Are Types needed for Natural Language?
In *Applied Logic: How, What and Why*, Pólos and Masuch eds, 79-120, Kluwer 1995

Fairouz Kamareddine*
Department of Computing Science
University of Glasgow
Scotland

November 30, 1996

Abstract

Logic, due to the paradoxes, is absent from the type free λ-calculus. This makes such a calculus an unsuitable device for Natural Language Semantics. Moreover, the problems that arise from mixing the type free λ-calculus with logic lead to type theory and hence formalisations of Natural Language were carried out in a strictly typed framework. It was shown however, that strict type theory cannot capture the self-referential nature of language ([Parsons 79], [Chierchia, Turner 88] and [Kamareddine, Klein 93]) and hence other approaches were needed. For example, the approach carried out by Parsons is based on creating a notion of floating types which can be instantiated to particular instances of types whereas the approaches of Chierchia, Turner and Kamareddine, Klein are based on a type free framework. In this paper, we will embed the typing system of [Parsons 79] into a version of the one proposed in [Kamareddine, Klein 93] giving an interpretation of Parsons' system in a type free theory where logic is present. In other words, we take the standpoint that type freeness is needed yet types are also indispensable. On this ground, by constructing types in the type free theory, we obtain a framework which can be seen as a formalisation of Parsons' claim that Natural Language needs type freeness in order to accommodate self referentiality yet many sentences should be understood as implicitly typed.

Keywords: Type Freeness, Logic, Types, NL Semantics, Self-reference.

1 Introduction

Mixing type freeness and logic leads to contradictions. This can be seen by taking the following simple example.

Example 1.1 *(Russell's paradox)*
Take the syntax of the type free terms of the λ-calculus:

*This article has been prepared while the author was on a study leave at the Department of Mathematics and Computing Science, Eindhoven University of Technology, the Netherlands. The author is grateful for the hospitality, financial and academic support of the university of Eindhoven, and for the productive and enjoyable year spent there.

$E := x \mid E'E'' \mid \lambda x.E'$

Increase this set of terms by adding negation so that whenever E is a term, $\neg E$ is also a term.[1] Now of course, $\lambda x.\neg xx$ is a term and applying it to itself one gets a contradiction (known as Russell's paradox).

One might deny this to be a contradiction by assuming non classical logics such as a three valued one. So that $a = \neg a$ is acceptable when a gets undefined as a truth value. In fact, in the type free λ-calculus, every expression E has a fixed point a such that $Ea = a$. In particular $\lambda x.\neg x$ has a fixed point a and one gets $a = \neg a$. This means of course that $(\lambda x.\neg xx)(\lambda x.\neg xx)$ is a fixed point of $\lambda x.\neg x$.

This will still create a problem when one tries to discuss the axioms and rules of the logic that is being used. The following example makes this point clear:

Example 1.2 *(Curry's paradox)*
Once propositional logic has been defined in the type-free λ-calculus, one must be precise about which of the three concepts below hold in that logic:

- Modus Ponens (MP): From $E \to E'$ and E, deduce E'.
- Deduction Theorem (DT): If Γ is a context, and $\Gamma \cup \{E\} \vdash E'$ then $\Gamma \vdash E \to E'$.
- β-conversion (β): $(\lambda x.E)E' = E[x := E']$.

If all three were present then one gets Curry's paradox. That is, one can show $\vdash E$ for any term E by taking the term $a = \lambda x.(xx \to E)$.

Up to here, only propositional logic has been discussed in the context of the type free λ-calculus. This is not without a reason. Propositional logic, as mentioned above, can be built inside the λ-calculus. The difficulties of Examples 1.1 and 1.2 can be avoided by using non classical logics or by not using all the above three concepts to the full. [Feferman 84] and [Kamareddine 92C] provide a clear introduction to the possible ways of avoiding these paradoxes.

What about quantificational logic? Why has it not been discussed above? The reason for this is very important. It was possible to define propositional logic inside the type free λ-calculus, because the semantic values of all the propositional connectives do actually exist in a model of the type free λ-calculus. That is, if continuity was our basic concept for constructing the model, then all the functions corresponding to the logical connectives will be continuous and hence elements of the model.

With the quantifiers we have another story. The models of the type free λ-calculus without logic cannot model the addition of the quantifiers. The reason for this is that even though $\forall$ is continuous, its presence will trivialise the model as is seen from the following example:

Example 1.3 *(Models of the type free λ-calculus alone are not enough)*
If $\forall$ existed in the model of the type free λ-calculus, one would get that:

$(\forall d \in D)([\![\Phi]\!]_{g[x:=d]} = 1) \Leftrightarrow [\![\Phi]\!]_{g[x:=u]} = 1$

where u is the bottom element of the domain (see [Turner 84] and [Kamareddine 92D]).

[1] Alternatively one can use any of the standard methods of the λ-calculus to define propositional logic, inside the λ-calculus.

This clause has serious consequences. To illustrate this, take in the formal language an element u' which names u (i.e. $[\![u']\!]_g = u$ always). Now see what happens if Φ is $x = u'$:

Applying the above clause one gets: $[\![x = u']\!]_{g[x:=u]} = 1 \Leftrightarrow (\forall d \in D)([\![x = u']\!]_{g[x:=d]} = 1)$.

This implies: $u = u \Leftrightarrow (\forall d \in D)(d = u)$.

Hence $(\forall d \in D)(d = u)$. That is absurd.

The presence of these foundational difficulties implied that logic and λ-calculus, could not be mated freely together. Some restrictions had to be made either on the logic or on type freeness. These restrictions resulted in the following two routes of research:

1.1 Route 1: Logic is more important than expressiveness

The first route placed a big emphasis on logic and deduction systems, but avoided the difficulty by restricting the language used to first or higher order allowing only a limited form of self-reference or polymorphism. Let us here have another look at the paradox and then list the three main examples of Route 1.

The paradox in Example 1.1 arises because any open well-formed formula was allowed to stand for a concept. In fact, Example 1.1 has assumed the following axiom:

Comprehension
For each open well-formed formula $\Phi[x], \exists y \forall x[(yx) \Leftrightarrow \Phi[x]]$ where y is not free in $\Phi[x]$.

By taking $\Phi[x]$ to be $\neg(xx)$ in the comprehension axiom above, one gets:
$\exists y \forall x[(yx) \Leftrightarrow \neg(xx)] \Longrightarrow \forall x[(yx) \Leftrightarrow \neg(xx)] \Longrightarrow [(yy) \Leftrightarrow \neg(yy)]$.

The comprehension axiom assumes that each open well-formed expression determines a concept whose extension exists and is the set of all those elements which satisfy the concept. One could restrict the comprehension principle so that $\Phi[x]$ stands for everything except $\neg(xx)$, but this will not save us from paradox. To see this let $\Phi[x]$ stand for $\neg(x_2x)$ where (y_2x) abbreviates $(\exists z)((yz) \wedge (zx))$. Again, ruling out this instance is not enough for one will still get the paradox if $\Phi[x]$ was taken to be $\neg(x_3x)$ where y_3x abbreviates $(\exists z_1, z_2)((yz_1) \wedge (z_1z_2) \wedge (z_2x))$. This process continues ad infinitum. Even if all such instances were ruled out, the problem will persist. The following example shows this:

Example 1.4 Take $\Phi(x)$ to be $\neg(\exists z_1, z_2, \ldots)[\ldots (z_2z_3) \wedge (z_1z_2) \wedge (xz_1)]$ and let y be the class obtained from the comprehension axiom for $\Phi[x]$.

- If (yy) then $\neg(\exists z_1, z_2, \ldots)[\ldots) \wedge (z_1z_2) \wedge (yz_1)]$. But one can take $z_1 = z_2 = \ldots = y$, and get a contradiction.
- If $\neg(yy)$ then $(\exists z_1, z_2, \ldots)$ $[\ldots z_2) \wedge (z_1z_2) \wedge (yz_1)]$. But as (yz_1) then $\Phi[y]$; however we have that $\neg\Phi[y]$. Contradiction.

For a further explanation of this process, see [Kamareddine 89] and [Kamareddine, Nederpelt 94B].

1.1.1 First Order Languages

The first route of avoiding the paradox by using a first order language, insisted that logic must be strongly present but that self-reference should not. In fact, in first order languages, a separation between functions and objects exists and a quantifier ranges only over objects and not over functions. Of course in such a language no paradox arises because one cannot have self-reference, as a function cannot be an object and so cannot apply to itself.

1.1.2 Second Order Languages

The problem is also faced with higher order languages. The following will show this to be the case.

Notation 1.5 The following metavariables are used:

- $F, G \ldots$ refer to n-place predicate variables.
- $x, y, z, w, \ldots$ refer to individual variables.
- $a, b, \ldots$ refer to singular terms. (These are the nominalisation of functions.)

The primitive symbols of the language are: $\Rightarrow, \neg, =, \forall, \lambda$. The others are defined in the metalanguage.

Definition 1.6 *(The two problematic Axioms)*
In order to deal with self application and to allow self-reference, we need the following axioms:

- *(A3*)* $\exists x(a = x)$, *for a singular term in which* x *is not free.*
- *(CP*)* $\exists F \forall x[F(x) \Leftrightarrow \Phi[x]]$ *where* F *does not occur free in* Φ.

The paradox comes from (CP*) together with (A3*) under various logical laws as can be seen from the following example:

Example 1.7 From (A3*), one can derive that $\forall x \Phi \Rightarrow \Phi[x := a]$. Substituting F for x in the special instance of (CP*): $\exists F \forall x[F(x) \Leftrightarrow \exists G[x = G \wedge \neg G(x)]]$ will lead to the paradox.

The problem here again has been avoided in many ways, one of them is to restrict the language, disallowing some forms of self-reference. Cocchiarella's two ways of avoiding the paradox for example, have been to restrict (CP*) or (A3*) (see [Cocchiarella 84]).

1.1.3 Simple Type Theory

Since Russell's letter to Frege, concerning the inconsistency of Frege's system, there have been many attempts at overcoming the paradox. The first two accounts of avoiding the paradox by restricting the language were due to Russell and Poincaré (see [Russell 1908] and [Poincaré 1900]). They both disallowed impredicative specification: only predicative specification has been used, where $A = \{x : \Phi(x)\}$ is predicative iff Φ contains no variable which can take A as a value. This theory obviously overcomes the paradox, for one assumes all the elements of the set before constructing it and so $\neg xx$ is no longer allowed.

