HTLC:
Hyperintensional Typed Lambda Calculus

Michal Fait
Department of Computer Science
VSB-Technical University of Ostrava,
Czech Republic
michal.fait@vsb.cz

Giuseppe Primiero
Department of Philosophy
University of Milan, Italy
giuseppe.primiero@unimi.it

Abstract

In this paper we introduce the logic HTLC, for Hyperintensional Typed Lambda Calculus. The system extends the typed λ-calculus with hyperintensions and related rules. The polymorphic nature of the system allows to reason with expressions for extensional, intensional and hyperintentsional entities. We inspect meta-theoretical properties and show that HTLC is complete in Henkin's sense under a weakening of the cardinality constraint for the domain of hyperintensions.

This collaboration was possible thanks to the EU project "Science without borders" No. CZ.02.2.69/0.0/0.0/16_027/0008463. The first author was funded by the Grant Agency of the Czech Republic (GACR) project GA18-23891S "Hyperintensional Reasoning over Natural Language Text", by the internal grant agency of VSB-Technical University of Ostrava, project No. SP2019/40, "Application of Formal Methods in Knowledge Modelling and Software Engineering I", and by the Moravian-Silesian regional program No. RRC/10/2017 "Support of science and research in Moravian-Silesian region 2017". The second author was supported by the Department of Philosophy "Piero Martinetti" of the University of Milan under the project "Departments of Excellence 2018-2022" awarded by the Ministry of Education, University and Research (MIUR). The authors wish to thank two anonymous reviewers for valuable comments on previous versions of this paper which have helped improving its presentation.

1 Introduction

The literature in philosophical [11] and computational logic [18] has increasingly been paying attention to the crucial distinction between reasoning about extensional (functional values, like individuals or truth-values); intensional (functions); and hyperintensional entities (abstract procedures, [6]; linguistic expressions, [18, 19, 10, 13]; proofs, [9, 8, 17, 21, 22]; or computations [3, 2]), including the dynamics of hyperintensions, [1, 15].

The encapsulation of extensional, intensional and hyperintensional layers of reasoning in one logical system has been offered by Transparent Intensional Logic (TIL), a hyperintensional, partial typed λ-calculus [6]: hyperintensional, because the meaning of TIL λ-terms are procedures producing functions rather than the denoted functions themselves; partial, because TIL is a logic of partial functions; and typed, because all the entities of TIL ontology receive a type. TIL is endowed with a procedural semantics which explicates the meanings of language expressions as abstract procedures encoded by the expressions. TIL is powerful in accounting for contexts and their relations, especially for some natural language phenomena like partial denotations and modal modification, see e.g. [12]. But although it is technically an extension of typed λ-calculus, it still misses a well-defined and agreed upon proof theory. Because of this, it is not possible to reflect its semantics in properties of derivations. A full system providing a proof-theoretic semantics to reason about all these types of entities (extensional, intensional, hyperintensional) seems still to be missing in the literature. In this paper, our goal is to provide an inferential engine common to extensions, intensions and hyperintensions, able to express their relations as well. Previous attempts in this direction are: either impure λ-calculi, because they attempt to capture all of TIL, [16]; or ND-systems for TIL, which do not offer a rule based semantics, see e.g. [7]. Our approach is limited compared to the semantics of TIL, because we only aim at expressing what can be formulated in standard proof-theoretic terms.

The system HTLC presented in this paper is an extension of a typed λ-calculus with hyperintensions. Expressions in this language are of the form $\Gamma \vdash t : T$ where, as usual in typed λ-calculus: Γ is the context of assumptions; t is a term and T is its type. Types express the extensional, intensional or hyperintensional nature of terms. Hence, terms of HTLC denote, as usual in λ-calculus, functions from set to set and their values, with the added hyperintensional terms denoting procedures or computations which we call *constructions*; the output produced by a construction is called its *product*. Hyperintensional terms are defined proof-theoretically by introduction and elimination rules, thereby extending a standard typed λ-calculus, see Figure 2. The Trivialization rule works as an introduction rule: given an extensional or intensional term t, Trivialization returns a hyperintensional term t^*, whose denotation is a construction. The Execution rule works as an elimination: given a hyperintensional term t^*, Execution returns the term t denoting the product of the corresponding construction.[1] Execution eventually provides a non-hyperintensional term as an output, in which case we say that the construction denoted by the hyperintensional term *produces* the object denoted by the non-hyperintensional one. We also obtain higher-order hyperintensional terms by Trivialization on a term denoting a construction; Execution applied on a term denoting a higher-order construction results in a term denoting a lower-order construction, until it is applied to a term denoting a construction, producing an extensional or intensional object.

To offer a very basic example, consider the number 2, which in our system is a term of type $\mathbb{N}$. Many different functional expressions may denote this number, for example [+ 1 1] and [− 5 3] are two of those. Each of those terms denoting the number 2 may have different constructions, or hyperintensions. For example, hyperintensional terms having [+ 1 1] as their product are: $Integer.sum(1, 1)$ where the operation at hand is the Java command for addition, or $S(S(0) + 0)$ which is the recursive equation presenting the number 2 with addition as successor. When moving from the term [+ 1 1] to the corresponding hyperintensional level, any of those two terms could be obtained; we will use in our language the expression $[+\ 1\ 1]^*$ to denote *any* hyperintensional term producing [+ 1 1]. When moving back to the functional level, the term [+ 1 1] should always be produced, together with its denotation the number 2. Hyperintensional terms having [− 5 3] as their products could be $Integer.minus(5, 3)$ and $S(S(S(S(S(0))))) - S(S(S(0)))$, where we assume subtraction can be redefined as a predecessor function. Each of those terms produces [− 5 3] and this in turn denotes 2. We will use in our language the expression $[-\ 5\ 3]^*$ to stand for any linguistic expression (written for instance in some programming language) denoting the operation [− 5 3]. The former is thus an hyperintension for the latter. Again, when moving from the hyperintensional level to the functional level, a unique functional term should be obtained. This functional term denotes only one entity, its hyperintensional counterpart denotes instead different constructions.

Although partly inspired by TIL, and reflecting some of its terminology, our approach differs in several aspects. First, we use a bottom-up approach: we start from well-typed extensional and intensional terms and define hyperintensional ones from them. Because of this strategy, there is always a term obtained by an instance of the Execution rule. In other words, our system does not allow the derivation of improper constructions, i.e. hyperintensions that do not produce any object. TIL cases of improperness caused by execution of non-constructions are avoided in our

[1]We borrow the names for these rules from TIL.

system by requiring that only trivialized canonical terms can be executed.[2] Second, another source of improperness in TIL is composition applied to partial functions, i.e. when application may fail for some specific argument: we deal only with total functions, thus function application always returns an output. Third, Composition in TIL can fail also if the types of arguments do not match with the type required by the bound variables in the body of the expression: in our system, type checking will forbid the rule application in such cases. Fourth, in our calculus the product of a construction is obtained by explicit application of Execution, i.e. it is never implicitly denoted. Fifth, our semantics does not use quantification over worlds and times. Finally, to show the logical and analytical (i.e. under term decomposition) equivalence of terms denoting constructions, our only means is to perform Execution and check identity by reduction on the terms denoting the corresponding products. In the following, to aid readability, we will sometimes avoid referring to the terms of the language as denoting constructions, functions, numbers etc., and we will conventionally refer to their denotations: hence we might say that in a derivation both intensions and hyperintensions occur, or that a function of constructions occurs, while technically we intend that terms denoting such objects occur in the derivation.[3]

The rest of this paper is structured as follows. In Section 2 we present the language of HTLC . We define first the polymorphic set of rules which technically reduces to a typing system à la Curry, i.e. where types are assigned to pure λ-terms; for HTLC, this means that the same rules set can be instantiated not only by extensional and intensional terms, but also by hyperintensional ones. We then formulate those rules for each of the relevant types, offering thereby the extensional, intensional and hyperintensional fragments. We offer examples of derivations with terms of different types, and in particular in Section 3 we formulate an example where the same expression is treated first at the extensional/intensional level, and then at the hyperintensional level. In Section 4 we provide the meta-theoretical results, including the definition of term occurrence, normalization across contexts and completeness with respect to a Henkin's style of general model. Finally we provide some conclusions and ideas for possible further extensions of our work.

[2]The restriction on the canonicity of trivialized terms to be executed is close in spirit to the constraint imposed by Martin-Löf on the application of β-reduction w.r.t the terms of his theory of types, according to which a λ-term has to be β-reduced only "from without" and not "from within", i.e. guaranteeing that the way in which β-reduction is performed eventually coincides with the evaluation of closed λ-terms, [14, p.160]. We owe this observation to one of the reviewers.

[3]For clarity: we use the expression "function of constructions" (or of functions, or of anything else) to indicate a function that takes constructions (or functions, or anything else) as input and is allowed to be heterogeneous, thus having something else as codomain.

