This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology. Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review. Copies of these notes are available from the author.

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

All rights reserved
editors: prof.dr.M.Rem
        prof.dr.K.M.van Hee.
Abstract

In this paper we shall meet the application of Scott domains to nominalisation and explain its problem of predication. We claim that it is not possible to find a solution to such a problem within semantic domains without logic. Frege structures are more conclusive than a solution to domain equations and can be used as models for nominalisation. Hence we develop a type theory based on Frege structures and use it as a theory of nominalisation.

**Keywords:** Frege structures, Nominalisation, Logic and Type freeness.
1 Frege structures, a formal introduction

Having in part I informally introduced Frege structures, I shall here fill in all the technical
details and show that Frege structures exist.

Consider $F_0, F_1, \ldots$, a family $F$ of collections where $F_0$ is a collection of objects, and
$(\forall n > 0)[F_n$ is a collection of $n$-ary functions from $F_0^n$ to $F_0]$.

**Definition 1.1 (An explicitly closed family)** A family $F$ as above is explicitly closed iff:
For every expression $e[x_1, \ldots, x_n]$ of the metalanguage built up in the usual way from variables ranging over $F_0$ and constants ranging over $\cup_n F_n$, the $n$-place function denoted by $< e[x_1, \ldots, x_n]/x_1, \ldots, x_n >$ is in $F_n$. More formally, $F$ is explicitly closed iff 1, 2 and 3 below hold:

1. Closure under constant functions: For each $a$ in $F_0$, the function $f_a$ is in $F_1$, where $(\forall x) f_a(x) = a$.

2. Closure under composition: For each $f$ in $F_m$, for each $f_1, \ldots, f_m$ in $F_k$, $f(f_1, \ldots, f_m)$ is in $F_k$ where $(f(f_1, \ldots, f_m))(x_1, \ldots, x_k) = f(f_1(x_1, \ldots , x_k), \ldots, f_m(x_1, \ldots , x_k))$.

3. Closure under projection: For each $n, i \geq 1, P_i^n$ is in $F_n$ where $P_i^n(a_1, \ldots , a_n) = a_i$ for each $a_i$ in $F_0$ and, $1 \leq i \leq n$.

For example, if $f$ and $g$ are unary functions of $F$ and $h$ is a binary function of $F$, then $< f \circ g(h(x_1, x_2))/x_1, x_2 >$ is a 2-ary function (i.e. in $F_2$).

In what follows, we assume such a closed family and call it $F$.

**Definition 1.2 (F-functional)** A function $D : F_{n_1} \times \cdots \times F_{n_k} \rightarrow F_0$ is an F-functional with respect to the explicitly closed family $F$, iff: $(\forall m \geq 0)(\forall f_1 \in F_{m+n_1}) \cdots (\forall f_k \in F_{m+n_k})$

$[< D(< f_1(\bar{y}, \bar{x_1})/x_1 >, \ldots , < f_k(\bar{y}, \bar{x_k})/x_k >)/y >]$ is in $F_m$

where $\bar{y}$ is a list of $m$-variables and $\bar{x_i}$ is a list of $n_i$ variables, for $i = 1, \ldots , k$.

Note that if $f_1, \ldots , f_k$ are 1-place functions and $D : F_1 \times \cdots \times F_1 \rightarrow F_0$ then $D(f_1, \ldots , f_k)$ is in $F_0$. What is the intuitive meaning of $F$-functionals? We know that an $F$-functional is a functional, so that it operates on functions. But once we include functionals in the structure, we need to ensure that any expression which contains functionals should actually be in the structure. Assume for the sake of argument that $D : F_{n_1} \times \cdots \times F_{n_k} \rightarrow F_0$ is an $F$-functional. Assume also that for some $m \geq 0, f_i$ is in $F_{m+n_i}$ for $i = 1, \ldots , k$. We know that according to the explicit closure, if $\bar{y}$ is a list of $m$-variables ranging over $F_0$ and for each $i, \bar{x_i}$ is a list of $n_i$ variables ranging over $F_0$, then $< f_i(\bar{y}, \bar{x_i})/x_i >$ is an element of $F_{n_i}$ for each $i$. Therefore it makes sense to talk of the expression $D(< f_1(\bar{y}, \bar{x_1})/x_1 >, \ldots , < f_k(\bar{y}, \bar{x_k})/x_k >)$. This expression however is open in $\bar{y}$ and if we abstract over $\bar{y}$ in this expression are we going to obtain an element of $F_m$? Nothing so far in the structure ensures that this is the case, and we must therefore impose the constraint that these functionals should have such a property. A functional which has this property is called an $F$-functional and now if $D$ is an $F$-functional then $[< D(< f_1(\bar{y}, \bar{x_1})/x_1 >, \ldots , < f_k(\bar{y}, \bar{x_k})/x_k >)/\bar{y} >]$ is in $F_m$. Hence we extend the definition of explicit closure to the following:

**Definition 1.3 (A super explicitly closed family)** Taking a family as above, we say that this family is super explicitly closed iff for every expression $e[\xi_1, \ldots , \xi_m]$ of the metalanguage,
built up in the usual way from variables ranging over \( \cup_n F_n \) and constants ranging over \( \cup_n F_n \) and over \( F \)-functionals, the \( m \)-place function denoted by \(< e[\xi_1, \ldots, \xi_m]/x_1, \ldots, x_m > \) is an \( F \)-functional.

This notion of explicit closure is going to provide us with the full comprehension principle we have been promising.

**Theorem 1.4** Any explicitly closed family which has variables for functions and objects, constants for objects, functions and \( F \)-functionals, is a super explicitly closed family. (The proof is by an easy induction.)

**Example 1.5** As an example of an explicitly closed family, consider \( P_\omega \) as described previously. Define \( F_0 \) to be the set of all subsets of \( \omega \) (i.e. \( P_\omega \)). Define, for each \( n \geq 0, F_n \) to be the set of all continuous functions from \( F_0^0 \rightarrow F_0 \). Using Part I, it can be easily seen that the constant functions, the projection functions, etc are continuous. It can also be seen that continuity is closed under composition and that any combination \(< e[x_1, \ldots, x_n] > \) of variables for objects and constants for both functions and objects results in the function denoted by \(< e[x_1, \ldots, x_n]/x_1, \ldots, x_n > \) being an element of \( F_n \). Therefore the family \( (F_n)_n \) just obtained from \( P_\omega \) (call it \( FE \)), is an explicitly closed family. Furthermore, \( FE \) is super explicitly closed as it can be proven not only that \(< e[x_1, \ldots, x_n]/x_1, \ldots, x_n > \) denotes a continuous function but also that for any expression \(< e[\xi_1, \ldots, \xi_n] > \) built in the usual way out of variables ranging over \( \cup_n F_n \) and constants ranging over both \( \cup_n F_n \) and \( F \)-functionals, \(< e[\xi_1, \ldots, \xi_n]/\xi_1, \ldots, \xi_n > \) denotes a continuous function.

So far, we have only explicit closure on our structure. But that is not enough to give a logic on the structure. In what follows, we see how to obtain such a logic.

Assume an explicitly closed family \( F \) and a list of logical constants which are the following \( F \)-functionals:

- \( \neg : F_0 \rightarrow F_0 \)
- \( \vee, \wedge, \neg, \exists : F_0 \times F_0 \rightarrow F_0 \)
- \( \forall, \exists : F_1 \rightarrow F_0 \)

**Definition 1.6** (Logical system) A logical system on a super explicitly closed family \( F \), relative to a set of logical constants as above, is \( < \text{PROP}, \text{TRUTH} > \) the set of two collections of objects such that \( \text{TRUTH} \subseteq \text{PROP} \). These two collections are closed under an adopted logical schemata for each logical constant. The logical schemata corresponds to the external logic and tells us, for each logical constant from the list, how to build new propositions out of other ones using the logical constant. It also gives the conditions of truth for the resulting proposition.

**The Logical Schemata**

- **NEGATION** If \( a \) is in \( \text{PROP} \) then \( \neg a \) is in \( \text{PROP} \) and \( \neg a \) is in \( \text{TRUTH} \) iff \( a \) is not in \( \text{TRUTH} \).

- **CONJUNCTION** If \( a, b \) are in \( \text{PROP} \) then \( a \wedge b \) is in \( \text{PROP} \) and \( a \wedge b \) is in \( \text{TRUTH} \) iff \( a \) is in \( \text{TRUTH} \) and \( b \) is in \( \text{TRUTH} \).

- **DISJUNCTION** If \( a, b \) are in \( \text{PROP} \) then \( a \vee b \) is in \( \text{PROP} \) and \( a \vee b \) is in \( \text{TRUTH} \) iff \( a \) is in \( \text{TRUTH} \) or \( b \) is in \( \text{TRUTH} \).
From now on, we shall use something is still missing: even though we built the logical system on the top of an explicitly and logic yet lose the bijection between objects and functions. Therefore, our structure must have such objects to other objects (via app) so that app(>'f,x)

closed structure, where functional abstraction if more in it. The next definition will tell us what.

Definition 1.7

Example 1.8

If we take the system FE of Example 1.5, and if we define \( \lambda : \mathcal{F}_1 \rightarrow \mathcal{F}_0 \) as \( \lambda f = \{(n,m) : m \text{ is in } f(e_n)\} \) where we take \( (n,m) \) to be \( 1/2(n+m)(n+m+1)+m \) and define \( \text{app} : \mathcal{F}_0 \times \mathcal{F}_0 \rightarrow \mathcal{F}_0 \) as \( \text{app}(a,b) = \{m : e_n \subseteq b \text{ for some } n, (n,m) \text{ is in } a\} \); then \( (\lambda, \text{app}) \) forms a \( \lambda \)-system for FE.

