Eindhoven University of Technology

Department of Mathematics and Computing Science

Nominalization, Predication and Type Containment

by

F. Kamareddine and E. Klein

92/23

Computing Science Note 92/23
Eindhoven, September 1992

Copies can be ordered from:
Mrs. F. van Neerven
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

Nominalization, Predication and Type Containment *

Fairouz Kamareddine †
Department of Computing Science
17 Lilybank Gardens
University of Glasgow
Glasgow G12 8QQ
email: `fairouz@dcs.glasgow.ac.uk`

Ewan Klein ‡
Centre for Cognitive Science
2 Buccleuch Place
University of Edinburgh
Edinburgh EH8 9LW
email: `klein@cogsci.ed.ac.uk`

April 13, 1992

1 Introduction

1.1 Hierarchical Types

Type disciplines have featured prominently in formal approaches to natural language since the work of Montague (e.g. [Mon73]). Montague avoided the paradoxes of naive set theory by adopting a version of Russell's cumulative hierarchy of types. Despite the successes of Montague's type system for English, it has met with criticism in recent years for being excessively rigid. One line of research, initiated by Partee and Rooth [RP82, PR83], has tried to achieve greater flexibility, especially in the treatment of quantifiers, by assigning each expression a *family* of types. Another line of work has moved in the direction of type-free theories of properties, in order to accommodate the difficulties raised by nominalization and self-application. In this paper, we will focus our attention on the second of these two endeavours.

Historically, type disciplines for languages have developed in close association with intended models for interpretation. The proposals we shall make can also be construed in this way, inasmuch as they were inspired in part by Aczel's [Acz80] notion of a *Frege structure*, which is intended to provide a consistent formulation of Frege's logical notion of set.

A system of types provides a classificatory scheme for the domain and range of functors. The type of an expression determines the domain in which that expression receives an interpretation. Thus, in (1) (where we use the notation $\alpha{:}\sigma$ to mean that expression α has type σ),

[*]We are grateful to David Beaver, Inge Bethke, Max Cresswell, Lex Holt and an anonymous referee for their comments on previous versions of this paper.

[†]Kamareddine is grateful to the Department of Mathematics and Computing Science, Technical University of Eindhoven, for their financial support and hospitality during the academic year 1991–92.

[‡]Klein's work has been carried out as part of the research programmes of the DYANA project (BR 3175), funded by CEC ESPRIT Basic Research, and of the Human Communication Research Centre, supported by the UK Economic and Social Research Council.

the proper noun Glasgow might be assigned type e, the type of entities, while the predicate fun is assigned type $\langle e,p\rangle$, which we construe as the type of objects which combine with expresions of type e to yield expressions of type p.

(1) Glasgow:e is fun:$\langle e,p\rangle$.

If we make the plausible assumption that the copular verb is here denotes the identity function on predicates, then standard rules of type inference yield the result that (1) is an expression of type p, the type of propositions.

In recent years, the semantic problems of nominalization in linguistically-motivated type theories have received increasing attention, particularly as a result of the work of Bealer, Chierchia and Turner [Bea82, Chi84, Chi85, ChT88, Tur87]. To illustrate, notice that we might want to assign different types to different kinds of syntactic subjects, as shown in the following two examples:

(2) a [Running around the lake]:$\langle e,p\rangle$ is fun:$\langle\langle e,p\rangle,p\rangle$
b [For us to run around the lake]:p is fun:$\langle p,p\rangle$

In (2a), we might expect the gerundive subject phrase to denote a property, hence to be assigned type $\langle e,p\rangle$. But if (2a) is to be of type p, fun will require a new type, namely $\langle\langle e,p\rangle,p\rangle$. Similarly, if the subject of (2b) denotes a proposition, then the type of the predicate has to be changed to $\langle p,p\rangle$. Yet there is no independent linguistic motivation for postulating distinct lexical entries for the different funs of each type.

A related problem arises when we consider cases of self-application, illustrated in (3a) and the simpler (though more artificial) instance (3b).[1]

(3) a [Being fun]:$\langle e,p\rangle$ is fun:$\langle\langle e,p\rangle,p\rangle$
b Fun:$\langle e,p\rangle$ is fun:$\langle\langle e,p\rangle,p\rangle$

Suppose we postulate a first-order predicate fun:$\langle e,p\rangle$, and a second order predicate of predicates fun:$\langle\langle e,p\rangle,p\rangle$. This allows us to deal with (3); but what happens if we want to affirm that fun:$\langle\langle e,p\rangle,p\rangle$ is fun? We are at the bottom of an infinitely ascending ladder of types:

(4) Fun:$\langle\langle e,p\rangle,p\rangle$ is fun:$\langle\langle\langle e,p\rangle,p\rangle,p\rangle$

There seem to be broadly three classes of response to these problems of 'type inflation': type-lowering, type-freedom, and polymorphism. We briefly consider these in turn.

[1] Despite appearances, such locutions are not entirely defined to the discourse of theoreticians; the following sentence was noted in the *Times Higher Education Supplement* of 28b September 1990, p.17:

In fact, the fun of research is more fun than fun.

Type-Lowering

We have just observed the potential difficulties which arise if the subject **running** in (5) is assigned the type $\langle e, p\rangle$ of verb phrases:

(5) Running hurts.

For then we are apparently forced to assign a correspondingly higher type to **runs**. The approach proposed by Chierchia (e.g., in [Chi84]) postulates a nominalisation operator $^\cap$ which maps propositional functions (and propositions) into entities.[2] That is, if **run′** (the semantic translation of **run**—we use Montague's prime notation for semantic constants) denotes a propositional function f, then $^\cap$**run′** is an expression of type e which denotes an individual correlated to f. We might assume that the morphological operation which relates the gerundive form **running** to the finite form **runs** has as its semantic counterpart the introduction of this $^\cap$ operator. The resulting semantic analysis is illustrated in (6):

(6) **hurt′**:$\langle e, p\rangle$($^\cap$**run′**:e)

Type-Freedom

From a technical point of view, it is not necessary to explicitly map propositional functions into their individual correlates. Instead, we can regard all properties as being a special sort of individual. Following Aczel [Acz80], Bealer [Bea82] and others, properties are those first-order objects which can be applied—using an explicit operation *app* of predication—to other objects so as to yield a proposition. This first-order approach is illustrated in (7):

(7) *app*(**hurt′**:e, **run′** :e):p

Although we have declared the types of the expressions in (7), they serve little purpose, since none of them are functional in nature.

Polymorphism

We say that a function is *polymorphic* if it yields appropriate outputs for inputs of a variety of types. There are at least two notions of polymorphism which can be invoked to deal with these problems. The first, called *parametric* polymorphism (cf. [CW85]), obtains polymorphic types by admitting type variables. In Milner's approach [Mil78], as implemented for the programming language ML, types containing type variables are called *generic*. Suppose, for example, that v is a type variable, and that we assign to **fun** the generic type $\langle v, p\rangle$. What happens when we try to determine the type of an expression involving self-application like **fun(fun)**? Assuming that the second occurrence of **fun** has the most general type (i.e.,$\langle v, p\rangle$), the first occurrence will have to be assigned a more complex type, namely $\langle\langle v, p\rangle, p\rangle$, where

[2] One of the earliest discussions of treating propositional arguments in a Montague framework, namely Thomason [Tho76], adopts a similar type-lowering operation.

the type variable v has itself been instantiated as $\langle v, p \rangle$. Although we are required to assign different types to functor and argument in such a case, it should be noted that the complexity of a functor's type is no greater than that required by the most general type of its argument; thus we avoid the 'infinitely ascending ladder of types' alluded to in our discussion of strictly hierarchical type systems. An approach similar in spirit to ML is adopted by Parsons [Pars79], where Montague's framework is modified to allow 'floating' types which again contain type variables. Although Parsons considers an interesting range of data, he does not explicitly discuss problems of nominalization.

A different route avoids type variables by using something which [CW85] call *inclusion* polymorphism. Suppose, for example, that σ_1, σ_2, and τ are types such that $\sigma_2 \preceq \sigma_1$, Xi.e., σ_2 is subsumed by, or *contained* in (cf. [Mit88]), σ_1, and let f be a functor of type $\langle \sigma_1, \tau \rangle$. Suppose further that α is a term, not of type σ_1, but of the more specific type σ_2. Then f is polymorphic in the sense that it can apply to α, and yields a value of type τ. From a semantic point of view, we model a type σ as a set D_σ of values, and containment as inclusion between such sets. Now if a function assigns values to members of a particular set D_{σ_1}, then it will also assign values to members of any subset D_{σ_2} of D_{σ_1}. How does this help us deal with nominalization? Our solution is to let the type $\langle e, p \rangle$ of predicates be contained within the type e of individuals. Then, for example, **fun** of type $\langle e, p \rangle$ can apply to any expression of type $\sigma \preceq e$, including fun itself.

1.2 Individuals, properties and functions

Our treatment takes subsumption polymorphism as a starting point—that is, we will develop a notion of type containment, but avoid type variables. In fact, the formal framework that we develop is flexible enough to encompass a range of different approaches to nominalization, including type-free ones. However, within the space of options, we have made certain theoretical choices which allow us to model certain linguistic generalizations. In this section, therefore, we will consider some of the motivating data.

In order not to prejudge the issues to be decided, we use the term *propositional functor* to refer to any expression f of English which can combine with an argument a so that the result $f(a)$ is a declarative sentence, i.e., capable of being used to assert a proposition. Thus, a finite verb phrase such as **walks** is a propositional functor, as is a declarative sentence lacking a direct object, such as **John annoys** ___. We assume that propositional functors denote propositional *functions*, though just what these are supposed to be is left till later.

We will use the more neutral term *predicative* to cover both propositional functors and words or phrases which intuitively express properties but which cannot combine with other expressions to make sentences. Again, we leave till later what the denotation of predicatives is, if not propositional functions.

The first generalization which we wish to capture is:

Claim 1.1 *Predicative expressions can appear in the position of noun phrase (NP) arguments to propositional functors.*

For example, predicatives can occur in subject position of tensed sentences, i.e., a position which is typically occupied by *NPs*:

(8) a *To run* will tire Mary.
 b *Running* annoys Mary.

Thus, according to our terminology, (8a) contains two predicatives, to run and will tire Mary; the latter is, in addition, a propositional functor.

It can also be observed that the distribution of predicatives sometimes extends beyond that of *NPs*. Thus we have:

Claim 1.2 *Predicative expressions can appear as arguments to propositional functors where NPs are prohibited.*

In particular, certain lexical items are subcategorized to require predicative arguments, as opposed to ordinary noun phrases. The examples in (9) contrast with those in (10):

(9) a John seems *to annoy Mary/happy*
 b With John *annoying Mary/happy/in love*, we can stop worrying.
 c Mary saw John *run/running/happy*

(10) a *John seems *that boy*
 b *With John *that boy*, we can stop worrying.
 c *Mary saw John *that boy*

It might be claimed that this patterning of data is purely syntactic. Certainly, it is true that items which require predicatives are usually subcategorized to take only a subset thereof. Thus, seems takes infinitival complements but not bare or gerundive *VPs*, while see patterns the opposite way. Despite these idiosyncracies, however, there are a variety of generalizations that can only be expressed on the assumption that the class of predicatives can be somehow picked out (cf. [Bach79, PS87]); and it is manifestly desirable to characterize this class semantically instead of invoking some arbitrary syntactic feature. In general, we adopt the position, fundamental to categorial grammar, that syntactic categories should be semantically motivated.

The next two claims have been particularly emphasized by Chierchia [Chi85, ChT88]. Recall Frege's view that a (propositional) function is 'unsaturated', or requires completion by an argument. On completion, the function yields a value, e.g., a proposition. Changing perspective slightly, we can say that only functions have the combinatorial potential to 'glue together' with arguments. The individual correlate of a function, by contrast, is 'inert': it cannot by itself combine with an argument to produce a value. Translated into the realm of grammar, we have:

Claim 1.3 *Tensed predicative expressions are propositional functors, but untensed predicatives are not.*

Thus, the examples in (11) do not express assertible propositions, whereas those in (12) do:

(11) a *John *to run.*
b *John *(be) happy.*

(12) a John *runs.*
b John *is happy.*

This claim, though attractive, seems to require modification when embedded infinitives are considered. Thus, Jacobson [Jac90] has recently drawn attention to data like

(13) Everyone likes their tea to be hot.

The crucial question about such an example is whether the substring **their tea to be hot** is an infinitival sentence (as opposed to a sequence of two distinct complements of **like**). Evidence in favour of there being a single constituent here is provided by standard tests:

(14) a What everyone likes is their tea to be hot.
b Everyone likes their tea to be hot and their beer to be cold.

Despite these examples, the fact that nonfinite verbs cannot combine directly with subjects in root clauses still requires explanation. In the present paper, therefore, we shall maintain Claim 1.3 as it stands, while accepting that further analysis of the issues is called for.