2 The System HTLC

The syntax of HTLC is a typed λ-calculus extended with a type for hyperintensional terms.

Definition 1 (Grammar).

$$T ::= \alpha \mid (T\ T_1 \ldots T_n) \mid *^T$$

$$\alpha ::= o \mid \iota \mid \tau \mid \ldots$$

$$t ::= x_i \mid [\lambda x_1 \ldots x_n.t] \mid [t\ t_1 \ldots t_n] \mid t^*$$

The type syntax for T is inductively defined by three kinds of entities:

- Extensional entities of type α;
- Intensional entities of type $(T\,T_1 \ldots T_n)$;
- Hyperintensional entities of type $*^T$.

The set of atomic types can depend on the application, including o (set of truth values: T, F), ι (infinite set of individuals, members of the universe), τ (as a meta-variable type for numbers: $\mathbb{N}, \mathbb{R}$ - e.g. sets of natural and real numbers respectively which might be added and should be defined through appropriate rules), and so on. Functions will be defined accordingly, as mappings from the Cartesian product of types $(T_1 \times \cdots \times T_n)$ into the type T, for any arbitrary type (hence involving sets of individuals, of truth values, of numbers and so on). We constraint these to total functions. We simplify the arrow notation of multi-argument functions $(T_1 \to \cdots \to T_n \to T)$ with the pair notation $(((T\ T_n) \ldots) T_1)$. As in standard typed λ-calculi we use association to the left when dealing with function types, so the curried version can be rewritten as $(T\ T_n\ \ldots\ T_1)$.[4] We can build higher order functions that take functions as arguments and return a function as value. Terms typed as hyperintensions denote abstract procedures whose outputs can be of any type (including hyperintensions as we admit higher-order constructions); these entities are constructions, computations or different "senses" in which lower order constructions, extensional or intensional entities can be produced.

[4]The notation $(T\ T_1\ \ldots\ T_n)$ from Definition 1 and the notation $(T\ T_n\ \ldots\ T_1)$ are equivalent: the second one can be obtained from the first one by a simple and harmless renumbering of arbitrary types T_x, and vice versa.

Terms of the language have the following form: variables x_i; abstraction terms $[\lambda x_1 \dots x_n.t]$ denoting functions; application terms $[t\ t_1 \dots t_n]$, denoting values of functions on given arguments; and finally hyperintensional terms t^*, recursively constructed and denoting constructions. We call a formula of our language an expression $t : T$, which declares a term t to belong to a given type T; as usual in the literature, a closed formula is one whose term does not contain any free variables, i.e. only λ-bound ones; a list of assumptions of formulae $\{x_1 : T_1, \dots, x_n : T_n\}$ is called a context; a judgement is the assertion of a formula under a given context, denoted as $x_1{:}T_1, \dots, x_n{:}T_n \vdash t : T$.

Rules of the system HTLC are given below in four parts. First, we describe the polymorphic fragment of our system with rules applicable to arbitrary terms of any type. Then we focus on the extensional fragment, where types are extensional values, especially truth values and functions defined on them; then we move up to the intensional fragment, where objects of interest are functions of basic types, and functions of higher order; finally, we present the hyperintensional fragment with constructions and functions of constructions.

2.1 The Polymorphic Fragment

Rules for terms of arbitrary types are inference rules of the standard typed λ-calculus (see Figure 1) extended with rules to define the meaning of hyperintensional terms (see Figure 2): the latter are called intra-context rules as they allow to move reasoning from the extensional and intensional contexts to the hyperintensional one, and back. Rules are applied on formulas of the form $t : T$, but we often say that a rule is applied on a term t of type T, or that a rule returns such a term.

The Assumption rule allows for the derivation of a typed variable from its own assumption. The Abstraction rule allows to construct a λ-term for a function type $(T\ T_1 \dots T_n)$ from the corresponding judgement inferring a term $t : T$ from variables $x_1, \dots, x_n$ having types $T_1, \dots, T_n$. The Application rule creates a term of type T denoting the value of a function, expressed by a term t on the n-tuple of arguments expressed by terms $t_1, \dots, t_n$.

While inferring terms denoting intensional entities is guaranteed by Abstraction, and their elimination is the result of Application, appropriate rules are given in Figure 2 to shift to and from terms denoting hyperintensional entities. In line with proof-theoretic semantics, we provide the meaning of hyperintensional terms by defining an introduction and an elimination rule. The former establishes the necessary conditions for the formulation of a construction; the latter provides the minimal consequences of its use. The rules must be considered in pairs: a detour Trivialization/Execution is well-behaving (i.e. harmonious) if given a term t of type

$$\frac{}{x_i : T \vdash x_i : T}\text{ Assumption}$$

$$\frac{\Gamma, x_1 : T_1, \ldots, x_n : T_n \vdash t : T}{\Gamma \vdash [\lambda x_1 \ldots x_n .t] : (T\ T_1 \ldots T_n)}\text{ Abstraction}$$

$$\frac{\Gamma \vdash t : (T\ T_1 \ldots T_n) \qquad \Gamma_1 \vdash t_1 : T_1 \ldots \Gamma_n \vdash t_n : T_n}{\Gamma, \Gamma_1, \ldots \Gamma_n \vdash [t\ t_1 \ldots t_n] : T}\text{ Application}$$

Figure 1: HTLC: Polymorphic Rules System

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash\ t^* : *^T}\text{ Trivialization}$$

$$\frac{\Gamma \vdash t^* : *^T}{\Gamma \vdash t : T}\text{ Execution}$$

Figure 2: HTLC: Intra-context Rules

T inducing a term t^* of type $*^T$ by Trivialization, an instance of the Execution rule applied to the latter will return the former. The rules are formulated for a general term, and their version for complex terms is explained below.

The Trivialization rule defines the process of going from a given term t to the hyperintensional term t^* which denotes a construction of the object denoted by t. When the trivialised term t is of type α, Trivialization allows to shift from a term denoting an extensional entity to a hyperintensional one producing it. When the trivialised term t is of type $(T\ T_1 \ldots T_n)$, Trivialization allows to shift from a term denoting an intensional entity to a hyperintensional one producing it. When the trivialised term is of type $*^T$, Trivialization results in a higher-order hyperintensional entity denoted by t^{*^*}, which produces still a (lower-order) hyperintensional entity. In this latter case we will iterate on the type: Trivialization on a term t^* of type $*^T$ returns a formula $t^{*^*} : *^{*^T}$. By convention and to simplify notation, in the following we do not iterate * on terms that are trivialized and were already of type $*^T$, but we agree just to rename the term; the relevant information on the iterated Trivialization is carried by the type. This allows to introduce the notion of order of the hyperintensional type:

$$\cfrac{\begin{array}{c}\vdots\\ \vdash \mathit{Compute} : (o\iota *^{\tau})\end{array} \qquad \begin{array}{c}\vdots\\ \vdash \mathit{Michal} : \iota\end{array} \qquad \cfrac{\cfrac{\begin{array}{c}\vdots\\ \vdash \mathit{Plus} : (\tau\tau\tau)\end{array} \qquad \begin{array}{c}\vdots\\ \vdash 5 : \tau\end{array} \qquad \begin{array}{c}\vdots\\ \vdash 1 : \tau\end{array}}{\vdash [\mathit{Plus}\,5\,1] : \tau}\,\text{Application}}{\vdash [\mathit{Plus}\,5\,1]^* : *^{\tau}}\,\text{Trivialization}}{\cfrac{\vdash [\mathit{Compute}\,\mathit{Michal}\,[\mathit{Plus}\,5\,1]^*] : o}{\vdash [\mathit{Compute}\,\mathit{Michal}\,[\mathit{Plus}\,5\,1]^*]^* : *^{o}}\,\text{Trivialization}}\,\text{Application}$$

Figure 3: HTLC: Trivialization example

Definition 2 (Hyperintensional type of order n)**.** *We say that a term t^* is of a hyperintensional type of order n, if and only if t^* results from n instances of the Trivialization rule. We denote the formula containing such term as*

$$t^* : *^{*_1^{\iddots^{*_{n-1}^{T}}}}$$

where T is either an extensional or intensional type.

In all the cases above, the necessary condition in order for a term of type $*^T$ to be properly typed is that the term to be trivialized be a properly typed term of type T. This, in turn, means that Trivialization is always defined in its argument, and therefore we do not allow improper constructions. As a result, the trivialized term can always be returned (by Execution, see below). An example of the use of the Trivialization rule is illustrated in Figure 3. To aid readability, in this example we keep empty the contexts on the left-hand side of all formulas: the *Compute* relation takes by Application an individual as first argument, and the construction of a function to add numbers 5 and 1 as second argument; the latter is obtained by Trivialization on the functional term whose denotation is the object “6”; the Application returns a truth-value; the proposition “Michal computes the sum of 5 and 1” is then Trivialized in the last step of the derivation.

