GFOL: A Term-Generic Logic for Defining \( \lambda \)-Calculi

Andrei Popescu  
University of Illinois at Urbana-Champaign  
popescu2@cs.uiuc.edu

Grigore Rosu  
University of Illinois at Urbana-Champaign  
grosu@cs.uiuc.edu

Abstract

Generic first-order logic (GFOL) is a first-order logic parameterized with terms defined axiomatically (rather than constructively), by requiring them to only provide generic notions of free variable and substitution satisfying reasonable properties. GFOL has a complete Gentzen system generalizing that of FO. An important fragment of GFOL, called HORN\(^2\), possesses a much simpler Gentzen system, similar to traditional context-based derivation systems of \( \lambda \)-calculus. HORN\(^2\) appears to be sufficient for defining virtually any \( \lambda \)-calculus (including polymorphic and type-recursive ones) as theories inside the logic. GFOL endows its theories with a default loose semantics, complete for the specified calculi.

Categories and Subject Descriptors F.4.1 [Mathematical Logic]: Lambda calculus and related systems, Model theory, Proof theory

General Terms Theory, Languages

Keywords generic first-order logic, substitution, \( \lambda \)-calculus

1. Introduction

First-order logic (FOL) is one of the best-established logics in computer science. The models of FOL, called first-order structures, as well as its complete Gentzen deduction system are well understood and intuitive, thus making FOL an attractive formalism with many applications in specification and verification of systems, data-bases, automated reasoning, etc. Problems can be represented as FOL theories, i.e., as sets of FOL formulae over corresponding operational and relational symbols; then FOL provides, in a uniform way, appropriate models together with complete deduction rules.

FOL does not allow variables to be bound in terms (but only in formulae, via quantifiers), thus providing a straightforward notion of substitution in terms. On the other hand, most calculus that are used in the domain of programming languages, and not only, are crucially based on the notion of binding of variables/\textit{terms}: terms “export” only a subset of their variables, their free variables, that can be substituted. Because of their complex formulation for terms, these calculi cannot be naturally defined as FOL theories. Consequently, they need to define their own models and deduction rules, and to state their own theorems of completeness, not always easy to prove. In other words, they are presented as entirely new logics, as opposed to theories in an existing logic, thus incurring all the drawbacks (and boredom) of repeating definitions and proofs following generic, well-understood patterns, but facing new “details”.

In this paper we define term-generic first-order logic, or simply generic first-order logic (GFOL), as a first-order logic parameterized by any terms that come with abstract notions of free variable and substitution. More precisely, in GFOL terms are elements in a generic set \( \text{Term} \) including a subset \( \text{Var} \) whose elements are called variables, that comes with functions \( \text{FV} : \text{Term} \to \mathcal{P}(\text{Var}) \) and \( \text{Subst} : \text{Term} \times \text{Term} \to \text{Term} \) for free variables and substitution, respectively, that satisfy some expected properties. GFOL models provide interpretations of terms that satisfy, again, some reasonable properties. We show that GFOL admits a complete Gentzen-like deduction system, which is syntactically very similar to that of FOL, its proof of completeness modifies the classic proof of completeness for FOL to use the generic notions of term, free variables, substitutions and their generic properties. Extensions of GFOL with equality and with multiple sorts are also discussed.

By not committing to any particular definition of term, GFOL can be instantiated to different types of terms, such as, e.g., standard FOL terms, or \( \lambda \)-terms, or different categories of typed \( \lambda \)-terms, etc. When instantiated to standard FOL terms, GFOL becomes, as expected, precisely FOL. However, when instantiated to more complex terms, e.g., the terms of \( \lambda \)-calculus, GFOL becomes a logic where a particular calculus is a particular theory. For example, the GFOL formulae for extensionality in untyped \( \lambda \)-calculus and typing of abstractions in simply-typed \( \lambda \)-calculus can be

\[
\forall x,y,(\forall z.x \equiv y \rightarrow x = y) \quad \text{and} \quad (\forall x.\text{type}(x,t) \rightarrow \text{type}(\lambda x.t,X)) \rightarrow \text{type}(\lambda x.t,X,t \rightarrow t')
\]

where \( x, y, z, t, t' \) denote data and type variables, respectively, and \( X \) denotes a data term. This way, a specification of a calculus in GFOL brings a meaningful complete semantics for that calculus, because the axioms are stated about some models, the content of the axioms making the models “desirable”. Indeed, GFOL models are “blank” models in the sense that they are only required to interpret the terms in a way that is consistent with substitution – it is the axioms that “personalize” the models; e.g., the above typing rule asks that, in any desirable model, \( \lambda x.t.X \) has type \( t \rightarrow t' \) whenever \( X \) has type \( t' \) independently of the choice of \( x \) of type \( t \).