The fourth claim can be regarded as a further specification of Claim 1.1. Chierchia suggests that it is an empirical generalization which holds for many, if not all, natural languages:

Claim 1.4 *Tensed predicative expressions cannot occur as arguments of propositional functors.*

Thus, ungrammaticality results if we attempt to replace the untensed predicatives in our previous examples by tensed predicatives:

(15) a **Runs* annoys Mary
b *John seems *annoys Mary/is happy*
c *John tries *annoys Mary/is happy*

Let us now consider how these observations might be rendered in a formal framework. The generally accepted interpretation of Claim 1.1 is that propositional functions have individual correlates. As a further terminological step, let us use the term *nominal predicatives* to refer to expressions which denote such individual correlates.[3] Our models, derived from Aczel's Frege Structures, contain a collection $\mathcal{F}_0$ of individuals (or objects, in Aczel's terminology),

[3] We eschew the term 'nominalization' because we do not wish to claim that such expressions have undergone any change of syntactic category, at least in the way this is usually understood.

and this set is 'big enough' to contain, for each function from objects to objects, an object that corresponds to that function. The collection of propositional functions, i.e., functions from $\mathcal{F}_0$ to propositions (which are also objects) is called PF. We implement the idea of individual correlates by letting the collection PF be explicitly mapped, *via* the λ operator, onto a subcollection SET of the domain of $\mathcal{F}_0$ individual objects. That is, each object in SET is the individual correlate of a propositional function. (See Lemma 2 for a proof that λ is bijective.)

Claim 1.2 shows that some lexical items select as their arguments nominal predicatives, which we take to denote objects in SET. Since we want to treat this selection as a kind of semantic dependency, this means that our type system should give us a type of those individuals which belong to SET.[4] We shall let $\langle e, p\rangle$ be the type required and, as we shall see later, we arrange things so that $\langle e, p\rangle$ is a *subtype* of the type e of individuals. Moreover, expressions which select objects of a particular type will be encoded as functions over the appropriate domain; consequently, we will have to allow λ-bound variables to be of type $\langle e, p\rangle$.[5]

If Claim 1.3 is given a semantic explanation, then we must capture the difference in combinatorial potential between tensed and untensed predicatives. As pointed out by Chierchia and Turner ([ChT88]), this distinction appears to be inadequately captured by first-order theories of properties such as that of Bealer [Bea82] in which propositions only result by virtue of explicitly applying a property to another object. For example, on such an approach, John walks would be expressed as (16a) or, adopting the approach of building the collection of individual correlates within $\mathcal{F}_0$, as (16b):

(16) a $app(\text{walk}':e, \text{john}':e)$
b $app(\text{walk}':\langle e, p\rangle, \text{john}':e)$

The Fregean view (which is vigorously disputed by Bealer [Bea82, Bea89]) holds that propositional functions should not be thought of as objects, but indeed as functions. This is reflected in our framework, therefore, by the decision to view propositional functions as elements of PF, not $\mathcal{F}_0$. This has the virtue of providing a natural explanation for Claim 1.4. For although elements of PF do have individual correlates in $\mathcal{F}_0$, they are not themselves objects, and as such are not potential arguments for other propositional functions.

As we will see, 'nominal' types (including predicative nominals) are all constructed as subtypes of e. Since, according to what we have just said, propositional functors are not nominals, they cannot be assigned a nominal type. We therefore require a new kind of type for such functors, one which is *not* a subtype of e. Expressions whose denotations lie outside the domain $\mathcal{F}_0$ of objects will be assigned what we call *metatypes*. Whenever σ and τ are (meta-)types, $(\sigma \rightarrow \tau)$ will be a metatype. Note that we will not need to quantify over propositional functions, nor will we need λ-expressions whose domain of interpretation is the collection of propositional functions—we can use nominalised properties instead! Hence, variables in our language will never be assigned metatypes.

[4] Despite the precedent of [ChT88], we have refrained from using the term 'sort' rather than 'type' for classifications of basic objects.

[5] Thus, our type $\langle e, p\rangle$ is equivalent to [ChT88]'s sort *nf* of nominalized functions.

We shall assume that uninflected (or base form) verb phrases denote objects rather than propositional functions; for example, **walk** will be of type $\langle e, p\rangle$. When verb phrases receive tense, they are mapped by a predication operator $^{\cup}$ into propositional functions, with the metatype $(e \rightarrow p)$. Thus if nonfinite **walk** translates as **walk**$'$:e, then tensed **walks** translates as $^{\cup}$**walk**$'$:$(e \rightarrow p)$. Putting the various pieces together, we replace (16) with (17), where the propositional functor is applied directly to its argument, rather than by the mediation of *app*:

(17) $[^{\cup}\mathbf{walk}'{:}(e \rightarrow p)(\mathbf{john}'{:}e)]{:}p$

By way of summary, we give the following tabular presentation of our articulation of the data:

Syntactic Notion	Semantic Notion	Domain	(Meta-)Type	Example
propositional functor	propositional function	PF	$(e \rightarrow p)$	**walks, is fun**
nominal predicative	set	SET	$\langle e, p\rangle$	**walk, be fun**

In this section, we have attempted to present and motivate the general structure of our approach, and it will be observed that we have followed [ChT88] closely in favouring a Fregean analysis over a first order property theory. Nevertheless, our formal framework differs from that of [ChT88] in many respects; this will become obvious in the following sections, where we give a more systematic presentation of the theory.

2 The Language $\mathcal{L}_{\preceq}$

2.1 Judgements and Type Containment

In the theory $\mathcal{L}_{\preceq}$ developed in this paper, we follow [Acz80] in starting from models of the type-free lambda calculus, on top of which an interpretation for logical connectives has been constructed; we then construct types within the set of objects. In place of the domain $\{0, 1\}$ of truth values, we have a domain PROP of propositions, included in which is the domain TRUTH of true propositions. These collections provide values for the types p and t respectively. As mentioned earlier, there is also a domain $\mathcal{F}_0$ of individuals, with associated type e. This domain turns out to be much richer than one might have expected. Indeed, it contains PROP (and hence TRUTH) as subcollections. In § 4, we shall look in more detail at the intended models; for the time being, however, we present the type structure.

Following usual practice in type theory (e.g., [CW85, Mit88]), we use a natural deduction format for rules of type inference. A simple example is the following:

$$\frac{\vdash \varphi{:}p}{\vdash \exists x{:}\sigma.\varphi{:}p}$$

The statement $\vdash \varphi{:}p$ is an assertion or *judgement* meaning that we can infer that φ is of type p. The rule as a whole is a logical implication; given the premiss, we can infer that $\exists x.\varphi$ is also of type p.

What we have presented is not quite sufficient, however; if φ contains occurrences of the variable x, the inference that it is of type p may in turn depend on the type of x; in other words, the judgement is made under the assumption, or in the context, $x{:}\sigma$. Using

$$\Gamma, x{:}\sigma$$

to represent a context Γ which contains the relevant assumption, we replace our earlier rule by the following:

$$\frac{\Gamma, x{:}\sigma \vdash \varphi{:}p}{\Gamma \vdash \exists x{:}\sigma.\varphi{:}p}$$

Let us now present these ideas in a more systematic format. A *type statement* is a pair, written $\alpha{:}\sigma$, consisting of an expression α and a type σ, read "α has type σ"; α is said to be the *subject* of the statement. A *signature* Σ is a finite set of distinct type statements the subjects of which are constants, while a *context* Γ is a finite set of distinct type statements, the subjects of which are variables or sentences. In the latter case, a statement of the form $\varphi{:}t$ indicates that φ is a sentence of the logic whose truth is being assumed in the course of a proof; that is, we are also using contexts in a sequent calculus style to encode the current set of assumptions required at each line of a proof.

As usual, we can regard signatures and contexts as functions from expressions to types. Thus, $dom(\Sigma)$ denotes the set of expressions to which the signature Σ assigns a type, and similarly for contexts. If $\mathcal{A}$ is a signature or a context, we write $\mathcal{A}, \alpha{:}\sigma$ in place of $\mathcal{A} \cup \{\alpha{:}\sigma\}$.

Although the system used here does not use the power of higher-order type theory (e.g., such as dependent types), we have nevertheless found it convenient to take as our framework the theory of expressions developed in the Edinburgh Logical Framework [HHP87]. As pointed out in the preceding section, we distinguish types, whose interpretations are constructed within the domain of objects, from metatypes, which have a disjoint interpretation as collections of functions and functionals. Types and metatypes are both *kinds*.

We need three further kinds, or classifications of types: *non-propositional types* (np-types for short), *fixed point types* (fp-types) and *well-behaved types* (wb-types); these are all interpreted within the domain of objects. As we shall see later, there is a sense in which an fp-type is a complex type which does not have any proper subtypes.

We will use σ and τ for types, $m\sigma$ for metatypes, and η, η_1, η_2 to range over both types and metatypes. We use c for constants (a special instance of which is $\perp$), x, y for variables, α, β 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	::=	*type* | *fp-type* | *np-type* | *wb-type* | *metatype*
Types	σ	::=	$e \mid t \mid p \mid \langle\sigma,\tau\rangle$
Metatypes	$m\sigma$	::=	$(\eta_1 \rightarrow \eta_2)$
Expressions	α	::=	$c \mid x \mid \lambda x{:}\sigma.\alpha \mid app(\alpha,\beta) \mid \neg\alpha \mid [\alpha \wedge \beta] \mid [\alpha \vee \beta]$ $\mid [\alpha \supset \beta] \mid [\alpha = \beta] \mid \forall x{:}\sigma.\alpha \mid \exists x{:}\sigma.\alpha$

We will omit square brackets around complex sentences except in those cases where the scope of a typing statement needs to be made explicit.

Type theory (cf. [M-L79]) provides rules for making judgements of various forms. The ones which we are concerned with are the following:

Judgements

$\vdash \Sigma$ *sig*	Σ *is a signature*
$\vdash_\Sigma \Gamma$ *context*	Γ *is a context*
$\Gamma \vdash_\Sigma \eta\ K$	η *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 $\preceq$, the containment relation.

We mentioned earlier that the inference rules by which judgements can be derived are formulated in natural deduction notation. We add glosses to a representative sample of the rules in order to help readers not familiar with this mode of presentation.

Valid Signature

(*null sig*) $$\frac{}{\vdash \emptyset\ \mathit{sig}}$$

The empty relation is a signature.

(: *sig*) $$\frac{\vdash \Sigma\ \mathit{sig} \qquad \vdash_\Sigma \eta\ K}{\vdash \Sigma, c{:}\eta\ \mathit{sig}} \quad \mathit{if}\ c \notin\ \mathit{dom}\ (\Sigma)$$

If η is a kind, and Σ doesn't already assign a (meta-)type to the constant c, then we can augment Σ with the statement c:η.

Valid Context

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

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

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

It should be noted that (: *context*) requires σ to be a type, not an arbitrary kind; thus, our contexts will not assign metatypes to any variables.

As we pointed out above, the following semantic domains are ordered by inclusion:

$$\text{TRUTH} \subseteq \text{PROP} \subseteq \mathcal{F}_0$$
$$\text{SET} \subseteq \mathcal{F}_0$$

And indeed there are other inclusions in the domains. This structure is reflected by the *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:

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

Rules for inferring judgements about containment will be given shortly. Before that, however, we present the various kinds required. Within the class of types, we distinguish three useful subsets: *non-propositional (np-type)*, *fixed-point (fp-type)*, and *well-behaved (wb-type)*. These are characterized in the following rules:

Kinds, Types and Metatypes

$$(\textit{base types}) \quad \frac{\vdash \Sigma\ \textit{sig} \qquad \vdash_\Sigma \Gamma\ \textit{context}}{\Gamma \vdash_\Sigma e\ \textit{type}}$$

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

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

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

(*np base*) $$\frac{\vdash\Sigma\ \mathit{sig} \qquad \vdash_\Sigma \Gamma\ \mathit{context}}{\Gamma\vdash_\Sigma e\ \mathit{np\text{-}type}}$$

(*np complex*) $$\frac{\Gamma\vdash_\Sigma \sigma\ \mathit{np\text{-}type} \qquad \Gamma\vdash_\Sigma \tau\ \mathit{type}}{\Gamma\vdash_\Sigma \langle\sigma,\tau\rangle\ \mathit{np\text{-}type}}$$

(*wb-types*) $$\frac{\Gamma\vdash_\Sigma \tau\ \mathit{np\text{-}type} \qquad \Gamma\vdash_\Sigma \sigma\ \mathit{type} \qquad \Gamma\vdash_\Sigma \tau\preceq\sigma}{\Gamma\vdash_\Sigma \langle\sigma,\tau\rangle\ \mathit{wb\text{-}type}}$$

(*fp-types*) $$\frac{\Gamma\vdash_\Sigma \sigma\ \mathit{type} \qquad \Gamma\vdash_\Sigma \tau\preceq p}{\Gamma\vdash_\Sigma \langle\langle\sigma,\tau\rangle,\tau\rangle\ \mathit{fp\text{-}type}}$$

If σ is a type and τ is contained in p (that is, $\tau = t$ or p), then $\langle\langle\sigma,\tau\rangle,\tau\rangle$ is an fp-type.

(*metatypes*) $$\frac{\Gamma\vdash_\Sigma \eta_1\ K \qquad \Gamma\vdash_\Sigma \eta_2\ K}{\Gamma\vdash_\Sigma (\eta_1 \rightarrow \eta_2)\ \mathit{metatype}}$$

There is a complementarity between np-types and fp-types, in the following sense. From $\Gamma\vdash_\Sigma \tau\ \ \mathit{np\text{-}type}$, we can conclude that $\Gamma \not\vdash_\Sigma \langle\langle\sigma,\tau\rangle,\tau\rangle\ \mathit{fp\text{-}type}$; for example, $\langle\langle\sigma,e\rangle,e\rangle$ is not an fp-type. Conversely, from $\Gamma\vdash_\Sigma \langle\langle\sigma,\tau\rangle,\tau\rangle\ \mathit{fp\text{-}type}$, we can conclude that $\Gamma \not\vdash_\Sigma \tau\ \ \mathit{np\text{-}type}$; for example, $\langle\langle\sigma,p\rangle,p\rangle$ is an fp-type, but p is not an np-type. Note, however, that np-types and fp-types are not mutually exclusive; for example, $\langle\langle e,p\rangle,p\rangle$ is both.