It became obvious however, that this theory had many unattractive features. Of these features we mention that at each level there exists a natural number system, such that $1, 2, 3, \ldots$

at each level n are different from $1, 2, 3, \ldots$ at level $n+1$. Moreover, polymorphic functions (that is functions which take arguments from many levels such as the polymorphic identity function) do not exist in Russell's type theory. In addition to that, this approach (of Russell and Poincaré) is rather unsatisfactory from the point of view of self-reference because one needs impredicative formulas such as the sentence *it is nice to be nice*. These formulas are fundamental to natural language semantics.

1.2 Route 2: Expressiveness is more important than logic

The second route placed the emphasis on the expressiveness of the language and the richness of functional application and self-reference, but at the expense of including logic in the language except if restrictions are made (such as using non-classical logics). Church's and Curry's work for example, was on the language side. They decided to enrich the syntax and the language but to avoid or restrict logic. They introduced sophisticated systems of λ-calculus and combinators, but the importance was shifted from logic to the expressiveness properties of the language. So fixed points were shown to exist, self application functions and solutions to all sorts of equations were shown to exist. Of course they could move freely in the jungle of the type free terms as logic was not the main theme. Moreover, they explained things like $a = \neg a$ by saying that every λ-term has a fixed point, in particular the λ-term $\lambda x.\neg x$. Their use of logic however was very elementary.

After a while, attention moved to various forms of the typed λ-calculus. This may have been due to the usefulness of the typing schemes, or to the presence already of some type systems which aimed at combining expressiveness and logic. The basic aim in this route became to provide systems which can type check as much as possible of self-referential terms. The line remained however, to ignore logic (as a deduction system) and to make sense of as many self-referential terms as possible.

This led to various formulations of typing systems; some of which can type check self-referential sentences such as the self-application function $\lambda x.xx$ and the fixed point operator $Y = \lambda f.(\lambda x.f(xx))\lambda x.f(xx)$ and some cannot. All these type systems, use the following as their underlying syntax of types $s ::= x|c|s \to s$ which says that a type is either a variable or a constant or an arrow. Type systems such as λ_2, λ_μ and $\lambda_\cap$ (see [Barendregt, Hemerik 90] and [Kamareddine 92A]), add other types to this set of types in order to typecheck more terms such as Y and $\lambda x.xx$.[2] Systems which use only the above syntax of types, even though they can be polymorphic, cannot typecheck Y or $\lambda x.xx$ (Milners's ML system in [Milner 78] is such an example).

Of course this rich variety of typing systems has not reached Natural Language Semantics. We find it a pity that in NL, some of these type systems have never been heard of. We believe that only a perfect combination of expressivity (and here type theory plays a role) and logic can be a sound system for NL. It might be asked moreover why did we move from expressivity and type freeness to type theory? This is indeed a very good question, to which we devote a whole section (see section 4).

[2] [Kamareddine, Nederpelt 94A] provides a way of unifying a significant number of type systems but again [Kamareddine, Nederpelt 94A] takes the line of route 2 and logic is missing.

1.3 The major themes of the paper

1.3.1 Theme 1

Routes 1 and 2 resulted in a gap between strong logics and fully expressive languages. The need to remove the gap created various theories such as Martin-Löf's type theory and Feferman's T_0 which were polymorphic, allowed self reference and contained a big fragment of logic (see [Martin-Löf 73] and [Feferman 79]).

While the polymorphically typed languages which contained logic (such as Martin-Löf's and Feferman's) were being developed, research on natural language was already based on Montague semantics and Russell's type theory and there were enough problems to tackle from the linguistic point of view that the limited formalism used was not regarded as a deficiency.

However the need for the combination of expressive languages and strong logics is unquestionable (see [Feferman 84]), and the necessity of such a combination for Natural Language is undoutable (see [Kamareddine, Klein 93]). This combination was the main concern of many linguists in the last decade ([Parsons 79], [Chierchia, Turner 88] and [Kamareddine, Klein 93]). This paper will hence attempt, as a first theme, to review these three fundamental aproaches. This will be done in Section 2. The approaches of [Parsons 79] and [Kamareddine, Klein 93] will be the centre of attention of the paper. The former will be interpreted in a version of the latter. The [Chierchia, Turner 88] approach will be used for comparisons.

1.3.2 Theme 2

The approach of [Kamareddine, Klein 93] is very attractive from the type theory point of view. The typing strategy provided there, is based on the structure of the models of the type free λ-calculus which demands that $(*a \to *b) \leq *a$ for $*a$ and $*b$ being any variable types. This ordering is the basis of applying functions to themselves as the following example shows:

Example 1.8 In $\lambda x.xx$, the operator occurrence of x requires that x be of type $*a \to *b$. For this operator occurrence of x to apply to the argument occurrence of x, the second x must also be of type $*a$.

Based on this observation, [Kamareddine, Klein 93] builds a relation between types which guarantees that every arrow type is included in its domain space. The system allows only typed abstraction, of the form $\lambda x : \sigma.\alpha$, but permits any two terms to apply to themselves. Logic, (including quantifiers) is present too. This might surely be thought to lead to the paradox by applying the term $(\lambda x : (e \to p).\neg xx)$ to itself. This will not be the case however, due to the notion of circular types (see Section 2.3).

This paper will start from the system of [Kamareddine, Klein 93], but will add variable types. This will enable the retrieval of the type free λ-calculus in a systematic way. Moreoever, the construction of types will become more general and one can make sense of all non paradoxical terms. In fact, with the addition of variable types, the new system turns out to have more polymorphic power than λ_2, λ_μ and $\lambda_\cap$ (see [Kamareddine 92A]) and allows typing the fixed point operator $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$, the self application function $\lambda x.xx$ and all the possible mixtures of Y and $\lambda x.xx$. This possiblity of type checking Y and $\lambda x.xx$ shows that the system allows all self-reference, as long as it is non paradoxical. This extension will be found in Section 2.3.

1.3.3 Theme 3

The need for self reference, which requires a type free framework, confuses the fact as to whether types are or are not needed. In fact, one often sees types being constructed inside a type free framework and vice versa. Hence it is very difficult to answer whether or not types are needed for natural language. If one looks back at the latest formalisations of natural language ([Parsons 79], [Chierchia, Turner 88] and [Kamareddine, Klein 93]), one finds them all jumping between type freeness and typing. Section 4 below discusses the questions of typeness and type freeness and supports Parsons' claim that NL is implicitly typed yet type freeness is needed to represent it.

1.3.4 Theme 4

Based on the claim of Theme 3 that NL is implicitly typed yet self reference is necessary, one is faced with the question of how to combine type freeness and typing in such a way that self reference of NL can be accommodated, yet grammatical or ungrammatical sentences can be explained. That is, one would like to have a rich typing scheme which can be used as a classification for good/bad sentences, while the freedom of applying functions to themselves is preserved. This paper claims that the approach of [Kamareddine, Klein 93] extended with variable types, embodies in it such type freedom and classification scheme. This will be shown by interpreting Parsons' theory in the theory provided in Theme 2 and by drawing a comparison with the theory of Chierchia, Turner.

In short, as a fourth theme of this paper, an embedding of the typing system of [Parsons 79] into a version of the one proposed in [Kamareddine, Klein 93] will be given. This embedding can be viewed as constructing a model which accommodates Parsons' claim of typing and non-typing of Natural Language. This model is a type free system where all types except the circular ones can be constructed. The comparison drawn between the three approaches will show that the typing scheme of [Kamareddine, Klein 93] is the most flexible for NL interpretation. The embedding will be done in Section 5, the usefulness of the extended system will be discussed in Section 6 and the comparisons of the three approaches will be carried out in Section 7.

2 Three polymorphic systems of Natural Language

2.1 Parsons' system

Parsons starts by explaining that if one is to accommodate various natural language constructs in Montague's approach, then there needs to be an infinity of categories which contain the same elements yet the types of those elements differ from one category to another. That is, he argues that Montague's approach is not polymorphic and that there is a need for a language which allows functions to take arguments from variable types and to return arguments in variable types. Moreover he claims that those variable types should be instantiated as necessary.

Example 2.1 *John talks about* could take either *Mary* or *a proposition* as arguments as can be seen from the following sentences:

1. *John talks about Mary*

2. *John talks about a proposition*

To deal with these polymorphic functions such as *talks about*, Parsons introduces two sorts of types: *the fixed types* and *the floating types*. The fixed types are always fixed; $\langle e, t\rangle$, the type of propositional functions is an example of a fixed type. Floating types on the other hand, change in value. They should be understood as variable types and can be instantiated to various types instances.

Example 2.2 The semantic types of both *individuals* and *propositions* are fixed types. The first is $\langle e, t\rangle$ and the second is $\langle\langle s, t\rangle, t\rangle$. Both individuals and propositions moreover, are syntactically common nouns.

To represent the association of types to categories, Parsons records information relevant to typing as a superscript to the category.

Example 2.3 women^{e} is of category CN^{e} and its type is $\langle e, t\rangle$ whereas proposition$^{\langle s,t\rangle}$ is of category CN$^{\langle s,t\rangle}$ and its type is $\langle\langle s, t\rangle, t\rangle$.

Syntactic rules should obey semantic typing as the following example shows:

Example 2.4 VP^{e} which is of type $\langle e, t\rangle$ can be combined with John^{e} which is of type $\langle\langle e, t\rangle, t\rangle$ but not with proposition$^{\langle s,t\rangle}$ which is of type $\langle\langle s, t\rangle, t\rangle$.

Up to here one can guarantee that the following are well formed:

- *That John runs or that he walks amazes Mary*
- *That John runs or he walks amazes Mary*
- *That John runs or walks amazes Mary*

whereas the following are not:

- *Bill or that John runs*
- *walks or obtains*
- *That John walks runs*
- *Bill obtains*

How does the idea of floating types accommodate the sentences of Example 2.1? That is, how can one make *talk about* take two different arguments, individuals as in 1 and propositions as in 2? The solution is simple, make *about* be a floating type. Before we explain further what type should *about* have, let us give the types and categories of the other constituents of the two sentences in example 2.1. This is done as follows:

- *John*, of category NP^{e} has the fixed type $\langle\langle e, t\rangle, t\rangle$.
- *talks*, of category VP^{e} has the fixed type $\langle e, t\rangle$.
- *Mary*, of category NP^{e} has the fixed type $\langle\langle e, t\rangle, t\rangle$.
- *a proposition*, of category NP$^{\langle s,t\rangle}$ has the fixed type $\langle\langle\langle s, t\rangle, t\rangle, t\rangle$.