Execution is the opposite process of going from a hyperintensional type (eventually of higher order) to an extensional or intensional one (or to a hyperintensional type of lower order). Given a canonical term t^* of type $*^{\alpha}$, Execution returns the term t denoting the product of this construction, that is a term of type α. When the term t^* to be executed is of type $*^{(T\ T_1 \dots T_n)}$, Execution returns the term t that denotes an intensional entity of type $(T\ T_1 \dots T_n)$. Given a term t^* of hyperintensional type of order $n+1$, Execution returns a term t^* of hyperintensional type of order n.[5] The condition on the canonicity of the term which is executed allows a

[5]It might be noted that our Execution rule has a stronger requirement than what is typical for

general formulation of the rule under a non-empty context, required not to include variables occurring in that term. On the other hand, this also means that a term to be executed might require first additional steps according to the computational rules of the system (see Figure 6) when not in canonical form. Hence, by closure under Abstraction of trivialized terms, there can be also a term of the form $[\lambda x^*.t^*]$ and type $(*^T\ *^{T_1})$: in this case, Execution is obtained by the following detour:

$$\dfrac{\vdash [\lambda x_1^*.t^*] : (*^T\ *^{T_1})}{\vdash [\lambda x_1.t] : (T\ T_1)}\,\text{Execution} \qquad \rightsquigarrow$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x_1 : T_1 \vdash x_1 : T_1}\,\text{Assum.}}{x_1 : T_1 \vdash x_1^* : *^{T_1}}\,\text{Triv.} \qquad \vdash [\lambda x_1^*.t^*] : (*^T\ *^{T_1})}{x_1 : T_1 \vdash [[\lambda x_1^*.t^*]\, x_1^*] : *^T}\,\text{App.} \qquad [[\lambda x_1^*.t^*]\, x_1^*] \to_\beta t^*[x_1^*/x_1^*] \equiv t^* : *^T}{x_1 : T_1 \vdash t^* : *^T}\,\text{Sub.Red.}}{x_1 : T_1 \vdash t : T}\,\text{Execution}}{\vdash [\lambda x_1.t] : (T\ T_1)}\,\text{Abstraction}$$

In this tree we start from trivializing an assumed variable to type $*^{T_1}$, to which we apply our λ-term. We then have a subject reduction step where $t^*[x_1^*/x_1^*]$ is syntactically equivalent to t^*, execute this closed term,[6] and abstract to obtain our desired (now executed) λ-term. In the following we always abbreviate this detour by direct Execution on each trivialized component of a λ-term and assume the computational step to obtain a closed term is always performed before execution.

By closure under Application of trivialized terms, there can be a term of the form $[[\lambda x_1^*.t^*]\, t_1^*]$ of type $*^T$: in this case, Execution returns products for each of the composing terms, combining the previous reduction with one additional available premise:

$$\dfrac{\dfrac{\vdash [\lambda x_1^*.t^*] : *^T *^{T_1} \qquad \vdash t_1^* : *^{T_1}}{\vdash [[\lambda x_1^*.t^*]\, t_1^*] : *^T}\,\text{Application}}{\vdash [[\lambda x_1.t]\, t_1] : T}\,\text{Execution}$$

$$\rightsquigarrow$$

proof-theoretic semantics, namely that it applies to canonical terms. Typically, an elimination rule is applicable to arbitrary terms of the required type and, as a result, a selector s is applied to this term. Then, if the term obtained is a redex (i.e. the selector is applied on a constructor), we can apply reduction. Our rule requires the term already in canonical form (namely to be built with constructor $*$), and we consider the reduction step as already performed and hidden, in order to avoid executing improper constructions.

[6]Note that t^* in this expression does not actually depend on x_1 in the context. Despite the fact that the variable x_1^* appearing in t^* has been obtained from x_1 by the application of a Trivialization rule, these two variables have to be taken as different, since in the derivation they are associated with two different types: the variable x_1 is associated with the type T_1, while the variable x_1^* is associated with the type $*^{T_1}$.

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash \mathit{Loves}^* : *^{(o\iota\iota)}\end{array}}{\Gamma \vdash \mathit{Loves} : (o\iota\iota)}\text{Exec.} \qquad \dfrac{\begin{array}{c}\vdots\\ \Gamma_1 \vdash \mathit{John}^* : *^{\iota}\end{array}}{\Gamma_1 \vdash \mathit{John} : \iota}\text{Exec.} \qquad \dfrac{\begin{array}{c}\vdots\\ \Gamma_2 \vdash \mathit{Mary}^* : *^{\iota}\end{array}}{\Gamma_2 \vdash \mathit{Mary} : \iota}\text{Exec.}}{\Gamma, \Gamma_1, \Gamma_2 \vdash [\mathit{Loves}\ \mathit{John}\ \mathit{Mary}] : o}\text{App.}}{\Gamma, \Gamma_1, \Gamma_2 \vdash [\mathit{Loves}\ \mathit{John}\ \mathit{Mary}]^* : *^{o}}\text{Triv.}$$

Figure 4: HTLC: Execution example

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma, x^* : *^{\iota}, y^* : *^{\iota} \vdash \mathit{Loves}^* : *^{o}\end{array}}{\Gamma \vdash [\lambda x^*\, y^*.\mathit{Loves}^*] : (*^{o} *^{\iota} *^{\iota})}\text{Abs.} \qquad \dfrac{\begin{array}{c}\vdots\\ \Gamma_1 \vdash \mathit{John} : \iota\end{array}}{\Gamma_1 \vdash \mathit{John}^* : *^{\iota}}\text{Triv.} \qquad \dfrac{\begin{array}{c}\vdots\\ \Gamma_2 \vdash \mathit{Mary} : \iota\end{array}}{\Gamma_2 \vdash \mathit{Mary}^* : *^{\iota}}\text{Triv.}}{\Gamma, \Gamma_1, \Gamma_2 \vdash [[\lambda x^*\, y^*.\mathit{Loves}^*]\ \mathit{John}^*\ \mathit{Mary}^*] : *^{o}}\text{App.}}{\Gamma, \Gamma_1, \Gamma_2 \vdash [[\lambda x\, y.\mathit{Loves}]\ \mathit{John}\ \mathit{Mary}] : o}\text{Exec.}$$

Figure 5: HTLC: Second Execution example

$$\dfrac{\dfrac{\vdash [\lambda x_1^*.t^*] : *^{T} *^{T_1}}{\vdash [\lambda x_1.t] : T\ T_1}\text{Execution} \qquad \dfrac{\vdash t_1^* : *^{T_1}}{\vdash t_1 : T_1}\text{Execution}}{\vdash [[\lambda x_1.t]\, t_1] : T}\text{Application}$$

Note that also in this case we require the application to be done on a closed term. Again, we always abbreviate this derivation by direct Execution on each trivialized component of an applied term.

An example of the use of Execution is depicted in Figure 4: we first derive a construction of the relation “loves” between two individuals (e.g. its linguistic sense which appears on this very page between the written names of those individuals); the hyperintensional term denoting the function “loves” as well as its arguments are all executed for the Application to be well-typed and to obtain the propositional content “John loves Mary”, whose type is a truth value; finally we can apply Trivialization back to obtain a construction of such propositional content, of type $*^{o}$. In the second example from Figure 5, the Execution of a construction is obtained by Application of trivialised terms, where each component of this application is a construction. In this case, Execution is required to act on all subterms according to the detour presented above, bringing each term to its product: the formula $[\lambda x^*\, y^*.\mathit{Loves}^*] : (*^{o} *^{\iota} *^{\iota})$ is executed to obtain the formula $[\lambda x\, y.\mathit{Loves}] : (o\ \iota\ \iota)$.

