Important Issues in Foundational Formalisms

FAIROUZ KAMAREDDINE, *Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, UK. E-mail: fairouz@dcs.gla.ac.uk*

Abstract

This article discusses my work in the last few years on logical formalisms which have been shown to be useful to various aspects of Natural and Programming Languages and for foundational formalisms. In this period, I have been involved in two extensive programs:

1. The first program concerns languages which exhibit various ways of combining *expressiveness* with *logic*. While I do not propose that any of these languages is ideal, I believe that they illustrate the fruitfulness of bringing together ideas from distinct disciplines. Central to the program will be *Logic*, *λ-calculus* and *Type Theory*, which have played an important role not only in foundational discussions, but also in applied formal semantics; specifically, the semantics of natural language (NL) and of programming languages (PL). The general goal here has been to find expressive and unifying theories which keep the earlier advantages but bring about new dimensions. This goal moreover extends to finding a general framework which can be used to compare earlier theories and to carry results from one theory to another without duplication of work. Issues that play a great role in the general framework include full expressiveness and logic, intensionality versus extensionality, polymorphism, internal definability of determiners and quantifiers, fixed point operators and self-application, avoidance of Russell's and Curry's paradoxes, and property and truth theories. This program comes from my ongoing work at and collaboration with Edinburgh where the stimulating environment of logic, foundation and language has been invaluable.
2. From the point of view of notation and language, I aimed at studying typed and type-free formalisms and at investigating a λ-notation which can be used to generalise the various existing type theories, and to improve the computational power of the λ-calculus by making substitution explicit, by refining reduction, by introducing definitions to the syntax and by rewriting terms so that their reduction can be made more efficient. This work comes from my ongoing collaboration with Eindhoven, the centre of AUTOMATH, the theorem prover which have inspired much research on language and formalisms.

I hope that by studying general formalisms from the outside and by going inside and unpacking the notation, one can gain even much further insights than achieved so far.

1 Full expressiveness and logic

As is well-known, combining full type-free λ-calculus with logic leads to contradiction. The reason is that one can use $R \equiv \lambda_x.\neg xx$ to derive $RR = \neg(RR)$. This is known as Russell's paradox. Even when Russell's paradox is avoided, one will still get a problem when one tries to discuss the axioms and rules of the logic that is being used. To be more precise, in any logic, one must state which of the following five concepts hold:

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

- Equality (=): If $E = E'$ and $\Gamma \vdash E$ then $\Gamma \vdash E'$.
- Start: If $E \in \Gamma$ then $\Gamma \vdash E$.

Usually, in logic we assume all the above five concepts. If we do this however in a theory which has our above syntax, then we get in trouble as follows:

Let $\perp$ be an equality between two terms that can't possibly be equal, such as $true = false$,

Let $a = \lambda x.(xx \supset \perp)$. The following is a proof of $\perp$.

1	$\{aa\} = aa \supset \perp$	(β)
2	$\{aa\} \vdash aa$	Start
3	$\{aa\} \vdash aa \supset \perp$	$1 + 2 + (=)$
4	$\{aa\} \vdash \perp$	$MP + 2 + 3$
5	$\vdash aa \supset \perp$	$DT + 4$
6	$\vdash aa$	$1 + 5 + (=)$
7	$\vdash \perp$	$MP + 5 + 6$

Hence Type Theory stepped in and played an important role in applied formal semantics. In natural language semantics, where logic has been of more concern than expressiveness, restrictive typing systems have been the norm. By contrast, programming language semantics has tended to focus on expressiveness (and functions) rather than logic, and hence depended on a less restrictive, or polymorphic, type systems. For example, Milner's functional language ML in [35] used a polymorphic type theory (Curry's $\lambda_{\rightarrow}$ system).

In Type Theory there are attempts at unifying the various formalisms (see [5], [3] and [25]) so that results can be carried across theories without duplication of work. It is moreover elegant to have unique formulations of Type Theories. After all, such unification will help to rid of the anarchy present as a result of so many different formulations. In fact, the presence of the paradoxes lead to the emergence of many Type Theories which vary in how they combine logic and expressiveness. Each such theory has been used for some applications, yet the need has come to extend results from one application to another. Hence, it is important to represent type theories in one unique framework. This is difficult as the framework which formulates one theory may be incompatible with the framework of the other. Unifying theories will result in elegant formulations, in man-effort saving (as work will no longer be duplicated) and in giving insight into the relations of one theory to another. This can improve the applications of the various existing theories. We shall in this section describe the theory T_Ω presented in [22] which acts as a unifying fomalisms for various existing formalisms of natural and programming languages. For further details, the reader is referred to [22].

1.1 The system T_Ω

T_Ω was presented in [22] and was shown to be an extension of various other systems. We will follow the line of Barendregt in [3], in constructing a tree which will have T_Ω at its top. All the other systems have been shown to have useful applications related to natural and programming languages. [36] for example, presents a polymorphic system which can accommodate self-referential terms. [37] presents a system based on linear logic but which attempts to add logical features to functional programming.

[6] presents a type free theory and interprets a fragment of English in it. [15] provides powerful tools for the formalisation of quantifiers and determiners, whereas [14] extends Milner's language ML (see [34]) with higher order polymorphism and logic. [14] moreover can be extended in a simple way to provide a type checker for the present system. The system of [18] interprets an extended fragment of natural language where self-reference and nominalisation are allowed. Martin-Löf's type theory moreover in [33] and Feferman's T_0 in [8], present systems which have been extensively used in PL. Those systems too are related to the present one as we shall see below. T_Ω accommodates most of the systems mentioned above, hence bringing in all the advantages. We will use σ and τ for types (two special instances of which are e the type of objects, p the type of propositions and t the type of truths), μ for metatypes and η, η_1, η_2 to range over both types and metatypes. (The reason why we use types and metatypes is related to avoiding the paradoxes. This is explained briefly below and in detail in [18].) We use c for constants (a special instance of which is $\bot$), x, y for expression variables, v for type variables drawn from a countable set V. (We assume that type variables and expression variables are disjoint.) α, β are used for arbitrary object language expressions and φ, ψ, χ for expressions which denote propositions. We use $\Gamma \vdash s$ to mean that s is derivable within context Γ, and $\Gamma \vdash_\Sigma s$ to mean that s is derivable from the signature Σ within context Γ. $\vdash s$ and $\vdash_\Sigma s$ stand respectively for $\emptyset \vdash s$ and $\emptyset \vdash_\Sigma s$, where $\emptyset$ is the empty context. The syntax of the various sorts of expression can now be specified as follows:

Signatures	Σ	$::=$	$\emptyset \mid \Sigma, c{:}\eta$
Contexts	Γ	$::=$	$\emptyset \mid \Gamma, x{:}\sigma \mid \Gamma, \alpha{:}\ t$
Kinds	K	$::=$	$\mathit{type} \mid \mathit{c\text{-}type} \mid \mathit{metatype}$
Types	σ	$::=$	$v \mid e \mid t \mid p \mid \langle \sigma, \tau \rangle$
Metatypes	μ	$::=$	$(\eta_1 \to \eta_2)$
Expressions	α	$::=$	$c \mid x \mid \lambda x{:}\sigma.\alpha \mid \mathit{app}(\alpha, \beta) \mid \Omega\alpha \mid {}^\cup\alpha \mid \alpha(\beta) \mid \neg\alpha \mid [\alpha \wedge \beta]$
			$\mid [\alpha \vee \beta] \mid [\alpha \supset \beta] \mid [\alpha = \beta] \mid \forall x{:}\sigma.\alpha \mid \exists x{:}\sigma.\alpha$

All the above expressions α should be obvious except for $\Omega\alpha$ and ${}^\cup\alpha$ which we explain as follows. $\Omega\alpha$ is to be understood as saying that α is a proposition. In [15], it was needed to make the construction of logic inside the type free λ-calculus non paradoxical. Although the system in this paper is in fact typed, we will see in Section 1.1.3 that it contains the system T_H of [15] which is type free. Moreover, the typing system will avoid Russell's paradox with the help of the notion of circular types which will be defined further on in this section. So it might seem that Ω is only cosmetic. This is not so however. We will below define a property theory and this will need the Ω operator. There will of course be a relation between the two ways of avoiding the paradox (i.e. the one presented here and that presented in [15]). It may be questioned why we have three kinds: types, ctypes and metatypes. In particular, why we need both types and metatypes and what is the difference between $\langle \sigma, \tau \rangle$ and $(\sigma \to \tau)$. First, for types we have the non-problematic types as $\langle e, e \rangle$ and $\langle e, p \rangle$ and the circular types which lead to the problems as $\langle \langle e, p \rangle, p \rangle$ (see the rule: *c-type base*). Abstraction over these circular types is what leads to paradoxes. For example, if $R \equiv \lambda_{x:<e,p>}.\neg(xx)$ then $RR = \neg(RR)$ and RR is of type p, which is a contradiction. This is the reason why in the rule (λ) in definition 1.2, we forbid abstracting over circular types. Hence, as $<< e, p >, p >$ is circular, R above is not allowed. As for metatypes, they play

another role. Namely, they give the type of a lifting function. Here, let us recall that we live in a domain where all types are subsumed by e. Hence, for example, an expression of type $\langle e, e\rangle$ is also an expression of type e. Hence, we are taking a similar line to that of Bealer in [4] where everything is an object and *app* applies an object to another. Now, sometimes we need to lift an object to a real function, so that if the object α was of type $\langle e, e\rangle$, the function $^{\cup}\alpha$ will be of metatype $e \to e$, and where $(^{\cup}\alpha)(\beta) = app(\alpha, \beta)$. This is necessary when we want to claim that our approach doesn't restrict itself to Bealer's claim that one only needs objects, nor to Chierchia's claim that one needs both objects and functions. We have a flexible account where one can use the one or the other. Moreover, we have two ways of forming predicative expressions: via λ (which forms a predicative expression denoting an object) and via $^{\cup}$ (which forms a predicative expression denoting a function). There are also two ways of applying a predicative expression to an argument: via *app*, which takes a predicative expression denoting an object and via real functional application, which takes a predicative expression denoting a function. These two ways of forming predicative expressions and of saturating them are related, as we shall see below.

Judgements

$\vdash \Sigma$ *sig*	Σ *is a signature*
$\vdash_\Sigma \Gamma$ *context*	Γ *is a context*
$\Gamma\vdash_\Sigma \sigma\ K$	σ *has kind* K
$\Gamma\vdash_\Sigma \sigma\preceq\tau$	*type* σ *is contained in type* τ
$\Gamma\vdash_\Sigma \sigma\approx\tau$	*type* σ *is equivalent to type* τ
$\Gamma\vdash_\Sigma \alpha{:}\sigma$	α *has type* σ

Note that the $\approx$ relation between types is the symmetric closure of $\preceq$ below.