The next step is to be able to combine *talks* with *about Mary* or with *about a proposition* to result in a construct of category VP^e. This construct will then be combined with *John* of category NP^e and the result will be a sentence. For this we will need the following syntactic rules:

- S2. If $\alpha \in CN^\tau$ then $F_0(\alpha)$, $F_1(\alpha)$ and $F_2(\alpha) \in NP^\tau$ where $F_0(\alpha) =$ every α, $F_1(\alpha) =$ the α and $F_2(\alpha) =$ a(n) α.
- S6. If $\alpha \in {}^{\tau_1} PREP^{\tau_2}$ and $\beta \in NP^{\tau_2}$ then $\alpha\beta \in ADV^{\tau_1}$
- S10. If $\alpha \in ADV^\tau$ and $\beta \in VP^\tau$ then $\beta\alpha \in VP^\tau$

Now, *talks* of category VP^e will combine with *about Mary* or *about a proposition* of category ADV^e according to rule S6. *about Mary* and *about a proposition* will belong to the same category ADV^e and will have the same fixed type $\langle\langle s, \langle e, t\rangle\rangle, \langle e, t\rangle\rangle$. With *Mary* and *a proposition* being of fixed types, it is *about* which should change its type as in the following two cases:

1. *about* is of category ${}^e\text{PREP}^e$ and has for type $\langle\langle s, f(NP^e)\rangle, f(ADV^e)\rangle$.
2. *about* is of category ${}^e\text{PREP}^{\langle s,t\rangle}$ and has for type $\langle\langle s, f(NP^{\langle s,t\rangle})\rangle, f(ADV^e)\rangle$.

To accommodate this multi-typing, Parsons considers *about* to be of category ${}^e PREP^\tau$ and to have, for a particular type τ, the type $\langle\langle s, f(NP^\tau)\rangle, f(ADV^e)\rangle$.[3] The following two tables show how this works:

Table 2.5

Parsons' account of John talks about Mary	*Category* -	*Rule* -	*Type*
about	${}^e PRED^e$		$\langle\langle s, \langle\langle e, t\rangle, t\rangle\rangle, \langle\langle s, \langle e, t\rangle\rangle, \langle e, t\rangle\rangle\rangle$
Mary	NP^e		$\langle\langle e, t\rangle, t\rangle$
about Mary	ADV^e	*S6*	$\langle\langle s, \langle e, t\rangle\rangle, \langle e, t\rangle\rangle$
talk	VP^e		$\langle e, t\rangle$
talk about Mary	VP^e	*S10*	$\langle e, t\rangle$
John	NP^e		$\langle\langle e, t\rangle, t\rangle$
John talks about Mary			t

Table 2.6

Parsons' account of John talks about a proposition	*category*	*Rule*	*Type*
proposition	$CN^{\langle s,t\rangle}$		$\langle\langle s, t\rangle, t\rangle$
a proposition	$NP^{\langle s,t\rangle}$	*S2*	$\langle\langle\langle s, t, \rangle t\rangle, t\rangle$
about	${}^e PRED^{\langle s,t\rangle}$		$\langle\langle s, \langle\langle\langle s, t\rangle t\rangle, t\rangle$ $, \langle\langle s, \langle e, t\rangle\rangle, \rangle e, t\rangle\rangle\rangle$
about a proposition	ADV^e	*S6*	$\langle\langle s, \langle e, t\rangle\rangle, \langle e, t\rangle$
talk	VP^e		$\langle e, t\rangle$
talk about a proposition	VP^e	*S10*	$\langle e, \rangle$
John	NP^e		$\langle\langle e, t\rangle, t\rangle$
John talks about a propostion			t

[3]Note that each instance of a floating type is a fixed type.

The following tables summarize the vocabulary of types and words of these types as used by Parsons.

Table 2.7

Categories PC	Corresponding semantic types f(PC)
s	t
CN^{τ}	$\langle \tau, t \rangle$
VP^{τ}	$\langle \tau, t \rangle$
NP^{τ}	$\langle f(VP^{\tau}), t \rangle$
${}^{\tau_1}V^{\tau_2}$	$\langle \langle s, f(NP^{\tau_2}) \rangle, f(VP^{\tau_1}) \rangle$
${}^{\tau_1}V^{\tau_2,\tau_3}$	$\langle \langle s, f(NP^{\tau_2}) \rangle, \langle \langle s, f(NP^{\tau_3}) \rangle, f(VP^{\tau_1}) \rangle \rangle$
ADV^{τ}	$\langle \langle s, f(VP^{\tau}) \rangle, f(VP^{\tau}) \rangle$
ADF	$\langle \langle s, t \rangle, t \rangle$
${}^{\tau_1}PREP^{\tau_2}$	$\langle \langle s, f(NP^{\tau_2}) \rangle, f(ADV^{\tau_1}) \rangle$

Table 2.8

Words of fixed type	Syn Type	Sem Type
man^e, $woman^e$, $park^e$, $fish^e$, pen^e, $unicorn^e$, $\text{-}body^e$	CN^e	$\langle e, t \rangle$
$fact^{\langle s,t \rangle}$, $proposition^{\langle s,t \rangle}$, $answer^{\langle s,t \rangle}$	$CN^{\langle s,t \rangle}$	$\langle \langle s, t \rangle, t \rangle$
run^e, $walk^e$, $talk^e$, $rise^e$, $change^e$	VP^e	$\langle e, t \rangle$
$obtain^{\langle s,t \rangle}$	$VP^{\langle s,t \rangle}$	$\langle \langle s, t \rangle, t \rangle$
$John^e$, $Mary^e$, $it_0^e, it_1^e \ldots$	NP^e	$\langle \langle e, t \rangle, t \rangle$
$The\ Pythagorean\ theorem^{\langle s,t \rangle}, it_0^{\langle s,t \rangle}, it_1^{\langle s,t \rangle} \ldots$	$NP^{\langle s,t \rangle}$	$\langle \langle \langle s, t \rangle, t \rangle, t \rangle$
${}^e eat^e$, ${}^e date^e$	${}^e V^e$	$\langle \langle s, \langle \langle e, t \rangle, t \rangle \rangle, \langle e, t \rangle \rangle$
${}^e believe^{\langle s,t \rangle}$, ${}^e assert^{\langle s,t \rangle}$	${}^e V^{\langle s,t \rangle}$	$\langle \langle s, \langle \langle \langle s, t \rangle, t \rangle, t \rangle \rangle, \langle e, t \rangle \rangle,$
${}^{\langle s,t \rangle} amaze^e$	${}^{\langle s,t \rangle} V^e$	$\langle \langle s, \langle \langle e, t \rangle, t \rangle \rangle, \langle \langle s, t \rangle, t \rangle \rangle$
${}^e buy^{e,e}$	${}^e V^{e,e}$	$\langle \langle s, \langle \langle e, t \rangle, t \rangle \rangle, \langle \langle s, \langle \langle e, t \rangle, t \rangle \rangle, \langle \langle e, t \rangle, t \rangle \rangle \rangle$
${}^e tell^{e,\langle s,t \rangle}$	${}^e V^{e,\langle s,t \rangle}$	
$rapidly^e$, $slowly^e$, $voluntarily^e$	ADV^e	$\langle \langle s, \langle e, t \rangle \rangle, \langle e, t \rangle \rangle$
$necessarily$	ADF	$\langle \langle s, t \rangle, t \rangle$
${}^e in^e$	${}^e PREP^e$	

Table 2.9

Words of floating types	Type	Semantic Type
$thing^{\tau}$	CN^{τ}	$\langle \tau, t \rangle$
$set^{\langle \tau,t \rangle}$	$CN^{\langle \tau,t \rangle}$	$\langle \langle \tau, t \rangle, t \rangle$
$property^{\langle s,\langle \tau,t \rangle \rangle}$	$CN^{\langle s,\langle \tau,t \rangle \rangle}$	$\langle \langle s, \langle \tau, t \rangle \rangle, t \rangle$
$exist^{\tau}$	VP^{τ}	$\langle \tau, t \rangle$
$it_0^{\tau}, it_1^{\tau}, \ldots$	NP^{τ}	$\langle \langle \tau, t \rangle, t \rangle$
${}^e find^{\tau}$, ${}^e lose^{\tau}$, ${}^e love^{\tau}$, ${}^e hate^{\tau}$, ${}^e seek^{\tau}$, ${}^e conceive^{\tau}$	${}^e V^{\tau}$	$\langle \langle s, \langle \langle \tau, t \rangle, t \rangle \rangle, \langle e, t \rangle \rangle$
${}^e give^{e,\tau}$	${}^e V^{e,\tau}$	
${}^e about^{\tau}$	${}^e PREP^{\tau}$	
${}^{\tau_1} be^{\tau_2}$	${}^{\tau_1} V^{\tau_2}$	$\langle \langle s, \langle \langle \tau_2, t \rangle, t \rangle \rangle, \tau_1, t \rangle \rangle$

2.2 The Chierchia, Turner system

The Chierchia, Turner system is based on Turner's theory of properties which appeared in [Turner 87]. In [Turner 87], Scott domains are completely abandoned and Frege's comprehension principle is restricted in such a way that the paradox is no longer derivable. Turner starts with a first order theory which has a pairing system and adds to this theory an operator p (to serve as the predication operator) together with the lambda operator. Then in this case, if one assumes full classical logic and Frege's comprehension principle, one will certainly derive the paradox.

Example 2.10 Take $a = \lambda x.\neg p(x,x)$, then $p(a,a) \Leftrightarrow \neg p(x,x)[x := a] \Leftrightarrow \neg p(a,a)$. Contradiction.

Let us look again at Example 1.2 and in particular at the third concept discussed there: namely, β. β could be divided into two parts:

1. **Contraction** $\quad p(\lambda x.E, E') \rightarrow E[E' := x]$
2. **Expansion** $\quad E[E' := x] \rightarrow p(\lambda x.E, E')$

Contraction causes no problems but expansion does in the presence of negation.

Example 2.11 If A is atomic then we can accept $A(t,x) \rightarrow p(\lambda x.A, t)$. But we cannot accept it when A is like Russell's property $\lambda x.\neg p(x,x)$, an atomic term proceeded by a negation sign.

This is exactly what guides Turner in setting his theory. For the theory now will have the following axioms replacing Frege's comprehension principle:

$(E1)$ $\quad A(t,x) \rightarrow p(\lambda x.A, t)$ when A is atomic.
(R) $\quad p(\lambda x.A, t) \rightarrow A(t,x)$.
(I) $\quad p(\lambda x.p(\lambda y.A, t), u) \rightarrow p(\lambda y.p(\lambda x.A, u), t)$

To build models for T above, one uses the fixed point operator to turn an ordinary model of the first order theory into a model which will validate in it as many instances of the comprehension axiom as possible. It will of course validate only the safe instances whereas the paradoxical ones will oscillate in truth-values. The inductive step to build the model should be obvious. The following example illustrates how they work:

Example 2.12 One way is to start with the first order model, and an operator PI which is empty at the beginning. Then at the next step, extend PI to also contain the pairs $\langle [\![\lambda x.A]\!], [\![t]\!]_g \rangle$ such that $[\![A]\!]_{g[x:=[\![t]\!]_g]} = 1$ and so on until one gets a limit ordinal χ where PI then is to have in it all the pairs $\langle e, d \rangle$ such that for some ordinal smaller than this χ, $< e, d >$ belongs to all the intermediate PI's.