Proof: \( \text{app}(\lambda f, a) = \{m : e_n \subseteq a \text{ for some } n \text{ and } (n,m) \text{ is in } \lambda f\} = \{m : e_n \subseteq a \text{ for some } n \text{ and } m \text{ is in } f(e_n)\} = \{m \text{ in } f(e_n) : e_n \subseteq a\} = f(a) \) by continuity.

Therefore \( (\lambda, \text{app}) \) is a \( \lambda \)-system for FE. Actually, FE contains \( \lambda \) and app and so it is a \( \lambda \)-structure, but we leave this to the next definition.

1 Recall that \( \forall, \exists \) and \( \lambda \) are functions from \( \mathcal{F}_1 \) to \( \mathcal{F}_0 \) and hence \( f \) does not necessarily contain any free variables. For example, \( \langle x/z \rangle \) is the identity function and contains no free variables, \( \lambda <x/z> \) can be written as \( \lambda x.x \). The same holds for \( \forall <x/z> \) and \( \exists <x/z> \). This might be confusing, as it might be asked if \( <x/z> \) has no free variables, then what does \( \lambda <x/z> \) mean? Despite the fact that \( <x/z> \) has no free variables (let us denote \( <x/z> \) by \( I \)), it is still an element of \( \mathcal{F}_1 \). I.e. it is still a function and we need to make it an object by nominalizing it. Therefore we turn it into an element of \( \mathcal{F}_0 \) by using \( \lambda \). Now \( \lambda f \) is in \( \mathcal{F}_0 \) and \( \text{app}(I, a) = I(a) = a \). In fact, \( \lambda \) does not abstract on free variables, it is \( <f> \) which does so. \( \lambda \) just turns a function into an object preserving the comprehension axiom: \( \text{app}(\lambda f, x) = f(x) \). When we say \( \lambda x.x \), we don't mean that \( \lambda \) abstracts over \( x \) in \( x \), rather we mean that we first abstract via \( <f> \) obtaining \( <x/z> \) and then we look for the nominal of \( <x/z> \)

2 Note the two equal signs, \( = \). The first is a functional from \( \mathcal{F}_0 \times \mathcal{F}_0 \) to \( \mathcal{F}_0 \) such that \( a=b \) is always a proposition and \( a=b \) is true iff we can prove from the rules of \( \lambda \) -calculus with logic that we are formulating that \( a = b \). For example, we know from above that \( \text{app}(\lambda f, x) = f(x) \), hence the proposition \( \text{app}(\lambda f, x) = f(x) \) is true.
Definition 1.9 (\(\lambda\)-structure) A \(\lambda\)-structure is an explicitly closed family \(F\) which has a \(\lambda\)-system.

Note that the \(\lambda\)-structure contains \(\lambda\) and \text{app} and that it is an explicitly closed family.

Example 1.10 Now take the \(\lambda\)-system \(FE\) given in Example 1.8. \(FE\) is also a \(\lambda\)-structure having \((\lambda, \text{app})\) as \(\lambda\)-system, because both \(\lambda\) and \text{app} are in \(FE\), as \(FE\) is explicitly closed.

Definition 1.11 (Frege structures) A Frege structure is a logical system relative to a list of logical constants on an explicitly closed family \(F\), together with a \(\lambda\)-system.

Example 1.12 As an example of a Frege structure, take the \(\lambda\)-structure \(FE\) given in Example 1.10 and which has a \(\lambda\)-system \((\lambda, \text{app})\). Aczel (in [Aczel 1985]) showed that each \(\lambda\)-structure can be extended to a Frege structure. Therefore we now have an example of a Frege structure.

Let us sketch the proof of how our particular \(\lambda\)-structure \(FE\) can be extended to a Frege structure. This will make the reader understand the notion of Frege structure, and get him used to working with it. Before proceeding, however, we must define two missing notions: that of an independent family of \(F\)-functionals and of a primitive \(F\)-functional.

Definition 1.13 We say that a family of \(F\)-functionals is independent iff for any two \(F\)-functionals in the family, the range of values of those \(F\)-functionals are disjoint.

This implies that if \(F\) and \(G\) belong to an independent family of \(F\)-functionals, then for any \(\tilde{f}\) and \(\tilde{g}\) such that \(F(\tilde{f}) = G(\tilde{g})\), we should definitely have \(F = G\). From independence only we cannot conclude that \(\tilde{f} = \tilde{g}\). For this we need primitivity and this is the next notion we define.

Definition 1.14 We say that an \(F\)-functional \(F : F_{n_1} \times \ldots \times F_{n_k} \to F_0\) is primitive iff there exists a projection \(P_i\) in \(F_{n_i+1}\) for each \(1 \leq i \leq k\) such that \(P_i(F(\tilde{f}), \tilde{a}) = f_i(\tilde{a})\) where \(\tilde{f} = f_1, \ldots, f_k\) is in \(F_{n_1} \times \ldots \times F_{n_k}\) and \(\tilde{a}\) is in \(F_1\).

The aim of primitive \(F\)-functionals is similar to injectivity; if we have \(F(\tilde{f}) = F(\tilde{g})\) then we should be able to deduce \(\tilde{f} = \tilde{g}\). It can be easily checked from the definition of \(F\)-primitiveness that this is the case.

The proof that we can extend any \(\lambda\)-structure into a Frege structure is based on two theorems. The first is one which asserts the existence of an independent family of primitive \(F\)-functionals on the \(\lambda\)-structure, which include the logical constants, \(\land, \lor\) etc. It simply states that:

Theorem 1.15 If for each natural number \(m\) we let \((v_{m_1}, \ldots, v_{m_k})\) be a finite sequence of natural numbers, then there is an independent family of primitive \(F\)-functionals:

\[
F_m : F_{v_{m_1}} \times \ldots \times F_{v_{m_k}} \to F_0, \text{ for } m = 0, 1, 2, \ldots
\]

The second is the well known fixed point theorem which applies to monotonic operators and helps us to find the logical schema of these logical constants. This theorem simply states the following:
Theorem 1.16 If \( A \) is a partially ordered collection of objects such that every chain in \( A \) has a least upper bound then any monotonic operator \( Y \) from \( A \) to \( A \) has a fixed point. That is \((\exists a \in A)[Y(a) = a]\).

Let us apply those two theorems to our \( \mathcal{F}E \) and obtain out of it a Frege structure. Up to here, we know that the \( \lambda \)-structure \( \mathcal{F}E \) exists and Theorem 1.15 enables us to find all the logical constants needed. What remains to turn it into a Frege structure is to find a logical system for the logical constants. This is the task of Theorem 1.16. The idea is to associate with each logical constant two predicates which will ultimately (after we get to the fixed point) give all the propositions obtained from the logical constant and all the truths respectively. The construction is well known mathematically and is similar to the one followed by Kripke in [Kripke 1963]. Now consider our \( \lambda \)-structure \( \mathcal{F}E \). We can be sure from Theorem 1.15 that we have a list of \( F \)-functionals which includes:

\[ \neg : F_0 \rightarrow F_0 \]
\[ \lor, \land, \rightarrow, =, \neq : F_0 \times F_0 \rightarrow F_0 \]
\[ \forall, \exists : F_1 \rightarrow F_0 \]

But we still need to make sure that they satisfy the closure properties we want to impose on them. I shall here try to make the construction a little easier than that described by Aczel (in [Aczel 1985]). To construct a logical schema for each constant, i.e. to define the whole logical system, we follow Aczel's intended construction but will carry an example with us at all times. The logical system is defined inductively. As the basis of the induction, we start with a pair \( \chi_0 = (\chi_{op}, \chi_{ot}) \) such that \( \chi_{ot} \subseteq \chi_{op} \). Intuitively, \( \chi_{op} \) is the set of propositions at stage 0 and \( \chi_{ot} \) is the set of truths at stage 0.

Example 1.17 Let \( \chi_0 = (\chi_{op}, \chi_{ot}) = (\{0,1\}, \{1\}) \). Note that both \( \{0,1\} \) and \( \{1\} \) are in \( P_w \).

Before proceeding to the induction step, we must define a couple of auxiliary predicates which ensure that the logical constants map their arguments into appropriate values. That is, for each logical constant \( F \), there is one predicate \( \Phi_F \) which tests whether a particular tuple of arguments has the correct status of propositionhood, and a second predicate \( \Psi_F \) which states the conditions under which the tuple will be mapped into TRUTH by \( F \). To see why we need this, recall the logical schema for negation that we presented under NEGATION above:

- (1) If \( a \) is in \( PROP \) then \( \neg a \) is in \( PROP \), and \( \neg a \) is in TRUTH iff \( a \) is not in TRUTH.

This is an instance of a general logical schema for those functionals \( F \) in a Frege structure which correspond to truth-functional connectives:

- (2) If \( f \) is in \( F_{n_1} \times \ldots \times F_{n_k} \) and \( C'(F,f) \), then \( F(f) \) is in \( PROP \); and \( F(f) \) is in TRUTH iff \( C(F,f) \), where \( C \) expresses \( F \)'s truth conditions and \( C' \) expresses \( F \)'s propositionhood.

Now it is \( \Phi_F \) which tests that the arguments \( f \) are in \( PROP \), while \( \Psi_F \) does the work of \( C \) in (2).