Valid Signature

$$(null\ \ sig\ ) \qquad \frac{}{\vdash \emptyset\ \ sig}$$

The empty relation is a signature.

$$(:sig) \qquad \frac{\vdash \Sigma\ sig \qquad \vdash_\Sigma \sigma\ K}{\vdash \Sigma, c{:}\sigma\ sig}\ \ if\ c \notin\ dom\ (\Sigma)$$

Valid Context

$$(null\ \ context\ ) \qquad \frac{\vdash \Sigma\ sig}{\vdash_\Sigma \emptyset\ \ context}$$

$$(:context) \qquad \frac{\vdash_\Sigma \Gamma\ context \qquad \Gamma\vdash_\Sigma \sigma\ type}{\vdash_\Sigma \Gamma, x{:}\sigma\ context}\ \ if\ x \notin\ dom\ (\Gamma)$$

$$(:truthcontext) \qquad \frac{\vdash_\Sigma \Gamma\ context}{\vdash_\Sigma \Gamma, \varphi{:}t\ context}\ \ if\ \varphi \notin\ dom\ (\Gamma)$$

There is a *containment* relation $\preceq$ (in fact, a partial order) which is imposed on the types. When $\sigma\preceq\tau$, we say that σ is *contained in*, or is a *subtype of*, τ. $\sigma\preceq\tau$ means that any expression which is of type σ is also of type τ; moreover, any object in the

model which belongs to the domain D_σ associated with σ also belongs to the domain D_τ associated with τ. The most salient containments in our system are the following:

$$\begin{array}{llll} t & \preceq & p & \preceq \quad e \\ \sigma & \preceq & e & \\ \langle \sigma, \tau \rangle & \preceq & \sigma & \end{array}$$

We assume here three notions of kinds: types such as $\langle e, p \rangle$, metatypes such as $(e \to p)$ and c-types (or circular types) as in $\langle \langle e, p \rangle, p \rangle$. It is basically the c-types which cause the paradox and therefore we will restrict abstraction to the non circular types:

Kinds, Types and Metatypes

$$(\textit{base types}) \qquad \frac{\vdash_\Sigma \Gamma \ \textit{context} \qquad v \in V}{\Gamma \vdash_\Sigma v \ \textit{type}}$$

$$\frac{\vdash_\Sigma \Gamma \ \textit{context}}{\Gamma \vdash_\Sigma e \ \textit{type}}$$

$$\frac{\vdash_\Sigma \Gamma \ \textit{context}}{\Gamma \vdash_\Sigma t \ \textit{type}}$$

$$\frac{\vdash_\Sigma \Gamma \ \textit{context}}{\Gamma \vdash_\Sigma p \ \textit{type}}$$

$$(\textit{complex types}) \qquad \frac{\Gamma \vdash_\Sigma \sigma \ \textit{type} \qquad \Gamma \vdash_\Sigma \tau \ \textit{type}}{\Gamma \vdash_\Sigma \langle \sigma, \tau \rangle \ \textit{type}}$$

$$(\textit{c-types base}) \qquad \frac{\Gamma \vdash_\Sigma \sigma \ \textit{type} \qquad \Gamma \vdash_\Sigma \tau \preceq p \qquad \Gamma \vdash_\Sigma \rho \preceq p}{\Gamma \vdash_\Sigma \langle \langle \sigma, \tau \rangle, \rho \rangle \ \textit{c-type}}$$

$$(\textit{c-types } 1) \qquad \frac{\Gamma \vdash_\Sigma \tau \ \textit{c-type} \qquad \Gamma \vdash_\Sigma \sigma \ \textit{type}}{\Gamma \vdash_\Sigma \langle \sigma, \tau \rangle \ \textit{c-type}}$$

$$(\textit{c-types } 2) \qquad \frac{\Gamma \vdash_\Sigma \tau \ \textit{c-type} \qquad \Gamma \vdash_\Sigma \sigma \ \textit{type}}{\Gamma \vdash_\Sigma \langle \tau, \sigma \rangle \ \textit{c-type}}$$

LEMMA 1.1
If $\langle \tau, \sigma \rangle$ is not a c-type, then neither τ nor σ is a c-type.

$\preceq$, which plays an important role in polymorphism is governed by the following conditions:

Containment

$$(e \preceq) \qquad \frac{\Gamma \vdash_\Sigma \sigma \ \textit{type}}{\Gamma \vdash_\Sigma \sigma \preceq e}$$

$$(p \preceq) \qquad \frac{\vdash_\Sigma \Gamma \ \textit{context}}{\Gamma \vdash_\Sigma t \preceq p}$$

$$(\textit{Dom} \preceq) \qquad \frac{\Gamma \vdash_\Sigma \sigma_1 \ \textit{type} \qquad \Gamma \vdash_\Sigma \sigma_2 \ \textit{type}}{\Gamma \vdash_\Sigma \langle \sigma_1, \sigma_2 \rangle \preceq \sigma_1}$$

$$(Ran\preceq)\quad \frac{\Gamma\vdash_\Sigma \sigma\ type \qquad \Gamma\vdash_\Sigma \tau_1\preceq\tau_2}{\Gamma\vdash_\Sigma \langle\sigma,\tau_1\rangle\preceq\langle\sigma,\tau_2\rangle}$$

$$(Id\preceq)\quad \frac{\Gamma\vdash_\Sigma \sigma\ type}{\Gamma\vdash_\Sigma \sigma\preceq\sigma}$$

$$(Trans\preceq)\quad \frac{\Gamma\vdash_\Sigma \sigma\preceq\tau \qquad \Gamma\vdash_\Sigma \tau\preceq\rho}{\Gamma\vdash_\Sigma \sigma\preceq\rho}$$

$$(Anti\preceq)\quad \frac{\Gamma\vdash_\Sigma \sigma\preceq\tau \qquad \Gamma\vdash_\Sigma \tau\preceq\sigma}{\Gamma\vdash_\Sigma \sigma\approx\tau}$$

DEFINITION 1.2 (TYPE INFERENCE FOR T_Ω)

$$(Base)\quad \frac{\vdash_\Sigma \Gamma \text{ context}}{\Gamma\vdash_\Sigma \alpha{:}\sigma} \text{ where } \alpha{:}\sigma\in\Gamma$$

$$(Contain)\quad \frac{\Gamma\vdash_\Sigma \sigma\preceq\tau \qquad \Gamma\vdash_\Sigma \alpha{:}\sigma}{\Gamma\vdash_\Sigma \alpha{:}\tau}$$

$$(\lambda)\quad \frac{\Gamma, x{:}\sigma\vdash_\Sigma \alpha{:}\tau \qquad \Gamma\not\vdash_\Sigma \langle\sigma,\tau\rangle \text{ c-type}}{\Gamma\vdash_\Sigma (\lambda x{:}\sigma.\alpha){:}\langle\sigma,\tau\rangle}$$

$$(app)\quad \frac{\Gamma\vdash_\Sigma \alpha{:}\langle\sigma,\tau\rangle \qquad \Gamma\vdash_\Sigma \beta{:}\sigma}{\Gamma\vdash_\Sigma app(\alpha,\beta){:}\tau}$$

$$(Funct)\quad \frac{\Gamma\vdash_\Sigma \alpha{:}(\sigma\to\tau) \qquad \Gamma\vdash_\Sigma \beta{:}\sigma}{\Gamma\vdash_\Sigma \alpha(\beta){:}\tau}$$

$$(^\cup I)\quad \frac{\Gamma\vdash_\Sigma \alpha{:}\langle e,\sigma\rangle}{\Gamma\vdash_\Sigma {}^\cup\alpha{:}(e\to\sigma)} \text{ where } \sigma \text{ is } p \text{ or } e$$

$$(\Omega I)\quad \frac{\Gamma\vdash_\Sigma \varphi{:}p}{\Gamma\vdash_\Sigma \Omega\varphi{:}t}$$

$$(\Omega E)\quad \frac{\Gamma\vdash_\Sigma \Omega\varphi : t}{\Gamma\vdash_\Sigma \varphi : p}$$

$$(=prop)\quad \frac{\Gamma\vdash_\Sigma \alpha{:}\sigma \qquad \Gamma\vdash_\Sigma \beta{:}\sigma'}{\Gamma\vdash_\Sigma [\alpha=\beta]{:}p} \text{(note } \sigma \text{ and } \sigma'. \text{ This gives more propositions)}$$

$$(=E)\quad \frac{\Gamma\vdash_\Sigma [\alpha=\beta]{:}t \qquad \Gamma\vdash_\Sigma \alpha{:}\sigma}{\Gamma\vdash_\Sigma \beta{:}\sigma}$$

$$(\neg prop) \qquad \frac{\Gamma \vdash_\Sigma \varphi{:}p}{\Gamma \vdash_\Sigma \neg\varphi{:}p}$$

$$(\neg I) \qquad \frac{\Gamma \vdash_\Sigma \varphi{:}p \qquad \Gamma, \varphi{:}t \vdash_\Sigma \bot{:}t}{\Gamma \vdash_\Sigma \neg\varphi{:}t}$$

$$(\neg E) \qquad \frac{\Gamma, \neg\varphi{:}t \vdash_\Sigma \bot{:}t \qquad \Gamma \vdash_\Sigma \varphi{:}p}{\Gamma \vdash_\Sigma \varphi{:}t}$$

$$(\wedge prop) \qquad \frac{\Gamma \vdash_\Sigma \varphi{:}p \qquad \Gamma \vdash_\Sigma \psi{:}p}{\Gamma \vdash_\Sigma [\varphi \wedge \psi]{:}p}$$

$$(\wedge I) \qquad \frac{\Gamma \vdash_\Sigma \varphi{:}t \qquad \Gamma \vdash_\Sigma \psi{:}t}{\Gamma \vdash_\Sigma [\varphi \wedge \psi]{:}t}$$

$$(\wedge E) \qquad \frac{\Gamma \vdash_\Sigma [\varphi \wedge \psi]{:}t}{\Gamma \vdash_\Sigma \varphi{:}t} \qquad \frac{\Gamma \vdash_\Sigma [\varphi \wedge \psi]{:}t}{\Gamma \vdash_\Sigma \psi{:}t}$$

$$(\vee prop) \qquad \frac{\Gamma \vdash_\Sigma \varphi{:}p \qquad \Gamma \vdash_\Sigma \psi{:}p}{\Gamma \vdash_\Sigma [\varphi \vee \psi]{:}p}$$

$$(\vee I) \qquad \frac{\Gamma \vdash_\Sigma \varphi{:}t \qquad \Gamma \vdash_\Sigma \psi{:}p}{\Gamma \vdash_\Sigma [\varphi \vee \psi]{:}t} \qquad \frac{\Gamma \vdash_\Sigma \varphi{:}p \qquad \Gamma \vdash_\Sigma \psi{:}t}{\Gamma \vdash_\Sigma [\varphi \vee \psi]{:}t}$$

$$(\vee E) \qquad \frac{\Gamma, \varphi{:}t \vdash_\Sigma \chi{:}t \qquad \Gamma, \psi{:}t \vdash_\Sigma \chi{:}t \qquad \Gamma \vdash_\Sigma [\varphi \vee \psi]{:}t}{\Gamma \vdash_\Sigma \chi{:}t}$$