The above discussion goes as far as the theory is concerned. The system used however by Chierchia and Turner, despite the fact that it is based on a type free theory, still constructs types (these are called sorts in that paper). In fact in their paper, the authors provide a lengthy discussion on the usefulness of types for NL.

The construction of types — recall they call them sorts — is very straightforward in [Chierchia, Turner 88], it goes as follows (Only PT_1 will be considered):

Definition 2.13 *(Sorts)*
Basic sorts: *The basic sorts are* e, u, nf, i, pw, Q. *These stand for individuals, urelements, nominalised functions, information units, possible worlds and generalised quantifiers respectively.*
Complex sorts: *The complex sorts are* $\langle a_1, \langle \ldots, \langle a_n, b\rangle \ldots\rangle$ *where for* $1 \leq i \leq n, a_i$ *and* b *are any of the basic sorts.*

In section 7, we will see a comparison between this system and that of [Kamareddine, Klein 93].

2.3 The extended version of the Kamareddine, Klein system

Here, an extended version of [Kamareddine, Klein 93] will be presented, rather than their initial system. The extension will be to allow type variables. This is done in order to allow the extraction of the type free terms from this calculus as will be seen in Section 3. Moreover, this extension will enable the representation of all non paradoxical sentences regardless of whether their forming parts were originally typed or not. That is for example, not only the fixed point operator for a particular type will be shown to exist, but the fixed point operator of any type (as long as it's not circular). Section 3 elaborates more on the properties of this system. Moreover, a type checker for this extended system has been written in [Kamareddine 92A]. Let us call this extended system λ_L standing of course for λ and Logic.

It is assumed that term variables are $x, x', y, y', z, z' \ldots$, that $V, V', V'', \ldots$ range over these variables and that $\alpha, \alpha_0, \alpha_1, \ldots, \beta, \beta_0, \beta_1 \ldots$, range over type variables. It is assumed further that $E, E', E'', \ldots E_1, E_2, \ldots, \Phi, \Psi, \ldots$, range over expressions and $T, T', T_1, T_2, \ldots$ range over type expressions.

Definition 2.14 *(Types)*
Types are constructed as follows:
$T ::= \beta \mid Basic \mid (T_1 \to T_2)$
$Basic ::= p \mid t \mid e$

This syntax of types is similar to that of [Kamareddine, Klein 93] except that type variables are allowed. Here p is the type of propositions, t is the type of truths (that is, of all the true propositions) and e is the type of objects. In fact e contains everything, variable types, basic types and arrow types. This is the case due to the subsumption relation $\leq$ on the types defined as follows:

Definition 2.15 *(Subsumption Relation)*
The ordering/subsumption relation on types is given by the following rules:
i) $T \leq e$
ii) $t \leq p$
iii) $(T \to T') \leq T$
iv) $T \leq T$
v) if $T \leq T'$ *and* $T' \leq T$ *then* $T = T'$
vi) if $T \leq T'$ *and* $T' \leq T''$ *then* $T \leq T''$
vii) if $T \leq T'$ *then* $(T_1 \to T) \leq (T_1 \to T')$

We say that by $(T \leq T')$, T *subsumes* T'; intuitively it means that any expression which is of type T is also of type T'.

It is mainly clause iii) of Definition 2.15 which enables one to have self application in the system and it is the notion of circular types defined below, which allows the avoidance of the paradoxes.

Definition 2.16 *(Monotypes)*
We say that a type T is a monotype if it contains no type variables.

This is how this system deviates from that of [Kamareddine, Klein 93] which allows only monotypes.

Definition 2.17 *(Circular Type)*
We say that a type $T \to T'$ is circular iff:

- *Either $T' \leq p$ and $T \equiv T_1 \to T_2$ where $T_2 \leq p$*
- *Or T is circular*
- *Or T' is circular*

Lemma 2.18 *If $T \to T'$ is not circular, then neither T nor T' are circular.*
Proof: *Obvious.* □

Example 2.19 *$(\beta \to p) \to t, ((e \to p) \to p) \to e$ and $(e \to p) \to (p \to p)$ are circular types.*

Remark 2.20 Here it will be asked what will happen to Noun Phrases and Generalised Quantifiers like *John*, which are usually taken to be of type $(e \to p) \to p$; i.e. their type is circular. The answer is to make *John* of type $(e \to e) \to p$ instead. This will be done via a function H to be defined in Section 5. Syntactically *John* and *runs* can combine because the first is CN^e and the second is VP^e, this is exactly like the treatment of Parsons where a VP^e takes an NP^e and returns a p. Semantically this mixing is allowed because the type of *John* can mix with the type of *run* which is $e \to p$ as $e \to p \leq e \to e$ according to our relation $\leq$.

Definition 2.21 *(Expressions)*
The following syntax of expressions is assumed:
$E = V|(E_1E_2)|(\lambda V : T.E_1)|(E_1 \wedge E_2)|(E_1 \to E_2)|(\neg E_1)|(\forall V : T.E_1)|(E_1 = E_2)$

Constants, disjunctive and existential expressions are omitted for the sake of clarity. It might be remarked here that our terms are typed, so how are we talking about type free terms? It will be shown however that the type free λ-calculus can be embedded in our system (see Section 3), and hence we have all the type free terms at our disposal. In fact, it is precisely the addition of variable types which enables such embedding.

Notation 2.22 Sometimes, when T contains only variable types and when none of these variable types occur in E, we write $\lambda V.E$ instead of $\lambda V : T.E$. For example, instead of $\lambda x : \alpha.x$ we write simply $\lambda x.x$.

Definition 2.23 *($\perp$)*
A particular expression $\perp$ will be defined in the usual way (such as: $\perp =_{df} \lambda xy.xy = \lambda xy.y$) and will have the property that it should never be derivable.

Finally, we assume the usual conventions for the dropping of parentheses when no confusion occurs, and the usual definition of implicit substitution of the λ-calculus in contrast to the explicit one presented in [Kamareddine, Nederpelt 93] and [Kamareddine, Nederpelt 94B].

When an expression E has type T we write $E : T$. In particular we write $\Phi : p$ for Φ a proposition and $\Phi : t$ for Φ true.

Definition 2.24 *(Environments)*
An environment is a set of type assignments $(V : T)$ which assigns the type T to the variable V, such that a variable is not assigned two different types. We let Γ range over environments.

Notation 2.25 *When $(V : T) \in \Gamma$, we say that the type of V in the environment Γ is T. Moreover, the notation $\Gamma \vdash E : T$ means that from the environment Γ, we can deduce that the expression E has type T.*

Definition 2.26 *(Typing λ-expressions)*
The following rules are used to type the expressions:

$$\frac{(V : T) \in \Gamma}{\Gamma \vdash V : T} \tag{1}$$

$$\frac{\Gamma \vdash E : T \qquad T \leq T'}{\Gamma \vdash E : T'} \tag{2}$$

$$\frac{\Gamma \vdash E_1 : T \to T' \qquad \Gamma \vdash E_2 : T}{\Gamma \vdash E_1 E_2 : T'} \tag{3}$$

$$\frac{(V : T) \cup \Gamma \vdash E : T'}{\Gamma \vdash \lambda V.E : T \to T'} \qquad \text{where } T \to T' \text{ is not circular} \tag{4}$$

$$\frac{\Gamma \vdash E_1 : T \qquad \Gamma \vdash E_2 : T}{\Gamma \vdash (E_1 = E_2) : p} \tag{5}$$

$$\frac{\Gamma \vdash (E_1 = E_2) : t \qquad \Gamma \vdash E_1 : T}{\Gamma \vdash E_2 : T} \tag{6}$$

$$\frac{\Gamma \vdash \Phi : p}{\Gamma \vdash \neg\Phi : p} \tag{7}$$

$$\frac{\Gamma \vdash \Phi : p \qquad \Gamma, \Phi : t \vdash \bot : t}{\Gamma \vdash \neg\Phi : t} \tag{8}$$

$$\frac{\Gamma, \neg\Phi : t \vdash \bot : t \qquad \Gamma, \Phi : p}{\Gamma \vdash \Phi : t} \tag{9}$$

$$\frac{\Gamma \vdash \Phi : p \qquad \Gamma \vdash \Psi : p}{\Gamma \vdash (\Phi \wedge \Psi) : p} \tag{10}$$

$$\frac{\Gamma \vdash \Phi : t \qquad \Gamma \vdash \Psi : t}{\Gamma \vdash (\Phi \wedge \Psi) : t} \tag{11}$$

$$\frac{\Gamma \vdash (\Phi \wedge \Psi) : t}{\Gamma \vdash \Phi : t} \qquad \frac{\Gamma \vdash (\Phi \wedge \Psi) : t}{\Gamma \vdash \Psi : t} \tag{12}$$

$$\frac{\Gamma, \Phi : t \vdash \Psi : p \qquad \Gamma \vdash \Phi : p}{\Gamma \vdash (\Phi \to \Psi) : p} \tag{13}$$

$$\frac{\Gamma, \Phi : t \vdash \Psi : t \qquad \Gamma \vdash \Phi : p}{\Gamma \vdash (\Phi \to \Psi) : t} \tag{14}$$

$$\frac{\Gamma \vdash \Phi : t \qquad \Gamma \vdash (\Phi \to \Psi) : t}{\Gamma \vdash \Psi : t} \tag{15}$$

$$\frac{\Gamma, E : T \vdash \Phi : p}{\Gamma \vdash \forall V : T.\Phi : p} \tag{16}$$

$$\frac{\Gamma, V : T \vdash \Phi : t}{\Gamma \vdash \forall V : T.\Phi : t} \qquad \textit{where } V \textit{ is not free in } \Phi \textit{ or any assumptions in } \Gamma \tag{17}$$

$$\frac{\Gamma \vdash \forall V : T.\Phi : t \qquad \Gamma \vdash E : T}{\Gamma \vdash \Phi[x := E] : t} \tag{18}$$

$$\Gamma \vdash [(\lambda V : T.E) = (\lambda V' : T.E[V := V'])] : t, \qquad \text{where } V' \text{ is not free in } E \tag{19}$$

$$\Gamma \vdash [(\lambda V : T.E)E' = E[V := E']] : t, \tag{20}$$

$$\frac{\Gamma \vdash [E_1 = E_2] : t \qquad \Gamma \vdash [E_1' = E_2'] : t}{\Gamma \vdash [E_1 E_1' = E_2 E_2')] : t} \tag{21}$$

$$\frac{\Gamma \vdash E : T}{\Gamma \vdash [E = E] : t} \tag{22}$$

$$\frac{\Gamma \vdash [E_1 = E_2] : t \qquad \Gamma \vdash [E_1 = E_3] : t}{\Gamma \vdash [E_2 = E_3] : t} \tag{23}$$

$$\frac{\Gamma \vdash [E_1 V = E_2 V] : t}{\Gamma \vdash [E_1 = E_2] : t} \qquad \text{where } V \text{ is not free in } E_1, E_2 \text{ or any assumptions in } \Gamma \tag{24}$$

