Type-Theoretic Logic with an Operational Account of Intensionality

Chris Fox (University of Essex)
Shalom Lappin (King's College London)

16th December 2013

1 Introduction

Classical intensional semantic representation languages, like Montague (1974)'s Intensional Logic (IL), do not accommodate fine-grained intensionality. Montague, following Carnap (1947), characterizes intensions as functions from worlds (indices of worlds and times) to denotations, and so reduces intensional identity to equivalence of denotation across possible worlds. As has frequently been noted in the formal semantics literature, this is too coarse a criterion for semantic identity, given that logically equivalent expressions are semantically indistinguishable. Logical equivalence is not a sufficient condition for intersubstitutability in all contexts. Specifically, it breaks down in complements of verbs and adjectives of propositional attitude.[1]

(1) a. Women are under-represented in political life.
$\Leftrightarrow$
b. Adult female homo sapiens are under-represented in political life.

(2) a. John believes that women are under-represented in political life.
$\not\Leftrightarrow$
b. John believes that adult female homo sapiens are under-represented in political life.

(3) a. A set is countable iff there is a bijection between all of its elements and a subset of the set of natural numbers.
$\Leftrightarrow$
b. A set is infinite iff there is a bijection between all of its elements and the elements of one its proper subsets.

(4) a. It is surprising that a set is countable iff there is a bijection between all of its elements and a subset of the set of natural numbers.
$\not\Leftrightarrow$

[1] This Introduction and Section 3 are based on Lappin (2012; 2013)

The main ideas for Section 3 developed out of Shalom Lappin's NASSLLI 2012 course "Alternative Paradigms of Computational Semantics", and from talks that he gave at the University of Gothenburg in April 2012, and in December 2012. He is grateful to the participants in the course and to the audiences of the talks for stimulating feedback. We would also like to thank Robin Cooper, and Dag Westerståhl for very helpful discussion of some of the issues addressed here.

b. It is surprising that a set is infinite iff there is a bijection between all of its elements and the elements of one its proper subsets.

To avoid this difficulty, a fine-grained theory of intensionality must be able to distinguish between provable equivalence and intensional identity. Such a theory is generally described as *hyperintensional*.

Fox & Lappin (2005; 2010) propose Property Theory with Curry Typing (PTCT) as an alternative framework for intensional semantic representation. As originally formalised, PTCT is essentially a first-order system that consists of three components: (i) an untyped λ-calculus, which generates the language of terms, (ii) a rich Curry-style typing system for assigning types to terms, (iii) and a first-order language of well-formed formulas for reasoning about the truth of propositional terms, where these are term representations of propositions. A tableaux proof theory constrains the interpretation of each component of this federated representation language, and it relates the expressions of the different components. Restrictions on each component prevent semantic paradoxes. A model theory allows us to prove the soundness and completeness of the proof theory.

As we shall see later, PTCT uses two notions of equality: intensional identity and extensional equivalence. Two terms can be provably equivalent by the proof theory, but not identical. In this case, they remain intensionally distinct.

In the original presentation of PTCT, the intensions of the representation language are encoded as terms in the untyped λ-calculus. Identity of PTCT terms is governed by the α, β, and η rules of the λ-calculus. But in addition to this, the terms of the untyped λ-calculus can be interpreted as encoding computable functions. This offers a clue as to how we might provide an intuitively appealing characterisation of intensionality in PTCT.

In Section 2 we give a presentation of PTCT. This is a revised formulation of the theory that abstracts away from some of the details of the original presentation. This version offers a more elegant account that avoids the need to use separate notations for intensions and wffs. Rather than having an essentially uninterpreted inscriptional notion of intensionality, in Section 3 we go on to characterise the difference between intensions and extensions in terms of the distinction between the operational and the denotational interpretations of computable functions.

2 PTCT in TPL

We first reprise the details of PTCT. Rather than repeat the original presentation, we characterise the syntax and the logical behaviour of the system within the framework of Type Predicate Logic (TPL, Turner, 2008; 2009). This allows us to give a more abstract specification of the theory. One key change is that we are no longer obliged to give explicit encodings of the intensional logic within the λ-calculus. Intensions and extensions now have a uniform presentation. In effect, it is possible to analyse the truth conditional behaviour of any expression of the appropriate type. Moreover, any expression will have an intensional interpretation if it occurs in an intensional context, and not just if it is expressed as a λ-term.

2.1 PTCT — Property Theory with Curry Typing

Property Theory with Curry Typing (PTCT) is a logic that supports fine-grained intensionality and a rich system of types, including polymorphic types. While being expressive, it is restricted to first-order power. To avoid logical paradoxes, PTCT only allows type membership constraints to appear for quantified variables and statements of equality and identity. In addition, there is no universal type, and type quantification itself is predicative in nature. This balance between expressivity and formal power appears appropriate for an analysis of natural language semantics.[2]

PTCT can be thought of as an amalgam of existing first order Property Theory, such as Turner (1992)'s axiomatisation of Frege Structures (Aczel, 1980), and Curry typing (Curry & Feys, 1958). Turner's Property Theory (PT) provides an axiomatic account of a logic with fine-grained intensionality, where the intensional identity is distinct from the extensional truth conditions of propositions, or the membership of properties. PT allows for compositional analysis within the logic by incorporating the untyped λ-calculus. This gives rise to potential problems with logical paradoxes of self-application. In the case of PT, these are avoided by restricting the class of terms that count as propositions.

For the purposes of natural language semantics, we need a notion of type. But, we require more flexible types than those offered by Simple Type Theory or Higher Order Logic. If we wish to account for a variety of semantic properties without resorting to various *ad hoc* type-shifting mechanisms, then we have to take a different approach. To this end, we adopt Curry-typing (CT). Among other things, this allows us to have genuine polymorphic types, as well as separation subtyping and other dependent types. Such types can be used in the analysis of anaphora and ellipsis. But we still need to be careful to avoid paradoxes and impredicativity.

PTCT gives us an appropriately expressive logic for natural language semantics, which is effectively constrained to the formal power of a many-sorted first-order language Fox & Lappin (2005; 2010). In the original presentation of PTCT we used tableau proof rules operating on expressions in a formal language defined by a conventional BNF grammar. In that account, PTCT has three parts, an essentially first-order language of well-formed formulae (wffs), a language of types, and a language of terms consisting of the untyped λ-calculus. The intensional expressions of PTCT are then given an explicit interpretation as terms in the untyped λ-calculus. The language of wffs allows us to formulate rules corresponding to Curry-typing and truth conditions for intensional terms.[3]

Here we formulate PTCT within the framework of Typed Predicate Logic (TPL, Turner, 2008; 2009) We are using TPL as a meta-theory in which we express the behaviour of PTCT through sequent rules. By adopting this approach, we formulate well-formedness rules for intensional propositions, and their extensional truth conditions, in TPL. We avoid the need for distinct languages for intensional representations and the extensional logic of wffs.[4]

[2] See Fox & Lappin (2005, Chapter 5) for discussion of the paradoxes.

[3] As we shall see, the interpretation of intensionality with the untyped λ-calculus is consistent with an operational interpretation of intensionality.