As an extension of the lambda calculus, in HTLC β-reduction is present in the

$$\Gamma \vdash [[\lambda x_1 \ldots x_n .t]\ t_1 \ldots t_n] \rightarrow_\beta t[x_1/t_1 \ldots x_n/t_n] : T$$

$$\frac{\Gamma, x_1 : T_1, \ldots, x_n : T_n \vdash t \rightarrow_\beta t' : T}{\Gamma \vdash [\lambda x_1 \ldots x_n .t] \rightarrow_\beta [\lambda x_1 \ldots x_n .t'] : (T\ T_1 \ldots T_n)}\ \beta\text{-Abstr}$$

$$\frac{\Gamma \vdash t \rightarrow_\beta t' : (T\ T_1 \ldots T_n) \qquad \Gamma_1 \vdash t_1 : T_1 \ldots \Gamma_n \vdash t_n : T_n}{\Gamma, \Gamma_1 \ldots \Gamma_n \vdash [t\ t_1 \ldots t_n] \rightarrow_\beta [t'\ t_1 \ldots t_n] : T}\ \beta\text{-App}$$

$$\frac{\Gamma \vdash t : (T\ T_1 \ldots T_i \ldots T_n) \qquad \Gamma_1 \vdash t_1 : T_1 \ldots \Gamma_n \vdash t_n : T_n \qquad \Gamma_i \vdash t_i \rightarrow_\beta t_i' : T_i}{\Gamma, \Gamma_1 \ldots \Gamma_n \vdash [t\ t_1 \ldots t_i \ldots t_n] \rightarrow_\beta [t\ t_1 \ldots t_i' \ldots t_n] : T}\ \beta\text{-App}$$

Figure 6: HTLC: β-rules

$$\frac{\Gamma \vdash t : T \qquad t \twoheadrightarrow_\beta t'}{\Gamma \vdash t' : T}$$

Figure 7: HTLC: Subject reduction

form of a computational step, i.e. it expresses a purely syntactic term transformation to go from a syntactically more complex to a reduced term. We present such computation step applied to each of the possible rules, see Figure 6: β-reduction on a lambda term $[[\lambda x_1 \ldots x_n.t]\, t_1 \ldots t_n]$ corresponds to the substitution of the terms $t_1 \ldots t_n$ for variables $x_1 \ldots x_n$ occurring inside the term t; it is closed under the rules for Abstraction and Application; it is moreover a type-preserving operation when we take its transitive and reflexive closure $\twoheadrightarrow_\beta$, a fact which can be formulated as a simple rule known as Subject reduction and illustrated in Figure 7.

The last set of rules for the polymorphic fragment reflects the structural behaviour of the system, see Figure 8: Weakening expresses the usual principle that context extension is a monotonic operation in view of derivable terms; Exchange reflects the unstructured nature of contexts; Contraction allows to merge two variables of the same type occurring in the same context (this latter operation is expressed in terms of variable substitution inside the derivable term).

$$\frac{\Gamma \vdash t : T}{\Gamma, x : T_1 \vdash t : T}\ \text{Weakening}$$

$$\frac{\Gamma, x_1 : T_1, x_2 : T_2 \vdash t : T}{\Gamma, x_2 : T_2, x_1 : T_1 \vdash t : T}\ \text{Exchange}$$

$$\frac{\Gamma, x_1 : T_1, x_2 : T_1 \vdash t : T}{\Gamma, x_3 : T_1 \vdash t[x_1/x_3; x_2/x_3] : T}\ \text{Contraction}$$

Figure 8: HTLC: Structural rules

2.2 The Extensional Fragment

In the extensional implementation of the HTLC rules, we reason with atomic types and functions defined over them. By creation of a function from atomic types, we move from a term occurring extensionally to a term occurring intensionally; and viceversa, by application of a function on an argument, we move from a term occurring intensionally to a term occurring extensionally.[7]

A first obvious interpretation for atomic types is by propositional terms with truth-values o as types, and functions defined on them. Rules of this fragment are illustrated in Figure 9. The system can be extended with connectives for conjunction and disjunction by adding pairs of propositions and projection on pairs respectively for appropriate introduction and elimination rules. A second possible interpretation of the extensional fragment is given by considering computational terms (programs) and the type of their outputs.

2.3 The Intensional Fragment

In the intensional fragment, we are able to reason about functions and higher order functions, see Figure 10. Beginning with functions of atomic types (i.e. functions $(T\ T_1 \dots T_n)$, where T and the T_i are all atomic types and hence are considered of order one), we can create functions defined over them (functions of higher order). Functions of hyperintensions can also be obtained by the Abstraction rule in the hyperintensional fragment (see Section 2.4). When reasoning with functions, we work with terms occurring intensionally; and when applying functions, we generate terms that occur either extensionally (if one deals with a function of atomic types), or intensionally themselves (if one deals with a higher-order function), or hyperinten-

[7]See Section 4.1 for an appropriate definition of term occurrence.

Assumption

$$\frac{}{x_i : o \vdash x_i : o}$$

Conditional proof (CP)

$$\frac{\Gamma, x_i : o \vdash t : o}{\Gamma \vdash \lambda x_i \,.\, t : (o\ o)}$$

Modus ponendo ponens (MPP)

$$\frac{\Gamma \vdash \lambda x_i \,.\, t : (o\ o) \qquad \Gamma_1 \vdash t_i : o}{\Gamma, \Gamma_1 \vdash t : o}$$

Figure 9: HTLC: Extensional fragment - propositional version

sionally (if one deals with a function of constructions). Function evaluation occurs therefore within the extensional fragment when we are reasoning with functions of atomic types; it occurs within the intensional fragment when working with higher-order functions; and it occurs within the hyperintensional fragment when working with functions of constructions.

An example of a higher order function is *Map* : $((o\ T_2)\,(T_2\ T_1)\,(o\ T_1))$, which takes two arguments of the function type, and it returns an object of a function type. In functional programming languages *Map* is used to apply a function to a list and return another list. In order to replace lists, whose type we do not have explicitly in our language, we can give up on ordering and define a set of type T by using a characteristic function of type $(o\ T)$. For the Map function, consider a set expressed by its characteristic function $(o\ T_1)$, and a function $(T_2\ T_1)$ applied to every element of the input set, to obtain an output set of type $(o\ T_2)$, again as the characteristic function of a set. For example, consider a term *Square* of type $(\mathbb{N}\ \mathbb{N})$ denoting the function that transforms any natural number into its square; and consider the term *Primes* of type $(o\ \mathbb{N})$ denoting the characteristic function that selects the subset of prime numbers from $\mathbb{N}$. Then we can think of [*Map Square Primes*] : $(o\mathbb{N})$ as the application of *Square* on all members of *Primes*. The result is a new set containing the squares of prime numbers. In this particular example, the typing of our map function is $Map : ((o\ \mathbb{N})\,(\mathbb{N}\ \mathbb{N})\,(o\ \mathbb{N}))$.

$$\frac{}{x_i : (T\ T_1 \dots T_n) \vdash x_i : (T\ T_1 \dots T_n)}\ \text{Assumption}$$

$$\frac{\Gamma, x_1 : F_1, \dots, x_n : F_n \vdash t : F}{\Gamma \vdash [\lambda x_1 \dots x_n \,.\, t] : (F\, F_1\, \dots\, F_n)}\ \text{Abstraction}$$

$$\frac{\Gamma \vdash t : (F\, F_1\, \dots\, F_n) \qquad \Gamma_1 \vdash t_1 : F_1 \dots \Gamma_n \vdash t_n : F_n}{\Gamma, \Gamma_1, \dots, \Gamma_n \vdash [t\ t_1 \dots t_n] : F}\ \text{Application}$$

where $F_i = (T_i\ T_{i_1} \dots T_{i_n})$

Figure 10: HTLC: Intensional fragment

2.4 The Hyperintensional Fragment

At the highest level, we introduce the hyperintensional fragment, where our objects of interest are procedures whose products are objects of either an extensional, or an intensional type, or procedures of lower order. Procedures are obtained by Trivialization on a term t of a given type T. Here, we assume that the term t is well-typed. Given a relation from the domain of basic types and function types to their constructions as co-domain, in our calculus this relation is one-to-many.

Execution works as an elimination rule for the type $*^T$; if the rule is applied to a higher-order construction, it lowers its degree. Note, however, that by Abstraction on constructions, we move from a term occurring hyperintensionally to a term occurring intensionally; and viceversa, by Application on constructions, we move from a term occurring intensionally to a term occurring hyperintensionally.[8] Therefore, rules of the hyperintensional level allow us to reason about constructions, the creation of functions of constructions and their evaluation; to reason about functions of constructions, we move down to the intensional fragment. Given a relation from the set of constructions as domain to the set of their products as co-domain, in our calculus such relation is many-to-one. The construction rules of this fragment are shown in Figure 11. Note that by the explicit requirement that the term t^* in the syntax is defined recursively, we admit variables x_i^*. While for terms appearing on the right-hand side of the symbol $\vdash$, the operator * is obtained by Trivialization, in the case of the Assumption rule for the hyperintenstional fragment, it is possible instead to let appear the * operator also on terms appearing on the left-hand side of $\vdash$, namely when these terms are variables. This is required to avoid improper constructions at the level of variables, i.e. obtaining hyperintensional terms on which no

[8]Again, in Section 4.1 we provide proper definitions of term occurrence.

$$\frac{}{x_i^* : *^T \vdash x_i^* : *^T}\ \text{Assumption}$$

$$\frac{\Gamma, x_1^* : *^{T_1}, \ldots, x_n^* : *^{T_n} \vdash t^* : *^T}{\Gamma \vdash [\lambda x_1^* \ldots x_n^* \,.\, t^*] : (*^T\ *^{T_1}\ \ldots *^{T_n})}\ \text{Abstraction}$$

$$\frac{\Gamma \vdash t : (*^T\ *^{T_1}\ \ldots *^{T_n}) \qquad \Gamma_1 \vdash t_1^* : *^{T_1} \ldots \Gamma_n \vdash t_n^* : *^{T_n}}{\Gamma, \Gamma_1, \ldots, \Gamma_n \vdash [t\ t_1^* \ldots t_n^*] : *^T}\ \text{Application}$$

Figure 11: HTLC: Hyperintensional fragment

Execution rule can be applied, and hence for which no product can be associated. To avoid this, we consider variables for hyperintensional types of the form x_i^*, then the Execution rule can always be applied on them, producing a variable of extensional or intensional type T.