As we will see later, λ-abstraction will only be permitted when the type of the abstract is a wb-type. A complex type is a wb-type just in case the range is an np-type which is contained in the domain type. For example, $\langle e,e\rangle$, $\langle e,\langle e,e\rangle\rangle$, $\langle e,\langle e,p\rangle\rangle$ and $\langle\langle e,p\rangle,\langle e,p\rangle\rangle$ are wb-types, but $\langle e,p\rangle$ and $\langle p,e\rangle$ are not.

The containment relation is governed by the following conditions:[6]

Containment

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

Objects in the domain D_σ of any type σ are also in D_e.

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

Truths are propositions.

[6] For a similar proposal, see [CG89].

$$(Dom\preceq)\quad \frac{\Gamma\vdash_\Sigma \sigma_1\preceq\sigma_2}{\Gamma\vdash_\Sigma \langle\sigma_2,e\rangle\preceq\langle\sigma_1,e\rangle}$$

Every function (returning arguments in D_e) defined on a domain D_{σ_2} is also defined on subsets D_{σ_1} of D_{σ_2}.

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

Every function with values in the range D_{τ_1} also yields values in supersets D_{τ_2} of D_{τ_1}.

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

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

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

$$(Fix\preceq)\quad \frac{\Gamma\vdash_\Sigma \langle\langle\sigma,\tau\rangle,\tau\rangle\ fp\text{-}type}{\Gamma\vdash_\Sigma \langle\langle\sigma,\tau\rangle,\tau\rangle\approx\langle\sigma,\tau\rangle}$$

The axiom ($Fix \preceq$) gives us fixed points for type containment. That is, if $\tau \preceq p$, then $\langle\sigma,\tau\rangle \approx \langle\langle\sigma,\tau\rangle,\tau\rangle \approx \langle\langle\langle\sigma,\tau\rangle,\tau\rangle,\tau\rangle \ldots$. While types such as $\langle e,e\rangle$, $\langle\langle e,e\rangle,e\rangle$, $\langle\langle\langle e,e\rangle,e\rangle,e\rangle$, ...are distinct, we need to be more restrictive about types such as $\langle e,p\rangle$, $\langle\langle e,p\rangle,p\rangle$, ...if we are to avoid the paradoxes. According to ($Dom \preceq$), since $\langle e,p\rangle \preceq e$, we should have $\langle e,p\rangle \preceq \langle\langle e,p\rangle,p\rangle$. The intuition behind calling $\langle\langle e,p\rangle,p\rangle$ an fp-type is that this containment is not proper; that is, we cannot get anything extra by going from $\langle e,p\rangle$ to $\langle\langle e,p\rangle,p\rangle$. In other words, we can only map sets into propositions to the extent that we map those sets *qua* objects into propositions.

As already noted, the containment relation plays a central role in our approach to polymorphism. In §4, we shall show that there are models of the typing system; that is, we will have functional domains from D_σ to D_τ which are included in D_e; moreover, when $D_\tau \subseteq$ PROP, we also have the result that objects in the function space domain 'D_σ to D_τ' are in D_σ.

Not all functions can be mapped down into the collection of objects, and following Aczel [Acz80], we shall call these *functionals*. That is, adopting Frege's correlation thesis [Fre77], we will see that all we need in the formal theory are objects, functions and functionals and that functions at a higher level than those three can be mapped down to the lower domains. Among the functionals we will count the interpretations of determiners and logical connectives—and indeed, these are expressions which do not admit of nominalization.

2.2 Type Inference Rules

In the preceding subsection, we gave a definition of the syntax of expressions occurring in judgements. These definitions were deliberately general, and could encompass a variety of logical systems. In specifying a particular calculus, such as $\mathcal{L}_{\preceq}$, we need to make explicit how the types of expressions of $\mathcal{L}_{\preceq}$ are inferred. It is to this task that we now turn.

The signature Σ of $\mathcal{L}_{\preceq}$ contains a finite number of statements $c{:}\eta$ which assign types and metatypes to constants of the language. For now, we are only concerned with logical constants and functionals:

Signature of $\mathcal{L}_{\preceq}$

$$\bot{:}p$$

$$\begin{array}{cc}
\neg{:}(e \to e) & \lambda{:}(e \to e) \\
\wedge{:}(e \to (e \to e)) & \vee{:}(e \to (e \to e)) \\
\supset{:}(e \to (e \to e)) & ={:}(e \to (e \to e)) \\
{}^{\cup}{:}(e \to (e \to e)) & app{:}(e \to (e \to e)) \\
\forall{:}((e \to e) \to e) & \exists{:}((e \to e) \to e)
\end{array}$$

Two comments on the above are called for. First, it will be noticed that, for example, $\neg$ is interpreted as a functional which maps *any* object in $\mathcal{F}_0$ into another such object; we cannot tell, for a given expression α, whether $\neg\alpha$ is a proposition unless we have some way of proving that α itself is a proposition. This will be made explicit in the axioms for type inference given below. Second, we will use conventional notation for the syntax of the various constants, writing $\varphi \wedge \psi$ in place of $\wedge(\varphi)(\psi)$, $app(x, y)$ in place of $app(x)(y)$, and $\forall x.\varphi$ in place of $\forall(\lambda x.\varphi)$.

A context Γ for $\mathcal{L}_{\preceq}$ contains a finite number of statements of the form $x{:}\sigma$, for any type σ. Recall however that Γ never assigns metatypes to variables.

Before launching into the type inference rules, we first define substitution on expressions, where we take $\alpha[\beta/x]$ to be the result of substituting β for all free occurrences of x in α.

$$\begin{array}{rcl}
x[\beta/x] & \equiv & \beta \\
x[\beta/y] & \equiv & x \text{ if } x \not\equiv y \\
c[\beta/x] & \equiv & c \\
(\lambda x.\alpha)[\beta/x] & \equiv & \lambda x.\alpha \\
(\lambda x.\alpha)[\beta/y] & \equiv & \lambda x.\alpha[\beta/y] \text{ if } x \not\equiv y \text{ and } x \text{ not free in } \beta \\
(\lambda x.\alpha)[\beta/y] & \equiv & \lambda z.\alpha[z/x][\beta/y] \text{ if } x \not\equiv y \text{ and } x \text{ is free in } \beta \text{ and } z \text{ is not free in } \alpha \text{ or } \beta \\
\Theta_1(\alpha)[\beta/x] & \equiv & \Theta_1(\alpha[\beta/x]), \text{ where } \Theta_1 \text{ is } \neg, {}^{\cup}, \wedge, \vee, \supset, =, app, \forall, \text{ or } \exists
\end{array}$$

The other clauses for substitution in logically complex expressions carry on as usual.

The next definition serves the following functions:

1. It gives rules by which the type of an arbitrary expression of $\mathcal{L}_{\preceq}$ can be inferred.
2. It exploits the type t of truths to give introduction (I) and elimination (E) rules for the logical connectives in $\mathcal{L}_{\preceq}$.

Definition 1 (Type Inference for $\mathcal{L}_{\preceq}$)

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

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

$$(\lambda) \quad \frac{\Gamma, x{:}\sigma \vdash_\Sigma \alpha{:}\tau \qquad \Gamma \vdash_\Sigma \langle\sigma,\tau\rangle\ wb\text{-}type}{\Gamma \vdash_\Sigma (\lambda x{:}\sigma.\alpha){:}\langle\sigma,\tau\rangle}$$

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

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

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

$$(= prop) \quad \frac{\Gamma \vdash_\Sigma \alpha{:}\eta \qquad \Gamma \vdash_\Sigma \beta{:}\eta}{\Gamma \vdash_\Sigma [\alpha = \beta]{:}p}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(∀*I*) $$\frac{\Gamma,x{:}\sigma\vdash_\Sigma \varphi{:}t}{\Gamma\vdash_\Sigma \forall x{:}\sigma.\varphi{:}t}$$ *where x is not free in φ or any assumptions in Γ*

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

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

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

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

Although most of these clauses are standard, it should perhaps be pointed out that the definition (⊃*prop*) of implication is somewhat unusual; following [Acz80]'s proposal, it has the consequence that if the antecedent φ of a conditional is not true, then $\varphi\supset\psi$ is a proposition whatever object ψ is.

2.3 Equality Axioms

We now give a set of equality axioms which are similar to those of the λ-calculus, except that we allow self-application and polymorphism. Note however that self-application is only possible for those expressions which have a complex type; indeed, this is what is required by clause (*app*) of the syntax above.

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

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

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

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

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

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

2.4 Russell's and Curry's Paradoxes

It might be thought that the theory presented above would fall foul of Russell's paradox, due to the fact that $\neg app(x,x)$ is a well-formed formula for x of any type $\langle\sigma,\tau\rangle$; hence by abstracting over $\neg app(x,x)$, we could obtain the equality

$$app(a,a) = \neg app(a,a)$$

where a is $\lambda x.\neg app(x,x)$.

For example, given the following proof,

$$\dfrac{\dfrac{\Gamma, x{:}\langle e,p\rangle\vdash_\Sigma x{:}\langle e,p\rangle \qquad \dfrac{\Gamma, x{:}\langle e,p\rangle\vdash_\Sigma x{:}\langle e,p\rangle}{\Gamma, x{:}\langle e,p\rangle\vdash_\Sigma x{:}e}(Contain)}{\Gamma, x{:}\langle e,p\rangle\vdash_\Sigma app(x,x){:}p}(app)}{\Gamma, x{:}\langle e,p\rangle\vdash_\Sigma \neg app(x,x){:}p}(\neg prop)$$

we might conclude that we can set a equal to the abstraction

$$\lambda x{:}\langle e,p\rangle.\neg app(x,x){:}\langle\langle e,p\rangle,p\rangle$$

and infer that $app(a,a)$ is of type p, leading to a contradictory proposition from the above equality.

However, one of the steps necessary to derive this contradiction is incorrect. That is, even if x is of type $\langle e,p\rangle$, and even though $\neg app(x,x)$ is a proposition, $\lambda x{:}\langle e,p\rangle.\neg app(x,x)$ is not typable in $\mathcal{L}_{\preceq}$. More specifically, it is excluded by virtue of clause (λ) in the definition of type inference, since we cannot derive $\Gamma\vdash_\Sigma p$ *np-type*, and hence cannot derive that $\langle\langle e,p\rangle,p\rangle$ is a wb-type.

In fact we have a more general result: the paradox does not arise for x of any type $\langle\sigma,\tau\rangle$, where $\tau = t$ or p. This is a consequence of the following lemma.

Lemma 1 *If x is of type $\langle\sigma,\tau\rangle$, $\tau = t$ or p, then $\lambda x{:}\langle\sigma,\tau\rangle.\neg app(x,x)$ is not typable.*

Proof *According to the definition of type inference for $\mathcal{L}_{\preceq}$, it is enough to show that we cannot derive $\Gamma\vdash_\Sigma p$ np-type or $\Gamma\vdash_\Sigma t$ np-type. This is obvious.*

□

Our manner of avoiding the paradox is somewhat new, we believe. It is similar to Russell's own approach in that type constraints are invoked to limit abstraction, but differs of course with respect to the non-hierarchical nature of the type system. Unlike Aczel [Acz80], we do not take the step of questioning the propositionhood of $app(a,a)$; and unlike Turner [Tur87], we do not restrict the axiom of β-conversion.

Let us turn now to the question of Curry's paradox. Recall the Deduction Theorem (in fact, our rule $(\supset I)$):

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

If we take a to be the formula

$$\lambda x{:}\sigma[app(x,x)\supset\bot],$$

then by β-conversion we derive

$$(ID)\ app(a,a) = [app(a,a)\supset\bot].$$

Now, it holds trivially that

$$app(a,a){:}t\vdash_\Sigma app(a,a){:}t.$$

Hence, by (ID) we derive

$$app(a,a){:}t\vdash_\Sigma [app(a,a)\supset\bot]{:}t,$$

and by $(\supset E)$ we get

$app(a,a){:}t \vdash_\Sigma \bot{:}t.$

In order to derive by (DT) that

$\vdash_\Sigma [app(a,a) \supset \bot]{:}t$

we must first be able to show

$\emptyset \vdash_\Sigma app(a,a){:}p,$

(where $\emptyset$ is the empty context). For we can derive the latter, we can use it in the following step:

$$\frac{app(a,a){:}t \vdash_\Sigma \bot{:}t \qquad \emptyset \vdash_\Sigma app(a,a){:}p}{\emptyset \vdash_\Sigma [app(a,a) \supset \bot]{:}t} \quad (\supset prop)$$

and also by (ID) and $(= E)$

$\emptyset \vdash_\Sigma app(a,a){:}t.$

Given the last two steps, we can again apply $(\supset E)$ to get

$\vdash_\Sigma \bot{:}t.$

The proof only goes through, however, if $\emptyset \vdash_\Sigma app(a,a){:}p$ is derivable. For this, we would have to assign the type $\langle\sigma,p\rangle$ to a, i.e., to $\lambda x{:}\sigma[app(x,x) \supset \bot]$. How could we show that

(18) $\vdash_\Sigma \lambda x{:}\sigma[app(x,x) \supset \bot]{:}\langle\sigma,p\rangle$?

This can only be the last step of an inference involving the rules (λ) or $(Contain)$. We consider the two cases in turn.