[4] Among other things, this formalisation of PTCT shows how the presentation of the theory

Rules concerning truth only apply to those expressions for which they are defined. As the rules need not apply to propositional arguments (such as arguments of intensional relations), we obtain fine-grained intensionality by default. In effect, all of our logical expressions correspond to the intensional propositions of the original presentations of PTCT. As in this version there is no overt language of wffs, we are free to adopt the usual syntax for logical expressions for our intensional propositions.[5] Where the original account had "$^{\mathsf{T}}\hat{p}$" representing the truth conditions for an intensional proposition p, we now have the more direct "$p\ \mathsf{True}$".

2.2 TPL — Typed Predicate Logic

In TPL, theories are expressed in terms of judgements and rules that govern these judgements.[6] This includes judgements of grammaticality and of truth.

$T\ \mathsf{Type}$	T is a type
$t : T$	t is of type T
$t\ \mathsf{Prop}$	t is a proposition
$t\ \mathsf{True}$	t is true (a true proposition), also written as just t

Therefore, the syntax and the logic of a theory are specified within the same formal framework.

Rules concerning these judgements are expressed in terms of deductions over sequents of the form

$$\Gamma \vdash \Phi$$

where Γ is the context, and Φ is a judgement that follows in the context. The use of contexts Γ simplifies the presentation of rules that involve discharged assumptions.

A typical sequent rule will have the form

$$\frac{\Gamma_1 \vdash \Phi_1 \quad \Gamma_2 \vdash \Phi_2 \quad \ldots \quad \Gamma_n \vdash \Phi_n}{\Gamma \vdash \Phi}$$

where $\Gamma_i \vdash \Phi_i$ are the premises and $\Gamma \vdash \Phi$ is the conclusion. In the case of rules concerning syntax, the concluding judgement Φ will typically be of the form $p\ \mathsf{Prop}$, or $t : T$. The rules governing logical behaviour will typically involve truth and be of the form $p\ \mathsf{True}$. Such judgements will often be abbreviated to just "p". A contextualised judgement of truth, $\Gamma \vdash p\ \mathsf{True}$, will be written as "$\Gamma \vdash p$".

When the context is the same for all premises and the conclusion, we elide the context "$\Gamma \vdash$", and write rules in the form

$$\frac{\Phi_1 \quad \Phi_2 \quad \ldots \quad \Phi_n}{\Phi}$$

Axioms of a theory are judgements that have no assumptions. These are written in the form

$$\frac{}{\Gamma \vdash \Phi}$$

need not be bound to an explicit interpretation of intensional terms as λ-terms in the logic.

[5] In the original presentation, the language of wffs was expressed in conventional logical syntax. The language of intensions then had to be distinguished from this.

[6] Elsewhere, "Typed Predicate Logic" is sometimes used to refer to constructive theories such as Martin-Löf (1984)'s Type Theory (MLTT). This is not the use intended here.

or simply

$$\frac{}{\Phi}$$

TPL is, then, a meta-theory that offers the resources for expressing theories in a succinct and uniform manner. All aspects of a theory's behaviour, including its syntax, can be expressed as sequent rules in TPL. We can illustrate the essential principles of formulating a theory in TPL using rules for classical conjunction and negation as examples.

Formation rules are used to specify the grammar of a theory. For example, we can give the formation rules for conjunction and negation in a propositional logic as follows.

$$\frac{\Gamma \vdash s\,\mathsf{Prop} \quad \Gamma \vdash t\,\mathsf{Prop}}{\Gamma \vdash (s \wedge t)\,\mathsf{Prop}}\ F\wedge \qquad \frac{\Gamma \vdash t\,\mathsf{Prop}}{\Gamma \vdash \neg\, t\,\mathsf{Prop}}\ F\neg$$

Rules governing the "logical" behaviour of such expressions can be given in terms of judgements of truth. These are formulated with constraints that ensure they only apply to the appropriate kinds of entities. We can exemplify this with introduction and elimination rules for conjunction and negation of propositions.

$$\frac{\Gamma \vdash s\,\mathsf{Prop} \quad \Gamma \vdash t\,\mathsf{Prop} \quad \Gamma \vdash s \quad \Gamma \vdash t}{\Gamma \vdash s \wedge t}\ +\wedge \qquad \frac{\Gamma \vdash s\,\mathsf{Prop} \quad \Gamma, s \vdash \bot}{\Gamma \vdash \neg s}\ +\neg$$

$$\frac{\Gamma \vdash s\,\mathsf{Prop} \quad \Gamma \vdash\ \mathsf{Prop} \quad \Gamma \vdash s \wedge t}{\Gamma \vdash s}\ -\wedge \qquad \frac{\Gamma \vdash s\,\mathsf{Prop} \quad \Gamma, \neg s \vdash \bot}{\Gamma \vdash s}\ -\neg$$

TPL also allows us to define various type systems. As an example, we can present a version of Simple Type Theory (Church, 1940), with entities e, propositions p, and functions $\langle S, T\rangle$ from S to T.

$$\frac{}{\Gamma \vdash e\,\mathsf{Type}} \qquad \frac{}{\Gamma \vdash p\,\mathsf{Type}}$$

$$\frac{\Gamma \vdash S\,\mathsf{Type} \quad \Gamma \vdash T\,\mathsf{Type}}{\Gamma \vdash \langle S, T\rangle\,\mathsf{Type}}\ F\langle\cdot,\cdot\rangle \qquad \frac{\Gamma \vdash f : \langle S, T\rangle \quad \Gamma \vdash a : S}{\Gamma \vdash fa : T}\ app$$

The uniform analysis allowed by TPL is particularly helpful in the case of PTCT, where the theory was previously formulated as a federation of languages (the λ-calculus, a first-order language of wffs, and a language of types). In TPL we give a single collection of formation rules that express all parts of the syntax of PTCT. The truth conditional behaviour of the system is also stated through TPL sequent rules. This obviates the need for distinct languages, and for different notations to encode intensions and extensions.

Before presenting the details of the formalisation of PTCT in TPL, we first make an observation about grammatical independence in TPL. For some systems formulated in TPL, the rules concerning grammar-related judgements (including $t : T$ and $t\,\mathsf{Prop}$) are independent of logical judgements of truth. In such cases the context Γ can be considered as having a proper part concerned only with judgements of grammatical correctness. That part of the context, and the rules governing syntactic judgements, are, then, independent of the rules and judgements concerning the logical behaviour of the system. In general the syntactic and logical behaviour of a system need not be independent of each other. So, for example, we can formulate a theory with a strong notion

of implication ($a \rightarrow^{+} b$), which forms a proposition ($a \rightarrow^{+} b\,\mathsf{Prop}$) if the antecedent is a proposition ($a\,\mathsf{Prop}$), if the consequent is a proposition ($b\,\mathsf{Prop}$), and the antecedent is true ($a\,\mathsf{True}$).

$$\frac{\Gamma \vdash a\,\mathsf{Prop} \quad \Gamma, a\,\mathsf{True} \vdash b\,\mathsf{Prop}}{\Gamma \vdash (a \rightarrow^{+} b)\,\mathsf{Prop}}$$

The syntactic judgements of such a theory then depend upon a logical judgement of truth. The syntax of PTCT is independent of its logical behaviour. We could partition the context into its logical and syntactic parts, but in this presentation we will keep things simple by adopting a uniform context Γ.

2.3 PTCT in TPL

We give the rules that define the core grammar of PTCT, starting with housekeeping structural rules, followed by conditions that govern the "syntax" of PTCT, and then the specification of a well-formed PTCT expression.

