, those leaves representing indefinite determiners should have an argument. Put another way, they should look something like this: `leaf(a,det,u1)`. When such a leaf is met, the constant `U1` of type pi is added to the typing environment.[3] Moreover, the representation `Nessie` will provide for this leaf will be the term

$$\lambda P.\lambda Q.\lambda i.\lambda j.\exists k_1.\exists k_2.\\ (i[u1]k_1 \wedge Q(u1,k_1,k_2) \wedge P(u1,k_2,j)).$$

To sum up: it is now possible to distinguish two leaves carrying the same family and lemma by including arguments in the leaves. These arguments are in fact names (provided by the syntactic layer) that can be used as constants to be added to the environment.

Earlier in this section, we mentioned that one possibility one could use to distinguish similar leaves from each other is to index them, Muskens-style, based on their position in the syntax tree. Although we rejected the idea of hard-wiring this approach inside `Nessie` itself, it is certainly possible to simulate such indexing with the help of the arguments facility. Indeed, one can very well imagine, for example, that a preprocessor counts the leaves and includes in each leaf an argument derived from its position in the syntax tree. From `Nessie`'s point of view, though, it is irrelevant what is done. Our implementation ensures that `Nessie` does not have to take care about which names are chosen, or how they are assigned. We believe that the strategy of allowing dynamic environment computations that takes into account more than merely lexical information is a strategy that will be worth exploring in other linguistic contexts, though we will not explore the idea further in this paper.

[3] This has the side-effect that the constant can be used by another leaf coming later in the syntax tree.

4. Computing DRSs

Now that `Nessie` has been adapted to take into account the requirements expressed by Muskens, we show how this tool can be used to handle *automatically* the kind of examples proposed by Muskens in his classic paper.

The paper contains a small fragment of English grammar. It is straightforward to express this grammar as a small DCG (Definite Clause Grammar), and to add the extra information needed to encode discourse referent distinctness. Here, for example, is how the rule for indefinite determiners can be expressed as a DCG rule. We assume that the input text has been annotated in Muskens style (though instead of giving a numeric superscript, we explicitly give the name of the discourse referent to be used). Thus the `N` symbol in the following, the annotation, is simply read as the next symbol in the input string:

```
det(leaf(a, det, N)) --> [a], [N].
```

Equipped with this little DCG, we can get parse trees for simple annotated texts. For example:

```
?- parse([a, u1, farmer, walks]).
unary(t,
  binary(s,
    binary(np, leaf(a, det, u1),
           unary(n0, leaf(farmer, noun))),
    unary(vp, unary(v0, leaf(walk, vin)))
  )
)
```

As we remarked above, we have explicitly annotated the input text with the name of the constant discourse referent to be introduced. There is nothing deep about this choice: we have chosen a manual annotation for the sake of simplicity, but nothing prevents the implementation of a more sophisticated, automated annotation mechanism. As far as `Nessie` is concerned, where the annotations come from makes absolutely no difference.

Let's now turn to the lexicon `Nessie` needs to compute a DRS from the previous parse tree. The lexicon we use follows Muskens original notation closely and thus looks like this:

```
type e; # Entities
type pi; # Registers
type s; # States
type drs = s -> s -> t; # aka box

const V : pi -> s -> e;
const seq : s->s->t;
const diff : s -> pi -> s -> t;

family vin {
  type = e->t;
  pattern = lam v : pi. lam i, j : s.
  (
    seq(i,j) &&
```

```
    _(V(v,j))
  )
};

lemma walk : vin;

family noun {
  type = e->t;
  pattern = lam v : pi. lam i, j : s.
            ( seq(i,j) && _(V(v,j)) )
};

lemma farmer : noun;

family det;

lemma a {
  family = det;
  const $1 : pi;
  term = lam P1, P2 : pi->drs.
         lam i, j : s. exists k1, k2 : s.
  (
    diff(i,$1,k1) && P2($1, k1, k2)
                  && P1($1,k2,j)
  )
};
```