3 Type freeness, logic and the paradoxes in the proposed system

The type free λ-calculus, has the following syntax of terms: $E ::= V | (E_1 E_2) | \lambda V.E_1)$. With Notation 2.22, the type free λ-calculus is retrieved. In fact here is how we can embed the type free λ-calculus (λ for short), in our system λ_L via the embedding function $\mathcal{J}$:

Definition 3.1
We define an embedding function $\mathcal{J} : \lambda \longrightarrow \lambda_L$, which embeds λ in λ_L as follows:

- $\mathcal{J}(V) = V$
- $\mathcal{J}(E_1 E_2) = \mathcal{J}(E_1)\mathcal{J}(E_2)$

- $\mathcal{J}(\lambda V.E_1) = \lambda V : \beta.\mathcal{J}(E_1)$ *where* β *is a fresh variable type. This is to avoid any type variable clashes inside terms.*

λ moreover, assumes the following three axioms (as we will not discuss reduction in the λ-calculus, we shall consider the axioms in terms of equality rather than reduction. Once reduction is introduced, the results below will still hold):

(α)	$\lambda V.E = \lambda V'.E[V := V']$	if V' is not free in E
(β)	$(\lambda V.E)E' = E[V := E']$	
(η)	$\lambda V.EV = E$	if V is not free in E.

Lemma 3.2 *If* $\lambda \vdash E = E'$ *then* $\lambda_L \vdash (E = E') : t$.
Proof*: By an easy induction on the derivation of* $E = E'$ *in* λ. □

Hence we have the full type free λ-calculus. Moreover, we have all the logical connectives (both propositional and quantificational). The question arises however, as to where exactly is the paradox avoided. One might wonder if the paradox is actually avoided. The reader is to be assured that this is the case. Let us start by looking at the type of the following term: $\lambda V.\neg V$. What type should this term have? Recall from our notational convention that this term is an abbreviation for something like $\lambda V : \alpha.\neg V$. The α will be unified with p and we get from equation (7) that $\neg V$ is of type p and the whole term gets type $p \to p$ from equation (4).

Can we then now find the fixed point of this term? I.e. can we find the a such that $a = \neg a$? The answer is no. We can apply $\lambda V.\neg V$ to any proposition and obtain a proposition. But once we want to apply it to the Russell's sentence, we have to make sense of the type of that sentence. But the Russell's sentence is not typeable in our system. This can be seen from the following lemma:

Lemma 3.3 $\lambda V : T \to T'.\neg VV$ *where* $T' \leq p$ *is not well-formed.*
Proof:

(i)	$V : T \to T'$	*hypothesis*
(ii)	$T \to T' \leq T$	*from* $\leq$
(iii)	$VV : T'$	*from* (3)
(iv)	$\neg VV : p$	*from* (8), *as* $T' \leq p$

But as $(T \to T') \to p$ *is circular, we cannot apply (4) to get that* $\lambda V.\neg VV$ *has type* $(T \to T') \to p$. *In fact we cannot type* $\lambda V.\neg VV$. *I.e. the type is circular.* □

It should be noted here that one can have type freeness and logic while avoiding the paradox without the use of the notion of circular types. [Kamareddine 92B] for example provides another way of avoiding the paradox.

We have built types such that all types (except the circular ones which cause the paradox) are possible. This should enable us to type all the terms that should not be problematic, that have types, but that other existsing theories cannot deal with. Moreover, it is obvious that some expressions have many types. For example, $\lambda x.x$ is of type $\alpha \to \alpha$ for any type variable α. Now let us illustrate with typing $\lambda x.xx$ and Y.

Example 3.4 *$\lambda x.xx$ has type $(\alpha_0 \to \alpha_1) \to \alpha_1$:*

(i)	$x : \alpha_0 \to \alpha_1$	*Assumption*
(ii)	$\alpha_0 \to \alpha_1 \leq \alpha_0$	*clause iii) of* $\leq$
(iii)	$x : \alpha_0$	$(i), (ii), (\ 2)$
(iv)	$xx : \alpha_1$	$(i), (iii), (\ 3)$
(v)	$\lambda x.xx : (\alpha_0 \to \alpha_1) \to \alpha_1$	$(i) \ldots (iv), (\ 4)$

Example 3.5 *$\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ has type $(\alpha_2 \to \alpha_2) \to \alpha_2$:*

(i)	$f : \alpha_2 \to \alpha_2$	*assumption*
(ii)	$x : (\alpha_1 \to \alpha_2) \to \alpha_2$	*assumption*
(iii)	$(\alpha_1 \to \alpha_2) \to \alpha_2 \leq \alpha_1 \to \alpha_2$	*clause iii) of* $\leq$
(iv)	$x : \alpha_1 \to \alpha_2$	$(ii), (iii), (\ 2)$
(v)	$xx : \alpha_2$	$(ii), (iv), (\ 3)$
(vi)	$f(xx) : \alpha_2$	$(i), (v), (\ 3)$
(vii)	$\lambda x.f(xx) : ((\alpha_1 \to \alpha_2) \to \alpha_2) \to \alpha_2$	$(ii) \ldots (vi), (\ 4)$
$(viii)$	$((\alpha_1 \to \alpha_2) \to \alpha_2) \to \alpha_2 \leq (\alpha_1 \to \alpha_2) \to \alpha_2$	*clause iii) of* $\leq$
(ix)	$\lambda x.f(xx) : (\alpha_1 \to \alpha_2) \to \alpha_2$	$(vii), (viii), (\ 2)$
(x)	$(\lambda x.f(xx))(\lambda x.f(xx)) : \alpha_2$	$(iii), (ix), (\ 3)$
(xi)	$\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) : (\alpha_2 \to \alpha_2) \to \alpha_2$	$(i) \ldots (x), (\ 4)$

Example 3.6 $(\lambda x : \alpha_0.x)y$ where $y : \alpha_1$ and α_0, α_1 are type variables, is also typable and the system will deduce that the type of $(\lambda x : \alpha_0.x)$ is $\alpha_0 \to \alpha_0$ and it will try to check and see if $\alpha_0 \leq \alpha_1$ but as α_1 is a variable, the system makes α_1 become α_0 and returns α_0 as the result. Here some work is involved in unifying these variable types and this can be found in [Kamareddine 92A].

It should be added moreover, that the theory provided in this paper has a tidy semantics which is provided in [Kamareddine, Klein 93] (excluding variable types). The models of this theory are constructed following the lines of [Aczel 80] or of [Scott 75]. Furthermore, [Kamareddine, Klein 94] provides a tree of theories where an extension of λ_L is the root and where all relevant theories of natural and programming languages are the roots of subtrees of the big tree, by showing that all the other theories are interpretable in that extension. The extension however differs from λ_L only by the addition of meta-types. Now, as meta-types have not been used in [Parsons 79] or [Chierchia, Turner 88], we can conclude that λ_L can be seen as superior to these two systems.

4 Type freeness or types

Let us recall the discussion in Section 1 where we said that the presence of the paradox led to two routes of research. The first route concentrated on logic and abandoned various forms of self-reference. The second route, abandoned logic and concentrated on self-reference. We said moreover that type theory was created under both routes. This was not without a reason of course. Moreover, the reason was not only due to the paradoxes. The fact is, type

theory provides with a powerful classification scheme which can explain the meaningfulness or senselessness of many constructs. In fact, looking at both programming and natural languages, one finds that types are indispensable. For an extensive discussion of why types are useful as a classification scheme for natural language, the reader is referred to [Chierchia, Turner 88]. In this paper however, we shall in order to complete the discussion of [Chierchia, Turner 88], ask four questions and attempt to answer them.

Question 4.1 Are types or levels necessary in the avoidance of the paradox?.
Answer Not necessarily. For example, ZF was another solution to the paradox where we don't need to classify sets iteratively ([Boolos 71]), yet the Foundation Axiom FA was included in ZF despite the fact that it was shown that antifoundation axioms are consistent with ZF (see [Aczel 84] for such a discussion). The formulation of the Foundation Axiom FA is $(\exists x)(x \in a) \rightarrow (\exists x \in a)(\forall y \in x)\neg(y \in a)$. As a corollary of it, we do not get solutions to $x = \{x\}$, or $x = \{\{x\}\}$. Moreover, the inclusion of FA was unnecessary and it was not the responsible axiom for avoiding the paradox.

Question 4.2 Are types needed?
Answer Yes of course. The fact that we ask for the full expressive power of the type free λ-calculus does not mean that types are not needed. In fact when we ask for a type free set theory, or a set theory where the definition of a set may be impredicative, we don't go and forget completely about sets. In type free theories, one asks for the furthest expressive power, where we can live with self reference and impredicativity but without paradoxes. The better such an expressive system is, the more we are moving towards type freeness. Just it is enough to remember that up to the discovery of the paradoxes, the most ideal system was of course type free. Due to the paradoxes, helas this type free paradise had to be abandoned. Types too found an attractive place in the history of foundation and in most areas of applications of logic. For after all types help in the classification of programs, in the mixing of terms (such as a noun and a verb) and so on. And moreover they play an important role in explaining the paradoxes (if such an explanation is actually possible). For example, Girard's system F ([Girard 86]) is no less type free than Feferman's theory T_0 yet types play a valuable role in that system with respect to impredicativity. The difference between F and T_0 might be in the explicitness or implicitness of the typing scheme. Now even though one works in a type free system such as that of Feferman, one needs to introduce types such as recursive types, dependent types and the like. After all many of our proofs are for a particular collection of objects and not for all possible objects. Exactly as in set theory, intersection, union and so on are absolute necessity. Note also that a fully type free language cannot accommodate an unrestricted logic or an unrestricted β-conversion. It is also the case that Natural Language implicitly has inside it a notion of type. In fact Parsons' paper gives many insights on how natural language is implicitly typed, yet type freeness must be present to deal with self referentiality.