These rules involve the types $\pi, \Lambda, \tau, \sigma$. Essentially π corresponds to the type of PTCT-expressible propositions. This is a smaller class than the propositions characterised by TPL's Prop (which can include propositions about PTCT propositions).[7]

The type Λ plays the role of a universal type for PTCT, but it is not expressible within PTCT. This restriction is necessary to avoid the introduction of paradoxes (Fox & Lappin, 2005, Chapter 5). In effect, Λ corresponds to the untyped λ-calculus in the original presentation of PTCT. The identity criteria for Λ determine the conditions of intensional identity for PTCT. We use the identity criteria of the untyped λ-calculus, but other criteria could, in principle, be adopted.

The type τ characterises those types that can be used within PTCT ("object" types), which exclude the universal type, and those things that are meta-level propositions about PTCT. The type σ is a smaller class of such types which are "safe" to quantify over in PTCT polymorphic types without producing impredicative types.

After giving the formation rules that characterise the grammar of PTCT, we define the logical character of the system, starting with the relevant structural rules, and then providing rules that constrain truth judgements about the language.

It is important to realise that while we may make judgements about the truth of a PTCT expression, and although we use conventional logical notation, a PTCT expression is not to be identified with its truth conditions. Distinct PTCT propositions may share the same truth conditions. This gives us fine-grained intensionality (hyperintensionality). The details are elaborated in Section 2.3.3, where distinct notions of intensional identity and extensional equivalence are formalised. The intensionality of PTCT is then illustrated with the example in Section 2.3.5.

For completeness, here we will reformulate all of PTCT within TPL, giving rise to a system that is equivalent to that given by Fox & Lappin (2005, Chapter 5).

[7] In earlier presentations of PTCT (Fox & Lappin, 2005; 2010) we used Prop to represent the type corresponding to PTCT propositions. Here, Prop has already been invoked to represent a judgement in TPL. For this reason, we use π for the type of PTCT propositions.

2.3.1 PTCT — The Language

We begin with rules for the syntax of PTCT. Some of the complexity of these rules is required in order to avoid logical paradoxes, while maximising the expressive power of the language within a relatively weak logic. We also introduce explicit type polymorphism (Section 2.3.4). To this end, the types are stratified into those that are expressible within PTCT, and the smaller class of types over which we may quantify in the rules for polymorphic types.

Structural Rules We assume the usual house-keeping rules of assumption, thinning, and substitution for judgements involving type membership.[8] The assumption rule allows us to infer that a type judgement follows from a context in which that judgement is assumed. The thinning rule embodies the idea that existing judgements continue to hold when such an assumption is added. The substitution rules governs the use of variables, and substitution of specific values for variables.

$$\frac{\Gamma \vdash T\ \mathsf{Type}}{\Gamma, x : T \vdash x : T}\ \mathit{Ass} \qquad \frac{\Gamma \vdash T\ \mathsf{Type} \quad \Gamma \vdash \Phi}{\Gamma, x : T \vdash \Phi}\ \mathit{Thin}$$

$$\frac{\Gamma, x : T \vdash \Phi[x] \quad \Gamma \vdash t : T}{\Gamma \vdash \Phi[x/t]}\ \mathit{Sub}$$

In these rules, Φ stands for any judgement of the theory. The structural rules are required in order for us to be able to manipulate sequents in an appropriate way when performing deductions in the theory. They are stated for completeness, and they do not form a distinctive part of PTCT.

Simply Stratified Types We have the types B for basic individuals, τ for the types that can appear in PTCT (which include B), and σ for the types that can be quantified over in PTCT (which are included in τ). The type π is the type of PTCT expressible propositions. The type π is included in σ, allowing us to formulate PTCT propositions that involve quantification over PTCT propositions.[9] We can declare these types axiomatically.

$$\overline{B\ \mathsf{Type}} \qquad \overline{\sigma\ \mathsf{Type}} \qquad \overline{\tau\ \mathsf{Type}} \qquad \overline{\Lambda\ \mathsf{Type}} \qquad \overline{\pi\ \mathsf{Type}}$$

The type B corresponds to the basic individuals. τ corresponds to types of terms in Λ that can appear in the object language. That is, elements of τ are types, and the elements of those types are in Λ. Furthermore, τ types are closed under function-type formation. If two types S, T can appear in the object language of PTCT, then so can the function type $T \Rightarrow S$.

In essence, PTCT has a simple stratification of types to avoid paradoxes and impredicativity, and remain weak. The type Λ characterises the "untyped"

[8] Here the sequents can involve just that part of the context Γ that concerns type judgements. This could be made explicit if we wished to demonstrate that particular syntactic judgements are independent of their logical behaviour.

[9] The logical paradoxes of self-application are avoided because of the typed equality/identity relation, typed quantification, and the absence of a universal type and "free-floating" type judgements within the language of PTCT propositions. There are other ways the hierarchy of types could be formulated. For example, we could characterise Λ as a type of types which includes τ. That is, if $T : \tau$, then $T : \Lambda$.

λ-calculus terms. It is not an object-level type and so cannot appear in the language of PTCT itself. The type τ characterise the object level types which can appear in PTCT expressions (although τ cannot appear in PTCT itself). The types that belong to τ are restricted in how they can appear. For example, there are no free-floating type judgements in PTCT expressions. The type σ characterises those object-level types over which it is safe to quantify within the object language. It excludes types that themselves include type quantification. They are used to characterise predicative polymorphic types in Section 2.3.4.

$$\frac{T : \tau}{T\ \mathsf{Type}}\ \tau_1 \quad \frac{T : \tau \quad t : T}{t : \Lambda}\ \tau_2 \quad \frac{S : \tau \quad T : \tau}{T \Rightarrow S : \tau}\ \Rightarrow_\tau F$$

σ characterises the class of τ types over which it is safe to quantify in the object language when we define polymorphic types (Section 2.3.4). We restrict this class to avoid impredicative definitions. If we did not require polymorphic types, then we could drop σ (or reduce it to τ).[10] This class of types includes the type of basic individuals, B, and PTCT propositions π, as well as function types, suitably constrained to σ.

$$\frac{T : \sigma}{T : \tau}\ \sigma \quad \frac{}{B : \sigma}\ \sigma_B \quad \frac{}{\pi : \sigma}\ \sigma_\pi \quad \frac{T : \sigma \quad S : \sigma}{T \Rightarrow S : \sigma}\ \Rightarrow_\sigma F$$

We make the judgements $t\ \mathsf{Prop}$ for general meta-level statements, and $t : \pi$ for statements expressed in the quantifiable object-level language (that is, in PTCT itself).

$$\frac{t : \pi}{t\ \mathsf{Prop}}\ \pi$$

Not all (meta-level) propositions (about PTCT) are expressible within PTCT.