In order to exemplify a derivation in which both terms of the hyperintensional and intensional types occur, we show a tree where we move from a construction $t^* : *^{(T\ T_1)}$ whose product is a function of type $(T\ T_1)$, to a function of type $(*^T *^{T_1})$: for this, we require first that the function at the *intensional* level be obtained by Execution and Application, followed by Trivialization and finally Abstraction:

$$\frac{\dfrac{\dfrac{\dfrac{\Gamma, x_1^* : *^{T_1} \vdash t^* : *^{(TT_1)}}{\Gamma, x_1^* : *^{T_1} \vdash t : (TT_1)}\ \text{Execution} \qquad \begin{array}{c}\vdots\\ \vdash t_1 : T_1\end{array}}{\Gamma, x_1^* : *^{T_1} \vdash [t\ t_1] : T}\ \text{Application}}{\Gamma, x_1^* : *^{T_1} \vdash [t\ t_1]^* : *^T}\ \text{Trivialization}}{\Gamma \vdash [\lambda x_1^* \,.\, [t\ t_1]^*] : (*^T *^{T_1})}\ \text{Abstraction}$$

In the opposite direction, we can easily proceed from a function of type $(*^T *^{T_1})$ whose product is a function of type $(T\ T_1)$ obtained by the detour illustrated in Section 2.1 for non-canonical terms, to a construction of type $*^{(T\ T_1)}$:

$$\frac{\dfrac{\Gamma \vdash [\lambda x_1^* . t^*] : (*^T *^{T_1})}{\Gamma \vdash [\lambda x_1 . t] : (T\ T_1)}\ \text{Execution}}{\Gamma \vdash [\lambda x_1 . t]^* : *^{(T\ T_1)}}\ \text{Trivialization}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\vdash =: (o\tau\tau)} \quad \dfrac{}{x : \tau \vdash x : \tau}\,\text{Assum} \quad \dfrac{\vdots}{\vdash 1 : \tau}}{x : \tau \vdash [=\ x\,1] : o}\,\text{Application}}{\vdash [\lambda x.[=\ x\,1]] : (o\tau)}\,\text{Abstraction} \qquad \dfrac{\dfrac{\vdots}{\vdash [\mathit{Succ}\ 0]^{*} : *^{\tau}}}{\vdash [\mathit{Succ}\ 0] : \tau}\,\text{Execution}}{\vdash [[\lambda x.[=\ x\,1]]\,[\mathit{Succ}\ 0]] : o}\,\text{Application}$$

Then we can perform β-reduction:

$$[[\lambda x.[=\ x\,1]]\,[\mathit{Succ}\,0]] \to_{\beta} [=\ [\mathit{Succ}\,0]\,1]$$

Figure 12: HTLC: Functional identity between numbers

3 A Comparative Example

In a language like TIL, it is possible to compute with non-executed constructions and their products. For example, one could construct the set of constructions producing number one, and then take one element in such a set, e.g. “Successor of 0”. The process of checking whether this element belongs to that set eventually results in checking the equality of the product of this construction with number 1. HTLC allows similar expressions, although it is more strict in terms of type matching, so that the type a function requires for its argument must always be properly met: i.e. either the functional term is of type $(T_1\ *^{T_2})$ and its argument of type $*^{T_2}$ (the output type of this function is not relevant, and it can be $*^{T_1}$ as well); or respectively $(T_1\ T_2)$ and T_2.

The formulation of such a function at the intensional level between numbers is reflected in the tree in Figure 12. In this example, the function $=$ takes two numbers as arguments $(\tau\tau)$ and it returns a truth-value (o). Given a variable in the type of numbers, and number 1, by Application and Abstraction we build the λ-term that takes a number to be substituted for a variable and it compares for equality with 1, in order to return a truth-value. Consider then a construction to produce the successor function of the number 0, i.e a term of type $*^{\tau}$ which denotes one of the possible ways of expressing the successor of 0, for example by stroke notation $0'$: by Execution we obtain the term denoting the actual product of the construction; by Application we pass the term denoting this number to the function of type $(o\tau)$, to obtain a truth value. In this case the identity is at the extensional level, between the product of a procedure (of order 1) and a number.

On the other hand, it is possible to express the same content at the hyperintensional level as the equality between procedures, see Figure 13. In this case we

$$\dfrac{\dfrac{\dfrac{\begin{matrix}\vdots\\ \vdash\approx: (o *^\tau *^\tau)\end{matrix} \qquad \dfrac{}{x : *^\tau \vdash x : *^\tau}\,\text{Assum} \qquad \begin{matrix}\vdots\\ \vdash 1^* : *^\tau\end{matrix}}{x^* : *^\tau \vdash [\approx\ x^*\, 1^*] : o}\,\text{App.}}{\vdash [\lambda x^*.[\approx\ x^*\, 1^*]] : (o\, *^\tau)}\,\text{Abstraction} \qquad \dfrac{\begin{matrix}\vdots\\ \vdash (Succ\, 0) : \tau\end{matrix}}{\vdash [Succ\, 0]^* : *^\tau}\,\text{Trivializ.}}{\vdash [[\lambda x^*.[\approx\ x^*\, 1^*]]\, [Succ\, 0]^*] : o}\,\text{Application}$$

Then we can perform β-reduction:

$$[[\lambda x^*.[\approx, x^*\, 1^*]]\, [Succ\, 0]^*] \to_\beta [\approx\ [Succ\, 0]^*\, 1^*]$$

Figure 13: HTLC: Functional Identity between Procedures

consider equality between a construction for a number and a construction for the number 1, returning a truth value. In this example, the function $\approx$ takes two constructions as arguments $(*^\tau *^\tau)$ and it returns a truth-value (o). Given a variable for the construction of numbers, and a construction for number 1, by Application and Abstraction we build the λ-term that takes a construction for the successor of 0 to be substituted for a variable, and it compares for identity with a construction for 1, in order to return a truth-value. Note that we derive here the argument by Trivialization. This term β-reduces to the identity between a construction for the successor of 0 and a construction for 1, with the identity being false or true, depending from which construction is chosen for number 1, i.e whether the same construction is selected. Note that this corresponds to the function between constructions and their products being many-to-one.

4 Meta-theory

4.1 Term occurrence

HTLC allows to identify extensional, intensional and hyperintensional terms by inspecting the derivation tree under consideration and looking at the rule applied at the relevant step. In the following, we provide appropriate definitions of the extensional, intensional or hyperintensional occurrence of a term at a given step of a derivation.[9]

[9]For the same properties TIL relies on the inductive definition of the structure of the relevant construction. For details, see [5].

Definition 3 (Extensional Occurrence). *A term t occurs extensionally at step n of a tree if and only if it results from:*

1. *an Assumption rule, and it is of type α;*
2. *an Application rule, and it is of type α;*
3. *an Execution rule on a term of type $*^{\alpha}$.*

Consider as an example the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash [\lambda x^*.\, t^*] : (*^o *^o) \qquad \vdash t_1^* : *^o}{\Gamma \vdash [[\lambda x^*.\, t^*]\ t_1^*] : *^o}\,\text{Application}}{\Gamma \vdash [[\lambda x.\, t]\ t_1] : o}\,\text{Execution}}{\Gamma, x_1 : o \vdash [[\lambda x.\, t]\ t_1] : o}\,\text{Weakening by } x_1}{\Gamma \vdash [\lambda x_1.[[\lambda x.\, t]\ t_1]] : (o\ o)}\,\text{Abstraction}}{\Gamma \vdash [\lambda x_1.[[\lambda x.\, t]\ t_1]]^* : *^{(o\ o)}}\,\text{Trivialization}$$

The term $[[\lambda x.t]\ t_1]$ of type o resulting by Execution from the hyperintensional term $[[\lambda x^*.t^*]\ t_1^*]$ of type $*^o$ occurs extensionally at the third step of the derivation.

Definition 4 (Intensional Occurrence). *A term t occurs intensionally at step n of a tree if and only if it results from:*

1. *an Assumption rule and it is of type $(T\ T_1 \dots T_n)$;*
2. *an Abstraction rule;*
3. *an Application rule, and it is of type $(T\ T_1 \dots T_n)$;*
4. *an Execution rule on a term of type $*^{(T\ T_1 \dots T_n)}$.*

Consider as an example the term $[\lambda x_1.[[\lambda x.\, t]\ t_1]]$ of type $(o\ o)$ in the above derivation: it results from Abstraction, and it occurs therefore intensionally at the fifth step of the derivation.