Example 1.18 \( \Phi_- \) and \( \Psi_- \) take arguments in \((\bigcup \chi_i) \times F_0 \) and

\[ \Phi_-(\chi_0, x) \text{ is: } x \text{ is in } \chi_{op} \]
\[ \Psi_-(\chi_0, x) \text{ is: } x \text{ is in not } \chi_{ot} \]

Thus, \( \Phi_-(\chi_0, x) \) is true of the set \( \chi_{op} = \{0,1\} \), and \( \Psi_-(\chi_0, x) \) is true of all elements in \( F_0 \setminus \chi_{ot} \), i.e. everything except the element 1.
In order to carry out the induction step of the construction, we introduce a principle which determines how the propositions and truths at stage \( i+1 \) are built from the propositions and truths at stage \( i \). The principle has two parts as follows:

**Principle 1.19** \( \chi_{i+1}p \) is the collection of those \( F(\tilde{f}) \) where \( F \) is a logical constant and \( \Phi_F(\chi_i, \tilde{f}) \).

**Principle 1.20** \( \chi_{i+1}t \) is the collection of those \( F(\tilde{f}) \) where \( F \) is a logical constant and both \( \Psi_F(\chi_i, \tilde{f}) \) and \( \Phi_F(\chi_i, \tilde{f}) \).

In other words, given the pair \((\chi_i p, \chi_i t)\), we construct \((\chi_{i+1}p, \chi_{i+1}t)\) in the following way:

First, \( \chi_{i+1}p \) has to contain all and only those elements \( F(\tilde{f}) \) such that \( f \) belongs to the propositions at stage \( i \), i.e. it is in \( \chi_i p \) according to \( \Phi_F(\chi_i, \tilde{f}) \); and second, \( \chi_{i+1}t \) must contain all and only those elements \( F(\tilde{f}) \) such that \( f \) belongs to both the propositions and the truths at stage \( i \), i.e. it is in \( \chi_i p \) and \( \chi_i t \) according to \( \Phi_F(\chi_i, \tilde{f}) \) and \( \Psi_F(\chi_i, \tilde{f}) \). Notice that the principle guarantees that \( \chi_{(i+1)p} \sim \chi_{(i+1)t} \).

**Example 1.21** We wish to build \( \chi_1 = (\chi_1 p, \chi_1 t) \) from \( (\chi_0 p, \chi_0 t) = (\{0, 1\}, \{1\}) \). By Principle 1.19, \( \chi_1 p \) is the set of objects \( \neg x \) such that \( \Phi_-(\chi_0, x) \), i.e. it is the set \( \{\neg 0, \neg 1\} \). By Principle 1.20, \( \chi_1 t \) is the set of objects \( \neg x \) such that \( \Phi_-(\chi_0, x) \) and \( \Psi_-(\chi_0, x) \), i.e. such that \( x \) belongs to \( \chi_0 p \) but does not belong to \( \chi_0 t \). The only thing which satisfies both these conditions is \( 0 \), so \( \chi_1 t = \{\neg 0\} \).

**Example 1.22** \( \Phi_\Lambda \) and \( \Psi_\Lambda \) take arguments in \((\cup \chi_i) \times (\mathbb{F}_0 \times \mathbb{F}_0)\) and

\[
\Phi_\Lambda(\chi_0, (x, y)) \text{ is: } x \text{ and } y \text{ are in } \chi_0 p \\
\Psi_\Lambda(\chi_0, (x, y)) \text{ is: } x \text{ and } y \text{ are in } \chi_0 t
\]

Thus, we can supplement the \( \chi_1 p \) of the previous example with the set of objects \( \Lambda(x, y) \) such that \( (x, y) \subseteq \chi_0 p \times \chi_0 p \), i.e. the set \( \{0 \land 0, 0 \land 1, 1 \land 0, \ldots\} \). Similarly, we add to \( \chi_1 t \) the set of objects \( \Lambda(x, y) \) such that \( (x, y) \subseteq \chi_0 p \times \chi_0 t \), i.e. the set \( \{1 \land 1\} \). Note that according to our example, the collection of objects in \( \text{TRUTH} \) at stage 1 is \( \{1 \land 1, \neg 0\} \).

Note also that \( \neg 0, 1 \land 1, 1 \lor 0 \) are distinct objects, even though they are all in \( \text{TRUTH} \) and all have the same truth value in Frege's terms. If we wish, we could reconstruct Frege's notion of the True and the False by forming the relevant equivalence classes, but Frege structures give us an intensional ontology. This is justified on the grounds that objects with the same truth value, e.g. \( \neg 0 \) and \( 1 \land 1 \) are equivalent in truth value but distinct.

We see that the pair is being enlarged at each step starting from the first step where we take \( \chi_0 p = \{0, 1\} \) and \( \chi_0 t = \{1\} \), with the property that for each \( i \) we have: \( \chi_i t \subseteq \chi_i p \). Note that we are not imposing the condition that \( \chi_i t \subseteq \chi_{(i+1)p} \) or \( \chi_i p \subseteq \chi_{(i+1)p} \); in fact our construction is monotonic in another sense which we shall see below. The aim is now to keep going up to a certain level \( \alpha \) where \( \chi_\alpha = (\chi_\alpha p, \chi_\alpha t) \) is a logical system, because it is obvious that \( \chi_1 \) at the levels we met so far are not logical systems. Take for example \( \chi_0 \) in our example above based on \( FE \). Then \( \chi_0 \) is not a logical system, as can be seen by taking the logical schema for \( \neg \).

If \( a \) is a proposition then \( \neg a \) is a proposition such that \( \neg a \) is true iff \( \neg a \) is not true.

\( \chi_0 \) is not a logical system because \( 1 \) is in \( \chi_0 p \) (supposed to represent propositions) but \( \neg 1 \) is not in \( \chi_0 p \). Nor is \( \chi_1 \) a logical system because \( \neg 1 \) is in \( \chi_{1p} \) but \( \neg \neg 1 \) is not in \( \chi_{1p} \) and so on. To solve this problem, let us consider the fixed point (if it exists) of this construction. It
may be that the fixed point is a logical system and if so, we have succeeded. Before we prove that the fixed point is a logical system, let us remind ourselves again of the construction. The construction is built through an operator $Y$ which takes us from level $i$ to level $i+1$ in such a way that $Y(x_i) = x_{i+1}$, where $x_i = (x_{ip}, x_{it})$, $x_{i+1} = (x_{i+1p}, x_{i+1t})$, $x_{it} \subseteq x_{ip}$, $x_{i+1t} \subseteq x_{i+1p}$. Moreover $x_{i+1p}$ and $x_{i+1t}$ are obtained as follows:

For any $F$-functional $F$, $x_{i+1p}$ is the collection of those $F(!)$ where $F$ is a logical constant and $F(x_i, !)$, and $x_{i+1t}$ is the collection of those objects $F(!)$ where $F$ is a logical constant and both $F(x_i, !)$ and $WF(x_i, !)$. Now we prove that any $x$ such that $x = Y(x)$ is a logical system. To show that, we have to prove that for each logical constant $F$, the logical schemata of $F$ holds in $x$.

Let $F$ be a logical constant whose logical schema is as follows:

If $1$ is in $F_{nl} x \ldots x F_{nk}$ and $F(!)$, then $F(!)$ is in $x_{it}$; and $F(!)$ is in $x_{it}$ iff $WF(x, !)$. Now let us prove that $F(f)$ is in $x_{it}$. Suppose $WF(x, !)$, since also $F(x, !)$ then $F(!)$ is in $x_{ip}$; but $x_{ip} = x_{it}$ (because $x = Y(x)$), therefore $F(f)$ is in $x_{it}$.

We define an ordering $\leq$ on $(x_i)$ as follows: $x_i \leq x_{i+1}$ if

- $x_{ip} \subseteq x_{i+1p}$, and
- if $x$ is in $x_{ip}$, then $x$ is in $x_{it}$ iff $x$ is in $x_{i+1t}$.

With this ordering we can show that $Y$ is monotonic. Note that the levels can be any ordinal even a transfinite one, for if we are at a finite ordinal $i$ we define $Y(x_i) := x_{i+1}$ as above. If we are at a limit ordinal $j$, we define $Y(x_j) = \bigcup x_i$ for $i < j$. Applying the fixed point theorem we get a fixed point of $Y$. The reason for this is of course the monotonicity of the operator $Y$, as we know that the ordering relation $\leq$ is a partial ordering on all those pairs.

2 Scott Domains and nominalisation

The ordering relation on Scott domains makes predication trivial. For, a predicate $P$ is true of all the objects in the model iff it is true of the bottom element. Both semanticians and computer scientists however, share an interest in quantification and hence this problem of predication that faced Turner (in [Turner 1984]) is a major issue for those interested in the semantics of either computer or natural languages and who base their work on Scott domains. The problem can be described as follows: Assume a language which has both objects and functions and assume that wffs are built out of other ones using $\land, \lor, \forall, \exists, \ldots$. If the model is
a Scott domain $E_{\infty}$ then there is no problem interpreting anything which is not a quantified sentence, as the interpretations of all such things are continuous functions and hence belong to the model. Let us choose the following interpretation for the quantifiers $\forall$ and $\exists$

$$[[\forall x \phi]]_{gut} = \begin{cases} 
1 & \text{if for each } d \in D, [[\phi]]_{g[d/x]ut} = 1 \\
0 & \text{if for some } d \in D, [[\phi]]_{g[d/x]ut} = 0 \\
\perp & \text{otherwise}
\end{cases}$$

$$[[\exists x \phi]]_{gut} = \begin{cases} 
1 & \text{if for some } d \in D, [[\phi]]_{g[d/x]ut} = 1 \\
0 & \text{if for each } d \in D, [[\phi]]_{g[d/x]ut} = 0 \\
\perp & \text{otherwise}
\end{cases}$$

Then the following is a proof of the continuity of the quantifier clause for $\forall$. Assume by induction that we have $[[\phi]]$ is continuous where $\phi$ does not involve quantifiers. To prove the continuity of $[[\forall x \phi]]$ (i.e. to prove it in $[\text{ASG} \to [S \to \text{EXT}]]$ where ASG is the collection of assignment functions, S is the collection of states consisting of worlds and times and EXT is the extensional domain of values), we prove it continuous separately in each of its arguments, according to a theorem related to semantic domains.