Question 4.3 So if types are needed why talk about type free theories? Why not ignore type freeness?
Answer. The reason is that we may not want to be inflexible from the start if we could afford to be flexible. Type free theories are very elegant and simple, so we can have a clear picture of how much we have and how is the paradox avoided. Then the detail of constructing types if followed will produce all the polymorphic higher order types that are needed. So a

lot of unnecessary details (like constructing types) are left till later which will make it easier to prove results about the strength of the system, the expressive power, completeness and so on. Also from the point of view of computation, type free theories could be regarded as first order theories and hence are computionally more tractable than typed theories. Completeness also holds for first order logics but has to be forced for higher order ones. Hence what I am arguing for is the use of type freeness followed by the construction of flexible polymorphic types. It is also the case that the self referentiality of language requires type freeness. So we can talk about a property having itself as a property. For example, the property of those things equal to themselves is equal to itself.

Question 4.4 Where does Natural Language fit between the type free and typed paradigms?
Answer. Natural language is implicitly typed in that sentences don't really carry their type with them but we do attribute types to them and to their constituents in order to make sense of certain combinations. Moreover, not only we attribute types to the constituents of a sentence to make sense of it, but many sentences, when spoken are immediately assumed to be well typed. This is an evidence that NL is implicitly typed.

5 Embedding Parsons' system into ours

Recall that the paradox was avoided by using the notion of circular types. Recall moreover Remark 2.20. Hence in our interpretation of Parsons' system, the categories and types will have to be changed accordingly. We will avoid intensions via s for the sake of clarity. What we will do is basically use the same syntax of expressions but make sure that the corresponding semantic types are not circular. Let us start by formalising the syntax of parsons' categories and semantic types (called here Pcategories and Ptypes respectively).

Definition 5.1 *(Ptypes)*
Parsons' types are defined by the following syntax:
$PT ::= e|t| < s, PT > | < PT, PT >$
We let $PT, PT', PT_1, PT_2 \ldots$ *range over Ptypes.*

Definition 5.2 *(Unlabelled Pcategories)*
The unlabelled categories used are the following:
$UC ::= Cu|Cr|Clr|Clrr$ *where*
$Cu ::= s|ADF$
$Cr ::= CN|VP|NP|ADV$
$Clr ::= V|PREP$
$Clrr ::= V$

Definition 5.3 *(Pcategories)*
The categories of Parsons are defined as follows:
$PC ::= Cu|Cr^{PT}|^{PT}Clr^{PT'}|^{PT}Clrr^{PT',PT''}$
We let $PC, PC', PC_1, PC_2, \ldots$ *range over Pcategories.*

We will define a function which rules out all the s's from a Parsons' type. This function is defined as follows:

Definition 5.4
The flattening function $ext : Ptypes \longrightarrow Ptypes$ *is defined as follows:*

- $ext(t) = t$
- $ext(e) = e$
- $ext(\langle s, PT\rangle) = ext(PT)$
- $ext(\langle PT, PT'\rangle) = \langle ext(PT), ext(PT')\rangle$ *if* $PT \neq s$.

Lemma 5.5 *ext is well defined.*
Proof: *This is easy because we never get pairs* $\langle PT, PT'\rangle$ *of the form* $\langle PT, s\rangle$. *That is, we never have to apply ext to s.* □

The function $\mathcal{I}$ below will take Ptypes into types.

Definition 5.6 *We define the function* $\mathcal{I} : Ptypes \longrightarrow Types$ *as follows:*

- $\mathcal{I}(e) = e$
- $\mathcal{I}(t) = p$
- $\mathcal{I}(\langle s, PT\rangle) = \mathcal{I}(PT)$
- $\mathcal{I}(\langle PT, PT'\rangle) = \mathcal{I}(PT) \to \mathcal{I}(PT')$

Lemma 5.7 $\mathcal{I}$ *is well defined.*
Proof: *Obvious.* □

Note that some $\mathcal{I}(PT)$ might be circular. For example $\mathcal{I}(\langle\langle e, t\rangle, t\rangle) = (e \to p) \to p$. For this reason we introduce the functions H and g. The function g will flatten the range types. This will be used inside the function H below, in order to avoid the circular types. For example, if we have the type $(e \to p) \to p$, which is circular, we look for $H((e \to p) \to p) = g((e \to p) \to p) = (e \to p) \to e$ which is not circular.

Definition 5.8 *The function* $g : Types \to Types$ *is defined as follows:*

- $g(\beta) = \beta$
- $g(T) = e$ *if* T *is basic*
- $g(T_1 \to T_2) = H(T_1) \to e$ *otherwise.*

Definition 5.9 *We define the function H:* $Types \to Types$ *as follows:*

- $H(\beta) = \beta$
- $H(T) = T$ *if* T *is basic.*
- $H(T_1 \to T_2) = H(T_1) \to H(T_2)$ *if* $T_1 \to T_2$ *is not circular*
  $H(T_1 \to T_2) = g(T_1 \to T_2)$ *otherwise*

Note that g and H are mutually recursive. Moreover, they are related by the following Lemma:

Lemma 5.10 *$g \circ H = H \circ g$.*
Proof: *By cases on Types.*

- *If T is a variable type then $g \circ H(T) = H \circ g(T) = T$.*
- *If T is a basic type then $g \circ H(T) = H \circ g(T) = e$.*
- *If $T \equiv T_1 \to T'$*
  - *Case $T_1 \to T'$ is non circular,*
    $g(H(T_1 \to T')) = g(H(T_1) \to H(T')) = H(H(T_1)) \to e$ and
    $H \circ g(T_1 \to T) = H(H(T_1) \to e) = H(H(T_1)) \to e$
  - *Case $T_1 \to T'$ is circular,*
    $g(H(T_1 \to T')) = g(g(T_1 \to T')) = g(H(T_1) \to e)) = H(H(T_1)) \to e$ and
    $H(g(T_1 \to T')) = H(H(T_1) \to e) = H(H(T_1)) \to e$

□

Lemma 5.11 *$H \circ g = g \circ g$.*
Proof: *By cases on T.*

- *If T is basic or is a variable type then obvious.*
- *$H \circ g(T \to T') = H(H(T) \to e)$ and*
  $g \circ g(T \to T') = g(H(T) \to e) = H(H(T) \to e$.

□

Lemma 5.12

- $H \circ g \neq H \circ H$
- $H \circ H \neq g \circ g$

Proof:

- $H \circ g(p) = e \neq H \circ H(p) = p$.
- $g \circ g(p) = e \neq H \circ H(p) = p$.

□

Lemma 5.13 *$H(T)$ and $g(T)$ are not circular for any T in types.*
Proof: *By induction on T in Types.*

- *If T is basic or s a variable type then obvious.*
- *If $T \equiv T_1 \to T_2$ where property holds for T_1 and T_2, then:*
  - *Case $T_1 \to T_2$ is circular*
    $H(T_1 \to T_2) = g(T_1 \to T_2) = H(T_1) \to e$ *which is not circular by IH and the definition of circular types.*
  - *Case $T_1 \to T_2$ is not circular*
    $g(T_1 \to T_2) = H(T_1) \to e$ *which is not circular by IH and the definition of circular types.*
    $H(T_1 \to T_2) = H(T_1) \to H(T_2)$. *Again, by IH, $H(T_1)$ and $H(T_2)$ are not circular by IH. Moreover, it can't be the case that $H(T_2) \leq p$ and that $H(T_1) \equiv T' \to T''$ where $T'' \leq p$, because if this was the case, we get $T_1 \to T_2$ is circular, absurd.*

□

The following Lemma is very useful. It says that once we have made sure the type is not circular (via H), then another application of H is useless. That is:

Lemma 5.14 $H \circ H = H$.
Proof: *By induction on T.*

- *If T is basic or is a variable type then obvious.*
- *Assume the property holds for T_1 and T_2 then*
  - *Case $T_1 \to T_2$ is not circular then*
    $H \circ H(T_1 \to T_2) = H(H(T_1) \to H(T_2)) =$
    $H(H(T_1)) \to H(H(T_2)) =^{IH} H(T_1) \to H(T_2) =$
    $H(T_1 \to T_2)$.
  - *Case $T_1 \to T_2$ is circular then*
    $H \circ H(T_1 \to T_2) = H(g(T_1 \to T_2)) =$
    $H(H(T_1) \to e) = H(H(T_1)) \to e =^{IH} =$
    $H(T_1) \to e = g(T_1 \to T_2) =$
    $H(T_1 \to T_2)$.

Note that we could have defined H and g so that for example $H((e \to p) \to p) = (e \to e) \to p$, but this faces two problems:

- First is that we lose all the closure properties stated in the above lemmas.
- Second, it is precisely this which makes our system superior to that of Parsons. In fact as we will see in the next section, Parsons system allows some sentences which involve polymorphic types but there are many more that he can't represent. These can be easily represented in our system.

We assume similar unlabelled syntactic categories as Parsons (as given in Definition 5.2) and let f be the function which maps the syntactic types of Parsons into his semantic ones. That is, f is defined in Table 2.7 and Tables 2.8 and 2.9 give examples of categories and their corresponding Ptypes. Our set of labelled categories will also be defined similarly to that of Parsons except that our labels are elements of $Types$ rather than of $Ptypes$. That is:

Definition 5.15 *(Categories)*
$C ::= Cu | Cr^{T} | {}^{T}Clr^{T'} | {}^{T}Clrr^{T',T''}$

In fact, categories can be defined in terms of Pcategories as follows:

Definition 5.16 *(Translating Pcategories to Categories)*
$\mathcal{C} : Pcategories \longrightarrow Categories$

- $\mathcal{C}(Cu) = Cu$
- $\mathcal{C}(Cr^{PT}) = Cr^{\mathcal{I}(PT)}$
- $\mathcal{C}({}^{PT}Clr^{PT'}) =^{\mathcal{I}(PT))} Clr^{\mathcal{I}(PT')}$
- $\mathcal{C}({}^{PT}Clrr^{PT',PT''}) =^{\mathcal{I}(PT)} Clr^{\mathcal{I}(PT'),\mathcal{I}(PT'')}$

We define f' to be our function which corresponds to Parsons' f. That is, f' takes a syntactic category and returns an element in $Types$.

f' is defined via Table 5.17. Moreover, Tables 5.18 and 5.19 show examples of the result of f'. Tables 5.17 ... 5.19 correspond to Tables 2.7 ... 2.9:

Table 5.17

Categories C	**Corresponding semantic types** $f'(C)$
s	p
CN^{τ}	$H(\tau \to p)$
VP^{τ}	$H(\tau \to p)$
NP^{τ}	$H(f'(VP^{\tau}) \to p)$
${}^{\tau_1}V^{\tau_2}$	$H(f'(NP^{\tau_2}) \to f'(VP^{\tau_1}))$
${}^{\tau_1}V^{\tau_2,\tau_3}$	$H(f'(NP^{\tau_2}) \to f'({}^{\tau_1}V^{\tau_3}))$
ADV^{τ}	$H(f'(VP^{\tau})) \to f'(VP^{\tau}))$
ADF	$p \to p$
${}^{\tau_1}PREP^{\tau_2}$	$H(f'(NP^{\tau_2}) \to f'(ADV^{\tau_1}))$

It is now easy to check that the words of fixed type of PTQA, which are listed below have the corresponding semantic types:[4]

[4]Note the semantic type corresponding to ${}^{p}\mathrm{V}^{e}$. This is because $p \to p \leq p$ and hence $((e \to e) \to p) \to (p \to p) \leq ((e \to e) \to p) \to p$ which is circular.