We will adopt the untyped λ-calculus to accommodate notions of abstraction and application. To this end, the type Λ is closed under application and abstraction.

$$\frac{s : \Lambda \quad t : \Lambda}{st : \Lambda}\ \Lambda_{\text{app}} \quad \frac{\Gamma, x : \Lambda \vdash t : \Lambda}{\Gamma \vdash \lambda x.t : \Lambda}\ \Lambda_{\text{abs}}$$

Terms in Λ can be Curry-typed, with the following rules controlling the functional types for abstraction and application.[11]

$$\frac{\Gamma, x : S \vdash t : T}{\Gamma \vdash \lambda x.t : S \Rightarrow T}\ \lambda_\Rightarrow + \quad \frac{s : S \Rightarrow T \quad t : S}{st : T}\ \lambda_\Rightarrow -$$

PTCT Propositions — Formation Rules The language π is a restricted logic, where there is no universal type and no untyped identity relation. Therefore types are restricted to τ, which prevents us from including statements about terms that fall outside the language of PTCT. We will see the role that σ plays in the specification of polymorphic types.

We introduce standard connectives and typed quantifiers.

$$\frac{t : \pi \quad s : \pi}{(t \wedge s) : \pi}\ \wedge F \quad \frac{t : \pi \quad s : \pi}{(t \vee s) : \pi}\ \vee F$$

[10] If we were to remove σ from the theory, then we would need to amend the rules so that B and π where made members of τ directly, rather than via their membership of σ.

[11] We define identity in PTCT in terms of λ-identity, although this is not essential.

$$\frac{t : \pi \quad s : \pi}{(t \to s) : \pi} \to F$$

We use $\bot$ to represent absurdity. Negation ($\neg$) is then defined in terms of $\bot$. Given this approach to negation, it is straightforward to produce a constructive version of the theory.

$$\frac{}{\bot : \pi} \bot F \qquad \neg s =_{\text{def}} s \to \bot \qquad s \leftrightarrow t =_{\text{def}} s \to t \wedge t \to s$$

Typed quantifiers can be formed with types belonging to τ, the object-language expressible types of PTCT.

$$\frac{\Gamma, x : T \vdash t : \pi \quad \Gamma \vdash T : \tau}{\Gamma \vdash (\forall x \epsilon T \cdot t) : \pi} \forall F \quad \frac{\Gamma, x : T \vdash t : \pi \quad \Gamma \vdash T : \tau}{\Gamma \vdash (\exists x \epsilon T \cdot t) : \pi} \exists F$$

We will use $=_T$ for full intensional identity, and $\cong_T$ for extensional equivalence, within type T. These relations are restricted to types within τ, to insure that they are PTCT expressible.

$$\frac{t : T \quad s : T \quad T : \tau}{(t =_T s) : \pi} = F$$

$$\frac{t : T \quad s : T \quad T : \tau}{(t \cong_T s) : \pi} \cong F$$

The distinction between equivalence and identity is given by the logical rules.

Proof by structural induction can demonstrate that expressions satisfy appropriate well-formedness criteria. For example, we can only prove $t : S \Rightarrow T$ for cases where $S, T : \tau$. We are concerned to limit which expressions belong to π. In general we do not want to admit anything corresponding to a "circular" expression into the type π. We need to exclude such expressions from π (and Prop) to prevent inconsistency through logical paradoxes.

As we define PTCT identity in terms of λ-equivalence, we need to allow statements about λ-equivalence that fall outside π. As we are using the untyped λ-calculus, the notion of λ-equivalence must be expressed in more general terms.

$$\frac{t : \Lambda \quad s : \Lambda}{(t =_\Lambda s)\,\mathsf{Prop}} =_\Lambda F$$

The core behaviour of PTCT is given in the next section, and logical rules for equivalence and identity are presented in Section 2.3.3. The behaviour of some more complex types is then given in Section 2.3.4.

2.3.2 PTCT — The Logic

We require rules to specify the logic of PTCT. Again, we adopt the usual housekeeping structural rules for the logic, but first we define identity for terms in Λ. We do this for the substitution rule, and then for intensional identity. Here Λ is intended to characterise the untyped λ-calculus.

$$\frac{s : \Lambda \quad t : \Lambda \quad r : \Lambda \quad s =_\Lambda r}{s =_\Lambda s[t/r]} \Lambda\alpha \quad \frac{\Gamma, x : \Lambda \vdash t : \Lambda \quad \Gamma \vdash s : \Lambda}{\Gamma \vdash (\lambda x.t)s =_\Lambda t[s/x]} \Lambda\beta$$

These rules correspond to α and β equivalence for the untyped λ-calculus. We can also add Λ-identity rules corresponding to η equivalence.

$$\frac{\Gamma, x : \Lambda \vdash t : \Lambda}{\Gamma \vdash (\lambda x.t)x =_\Lambda t} \; \Lambda\eta$$

The structural rules are the usual ones for assumption, thinning, and substitution, but now applied to the logic, rather than to the type theory.

$$\frac{\Gamma \vdash t\,\mathsf{Prop}}{\Gamma, t \vdash t} \; Ass' \qquad \frac{\Gamma \vdash t\,\mathsf{Prop} \quad \Gamma \vdash \Phi}{\Gamma, t \vdash \Phi} \; Thin'$$

$$\frac{t : \Lambda}{t =_\Lambda t} \; id_1 \qquad \frac{t =_\Lambda s \quad \phi[t]}{\phi[s]} \; id_2$$

These rules are not distinctive of PTCT. They simply allow us to manipulate sequents in a way that is necessary to sustain a normal logic for the theory.

As we previously noted, we define negation in terms of bottom ($\bot$). A constructive version of the theory can be formulated by simply dropping the second of the following two rules.

$$\frac{\bot \quad t : \pi}{t} \; \bot- \qquad \frac{\Gamma, \neg t \vdash \bot}{\Gamma \vdash t} \; \neg-$$

Conjunction, disjunction, and implication all behave in the usual way.

$$\frac{s \quad t}{(s \wedge t)} \; \wedge+ \qquad \frac{(s \wedge t)}{s} \; \wedge-_1 \qquad \frac{(s \wedge t)}{t} \; \wedge-_2$$

$$\frac{s \quad t}{(s \vee t)} \; \vee+_1 \qquad \frac{t \quad s}{(t \vee s)} \; \vee+_2$$

$$\frac{\Gamma \vdash (s \vee t) \quad \Gamma, s \vdash r \quad \Gamma, t \vdash r}{\Gamma \vdash r} \; \vee-$$

$$\frac{\Gamma, s \vdash t}{\Gamma \vdash s \to t} \; \to+ \qquad \frac{s \to t \quad s}{t} \; \to-$$

The quantifiers behave like normal typed quantifiers, where the types are drawn from τ, the types that can be represented in the object language of PTCT.

$$\frac{\Gamma, x : T \vdash t \quad \Gamma \vdash T : \tau}{\Gamma \vdash \forall x \epsilon T \cdot t} \; \forall+ \qquad \frac{s : T \quad \Gamma \vdash t[s/x] \quad T : \tau}{\exists x \epsilon T \cdot t} \; \exists+$$

$$\frac{\forall x \epsilon T \cdot t \quad s : T}{t[s/x]} \; \forall- \qquad \frac{\Gamma \vdash \exists x \epsilon T \cdot t \quad \Gamma, x : T, t \vdash r}{\Gamma \vdash r} \; \exists-$$

2.3.3 Identity and Equivalence

Statements involving identity and equivalence are only felicitous if the terms to which the relation applies have the same type. This type must be in τ for such statements to be PTCT expressible propositions. Without this restriction, it is possible to derive a logical paradox (see Fox & Lappin, 2005, Chapter 5)).