$$(\supset prop) \qquad \frac{\Gamma \vdash_\Sigma \psi{:}p \qquad \Gamma \vdash_\Sigma \varphi{:}p}{\Gamma \vdash_\Sigma [\varphi \supset \psi]{:}p}$$

$$(\supset I) \qquad \frac{\Gamma, \varphi{:}t \vdash_\Sigma \psi{:}t \qquad \Gamma \vdash_\Sigma \varphi{:}p}{\Gamma \vdash_\Sigma [\varphi \supset \psi]{:}t}$$

$$(\supset E) \qquad \frac{\Gamma \vdash_\Sigma \varphi{:}t \qquad \Gamma \vdash_\Sigma [\varphi \supset \psi]{:}t}{\Gamma \vdash_\Sigma \psi{:}t}$$

$$(\forall prop) \qquad \frac{\Gamma, x{:}\sigma \vdash_\Sigma \varphi{:}p}{\Gamma \vdash_\Sigma \forall x{:}\sigma.\varphi{:}p}$$

$$(\forall I) \qquad \frac{\Gamma, x{:}\sigma \vdash_\Sigma \varphi{:}t}{\Gamma \vdash_\Sigma \forall x{:}\sigma.\varphi{:}t} \quad \text{where } x \text{ is not free in } \varphi \text{ or any assumptions in } \Gamma$$

$$(\forall E) \qquad \frac{\Gamma\vdash_\Sigma \forall x{:}\sigma.\varphi{:}t \qquad \Gamma\vdash_\Sigma \alpha{:}\sigma}{\Gamma\vdash_\Sigma \varphi[\alpha/x]{:}t}$$

$$(\exists prop) \qquad \frac{\Gamma, x{:}\sigma\vdash_\Sigma \varphi{:}p}{\Gamma\vdash_\Sigma \exists x{:}\sigma.\varphi{:}p}$$

$$(\exists I) \qquad \frac{\Gamma, x{:}\sigma\vdash_\Sigma \varphi[\alpha/x]{:}t}{\Gamma\vdash_\Sigma \exists x{:}\sigma.\varphi{:}t}$$

$$(\exists E) \qquad \frac{\Gamma\vdash_\Sigma \exists x{:}\sigma.\varphi{:}t \qquad \Gamma, \varphi[\alpha/x]{:}t\vdash_\Sigma \psi{:}t}{\Gamma\vdash_\Sigma \psi{:}t}$$

$$(\alpha) \quad \Gamma\vdash_\Sigma [(\lambda x{:}\sigma.\alpha) = (\lambda y{:}\sigma.\alpha[y/x])]{:}t, \text{ where } y \text{ is not free in } \alpha.$$

$$(\beta) \quad \Gamma\vdash_\Sigma [app(\lambda x{:}\sigma.\alpha, a) = \alpha[a/x]]{:}t,$$

$$(\gamma) \quad \frac{\Gamma\vdash_\Sigma [\alpha_1 = \alpha_2]{:}t \qquad \Gamma\vdash_\Sigma [\beta_1 = \beta_2]{:}t}{\Gamma\vdash_\Sigma [app(\alpha_1, \beta_1) = app(\alpha_2, \beta_2)]{:}t}$$

$$(\delta) \quad \frac{\Gamma\vdash_\Sigma \alpha{:}\sigma}{\Gamma\vdash_\Sigma [\alpha = \alpha]{:}t}$$

$$(\epsilon) \quad \frac{\Gamma\vdash_\Sigma [\alpha_1 = \alpha_2]{:}t \qquad \Gamma\vdash_\Sigma [\alpha_1 = \alpha_3]{:}t}{\Gamma\vdash_\Sigma [\alpha_2 = \alpha_3]{:}t}$$

$$(\zeta) \quad \frac{\Gamma\vdash_\Sigma [app(\alpha_1, x) = app(\alpha_2, x)]{:}t}{\Gamma\vdash_\Sigma [\alpha_1 = \alpha_2]{:}t} \text{ where } x \text{ is not free in } \alpha_1, \alpha_2 \text{ or any assumptions in } \Gamma.$$

1.1.1 Interpreting Λ, the type-free λ-calculus in T_Ω

The type free λ-calculus (Λ for short) has $\alpha ::= x|(\alpha_1\alpha_2)|(\lambda x.\alpha_1)$ for terms. We can embed the type free λ-calculus in our system T_Ω via the embedding function $\mathcal{J}_*$:

DEFINITION 1.3
We define an embedding function $\mathcal{J}_* : \Lambda \mapsto T_\Omega$, which embeds Λ in T_Ω as follows:

- $\mathcal{J}_*(x) = x$
- $\mathcal{J}_*(\alpha_1\alpha_2) = app(\mathcal{J}_*(\alpha_1), \mathcal{J}_*(\alpha_2))$
- $\mathcal{J}_*(\lambda x.\alpha_1) = \lambda x : v.\mathcal{J}_*(\alpha_1)$ where v is a fresh type variable. This is to avoid any type variable clashes inside terms.

Note that we use $free(\alpha)$ for the set of the free variables of α.

LEMMA 1.4
For any expression α and variable x, $x \in free(\alpha)$ iff $x \in free(\mathcal{J}_*(\alpha))$.

PROOF. By an easy induction on α in Λ. ∎

LEMMA 1.5
$\mathcal{J}_*(\alpha[\alpha'/x]) = \mathcal{J}_*(\alpha)[\mathcal{J}_*(\alpha')/x]$.

PROOF. By an easy induction on α in Λ. ∎

As we consider the pure type-free λ-calculus (i.e. no constants are used), we can ignore signatures in this subsection. Hence, we drop the subscript Σ from $\Gamma \vdash_\Sigma \alpha{:}\sigma$.

LEMMA 1.6
For any term $\alpha \in \Lambda, \exists \Gamma$, unification function θ, σ such that $\Gamma \vdash \theta(\mathcal{J}_*(\alpha)){:}\sigma$.

PROOF. This is long, but straightforward by induction on the terms in Λ. ∎

The following is an example which illustrates this lemma:

EXAMPLE 1.7
Here we see how this lemma applies to $\lambda x.xx$. We know that $\mathcal{J}_*(\lambda x.xx) = \lambda x{:}v_1.app(x,x)$. Now, if we take $\theta(v_1) = \langle v_1, v_2\rangle$ and the empty context, then we can show that $\vdash \theta(\lambda x{:}v_1.app(x,x)){:}\langle\langle v_1, v_2\rangle.v_2\rangle$ as follows:
(note that $\theta(\lambda x{:}v_1.app(x,x)) = \lambda x{:}\langle v_1, v_2\rangle.app(x,x)$

1.	$x{:}\langle v_1, v_2\rangle$	$\vdash$	$x{:}\langle v_1, v_2\rangle$	$(Base)$
2.	$x{:}\langle v_1, v_2\rangle$	$\vdash$	$\langle v_1, v_2\rangle \preceq v_1$	$(Dom_\preceq)$
3.	$x{:}\langle v_1, v_2\rangle$	$\vdash$	$x{:}v_1$	$1, 2, (Contain)$
4.	$x{:}\langle v_1, v_2\rangle$	$\vdash$	$app(x,x){:}v_2$	$1, 3, (App)$
5.		$\vdash$	$\lambda x{:}\langle v_1, v_2\rangle.app(x,x){:}\langle\langle v_1, v_2\rangle, v_2\rangle$	$4, (\lambda)$

Λ moreover assumes the usual α, β and η axioms.

LEMMA 1.8
If $\Lambda \vdash \alpha = \alpha'$ then there exists Γ, θ such that $\Gamma \vdash [\theta(\mathcal{J}_*(\alpha)) = \theta(\mathcal{J}_*(\alpha'))] : t$.

PROOF. By an easy induction on the derivation of $\alpha = \alpha'$ in Λ. ∎

Hence we have the full type free λ-calculus in T_Ω. With this interpretation, we are free now to write some of our expressions as type free terms. That is, a term of the form $\lambda x.\alpha$ is an acceptable term of our theory, even though it doesn't occur in the syntax given for expressions.

1.1.2 Interpreting the system λ_L in T_Ω

The types in λ_L (see [20]) are exactly those of T_Ω, but λ_L does not have any metatypes. The ordering on the types is exactly the same in both systems. All the typing rules of λ_L are also typing rules of T_Ω. Moreover, the expressions of λ_L are as follows:
$\alpha ::= x \mid app(\alpha,\beta) \mid \lambda x{:}\sigma.\alpha \mid \neg\alpha \mid [\alpha \wedge \beta] \mid [\alpha \supset \beta] \mid \forall x{:}\sigma.\alpha \mid \alpha = \beta$

In fact, all the expressions of λ_L are also expressions of T_Ω. Hence, the translation function from Λ_L to T_Ω, $\mathcal{J}_{\lambda_\mathcal{L}}$ is simply the identity function.

Now, as all the expressions, types, type ordering and type inference rules of λ_L are included in T_Ω, then the following lemma is easily provable:

LEMMA 1.9
If in λ_L, we prove $\Gamma \vdash \alpha{:}\sigma$ then in T_Ω, we prove $\Gamma \vdash \mathcal{J}_{\lambda_\mathcal{L}}(\alpha){:}\sigma$.

COROLLARY 1.10
If in λ_L, we prove $\Gamma \vdash \alpha{:}p$ then in T_Ω, we prove $\Gamma \vdash \mathcal{J}_{\lambda_\mathcal{L}}(\alpha){:}p$.

1.1.3 Interpreting the system T_H in T_Ω

The system T_H of [15] has the following syntax of terms:

$$\alpha ::= x \mid \lambda x.\alpha \mid \alpha\beta \mid \Omega\alpha \mid [\alpha \wedge \beta] \mid [\alpha \vee \beta] \mid [\alpha \supset \beta] \mid [\alpha = \beta] \mid \forall x.\alpha \mid \exists x.\alpha \tag{1.1}$$

We interpret the first three terms exactly as we interpreted the type free λ-calculus. We interpret the terms $[\alpha\ op\ \beta]$ where op is $=, \wedge, \vee$ or $\supset$ by $[\mathcal{J}_{\mathcal{TH}}(\alpha)\ op\ \mathcal{J}_{\mathcal{TH}}(\beta)]$ where $\mathcal{J}_{\mathcal{TH}}(\alpha)$ is the interpretation of α in T_Ω. We interpret $\Omega\alpha$ by $\Omega(\mathcal{J}_{\mathcal{TH}}(\alpha))$, $\forall x.\alpha$ and $\exists x.\alpha$ similarly to the interpretation of $\lambda x.\alpha$. For example, $\mathcal{J}_{\mathcal{TH}}(\forall x.\alpha) = \forall x{:}v.\alpha$ where v is a fresh type variable.

LEMMA 1.11
If $T_H \vdash \alpha$ then $\exists\Gamma$ such that in T_Ω, we can prove $\Gamma \vdash \mathcal{J}_{\mathcal{TH}}(\alpha) : t$.