Case 1: The premisses of the inference must include $x{:}\sigma \vdash_\Sigma [app(x,x) \supset \bot]{:}p$, which in turn is only derivable by $(\supset prop)$ from the assumption that $x{:}\sigma \vdash_\Sigma app(x,x){:}p$. However, we can only prove the latter if for some σ', $\sigma = \langle\sigma',p\rangle$, where $\langle\sigma',p\rangle \preceq \sigma'$. But in this case, we would have to show that $\Gamma \vdash_\Sigma p$ *np-type*, which is impossible.

Case 2: We have to find some type τ such that $\tau \preceq \langle\sigma,p\rangle$ and $x{:}\sigma \vdash_\Sigma \lambda x{:}\sigma[app(x,x) \supset \bot]{:}\tau$. The only applicable derivation rule is $(Ran \preceq)$, setting τ to be $\langle\sigma,t\rangle$. However, the judgement $\vdash_\Sigma \lambda x{:}\sigma[app(x,x) \supset \bot]{:}\langle\sigma,t\rangle$ is not derivable, for the same reasons as those given in Case 1 above, since we cannot show that $\Gamma \vdash_\Sigma p$ *np-type*.

3 Models of $\mathcal{L}_{\preceq}$

3.1 Frege Structures

As pointed out earlier, our models are constructed using the notion of a Frege structure as defined by Aczel [Acz80]. We begin with a collection $\mathcal{F}_0$ of objects, and for each natural number $n \geq 1$, we define $\mathcal{F}_n$ as $\{f : \mathcal{F}_0{}^n \mapsto \mathcal{F}_0\}$, where $\mathcal{F}_0{}^n = \overbrace{\mathcal{F}_0 \times \mathcal{F}_0 \times \cdots \mathcal{F}_0}^{n\ times}$. In particular, $\mathcal{F}_1$ is the set of all unary functions from $\mathcal{F}_0$ to $\mathcal{F}_0$. Within $\mathcal{F}_0$ we pick out PROP, the collection of propositions, and TRUTH, the collection of all true propositions. (Thus, Aczel makes a crucial departure from Frege in denying that all true propositions can be identified with the True).

So far, then our Frege structures contain objects, functions, propositions and truths. To these, we need to add functionals, logical connectives, and some closure conditions. We now show how they are supplied.

Two functionals are required in order to provide a model for the lambda calculus:[7]

$$\begin{aligned}&\lambda : \mathcal{F}_1 \mapsto \mathcal{F}_0\\ &\mathbf{app} : \mathcal{F}_0 \times \mathcal{F}_0 \mapsto \mathcal{F}_0\end{aligned}$$

These obey a *comprehension* principle such that whenever f is a propositional function in $\mathcal{F}_1$, i.e., f is an element of $\mathcal{F}_1$ which maps its arguments into PROP, then

$$\mathbf{app}(\lambda x.f(x), a) = f(a).$$

Let PF be the collection of unary propositional functions in a Frege structure:

$$\text{PF} = \{f \in \mathcal{F}_1 | \text{for all } x\ in \mathcal{F}_0, f(x) \text{ is in PROP}\}$$

We can now identify a further subcollection of $\mathcal{F}_0$, namely SET, as the individual correlates of propositional functions under λ:

Definition 2 (Sets) *An object is in* SET *iff it is* λf *for some* f *in* PF.

The distinguishing characteristic of sets (i.e. elements of SET) is that they can be predicated of any object in $\mathcal{F}_0$ to yield a proposition:

Definition 3 (Predication) *If* a *is in* SET, *then* $\mathbf{app}(a, b)$ *is in* PROP, *for any object* b *in* $\mathcal{F}_0$.

Comprehension can be restated as follows:

[7] We adopt the notational convention of using boldface terms to denote elements of the model, reserving italics for expressions of the object language. For example, **app** is a functional in the model which corresponds to the functor *app* in the language.

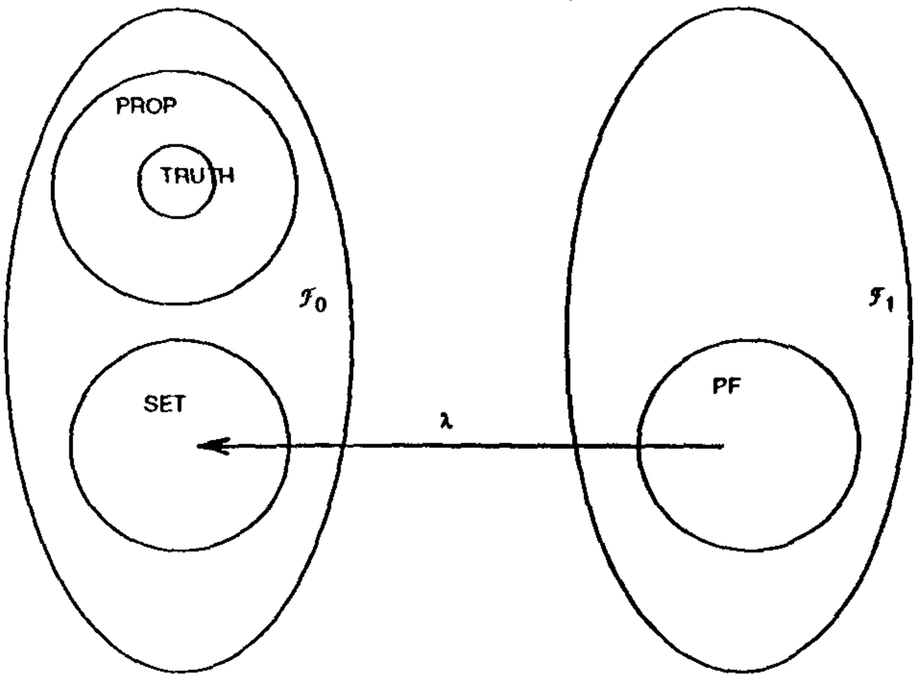

Figure 1: The functional λ

Definition 4 (Comprehension) *If f is in* PF, *then λf is a set a such that for any object b,* app(a, b) *is in* PROP, *and* app(a, b) *is in* TRUTH *iff $f(b)$ is in* TRUTH.

The notions that we have introduced so far—objects, functions, PROP, TRUTH, SET, comprehension and predication—are based on a model of the λ-calculus. In order to ensure that they have the properties we want, our models should also contain a logic. We know that such a construction is not straightforward; for instance, logic cannot be built in a simple way on the top of Scott domains (cf. [Sco76]). The construction provided by Aczel inductively increases the two basic collections of propositions and truths, and the fixed point theorem is then applied to provide the limit of these newly obtained collections, resulting in PROP and TRUTH. Hence PROP is closed under all the logical connectives $\wedge, \vee, \neg, \ldots$ (more strictly, the functionals corresponding to these connectives) and TRUTH is the collection of all true propositions. The organization of $\mathcal{F}_0$ and $\mathcal{F}_1$ in a Frege structure is illustrated in Figure 1.

We now need to ensure that we have full abstraction. That is, if $\xi[x_1, x_2, \ldots, x_n]$ is an expression formed out of objects, functions, functionals and variables (ranging over objects in a Frege structure), where $x_1, x_2, \ldots, x_n$ are free variables of ξ, then there is a function f in the Frege structure such that $f(a_1, a_2, \ldots, a_n) = \xi[a_1/x_1, a_2/x_2, \ldots, a_n/x_n]$, where the substition of objects for variables is simultaneous.

We assume our construction is based on the model E_∞ of the untyped λ-calculus [Sco76]. We then take

$$B_p = \{0, 1\} \subseteq \mathcal{F}_0 = E_\infty.$$

We next construct the logical constants so that PROP is the smallest set containing B_p which is closed under the relevant clauses for connectives presented in Definition 1. Here, we give just two examples.

($\wedge$ schema) If φ is in PROP and ψ is in PROP, then $\varphi \wedge \psi$ is in PROP, and $\varphi \wedge \psi$ is in TRUTH iff φ is in TRUTH and ψ is in TRUTH.

($\forall$ schema) For all n-ary functions f, if $f(x_1, x_2, \ldots, x_n)$ is in PROP for all $\langle x_1, x_2, \ldots, x_n\rangle \in \mathcal{F}_0{}^n$, then $\forall f$ is in PROP, and $\forall f$ is in TRUTH iff $f(a_1, a_2, \ldots, a_n)$ is in TRUTH for all $\langle a_1, a_2, \ldots, a_n\rangle \in \mathcal{F}_0{}^n$.

Whenever φ is a wff open in x, we understand $\langle\varphi|x\rangle$ to be the function f in $\mathcal{F}_1$ such that for any a in $\mathcal{F}_0$, $f(a) = \varphi[a/x]$. Since we have full abstraction, we can assume that f exists. Now, we take $|\ |: \mathcal{F}_0 \mapsto \mathcal{F}_1$ to be the functional such that

$$|a| = \langle \mathrm{app}(a, x) \mid x\rangle.$$

In general, we define $|\ |_n : \mathcal{F}_0 \mapsto \mathcal{F}_n$ such that

$$|\ a\ |_n = \langle \mathrm{app}_n(a, \vec{x}) \mid \vec{x}\rangle$$

where $\mathrm{app}_1 = \mathrm{app}$ and $\mathrm{app}_{n+1}(a, b, \vec{b}) = \mathrm{app}_n(\mathrm{app}(a, b), \vec{b})$, and $\vec{x}, \vec{b}$ are sequences of n variables or elements of $\mathcal{F}_0$. Now,

$$\lambda_n^m : \mathcal{F}_m \mapsto \mathcal{F}_n, m > n$$

is defined inductively, for $\vec{a} = a_1, \ldots, a_n$ as

$$\begin{aligned} (\lambda_n^{n+1} f)(\vec{a}) &= \lambda\langle f(\vec{a}, x) \mid x\rangle \\ (\lambda_n^{n+m+1} f)(\vec{a}) &= \lambda_n^{n+1}(\lambda_{n+1}^{n+m+1} f) \end{aligned}$$

In particular, λ_0^n nominalizes an n-ary function f returning $\lambda_0^n f$ in $\mathcal{F}_0$.

We take

$$\text{SET} = \{\lambda_0^n f | f \text{ is any propositional function}\}.$$

Definition 5 *A* Frege Structure *is a triple $\mathcal{F} = \langle \mathcal{F}_0, \text{PROP}, \text{SET}\rangle$ constructed as above.*

It might be unclear why we have only included $\mathcal{F}_0$, PROP, and SET in the structure and ignored functions in general (though not functionals). The reason for this is that the principle of extensionality holds in E_∞ and hence we have a bijection between $\mathcal{F}_0$ and $\mathcal{F}_n$ for $n \leq 1$. In fact, we can show that $\lambda(|a|) = a$.

Lemma 2 *The functional $\lambda : \mathcal{F}_1 \mapsto \mathcal{F}_0$ is bijective.*

Proof

1. λ *is* primitive, *that is, if* $\lambda f = \lambda g$*, then* $f = g$.

2. λ is surjective, *because if* a *is in* $\mathcal{F}_0$ *then* $|a|$, *i.e.* $\langle \mathrm{app}(a,x)|x\rangle$, *is in* $\mathcal{F}_1$ *and* $\lambda(|a|) = a$. □

Hence we have sufficient structure within $\mathcal{F}_0$ to do everything we want without having to consider $\mathcal{F}_1$.

So far, we have not said anything about the interpretation of the predication operator $^{\cup}$. It will be recalled that, by virtue of ($^{\cup}$ *I*), whenever α is an expression of type $\langle e,p\rangle$, $^{\cup}\alpha$ is of type $(e \rightarrow p)$. If α denotes the nominalization λf of a propositional function f, then we want $^{\cup}\alpha$ to denote $\langle \mathrm{app}(\lambda f, x)|x\rangle$. However, the functional corresponding to $^{\cup}$ must carry *any* object in $\mathcal{F}_0$ into an appropriate value in $\mathcal{F}_0$. What happens if α denotes an object a not in SET? In this case, $\langle \mathrm{app}(a,x)|x\rangle$ will not be a propositional function; that is, for any argument b, $\mathrm{app}(a,b)$ will just denote an arbitrary object in $\mathcal{F}_0$. We take this to be an acceptable alternative to the approach used in [Par84] where $^{\cup}$ is interpreted as a partial function, defined only on objects in SET. Hence, we always let $^{\cup}\alpha$ denote $\langle \mathrm{app}([\![\alpha]\!], x)|x\rangle$, where $[\![\alpha]\!]$ is the interpretation of α.

We shall now show how to construct domains inside $\mathcal{F}_0$ such that the types described earlier can be mapped into them.

3.2 Domains

We distinguish between two kinds of domains, *Dom*$_1$ and *Dom*$_2$. We use X_1, Y_1 to range over *Dom*$_1$, X_2, Y_2 to range over *Dom*$_2$, and X, Y to range over both domains. We also assume that $*$ is a distinguished element in $\mathcal{F}_0$ which will be used to give functions a undefined value.

$$\begin{array}{lll} Dom_1 & ::= & \mathcal{F}_0 \mid X_1 \Rightarrow Y_1 \\ Dom_2 & ::= & \text{PROP} \mid \text{TRUTH} \mid X \hookrightarrow X_2 \end{array}$$

Definition 6 ($\Rightarrow$ Function Space)
$X \Rightarrow Y = \{x \in \mathcal{F}_0 :$ *for all* $x' \in X[\mathrm{app}(x,x') \in Y]\}$.

Definition 7 ($\hookrightarrow$ Function Space)
$X \hookrightarrow Y = \{x \in X :$ *for all* $x'[$ *if* $x' \in X$ *then* $\mathrm{app}(x,x') \in Y]$, *else* $app(x,x') = *\}$.