Identity within a type, $=_T$, is an intensional relation. Here we define it in terms of $=_\Lambda$. This is internalised within PTCT. Effectively it captures the notion of λ-equivalence for the untyped λ-calculus.

$$\frac{T : \tau \quad s : T \quad t : T \quad s =_\Lambda t}{s =_T t} =_T$$

Extensional equivalence ($\cong_T$) for propositions and properties of PTCT is governed by the following rules.

$$\frac{s : \pi \quad t : \pi \quad s \leftrightarrow t}{s \cong_\pi t} \cong_\pi + \quad \frac{s \cong_\pi t}{s \leftrightarrow t} \cong_\pi -$$

$$\frac{s : T \Rightarrow \pi \quad t : T \Rightarrow \pi \quad \forall x : T \cdot sx \leftrightarrow tx}{s \cong_{T \Rightarrow \pi} t} \cong_{\text{PTY}} + \quad \frac{s \cong_{T \Rightarrow \pi} t}{\forall x \epsilon T \cdot sx \leftrightarrow tx} \cong_{\text{PTY}} -$$

Thus two propositions are extensionally equivalent if they share the same truth conditions, and two properties are extensionally equivalent if they apply to the same terms, but identity of truth conditions does not make them identical. The notion of equivalence can be generalised to operational extensional equivalence, where two functions are extensionally equivalent if they always return an identical value when applied to any argument.

$$\frac{\Gamma \vdash f : S \Rightarrow T \quad \Gamma \vdash g : S \Rightarrow T \quad \Gamma, x : S \vdash fx \cong_T gx}{\Gamma \vdash f \cong_{S \Rightarrow T} g} \cong_{S \Rightarrow T} +$$

This also works in the other direction. If two expressions are operationally equivalent, then they give the same result when applied to the same argument.

$$\frac{x : S \quad f \cong_{S \Rightarrow T} g}{fx \cong_T gx} \cong_{S \Rightarrow T} -$$

As with the formation rules, proof by structural induction can demonstrate that with these rules we can only consider the truth conditions of expressions that satisfy appropriate well-formedness criteria. In particular, we can only consider the truth conditions of PTCT expressions that are in π. Such expressions exclude those that might otherwise represent paradoxical assertions.

Next we consider examples of richer PTCT types that can be defined within this framework.

2.3.4 More complex types in PTCT

We illustrate how more complex types can be added to PTCT with examples of explicit polymorphism, and separation sub-types.

Polymorphic Types Polymorphic types involve type variables. There are a range of ways of introducing such types. One is *schematic* polymorphism, where type variables are used to stand for all possible types, but where they do not have an explicit meaning within the language itself.

We are here interested in explicit polymorphism, where type variables and type quantification occur within the language. Without appropriate constraints, such type quantification can lead to an impredicative theory, because a polymorphic type will be defined in terms of quantification that ranges over types that include the very type that is being defined.

In PTCT, we avoid impredicativity by stratifying the types into those that can appear within PTCT, which belong to τ (the "object"-language types of PTCT), and a smaller class over which we can quantify within the types of PTCT, the types that belong to σ. They include B and π, but not the polymorphic types themselves.

We use the notation $\Pi X \cdot T$ to represent a polymorphic type. Here, X is a variable over types, T a type in which X may appear, and Π is universal quantification (over X in T).

$$\frac{\Gamma, X : \sigma \vdash T : \tau}{\Gamma \vdash \Pi X \cdot T : \tau}\ \Pi F$$

A term will belong to the type $\Pi X \cdot T$ if and only if it belongs to T for all possible values of X.

$$\frac{\Gamma, X : \sigma \vdash t : T}{\Gamma \vdash t : \Pi X \cdot T}\ \Pi+ \quad \frac{t : \Pi X \cdot T \quad S : \sigma}{t : T[S/X]}\ \Pi-$$

We can interpret a conjunction such as "*and*" as having a polymorphic type, which expresses the idea that "*and*" combines expressions of the same type, and returns an expression of that type, where this type is $\Pi X \cdot X \Rightarrow (X \Rightarrow X)$.[12]

Separation Separation types are a form of subtype. They are types of the form $\{x \epsilon T \cdot s\}$ that include a propositional constraint s on membership.

$$\frac{\Gamma \vdash T : \tau \quad \Gamma, x : T \vdash s : \pi}{\Gamma \vdash \{x \epsilon T \cdot s\} : \tau}\ SepF$$

A term t will be of the type $\{x \epsilon T \cdot s\}$ if and only if t is in T, and $s[x/t]$ is true.

$$\frac{\{x \epsilon T \cdot s\} : \tau \quad t : T \quad s[x/t]}{t : \{x \epsilon T \cdot s\}}\ Sep+$$

$$\frac{t : \{x \epsilon T \cdot s\}}{t : T}\ Sep-_1 \quad \frac{t : \{x \epsilon T \cdot s\}}{s[x/t]}\ Sep-_2$$

We use separation types to give accounts of anaphora and ellipsis resolution (Fox, 2000; Fox & Lappin, 2005).

2.3.5 Examples

The application of PTCT to natural language is discussed at some length by Fox & Lappin (2005). In that work, the interested reader can find an analysis of anaphora, ellipsis, quantifier scoping and polymorphic expressions within PTCT. Here we will limit ourselves to a couple of examples involving a proposition in both an extensional and an intensional context.

The sentence "*Every man loves a woman*" can be represented by

$$\forall x \epsilon B \cdot man'(x) \rightarrow \exists y \epsilon B \cdot woman'(y) \wedge loves'(x, y)$$

[12] If we were also to add pairings and projections to PTCT, we could "uncurry" this type, and represent it as $\Pi X \cdot (X, X) \Rightarrow X$.

For the representation of the sentence to be in π (and hence a proposition), the following type declarations are required.

$$man' : B \Rightarrow \pi \quad woman' : B \Rightarrow \pi \quad loves' : B \Rightarrow B \Rightarrow \pi$$

The sentence has truth conditions, and so it implicitly has an extensional interpretation. This contrasts with our second example.

The sentence "*John believes every man loves a woman*" is represented as

$$believes'(John')(\forall x \epsilon B \cdot man'(x) \rightarrow \exists y \epsilon B \cdot woman'(y) \wedge loves'(x, y)))$$

In order for the representation of the sentence to be in π (and hence a proposition), we need the following type declaration.

$$believes' : B \Rightarrow (\pi \Rightarrow \pi)$$

In this example, it is important to note that the inner proposition — the "content" of the believe relation — is *not* collapsed to a truth value, nor to a set of worlds. This "inscriptional" intensionality is sufficiently fine-grained to distinguish between distinct mathematical truths — as in (2) — even though they are truth-conditionally equivalent outside an intensional context — as in (1).

We will not provide a formal proof of the equivalence of the TPL formalisation of PTCT and the earlier version. However, it is important to observe a correspondence that is relevant for the second part of the paper. In the TPL characterisation of PTCT, the expressions in π, which have the syntax of propositions, are all elements of Λ — the class of expressions governed by rules corresponding to those of the untyped λ-calculus. Although we have abstracted away from the distinct languages of the original account of PTCT, the *de facto* intensional propositions of PTCT can still be interpreted as λ-calculus representable terms. The rules for PTCT allow us to give truth conditions to these propositions corresponding to these terms when they appear in non-intensional contexts.