Definition 5 (Hyperintensional Occurrence). *A term t occurs hyperintensionally at step n of a tree if and only if it results from:*

1. *a Trivialization rule;*
2. *an Assumption rule, and it is of type $*^T$;*

3. *an Application rule, and it is of type* $*^T$*;*

4. *an Execution rule on a term of type* $*^{*^T}$.

Consider as an example the term $[\lambda x_1.[[\lambda x.\, t]\ t_1]]^*$ of type $*^{(o\ o)}$ in the above derivation: it results from Trivialization, and it occurs therefore hyperintensionally at the last step of the derivation.

Finally, we are also able to define the occurrence of a term within a hyperintensional term by inspecting the result of an Execution rule.

Definition 6. *A term* t *occurs extensionally, respectively intensionally, or hyperintensionally in a hyperintensional term* t^* *at step* n *of a tree if and only if at step* $n+1$ *the term* t *occurs:*

1. *extensionally according to Definition 3, case 3;*

2. *intensionally according to Definition 4, case 4;*

3. *hyperintensionally according to Definition 5, case 4.*

Consider as an example again the term $[\lambda x_1.[[\lambda x.\, t]\ t_1]]^*$ of type $*^{(o\ o)}$ obtained by Trivialization in the above derivation from this section: it is a hyperintensional term in which a term occurs intensionally, i.e. an application of the Execution rule in a next step would return an intensional term.

4.2 Normalization

Execution is a converging rule, i.e. it is possible that distinct constructions can be generated from β-equivalent terms (i.e. terms for which the symmetric closure of $\twoheadrightarrow_\beta$ holds) of a base type or of a function type by Trivialization, and hence they return the same entity when executed. This is the classical example of failing identity for hyperintensions, where the expressions "bachelor" and "unmarried man" might fail to be identified as equal, but would eventually be applied truthfully to the same individual. Accordingly, an application of Trivialization on distinct but reducible terms t_1, t_2 may induce distinct hyperintensional terms t_1^*, t_2^*, each denoting a distinct construction of the same product. This relation in its general formulation is an instance of the so-called Diamond Lemma for terms related by Trivialization:

Lemma 1 (Trivialized Diamond)**.** *Let* $\Gamma \vdash t_1 : T$*,* $\Gamma \vdash t_2 : T$ *and* $t_1 \twoheadrightarrow_\beta t_2$*. Let, moreover,* $t_1^* : *^T$ *be obtained from* $t_1 : T$ *by Trivialization, and* $t_2^* : *^T$ *be obtained from* $t_2 : T$ *by Trivialization. Then* $t_1^* \twoheadrightarrow_\beta t_2^*$*.*

Proof. By induction on t_1^*, we only reason on the base atomic case. First reduce t_1^* to t_1 by Execution; now obtain t_2^* from t_2 by Trivialization. Let us now denote with $\twoheadrightarrow_{Exec}$ the transitive closure of Execution and $\twoheadrightarrow_\beta$; with $\rightarrow_{Triv}$ the term transformation resulting from Trivialization and with $\rightarrow_{Exec}$ the term transformation resulting from Execution. Then $t_1^* \twoheadrightarrow_{Exec} t_2 \rightarrow_{Triv} t_2^*$. If t_1^* and t_2^* are syntactically identical terms, the Lemma is trivially satisfied. Else: if it is not the case that $t_1^* \twoheadrightarrow_\beta t_2^*$ but $t_1^* \rightarrow_{Exec} t_1$ and $t_2^* \rightarrow_{Exec} t_2$, then because of failure of subject reduction t_1 and t_2 cannot have the same type, against the assumption that $t_1 \twoheadrightarrow_\beta t_2$. □

Note that subject reduction implies only β-reduction of constructions, which might not be guaranteed and is a weaker requirement than the notoriously problematic identity of hyperintensions. We can also show convergence for the transitive and reflexive closure of β-reduction (for the general case, i.e. also considering hyperintensions, not used in the above last step of the Diamond Lemma):

Theorem 1 (Church-Rosser)**.** *If $\Gamma \vdash t : T$, $t \twoheadrightarrow_\beta t'$ and $t \twoheadrightarrow_\beta t''$ then there is a term u such that $t' \twoheadrightarrow_\beta u$ and $t'' \twoheadrightarrow_\beta u$ and $\Gamma \vdash u : T$.*

Proof. By induction on t, t', t'', and u using subject reduction, and the Diamond Lemma if the term u is of the form t'^* (and thus t''^*). □

4.3 Completeness

A recent conjecture presented in [4] states that the non-hyperintensional fragment of total functions, without modalities (quantifying over possible worlds and times) of TIL is Henkin complete. HTLC only expresses total functions and proper constructions, without modalities quantifying over possible worlds and times. On this basis, we show the following version of completeness:

Theorem 2. *For any consistent set of closed well-formed formulas Λ of the form $t : o$ from HTLC there is a general model, in which*

- *the domain of basic and function types is denumerable,*
- *and the domain for hyperintensional types is non-denumerable but strongly reducible to a model with a denumerable domain for the lower types,*

with respect to which Λ is satisfiable.

Proof. We first consider standard properties of any consistent set of closed formulas Γ in HTLC.[10] We also use normalization by β-equivalence across contexts as a

[10]Consistency in a typed λ-calculus is notoriously guaranteed by the impossibility of typing a term $\lambda f.[\lambda x.[f[xx]]\lambda x.[f[xx]]]$. We assume here therefore that recursion can only be externally added to the language in order to preserve consistency of any set of formulas Γ.

congruence relation on terms. We now build the standard model:

$$M := \{D_\alpha, D_{(T\ T_1 \ldots T_n)}, D_{(*^T)}\}$$

containing a family of domains, one for each type:

- D_α stands for a meta-variable for each of the domains of basic types, i.e. $D_o, D_\iota, \ldots$ truth-values $\{True,\ False\}$, individuals, and any other required basic type;
- $D_{(T\ T_1 \ldots T_n)}$ is the domain of all total functions, with input of types $(T_1, \ldots, T_n)$ and values of type T, i.e. terms which after reduction by Application and possibly Execution are in the domain D_α;
- $D_{(*^T)}$ is the domain of all proper constructions, i.e. the set of elements of type $*^T$ generated from elements in the types α or $(T\, T_1 \ldots T_n)$.

Note that we define the standard model, and subsequently adding an evaluation on the equivalence class of formulas for the general model, by considering only the domain of constructions of order 1. This is required because the full evaluation of such domain can only be given by obtaining the terminal product of the construction, i.e. for proper constructions. This also means that when in the presence of higher order constructions, completeness can be guaranteed only by multiple Execution.

By an assignment ϕ with respect to the standard model we mean a relation from the set of variables into the domain of the appropriate type, i.e. the value of $\phi(x : T)$ is an element of D_T. We now define a relation V_ϕ associated with an assignment ϕ with respect to the standard model such that it assigns every formula to elements of a domain:

- for a formula $x_i : T$, the evaluation $V_\phi(x_i : T) = \phi(x_i : T)$;
- for a formula $[\lambda x_1, \ldots, x_n.t] : (T\ T_1 \ldots T_n)$, the evaluation $V_\phi([\lambda x_1, \ldots, x_n.t] : (T\ T_1 \ldots T_n))$ assigns an element in the domain $D_{(T\ T_1 \ldots T_n)}$, i.e. a function whose value for arguments $d_i \in D_{T_i}$ is $V_\psi(t : T)$, where ψ is an assignment with the same values as ϕ for all variables in t except for x_i, while $\psi(x_i : T_i)$ is d_i; and, $V_\phi([\lambda x_1.t] : (o\ o))$ has overall value $False$ iff $V_\phi(x_1 : o) = True$ and $V_\phi(t : o) = False$, otherwise $True$;
- for a formula $[t\, t_1 \ldots t_1] : T$, the evaluation $V_\phi([t\, t_1 \ldots t_n] : T)$ assigns the value of the function $V_\phi(t : (T\ T_1 \ldots T_n))$ in the domain D_T for the values of the arguments $V_\phi(t_i : T_i)$ in the domain D_{T_i};

- for a formula $t^* : *^T$, the evaluation $V_\phi(t^* : *^T)$ assigns elements in the domain D_{*^T} that produce elements in the domain D_T for each evaluation $V_\phi(t : T)$.