Table 5.18

Words of fixed type	Syn Type	Sem Type
man^e, $woman^e$, $park^e$, $fish^e$, pen^e, $unicorn^e$, $\text{-}body^e$	CN^e	$e \to p$
$fact^p$, $proposition^p$, $answer^p$	CN^p	$p \to p$
run^e, $walk^e$, $talk^e$, $rise^e$, $change^e$	VP^e	$e \to p$
$obtain^p$	VP^p	$p \to p$
$John^e$, $Mary^e$, $it_0^e, it_1^e \ldots$	NP^e	$(e \to p) \to e$
$The\ Pythagorean\ theorem^p, it_0^p, it_1^p \ldots$	NP^p	$(p \to p) \to e$
${}^e eat^e$, ${}^e date^e$	${}^e V^e$	$((e \to p) \to e) \to (e \to p)$
${}^p believe^p$, ${}^e assert^p$	${}^e V^p$	$((p \to p) \to e) \to (e \to p)$
${}^p amaze^e$	${}^p V^e$	$((e \to p) \to e) \to (p \to p)$
${}^e buy^{e,e}$	${}^e V^{e,e}$	$((e \to p) \to e) \to (((e \to p) \to e) \to (e \to p))$
${}^e tell^{e,p}$	${}^e V^{e,p}$	$((e \to p) \to e) \to (((p \to p) \to e) \to (e \to p))$
$rapidly^e$, $slowly^e$, $voluntarily^e$	ADV^e	$((e \to p) \to (e \to p))$
$necessarily$	ADF	$p \to p$
${}^e in^e$	${}^e PREP^e$	$((e \to p) \to e) \to ((e \to p) \to (e \to p))$

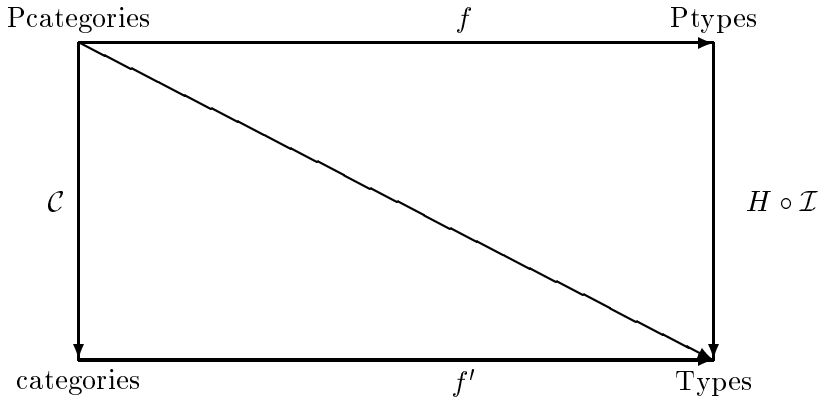
The syntactic rules of PTQA are exactly those listed in Parsons' paper. We are in the same position as Parsons in that the sentences *walks or obtains*, *Bill obtains*, *That John walks runs*, ... are ungrammatical. The formation of these sentences depends on the syntactic rule S4 and has nothing to do with the subsumption of types. The above fixed types will not accommodate polymorphism which will be able to deal with *John talks about Mary* and *John talks about a proposition*. For this we will follow Parsons in his notion of floating types.[5]

Table 5.19

Words of floating types	Type	Semantic Type
$thing^\tau$	CN^τ	$H(\tau \to p)$
$set^{\tau \to p}$	$CN^{\tau \to p}$	$H(\tau \to p) \to e$
$property^{\tau \to p}$	$CN^{\tau \to p}$	$H(\tau \to p) \to e$
$exist^\tau$	$VP^{\tau \to p}$	$H(\tau, p)$
$it_0^\tau, it_1^\tau, \ldots$	NP^τ	$H(f'(VP^\tau) \to p) = H(H(\tau \to p) \to p)$
${}^e find^\tau$, ${}^e lose^\tau$, ${}^e love^\tau$, ${}^e hate^\tau$, ${}^e seek^\tau$, ${}^e conceive^\tau$	${}^e V^\tau$	$H(f'(NP^\tau)) \to (e \to p)$
${}^\tau have^{\tau \to p}$, ${}^\tau exemplify^{\tau \to p}$	${}^\tau V^{\tau \to p}$	$H(f'(NP^{\tau \to p}) \to f'(VP^\tau))$
${}^e give^{e,\tau}$	${}^e V^{e,\tau}$	$H(f'(NP^e) \to f'({}^e V^\tau))$
${}^e about^\tau$	${}^e PREP^\tau$	$H(f'(NP^\tau) \to ((e \to p) \to (e \to p)))$
${}^{\tau_1} be^{\tau_2}$	${}^{\tau_1} V^{\tau_2}$	$H(f'(NP^{\tau_2}) \to f'(NP^{\tau_1}))$

Helas however, we still do not have f' and f related by the following equation:
$f'(\mathcal{C}(PC)) = H(\mathcal{I}(f(PC)))$.
In fact, the following diagram does not commute:
That is: $H \circ \mathcal{I} \circ f \neq f' \circ \mathcal{C}$.

[5]Note the semantic type of ${}^e V^\tau$. This is because $f(NP^\tau) \to (e \to p)$ is not circular.



Example 5.20 $H(\mathcal{I}(f({}^eV^e))) = ((e \to p) \to e) \to e$, whereas $f' \circ \mathcal{C}({}^eV^e) = ((e \to p) \to e) \to (e \to p)$.

According to this translation, the accounts of *John talks about Mary* and *John talks about a proposition* are the same as Parsons except that the values that we obtain out of the table says that the type of the sentence is e rather than p. The type however is still p as the sentence is of category s and we are consistent because $p \leq e$.

Table 5.21

Our account of John talks about Mary	*category*	*Rule*	*Type*
about	${}^ePRED^e$		$((e \to p) \to e) \to ((e \to p) \to (e \to p))$
Mary	NP^e		$(e \to p) \to e$
about Mary	ADV^e	*S6*	$(e \to p) \to (e \to p)$
talk	VP^e		$e \to p$
talk about Mary	VP^e	*S10*	$e \to p$
John	NP^e		$(e \to p) \to e$
John talks about Mary	s		e

Table 5.22

Our account of John talks about a proposition	*Category*	*Rule*	*Type*
proposition	CN^p		$p \to p$
a proposition	NP^p	*S2*	$(p \to p) \to e$
about	${}^ePRED^p$		$((p \to p) \to e) \to ((e \to p) \to (e \to p))$
about a proposition	ADV^e	*S6*	$(e \to p) \to (e \to p)$
talk	VP^e		$e \to p$
talk about a proposition	VP^e	*S10*	$e \to p$
John	NP^e		$(e \to p) \to e$
John talks about a propostion	s		e

Also, like him this approach captures that *a property runs* is ungrammatical. Up to here, all Parsons framework is accommodated in a type free theory with logic and where the paradoxes are avoided via circular types. In order to give Parsons' framework an interpretation in this type free theory, we kept exactly the same syntax and syntatctic categories, yet we changed the semantic domains. This is because for type free λ-calculus, to have logic inside it, there must be a way to avoid the paraodxes.

6 Paradoxical sentences, Parsons approach and usefulness of λ_L

Parsons' approach is very attractive and explains in an elegant way the grammaticality or ungrammaticality of sentences. For example, we can say that *john runs* but not that *a property runs*. The problem that we find with his approach is its limitation in terms of self reference. For example, Parsons approach rules out sentences such as a property has itself. In fact the following examples which are an implementation of the theory of types in which we implemented Parsons system ([Kamareddine 92A]) will give a feel of how the system works:

	Expressions	**Types**
1	$\lambda x.x$	$\beta_0 \to \beta_0$
2	$\lambda x:e.x$	$e \to e$
3	$\lambda x.xx$	$(\beta_0 \to \beta_1) \to \beta_1$
4	$(\lambda x.xx)(\lambda x.xx)$	β_1
5	$\lambda x:p.xx$	$p \to \beta_0$
6	$\lambda x:e \to p.xx$	error: $(e \to p) \to p$ is circular
7	$\forall x:(\beta_0 \to \beta_1).xy$	p
8	$\forall x:e.x$	error, not a proposition
9	$\forall x:(e \to \beta_1).xy$	p
10	$\forall x.xx$	p
11	$\lambda x:(\beta_0 \to \beta_1).xy$	$(\beta_0 \to \beta_1) \to \beta_1$
12	$\lambda f.(\lambda s:e \to pf(ss))(\lambda s:e \to pf(ss))$	error: $(p \to p) \to p$ is circular
13	$\lambda f:e \to p.(\lambda s:e \to pf(ss))(\lambda s:e \to pf(ss))$	error: $(e \to p) \to p$ is circular
14	$\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$	$(\beta_2 \to \beta_2) \to \beta_2$
15	$(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))(\lambda x:p.xx)$	p
16	$(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))$	β_2
17	$(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))(\lambda x.xx)$	β_2
18	$(\lambda x.xx)(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))$	β_1
19	$\lambda x.\neg xx$	error, circular type
20	$\lambda x:(\beta_0 \to t).\neg xx$	error, circular type
21	$\lambda x:(\beta_0 \to p).\neg xx$	error, circular type
22	$\lambda x.xx \to \bot$	error, circular type

Example 6 shows that Parsons' system cannot have a property which holds of itself. Example 12 shows that he can't find the fixed points of properties. Example 11 shows that he can have *everything holds of everything* and so on. Of course we can solve the problem of a property holds of itself. For example the first thing that we can do is take *have* to have for syntactic type ${}^{\tau}\text{have}^{\tau}$, and to have semantic type ${}^{\tau}V^{\tau}$. This will also enable him to say *John has a letter*, *John has Mary* and so on. This idea however would have to be

carried out very carefully because syntactically this is how Parsons avoided the paradox. In our language however, syntactically anything is acceptable because we are working in a type free framework. Hence it is semantically that we have to explain the meaning of a property having itself or not having itself. Assume here that we change *have* to the following, then our generation of *a property has itself* and of *a property not having itself* are as follows:

property	$\mathrm{CN}^{\tau\to p}$	$H(\tau\to p)\to e$
a property	$\mathrm{NP}^{\tau\to p}$	$H((\tau\to p)\to p)\to e = (H(\tau\to p)\to e)\to e$
$^{\tau\to p}\mathrm{has}^{\tau\to p}$	$^{\tau\to p}\mathrm{V}^{\tau\to p}$	
itself	$\mathrm{NP}^{\tau\to p}$	
has itself	$\mathrm{VP}^{\tau\to p}$	
a property has itself	s	e

A property does not have itself is dealt with by adding the syntactic clause for *not*. If β is in $\mathrm{VP}^{\tau\to p}$ then *not* β is in $\mathrm{VP}^{\tau\to p}$.