Let us prove the continuity of $[[\forall x \phi]]$ for $g$ in ASG. Take an $\omega$-sequence $(g_n)_n$ and prove that $[[\forall x \phi]]_{g_{n\omega}} = \bigcup [[\forall x \phi]]_{g_n\omega t}$.

- Assume $[[\forall x \phi]]_{g_{n\omega}} = 0 \iff$ by definition,
  $$(\exists d \in D)([[\phi]]_{g_{n[d/x]ut}} = 0) \iff \text{by induction},$$
  $$(\exists d \in D)(\bigcup [[\phi]]_{g_n[d/x]ut} = 0) \iff \text{by the structure of BOOL},$$
  $$(\exists d \in D)(\exists n \in w)([[\phi]]_{g_n[d/x]ut} = 0) \iff \text{by logical laws},$$
  $$(\exists n \in w)(\exists d \in D)([[\phi]]_{g_n[d/x]ut} = 0) \iff \text{by definition},$$
  $$(\exists n \in w)(\forall x [[\phi]]_{g_n[d/x]ut} = 0) \iff \text{by the structure of Bool},$$
  $$\bigcup [[\forall x \phi]]_{g_n\omega t} = 0$$

- Assume $[[\forall x \phi]]_{g_{n\omega}} = 1 \iff$ by definition,
  $$(\forall d \in D)([[\phi]]_{g_{n[d/x]ut}} = 1) \iff \text{by induction},$$
  $$(\forall d \in D)(\bigcup [[\phi]]_{g_n[d/x]ut} = 1) \iff \text{by the structure of BOOL},$$
  $$(\forall d \in D)(\exists n \in w)([[\phi]]_{g_n[d/x]ut} = 1) \iff u \subseteq d \text{ and monotonicity},$$
  $$(\exists n \in w)([[\phi]]_{g_n[d/x]ut} = 1) \iff \text{monotonicity},$$
  $$(\exists n \in w)(\forall d \in D)([[\phi]]_{g_n[d/x]ut} = 1) \iff \text{by definition},$$
  $$(\exists n \in w)(\forall x [[\phi]]_{g_n[d/x]ut} = 1) \iff \text{by the structure of Bool},$$
  $$\bigcup [[\forall x \phi]]_{g_n\omega t} = 1$$

Therefore $[[\forall x \phi]]$ is continuous.

Note that this interpretation of quantifiers is abandoned later by Turner (in [Turner' 1984]) and he decided to adopt the following clauses instead:

$$[[\forall x \phi]]_{gut} = \begin{cases} 
1 & \text{if for each } d \in E_{\infty} \setminus \bigcup E_n, [[\phi]]_{g[d/x]ut} = 1 \\
0 & \text{if for some } d \in E_{\infty} \setminus \bigcup E_n, [[\phi]]_{g[d/x]ut} = 0 \\
\perp & \text{otherwise}
\end{cases}$$
Of course working with Scott domains, you have always to check for continuity and this is the case with the new clauses. It can easily be proved that continuity does in fact hold and so we can still think of Scott domains as models.

We now describe the problem which made Turner move from the first definition of quantifiers to the second one. By adopting the first definition, we had: 
\[ [[\exists x \phi]]_{gwt} = \begin{cases} 
1 & \text{if for some } d \in E_\infty \setminus \cup E_n, [[\phi]]_{g[d/x]w} = 1 \\
0 & \text{if for each } d \in E_\infty \setminus E_n, [[\phi]]_{g[d/x]w} = 0 \\
1 & \text{otherwise} 
\end{cases} \]

As [[\phi]] is continuous, therefore monotonic and as \( u \) is the undefined element, then we get: 
\[ (\forall d \in D) [[\phi]]_{g[d/x]w} = 1 \]iff\[ [[\phi]]_{g[u/x]w} = 1. \]

This clause has serious consequences. I shall illustrate this by taking in the formal language an element \( u' \) which names \( u \) (i.e. \( [[u']]_{gwt} = u \) always). Now see what happens if we take \( \phi \) to be: \( x = u' \). Applying the above clause we get:
\[ [[x = u']]_{g[u/x]w} = 1 \]iff\[ (\forall d \in D) [[x = u']]_{g[d/x]w} = 1 \]which implies:
\[ u = u \]iff\[ (\forall d \in D) (d = u). \]

That is absurd. We have to do something about this and the first solution that one thinks of is to exclude the undefined element from the quantifier clause. Therefore, instead of letting \( d \) range over all of \( D \), we let it range over \( D^+ \) (i.e. \( D \setminus \{u\} \)). But now Scott domains can no longer be models under this interpretation, for we no longer have [[\forall x \phi]] is continuous. If we go back to the proof of continuity given above, we see that we had to use the undefined element in order to prove continuity. Turner, realising this, exploits an important aspect of the structure of Scott domains. We explained earlier the existence of finite and infinite elements in \( E_\infty \) and said that for each element \( d \) of \( E_\infty \), \( d \) is the limit of \( (e_n) \) where \( e_n \) belongs to \( E_n \) and each \( E_n \) is the domain of finite elements. The infinite (or ideal elements) are those which are in \( E_\infty \cup \cup E_n \). By restricting the quantification over these ideal elements only, we can prove again the closure of Scott models. However, by so restricting quantification, only infinite elements can be quantified over and finite elements are ignored.

### 3 Frege structures and nominalisation

Frege structures are not only a collection of collections of functions (as in the case of \( E_\infty \)), but they also have a certain logic which works on them, and whose availability solves also the problem of Section 1.1 of part I. Therefore, Frege structures solve both problems of section 1 of part I. In a Frege structure, quantifiers and other connectives are built inductively step by step so that at the fixed point one gets all these logical constants. This availability of logic, makes Frege structures attractive candidates for the semantics of nominalisation. Their other advantage is the type theory that can be built inside them which accommodates self-application. In fact, we mentioned in 2.2.1 of part I that the theory of types was not adequate to the semantics of nominalisation. The typing constraints according to Church's type theory are too restrictive for nominalisation and we need to have functions which can apply to themselves or to items of the same type. Abandoning Church's type theory does not imply getting rid of all the typed theories. We can still keep to typed languages but make the typing adequate to deal with nominalisation. This section will develop a type theory based on Frege structures such that for any two types \( \sigma, \tau \) the type \( < \sigma, \tau > \) is subsumed by the type \( \sigma \). Some types will be circular or vacuous and they will be responsible for avoiding the paradoxes which
threaten theories that combine type freeness and logic. Basically, our method is to allow type
dreness yet to restrict the abstraction of various formulae which belong to various types.
Types can be basic or functional space types. Amongst the functional space types we have
those types which are circular or vacuous. Abstraction is restricted to those formulae which
when abstracted over will belong to a non circular, non vacuous type.

3.1 Polymorphic types

The set of types is the smallest set $\mathcal{T}$ such that

1. $p, t, e$ are in $\mathcal{T}$ and are all distinct.
2. If $\sigma, \tau$ are in $\mathcal{T}$ then $<\sigma, \tau>$ is in $\mathcal{T}$.

The types defined in 1 are basic types, $p$ is the type of propositions, $t$ is the type of those
true propositions (which are many according to the intensional framework) and $e$ is the type
of objects. Of course not every object should be a proposition and not every proposition
should be a truth. 2 gives the complex types. We impose a subsumption relation $\leq$ on the
types as follows:

1. $\sigma \leq e$
2. $t \leq p$
3. $<\sigma, \tau> \leq \sigma$

We also require that $\leq$ be a partial ordering and therefore impose the following addi­tional conditions:

4. $\sigma \leq \sigma$
5. if $\sigma \leq \tau$ and $\tau \leq \sigma$, then $\sigma = \tau$
6. if $\sigma \leq \tau$ and $\tau \leq \rho$ then $\sigma \leq \rho$
7. if $\tau \leq \rho$, then $<\sigma, \tau> \leq <\sigma, \rho>$
8. if $\tau \leq \rho$, then $<\sigma, \tau> \leq <\sigma, \tau>, \tau>$

1-6 are obvious. As an example of 7, take the propositional functions which are of type
$<e, p>$; these functions are also of type $<e, e>$. 8 is there to capture those circular types.
In fact we have the following lemma:

Lemma 3.1 If $\tau \leq \rho$, then $<\sigma, \tau> = <<<\sigma, \tau>, \tau> = <<<\sigma, \tau>, \tau>, \tau> = ...$
$<<<\sigma, \tau>, \tau>, ..., \tau>

Proof: obvious from 3, 5 and 8. $\square$

When $\sigma \leq \tau$, we say that $\tau$ subsumes, or is a more general type than, $\sigma$; intuitively, it
means that any expression which is of type $\sigma$ is also of type $\tau$. Note that $e$ is the maximal
element of the partial order, since it subsumes every type. We shall see that the subsumption
relation plays a central role in polymorphism, and that there are models of such a typing
system; that is, we will have functional domains $X \Rightarrow Y$ such that $(X \Rightarrow Y) \subseteq X$.