2.4 Summary

The recasting of PTCT in TPL can be thought of as a "flattened" version of a stratified logic (Turner, 2005), where Λ is akin to a universal type U_0, but one which lies outside the object-level logic. In this system there are just two levels of types (τ and σ) to avoid impredicative type quantification, rather than a fully-fledged stratification.

Incorporating a semantic theory into a framework like TPL brings into sharp relief the the core properties of PTCT that we wish to focus on. It also allows us to treat the syntax of a theory on a par with its logical behaviour, and in a formalism in which the two can interact. The syntax is here incorporated into representation language itself. But it would also be possible to formulate a syntactic theory of natural language itself within the same framework, in a flexible way, that allows the syntactic properties of different languages to diverge as appropriate.

The intensionality of PTCT is fine-grained. It is effectively a form of inscriptional identity. But there is more to say about intensional difference than a distinction among terms in the object language.

We interpret the untyped λ-calculus (as characterised by Λ in the current presentation of PTCT) as providing a characterisation of the class of computable functions. This permits us to apply the distinction between operational and denotation semantics, as used in programming languages, to expressions in PTCT in a very direct way. We develop this view of intensionality in the following section.

3 An Operational Account of Fine-Grained Intensionality

PTCT allows us to sustain both the logical equivalence of (1a) and (1b), for example, and the non-equivalence of (2a) and (2b). The former are provably equivalent, but they correspond to non-identical propositional terms in PTCT.

The proof theory of PTCT induces a prelattice on the terms in π. In this prelattice the members of an equivalence class of mutually entailing propositional terms (terms that encode mutually entailing propositions) are non-identical, and so they correspond to distinct propositions.[13] While this result achieves the formal property of fine-grained intensionality, it does not, in itself, explain what intensional non-identity consists in, beyond the fact that two distinct expressions in the language of terms are identified with different intensions. This leaves us with what we can describe as a problem of ineffability. Intensional difference is posited as (a certain kind of) inscriptional distinctness, but this reduction does not offer a substantive explanation of the semantic properties that ground the distinction. Intensional difference remains ineffable.

This is an instance of a general problem with inscriptional treatments of fine-grained intensionality.[14] They identify differences of meaning with distinctions among terms in a semantic representation language. But without an account of how difference in terms generates intensional distinction, the inscriptional view leaves intensional non-identity unexplained. Inscriptionalist theories avoid problems created by the characterisation of intensions as functions on possible worlds at the risk of rendering intensions primitive to the point of inscrutability.

3.1 Expressing Intensional Difference Operationally

We can characterize the distinction between intensional identity and provable equivalence computationally by invoking the contrast between operational and denotational semantics in programming languages. Two simple examples illustrate this contrast.

For the first example take the function $predecessorSet(x)$, which maps an object in an ordered set (an element drawn from a set that has a successor or predecessor relationship) into the set of its predecessors. So, for example, if $x \in \{0,1,2,3,4,5\}$, $predecessorSet(x) = PredSet_x \subset \{0,1,2,3,4,5\}$ such that $\forall y_{\in PredSet_x}(y < x)$. It follows that $predecessorSet(0) = \emptyset$.

[13] Fox *et al.* (2002); Fox & Lappin (2005); Pollard (2008) construct higher-order hyperintensional semantic systems using an extended version of Church's SST and a prelattice of propositions in which the entailment relation is a preorder.

[14] See Fox & Lappin (2005) for a discussion of inscriptionalist theories of intentionality.

It is possible to define (at least) two variants of this function, $predecessorSet_a$ and $predecessorSet_b$, that are denotationally equivalent but operationally distinct. $predecessorSet_a$ is specified directly in terms of an immediate predecessor relation, while $predecessorSet_b$ depends upon a successor relation.

(5) a. $predecessorSet_a(x) = PredSet_x$, if
$\forall y(y \in PredSet_x \rightarrow predecessor(y, x))$.

b. $predecessor(y, x)$ if $predecessor_{immediate}(y, x)$;
$predecessor(y, x)$ if $predecessor_{immediate}(y, z)$ and $predecessor(z, x)$.

(6) a. $predecessorSet_b(x) = PredSet_x$, if
$\forall y(y \in PredSet_x \rightarrow successor(x, y))$.

b. $successor(y, x)$ if $successor_{immediate}(y, x)$;
$successor(y, x)$ if $successor_{immediate}(y, z)$ and $successor(z, x)$.

The second example we present here involves functions $g : \Sigma^* \rightarrow \{1, 0\}$ from Σ^*, the set of strings formed from the alphabet of a language, to the Boolean values 1 and 0, where $g(s) = 1$ if $s \in L$, and 0 otherwise. Let g_{csg1} be defined by the Definite Clause Grammar (DCG) in (7), and g_{csg2} by the DCG in (8).[15]

(7) $S \rightarrow [a],\ S(i).$
$S(I) \rightarrow [a],\ S(i(I)).$
$S(I) \rightarrow Bn(I),\ Cn(I).$
$Bn(i(I)) \rightarrow [b],\ Bn(I).$
$Bn(i) \rightarrow [b].$
$Cn(i(I)) \rightarrow [c],\ Cn(I).$
$Cn(i) \rightarrow [c].$

(8) $S \rightarrow A(I),\ B(I),\ C(I).$
$A(i) \rightarrow [a].$
$A(i(I)) \rightarrow [a],\ A(I).$
$B(i) \rightarrow [b].$
$B(i(I)) \rightarrow [b],\ B(I).$
$C(i) \rightarrow [c].$
$C(i(I)) \rightarrow [c],\ C(I).$

Both of these DCGs define the same context-sensitive language, $\{a^n b^n c^n \mid 1 \leq n\}$. This is the language whose strings consist of n occurrences of a, followed by n bs, and then n cs, with n equal to at least 1. The number of as, bs, and cs match in all strings. Each DCG uses a counting argument I for a non-terminal symbol to build up a stack of indices i that gives the successive number of occurrences of as, bs, and cs in a string. But the grammar in (7) counts from the bottom up, adding an i for each non-terminal that the recognizer encounters. By contrast the grammar in (8) imposes the requirement that the three stacks for the non-terminals A, B, and C be identical, and then it computes

[15] See Pereira & Shieber (1987) for an explanation of Definite Clause Grammars. The DCG in (7) is from Gazdar & Mellish (1989). Matthew Purver and Shalom Lappin constructed the DCG in (8) as a Prolog programming exercise for a computational linguistics course that Shalom Lappin gave in the Computer Science Department at King's College London in 2002.

the indices top down. The two grammars are computationally distinct, and using each of them to recognize a string can produce different sequences of operations, of different lengths and relative efficiency. Therefore, g_{csg1} and g_{csg2} are operationally distinct, but denotationally equivalent. They compute the same string set, but through different procedures.

3.2 Computable Functions and Natural Language Expressions

Given the distinction between denotational and operational meaning we can now interpret the non-identity of terms in the representation language as an operational difference in the functions that these terms express. But a class of such terms can still be provably equivalent in the sense that they yield the same values for the same arguments by virtue of the specifications of the functions that they correspond to. This provides a straightforward account of fine-grained intensionality in PTCT which avoids taking intensional difference as ineffable. This is consistent with the formulation of PTCT where intensions are represented by λ-terms, which can be interpreted as computable functions. In general, there is nothing to preclude other models of computation from being adopted for the same purpose.