PROOF. By an easy induction on the derivation of $T_H \vdash \alpha$. ∎

1.1.4 Interpreting the system $\mathcal{L}_{\underline{\preceq}}$ in T_Ω

The system $\mathcal{L}_{\underline{\preceq}}$ of [18] has the same signature and contexts as T_Ω. Kinds however in $\mathcal{L}_{\underline{\preceq}}$ are different. They include, like the system here, types and metatypes. c-types however are replaced by three other types, le-, fp- and wb-types. The idea is that fp-types play a similar role to c-types. That is, they are both circular. le- and wb-types however, are there to avoid negative judgements in the type inference rule (λ). That is, instead of adding the condition $\not\vdash \sigma$ c-type we add the condition $\vdash \sigma$ wb-type, meaning that σ is a well behaved type and that abstracting to the type σ will not lead to contradiction. le-types were an intermediate step between fp-types and wb-types. That is not the end of the story. In fact, the typing system obtained in [18] is rather different from that of this paper. We can understand the difference by giving two types which are comparable in one and not in the other.

EXAMPLE 1.12
In T_Ω, $\langle p, e\rangle \preceq p$ but in [18], there is a lemma which says that if $\langle \tau_1, \tau_2\rangle \preceq \sigma$ then either $\sigma = e$ or σ is a complex type. Hence, it is not the case that $\langle p, e\rangle \preceq p$ in [18].

Moreover, in [18], as $p \preceq e$ then $\langle e, e\rangle \preceq \langle p, e\rangle$ which is not derivable in T_Ω.

The syntax of expressions of $\mathcal{L}_{\underline{\preceq}}$ is as follows:

$$\begin{array}{rl} \alpha ::= c & \mid x \mid \lambda x{:}\sigma.\alpha \mid app(\alpha, \beta) \mid \alpha(\beta) \mid \neg\alpha \mid {}^{\cup}\alpha \mid [\alpha \wedge \beta] \mid [\alpha \vee \beta] \mid [\alpha \supset \beta] \\ & \mid [\alpha = \beta] \mid \forall x{:}\sigma.\alpha \mid \exists x{:}\sigma.\alpha \end{array} \tag{1.2}$$

T_H and $\mathcal{L}_{\underline{\preceq}}$ are related. In fact, $\mathcal{L}_{\underline{\preceq}}{}^{c}$ (which is $\mathcal{L}_{\underline{\preceq}}$ without constants) can be interpreted in T_H as follows: we take variables to variables, $\lambda x{:}\sigma.\alpha$ to $\lambda x.\alpha$, $app(\alpha, \beta)$ and $\alpha(\beta)$ to $\alpha\beta$, ${}^{\cup}\alpha$ to α, $\neg\alpha$ to $\neg\alpha$ and $[\alpha\ op\ \beta]$ to the obvious interpretation.

LEMMA 1.13
If $\Sigma \vdash_{\mathcal{L}_{\underline{\preceq}}} \varphi{:}p$ then $\Sigma \vdash_{T_H} \Omega\varphi$.

PROOF. By an easy induction on $\Sigma \vdash_{\mathcal{L}_{\underline{\preceq}}} \varphi{:}p$. ∎

1.1.5 Interpreting the system L_λ in T_Ω

The types in L_λ [14] are exactly those of T_Ω, but L_λ does not have any metatypes. The ordering on the types is exactly the same in both systems. All the typing rules of L_λ are also typing rules of T_Ω. Moreover, the expressions of L_λ are as follows:

$$(1.3) \qquad \alpha ::= x \mid \alpha\beta \mid \lambda x.\alpha \mid \lambda x{:}\sigma.\alpha \mid \Omega\alpha \mid \neg\alpha \mid [\alpha \wedge \beta] \mid [\alpha \supset \beta] \mid \forall x{:}\sigma.\alpha \mid \forall x.\alpha$$

In fact, all the expressions of L_λ are also expressions of T_Ω (look at the interpretation of the type free terms as done previously).

Now, as all the expressions, types, type ordering and type inference rules of L_λ are included in T_Ω, then the following lemma is easily provable (proof similar to that of lemma 1.9):

LEMMA 1.14
If in L_λ, $\Gamma \vdash \alpha{:}\sigma$ then in T_Ω, $\Gamma \vdash \mathcal{J}_{\mathcal{L}_\lambda}(\alpha){:}\sigma$.

COROLLARY 1.15
If in L_λ, $\Gamma \vdash \alpha{:}p$ then in T_Ω, $\Gamma \vdash \mathcal{J}_{\mathcal{L}_\lambda}(\alpha){:}p$.

1.1.6 Interpreting the Chierchia-Turner system

β could be divided into two parts where we replace equality by an asymmetric relation $\twoheadrightarrow$:

1. **Contraction** $\quad (\lambda x.\alpha)\alpha' \twoheadrightarrow \alpha[\alpha'/x]$
2. **Expansion** $\quad \alpha[\alpha'/x] \twoheadrightarrow (\lambda x.\alpha)\alpha'$

Contraction causes no problems but expansion does in the presence of negation. This is what guided Turner and Chierchia in developing their theory PT_1 [6]. We now show this can be interpreted in T_Ω. The construction of types (sorts) in PT_1 is very straightforward.

DEFINITION 1.16 (SORTS)
The basic sorts of PT_1 are e, u, nf and i. These stand for individuals, urelements, nominalized functions and information units, respectively. The only complex sort is $(e \to e)$.

DEFINITION 1.17 (SYNTAX OF PT_1)
The syntax of PT_1 is as follows: For any sort σ, let ME_σ be the meaningful expressions of sort σ. If $\sigma = e, i, u$ or nf, then Var_σ is a denumerable set of variables of sort σ. If σ is any sort, Con_σ is a set of constants of sort σ. The expressions of each sort are defined as follows:

i. $Var_\sigma, Con_\sigma \subseteq ME_\sigma$
ii. If $\alpha \in ME_e$ and $x \in Var_e$, then $\lambda x.\alpha \in ME_{(e\to e)}$
iii. If $\alpha \in ME_{nf}$, then $^\cup\alpha \in ME_{(e\to e)}$
iv. If $\alpha \in ME_{(e\to e)}$, then $^\cap\alpha \in ME_{nf}$
v. If $\alpha \in ME_{(e\to e)}$ and $\beta \in ME_e$, then $\alpha(\beta) \in ME_e$
vi. $ME_i \subseteq ME_u; ME_u, ME_{nf} \subseteq ME_e$
vii. If $\alpha \in ME_e$, then $\dagger\,\alpha \in ME_i$
viii. If $\psi, \varphi \in ME_i, \alpha, \alpha' \in ME_e$, and $x \in Var_\sigma$, for any sort σ, then $\alpha = \alpha', \neg\psi, \psi \vee \varphi, \psi \wedge \varphi, \forall x.\psi, \exists x.\psi, \psi \supset \varphi, \psi \leftrightarrow \varphi$ are all in ME_i

Note that $\dagger\alpha$ asserts the truth of α, such that if $\alpha \in ME_i$ then $\dagger\alpha$ is its truth value but if $\alpha \notin ME_i$ then $\dagger\alpha$ will be false.

DEFINITION 1.18 (AXIOMS OF PT_1)
The axioms of the theory are as follows:

(1) $(\lambda x.\alpha)\alpha' = \alpha[\alpha'/x]$

(2) i. $\dagger\psi$, where ψ is a tautology
ii. $\psi \supset \dagger\psi$, where ψ is atomic, i.e. of the form $\beta(\alpha)$
iii. $\dagger\psi \supset \psi$
iv. $(\forall x.\dagger\psi) \supset \dagger(\forall x.\psi)$
v. $(\dagger\psi \wedge \dagger(\psi \supset \varphi)) \supset \dagger\varphi$
vi. $\dagger(\neg\dagger\psi) \leftrightarrow \dagger\dagger\neg\psi$

That is, one can go from ψ to $\dagger\psi$ if ψ is an information unit (i.e. a proposition) and is atomic.

Now let us interpret PT_1 in T_Ω. First we start by interpreting the sorts into our kinds. For this we introduce the function $\tau : Sorts \mapsto Kind \cup \{u'\}$ such that:

$$\begin{array}{lcl} \tau(e) & = & e \\ \tau(i) & = & p \\ \tau(u) & = & u' \\ \tau(nf) & = & \langle e, e\rangle \\ \tau(e \to e) & = & e \to e \end{array}$$

The sort u strictly corresponds to our type e minus the type $\langle e, e\rangle$. Since we have no way of proving that there is such a type in T_Ω, we postulate u' which represents u. This will not affect our discussion below, and hence we shall proceed.

We introduce for each expression α of PT_1 the relevant environment of α, $env(\alpha)$ as follows:

1.	$env(\alpha)$	$=$	$(\alpha{:}\tau(\sigma))$	if $\alpha \in var_\sigma \cup Con_\sigma$
2.	$env(\lambda x.\alpha)$	$=$	$env(\alpha)$	if $\alpha \in ME_e$ and $x \in Var_e$
3.	$env(\cup\alpha)$	$=$	$env(\alpha)$	if $\alpha \in ME_{nf}$
4.	$env(\cap\alpha)$	$=$	$env(\alpha)$	if $\alpha \in ME_{e\to e}$
5.	$env(\alpha(\beta))$	$=$	$env(\alpha) \cup env(\beta)$	if $\alpha \in ME_{e\to e}$ and $\beta \in ME_e$
6.	$env(\dagger\alpha)$	$=$	$env(\alpha)$	if $\alpha \in ME_e$
7.	$env(\alpha = \beta)$	$=$	$env(\alpha) \cup env(\beta)$	if $\alpha, \alpha' \in ME_e$
8.	$env(\neg\psi)$	$=$	$env(\psi)$	if $\psi \in ME_i$
9.	$env(\psi\ op\ \varphi)$	$=$	$env(\psi) \cup env(\varphi)$	if $\psi, \varphi \in ME_i, op = \wedge, \vee, \supset, \leftrightarrow$
10.	$env(\mathcal{Q}x.\psi)$	$=$	$env(\psi)$	if $\psi \in ME_i, x \in Var_\sigma$ and $\mathcal{Q} = \forall, \exists$

What if $env(\alpha)$ and $env(\alpha')$ overlap? That is, what if $env(\alpha)$ contains $(x{:}e)$ and $env(\alpha')$ contains $(x{:}p)$? If this is the case, we can solve it by taking $(x{:}p)$ to be the common element. This should not occur however if we assume that the variables and constants of each sort are disjoint from those of any other sort. We now introduce a mapping Tr which takes expressions of PT_1 and returns expressions in T_Ω. This is

defined as follows:

1.	$Tr(\alpha)$	$=$	α	if $\alpha \in var_\sigma \cup Con_\sigma$
2.	$Tr(\lambda x.\alpha)$	$=$	$\cup(\lambda x{:}e.\, Tr(\alpha))$	if $\alpha \in ME_e$ and $x \in Var_e$
3.	$Tr(\cup\alpha)$	$=$	$\cup\, Tr(\alpha)$	if $\alpha \in ME_{nf}$
4.	$Tr(\cap\alpha)$	$=$	$\lambda x{:}e.app(Tr(\alpha), x)$	if $\alpha \in ME_{e\to e}$
5.	$Tr(\alpha(\beta))$	$=$	$Tr(\alpha)(Tr(\beta))$	if $\alpha \in ME_{e\to e}$ and $\beta \in ME_e$
6.	$Tr(\dagger\alpha)$	$=$	$\begin{cases} Tr(\alpha) & \text{if } Tr(\alpha){:}p \\ c_1 = c_2 & \text{otherwise} \end{cases}$	if $\alpha \in ME_e, c_1, c_2$ different constants
7.	$Tr(\alpha = \alpha')$	$=$	$Tr(\alpha) = Tr(\alpha')$	if $\alpha, \alpha' \in ME_e$
8.	$Tr(\neg\psi)$	$=$	$\neg\, Tr(\psi)$	if $\psi \in ME_i$
9.	$Tr(\psi\ op\ \varphi)$	$=$	$Tr(\psi)\ op\ Tr(\varphi)$	if $\psi, \varphi \in ME_i, op = \wedge, \vee, \supset$
10.	$Tr(\psi \leftrightarrow \varphi)$	$=$	$(Tr(\psi) \Leftrightarrow Tr(\varphi))$	if $\psi, \varphi \in ME_i, \psi \Leftrightarrow \varphi$[1]
11.	$Tr(op\ x.\psi)$	$=$	$op\ x{:}\sigma.\, Tr(\psi)$	if $\psi \in ME_i, x \in Var_\sigma$ and $op = \forall, \exists$

Note that $Tr(\dagger\alpha)$ is always of type p. This is the reason why we couldn't take $Tr(\dagger\alpha)$ to be $T(\alpha) \equiv \Omega(Tr(\alpha)) \supset \alpha$. Note moreover that for any expression α of PT_1, it is decidable whether $Tr(\alpha)$ is a proposition or not; i.e. it is decidable whether $\Omega(Tr(\alpha)){:}t$ or not. This can be seen by the following lemma.

LEMMA 1.19
For any expression α of PT_1, if $\alpha \in ME_e$ then $env(\alpha) \vdash_\Sigma Tr(\alpha){:}p$ is decidable and if $\alpha \in ME_{e\to e}$ then for any $a \in ME_e$, $env(\alpha) \vdash_\Sigma Tr(\alpha(a)){:}p$ is decidable.

PROOF. By a double induction on α in PT_1. ∎

LEMMA 1.20
If $\alpha \in ME_\sigma$ where α is an expression of PT_1, then $env(\alpha) \vdash_\Sigma Tr(\alpha){:}\tau(\sigma)$.

PROOF. By an easy induction on α in PT_1. Work with the assumption of the existence of u' which denotes u and which satisfies its inclusion relationships. ∎

LEMMA 1.21
The axioms of PT_1 are all valid in T_Ω.

PROOF. This is easy, by going through definition 1.18. ∎

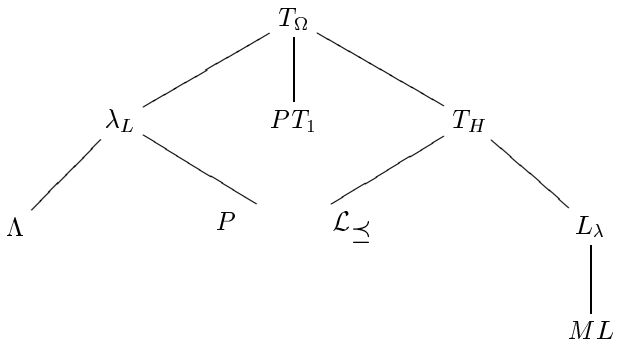
The above shows that PT_1 of [6] can be considered as a subtheory of T_Ω.

1.1.7 T-tree

Now collecting the results, we draw the picture which relates all these various theories. We add Milner's ML as it has been shown in [14] to be interpretable in L_λ.

---

[1] i.e. $(\psi \supset \varphi) \wedge (\varphi \supset \psi)$.



T_Ω is the system of [22].
Λ is the type free λ-calculus.
PT_1 is the Chierchia-Turner system of [6].
T_H is the system of [15].
λ_L is the system of [20].
P is the system of Parsons in [36]
$\mathcal{L}_{\preceq}$ is the system of [18].
L_λ is the system of [14].
ML is Milner's ML system of [35].

2 Property theories

Our interest in property theory stems from the fact that properties and propositions are strongly related and provide the logical part of the system (whereas type theory is going to provide the expressive part of the system). Our domain of properties will satisfy important closure properties which will make the logic simple to reason with. Moreover, properties will play an important role in predicatives which can be looked at in the Fregean sense or in the sense of Bealer. Our approach furthermore, permits us to distinguish between predication and abstraction. To talk about such predicative expressions, we introduce in our language T_Ω the operator Δ (see [15]), understanding $\Delta\alpha$ to mean that α is a property. Δ is defined as $\Delta\alpha =_{df} \forall x.\Omega(app(\alpha, x))$. That is, something is a property iff whenever it applies to an object, the result is a proposition. We construct further properties in the following way:

1. $\alpha \cup \alpha' =_{df} \lambda x.(app(\alpha, x) \vee app(\alpha', x))$
2. $\alpha \cap \alpha' =_{df} \lambda x.(app(\alpha, x) \wedge app(\alpha', x))$
3. $\overline{\alpha} =_{df} \lambda x.\neg app(\alpha, x)$
4. $\alpha \to \alpha' =_{df} \lambda x.\forall y(app(\alpha, y) \supset app(\alpha', app(x, y))$
5. $\Theta =_{df} \lambda x.(x = x)$
6. $\odot =_{df} \lambda x.\neg(x = x)$

LEMMA 2.1
The following are provable

1. $\vdash \Delta\Theta{:}t$
2. $\vdash \Delta \odot{:}t$

3. $\vdash \{\Delta\alpha{:}t, \Delta\alpha'{:}t\} \vdash \Delta(\alpha \cup \alpha'){:}t$
4. $\{\Delta\alpha{:}t, \Delta\alpha'{:}t\} \vdash \Delta(\alpha \cap \alpha'){:}t$
5. $\vdash \{\Delta\alpha{:}t\} \vdash \Delta\overline{\alpha}{:}t$
6. $\vdash \{\Delta\alpha{:}t, \Delta\alpha'{:}t\} \vdash \Delta(\alpha \to \alpha'){:}t$

The following lemma shows that the internal logic (which occurs inside the λ, such as $\lambda x.\varphi \supset \psi$ and $\lambda x.\varphi$ give $\lambda x.\psi$) and the external logic (which occurs outside and is the usual one), can be unified. That is, the logic of our propositions and the logic of our properties are the same. Computationally, this means that logical connectives can be pushed inside the λ-operator.

LEMMA 2.2
The following are provable

1. $[app(\lambda x.\alpha, \beta) \wedge app(\lambda x.\alpha', \beta) = app(\lambda x.(\alpha \wedge \alpha'), \beta)]{:}t$
2. $[app(\lambda x.\neg\alpha, \beta) = \neg app(\lambda x.\alpha, \beta)]{:}t$
3. $[app(\lambda x.\alpha, \beta) \vee app(\lambda x.\alpha', \beta) = app(\lambda x.(\alpha \vee \alpha'), \beta)]{:}t$
4. $[app(\overline{\alpha}, \beta) = \neg app(\alpha, \beta)]{:}t$
5. $[app(\alpha \cap \alpha', \beta) = app(\alpha, \beta) \wedge app(\alpha', \beta)]{:}t$
6. $[app(\alpha \cup \alpha', \beta) = app(\alpha, \beta) \vee app(\alpha', \beta)]{:}t$
7. $[app(\overline{(\overline{\alpha})}, \beta) = \neg\neg app(\alpha, \beta)]{:}t$
8. $\{\Delta\alpha{:}t, \Delta\alpha'{:}t\} \vdash [app(\overline{(\alpha \cup \alpha')}, \beta) \Leftrightarrow app((\overline{\alpha}), \beta) \wedge app((\overline{\alpha'}), \beta)]{:}t$
9. $\{\Delta\alpha{:}t, \Delta\alpha'{:}t\} \vdash [app(\overline{\alpha} \cup \overline{\alpha'}, \beta) = app(\overline{\alpha}, \beta) \vee app(\overline{\alpha'}, \beta)]{:}t$
10. $\{\Delta\alpha{:}t, \Delta\alpha'{:}t\} \vdash [app(\overline{\alpha} \cap \overline{\alpha'}\beta) \Leftrightarrow app(\overline{(\alpha \cup \alpha')}, \beta)]{:}t$
11. $\{app(\alpha, \beta){:}t\} \vdash [app(\overline{\overline{\alpha}}, \beta)]{:}t$
12. If $\Gamma\vdash_\Sigma \Omega\alpha{:}t$ then $\Gamma\vdash_\Sigma [\forall y.app(\lambda x.\alpha, \alpha') \supset app(\lambda x.\forall.y\alpha, \alpha')]{:}t$
13. If $\Gamma\vdash_\Sigma \Omega\alpha{:}t$ then $[\exists y.app(\lambda x.\alpha, \alpha') \supset app(\lambda x.\exists y.\alpha, \alpha')]{:}t$

Now we discuss what would happen to the lemmas above if we change the functional application of the λ-calculus by a more intensional application, call it *pred*. That is, from $app(\alpha, x) = app(\beta, y)$, we can deduce nothing about the relationship between α and β and x and y. *pred* on the other hand, will satisfy the condition that if $pred(\alpha, a) = pred(\beta, b)$ then $\alpha = \beta$ and $a = b$. So let us introduce *pred* such that

$$(2.1)\quad \frac{pred(\alpha, x){:}t}{app(\alpha, x){:}t} \qquad \frac{app(\alpha, x){:}t}{pred(\alpha, x){:}t} \qquad \frac{\Omega(pred(\alpha, x)){:}t}{\Omega(app(\alpha, x)){:}t} \qquad \frac{\Omega(app(\alpha, x)){:}t}{\Omega(pred(\alpha, x)){:}t}$$

$$(2.2)\quad [\forall x(pred(\alpha, x) = pred(\beta, x))]{:}t \Rightarrow [\alpha = \beta]{:}t$$

$$(2.3)\quad [pred(\alpha, a) = pred(\beta, b)]{:}t \Rightarrow [(\alpha = \beta \wedge a = b)]{:}t$$

LEMMA 2.3
If $\Gamma\vdash_\Sigma (\Delta\alpha){:}t$ then $\Gamma\vdash_\Sigma [\forall x.(app(\alpha, x) \Leftrightarrow pred(\alpha, x))]{:}t$.

3 Definability of determiners and quantifiers in T_Ω

We define the two determiners every' and a' in our framework:

$$\underline{every'} =_{df} \lambda x.\lambda y.\forall z(xz \to yz)$$

$$\underline{a'} =_{df} \lambda x.\lambda y.\exists z(xz \wedge yz)$$

The characteristic property of every', $\subseteq$, is defined by: $P_1 \subseteq P_2 =_{df} \forall x(P_1 x \to P_2 x)$.