Our next task is to extend the definition of type so as to characterize the vacuous types,
that is, the types which may be associated with empty domains. It is useful to first introduce
the auxiliary notion of a p(propositional)-chain type. This is defined inductively as follows:
Definition 3.2 (P-Chain Type)

1. If $p \leq p$ and $\tau = e$ or $\tau = p$ or $\tau = t$ then $\langle \tau, p \rangle$ is a p-chain type.

2. If $\tau$ is a p-chain type, and $p \leq p$ then $\langle \tau, p \rangle$ is a p-chain type.

Example 3.3 $\langle e, p \rangle$, $\langle p, p \rangle$, $\langle t, p \rangle$, $\langle e, e, p \rangle, p \rangle$ (which is equal to $\langle e, p \rangle$), $\langle p, p \rangle, p \rangle$ (which is equal to $\langle p, p \rangle$), $\langle e, t \rangle, \langle t, e \rangle \rangle \ldots$ are p-chain types. Moreover, whenever $\sigma$ is a p-chain type, then so are $\langle \sigma, t \rangle$, $\langle \sigma, p \rangle$, $\langle \sigma, t, \tau \rangle \rangle$ and $\langle \sigma, \langle p, \tau \rangle \rangle$ (for any type $\tau$).

Note however that the following are not p-chain types: $e, \langle e, e \rangle, \langle e, e \rangle \rangle, \ldots$

Vacuous types below will be associated with empty domains.

Definition 3.4 (Vacuous Types) $\sigma$ is a vacuous type iff:

1. $\sigma = \langle \tau, p \rangle$ where $\tau$ and $p$ are p-chain types, and neither $\tau \leq p$ nor $p \leq p$ or

2. $\sigma = \langle \tau, p \rangle$ where $p$ is a vacuous type, or

3. $\sigma \leq \tau$, where $\tau$ is vacuous.

From 2 and 3 we can conclude that a function space $\langle \sigma, \tau \rangle$ is vacuous if its domain $\sigma$ is vacuous, using $\langle \sigma, \tau \rangle \leq \sigma$.

Example 3.5 The following instances of $\sigma = \langle \tau, p \rangle$ are vacuous:

- $\sigma = \langle \langle e, p \rangle, \langle e, p \rangle \rangle$, by clause 1, since $\tau = p = \langle e, p \rangle$ and not $\langle e, p \rangle \leq p$.

- $\sigma = \langle \langle e, t \rangle, \langle t, e \rangle \rangle, \langle e, t \rangle, \langle t, e \rangle \rangle \rangle$

There are p-chain types which are not vacuous; for example $\langle e, p \rangle$. There are types that are vacuous but not p-chains. For example $\langle \langle e, p \rangle, \langle e, p \rangle \rangle$. There are types which are neither vacuous nor p-chains. For example, $e, \langle e, e \rangle, \ldots$.

3.2 The Syntax of Tpol

The basic expressions of Tpol are as follows:

1. For each type $\sigma$, there exists an infinite number of constants. Constants of type $\sigma$ are referred to as $c_{\sigma}$.

2. For each type $\sigma$, there exists an infinite number of variables. Variables of type $\sigma$ are referred to as $u_{\sigma}$.

Expressions of type $\sigma$, are defined recursively as follows:

1. $u_{\sigma} : \sigma$.

2. $c_{\sigma} : \sigma$.

3. If $\alpha : \tau, u : \sigma$ and $\langle \sigma, \tau \rangle$ is a type which is not vacuous nor circular, then $\lambda u.\alpha : \langle \sigma, \tau \rangle$. 

12
4. If $\alpha : \langle \sigma, \tau \rangle$ and $\beta : \sigma'$, where $\sigma' \leq \sigma$, then $\text{app}(\alpha, \beta) : \tau$.

5. If $\alpha : \sigma, \beta : \sigma'$ and $\sigma \leq \sigma'$, then $\alpha =_{\sigma'} \beta : \rho$.

Suppose $\phi : \rho$ and $\psi : \rho$ then

6. $\neg \phi : \rho$ and $\neg \phi : \tau$ iff not $\phi : \tau$.

7. $[\phi \lor \psi] : \rho$ and $[\phi \lor \psi] : \tau$ iff $\phi : \tau$ or $\psi : \tau$.

8. $[\phi \land \psi] : \rho$ and $[\phi \land \psi] : \tau$ iff $\phi : \tau$ and $\psi : \tau$.

9. $[\phi \supset \psi] : \rho$ and $[\phi \supset \psi] : \tau$ iff $\phi : \tau$ whenever $\phi : \tau$.

10. $[\phi \equiv \psi] : \rho$ and $[\phi \equiv \psi] : \tau$ iff $\phi : \tau$ if $\phi : \tau$.

11. If $\phi : \rho$ and $u$ is a variable of any type $\sigma$ then $\forall u \phi : \rho$, and $\forall u \phi : \tau$ iff $\phi[a/u] : \tau$ for every constant $a : \sigma$.

12. If $\phi : \rho$ and $u$ is a variable of any type $\sigma$ then $\exists u \phi : \rho$, and $\exists u \phi : \tau$ iff $\phi[a/u] : \tau$ for some constant $a : \sigma$.

13. If $\sigma' \leq \sigma$, then $\alpha : \sigma'$ implies $\alpha : \sigma$.

Notice that we have placed a syntactic restriction of $\lambda$-abstraction to ensure that abstracts never have vacuous or circular types.

3.2.1 Axioms

In our system, self-application is only possible for those expressions which have a complex type; indeed, this is what is required by clause 4 of the syntax above.

- (a) $(\lambda x. \alpha) : \langle \sigma, \tau \rangle = (\lambda y. \alpha[y/x]) : \langle \sigma, \tau \rangle$, where $y$ is not free in $\alpha$.
- (b) $\text{app}(\lambda x. \alpha) : \langle \sigma, \tau \rangle, \beta : \sigma' = \alpha[\beta/x] : \tau$, if $\sigma' \leq \sigma$
- (c) If $\alpha_1 : \langle \sigma, \tau \rangle = \alpha_2 : \langle \sigma, \tau \rangle$ and $\beta_1 : \sigma = \beta_2 : \sigma$, then $\text{app}(\alpha_1, \beta_1) : \tau = \text{app}(\alpha_2, \beta_2) : \tau$
- (d) If $(\alpha_1 = \alpha_2) : \sigma$ and $(\alpha_1 = \alpha_3) : \sigma$, then $(\alpha_2 = \alpha_3) : \sigma$
- (e) If $\text{app}(\alpha_1, x) : \tau = \text{app}(\alpha_2, x) : \tau$, then $\alpha_1 : \langle \sigma, \tau \rangle = \alpha_2 : \langle \sigma, \tau \rangle$ where $x : \sigma$ is not free in $\alpha_1, \alpha_2$ or any other assumption.
- (z) $\text{app}(\lambda x. \alpha_1, \beta_2) = \text{app}(\lambda x'. \alpha_1, \beta_2)$ where $x : \sigma, x' : \sigma', \sigma' \leq \sigma$, and $\beta_2$ is any term of type $\sigma'$.
- (r) $(\alpha : \sigma =_{\sigma'} \alpha : \sigma') \equiv (\alpha : \sigma =_{\sigma} a : \sigma')$ if $\sigma' \leq \sigma$.
- (n) $(\alpha : \sigma = \alpha : \sigma) : \tau$
The following version of \( \eta \)-conversion is derivable:

If \( E : \langle \sigma, \sigma' \rangle \) then \( \lambda x. E x : \langle \sigma, \sigma' \rangle = E : \langle \sigma, \sigma' \rangle \) for \( x : \tau \) free in \( E \) and \( \tau \leq \langle \sigma, \sigma' \rangle \).

Proof

\[
\lambda x. E x : \langle \tau, \langle \sigma, \sigma' \rangle \rangle \leq \tau \leq \langle \sigma, \sigma' \rangle \quad E : \langle \sigma, \sigma' \rangle \quad y : \sigma \text{ from } (\beta)
\]

\[
(\lambda x. E x) y : \sigma' = E[x/y] : \sigma' = E y : \sigma' \text{ from } (\epsilon)
\]

Axioms \((\alpha), (\beta), (\gamma)\) and \((\delta)\) are standard typed \( \lambda \)-calculus axioms. Axiom \((\epsilon)\) is the extensionality axiom. It says that if \( \alpha_1 \) and \( \alpha_2 \) give the same results for the same arguments, then they are equal. Axiom \((\zeta)\) says that if \( f : A \rightarrow B \) and if \( f/A' \) is the restriction of \( f \) to \( A' \subseteq A \), then \( f \) and \( f/A' \) give the same results for all elements in \( A' \). Axiom \((\theta)\) says that if \( \alpha : \sigma' \) and if \( \sigma' \leq \sigma \) then saying that \( \alpha \) equals to itself in \( \sigma \) is the same as saying that \( \alpha \) is equal to itself in \( \sigma' \). Axiom \((\rho)\) is the reflexivity of \( = \).