It is reasonable to ask what it could mean to characterise the interpretation of a natural language expression as a computable function. Rich type theories, like PTCT and Type Theory with Records (TTR, Cooper, 2012), are based on the type systems used in programming languages. In some of these systems propositions are identified with proofs, where the proof of a proposition is the procedure applied to establish that it is assertable.[16] This can be a formal procedure, like the application of the rules of a proof theory, but it need not be. It could also be the sequence of operations involved in making the observations that support the application of a classifier predicate to an object or an event.

Although we have used examples of computable functions from set theory and formal grammar to illustrate our operational account of intensions, we assume that it will be possible to formulate computable functions for most predicative and modifying expressions in natural language. These functions will encode the procedures through which we determine the denotations of the terms. The success of the approach that we are proposing does, of course, depend upon the viability of this assumption, and we take this to be an empirical issue.

On the view proposed here we are taking the semantic content of terms in a natural language to be the functions that we use to compute the denotations of these expressions. If such a term is a predicate, then the function that corresponds to its meaning encodes the procedure through which we determine the values that it returns for its domain of arguments (n-tuples of arguments for relational predicates).

Two predicates may correspond to distinct functions that happen to yield the same values for each argument in a given domain, but they would diverge if defined for an alternative domain. This would be the situation for predicates that are contingently co-extensive. However, as we have seen in Section 3.1, it is possible for two (or more) distinct computable functions to be provably

[16] See Martin-Löf (1984) for a type theory in which propositions are characterised as proofs in a formal system.

equivalent. In this case they will generate the same range of values for all domains for which they are defined, through different sequences of operations, by virtue of the way in which these sequences are specified.

3.3 Two Alternative Operational Approaches

Muskens (2005) suggests a similar approach to hyperintensionality. He identifies the intension of an expression with an algorithm for determining its extension.[17] There are two major points of difference between Musken's theory and the one proposed here. First, he embeds his account in a logic programming approach, which he seems to take as integral to his explanation of hyperintensionality, while here the analysis is developed in a functional programming framework. This is, in fact, not an issue of principle. The same algorithm can be formulated in any programming language. So, for example, the definitions of $predecessorSet_a$ and $predecessorSet_b$ correspond to two Horn clause definitions in Prolog for variant predecessor predicates, `predecessorA(Y,X)` and `predecessorB(Y,X)`.

(9)
```
predecessorA(Y,X) :- predecessorImmediate(Y,X).
predecessorA(Z,X) :-
predecessorImmediate(Y,X),
predecessorA(Y,Z).
```

(10)
```
predecessorB(Y,X) :- successor(X,Y).
successor(X,Y) :- successorImmediate(X,Y).
successor(X,Z) :-
successorImmediate(X,Y),
successor(Y,Z).
```

Similarly, the DCGs in (7) and (8) that we used to define g_{csg1} and g_{csg2}, respectively, are (close to) Prolog executable code.

However, the functional programming formulation of the operational view of fine-grained intensionality follows straightforwardly from PTCT, where we can use the untyped λ-calculus to generate the intensional terms of the semantic representation language, and these encode computable functions. PTCT also offers rich Curry typing, supports polymorphism, and allows us to reason about truth and entailment, within an essentially first-order system.

The fact that it implies the operational account of intensional difference without further stipulation renders it attractive as a framework for developing computational treatments of natural language semantic properties.

The second, more substantive point of difference concerns the role of possible worlds in characterizing intensions. Muskens develops his hyperintensional semantics on the basis of Thomason (1980)'s Intentional Logic. In this logic Thomason proposes a domain of propositions as intensional objects, where the set of propositions is recursively defined with intensional connectives and quantifiers. He posits a homomorphism that maps propositions (and their

[17] Duží *et al.* (2010) also adopt an operational view of hyperintensionality within Tichý (1988)'s Transparent Intensional Logic (in the form of "constructions"). However, the computational details of their account are left largely unspecified. Both Muskens (2005) and Duží *et al.* (2010) regard their respective proposals as working out Frege (1892)'s idea that a sense is a rule for identifying the denotation of an expression.

constituents) to their extensions, and he constrains this homomorphism with several meaning postulates that restrict this mapping.[18] Muskens modifies and extends Thomason's logic by specifying a homomorphism between the intensional expressions of the logic and their extensions across the set of possible worlds. A proposition is mapped to the set of worlds in which it is true. As the homomorphism can be many-to-one, distinct propositions can receive the same truth-value across worlds.[19]

By contrast, PTCT adopts something akin to Thomason's possible worlds-free strategy of mapping propositions to truth-values. As we noted in Section 2, the judgement p True in effect makes a claim of truth for an otherwise intrinsically intensional expression p in π. In the original presentation of PTCT, this claim is achieved through an overt truth predicate that forms a wff from the term representation of a proposition. On this earlier account $\mathsf{T}(p)$ asserts the truth of the proposition that the term p in π represents. Therefore, like Thomason's Intentional Logic, PTCT de-modalizes intensions. This is a positive result. It is not clear why, on the fine-grained view, possible worlds must be essentially connected with the specification of intensions.

On both Musken's account and the one proposed here, the content of an intension is the set of computational operations through which it determines its denotational value, where these do not make essential reference to possible worlds. In the case of a proposition, the denotation that it determines is a truth-value, rather than a truth-value relative to a world. There may be independent epistemic, or even semantic reasons for incorporating possible worlds into one's general theory of interpretation, but worlds are not required for an adequate explanation of fine-grained intensionality. On the contrary, such an explanation must dispense with the original characterization of intensions as functions from worlds to extensions in order to explain the persistence of intensional difference beyond provable equivalence. Therefore, a radically possible worlds-free view of fine-grained intensionality offers the cleaner approach.

While theories of fine-grained intensionality may avoid the reduction of intensional identity to provable equivalence, many of them do not go beyond a bare inscriptionalist treatment of intensional difference. Therefore they leave this notion ineffable. On the proposal developed here intensional difference consists in the operational distinctions among computable functions, and extensional identity is the denotational equivalence of the values that functions compute. This account grounds fine-grained intensionality in a way that naturally accommodates cases of intensional difference combined with provable denotational equivalence.

Given that we can naturally interpret PTCT as employing the untyped λ-calculus to generate the Curry-typed term representations for the intensions of the language, and these terms encode computable functions, the proposed operational characterization of intensional difference is, in a sense, implicit in

[18] Fox & Lappin (2005) point out that Thomason's logic is problematic because it does not characterize the algebraic structure of the domain of propositions. It does not offer a proof theory that defines entailment for propositions, and so it leaves the relation between intentional identity and extensional equivalence crucially under determined.

[19] Fox *et al.* (2002); Fox & Lappin (2005); Pollard (2008) adopt a similar view for the fine-grained higher-order logics that they construct. They define worlds as untrafilters in the prelattice of propositions, and they take the truth of a proposition, relative to a world, to be its membership in such an ultrafilter. As entailment in the prelattice is defined by a preorder, distinct propositions can belong to the same set of ultrafilters.

this semantic framework.

This account yields a radically non-modal view of intensions in which possible worlds play no role in their specification or their interpretation. An intension is identified directly with the sequence of operations performed in computing the value of the function that expresses it. Fine-grained intensionality becomes the operational contents of computable functions.