LEMMA 3.1
$\subseteq$ is a transitive, reflexive and equisymmetric relation on properties.

LEMMA 3.2
If P_1 and P_2 are properties then $\Omega(every' P_1 P_2)$ and $every' P_1 P_2 = P_1 \subseteq P_2$.

We define $P_1 \cap^1 P_2 =_{df} \exists z(P_1 z \wedge P_2 z)$.

LEMMA 3.3
If P_1 and P_2 are properties then $\Omega(a' P_1 P_2)$ and $a' P_1 P_2 = P_1 \cap^1 P_2$.

Outside the collection of properties, we cannot draw useful conclusions about every' because we cannot decide the propositionhood of an arbitrary formula in which $\to$ is the main connective. This is not a disadvantage as we only want every' to have meaning when we are working with properties. Moreover, we cannot define the type of every' or of determiners inside our formal language. That is if we define Quant and Det as follows

$$\text{Quant } t =_{df} \forall x(\Delta x \to \Omega(tx))$$

$$\text{Det } t =_{df} \forall x(\Delta x \to \text{ Quant } (tx)).$$

then there is no way to prove that Det and Quant always return propositions when applied to terms, because $\forall x(\Delta x \to$ Quant $(tx))$ and $\forall x(\Delta x \to \Omega(tx))$ are not propositions for any t. In fact even if t is a property, we still do not have a guarantee that Det t and Quant t are propositions, due to the fact that Δx is not a proposition. This is not serious as there is no particular reason for wanting determiners and quantifiers to be determinate. We can prove many desirable features of our determiners, so why insist on determinability?

Having determiners such as every', a' is one thing; being able to deduce that every', a' are determiners is something else. I.e. can we prove that Det(every'), Det(a'), etc..? Take for every', $\lambda x.\lambda y.\forall z[xz \to yz]$. To show that Det(every') we have to show that $\forall x(\Delta x \to \forall y(\Delta y \to H(every' xy)))$. But to be able to show the implication we need to have $\Omega(\Delta x)$, and $\Omega(\Delta y)$, which we cannot assume. For this we need an extension for implication as follows:

We always have that if $\{a\} \vdash b$ then $\{\Omega a\} \vdash a \to b$ (our version of the deduction theorem). We need that if $\{\Omega a\} \vdash b$ then $\vdash \Omega a \to b$. Can we assert this rule? That is:

$$(*) \qquad \text{If } \{\Omega a\} \vdash b \text{ then } \vdash \Omega a \to b.$$

LEMMA 3.4
Det(every'), Det(a'), if (*) holds. (See [15] for the proof.)

Here we are concerned with some characteristics of determiners that can be proven in our theory. We start with the first theorem that asserts that the result of applying a quantifier to a property results in a proposition.

LEMMA 3.5

(*i*) $\{Quant(Q), DP\} \vdash \Omega(QP)$
(*ii*) $\{Quant(a), Quant(b)\} \vdash Quant(a \cap b)$
(*iii*) $\{Quant(a), Quant(b)\} \vdash Quant(a \cup b)$
(*iv*) $\{Quant(a)\} \vdash Quant(a^c)$ where a^c is the complement of a
(*v*) $\{every' P_1 P_2, every' P_2 P_3\} \vdash every' P_1 P_3$

4 Intensionality and extensionality

It is often observed that sentence accent, as an indicator of focus, can affect the interpretation of sentences, or at least the contexts in which they are appropriate. Thus

(4.1) Felix ate THE PIE

(with accent on *the pie*) is felicitous as an answer to the (perhaps only implicit) question *What did Felix do?* or *Who did Felix ate?*. By contrast,

(4.2) FELIX ate the pie

answers the question *Who ate the pie?*

[31] has suggested that the information structure of such sentences should be represented by separating sentence meaning into a pair consisting of a focus part and a background part, where 'the background is of a type that can be applied to the focus'. Moreover, [31], following [11], has proposed that even if there is no focus-sensitive operator (such as *only*), the focus should be 'bound' by an illocutionary operator that expresses the sentence mood. Suppose, for example, that ASSERT is the assertion operator. Then (4.1) will receive the following representation:

(4.3) $\text{ASSERT}(\langle \lambda x.eat(felix, x), \textit{the-pie} \rangle)$

One question which this proposal raises is whether 'free' focus constructions such as (4.1) and (4.2) can ever occur in embedded constructions. Thus consider the following examples:

(4.4) Sandy was surprised that Felix ate THE PIE.

(4.5) Sandy was surprised that FELIX ate the pie.

Intuitively, these two sentences can have different truth conditions. Suppose, for example, that Felix is known to be both a glutton and a gourmand. Given the alternative delicacies available, it may surprise Sandy that Felix chose the pie to eat. Yet knowing that the pie did in fact get eaten, Sandy may not be surprised that it was Felix that did the eating. If this is correct, then it seems unlikely that the partitioning of meaning into focus and background can be entirely separated out from propositional content.

The relation *pred* which we introduced appears to give us an appropriate amount of structure within propositions. By comprehension, we know that the following equations hold:

(4.6) $app(\lambda x.eat(felix, x), the\text{-}pie) =$
$eat(felix, the\text{-}pie) =$
$app(\lambda y.eat(y, the\text{-}pie), felix)$

However, *pred* does *not* support these identities:

(4.7) $pred(\lambda x.eat(felix, x), the\text{-}pie) \neq pred(\lambda y.eat(y, the\text{-}pie), felix)$

Obviously, there is an additional pragmatic burden being supported by information structure. Nevertheless, it seems clear that the apparatus we have defined gives the right kind of fine-grainedness at the propositional level to support the distinctions which need to be drawn.

In fact, the problem of identifying $app(\lambda x.eat(felix, x), the\text{-}pie)$ with $app(\lambda y.eat(y, the\text{-}pie), felix)$ is a problem of intensionality. Our account does not face this problem as we have another predicate supported by our logic which is intensional. So even though $pred(\lambda x.eat(felix, x), the\text{-}pie)$ has the same truth value as $pred(\lambda y.eat(y, the\text{-}pie), felix)$, they are not equal. This problem is similar to another one of Bealer and Aczel in [2] which is as follows:

	$Rajneeshee$	$=$	$\lambda x.follows(x, Rajneesh)$
	$Fondalee$	$=$	$\lambda x.follows(JaneFonda, x)$
	$app(Rajneeshee, JaneFonda)$	$=$	$follows(JaneFonda, Rajneesh)$
	$app(Fondalee, Rajneesh)$	$=$	$follows(JaneFonda, Rajneesh)$
Therefore	$app(Rajneeshee, JaneFonda)$	$=$	$app(Fondalee, Rajneesh)$

This conclusion might be questioned since someone could believe that *Rajneeshee* holds of Fonda, without believing that *Fondalee* holds of Rajneesh. The solution here is to use *pred* instead of *app*. So we obtain that $pred(Rajneeshee, JaneFonda)$ is equivalent in truth value to $pred(Fondalee, Rajneesh)$ but not equal to it. This is another example of the suitability of our framework for intensional and finely-grained contexts.

5 Polymorphism

Types or levels are not necessary in the avoidance of the paradox. The Foundation Axiom FA was included in ZF despite the fact that it was shown that antifoundation axioms are consistent with ZF (see [1] for such a discussion). In fact, it is the Axiom of Separation which avoids the paradox. Moreover, the claim in the foundation of NL has been concentrating on abandoning well foundedness. It has been put forward that non well foundedness and type freeness are necessary for NL. [21], for example, provides a unified account of plurals and singulars by using the concept of non well foundedness and type freeness and [18] uses the notion of type freeness to give a more general interpretation of NL.

The fact that we ask for the full expressive power of the type free λ-calculus does not mean that types are not useful. 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. 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, alas this type free paradise had to be abandoned. Types found an attractive place in the history of foundation and in most areas of applications of logic. However, types are useful yet we must have as much type freeness as possible. In fact 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 the paradox is 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 and the expressive power. 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 we are 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. We can talk about more involved self-referential properties such as the property of properties that apply to themselves, the set of functions which given an argument x, apply the function $\lambda x.f(xx)$ to itself.

5.1 Promiscuity and polymorphism

From a pretheoretic point of view, natural language expressions clearly enjoy a great deal of combinatorial flexibility. A familiar example is the conjunctive *and* which places very few constraints on the category of its arguments, except perhaps that they be of the same category. Similarly, many verbs can combine with a range of different complements:

(5.1)
a. Lee proved that 13 was a prime number.
b. Lee proved the proposition that 13 was a prime number.
c. Lee proved his claim.
d. Lee proved it.

Such combinatorial flexibility deserves a name: let us call it *functional promiscuity*, following the lead of [12]. How should we model functional promiscuity? We could take the approach favoured by [4], and claim that natural language is entirely type free; or else we could say that there are some type restrictions, but that the type system has enough slop in it to allow the requisite amount of promiscuity.

Although Bealer's approach certainly deserves to be explored, it seems to be committed to the view that syntactic categories in natural language are entirely arbitrary, in that they have no semantic import. It seems implausible that we can analyse natural languages in an economical manner while completely eschewing syntactic categories. Yet it also seems implausible that, say, the distinction between noun phrases

and sentences is completely unmotivated from a semantic point of view. Yet if we concede that syntactic categories do have some correlation with semantic domains, then we are essentially admitting types after all.

Let us assume, then, that types are an appropriate tool in the task of analysing natural languages. Then we might still jump in one of two ways in the face of data like (5.1). We could conclude that each of the complements shown in (5.1) is of the same type, in which case we would be forced to the conclusion that words like *it*, *this* and *something* have multiple types. Alternatively, we might suppose that *it* has just the type of singular *NP*s, as distinct from the type of propositions, in which case we have to conclude that *prove* is polymorphically typed.

As Parsons ([36]) shows, some amount of polymorphism is also entailed on the approach where noun phrases like *the proposition* are analysed as having the same type as subordinate clauses such as *that 13 was a prime number*. For then we see that, for example, *about* must be polymorphic:

(5.2) a. Kim talked about the proposition.
b. Kim talked about Sandy.

[20] gives a detailed account of Parsons' approach and interprets it in a theory λ_L which we showed earlier to be a subtheory of T_Ω. Hence Parsons account can also be looked at as a subtheory of T_Ω.