### 3.2.2 Russell's and Curry's Paradoxes

Russell's paradox does not occur here because paradoxical expressions of the form \( \lambda x. \neg \text{app}(x, x) \) are not well-formed. In fact, we have the following lemma:

**Lemma 3.6** If \( x \) is of type \( \langle \sigma, p \rangle \), then \( \lambda x. \neg \text{app}(x, x) \) of type \( \langle \sigma, p \rangle, p \rangle \) is not well-formed.

**Proof** According to the definition of meaningful expressions, it is enough to show that \( \langle \sigma, p \rangle, p \rangle \) is a circular type. This is obvious from Lemma 3.1. □

In fact, we have an even stronger lemma:

**Lemma 3.7** If \( x \) is of type \( \langle \sigma, \tau \rangle \), where \( \tau \leq p \), then \( \lambda x. \neg \text{app}(x, x) \) of type \( \langle \sigma, \tau \rangle, p \rangle \) is not well-formed.

**Proof** Exactly as that of Lemma 3.6. □

With these lemmas, if \( x : \langle \sigma, \tau \rangle \), where \( \tau \leq p \), then \( \text{app}(x, x) \) is of type \( \tau \leq p \). Hence \( \neg \text{app}(x, x) \) is of type \( p \). But \( \lambda x. \neg \text{app}(x, x) \) is not well-formed in \( T_{pol} \), due to clause 3 in the definition of the expressions of a type, since its type, namely \( \langle \langle e, p \rangle, p \rangle \), is circular.

Curry's paradox comes from the presence of \((DT), (MP)\) and \( \beta \) where \((DT)\) and \((MP)\) are as follows:

\[(DT) \quad \Gamma \cup \{\phi\} \vdash \psi \text{ implies } \Gamma \vdash \phi \rightarrow \psi,\]
\[(MP) \quad \Gamma \vdash \phi \rightarrow \psi \text{ and } \Gamma \vdash \phi \text{ implies } \Gamma \vdash \psi,\]

If we take \( a \) to be the formula \( \lambda x. \text{app}(x, x) \rightarrow \bot \), then

1. \( \text{app}(a, a) = \text{app}(a, a) \rightarrow \bot \) by \( \beta \)-conversion
2. \( \text{app}(a, a) \vdash \text{app}(a, a) \), trivial
3. \( \text{app}(a, a) \vdash \text{app}(a, a) \rightarrow \bot \) by 1
4. \( \text{app}(a, a) \vdash \bot \) by \((MP)\) applied to 2 and 3
5. \( \text{app}(a, a) \vdash \bot \) by \((DT)\)
6. \( \vdash \text{app}(a, a) \) by 1
7. \( \vdash \bot \) by \((MP)\) applied to 5 and 6
However, our $(DT)$ and $(MP)$ have the following form:

$(DT) \quad \Gamma \cup \{ \phi : t \} \vdash \psi : t \implies \Gamma \cup \{ \phi : p \} \vdash (\phi \rightarrow \psi) : t$

$(MP) \quad \Gamma \vdash (\phi \rightarrow \psi) : t$ and $\Gamma \vdash \phi : t$ implies $\Gamma \vdash \psi : t$.

If we take $a$ to be the formula $\lambda x.(app(x,x) \rightarrow \bot)$, then

1. $app(a,a) = app(a,a) \rightarrow \bot$ by $\beta$-conversion
2. $app(a,a) : t \vdash app(a,a) : t$, trivial
3. $app(a,a) : t \vdash (app(a,a) \rightarrow \bot) : t$ by 1
4. $app(a,a) : t \vdash \bot : t$ by $(MP)$ applied to 2 and 3
5. $app(a,a) : p \vdash (app(a,a) \rightarrow \bot) : t$ by $(DT)$
6. $app(a,a) : p \vdash app(a,a) : t$ by 1
7. $app(a,a) : p \vdash \bot : t$ by $(MP)$ applied to 5 and 6

However, we cannot show that $app(a,a) : p$. In fact $\lambda x.(app(x,x) \rightarrow \bot)$ is not well formed due to Lemma 3.6 above as its type is $<\sigma',p,p>$. This is because if $x$ is of some type $\sigma$, since $app(x,x)$ has to be of type $p$, we can infer that $\sigma$ must be of the form $<\sigma',p>$. From this it follows that $a$ is of type $<\sigma',p,p>$, which is circular.

### 3.2.3 Models of Tpol

For the present paper we shall concentrate on $F_0$, PROP and SET (where $PROP \cap SET = \emptyset$) and then we shall construct domains inside $F_0$ which represent the types described in our theory Tpol.

Given domains $X,Y$ already in the Frege structure, we build new domains as follows:

$(DOM) \quad X \Rightarrow Y = \{ x \in X : \forall x' \in X [app(x,x') \in Y] \}$

As a special case of $(DOM)$, the domain $(F_0 \Rightarrow PROP) = SET$ inside $F_0$ contains the nominals of propositional functions. Now let us see if the structure of types can be captured by the domains.

**Lemma 3.8** If $X,Y$ are domains, then $(X \Rightarrow Y) \subseteq X$.

Proof Obvious. $\square$

**Lemma 3.9** If $X$ and $Y$ are domains built as above, then $Y \subseteq Y'$ implies $(X \Rightarrow Y) \subseteq (X \Rightarrow Y')$.

Proof If $x \in X \Rightarrow Y$, then $\forall x' \in X, app(x,x') \in Y$, by $(DOM)$. Since $Y \subseteq Y'$, it follows that $\forall x' \in X, app(x,x') \in Y'$ and so $x \in X \Rightarrow Y'$. $\square$

**Lemma 3.10** If $X$ and $Y$ are domains built as above, then $X \subseteq X'$ implies $(X \cap (X' \Rightarrow Y)) \subseteq (X \Rightarrow Y)$.

Proof If $x \in X \cap (X' \Rightarrow Y)$ then $x \in X$, and $x \in (X' \Rightarrow Y)$; by $(DOM)$, $\forall x' \in X', app(x,x') \in Y$. Hence, we have both that $x \in X$ and, since $X \subseteq X'$, $\forall x' \in X, app(x,x') \in Y$. Therefore $x \in X \Rightarrow Y$. $\square$
We now inductively define a relation $\leq$ between arbitrary domains $X$ and the domain $\text{SET}$. This relation is related to the notion of a p-chain type which we defined earlier. The relation $X \leq \text{SET}$ holds iff

1. $X = \text{SET}$, or
2. $X = (X' \Rightarrow Y')$ where $X' \leq \text{SET}$ and $Y' \leq \text{PROP}$. 

We say that a domain $X$ is inductively predicable iff $X \leq \text{SET}$.

**Lemma 3.11** If $X \leq \text{SET}$ then $X \subseteq \text{SET}$.

Proof The proof is by an easy induction. If $X = \text{SET}$ then the property holds. Assume by induction that the property holds up to $X'$, and show that the property holds for $X = (X' \Rightarrow Y')$ where $X' \leq \text{SET}$. By Lemma 3.9, $(X' \Rightarrow Y) \subseteq X'$, and since $X' \subseteq \text{SET}$ by inductive hypothesis, we have by transitivity that $X \subseteq \text{SET}$. $\square$

The following lemma informs us that if $X, Y$ are inductively predicable then $X \Rightarrow Y$ is empty. When we give the denotation of our various types, we will find that the domains associated with vacuous types are always empty.

**Lemma 3.12** $\text{SET} \Rightarrow X$ is empty whenever $X \leq \text{SET}$.

Proof The proof is by induction on $X$:

1. If $X = \text{SET}$ then $\text{SET} \Rightarrow \text{SET}$ is empty, for the following reason. Suppose $x$ is in $\text{SET} \Rightarrow \text{SET}$. Then for every $x' \in \text{SET}$, $\text{app}(x, x') \in \text{SET}$. But $\text{app}(x, x')$ is also in $\text{PROP}$, by the definition of $x$ being a $\text{SET}$. Hence, $\text{PROP} \cap \text{SET}$ is not empty. Contradiction.

2. Assume $\text{SET} \Rightarrow X$ is empty for $X \leq \text{SET}$, and show that the domain $Y = \text{SET} \Rightarrow (X \Rightarrow Y)$ is empty. Suppose $Y$ is not empty, then if $x$ is in $\text{SET} \Rightarrow (X \Rightarrow Y)$, then for any $x' \in \text{SET}$, $\text{app}(x, x')$ is in $X \Rightarrow Y$. Hence $\text{app}(x, x') \in X$ for any $x' \in \text{SET}$. Hence $x$ is in $\text{SET} \Rightarrow X$ which is empty. Contradiction. $\square$

**Theorem 3.13** $X \Rightarrow Y$ is empty for $X, Y \leq \text{SET}$.

Proof The proof is by induction on $X \leq \text{SET}$. If $X = \text{SET}$ then the theorem holds according to Lemma 3.12. Assume the property holds for $X' \leq \text{SET}$, that is, the domain $X' \Rightarrow Y$ is empty for any $Y \leq \text{SET}$; we must show that $(X' \Rightarrow Y') \Rightarrow Y$ is empty for $Y' \subseteq \text{PROP}$. If $Z$ is not empty, i.e. there is some $a$ in $(X' \Rightarrow Y') \Rightarrow Y$, then $a$ is also in $X' \Rightarrow Y'$ and for all $x \in X' \Rightarrow Y'$, $\text{app}(a, x)$ is in $Y' \subseteq \text{PROP}$. But for all $x$ in $X' \Rightarrow Y'$, $\text{app}(a, x)$ is in $Y' \subseteq \text{SET}$. Hence $\text{app}(a, x)$ is in $\text{PROP} \cap \text{SET}$ which is empty, absurd. $\square$

**Example 3.14** The following domains are empty:

- $\text{SET} \Rightarrow \text{SET}$
- $\text{SET} \Rightarrow (\text{SET} \Rightarrow \text{PROP})$
- $(\text{SET} \Rightarrow \text{PROP}) \Rightarrow \text{SET}$ and
- every domain built recursively out of the above three using $\Rightarrow$. 