We shall write 'f is *true*' instead of 'f is in TRUTH'. Similarly we use 'f is *false*' instead of 'f is in PROP− TRUTH'. We also assume the presence of two special elements of the Frege structure, 1 in TRUTH and 0 in PROP−TRUTH.

Definition 8 (Internal Definability) *We say that a collection C is internally definable in a Frege structure if the following holds: there is some f in* PF *such that for all x in $\mathcal{F}_0$, $f(x)$ is true iff x is in C. In this case, f is the* characteristic function *of C.*

The domain Dom_1 can be understood as the collection of objects which provide interpretations for types not involving p and t. All the domains of Dom_1 are internally definable. This can be proved by induction as follows:

Lemma 3 *1. $\mathcal{F}_0$ is internally definable, by taking the function $f : \mathcal{F}_0 \mapsto$ PROP, where $f(x) = 1$, forall $x \in \mathcal{F}_0$*

2. Assume X_1, Y_1 are internally definable by the propositional functions f and g respectively. Then we want to show that the collection $X_1 \Rightarrow Y_1$ is also internally definable.

$f{:}\mathcal{F}_0 \mapsto$ PROP where

$$f(x) \text{ is } \begin{cases} \textit{true} & \textit{for all } x \in X_1 \\ \textit{false} & \textit{otherwise} \end{cases}$$

$g{:}\mathcal{F}_0 \mapsto$ PROP where

$$g(y) \text{ is } \begin{cases} \textit{true} & \textit{for all } y \in Y_1 \\ \textit{false} & \textit{otherwise} \end{cases}$$

Let $h : \mathcal{F}_0 \mapsto$ PROP, where

$$h(z) = \forall x[f(x) \supset g(\text{app}(z,x))]$$

Now,

(a) h is a propositional function because f and g are, and

(b) we have to prove that $z \in X_1 \Rightarrow Y_1$ iff $h(z)$ is true.

i. Assume $z \in X_1 \Rightarrow Y_1$. Let $x \in X_1$; then $f(x) \in$ PROP because $f \in$ PF and $\text{app}(z,x) \in Y_1$ because $z \in X_1 \Rightarrow Y_1$. $g(\text{app}(z,x))$ is true because g internally defines Y_1. Hence $f(x) \in$ PROP and $f(x) \in$ TRUTH implies $g(\text{app}(z,x)) \in$ TRUTH. Hence $f(x) \supset g(\text{app}(z,x))$ is true. But this holds for every x, hence $\forall x[f(x) \supset g(\text{app}(z,x))]$ is true. Hence $h(z)$ is true.

ii. Assume $h(z)$ is true. $z \in \mathcal{F}_0$, of course. Let $x \in X_1$, then $f(x) \supset g(\text{app}(z,x))$ is true. But $f(x)$ is true because $x \in X_1$. So $g(\text{app}(z,x))$ is true, and $\text{app}(z,x) \in Y_1$, since g internally defines Y. Hence $z \in X_1 \Rightarrow Y_1$.

Hence $z \in X_1 \Rightarrow Y_1 \iff h(z)$ is true.

□

Now that we know Dom_1 is the domain of internally definable collections, we can write $X \Rightarrow Y$ using f_X and f_Y, the characteristic functions of X and Y:

$$X \Rightarrow Y = \{x \in \mathcal{F}_0 : \text{ for all } x'[f_X(x') \supset f_Y(\text{app}(x,x'))]\}$$

Dom_2, on the other hand, involves domains which are not internally definable. For example, the two basic domains PROP and TRUTH cannot be internally defined. In fact, according to Tarski's theorem on the undefinability of truth, we cannot have a propositional function in the object language which internally defines truth; this implies that we cannot have a propositional function which internally defines propositions; see [Acz80] for discussion.

It might be asked whether the existence of judgements like $\Gamma \vdash_\Sigma \alpha{:}t$ means that we have in effect committed ourselves to the internal definability of truth. The first point to note is that typing statements are not propositions *in* $\mathcal{L}_{\preceq}$, but judgements *about* the language. Second, we have no way of telling for an arbitrary expression α whether the judgement $\Gamma \vdash_\Sigma \alpha : t$ holds. In particular, since contexts Γ are finite, they will not necessarily determine the type of an arbitrary variable.

Recall that app is the functional in the Frege structure which corresponds to *app* in the language of $\mathcal{L}_{\preceq}$. We saw that in a standard Frege structure, $\mathcal{F}_1 = \{f : \mathcal{F}_0 \mapsto \mathcal{F}_0\}$ is the collection of all functions from $\mathcal{F}_0$ to $\mathcal{F}_0$, and contains a subcollection PF of unary propositional functions. We also saw earlier that λ is a bijective map from $\mathcal{F}_1$ to SET. What we now have to check is that, as a special case of Definition 6, there is an appropriate domain $\mathcal{F}_0 \hookrightarrow \mathrm{PROP}$ inside $\mathcal{F}_0$ which will contain the nominals of propositional functions. In fact $(\mathcal{F}_0 \hookrightarrow \mathrm{PROP}) = \mathrm{SET}$ (easy to prove).

Our next lemma illustrates the fact that the domains constructed above do indeed model the types in our language.

Lemma 4 *If X_1, Y_1 are any domains in Dom_1, then $(X_1 \Rightarrow Y_1) \subseteq \mathcal{F}_0$.*
The proof is trivial.

□

In other words, every function in Dom_1 is an object. This enables us to interpret self-application and nominalization.

Lemma 5 *If X is any domain and Y_2 is in Dom_2 then $(X \hookrightarrow Y_2) \subseteq X$.*
The proof is trivial.

□

Lemma 6 *If X_1, Y_1, Y_1' are in Dom_1, then $Y_1 \subseteq Y_1'$ implies $(X_1 \Rightarrow Y_1) \subseteq (X_1 \Rightarrow Y_1')$.*

Proof *If $x \in X_1 \Rightarrow Y_1$, then $\forall x' \in X_1, \mathrm{app}(x,x') \in Y_1$, by Definition 6. Since $Y_1 \subseteq Y_1'$, it follows that for all $x' \in X_1, \mathrm{app}(x,x') \in Y_1'$ and so $x \in X_1 \Rightarrow Y_1'$.*

□

Lemma 7 *If X, Y_2, Y_2' are domains such that Y_2, Y_2' are in Dom_2, then $Y_2 \subseteq Y_2'$ implies $(X \hookrightarrow Y_2) \subseteq (X \hookrightarrow Y_2')$.*

Proof *Same as above.*

□

Lemma 8 *If X_1, X_1' and Y_1 are in Dom_1, then $X_1 \subseteq X_1'$ implies $(X_1' \Rightarrow Y_1) \subseteq (X_1 \Rightarrow Y_1)$.*

Proof *If $x \in (X_1' \Rightarrow Y_1)$, then by Definition 6, for all $x' \in X_1'$, $\mathrm{app}(x,x') \in Y_1$. Since $X_1 \subseteq X_1'$, then for all $x' \in X_1, \mathrm{app}(x,x') \in Y_1$. Therefore $x \in X_1 \Rightarrow Y_1$.*

□

A *model* for $\mathcal{L}_{\preceq}$ is a 6-tuple $\mathcal{M} = \langle \mathcal{F}, \Rightarrow, \hookrightarrow, \mathcal{I}, D, g \rangle$, where

1. $\mathcal{F}$ is a Frege Structure ,
2. $\Rightarrow$ and $\hookrightarrow$ are defined as above,
3. $\mathcal{I}$ is an interpretation function which takes any constant of kind η to an object in D_η, and takes $\perp$ to the element 0 in PROP,
4. D is a function which maps types into domains of $\mathcal{M}$ as follows:
   (a) $D_e = \mathcal{F}_0$
   (b) $D_p =$ PROP
   (c) $D_t =$ TRUTH
   (d) $D_{(\sigma,\tau)} = \begin{cases} D_\sigma \hookrightarrow D_\tau, \text{ if } D_\tau \in Dom_2 \\ D_\sigma \Rightarrow D_\tau \text{ if } D_\sigma \in Dom_1 \text{ and } D_\tau \in Dom_1 \end{cases}$
   (e) $D_{(\sigma \rightarrow \tau)} = \{f : f \text{ is an } \mathcal{F}\text{-functional such that for all } x \in D_\sigma, f(x) \in D_\tau\}$,
5. g is an assignment function which takes any variable of type σ to an object in D_σ.

Note that $Dom_1 \cap Dom_2$ is empty and that Dom_1 will interpret np-types which are not fp-types, among others. Dom_2 will interpret the fp-types, among others.

Since we do not allow variables to range over $\mathcal{F}$-functionals, the interpretation function $\mathcal{I}$ is sufficient to determine the denotation of functors.

We now define a valuation function $[\![\ ]\!]$ which given an expression α and an assignment g yields a value in $Dom_1 \cup Dom_2$.

1. $[\![c]\!]_{\mathcal{M},g} = \mathcal{I}(c)$
2. $[\![x]\!]_{\mathcal{M},g} = g(x)$
3. $[\![app(\alpha,\beta)]\!]_{\mathcal{M},g} = \text{app}([\![\alpha]\!]_{\mathcal{M},g}, [\![\beta]\!]_{\mathcal{M},g})$
4. $[\![{}^{\cup}(\alpha)]\!]_{\mathcal{M},g} = \mid [\![\alpha]\!]_{\mathcal{M},g} \mid$
5. $[\![\lambda x{:}\sigma.\varphi]\!]_{\mathcal{M},g} = \lambda f$, where $f \in \mathcal{F}_1$ and $f(a) = [\![\varphi]\!]_{\mathcal{M},g[a/x]}$ for all $a \in D_\sigma$
6. $[\![\neg\varphi]\!]_{\mathcal{M},g} = \neg[\![\varphi]\!]_{\mathcal{M},g}$
7. $[\![\varphi \wedge \psi]\!]_{\mathcal{M},g} = [\![\varphi]\!]_{\mathcal{M},g} \wedge [\![\psi]\!]_{\mathcal{M},g}$
8. $[\![\varphi \vee \psi]\!]_{\mathcal{M},g} = [\![\varphi]\!]_{\mathcal{M},g} \vee [\![\psi]\!]_{\mathcal{M},g}$
9. $[\![\varphi \supset \psi]\!]_{\mathcal{M},g} = [\![\varphi]\!]_{\mathcal{M},g} \supset [\![\psi]\!]_{\mathcal{M},g}$

10. $[\forall x{:}\sigma.\varphi]_{\mathcal{M},g} = \forall f$, where $f \in \mathcal{F}_1$ and $f(a) = [\varphi]_{\mathcal{M},g[a/x]}$ if $a \in D_\sigma$, and $f(a) = 0$ otherwise

11. $[\exists x{:}\sigma.\varphi]_{\mathcal{M},g} = \exists f$, where $f \in \mathcal{F}_1$ and $f(a) = [\varphi]_{\mathcal{M},g[a/x]}$ if $a \in D_\sigma$, and $f(a) = 0$ otherwise

It will be observed that these valuation clauses depend on the existence of the appropriate functionals (e.g., $app, \neg, \wedge, \vee, \forall, \exists$) in the Frege structure. It would be straightforward to convert the clauses for propositions into truth-theoretic definitions, along the following lines:

8′ $[\varphi \supset \psi]_{\mathcal{M},g} \in \text{TRUTH} \iff [\psi]_{\mathcal{M},g} \in \text{TRUTH}$ whenever $[\varphi]_{\mathcal{M},g} \in \text{TRUTH}$

9′ $[\forall x{:}\sigma.\varphi]_{\mathcal{M},g} \in \text{TRUTH} \iff [\varphi]_{\mathcal{M},g[a/x]} \in \text{TRUTH}$ for all $a \in D_\sigma$

Lemma 9 *$D_{\langle\sigma,\tau\rangle} = (D_\sigma \Rightarrow D_\tau) \subseteq D_e$ if $D_\sigma, D_\tau \in Dom_1$*

Proof *Obvious by Lemma 4.*

□

Lemma 10 *$D_{\langle\sigma,\tau\rangle} \subseteq D_\sigma$ if $D_\tau \in Dom_2$*

Proof *If $D_\tau \in Dom_2$, then $D_{\langle\sigma,\tau\rangle} = (D_\sigma \hookrightarrow D_\tau) \subseteq D_\sigma$ by Lemma 5 .*

□

Lemma 11 *For any type σ, D_σ is either in Dom_1 or in Dom_2.*

Proof *by induction on the construction of types.*

- *If $\sigma = p$, t, or e, then obvious.*
- *If $\sigma = \langle\sigma_1, \sigma_2\rangle$, where the property holds for σ_1 and σ_2, then also obvious.*

□

Lemma 12 *$D_\sigma \subseteq D_e$ for any type σ.*[8]

Proof *By induction on σ.*

- *σ is base type (i.e., e, t or p). Obvious.*
- *Assume $\sigma = \langle\sigma_1, \sigma_2\rangle$, where $D_{\sigma_1} \subseteq D_e$ and $D_{\sigma_2} \subseteq D_e$.*

  Case 1 *$D_\sigma = D_{\sigma_1} \hookrightarrow D_{\sigma_2}$. Then $D_\sigma \subseteq D_{\sigma_1}$ by Lemma 10. Since $D_{\sigma_1} \subseteq D_e$, by induction hypothesis, we have $D_\sigma \subseteq D_e$.*

  Case 2 *$D_\sigma = D_{\sigma_1} \Rightarrow D_{\sigma_2}$. Then $D_\sigma \subseteq D_e$ by Lemma 9.*

□

[8]Of course, the domain for $(\tau_1 \rightarrow \tau_2)$ is not contained in D_e; but this follows from the fact that $(\tau_1 \rightarrow \tau_2)$ is not a type but a metatype.