5.2 Fixed points, self application and a programming example

λ-calculus is at the heart of the denotational semantics of programming languages. Programming languages moreover range between the strictly and inflexibly typed languages (such as Pascal where you can only apply functions to a certain type) and the polymorphically typed ones such as Milner's ML. Even the polymorphically typed languages are not polymorphic enough. In fact, the programming discipline which praises polymorphism non stop and which claims to be offering highly polymorphic languages, namely *functional programming*, has not yet provided a language which can make sense of the type of a fixed point operator, or any function which involves self application. This is somewhat anomalous, as functional languages are claimed to be based on the λ-calculus (and in particular on the type free or the polymorphic λ-calculus). Now in these λ-calculus, the fixed point operators and self application play a very important role. Without them, we could not show that the solution to the recursive equations exists. So isn't it strange that the most important items such as self application and the fixed point operators cannot be typechecked in functional languages? After all they are the items which show us what the computable/non computable functions are. They are the items which solve the recursive equations, and they are the items which inform us about the looping/nonlooping programs. Furthermore, Milner's ML is based on the language $\lambda_{\to Curry}$ which cannot typecheck $\lambda_x.xx$ nor Y. The polymorphism of ML which is based on $\lambda_{\to Curry}$ is not strong enough. The polymorphism introduced in this paper however, is strong enough to type check items involving self application. We shall illustrate this below.

EXAMPLE 5.1
The translation of $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ in T_Ω has type $\langle\langle v_2, v_2\rangle, v_2\rangle$. Before we show this, let us write A for $\lambda x{:}\langle\langle v_1, v_2\rangle, v_2\rangle.app(f, app(x, x))$ and write B for

$\lambda x{:}\langle v_1, v_2\rangle.app(f, app(x, x))$. Now, the magical part of the program which takes the type of f to be $\langle v_2, v_2\rangle$ and the type of x to be $\langle\langle v_1, v_2\rangle, v_2\rangle$ is a very important part of [14] and there is no room to discuss it here. But let us see how, when the types of f and x are chosen, the type checker deduces the type of the translation of Y.

(i)	$f : \langle v_2, v_2\rangle$	assumption
(ii)	$x : \langle\langle v_1, v_2\rangle, v_2\rangle$	assumption
(iii)	$\langle v_1, v_2\rangle, v_2\rangle \leq \langle v_1, v_2\rangle$	$(Dom \preceq)$
(iv)	$x : \langle v_1, v_2\rangle$	$(ii), (iii), (Contain)$
(v)	$app(x, x) : v_2$	$(ii), (iv), (app)$
(vi)	$app(f, app(x, x)) : v_2$	$(i), (v), (app)$
(vii)	$\lambda x{:}\langle\langle v_1, v_2\rangle, v_2\rangle.app(f, app(x, x)) : \langle\langle\langle v_1, v_2\rangle, v_2\rangle, v_2\rangle$	$(ii) \ldots (vi), (\lambda)$
$(viii)$	$\langle\langle\langle v_1, v_2\rangle, v_2\rangle, v_2\rangle \leq \langle\langle v_1, v_2\rangle, v_2\rangle$	$(Dom \preceq)$
(ix)	$\lambda x{:}\langle v_1, v_2\rangle.app(f, app(x, x)) : \langle\langle v_1, v_2\rangle, v_2\rangle$	$(vii), (viii), (Contain)$
(x)	$app(A, B) : v_2$	$(iii), (ix), (app)$
(xi)	$\lambda f{:}\langle v_2, v_2\rangle.app(A, B) : \langle\langle v_2, v_2\rangle, v_2\rangle$	$(i) \ldots (x), (\lambda)$

The type of Y is really what it should be. Not only that, but functional languages took polymorphism on their shoulders and avoided logic due to the reason that logic and strong polymorphism together lead to paradoxes. Now we have showed that our system supports a higher polymorphism than functional languages but it also contains logic as we've seen before. In fact this system has been used to extend ML with polymorphism and logic in [14]. And even though the system allows terms such as $\lambda x.xx$ and type check them, all terms which are paradoxical are not typed and the system displays the message that their type is circular. So Russell's and Curry's sentences cannot be type checked and we are told that they are circular. Of course here, one may wonder if the paradox is really avoided, and may give as an example $\alpha \equiv \lambda y.(\lambda x.y(xx))$ which is typechecked to $\langle\langle v_2, v_2\rangle, \langle\langle v_2, v_2\rangle, v_2\rangle\rangle$, and then instantiate it to $\neg\alpha$ which would be of type $\langle\langle p, p\rangle, p\rangle$. This does not hold however because $\langle\langle p, p\rangle, p\rangle$ is circular and the system does not accept such instantiation.

Now, let us say a few words about the computable tractability of the type system T_Ω. This question is particularly important as we have a rich set of types and as the subsumption relation may lead to complex (and non-terminating) type checkers. We have no problems however with T_Ω. The reason being that the system of [14] is the same system as this paper except that there, we did not have metatypes. Now, subsumption does not play any role in metatypes. So computable tractability (which is a characteristic of the system of [14] and of its type checker) transforms easily to T_Ω. In fact, one can take the implementation we have for [14] and extend it with just the rules for metatypes and we obtain an automatic type checker for T_Ω. Finally, let us list some terms and say how the type checker of [14] treats them and type check them. This is relevant for this paper as if we write a type checker for T_Ω, then it will behave exactly the type checker behaves for [14] except of course that there are no metatypes. Hence, on types we are the same. Note that if a term contains λx where x is not explicitly typed (as in the first term below) then the type checker will find the type itself.

	Expressions	**Types**
1	$\lambda x.x$	$\langle v_0, v_0\rangle$
2	$\lambda x : e.x$	$\langle e, e\rangle$
3	$\lambda x.app(x, x)$	$\langle\langle v_0, v_1\rangle, v_1\rangle$
4	$app((\lambda x.app(x, x)), (\lambda x.app(x, x)))$	v_1
5	$\lambda x : p.app(x, x)$	$\langle p, v_0\rangle$
6	$\lambda x : \langle e, p\rangle.app(x, x)$	error: $\langle\langle e, p\rangle, p\rangle$ is c-type
7	$\forall x : \langle v_0, v_1\rangle.app(x, y)$	p
8	$\forall x : e.x$	error, not a proposition
9	$\forall x : \langle e, v_1\rangle.app(x, y)$	p
10	$\forall x.app(x, x)$	p
11	$\lambda x : \langle v_0, v_1\rangle.app(x, y)$	$\langle\langle v_0, v_1\rangle, v_1\rangle$
12	$\lambda x.\neg app(x, x)$	error, c-type

Here don't be alarmed by the type of the sentences 7-10. These are sentences which involve $\forall$ and hence their type should be p. When the system can't make the type p, it returns an error message as in sentence 8.

6 The item notation

The work described in the previous section extends to various other applications that I have not described in this paper due to lack of space. The second program however that I have been involved with is related to a new notation (the item notation) influenced by the AUTHOMATH of de Bruijn. The results that we have obtained in the last four years are very nice and are summarized in our literature below. Of these results, I will briefly describe some points. First let me explain what is the item notation.

The item notation is very simple. It follows the AUTHOMATH by writing the argument before the function. The difference however is that parenthesis in a term are grouped differently than in usual lambda calculus or in AUTHOMATH. The best to describe the item notation is to give the translation from classical lambda calculus to item notation based one. So that, if $\mathcal{I}$ translates classical terms into our notation, then $\mathcal{I}(AB)$ is written as $(\mathcal{I}(B)\delta)\mathcal{I}(A)$ and $\mathcal{I}(\lambda_{x:A}.B)$ is written as $(\mathcal{I}(A)\lambda_x)\mathcal{I}(B)$. Both $(A\delta)$ and $(A\lambda_x)$ are called **items**.

6.1 Explicit substitution

Substitution is the most basic operation of the λ-calculus. Manipulation of λ-terms depends on substitution. The α- and β-axioms are given in terms of substitution. What substitution are we talking about? Substitution in the λ-calculus is usually defined (up to some variation) as $t[x := t']$. So what is happening in $t[x := t']$? We are replacing all free occurrences of x in t by t', but without any disastrous side effects such as binding occurrences of variables which were originally free. Take for example $xx[x := y]$. This will result in yy. $(\lambda_{y:u}.xy)[x := y]$ will result in $\lambda_{z:u}.yz$. So this process of substitution works fine. It is a metalevel process however. That is, this substitution takes t, x, t' and returns a final result $t[x := t']$. The various stages of moving from the t, x, t' to $t[x := t']$ are lost and nothing matters but the result.

This works fine for many applications but fails in areas which are now becoming vital in Computer Science. In functional programming for example, there is an interest in partial evaluation. That is, given $xx[x := y]$, we may not be interested in having yy as the result of $xx[x := y]$ but rather only $yx[x := y]$. In other words, we only substitute one occurrence of x by y and continue the substitution later. This issue of being able to follow substitution and decide how much to do and how much to postpone, has become a major one in functional language implementation. However, in order to have this spreading control over substitution and to be able to manipulate those partially substituted terms, we must render the latter from being a metalevel notion to an object level notion. It turns out that our new notation will enable such rendering efficiently and will enable the representation of various forms of substitution: *local, global, implicit* and *explicit*.

[19] introduces substitution which is object level but which can evaluate λ-terms fully obtaining the result of the metalevel substitution. More precisely, we introduce the process of *stepwise substitution*, which is meant to refine reduction procedures. Since substitution is the fundamental operation in β-reduction, being in its turn the most important relation in lambda calculus, we are in the heart of the matter. The stepwise substitution is embedded in the calculus, thus giving rise to what is nowadays called *explicit substitution*. It is meant as the final refinement of β-reduction, which has – to our knowledge – not been studied before to this extent. This substitution relation, being the formalization of a process of stepwise substitution, leads to a natural distinction between a global and a local approach. With **global substitution** we mean the intended replacement of a whole class of bound variables (all bound by the same abstraction-λ) by a given term; for **local substitution** we have only one of these occurrences in view. Both kinds of substitution play a role in mathematical applications, global substitution in the case of function application and local substitution for the 'unfolding' of a particular instance of a defined name. We discuss several versions of stepwise substitution and the corresponding reductions. We also extend the usual notion of β-reduction, an extension which is an evident consequence of local substitution. The framework for the description of terms, as explained before, is very adequate for this matter.

6.2 Generalising reduction and term reshuffling

EXAMPLE 6.1
In the classical term $t \equiv ((\lambda_{x_7:X_4}.(\lambda_{x_6:X_3}.\lambda_{x_5:X_1\to X_2}.x_5x_4)x_3)x_2)x_1$, we have the following redexes (the fact that neither x_6 nor x_7 appear as free variables in their respective scopes does not matter here; this is just to keep the example simple and clear):

1. $(\lambda_{x_6:X_3}.\lambda_{x_5:X_1\to X_2}.x_5x_4)x_3$
2. $(\lambda_{x_7:X_4}.(\lambda_{x_6:X_3}.\lambda_{x_5:X_1\to X_2}.x_5x_4)x_3)x_2$

In item notation t becomes $(x_1\delta)(x_2\delta)(X_4\lambda_{x_7})(x_3\delta)(X_3\lambda_{x_6})((X_1 \to X_2)\lambda_{x_5})(x_4\delta)x_5$. Here, the two classical redexes correspond to $\delta\lambda$-pairs as follows:

1. $(\lambda_{x_6:X_3}.\lambda_{x_5:X_1\to X_2}.x_5x_4)x_3$ corresponds to $(x_3\delta)(X_3\lambda_{x_6})$. $((X_1 \to X_2)\lambda_{x_5})(x_4\delta)x_5$ is ignored as it is easily retrievable in item notation. It is the maximal subterm of t to the right of $(X_3\lambda_{x_6})$.