Note that V_ϕ is one-to-many because $V_\phi(t^* : *^T)$ assigns many elements in the domain D_{*^T}. With V_ϕ defined, let us define the standard notion of valid formula:

Definition 7 (Validity in the standard sense)**.** *A wff $t : o$ is valid in the standard sense if $V_\phi(t : o) = True$ for every assignment ϕ wrt the standard model.*

We now define a frame F by induction on T containing a family of domains, one for each type as defined above:

$$F := \{D_\alpha, D_{(o\ T_1 \dots T_n)}, D_{(*^T)}\}$$

Recall that we use β-equivalence as a congruence relation, thus two terms t and t' are equivalent iff $t =_\beta t'$ (i.e. the symmetric closure of $\twoheadrightarrow_\beta$). Given a formula $t : T$, we denote with $\lceil t : T \rceil$ the equivalence class of formulas containing terms congruent with t. Then we can define the following:

Definition 8 (General Model)**.** *A frame F is called a general model if a one-to-many relation $f(\lceil t : T \rceil)$ of equivalence classes of closed formulas $t : T$ is such that it assigns elements in the domain D_T.*

We now build the frame which is a model of a maximal consistent set of closed formulas Γ, which is clearly a superset of Λ, as follows:

- $f(\lceil t : o \rceil)$ is the value true or false depending on term t being in the set Γ or not, and hence D_o is the set of truth values $\{$*True*, *False*$\}$;

- $f(\lceil t : \iota \rceil)$ is the equivalence class of individuals $\lceil t : \iota \rceil$, hence D_ι is the set of equivalence classes of all terms of the type of individuals;

- and accordingly so for any other type in α;

- Assuming that D_o and D_{T_i} have been defined, as well as the value of f for all equivalence classes of terms of types o, and T_i and that every element of D_o, $D_{T_1}, \dots, D_{T_n}$ is the value of f for some $\lceil t : o \rceil$, $\lceil t_1 : T_1 \rceil, \dots, \lceil t_n : T_n \rceil$, respectively; then $f(\lceil t : (o\ T_1 \dots T_n) \rceil)$ is the function whose value in domain $D_{(o\ T_1,\dots,T_1)}$ is given by the value for the element $f(\lceil t_1 : T_1 \rceil)$ of D_{T_1}, up to $f(\lceil t_n : T_n \rceil)$ of D_{T_n} and returns the value of $f(\lceil t : o \rceil)$ of D_o;[11]

[11] Note that here we consider functions which have arguments of every possible type, including higher-order functions and hyperintensions, but only outputs of type o, i.e. truth-values. This allows us to define formulas in frames as those for which satisfiability and validity are defined.

- Assuming that D_α and $D_{(o\ T_1 \dots T_n)}$ have been defined, as well as the value of f for all equivalence classes of terms of types α and of type $(o\ T_1 \dots T_n)$ and that every element of D_α is the value of f for some equivalence class $\lceil t : o \rceil$ or $\lceil t : \iota \rceil$ and that every element of $D_{(o\ T_1 \dots T_n)}$ is the value of f for some equivalence class of corresponding terms; then $f(\lceil t^* : *^T \rceil)$ is a construction with value $f(\lceil t^* : *^T \rceil)$ in D_{*^T} for some element $f(\lceil t : T \rceil)$ of D_T.

- Assuming that D_{*^T} has been defined, as well as the value of f for all equivalence classes of terms of types $*^T$ and that every element of D_{*^T} is one of the values of f for some $t^* : *^T$; then $f^{-1}(\lceil t^* : *^T \rceil)$ is the execution of construction whose unique value for the element $f(\lceil t^* : *^T \rceil)$ of D_{*^T} is $f(\lceil t : T \rceil)$ of D_T.

Note that f is one-to-many as well because the domain D_{*^T} includes many values for $f(\lceil t^* : *^T \rceil)$, while the function $f^{-1}(\lceil t^* : *^T \rceil)$ returns the only input producing all such outputs. We now define formula validity and satisfiability of a set of formulas for the general type $T = \{\alpha, (o\ T_1 \dots T_n), *^T\}$:

Definition 9 (Validity in the general sense)**.** *A wff $t : o$ is valid in the general sense if $V_\phi(t : o) = True$ for every assignment ϕ wrt the general model.*

Definition 10 (Satisfiable set of formulas)**.** *A set of formulas Λ is satisfiable with respect to the frame $\{D_T\}$ for any type T, if there exists a valuation ϕ such that $V_\phi(t : o) = True$ for every formula $t : o$ in Λ.*

Lemma 2. *For every ϕ and $t : T$, $V_\phi(t : T) = f(\lceil t : T \rceil)$*

The proof of this intermediary Lemma is by induction on $t : T$.

- If $t : \alpha$ is of the form $x_i : \alpha$ and $\phi(x : \alpha)$ is the element $f(\lceil t : T \rceil)$ in the domain D_α, then $\phi(x : \alpha)$ is a consistent formula $t : \alpha$ such that $V_\phi(x : \alpha) = \phi(x : \alpha) = f(\lceil t : \alpha \rceil) = V_\phi(t : \alpha)$.

- If $t : T$ is of the form $[\lambda x_1 \dots x_n.t] : (o\ T_1 \dots T_n)$ and $V_\phi([\lambda x_1 \dots x_n.t])$; then the element $f(\lceil t : o \rceil)$ is a consistent formula in the domain D_o when each $\phi(x_i : T_i)$ is a closed formula $t_i : T_i$ in the domain D_{T_i}, and if $V_\phi(x_i : T_i) = \phi(x_i : T_i) = f(\lceil t_i : T_i \rceil) = V_\phi(t_i : T_i)$ then $V_\phi(t : o) = \phi(t : o) = f(\lceil t : o \rceil) = V_\phi(t : o)$.

- If $t : T$ is of the form $[t\ t_1 \dots t_n] : o$ and $V_\phi([t\ t_1 \dots t_n])$ is the element $f(\lceil [t\ t_1 \dots t_n] : o \rceil)$ in the domain D_o, then $\phi([t\ t_1 \dots t_n])$ is a closed formula such that every $t_i : T_i$ is one element in the corresponding domain D_{T_i}, in which case $[t\ t_1 \dots t_1] : o$ is a closed formula interpreted in the domain $D_{(o\ T_{1,} \dots T_n)}$, such that $V_\phi([t\ t_1 \dots t_n]) = \phi([t\ t_1 \dots t_n] : o) = f(\lceil [t\ t_1 \dots t_n] \rceil : o) = V_\phi([t\ t_1 \dots t_n])$.

- Let us consider here the novel case $t^* : *^T$. We assume that $f(\lceil t : T \rceil)$ has already been defined for $T = \alpha$ or $T = (o\ T_1 \dots T_n)$, and $V_\phi(t : T) = f(\lceil t : T \rceil)$ and $V_\phi(t : (T\ T_1 \dots T_n)) = f(\lceil t : (T\ T_1 \dots T_n) \rceil)$ respectively. Now the value of $f(t^* : *^T)$ is defined as one of the elements in the Domain D_{*^T} as the relation is one-to-many. Consider any two terms $\{t_1^*, t_2^*\}$ such that $\{t_1^*, t_2^*\} \in D_{*^T}$, then $t_1^* \rightarrow_\beta t_2^*$ must hold by Lemma 1, assuming it holds that $t_1 \rightarrow_\beta t_2$ for $\{t_1, t_2\} \in D_T$. Hence, if $V_\phi(t_1 : T) = f(t_1 : T) = f(t_2 : T) = V_\phi(t_2 : T)$, then $f(t : T) = f^{-1}(t^* : *^T)$, for t any of t_1, t_2 and t^* any of t_1^*, t_2^*. Hence $V_\phi(t : T) = f^{-1}(\lceil t^* : *^T \rceil)$.

The frame $F = \{D_\alpha, D_{(o\ T_1 \dots T_n)}, D_{(*^T)}\}$ is a general model because for every formula $t : T$ and assignment ϕ, $V_\phi(t : T)$ is an element of the domain D_T for each element of $f(\lceil t : T \rceil)$. The elements of D_α and $D_{(o\ T_1 \dots T_n)}$ are in one-to-one correspondence with the normalised set (equivalence class) of formulas, hence the domains are infinitely enumerable (and possibly finite). For the domain D_{*^T} though, this is not the case as the relation between the values of $f(t : T)$ and $f(t^* : *^T)$ is one-to-many. Any value of $V_\phi(t^* : *^T)$ in the domain D_{*^T} normalizes with respect to the value of $V_\phi(t : T)$ in the domain D_T for which $f^{-1}(\lceil t^* : *^T \rceil)$ holds. Since for every formula $t : T$ its denotation in the domain is given by $V_\phi(t : T)$ for ϕ arbitrary, and for every $t : \alpha$ and $t : (o\ T_1 \dots T_n)$ of any consistent Γ there is such a term in the appropriate domain D_α and $D_{(o\ T_1 \dots T_n)}$ respectively; and for every $t^* : *^T$ there is a function which returns an element in D_α or $D_{(o\ T_1 \dots T_n)}$; it follows that for every $\Lambda \subseteq \Gamma$ a valuation $V_\phi(t : T)$ assigns an element in the domain D_T, i.e. Λ is satisfiable with respect to the model M, and D_T is denumerable for $T = \{\alpha, (o\ T_1 \dots T_n)\}$, while elements of D_{*^T} normalize with respect to elements in D_T. □

Theorem 3. *Any closed wff $t : T$ is derivable in HTLC if and only if $t : T$ is valid in the general sense.*

Proof. We prove by induction from the basic case.