Lemma 13 *If $\sigma \preceq \tau$, then $D_\sigma \subseteq D_\tau$.*

Proof

($e \preceq$) *Case $\sigma \preceq e$.*
$D_\sigma \subseteq D_e$ always holds, by Lemma 12.

($p \preceq$) *Case $t \preceq p$.*
$D_t \subsetneq D_p$, since TRUTH $\subsetneq$ PROP.

(*Ran* $\preceq$) *Case $\tau_1 \preceq \tau_2$.*

1. *If $D_{\tau_1}, D_{\tau_2} \in Dom_2$ then use Lemma 7.*

2. *If $D_{\tau_1}, D_{\tau_2} \in Dom_1$ then use Lemma 6.*

3. *It cannot be the case that $D_{\tau_1} \in Dom_1$ and $D_{\tau_2} \in Dom_2$.*

4. *If $D_{\tau_1} \in Dom_2$ and $D_{\tau_2} \in Dom_1$ then $D_{\langle\sigma,\tau_1\rangle} = (D_\sigma \hookrightarrow D_{\tau_1})$ and $D_{\langle\sigma,\tau_2\rangle} = (D_\sigma \Rightarrow D_{\tau_2})$. It is easy to check $(D_\sigma \hookrightarrow D_{\tau_1}) \subset (D_\sigma \Rightarrow D_{\tau_2})$.*

(*Id* $\preceq$, *Trans* $\preceq$, *Anti* $\preceq$) *Obvious.*

(*Prop* $\preceq$) *$\tau \preceq p$ implies $D_{\langle\sigma,\tau\rangle} \subsetneq D_\sigma$. This holds since $D_{\langle\sigma,\tau\rangle} = (D_\sigma \hookrightarrow D_\tau) \subsetneq D_\sigma$, by Lemma 5.*

(*Fix* $\preceq$) *$\tau \preceq p$ implies $D_{\langle\langle\sigma,\tau\rangle,\tau\rangle} = D_{\langle\sigma,\tau\rangle}$. We need to show that $(D_\sigma \hookrightarrow D_\tau) \hookrightarrow D_\tau = D_\sigma \hookrightarrow D_\tau$. From the proof of (Prop $\preceq$) above, it follows that $D_{\langle\langle\sigma,\tau\rangle,\tau\rangle} \subsetneq D_{\langle\sigma,\tau\rangle}$. The reverse inclusion is established as follows. Let $x \in (D_\sigma \hookrightarrow D_\tau)$. This implies that for all $x' \in (D_\sigma \hookrightarrow D_\tau) \subsetneq D_\sigma$, $\mathbf{app}(x,x') \in D_\tau$. Hence $x \in (D_\sigma \hookrightarrow D_\tau) \hookrightarrow D_\tau$.*

(*Dom* $\preceq$) *$\sigma_1 \preceq \sigma_2$ implies $D_{\langle\sigma_2,e\rangle} \subsetneq D_{\langle\sigma_1,e\rangle}$. By the induction hypothesis, $\sigma_1 \preceq \sigma_2$ implies $D_{\sigma_1} \subsetneq D_{\sigma_2}$. D_e is in Dom_1 and as we restrict types so that a domain type is never strictly less than the range type, then D_{σ_1} and D_{σ_2} must be in Dom_1. Hence by definition, $D_{\langle\sigma_2,e\rangle} = D_{\sigma_2} \Rightarrow D_e$. Let $x \in D_{\langle\sigma_2,e\rangle}$. Hence $x \in (D_{\sigma_2} \Rightarrow D_e)$. So $x \in \mathcal{F}_0$ and for all $x' \in D_{\sigma_2}$, $\mathbf{app}(x,x') \in D_e$ and $D_{\langle\sigma_1,e\rangle} = D_{\sigma_1} \Rightarrow D_e$. Since $D_{\sigma_1} \subsetneq D_{\sigma_2}$, it follows that $x \in \mathcal{F}_0$ and for all $x' \in D_{\sigma_1}$, $\mathbf{app}(x,x') \in D_e$. Hence $x \in D_{\sigma_1} \Rightarrow D_e$.*

□

Theorem 1 *If $\Gamma \vdash \alpha{:}\sigma$, where $D_\sigma \in Dom_1$ then $[\![\alpha]\!]_{\mathcal{M},g} \in D_\sigma$.*

Proof

- *If α is a constant c or variable x, this is obvious from the definition of $\mathcal{I}$ and g.*

- *Let us assume the property holds for expressions α, β, and show that it holds for $app(\alpha, \beta)$. $\Gamma \vdash app(\alpha, \beta){:}\tau$ iff $\Gamma \vdash \alpha{:}\langle \sigma, \tau \rangle$ and $\Gamma \vdash \beta{:}\sigma$. So $[\![\alpha]\!]_{\mathcal{M},g} \in D_{\langle \sigma, \tau \rangle}$, $[\![\beta]\!]_{\mathcal{M},g} \in D_\sigma$, and $[\![app(\alpha,\beta)]\!]_{\mathcal{M},g} = \mathbf{app}([\![\alpha]\!]_{\mathcal{M},g}, [\![\beta]\!]_{\mathcal{M},g})$. The latter belongs to D_τ, as $D_{\langle \sigma, \tau \rangle} = D_\sigma \Rightarrow D_\tau$.*

- *Let us prove $[\![\lambda x{:}\sigma.\alpha]\!]_{\mathcal{M},g} \in D_{\langle \sigma, \tau \rangle}$, where $\Gamma \vdash_\Sigma \tau$ np-type, $\Gamma \not\vdash_\Sigma \sigma \prec \tau$, and by induction hypothesis $[\![\alpha]\!]_{\mathcal{M},g[a/x]} \in D_\tau$ for all $x{:}\sigma$.*

  Since $D_{\langle \sigma, \tau \rangle}$ is in Dom_1 then $D_{\langle \sigma, \tau \rangle} = D_\sigma \Rightarrow D_\tau$. Hence $[\![\lambda x{:}\sigma.\alpha]\!]_{\mathcal{M},g} \in \mathcal{F}_0$, and for all $a \in D_\sigma$, $\mathbf{app}([\![\lambda x{:}\sigma.\alpha]\!]_{\mathcal{M},g}, a) = \mathbf{app}(\lambda f, a) = f(a)$, where $f(a) = [\![\alpha]\!]_{\mathcal{M},g[a/x]} \in D_\tau$. Hence $[\![\lambda x{:}\sigma.\alpha]\!]_{\mathcal{M},g} \in D_\sigma \Rightarrow D_\tau$.

$\square$

4 A Fragment of English

The English fragment that we consider is intentionally simple,[9] and will focus attention on issues of polymorphism and self-application. One possible way of setting up the grammar would be to follow Montague in using the standard fractional notation of categorial grammar, together with a homomorphism which maps the categories into semantic types. However, for our purposes, it would be preferable to build the syntactic categories directly on top of the types. Consequently, the categories of the grammar will consist of *decorated* types and metatypes of $\mathcal{L}_\prec$; that is, types and metatypes annotated with phrase structure labels. The latter will provide us with the power to draw somewhat finer distinctions of the kind required for English syntax. For example, intransitive verbs, adjectives and common nouns will all belong to the type $\langle e, p \rangle$; however, this type will be annotated as $\langle e, p \rangle^V$, $\langle e, p \rangle^A$, or $\langle e, p \rangle^N$, respectively. The list of admissible labels is the following: S (sentences), V (verbs), N (nouns), CN (common nouns), A (adjectives), P (prepositions), Adv (adverbials). In some cases, we modestly extend these labels with features. For example, we use 'P[to]' as the label for prepositional phrases whose head is the word **to**. We use X as an underspecified category label; this will be useful when we want to give a maximally general decoration to a type.

Whenever η is a kind, and C is a category label, then σ^C is a decorated kind. The rules given previously for constructing a complex kind can be generalized in the obvious way to decorated kind. We use the symbols 's, t, r' as metavariables ranging over decorated kinds. It is obvious that we can simply strip the labels off a decorated kind s to recover our original kind. We use '$^\circ s$' to denote the stripped-down version of s, where $^\circ\langle s, t \rangle^C = \langle ^\circ s, ^\circ t \rangle$, and $^\circ(s \to r)^C = (^\circ s \to ^\circ t)$.

An English grammar object will be a triple

$$(w, s, \alpha)$$

[9] In particular, we do not treat quantified noun phrases. It would be straightforward to implement [ChT88]'s treatment of type-shifting for quantifier arguments. It is unclear to us, however, what the appropriate analysis of scope would be in the current setting; [Moo90]'s approach seems promising.

where w is a phonological (in practice, orthographic) form, s is a decorated type, α is an expression of $\mathcal{L}_{\preceq}$, and moreover $\Gamma \vdash_{\Sigma} \alpha{:}^{\circ}s$, with Σ as specified before..

As a typographical convenience, we shall also employ the following vertical format for these triples:

w
s
α

For example, the representations of the words **John** and **kiss** are:

(19)	John	kiss
	e^N	$\langle e^N, \langle e,p\rangle^V\rangle\ ^V$
	john′	kiss′

Thus, **kiss** has the type of a verbal expression which will combine with something of type e^N to make something of type $\langle e,p\rangle^V$. The decorated type therefore combines standard categorial information, which would usually be notated VP/NP (i.e., a functor which combines with an NP to make VP, together with the semantic type that such a category would be mapped into.

The rules of type inference are like those for $\mathcal{L}_{\preceq}$, except that we add the following provisos, where s, t are decorated types, X, Y are category decorations, and $\preceq^*$ is a partial order over category labels:

Definition 9 (Containment of decorated types)

1. $s^X \preceq t^Y$ *iff* $^{\circ}s \preceq {^{\circ}t}$ *and* $X \preceq^* Y$.
2. $\preceq^*$ *is reflexive, transitive and antisymmetric.*
3. $X \preceq^* X$.
4. $P \preceq^* N$.

It will be noticed that the type assigned to **kiss**, namely,

$$(20)\quad \langle e^N, \langle e,p\rangle^V\rangle\ ^V,$$

appears redundant in the sense that not only is the type as whole specified to be V, but the result type, $\langle e,p\rangle^V$, is also so specified. Yet inasmuch as **kiss** is the head of verb phrase, it should be predictable that the result type has the same category decoration as the whole complex type. In response to this observation, we adopt the convention that if the result type lacks a decoration, then it can be inferred from the decoration of the enclosing decorated type; in other words, a type like (21) is shorthand for (20).

(21) $\langle e^N, \langle e, p\rangle\rangle^V$

We could make this more explicit by means of a modified inference rule of the following sort (where σ is restricted to *un*decorated types):

(22) $$\frac{\alpha{:}\langle s, \sigma\rangle^C \qquad \beta{:}s}{app(\alpha, \beta){:}\sigma^C}$$

In fact, we will not adopt (22) exactly as it stands, but will propose a further slight modification in the next section.

Our grammar for English is non-directional, in the sense that we do not encode whether a functor seeks its argument to the left or to the right. Modifying the notation to allow this would be trivial, but would add an extra degree of complexity which would detract from the main thrust of the exposition. For convenience, we shall simply write the premisses of a type inference rule in the correct left-to-right order, stipulate that the string in the conclusion is the right of concatenating the strings of the premisses. This is shown in the following schema for type inference in the fragment, (where '$\frown$' indicates concatenation):

Definition 10 (Inference Schema for English)

$$\frac{(w_1, s_1, \alpha_1) \qquad (w_2, s_2, \alpha_2)}{(w_1 {}^\frown w_2, s_3, \alpha_3)}$$

is valid just in case the corresponding inference

$$\frac{\Gamma \vdash_\Sigma \alpha_1{:}{}^\circ s_1 \qquad \Gamma \vdash_\Sigma \alpha_2{:}{}^\circ s_2}{\Gamma \vdash_\Sigma \alpha_3{:}{}^\circ s_3}$$

is derivable for the undecorated types ${}^\circ s_1$, ${}^\circ s_2$, *and* ${}^\circ s_3$.

4.1 Argument Asymmetries

A number of authors have claimed that there are semantic asymmetries between subject and object arguments. Marantz [Mar84], for example, has noted that the choice of direct object can significantly affect the semantic role played by the subject, whereas choice of subject has no such effect on the role of the object. This is illustrated in the contrast between (23) and (24):

(23) a throw a baseball
b throw one's support behind a candidate
c throw a party

(24) a The player threw NP
b The politician threw NP
c The social directory threw NP

A second asymmetry is that it is a matter of lexical idiosyncrasy whether a verb takes a direct object, indirect object, and so on, whereas by and large, it is entirely predictable that verbs (at least in English) take a syntactic subject.