16
3.2.4 Semantics of Types

A model $M$ for Tpol is a quadriple $<F,\Rightarrow,C,D>$, where

1. $F$ is a Frege structure in which $\text{PROP} \cap \text{SET} = \emptyset$,
2. $\Rightarrow$ is as defined above by $(\text{DOM})$,
3. The function $D$ which maps types into domains of $M$ is defined as follows:
   - $D_e = F_0$,
   - $D_p = \text{PROP}$,
   - $D_t = \text{TRUTH}$,
   - $D_{<\sigma,\tau>} = D_\sigma \Rightarrow D_\tau$, where $<\sigma,\tau>$ is non-vacuous.

4. $C$ is an interpretation function which takes any constant of type $\sigma$ to an object in $D_\sigma$.

We also assume the existence of an assignment function $\alpha$ which takes any variable of a non-vacuous type $\sigma$ to an object in $D_\sigma$.

Lemma 3.15 $D_{<\sigma,\tau>} = (D_\sigma \Rightarrow D_\tau) \subseteq D_\sigma$ where $<\sigma,\tau>$ is non vacuous.

Proof Obvious by Lemma 3.8. □

Lemma 3.16 If $D_\tau \subseteq D_\rho$ then $(D_\sigma \Rightarrow D_\tau) \subseteq (D_\sigma \Rightarrow D_\rho)$.

Proof If $a$ is in $(D_\sigma \Rightarrow D_\tau)$ then $a \in D_\sigma$ and $[\forall x \in D_\sigma](\text{app}(a,x) \in D_\tau)$ and $[\forall x \in D_\sigma](\text{app}(a,x) \in D_\rho)]$. □

Lemma 3.17 If $\sigma \leq \tau$ then $D_\sigma \subseteq D_\tau$.

Proof by induction on $\sigma \leq \tau$.

1. If $\sigma = e$ then $\tau = e$ and $D_\sigma = D_\tau$.
2. If $\sigma = t$ and $\tau = p$ then $D_\sigma = \text{TRUTH}$ and $D_\tau = \text{PROP}$.
3. If $\sigma = <\tau,\rho>$ then $D_\sigma = D_{<\tau,\rho>} \subseteq D_\tau$ by Lemma 3.15.

4. If $\tau \leq p$ then $D_{<\sigma,\tau>} \subseteq D_{<\sigma,\tau,\rho>}$

Proof $(D_\sigma \Rightarrow D_\tau) \subseteq D_\sigma$. Hence by Lemma 3.10,

$(D_\sigma \Rightarrow D_\tau) \cap (D_\sigma \Rightarrow D_\tau) \subseteq (D_\sigma \Rightarrow D_\tau) \Rightarrow D_\tau$.

Hence $(D_\sigma \Rightarrow D_\tau) \subseteq (D_\sigma \Rightarrow D_\tau) \Rightarrow D_\tau$.

Assume that $\tau \leq p$ implies $D_\tau \subseteq D_\rho$. Then $D_{<\sigma,\tau>} \subseteq D_{<\sigma,\rho>}$, by Lemma 3.16.

Note that due to Lemma 3.16, if $<\sigma,\tau>$ is circular, then $D_{<\sigma,\tau>} = D_{<\sigma,\tau,\tau>}$. □

Lemma 3.18 If $\sigma$ is a $p$-chain type and not $\sigma \leq p$ then $D_\sigma \leq \text{SET}$.

Proof The proof is by induction on $\sigma$.

If $\sigma = <e,\tau>$ where $\tau \leq p$ then $D_\sigma \subseteq F_0 \Rightarrow \text{PROP} = \text{SET} \leq \text{SET}$.

Take $\sigma = <\tau,\tau'>$, where $\sigma$ is not $\leq p$ and $\tau$ is a p-chain type and property holds for $\tau$.

- case 1 not $\tau \leq p$, then $D_{<\tau,\tau'>} = D_\tau \Rightarrow D_{\tau'}$ where $D_{\tau'} \subseteq \text{PROP}$ and $D_{\tau} \leq \text{SET}$ by inductive hypothesis. Hence $D_{<\tau,\tau'>} \leq \text{SET}$.
• case 2 $\tau \leq p$ then $<\tau, \tau'> \leq \tau \leq p$. But it is not the case that $<\tau, \tau'> \leq p$ absurd. Hence $\tau$ is not $\leq p$. □

**Lemma 3.19** If $\sigma$ is vacuous then $D_\sigma$ is empty.

**Proof** If $\sigma$ is vacuous, then $\sigma = <\tau, \rho>$ where $\tau$ and $\rho$ are $p$-chain types not $\leq p$ or either $\tau$ or $\rho$ is vacuous. If either is vacuous then nothing to prove. Else, $D_\sigma = D_\tau \Rightarrow D_\rho$ where $D_\tau, D_\rho \leq \text{SET}$ according to Lemma 3.18. Hence by Theorem 3.19, $D_\sigma$ is empty. □

### 4 COMPARISON AND CONCLUSION

In this part, we showed that Frege structures provide a solution to both problems; we provided a type theory where any function belongs to its domain and hence the theory is a suitable framework for nominalisation. Now we assess further the advantages one obtains with Frege structures. We start with type freeness and the fact that $\text{SET}$ is isomorphic to Propositional functions $F_0 \rightarrow \text{PROP}$ and that $\text{SET} \subseteq F_0$. Also, we have the two following functionals:

$$||| : \text{SET} \rightarrow PF_1$$

$$\lambda : PF_1 \rightarrow \text{SET}.$$  

If we assume that the interpretation of verbs takes place in $F_i$ for $i \geq 1$ and thus that $[[\text{walk}]]$ is in $F_1$, then we get: $[[\text{to walk}]]_g = \lambda ([[\text{walk}]]_g)$.  

Now it is straightforward to interpret things like to walk hurts, for: $[[\text{to walk hurts}]]_g = [[\text{hurt}]]_g([[\text{to walk}]]_g) = [[\text{hurt}]]_g(\lambda ([[\text{walk}]]_g))$.  

The advantage of what we just offered lies in the elegance of classifying the denotation of our items. With Montague's and Turner's approaches, one has always to check whether the denotation of an item is in the right domain. With our approach, we do not need to check whether $[[\text{to walk}]]_g$ is in $F_0$ or not using some confusing domain equations. All we had to say was that $[[\text{walk}]]_g$ is in $F_1$; therefore $\lambda [[\text{walk}]]_g$ is in $F_0$. This actually seems to be an encouraging advantage about Frege structures: nominalisation and self reference are a natural process inside the Frege structure. It also seems that we have real application, unlike in Scott domains where application is only through the isomorphic embedding. This is because instead of interpreting things as above into $F_i$, for $i \geq 0$, we can restrict everything to $F_0$ obtaining $[[\text{fun is fun}]]_g = \text{pred}([[\text{fun}]]_g, [[\text{fun}]]_g)$.  

Therefore it seems that by using Frege structures we get the following advantages over Scott domains,

1. Real self application
2. Less cumbersome checking for the right typing than that involved with Scott domains. It is mainly checking the propositionhood of various items to obtain the type of the resulting item.
3. No redundant semantic types
4. Nominalisation seems to flow naturally
5. Quantification

For the sake of completeness, we mention a new approach to a theory of properties proposed by Turner (in [Turner 1987]) which abandons completely the use of Scott domains. Turner's new theory is one which starts from Frege's comprehension principle and restricts
it in such a way that the paradox is no longer derivable. Turner starts with a first order theory which has a pairing system and adds to this theory a new operator \( p \) (to serve as the predication operator) together with the lambda operator. Then in this case, if one assumes full classical logic and Frege's comprehension principle, one will certainly derive the paradox; for, take \( a = \lambda x. \neg p(x, x) \), then \( p(a, a) \leftrightarrow \neg p(x, x)[a/x] \leftrightarrow \neg p(a, a) \). Contradiction.