It must be noted here that parsons' system is much weaker than that described by listing the 22 examples above. In fact, parsons' system is not capable of typechecking term 14 above (which is the fixed point operator). In fact, we have improved Parsons' system by allowing it to accommodate and type check many sentences that it could not do originally. Even more, we don't have the limitation of Parsons' system. That is a property can apply to itself in our system. It is not without a reason that the negation operator accepts objects and returns objects rather than just accepting propositions and returning propositions. If we allowed the latter, we will fall foul of the paradox. For example, $\lambda x:(e\to p).\neg xx$ applied to itself gives that a proposition is equal to its negation. According to our approach however, $\lambda x(e\to e).\neg xx$ applied to itself will be equal to its negation. This however, will not result in a paradox, because it is not obvious how to show that the result is a proposition. So in summary, for the non problematic sentences, we get propostions but for the probelmatic ones, we restrict the types to those non circular via the function H.

7 Comparison and Conclusion

From the previous section, we have improved a lot in the expressivity of Parsons' system by allowing him to talk about sentences that he could not talk about previously. Even more, we said that with our flexible typing scheme, we can allow any sentence and type check it as long as its type was not circular. If the type is circular, we change the final type of the sentence so that a paradox is impossible to derive. This approach is certainly flexible. Furthermore, all the type free λ-calculus is accommodated in this approach, all self reference and all logic. Let us now complete the comparison that we started in the previous section by remarking on the differences between our system and that of [Chierchia, Turner 88].

There is a broad correspondence between our type '$\langle e,p\rangle$' and the sort 'nf' of the Chierchia and Turner paper, and to this extent the two fragments are quite similar. However, [Chierchia, Turner 88]'s semantic domain D_{nf} is the nominalization of all functions from e to e, rather than those from e to p.

Second, for Chierchia and Turner, only expressions of type nf are nominals. Since their nominalization operator is exclusively defined for expressions of type $\langle e,e\rangle$, and they do not have any kind of type containment for functional types, they do not allow transitive verbs like **love** and ditransitives like **give** to be nominalised. Yet examples such as 1 (from [Parsons 79])

and 2 in the example below, show that untensed transitive verbs enter into the same nominal patterns as intransitives:

Example 7.1

1. To love is to exalt.
2. To give is better than to receive.

By contrast, our approach can accommodate such data straightforwardly.

Third, recall that Turner abandoned the comprehension principle. Now the abandonment of Frege's full comprehension axiom will impose the use of two logics, one inside the predication operator in addition to the usual one for wffs. This is due to the fact that breaking the equivalence between $p(\lambda x.A, t)$ and $A(t, x)$ will disconnect the reasoning about wffs and properties.

We have argued in this paper that Natural Language items cannot be rigidly typed and that if we start from the type free λ-calulus, we can flexibly type natural language terms. That is anything is an expression and anything non problematic will have a type. These types are polymorphic in the sense that expressions can have variable types and these variable types may be instantiated to anything. For example, the identity function has type $\beta_0 \rightarrow \beta_0$, and the identity function applied to of type e will result in elements of type e. The polymorphic power of the system comes from the ability to typecheck all polymorphic functions even those which are problematic in other systems. For example the fixed point operator, $\mathrm{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ is typechecked to $(\beta_2 \rightarrow \beta_2) \rightarrow \beta_2$ and even can apply to itself. Even YY is typechecked to β_2. $\omega = \lambda x.xx$ is also typechecked to $(\beta_1 \rightarrow \beta_1) \rightarrow \beta_1$ and ω applied to itself is typechecked to β_1 . As said earlier, these types can be instantiated so that YI where I is the identity function over e (i.e. $\mathrm{I} = \lambda x : e.x$), is typechecked to e naturally. We believe this system is one of the first which can typecheck all the above while remaining a very expressive and simple one. Another nice characteristic of the system is its ability to combine logic and the type free λ-calculus while remaining consistent. So even though the Russell sentence $(\lambda x.\neg(xx))$ is a well formed sentence of the system, its type cannot be found. In fact, the system returns an error message explaining that this sentence has a circular type. The same thing applies to Curry's sentence $(\lambda x.xx \rightarrow \bot)$. Finally, the typing scheme that we presented can have a wide range of applications (see [Kamareddine, Klein 93], [Kamareddine, Klein 94] and [Kamareddine 94]). The reason being that even though types are very informative either in programming or in natural languages, type freeness and the non-restricted typing schemes are a necessity in interpreting many natural and programming language constructs. We believe in the need to have your cake and eat it in the disciplines of programming and natural languages. That is, we believe it necessary not to be too scared of the paradoxes to the point of using too restricted languages. We must have the courage to touch as much as we can the boundary of logic and type freeness between safety and danger.

References

[Aczel 80] Aczel, P., Frege structures and the notions of truth and proposition, *Kleene Symposium*, 1980.

[Aczel 84] Aczel, P., *Non well founded sets*, CSLI lecture notes, No 14, 1984.

[Barendregt, Hemerik 90] Barendregt, H., and Hemerik, C., Types in Lambda calcului and programming languages, in: European Symposium on programming, ed. N. Jones, *Lecture notes in Computer Science 423*, Springer, pp. 1-36, 1990.

[Boolos 71] Boolos, g., The iterative conception of sets, *Journal of Philosophy LXVIII*, pp 215-231, 1971.

[Chierchia, Turner 88] Chierchia, and Turner, R., Semantics and property theory, *Linguistics and Philosophy 11*, pp 261-302, 1988.

[Cocchiarella 84] N. Cocchiarella, Frege's Double Correlation Thesis and Quine's set theories NF and ML, *Journal of Philosophical Logic 14*, pp. 1-39, 1984.

[Feferman 79] Feferman, S., Constructive theories of functions and classes, *Logic Colloquium '78*, M. Boffa et al (eds), pp 159-224, North Holland, 1979.

[Feferman 84] Feferman, S., Towards useful type free theories I, *Journal of Symbolic logic 49*, pp 75-111, 1984.

[Girard 86] Girard, J.Y., The system F of variable types, fifteen years later, *Theoretical Computer Science 45*, pp 159- 192, North-Holland, 1986.

[Kamareddine 89] Kamareddine, F., *Semantics in a Frege structure*, PhD thesis, University of Edinburgh, 1989.

[Kamareddine 92A] Kamareddine, F., A system at the cross roads of logic and functional programming, *Science of Computer Programming 19*, pp. 239-279, 1992.

[Kamareddine 92B] Kamareddine, F., λ-terms, logic, determiners and quantifiers, *Journal of Logic, Language and Information*, Volume 1, No 1, pp 79-103, 1992.

[Kamareddine 92C] Kamareddine, F., Set Theory and Nominalisation, Part I, *Journal of Logic and Computation*, Volume 2, No 5, pp. 579-604, 1992.

[Kamareddine 92D] Kamareddine, F., Set Theory and Nominalisation, Part II, *Journal of Logic and Computation*, Volume 2, No 6, pp 687-707, 1992.

[Kamareddine, Klein 93] Kamareddine, F., and Klein, E., Polymorphism, Type containment and Nominalisation, *Journal of Logic, Language and Information 2*, pp 171-215, 1993.

[Kamareddine, Nederpelt 93] Kamareddine, F., and Nederpelt, R.P., On Stepwise explicit substitution, *International Journal of Foundations of Computer Science 4 (3)*, 197-240, 1993.

[Kamareddine, Klein 94] Kamareddine, F., and Klein, E., Polymorphism and Logic in Natural and Programming Languages, submitted for publication.

[Kamareddine 94] Kamareddine, F., Non well-typedness and Type-freeness can unify the interpretation of functional application, to appear in the *Journal of Logic, Language and Information*, 1994.

[Kamareddine, Nederpelt 94A] Kamareddine, F., and Nederpelt, R.P., A unified approach to Type Theory through a refined λ-calculus, Proceedings of the 1992 conference on Mathematical Foundations of Programming Langauge Semantics, edited by Michael Mislove et al, 1994.

[Kamareddine, Nederpelt 94B] Kamareddine, F., and Nederpelt, R.P., *The beauty of the λ-calculus*, to appear.

[Martin-Löf 73] Martin-Löf, P., An intuitionistic theory of types: predicative part, *logic colloquium '73* , Rose and Shepherdson (eds), North Holland, 1973.

[Milner 78] Milner, R., A theory of type polymorphism in programming, *Journal of Computer and System Sciences*, Volume 17, No 3, 1978.

[Parsons 79] Parsons, T., Type Theory and Natural Language, *Linguistics, Philosophy and Montague grammar*, S Davis and M Mithum (eds), University of Texas press, pp 127-151, 1979.

[Poincaré 1900] H. Poincaré, Du role de l'intuition et de la logique en mathematiques, C.R. du II Congr. Intern. des Math., pp. 200-202, 1900.

[Russell 1908] B. Russell, Mathematical logic as based on the theory of types, *American Journal of of Math. 30*, pp. 222-262, 1908.

[Scott 75] Scott, D., Combinators and classes, in *Lambda Calculus and Computer Science*, Lecture Notes in Computer Science 37, Böhm (ed), Springer, Berlin, pp 1-26, 1975.

[Turner 84] Turner, R., Three Theories of Nominalized Predicates, *Studia Logica XLIV2*, pp. 165-186, 1984.

[Turner 87] Turner, R., A Theory of properties *Journal of Symbolic Logic 52*, pp. 63-89, 1987.