In the current context, a third asymmetry can be noted. Whether a verb is tensed affects its ability to combine with a subject, but not its ability to combine with object arguments and complements:

(25) a to kiss Mary/kissed Mary
b *John to kiss Mary/John kissed Mary

The semantic framework we have developed gives us an account, though not a complete one, of this third fact. For there are two distinct ways in which a syntactic functor can combine semantically with an argument: either via the *app* relation, or by normal functional application. Moreover, *app* is invoked for functors which we earlier called 'nominal predicatives', i.e., expressions which denote objects in the Frege structure domain $\mathcal{F}_0$; by contrast, expressions which denote propositional functions live outside $\mathcal{F}_0$ and therefore cannot act as nominal arguments. What is lacking still is a mechanism which precludes the use of *app* to effect the semantic counterpart of ***John to kiss Mary.**

Let us start by looking at transitive and intransitive verbs. The base, or nontensed, form of an intransitive verb like **run** is translated as a constant run′ of type $\langle e,p\rangle^V$; as we observed in §1, such constants denote (a special sort of) nominal objects, not propositional functions. Similarly, the base form of a transitive verb such as **kiss** is translated as a constant kiss′ of type $\langle e^N,\langle e,p\rangle^V\rangle^V$, which also denotes a sort of nominal object. As we observed earlier, this decorated type corresponds to a functor category of the form V/N in categorial grammar; i.e., something which combines with an N to make a V. By contrast, it is hard to see what the categorial equivalent of $\langle e,p\rangle^V$ might be, since we have omitted any mention of the argument's syntactic label; that is, there is no syntactic specification for the e component of the type. Let us adopt the stipulation that an expression with decorated type $\langle s,t\rangle^A$ is only a *syntactic functor* if s is itself a decorated type of the form r^B. It then follows that no expression with type $\langle e,p\rangle^V$ is a syntactic functor.

This leads us to the following formulation of a modified axiom for *app* within the English fragment:

Definition 11

$$\frac{(w_1,\langle s,\sigma\rangle^C,\alpha_1) \qquad (w_2,s,\alpha_2)}{(w_1{}^\frown w_2,\sigma^C,app(\alpha_1,\alpha_2))}$$

is valid just in case s is a decorated type, and the corresponding inference

$$\frac{\Gamma\vdash_\Sigma \alpha_1:\langle{}^\circ s,\sigma\rangle \qquad \Gamma\vdash_\Sigma \alpha_2:{}^\circ s}{\Gamma\vdash_\Sigma app(\alpha_1,\alpha_2):\sigma}$$

is derivable for the undecorated type $^\circ s$.

This licenses derivations like the following:

Example 1

$$\frac{\begin{array}{l}\textbf{kiss}\\ \langle e^N,\langle e,p\rangle\rangle^{V[\mathrm{BASE}]}\\ \textbf{kiss}'\end{array} \qquad \begin{array}{l}\textbf{Mary}\\ e^N\\ \textbf{mary}'\end{array}}{\begin{array}{l}\textbf{kiss Mary}\\ p^V\\ app(\textbf{kiss}',\textbf{mary}')\end{array}}$$

By contrast, the following inference is prohibited, since walk has not been categorized as a syntactic functor:

(26)

$$\frac{\begin{array}{l}\textbf{John}\\ e^N\\ \textbf{john}'\end{array} \qquad \begin{array}{l}\textbf{walk}\\ \langle e,p\rangle^V\\ \textbf{walk}'\end{array}}{\begin{array}{l}\textbf{John walk}\\ p^V\\ app(\textbf{walk}',\textbf{john}')\end{array}}$$

Before pursuing this point further, let us introduces some new notation to indicate the iterated application of a functor α to a series of arguments:

Definition 12 (Multiple Application)

$$[\alpha,x_1,\ldots,x_n] =_{df} app(\ldots(app(\alpha,x_1),\ldots),x_n)$$

Example 2 *Assuming* **give′** *to be of type* $\langle e^{P[\mathrm{to}]},\langle e^N,\langle e,p\rangle\rangle\rangle^V$, *we have the following semantic translation for* **give the cat to Mary:**

$$[\textbf{give}',\textbf{mary}',(\textbf{the cat})']:\langle e,p\rangle = app(app(\textbf{give}',\textbf{mary}'),(\textbf{the cat})'):\langle e,p\rangle$$

The last step in the derivation of this example is:

$$\frac{\begin{array}{l}\textbf{give to Mary}\\ \langle e^N,\langle e,p\rangle\rangle^V\\ [\textbf{give}',\textbf{mary}']\end{array} \qquad \begin{array}{l}\textbf{the cat}\\ e^N\\ \textbf{the cat}'\end{array}}{\begin{array}{l}\textbf{give the cat to Mary}\\ \langle e,p\rangle^V\\ [\textbf{give}',\textbf{mary}',(\textbf{the cat})']\end{array}}$$

Since the decorated type for **give**, namely $\langle e^{P[\mathrm{to}]}, \langle e^N, \langle e, p\rangle\rangle\rangle^V$, does give a syntactic specification of its argument expressions, we treat it as a syntactic functor, and use *app* at the semantic level to combine it with its object arguments **to Mary** and **the cat**. However, the resulting IV phrase has type $\langle e, p\rangle^V$, and this as we saw above cannot combine directly with a subject NP according to our rules.

What we must do now is make explicit the way in which tense is introduced. From a semantic point of view, it would be easiest to take a phrase like (untensed) **give the cat to Mary** and map it into the (tensed) phrase **gives the cat to Mary**. But from a morphological point of view, it would be more straightforward to have a lexical rule, mapping base form **give** into tensed **gives**. We will side with morphological economy, and effect the change at the lexical level; for simplicity, we will restrict attention to the mapping from base form verbs of type $s^{V[\mathrm{BASE}]}$ into third person singular present tense verbs of type $s^{V[3\mathrm{S}]}$. Thus, we have the following inference schema (where $f_{3\mathrm{S}}$ is a function which supplies the third person singular present inflection):

Definition 13 (Inflection Schema)

$$\frac{\begin{array}{l}\alpha\\ \langle s_1, \langle \ldots \langle s_n, \langle e, p\rangle\rangle \ldots\rangle\rangle^{V[\mathrm{BASE}]}\\ \alpha'\end{array}}{\begin{array}{l}f_{3\mathrm{S}}(\alpha)\\ \langle s_1, \langle \ldots \langle s_n, \langle e^X \rightarrow p\rangle\rangle \ldots\rangle\rangle^{V[3\mathrm{S}]}\\ \lambda x_1{:}^{\circ}s_1 \ldots \lambda x_n{:}^{\circ}s_n.^{\cup}[\alpha', x_1, \ldots, x_n]\end{array}},$$

Example 3

$$\frac{\begin{array}{l}\textbf{kiss}\\ \langle e^N, \langle e, p\rangle\rangle^{V[\mathrm{BASE}]}\\ \textbf{kiss}'\end{array}}{\begin{array}{l}\textbf{kisses}\\ \langle e^N, \langle e^X \rightarrow p\rangle\rangle^{V[3\mathrm{S}]}\\ \lambda x{:}e.^{\cup}[\textbf{kiss}', x]\end{array}}$$

$$\frac{\begin{array}{l}\textbf{be}\\ \langle \langle e, p\rangle^X, \langle e, p\rangle\rangle^{V[\mathrm{BASE}]}\\ \textbf{be}'\end{array}}{\begin{array}{l}\textbf{is}\\ \langle \langle e, p\rangle^X, \langle e^X \rightarrow p\rangle\rangle^{V[3\mathrm{S}]}\\ \lambda x{:}\langle e, p\rangle.^{\cup}[\textbf{be}', x]\end{array}}$$

As a further illustration, we show how a tensed intransitive verb combines with a subject noun phrase:

Example 4

$$\begin{array}{ll} \textbf{John} & \textbf{walks} \\ e^N & (e^N \to p^S)^V \\ \textbf{john}' & {}^{\cup}\textbf{walk}' \\ \hline \multicolumn{2}{c}{\textbf{John walks}} \\ \multicolumn{2}{c}{p^S} \\ \multicolumn{2}{c}{{}^{\cup}\textbf{walk}'(\textbf{john}')} \end{array}$$

In a similar manner, the tensed verbs illustrated above will combine with their arguments to yield the following translations:

Example 5

John kisses Mary *translates as*

$$(app(\lambda x{:}e.{}^{\cup}[\textbf{kiss}', x], \textbf{mary}'))(\textbf{john}') = {}^{\cup}[\textbf{kiss}', \textbf{mary}'](\textbf{john}')$$

John is happy *translates as*

$$(app(\lambda x{:}\langle e, p\rangle.{}^{\cup}[\textbf{be}', x], \textbf{happy}'))(\textbf{john}') = {}^{\cup}[\textbf{be}', \textbf{happy}'](\textbf{john}')$$

To recapitulate then, the two different kinds of semantic combination, namely *app* and standard function application, are correlated with two kinds of arguments in a proposition: on the one hand, those arguments which in phrase structure grammars are deemed to be subconstituents of the verb phrase, and on the other hand, the subject. This distinction corresponds loosely to the one drawn by Williams between 'internal' and 'external' arguments [Wil81], though it is obviously impossible for us to follow Williams' claim that there are propositions which possess no external arguments.

Table 1 summarizes the assignment of categories to expressions of English in our fragment.

4.2 Nominalization and Polymorphism

As we indicated at the beginning of this paper, we do not employ a rule of nominalization as such. Rather, some expressions—the ones categorised as 'nominalizable' in Table 1—have kinds which are contained in the type e of individuals. The type inference rule (*Contain*) then comes into play to derive the more general type. This is illustrated in the next derivation:

Informal Name	*Type*	*Basic Expressions*
	Nominalizable Expressions	
NP	e^N	**John, Mary**
CN_{count}	$\langle e,p\rangle^{CN}$	**dog, man, woman, park**
CN_{mass}	$\langle e,p\rangle^{N}$	**water, gold, fun**
ADJ	$\langle e,p\rangle^{A}$	**happy, drunk, old**
PP	$\langle\langle e,p\rangle^X,\langle e,p\rangle^X\rangle^P$	$\emptyset$
TV	$\langle e^N,\langle e,p\rangle\rangle^V$	**kiss, seek**
	$\langle e^X,\langle e,p\rangle\rangle^V$	**believe, know**
	$\langle\langle e,p\rangle^{V[to]},\langle e,p\rangle\rangle^V$	**try, want**
	$\langle\langle e,p\rangle^X,\langle e,p\rangle\rangle^V$	**be**
TTV	$\langle e^{P[to]},\langle e^N,\langle e,p\rangle\rangle\rangle^V$	**give, send**
	$\langle\langle e,p\rangle^{V[to]},\langle e^N,\langle e,p\rangle\rangle\rangle^V$	**force, believe**
IV	$\langle e,p\rangle^V$	**run, walk, talk**
S	p^S	$\emptyset$
S′	$p^{S[comp]}$	$\emptyset$
IV_{inf}	$\langle e,p\rangle^{V[to]}$	$\emptyset$
	Non-nominalizable Expressions	
Det	$(\langle e,p\rangle^{CN}\to e^N)^{Det}$	**the, a, some**
VP	$(e^X\to p^S)$	$\emptyset$
AdSent	$(p^S\to p^S)^{Adv}$	**necessarily, possibly**
AdVerb	$(\langle e,p\rangle^V\to\langle e,p\rangle^V)^{Adv}$	**slowly, rudely**
P	$(e^N\to\langle\langle e,p\rangle^X,\langle e,p\rangle^X\rangle)^P$	**in, with**
AdNom	$(\langle e,p\rangle^N\to\langle e,p\rangle^N)^A$	**former**
COMP	$(\langle e,p\rangle^V\to\langle e,p\rangle^{V[to]})$	**to**
	$(p^S\to p^{S[comp]})$	**that**

Table 1: Categories and expressions in the fragment

(27)

$$\dfrac{\dfrac{\begin{array}{c}\textbf{fun}\\ \langle e,p\rangle^N\\ \textbf{fun}'\end{array}}{\begin{array}{c}\textbf{fun}\\ e^N\\ \textbf{fun}'\end{array}}(Contain)\qquad \dfrac{\begin{array}{c}\textbf{is}\\ \langle\langle e,p\rangle^X,(e\to p)\rangle^V\\ \lambda x{:}\langle e,p\rangle.{}^{\cup}[\textbf{be}',x]\end{array}\qquad \begin{array}{c}\textbf{fun}\\ \langle e,p\rangle^N\\ \textbf{fun}'\end{array}}{\begin{array}{c}\textbf{is fun}\\ (e^X\to p^S)^V\\ {}^{\cup}[\textbf{be}',\textbf{fun}']\end{array}}(app)}{\begin{array}{c}\textbf{fun is fun}\\ p^S\\ {}^{\cup}[\textbf{be}',\textbf{fun}'](\textbf{fun}')\end{array}}(Funct)$$

We have chosen to analyse **fun** as a mass noun rather than an adjective, on the grounds that collocations involving noun modifiers, as (28a), seem significantly better than those involving adjectival modifiers, as (28b):

(28) a It wasn't much fun.

b ?It was extremely/very fun.

Nothing crucial hangs on this decision. Nevertheless, it follows on our account that all mass nouns can occur as nominal arguments. They can also occur as predicative complements by virtue of the polymorphic type assigned to **be**.