Moschovakis (2006) proposes an operational treatment of meaning within the framework of the typed λ-calculus. He constructs a language L^{λ}_{ar} as an extension of Gallin (1975)'s Ty_2. He specifies acyclic recursive procedures for reducing the terms of L^{λ}_{ar} to unique canonical forms, and he identifies the meaning ("referential intension") of a term in this language with the "abstract algorithm" for computing its denotation.

There are two major points of difference between Moschovakis' algorithmic theory of intensions and the account proposed here. First, while in PTCT α, β, and η reduction sustain intensional identity, in L^{λ}_{ar} β reduction does not. His primary motivation for this move seems to be his concern to maintain the non-synonymy of sentences like "*John loves himself*" on one hand, and those like "*John loves John*" on the other. In L^{λ}_{ar} the canonical form of the former is $(\lambda(x)loves'(x,x))(j)$ where $j = john'$, while that of the latter is $loves'(j_1, j_2)$ where $j_1 = John', j_2 = John'$.[20]

But this issue would appear to be an artifact of the way that Moschovakis has chosen to formalize proper names and reflexive pronouns. If one represented them as distinct sorts of generalized quantifiers, or constants, then this problem would not arise. In any case, it is not a deep question of principle. We take α, β, and η reduction to support intensional identity because they are normalizing operations on λ-terms in the semantic representation language, and so they do not correspond, in any obvious way, to processes or relations of natural languages. However, it is perfectly possible to narrow the specification of intensional identity in PTCT to exclude β (as well as η, and even α) reduction, without altering the proposed account of intensions as computable functions. This would simply involve imposing a particularly fine-grained notion of intensional identity.

The second point of difference is more significant. Moschovakis specifies a Kripke frame semantics for L^{λ}_{ar} which is a variant of Montague's possible worlds models (he refers to them as "Carnap states"). These are n-tuples of indices corresponding to worlds, times, speakers, and other parameters of context. Intensions are characterized as algorithmic procedures for determining the denotation of a term relative to a world and the other elements of such an n-tuple. Therefore, like Muskens, Moschovakis' operational view of intensions treats them as inextricably bound up with possible worlds. The arguments brought against this view in Muskens' case apply with equal force here. An important advantage of the proposed account is that it factors modality and possible worlds out of the specification of intensions.

[20] We are using Moschovakis' notation here.

4 Conclusion

We have reprised Property Theory with Curry Typing (PTCT), reformulating it within Typed Predicate Logic (TPL). This gives an elegant presentation of the theory at a more abstract level. It avoids the need to invoke distinct levels of representation in the semantic analysis of natural language. Instead, we use a single language, TPL, to express the conditions of the entire theory, including its syntax, in place of a federation of separate but related languages for terms, types, and wffs.

We show how to construe the intensional language in computational terms, with intensional meaning corresponding to the operational semantics of computable functions, and extensions to their denotational semantics. Our account yields a general foundation for understanding the intensional–extensional contrast that avoids the ineffability of inscriptional theories by grounding this distinction in a central element in the semantics of programming languages.

References

Aczel, Peter (1980), Frege structures and the notions of proposition, truth and set, in Barwise, Keisler, & Keenan (eds.), *The Kleene Symposium*, North Holland, North Holland Studies in Logic, (31–39).

Carnap, Rudolf (1947), *Meaning and Necessity*, University of Chicago Press, Chicago.

Church, Alonzo (1940), A formulation of the Simple Theory of Types, *The Journal of Symbolic Logic* 5(2):56–68.

Cooper, Robin (2012), Type theory and semantics in flux, in Ruth Kempson, Tim Fernando, & Nicholas Asher (eds.), *Philosophy of Linguistics*, Elsevier, Amsterdam, (271–323).

Curry, Haskell B. & Robert Feys (1958), *Combinatory Logic*, volume 1 of *Studies in Logic*, North Holland.

Duží, Marie, Bjorn Jespersen, & Pavel Materna (2010), *Procedural Semantics for Hyperintensional Logic*, Springer, Dordrecht, New York.

Fox, Chris (2000), *The Ontology of Language*, CSLI Lecture Notes, CSLI, Stanford.

Fox, Chris & Shalom Lappin (2005), *Formal Foundations of Intensional Semantics*, Blackwell, Oxford.

Fox, Chris & Shalom Lappin (2010), Expressiveness and complexity in underspecified semantics, *Linguistic Analysis* 36:385–417.

Fox, Chris, Shalom Lappin, & Carl Pollard (2002), A higher-order, fine-grained logic for intensional semantics, in G. Alberti, K. Balough, & P. Dekker (eds.), *Proceedings of the Seventh Symposium for Logic and Language*, Pecs, Hungary, (37–46).

Frege, Gottlob (1892), On sense and reference, in Peter Geach & Max Black (eds.), *Translations from the Philosophical Writings of Gottlob Frege, 3rd Edition*, Basil Blackwell, Oxford, (56–78).

Gallin, Daniel (1975), *Intensional and Higher-Order Modal Logic*, North-Holland, Amsterdam.

Gazdar, Gerald & Christopher S. Mellish (1989), *Natural Language Processing in Prolog*, Addison-Wesley, Waltham, MA.

Lappin, Shalom (2012), An operational approach to fine-grained intensionality, in Thomas Graf, Denis Paperno, Anna Szabolcsi, & Jos Tellings (eds.), *Theories of Everything: In Honor of Ed Keenan*, UCLA Working Papers in Linguistics 17, (Creative Commons Non-Commercial License (`http://creativecommons.org/licenses/by-nc/3.0/`)).

Lappin, Shalom (2013), Intensions as computable functions, *Linguistics Issues in Language Technology* 9:1–12.

Martin-Löf, Per (1984), *Intuitionistic Type Theory*, Studies in Proof Theory (Lecture Notes), Bibliopolis, Napoli.

Montague, Richard (1974), *Formal Philosophy: Selected Papers of Richard Montague*, Yale University Press, New Haven, CT/London, UK, edited with an introduction by R. H. Thomason.

Moschovakis, Yiannis N. (2006), A logical calculus of meaning and synonymy, *Linguistics and Philosophy* 29:27–89.

Muskens, Reinhard A. (2005), Sense and the computation of reference, *Linguistics and Philosophy* 28:473–504.

Pereira, Fernando C. N. & Stuart M. Shieber (1987), *Prolog and Natural-Language Analysis*, volume 10 of *CSLI Lecture Notes Series*, Center for the Study of Language and Information.

Pollard, Carl (2008), Hyperintensions, *Journal of Logic and Computation* 18:257–282.

Thomason, Richmond H. (1980), A modeltheory for propositional attitudes, *Linguistics and Philosophy* 4:47–70.

Tichý, Pavel (1988), *The Foundations of Frege's Logic*, De Gruyter, Berlin.

Turner, Raymond (1992), Properties, propositions and semantic theory, in M. Rosner & R. Johnson (eds.), *Computational Linguistics and Formal Semantics*, Cambridge University Press, Cambridge, Studies in Natural Language Processing, (159–180).

Turner, Raymond (2005), Semantics and stratification, *Journal of Logic and Computation* 15(2):145–158.

Turner, Raymond (2008), Computable models, *Journal of Logic and Computation* 18(2):283–318.

Turner, Raymond (2009), *Computable Models*, Springer-Verlag, London.