$\leftarrow$ 1. For $t : o$, the formula is valid by Definition 9 iff $V_\phi(x_i : \alpha)$ is valid once every free variable x_i occurring in t has been substituted, and there is an element in the corresponding domain of the general model D_o. In such a case, any formula $t' : o$ with variable x' with evaluation $V_\phi(t' : o) = False$ i.e. such that $V_\phi(x_i' : \alpha) = False$ cannot be consistent with $t : o$; in particular, $V_\phi([\lambda x_i'.t : (o\ \alpha)]) = True$ and $V_\phi([t\ t'] : o) = True$;

2. For $t : T$ of the form $[\lambda x_1 \dots x_n.t] : (o\ T_1 \dots T_n)$, the argument generalizes the previous one with several arguments;

3. For $t : T$ of the form $t^* : *^T$, such a formula is valid iff for every value of $V_\phi(t^* : *^T)$ the picked element from the domain D_{*T} (satisfying congruence with any other in the same domain) corresponds to $V_\phi(t : o) = True$. Hence, one reduces first to such a value by an application of Execution and then the argument runs according to the previous step for either $t : \alpha$ or $[\lambda x_1 \dots x_n.t] : (o\ T_1 \dots T_n)$

$\rightarrow$ Starting from our Assumption as axiom, both Abstraction and Application preserve validity, using Execution where necessary. Note that Trivialization is not invoked and all intra-context operations are from the domain D_{*T} to the domain D_T.

□

To conclude, we reformulate the last step of the completeness proof to show satisfiability based on compactness; the only constraint is again that in order to preserve denumerability of the domain of reference, sets of formulas including hyperintensions need to be reduce to the corresponding formulas with executed terms:

Theorem 4. *A set Γ of closed well-formed formulas is satisfiable with respect to*

- *some model of denumerable domains $D_o, D_{o\ T_1,\dots,T_n}$*
- *and some model of a non-denumerable domain D_{*T}*

if and only if every finite subset Λ of Γ is satisfiable.

Proof. $\rightarrow$
- If Γ is not satisfiable with respect to some model of a denumerable domain $D_o, D_{o\ T_1,\dots,T_n}$ then it is inconsistent by Theorem 2, i.e. in particular $\Gamma \vdash [\lambda x_n.t_n] : (o_n\ T_n)$. Then there is a finite $\Lambda \subset \Gamma$ such that $\Lambda = \{x_1 : A_1, \dots x_{n-1} : A_{n-1}\}$ and $\vdash [\lambda x_1, \dots, x_{n-1}, x_n.t_n] : (o_n\ T_1, \dots, T_n)$; but then this formula is valid because derivable for Theorem 3 hence there is also a $V_\phi(x_i : T_i) = False, i = 1 \dots n-1$ for every ϕ with respect to any model, hence Λ is not satisfiable. Then, if every $\Lambda \subset \Gamma$ is satisfiable, also Γ is satisfiable with respect to a denumerable domain.
- if Γ has formulas of the form $t : *^T$, then the domain D_{*T} is non-denumerable; apply Execution to reduce to Γ' with respect to denumerable domains $D_o, D_{o\ T_1,\dots,T_n}$. Proceed as above.

$\leftarrow$ Immediate: if every subset of Γ is satisfiable, then Γ is.

□

5 Conclusions

The system HTLC is an extension of typed λ-calculus with hyperintensions. The system is presented with a polymorphic rules set which can be applied to terms of arbitrary types: a triplet of rules, namely Assumption, Abstraction and Application known from λ-calculus are extended by Trivialization and Execution rules for terms denoting hyperintensions, and reason with them. We have provided formal definitions of term occurrence (which corrresponds to a proof inspection for type-checking) and we formulated appropriate versions of the Diamond Lemma and the Church-Rosser Theorem valid with respect to the extension to terms denoting hyperintensions. Finally, the system is shown to be complete in Henkin's sense, with respect to a general model of basic types, functions whose values belong to the set of truth values, and constructions of these types. The important difference to be drawn with standard completeness for Henkin's model concerns the cardinality of the model for hyperintensions, which cannot be denumerable. Nonetheless, our systems guarantees strong reducibility to the denumerable model of the trivialised term for each hyperintensional one.

Further possible investigations of this system concern: a computational interpretation of the extensional fragment, and the appropriate interpretation of both intensional (by higher order computations) and hyperintensional fragments (e.g. in terms of monads); a modal extension of the language, to express more precisely contexts in which lifting to hyperintensional terms is valid, e.g. on the lines formulated in [20]; and an implementation for type-checking purposes.

References

[1] F. Berto. Simple hyperintensional belief revision. *Erkenntnis*, 84:559–575, 2019.

[2] L. Burke. P-HYPE: A monadic situation semantics for hyperintensional side effects. *Proceedings of Sinn und Bedeutung*, 23(1):201–218, 2019.

[3] S. Charlow. *On the semantics of exceptional scope.* PhD thesis, New York University, 2014.

[4] M. Duží. Hyperintensions as abstract procedures. Presented at Congress on Logic Methodology and Philosophy of Science and Technology 2019.

[5] M. Duží, M. Fait, and M. Menšík. Context recognition for a hyperintensional inference machine. In *AIP Conference Proceedings of* ICNAAM 2016*, International Conference of Numerical Analysis and Applied Mathematics*, volume 1863, 2017.

[6] M. Duží, B. Jespersen, and P. Materna. *Procedural Semantics for Hyperintensional Logic - Foundations and Applications of Transparent Intensional Logic*, volume 17 of *Logic, Epistemology, and the Unity of Science.* Springer, 2010.

[7] M. Duží and M. Menšík. Inferring knowledge from textual data by natural deduction. *Computación y Sistemas*, 24(1), 2020.

[8] Ch. Fox and S. Lappin. Type-theoretic logic with an operational account of intensionality. *Synthese*, 192(3):563–584, 2015.

[9] N. Francez. The granularity of meaning in proof-theoretic semantics. In Nicholas Asher and Sergei Soloviev, editors, *Logical Aspects of Computational Linguistics - 8th International Conference, LACL 2014, Toulouse, France, June 18-20, 2014. Proceedings*, volume 8535 of *Lecture Notes in Computer Science*, pages 96–106. Springer, 2014.

[10] M. Jago. Hyperintensional propositions. *Synthese*, 192(3):585–601, 2015.

[11] B. Jespersen and M. Duží. Introduction. *Synthese*, 192(3):525–534, Mar 2015.

[12] B. Jespersen and G. Primiero. Alleged assassins: Realist and constructivist semantics for modal modification. In Guram Bezhanishvili, Sebastian Löbner, Vincenzo Marra, and Frank Richter, editors, *Logic, Language, and Computation - 9th International Tbilisi Symposium on Logic, Language, and Computation, TbiLLC 2011, Kutaisi, Georgia, September 26-30, 2011, Revised Selected Papers*, volume 7758 of *Lecture Notes in Computer Science*, pages 94–114. Springer, 2011.

[13] H. Leitgeb. HYPE: A system of hyperintensional logic (with an application to semantic paradoxes). *J. Philosophical Logic*, 48(2):305–405, 2019.

[14] P. Martin-Löf. Constructive mathematics and computer programming. In *Proc. of a Discussion Meeting of the Royal Society of London on Mathematical Logic and Programming Languages*, page 167–184, USA, 1985. Prentice-Hall, Inc.

[15] A. Özgün and F. Berto. Dynamic hyperintensional belief revision. *Review of Symbolic Logic*, pages 1–46, `https://doi.org/10.1017/S1755020319000686`.

[16] I. Pezlar. *Investigations into Transparent Intensional Logic: A Rule-based Approach.* PhD thesis, Masaryk University, 2016.

[17] I. Pezlar. Proof-theoretic semantics and hyperintensionality. *Logique et Analyse*, 61(242):151–161, 2018.

[18] C. Pollard. Hyperintensions. *Journal of Logic and Computation*, 18(2):257–282, 2008.

[19] C. Pollard. Agnostic hyperintensional semantics. *Synthese*, 192(3):535–562, 2015.

[20] G. Primiero. A contextual type theory with judgemental modalities for reasoning from open assumptions. *Logique & Analyse*, 55(220):579–600, 2012.

[21] L. Tranchini. Proof-theoretic harmony: towards an intensional account. *Synthese*, 2016, `https://doi.org/10.1007/s11229-016-1200-3`.

[22] L. Tranchini and A. Naibo. Harmony, stability, and the intensional account of proof-theoretic semantics. Presented at Congress on Logic Methodology and Philosophy of Science and Technology 2019, HaPoC Symposium on Identity in computational formal and applied systems.

Received 6 February 2020