As another example of polymorphism, consider the behaviour of prepositional phrases. We would like to be able to treat them as IV and CN modifiers, as illustrated below (for brevity, we have omitted the (*Contain*) step which maps $\langle e,p\rangle^V$ to $\langle e,p\rangle^X$):

(29)

$$\dfrac{\begin{array}{l}\textbf{in the park}\\ \langle\langle e,p\rangle^X,\langle e,p\rangle^X\rangle^P\\ [\textbf{in}',(\textbf{the park})']\end{array} \qquad \begin{array}{l}\textbf{walk}\\ \langle e,p\rangle^V\\ \textbf{walk}'\end{array}}{\begin{array}{l}\textbf{walk in the park}\\ \langle e,p\rangle^V\\ [\textbf{in}',(\textbf{the park})',\textbf{walk}']\end{array}}$$

$$\dfrac{\begin{array}{l}\textbf{in the park}\\ \langle\langle e,p\rangle^X,\langle e,p\rangle^X\rangle^P\\ [\textbf{in}',(\textbf{the park})']\end{array} \qquad \begin{array}{l}\textbf{man}\\ \langle e,p\rangle^N\\ \textbf{man}'\end{array}}{\begin{array}{l}\textbf{man in the park}\\ \langle e,p\rangle^V\\ [\textbf{in}',(\textbf{the park})',\textbf{man}']\end{array}}$$

We would also like prepositional phrases to act like nominal arguments. The desired result is achieved as follows (again, we have omitted the (*Contain*) step which maps $\langle e,p\rangle^P$ to $\langle e,p\rangle^N$, using the condition $P\preceq^* N$ given in Definition 9):

(30)

$$\dfrac{\begin{array}{c}\textbf{give}\\ \langle e^{P[to]}, \langle e^{N}, \langle e,p\rangle\rangle\rangle^{V}\\ \textbf{give}'\end{array} \qquad \dfrac{\begin{array}{c}\textbf{to Mary}\\ \langle\langle e,p\rangle^{X},\langle e,p\rangle^{X}\rangle^{P[to]}\\ (\textbf{to Mary})'\end{array}}{\begin{array}{c}\textbf{to Mary}\\ e^{P[to]}\\ (\textbf{to Mary})'\end{array}}(Contain)}{\begin{array}{c}\textbf{give to Mary}\\ \langle e^{N},\langle e,p\rangle\rangle^{V}\\ {}^{\cup}[\textbf{give}',(\textbf{to Mary})']\end{array}}(app)$$

4.3 Comparison with the Chierchia-Turner Fragment

We conclude with some brief remarks relating our approach to the fragment proposed by [ChT88].

First, it will be observed that there is a broad correspondence between our type '$\langle e, p\rangle$' and their sort 'nf', standing for nominalized functions, and to this extent the two fragments are quite similar. However, [ChT88]'s semantic domain D_{nf} is the nominalization of all functions from e to e, rather than those from e to p; i.e., it corresponds to the whole codomain of λ, not just to the collection SET.

Second, for Chierchia and Turner, only expressions of type nf are nominals. Since their nominalization operator is exclusively defined for expressions of type $\langle e, e\rangle$[10], 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 (31a) (from [Par86]) and (31b) show that untensed transitive verbs enter into the same nominal patterns as intransitives:

(31) a To love is to exalt.

b To give is better than to receive.

By contrast, we have $\langle e^{N}, \langle e, p\rangle\rangle^{V[to]} \preceq e^{V[to]}$, and can thus accommodate such data straightforwardly.

[10]This type corresponds to our metatype $(e \rightarrow e)$.

References

[Acz80] Aczel, P. (1980) 'Frege Structures and the Notions of Proposition, Truth and Set.' In Barwise, J., Keisler, H. J. and Kunen, K. (eds.) *The Kleene Symposium*, pp31-59. Amsterdam: North Holland.

[Bach79] Bach, E. (1979) 'Control in Montague Grammar.' *Linguistic Inquiry* 10, 515–531.

[Bach80] Bach, E. (1980) 'In Defence of Passive.' *Linguistics and Philosophy* 3, 297–341.

[Bea82] Bealer, G. (1982) *Quality and Concept*. Oxford: Clarendon Press.

[Bea89] Bealer, G. (1989) 'On the Identification of Properties and Propositional Functions.' *Linguistics and Philosophy* 12, 1–14.

[CW85] Cardelli, L. and P. Wegner (1985) 'On understanding types, data abstraction and polymorphism.' *Computing Surveys* 17, 471–522.

[Chi84] Chierchia, G. (1984) *Topics in the Syntax and Semantics of Infinitives and Gerunds*. Unpublished PhD Thesis, University of Massachusetts.

[Chi85] Chierchia, G. (1985) 'Formal Semantics and The Grammar of Predication.' *Linguistic Inquiry* 16, pp.417–443.

[ChT88] Chierchia, G. and R. Turner (1988) 'Semantics and Property Theory.' *Linguistics and Philosophy* 11, pp.261–302.

[CG89] Curien, P.-L. and G. Ghelli (1989) 'Coherence of Subsumption.' Unpublished ms, Liens (CNRS), Paris.

[Fre77] Frege, G. (1977) *Translations from the Philosophical Writings of Gottlob Frege*. Geach, P. and Black, M. (eds.), 3rd Edition, pp56-78. Oxford: Basil Blackwell.

[HHP87] Harper, R., Honsell, F., and G. Plotkin (1987) 'A Framework for Defining Logics.' *Second Annual Symposium on Logic in Computer Science*, IEEE, pp.194–204.

[Jac90] Jacobson, P. (1990) 'Raising as Function Composition.' *Linguistics and Philosophy* 13, pp.423–475.

[Mar84] Marantz, A.P. (1984) *On the Nature of Grammatical Relations*. MIT Press, Cambridge, Massachusetts.

[M-L79] Matin-Löf, P. (1978) 'Constructive Mathematics and Computer Programming.' In *Logic, Methodology and Philosophy of Science, VI, 1979*, pp.153–175, North-Holland.

[Mil78] Milner, R. (1978) 'A Theory of Type Polymorphism in Programming.' *Journal of Computer and System Sciences* 17, pp.348–375.

[Mit88] Mitchell, J. C. (1988) 'Polymorphic Type Inference and Containment.' *Information and Computation* 76, pp.211–249.

[Mon73] Montague, R. (1973) 'The proper treatment of quantification in ordinary English.' In Hintikka, J., Moravcsik, J. M. E. and Suppes, P. (eds.) *Approaches to Natural Language*. Dordrecht: D. Reidel. Reprinted in R. H. Thomason (ed.) (1974), *Formal Philosophy: Selected Papers of Richard Montague*, pp247-270. Yale University Press: New Haven, Conn.

[Moo90] Moortgat, M. (1990) 'Discontinuous Type Constructors.' Unpublished paper presented to Workshop on Categorial Grammar and Linear Logic, 2nd European Summer School in Logic, Language and Information, Leuven, August 1990.

[Pars79] Parsons, T. (1979) 'The theory of types and ordinary language.' In S. Davies and M. Mithun (eds.) *Linguistics, Philosophy and Montague Grammar*, University of Texas Press, Austin.

[PR83] Partee, B. H. and M. Rooth (1983) 'Generalized conjunction and type ambiguity.' In R. Bäuerle, C. Schwarze, and A. von Stechow (eds.) *Meaning, Use, and Interpretation of Language*, De Gruyter.

[Par84] Partee, B. H. (1984) 'Compositionality.' In F. Landman and F. Veltman (eds.) *Varieties of Formal Semantics: Proceedings of The Fourth Amsterdam Colloquium, Sept 1982* Foris Press, Dordrecht.

[Par86] Partee, B. H. (1986) 'Ambiguous pseudo-clefts with ambiguous *be*.' In S. Berman, J. Choe and J. McDonough (eds.) *Proceedings of the Sixteenth Annual Meeting of the North Eastern Linguistic Society*, University of Massachusetts, Amherst.

[PS87] Pollard, C. and I. A. Sag (1987) *Information-Based Syntax and Semantics, Vol. 1.* CSLI Lecture Notes, No. 13.

[RP82] Rooth, M. and B. H. Partee Mats Rooth 'Conjunction, type ambiguity, and wide scope 'or'.' In M. Barlow, D. Flickinger and M. Westcoat (eds.) *Proceedings of the Second West Coast Conference on Formal Linguistics*, pp353-362.

[Sco76] Scott, D. (1976) 'Data Types as Lattices.' *SIAM* Journal of Computing, 5, 522–587.

[Tho76] Thomason, R. H. (1976) 'On the Semantic Interpretation of the Thomason 1972 Fragment'. Distributed by Indiana University Linguistics Club, Bloomington, Indiana.

[Tur87] Turner, R. (1987) 'A Theory of Properties.' *Journal of Symbolic Logic* 52, 63–86.

[Wil81] Williams, E. (1981) 'Argument Structure and Morphology.' *Linguistic Research* 1, 81–114.

90/18	J.Coenen E.v.d.Sluis E.v.d.Velden	Design and implementation aspects of remote procedure calls, p. 15.
90/19	M.M. de Brouwer P.A.C. Verkoulen	Two Case Studies in ExSpect, p. 24.
90/20	M.Rem	The Nature of Delay-Insensitive Computing, p.18.
90/21	K.M. van Hee P.A.C. Verkoulen	Data, Process and Behaviour Modelling in an integrated specification framework, p. 37.
91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses "if...,then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of *P*-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.
91/07	E.Poll	CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.
91/08	H. Schepers	Terminology and Paradigms for Fault Tolerance, p. 25.
91/09	W.M.P.v.d.Aalst	Interval Timed Petri Nets and their analysis, p.53.
91/10	R.C.Backhouse P.J. de Bruin P. Hoogendijk G. Malcolm E. Voermans J. v.d. Woude	POLYNOMIAL RELATORS, p. 52.
91/11	R.C. Backhouse P.J. de Bruin G.Malcolm E.Voermans J. van der Woude	Relational Catamorphism, p. 31.
91/12	E. van der Sluis	A parallel local search algorithm for the travelling salesman problem, p. 12.
91/13	F. Rietman	A note on Extensionality, p. 21.
91/14	P. Lemmens	The PDB Hypermedia Package. Why and how it was built, p. 63.

91/15	A.T.M. Aerts K.M. van Hee	Eldorado: Architecture of a Functional Database Management System, p. 19.
91/16	A.J.J.M. Marcelis	An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.
91/17	A.T.M. Aerts P.M.E. de Bra K.M. van Hee	Transforming Functional Database Schemes to Relational Representations, p. 21.
91/18	Rik van Geldrop	Transformational Query Solving, p. 35.
91/19	Erik Poll	Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.
91/20	A.E. Eiben R.V. Schuwer	Knowledge Base Systems, a Formal Model, p. 21.
91/21	J. Coenen W.-P. de Roever J.Zwiers	Assertional Data Reification Proofs: Survey and Perspective, p. 18.
91/22	G. Wolf	Schedule Management: an Object Oriented Approach, p. 26.
91/23	K.M. van Hee L.J. Somers M. Voorhoeve	Z and high level Petri nets, p. 16.
91/24	A.T.M. Aerts D. de Reus	Formal semantics for BRM with examples, p. 25.
91/25	P. Zhou J. Hooman R. Kuiper	A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and complete ness, p. 52.
91/26	P. de Bra G.J. Houben J. Paredaens	The GOOD based hypertext reference model, p. 12.
91/27	F. de Boer C. Palamidessi	Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
91/28	F. de Boer	A compositional proof system for dynamic proces creation, p. 24.
91/29	H. Ten Eikelder R. van Geldrop	Correctness of Acceptor Schemes for Regular Languages, p. 31.
91/30	J.C.M. Baeten F.W. Vaandrager	An Algebra for Process Creation, p. 29.
91/31	H. ten Eikelder	Some algorithms to decide the equivalence of recursive types, p. 26.

91/32	P. Struik	Techniques for designing efficient parallel programs, p. 14.
91/33	W. v.d. Aalst	The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.
91/34	J. Coenen	Specifying fault tolerant programs in deontic logic, p. 15.
91/35	F.S. de Boer J.W. Klop C. Palamidessi	Asynchronous communication in process algebra, p. 20.
92/01	J. Coenen J. Zwiers W.-P. de Roever	A note on compositional refinement, p. 27.
92/02	J. Coenen J. Hooman	A compositional semantics for fault tolerant real-time systems, p. 18.
92/03	J.C.M. Baeten J.A. Bergstra	Real space process algebra, p. 42.
92/04	J.P.H.W.v.d.Eijnde	Program derivation in acyclic graphs and related problems, p. 90.
92/05	J.P.H.W.v.d.Eijnde	Conservative fixpoint functions on a graph, p. 25.
92/06	J.C.M. Baeten J.A. Bergstra	Discrete time process algebra, p.45.
92/07	R.P. Nederpelt	The fine-structure of lambda calculus, p. 110.
92/08	R.P. Nederpelt F. Kamareddine	On stepwise explicit substitution, p. 30.
92/09	R.C. Backhouse	Calculating the Warshall/Floyd path algorithm, p. 14.
92/10	P.M.P. Rambags	Composition and decomposition in a CPN model, p. 55.
92/11	R.C. Backhouse J.S.C.P.v.d.Woude	Demonic operators and monotype factors, p. 29.
92/12	F. Kamareddine	Set theory and nominalisation, Part I, p.26.
92/13	F. Kamareddine	Set theory and nominalisation, Part II, p.22.
92/14	J.C.M. Baeten	The total order assumption, p. 10.
92/15	F. Kamareddine	A system at the cross-roads of functional and logic programming, p.36.
92/16	R.R. Seljée	Integrity checking in deductive databases; an exposition, p.32.
92/17	W.M.P. van der Aalst	Interval timed coloured Petri nets and their analysis, p. 20.

92/18	R.Nederpelt F. Kamareddine	A unified approach to Type Theory through a refined lambda-calculus, p. 30.
92/19	J.C.M.Baeten J.A.Bergstra S.A.Smolka	Axiomatizing Probabilistic Processes: ACP with Generative Probabilities, p. 36.
92/20	F.Kamareddine	Are Types for Natural Language? P. 32.
92/21	F.Kamareddine	Non well-foundedness and type freeness can unify the interpretation of functional application, p. 16.
92/22	R. Nederpelt F.Kamareddine	A useful lambda notation, p. 17.
92/23	F.Kamareddine E.Klein	Nominalization, Predication and Type Containment, p. 40.
92/24	M.Codish D.Dams Eyal Yardeni	Bottum-up Abstract Interpretation of Logic Programs, p. 33.
92/25	E.Poll	A Programming Logic for $F\omega$, p. 15.