2. $(\lambda_{x_7:X_4}.(\lambda_{x_6:X_3}.\lambda_{x_5:X_1\to X_2}.x_5x_4)x_3)x_2$ corresponds to $(x_2\delta)(X_4\lambda_{x_7})$.
   Again $(x_3\delta)(X_3\lambda_{x_6})((X_1 \to X_2)\lambda_{x_5})(x_4\delta)x_5$ is ignored for the same reason as above.

There is however a third redex which is not immediately visible in the classical term; namely, $(\lambda_{x_5:X_1\to X_2}.x_5x_4)x_1$. Such a redex will only be visible after we have contracted the above two redexes (we will not discuss the order here). In fact, assume we contract the second redex in the first step, and the first redex in the second step. I.e.

$$\begin{array}{ll}((\lambda_{x_7:X_4}.(\lambda_{x_6:X_3}.\lambda_{x_5:X_1\to X_2}.x_5x_4)x_3)x_2)x_1 & \to_\beta \\ ((\lambda_{x_6:X_3}.\lambda_{x_5:X_1\to X_2}.x_5x_4)x_3)x_1 & \to_\beta \\ (\lambda_{x_5:X_1\to X_2}.x_5x_4)x_1 & \to_\beta x_1x_4\end{array}$$

Now, even though all these three redexes are *needed* in order to get the normal form of t, only the first two were visible in the classical term at first sight. The third could only be seen once we had contracted the first two redexes. In item notation, the third redex $(\lambda_{x_5:X_1\to X_2}.x_5x_4)x_1$ is visible as it corresponds to the matching $(x_1\delta)((X_1 \to X_2)\lambda_{x_5})$ where $(x_1\delta)$ and $((X_1 \to X_2)\lambda_{x_5})$ are separated by the segment $(x_2\delta)(X_4\lambda_{x_7})(x_3\delta)(X_3\lambda_{x_6})$. Hence, by extending the notion of a redex from being a δ-item adjacent to a λ-item, to being a matching pair of δ- and λ-items, we can make more redexes visible. This extension furthermore is simple, as in $(t_1\delta)\overline{s}(\rho\lambda_v)$, we say that $(t_1\delta)$ and $(\rho\lambda_v)$ match if $\overline{s}$ has the same structure as a matching composite of opening and closing brackets, each δ-item corresponding to an opening bracket and each λ-item corresponding to a closing bracket. For example, in t above, $(x_1\delta)$ and $((X_1 \to X_2)\lambda_{x_5})$ match as $(x_2\delta)(X_4\lambda_{x_7})(x_3\delta)(X_3\lambda_{x_6})$ has the bracketing structure [][] (see Figure 1 which is drawn ignoring types just for the sake of argument). With

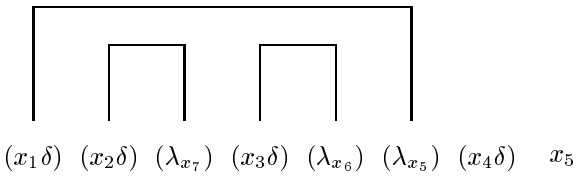


FIG. 1. Redexes in item notation

this extension of redexes, we refine β-reduction in two different ways:

1. By changing (β) from $(t_1\delta)(\rho\lambda_v)t_2 \to_\beta t_2[v := t_1]$ to $(t_1\delta)\overline{s}(\rho\lambda_v)t_2 \rightsquigarrow_\beta \overline{s}(t_2[v := t_1])$ if $(t_1\delta)$ and $(\rho\lambda_v)$ match.
2. By *reshuffling* terms so that matching δ's and λ's occur adjacently. Hence Figure 1 will be redrawn as in Figure 2.

[26] shows that $\rightsquigarrow\!\!\!\!\to_\beta$ (the reflexive transitive closure of $\rightsquigarrow_\beta$) is a generalisation of $\twoheadrightarrow_\beta$. We then show that $\lambda_\to$ with $\rightsquigarrow\!\!\!\!\to_\beta$ satisfies all the desirable typing properties.

[27] extends the Barendregt cube with this generalised reduction and shows that all the above properties hold for this extension. Moreover, [26] shows that term reshuffling is correct. In particular, we show that $\lambda_\to$ accommodated with term reshuffling TS, satisfies the following:

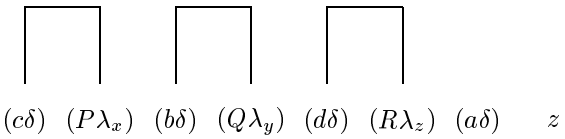


FIG. 2. Term reshuffling in item notation

1. Reshuffling a term, moves all δ's next to their matching λ's.
2. Reshuffling terms preserves $\rightarrow_\beta$. That is, if $t \rightsquigarrow_\beta t'$ then there exists t'' such that $TS(t) \rightarrow_\beta t''$ and $TS(t') \equiv TS(t'')$.
3. Reshuffling terms preserves types. That is, if $\Gamma \vdash t : \rho$ then $\Gamma \vdash TS(t) : \rho$.

6.3 Extending theories with definitions

In many type theories and lambda calculi, there is no possibility to introduce definitions which are abbreviations for large expressions and which can be used several times in a program or a proof. This possibility is essential for practical use, and indeed implementations of Pure Type Systems such as Coq ([7]), Lego ([32]) and HOL ([10]) do provide this possibility. But what are definitions and why are they attractive? Definitions are name abbreviating expressions and occur in contexts where we reason about terms.

EXAMPLE 6.2
Let $id = (\lambda_{x:A}.x) : A \rightarrow A$ *in* $(\lambda_{y:A\rightarrow A}.id)id$ defines id to be $(\lambda_{x:A}.x)$ in a complex expression in which id occurs two times.

The intended meaning of a definition is that the definiendum x can be substituted by the definiens a in the expression b. In a sense, an expression let $x : A$ be a in b is similar to $(\lambda_{x:A}.b)a$. It is not intended however to substitute all the occurrences of x in b by a. Nor is it intended that the definition be a part of our term. Rather, the definition will live in the environment (or context) in which we evaluate or reason about the expression.

One of the advantages of the definition let $x : A$ be a in b over $(\lambda_{x:A}.b)a$ is that it is convenient to have the freedom of substituting only some of the occurrences of an expression in a given formula. Another advantage is that defining x to be a in b can be used to type b. [27] introduces definitions to Barendregt's cube and shows that Church Rosser, Subject Reduction, Unicity of Typing and Strong Normalisation all hold for this extension.

Acknowledgements

I am grateful for the useful comments of Richard Oehrle and Aarne Ranta.

References

[1] Aczel, P., *Non well founded sets*, CSLI lecture notes 14, 1984.

[2] Aczel, P., Properties and propositional functions, privately circulated note, University of Manchester.

[3] Barendregt, H., Lambda calculi with types, *Handbook of logic in Computer Science*, volume II, ed. Abramsky s., Gabbay D.M., Maibaum T.S.E., Oxford University press, 1992.

[4] Bealer, G., *Quality and Concept*, Clarendon Press, 1982.

[5] Bruijn, N.G. de, Generalizing AUTOMATH by means of a lambda-typed lambda calculus, in: *Mathematical Logic and Theoretical Computer Science*, ed. Kueker, D.W., Lopez-Escobar E.G.K., and Smith, C.H. Lecture Notes in Pure and Applied Mathematics, 106, Marcel Dekker, 71-92, 1978.

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

[7] Dowek, G. et al. The Coq proof assistant version 5.6, users guide, rapport de recherche 134, INRIA, 1991.

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

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

[10] Gordon M.J.C. and Melham, T.F. (eds), *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, Cambridge University Press, 1993.

[11] Jacobs, J., Funktionale Satzperspektive und Illokutionssemantik, *Linguistische Berichte* 91, 25–28, 1984.

[12] Hobbs, J.R., Ontological promiscuity, Proceedings of the 23rd Meeting of the Association for Computational Linguistics, 61–69, 1985.

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

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

[15] Kamareddine, F., λ-terms, logic, determiners and quantifiers, *Logic, Language and Information 1 (1)*, 79-103, 1992.

[16] Kamareddine, F., Set Theory and Nominalisation, Part I, *Logic and Computation 2 (5)*, 579-604, 1992.

[17] Kamareddine, F., Set Theory and Nominalisation, Part II, *Logic and Computation 2 (6)*, 687-707, 1992.

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

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

[20] Kamareddine F., Are types needed for Natural Language?, preproceedings for the applied Logic conference, Amsterdam, December 1992. Also in *Applied Logic: What and Why*, edited by Michael Masuch and Laszlo Polas, Kluwer, 1994.

[21] Kamareddine F., Non well foundedness and type freeness can unify the interpretation of functional application, submitted for publication.

[22] A framework for polymorphism and logic, submitted for publication.

[23] Kamareddine, F., and Nederpelt, R.P., A useful lambda notation, submitted for publication.

[24] Kamareddine, F., and Nederpelt, R.P., Canonical Typing and π-conversion in the Barendregt Cube, *Journal of Functional Programming*, 1995 (to appear).

[25] Kamareddine, F., and Nederpelt, R.P., A unified approach to type theory through a refined λ-calculus, paper presented at the 1992 conference on *Mathematical Foundations of Programming Semantics*, also appears in *Theoretical Computer Science* 136:183–216, 1994.

[26] Bloo, R., Kamareddine, F. and Nederpelt, R., Beyond β-reduction in Church's $\lambda_{\rightarrow}$, submitted for publication.

[27] Bloo, R., Kamareddine, F. and Nederpelt, R., The Barendregt Cube with definitions and generalised reduction, submitted for publication.

[28] Kamareddine, F., and Nederpelt, R.P., Generalising reduction in the λ-calculus, submitted for publication.

[29] Kamareddine, F., and Nederpelt, R.P., *The beauty of the λ-calculus*, in preparation.

[30] Kamareddine F., *Polymorphism and Logic in Programming and Natural Languages*, in preparation.

[31] Krifka, M., A compositional semantics for multiple focus constructions, in Moore *et al* (eds.) 1991, pp127–158.

[32] Luo Z., and Pollack, R., LEGO proof development system: User's manual, Technical report ECS-LFCS-92-211, LFCS, University of Edinburgh, 1992.

[33] Martin-Löf, P., An intuitionistic theory of types: predicative part, *Logic Colloquium '73* , Rose and Shepherdson (eds), 1973.

[34] Milner, R., A theory of type polymorphism in programming, *Journal of Computer and System Sciences 17 (3)*, 348-375, 1978.

[35] Milner, R., A proposal for standard ML, in proceedings of *the ACM Symposium on Lisp and Functional Programming*, Austin, 1984.

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

[37] Reddy, U., A Typed Foundation for Directional Logic Programming, Departmental research report No FP-93-2b, Computing Sc, Glasgow University, 1993.

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

Received 23 January 1994