Two remarks should be made concerning this lexicon. First, as the reader may have noticed, the family det contains no type specification. This is because it is a closed family, whose lemmas add no constant to the typing environment. Second, following Muskens, we have introduced constants to make the lambda-terms (a bit) easier to read: $V(\delta, i)$ returns the entity contained in the discourse referent δ in state i; $seq(i, j)$ is short for $\forall v : \pi.V(v, i) = V(v, j)$ (this was written $i[]j$ in Muskens' paper), and *diff*(i, u, j) means that states i and j differ at most on the register u (this was written $i[u]j$ in Muskens' paper).

Given this lexicon, `Nessie` provides the following lambda term for the tree given above:

```
lam i, lam j, exists k1, exists k2,
  (diff(i,u1,k1) && seq(k1,k2) &&
   walk(V(u1,k2)) && seq(k2,j) &&
   farmer(V(u1,j))
```

which is a genuine DRS.

5. Conclusion

In recent years, Discourse Representation Theory (which only a decade ago was a largely theoretical area of research) has begun to be used as a tool in computational linguistics. For example, (Blackburn et al., 2001) shows how DRT (coupled with the inference services provided by the automated reasoning community) can provide a computational solution to presupposition resolution, while (Curran et al., 2007) shows that DRT can be used as a semantic representation with large-scale stochastically-guided grammars.

The work reported here is only in its early stages, but there are at least two reasons why we consider it is worth pursuing further. First, on the theoretical side, Muskens' work was, for a long time, the deepest analysis yet given of the link between semantic construction and classical logic [4]. As deep semantic analysis ultimately stands or falls on the connections it makes with logic and inference, it is vital to explore the computational consequences of Muskens' works in detail. But there is a second, more practical, reason for our interest. As we said at the start of the paper, Muskens argues that TY_n is a *general* framework for semantic analysis, and has given ample evidence in support of this claim. In this paper we have concentrated purely on the links with DRT, but in principle the framework offers much more: a systematic way of combining the insights from many different approaches, ranging from DRT, through situation semantics and classical possible world semantics, to event based semantics.

`Nessie` was designed to provide a computational environment in which these possibilities could be explored. The key point that has emerged from the present study is that by thinking of environments dynamically, we can incorporate the insights of DRT without jeopardising `Nessie`'s stand-alone status. We hope that this will open the way to TY_n based approaches to DRT that incorporate the insights of other semantic frameworks.

6. References

Blackburn, P., J. Bos, M. Kohlhase, and H de Nivelle, 2001. Inference and computational semantics. In *Computing Meaning, Volume 2*. Kluwer, pages 11–28.

Curran, J., S. Clark, and J Bos, 2007. Linguistically motivated large-scale NLP with C&C and Boxer. In *Proceedings of the Demonstrations Section of the 45th Annual Meeting of the Association for Computational Lingusitics (ACL-07)*.

de Groote, P., 2006. Towards a Montagovian Account of Dynamics.

Kamp, H. and U. Reyle, 1993. *From Discourse to Logic*. Kluwer.

Montague, R., 1974. *Formal Philosophy. Selected Papers of Richard Montague. Edited and with an Introduction by Richmond H. Thomason*. New Haven: Yale University Press.

Muskens, R., 1991. Anaphora and the logic of change. In J. van Eijck (ed.), *JELIA '90, European Workshop on logics in AI*, Springer Lecture Notes. Springer, pages 414–430.

Muskens, R., 1996a. *Meaning and Partiality*. Studies in Logic, Language and Information. CSLI Publications.

Muskens, R. A., 1996b. Combining Montague Semantics and Discourse Representation. *Linguistics and Philosophy*, 19:143–186.

[4] In a recent paper (de Groote, 2006), Philippe de Grooote has proposed an alternative encoding of DRT in TY_n. We do think that this formalism is worth experimenting with and hope to do so in future works.