Of course, the problem does not come from contraction, i.e. \( p(\lambda x. A, t) \rightarrow A(t, x) \) is always true. But the converse implication (i.e. expansion) is problematic. This is due to negation, i.e. if \( A \) is atomic then we can accept \( A(t, x) \rightarrow p(\lambda x. A, t) \). But we cannot accept it when \( A \) is like Russell's property, an atomic term proceeded by a negation sign. This is exactly what guides Turner in setting his theory. For the theory now will have the following axioms replacing Frege's comprehension principle:

\[
\begin{align*}
(E1) & \quad A(t, x) \rightarrow p(\lambda x. A, t) \text{ when } A \text{ is atomic.} \\
(R) & \quad p(\lambda x. A, t) \rightarrow A(t, x). \\
(I) & \quad p(\lambda x. p(\lambda y. A, t), u) \rightarrow p(\lambda y. p(\lambda x. A, u), t)
\end{align*}
\]

Now the abandonment of Frege's full comprehension axiom will impose the use of two logics, one inside the predication operator in addition to the usual one for wffs. This is due to the fact that breaking the equivalence between \( p(\lambda x. A, t) \) and \( A(t, x) \) will disconnect the reasoning about wffs and properties. To build models for \( T \) above, one uses the fixed point operator to turn an ordinary model of the first order theory into a model which will validate in it as many instances of the comprehension axiom as possible. It will of course validate only the safe instances whereas the paradoxical ones will oscillate in truth-values. The inductive step to build the model should be obvious. As an example, one can start with the first order model, and an operator \( PI \) which is empty at the beginning. Then at the next step, extend \( PI \) to also contain the pairs \( < [\lambda x. A], [[t]]_{\text{EM}} > \) such that \( [[A]]_{[\lambda [t]]_{\text{EM}}} = 1 \) and so on until one gets a limit ordinal \( \chi \) where \( PI \) then is to have in it all the pairs \( < e, d > \) such that for some ordinal smaller than this \( \chi, < e, d > \) belongs to all the intermediate \( PI \)'s. Now we no longer have a full comprehension principle and we cannot do with properties what we can do with formulae. But there are still a great deal of things that one can identify between properties and wffs; for example, from \( P(\lambda x. A, t) \) and \( P(\lambda x. B, t) \) one can derive \( p(\lambda x. A \land B, t) \). Turner showed however that theories of Frege structures are weaker than his theory of properties which is a fact that may stand to our advantage for the following reasons. Firstly, Turner can prove at least as much in his theory as one can in a theory based on Frege structures. Secondly, Turner is paying a price for the strength of his theory — mainly his use of two logics (internal and external) rather than one only. On balance it seems better to use a theory based on Frege structures for properties. Doing so gains the advantages of Turner without the complications.

From the point of view of typing, whereas I use a type free theory, Cocchiarella uses a second order one. There are however some similarities and differences in these two ways of typing that I would like to illustrate. According to axiom (9) under 1.1.2 of Part I, we have \( ME_n \subseteq ME_0 \) for all \( n > 1 \), where \( ME_n \) are the meaningful expressions of any type \( n \). For us, we have that \( ME_n \subseteq ME_0 \) for any \( 1 \leq n \) but the pictures of both approaches are quite different. According to our approach these types are related to each other in a chain like way. That is \( ME_n \subseteq ME_{n-1} \ldots \subseteq ME_0 \). For Cocchiarella we have that each \( ME_n \subseteq ME_0 \) for \( n > 1 \), yet no relation exists between \( ME_n \) and \( ME_m \) for \( n \neq m \). Also for Cocchiarella, propositions are not included in objects, even though they can be embedded in \( ME_0 \) by axiom (8) under the same paragraph. Hence Cocchiarella's whole structure can be understood as a collection of objects, which has a denumerably infinite number of subcollections called
functions but where propositions are outside the domain of objects and can be mapped into it. This structure for Cocchiarella is not a structure of types in the sense that we have in the typing structure in [Kamareddine 1988]. In fact everything that Cocchiarella has so far we have; as can be seen in [Kamareddine 1988], a Frege structure is $F_0, \ldots, F_n$, where $F_0$, is the collection of objects, $F_k$ is the collection of $k$-ary functions and each of these $F_k$, can be embedded in $F_0$, by $\lambda x$. What we have in addition is a typing system constructed inside $F_0$, which cannot be found in Cocchiarella's theory. Also, our system is first order in that the quantification over objects and functions is the same, whereas Cocchiarella's system is second order.

5 Acknowledgments

The author is indebted to the anonymous referees whose suggestions played a crucial role in improving both the form and the contents of the paper.

References


In this series appeared:

<table>
<thead>
<tr>
<th>Volume</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>90/2</td>
<td>K.M. van Hee P.M.P. Rambags</td>
<td>Dynamic process creation in high-level Petri nets, pp. 19.</td>
</tr>
<tr>
<td>90/4</td>
<td>A. Peeters</td>
<td>Decomposition of delay-insensitive circuits, p. 25.</td>
</tr>
<tr>
<td>90/5</td>
<td>J.A. Brzozowski J.C. Ebergen</td>
<td>On the delay-sensitivity of gate networks, p. 23.</td>
</tr>
<tr>
<td>90/8</td>
<td>M.B. Josephs</td>
<td>Receptive Process Theory, p. 16.</td>
</tr>
<tr>
<td>90/9</td>
<td>A.T.M. Aerts P.M.E. De Bra K.M. van Hee</td>
<td>Combining the functional and the relational model, p. 15.</td>
</tr>
<tr>
<td>90/10</td>
<td>M.J. van Diepen K.M. van Hee</td>
<td>A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).</td>
</tr>
<tr>
<td>90/11</td>
<td>P. America F.S. de Boer</td>
<td>A proof system for process creation, p. 84.</td>
</tr>
<tr>
<td>90/12</td>
<td>P. America F.S. de Boer</td>
<td>A proof theory for a sequential version of POOL, p. 110.</td>
</tr>
<tr>
<td>90/14</td>
<td>F.S. de Boer</td>
<td>A proof system for the language POOL, p. 70.</td>
</tr>
<tr>
<td>90/15</td>
<td>F.S. de Boer</td>
<td>Compositionality in the temporal logic of concurrent systems, p. 17.</td>
</tr>
<tr>
<td>90/16</td>
<td>F.S. de Boer C. Palamidessi</td>
<td>A fully abstract model for concurrent logic languages, p. 23.</td>
</tr>
<tr>
<td>90/17</td>
<td>F.S. de Boer C. Palamidessi</td>
<td>On the asynchronous nature of communication in logic languages: a fully abstract model based on sequences, p. 29.</td>
</tr>
<tr>
<td>Year</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>----------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>91/02</td>
<td>R.P. Nederpelt, H.C.M. de Swart</td>
<td>Implication. A survey of the different logical analyses &quot;if...,then...&quot;, p. 26.</td>
</tr>
<tr>
<td>91/03</td>
<td>J.P. Katoen, L.A.M. Schoenmakers</td>
<td>Parallel Programs for the Recognition of P-invariant Segments, p. 16.</td>
</tr>
<tr>
<td>91/05</td>
<td>D. de Reus</td>
<td>An Implementation Model for GOOD, p. 18.</td>
</tr>
<tr>
<td>91/06</td>
<td>K.M. van Hee</td>
<td>SPECIFICATIEMETHODEN, een overzicht, p. 20.</td>
</tr>
<tr>
<td>91/07</td>
<td>E.Poll</td>
<td>CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.</td>
</tr>
<tr>
<td>91/12</td>
<td>E. van der Sluis</td>
<td>A parallel local search algorithm for the travelling salesman problem, p. 12.</td>
</tr>
<tr>
<td>91/14</td>
<td>P. Lemmens</td>
<td>The PDB Hypermedia Package. Why and how it was built, p. 63.</td>
</tr>
</tbody>
</table>

An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.

Transforming Functional Database Schemes to Relational Representations, p. 21.

Transformational Query Solving, p. 35.

Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.


Assertional Data Reification Proofs: Survey and Perspective, p. 18.


Z and high level Petri nets, p. 16.

Formal semantics for BRM with examples, p. 25.

A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and completeness, p. 52.

The GOOD based hypertext reference model, p. 12.

Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.

A compositional proof system for dynamic process creation, p. 24.

Correctness of Acceptor Schemes for Regular Languages, p. 31.

An Algebra for Process Creation, p. 29.

Some algorithms to decide the equivalence of recursive types, p. 26.
<table>
<thead>
<tr>
<th>Reference</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>91/33</td>
<td>W. v.d. Aalst</td>
<td>The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.</td>
</tr>
<tr>
<td>91/34</td>
<td>J. Coenen</td>
<td>Specifying fault tolerant programs in deontic logic, p. 15.</td>
</tr>
<tr>
<td>92/01</td>
<td>J. Coenen, J. Zwiers, W.-P. de Roever</td>
<td>A note on compositional refinement, p. 27.</td>
</tr>
<tr>
<td>92/02</td>
<td>J. Coenen, J. Hooman</td>
<td>A compositional semantics for fault tolerant real-time systems, p. 18.</td>
</tr>
<tr>
<td>92/03</td>
<td>J.C.M. Baeten, J.A. Bergstra</td>
<td>Real space process algebra, p. 42.</td>
</tr>
<tr>
<td>92/05</td>
<td>J.P.H.W.v.d.Eijnde</td>
<td>Conservative fixpoint functions on a graph, p. 25.</td>
</tr>
<tr>
<td>92/06</td>
<td>J.C.M. Baeten, J.A. Bergstra</td>
<td>Discrete time process algebra, p. 45.</td>
</tr>
<tr>
<td>92/07</td>
<td>R.P. Nederpelt</td>
<td>The fine-structure of lambda calculus, p. 110.</td>
</tr>
<tr>
<td>92/10</td>
<td>P.M.P. Rambags</td>
<td>Composition and decomposition in a CPN model, p. 55.</td>
</tr>
<tr>
<td>92/13</td>
<td>F. Kamareddine</td>
<td>Set theory and nominalisation, Part II, p. 22.</td>
</tr>
<tr>
<td>92/14</td>
<td>J.C.M. Baeten</td>
<td>The total order assumption, p. 10.</td>
</tr>
<tr>
<td>92/15</td>
<td>F. Kamareddine</td>
<td>A system at the cross-roads of functional and logic programming, p. 36.</td>
</tr>
<tr>
<td>92/16</td>
<td>R.R. Seljée</td>
<td>Integrity checking in deductive databases; an exposition, p. 32.</td>
</tr>
</tbody>
</table>