Even though the completeness (being equivalent to semi-decidability) of a fragment of a logic (whose syntax is decidable) follows from the completeness of the richer logic, there are good reasons to develop complete proof systems for certain particular sublogics as well. Besides a better understanding and self-containment of the sublogic, one important reason is the granularity of proofs. Indeed, proofs of goals in the sublogic that use the proof-system of the larger logic may be either long and “junkish” and may look artificial in the context of the sublogic. For example, equational logic admits a very intuitive complete proof system [4], that simply “replaces equals by equals”, thus avoiding the more intricate first-order proofs. An important goal of this paper is to

\[\text{copyright notice will appear here}\]
also investigate conditions under which sublogics of GFOIL admit specialized coarse-granularity proof systems.

It appears that a certain fragment of GFOIL, that we call HORN², is sufficient for calculi-specialization purposes. HORN² consists of GFOIL sentences of the form
\[ \forall \tau. \exists \eta, a, b. (\tau, \eta) \Rightarrow (a(T), b(T)) \Rightarrow (c(T)) \]
with \( a, b, c \) atomic formulas (\( \tau, \eta \) denote tuples of variables), i.e., generalized Horn implications whose conditions are themselves Horn implications. We show that, under a certain reasonable proof-theoretic restriction that we call amenability, a HORN² theory admits a complete Gentzen system that "implements" each HORN² formula as above into a deduction rule of the form
\[ \Gamma, a(\tau, T) \Rightarrow b(\tau, T) \text{ for any } i = 1, \ldots, n \]
\[ \Gamma \Rightarrow c(T) \]
where \( \tau \) is a fresh tuple of variables replacing \( \tau \), and \( T \) is a tuple of terms substituting \( \eta \).

Completions of HORN² have, as particular cases, term-generic versions of completeness results for conditionally-evaluated [4], Horn [17], and extensional [24] logics. Even more importantly, this completeness result qualifies HORN² as a higher-level notation for describing derivation systems for calculi, as it enables one to faithfully recover the original proof systems of the specified calculi in a uniform way, from their HORN² theories. For instance, the Horn deduction rule corresponding to the previously mentioned typing axiom is the familiar context-based typing rule for abstractions:
\[ \Gamma, \text{type}(f(z, t)) \Rightarrow \text{type}(f(T, t')), \text{ where } z \text{ is fresh wrt } \Gamma \]
\[ T \Rightarrow \text{type}(\lambda z : T. t \to t') \]
(Above, we viewed the type declaration \( z : t \) as an atomic formula \( \text{type}(f(z, t)) \), of the same category with \( \text{type}(\lambda z : T. t \to t') \)); note that the freshness assumption coincides with the usual requirement that \( z \) does not occur on the left of a type declaration in \( \Gamma \), in order to avoid conflicts.) The HORN² notation is not only compact and syntax-detail-free (like the one advocated by HOAS [19]), but also, by its very nature, model-theoretically meaningful.

Our main two contributions in this paper are:

- We show that the development of first-order logic is largely orthogonal to the particular syntax of terms. While previous work dealing with general terms models binding explicitly, we develop a logic, GFOIL, that abstracts away bindings by considering terms as "black-boxes" that export substitutions and free variables; all particular known terms with bindings satisfy the GFOIL generic axioms; and

- We provide a convenient notation and intuition for defining \( \lambda \)-caluli, that encourages a semantic specification style. GFOIL endows the specified calculi with a default complete semantics, via the GFOIL model of their defining theory.

Regarding the latter point, an early disclaimer is in order. The semantics that GFOIL brings for the specified calculi falls into the category of loose, or logical, semantics. Examples of loose semantics for \( \lambda \)-calculi include: (so called) "syntactic" models for untyped \( \lambda \)-calculus, Henkin models for simply-typed \( \lambda \)-calculus, Kripke-style models for recursive types, and Girard’s qualitative domains and Bruce-Meyer-Mitchell models for System F, not to mention all their categorical variants. For extensive presentations of these and many other loose semantics, we recommend the monographs [3, 10, 15]. For a particular calculus defined as a GFOIL theory, the implicit GFOIL semantics has all the advantages, but, naturally also all the drawbacks, of loose semantics. It is not the concern of this paper to advocate for a loose or for a fixed-model semantics, especially because we believe that there is no absolute answer. What we consider to be a particularly appealing aspect of GFOIL semantics though is its uniform, calculus-independent nature. And the "general-purpose" GFOIL semantics tends to be equivalent to the "domain-specific" ones developed for various calculi.

The remainder of this paper is structured as follows. Section 2 introduces GFOIL (syntax, models and some properties), its fragment HORN², and complete Gentzen systems for them. Section 3 presents and discusses specifications of various \( \lambda \)-calculi. Section 4 compares, for untyped \( \lambda \)-calculus and System F, their ad hoc complete semantics already defined in the literature with this GFOIL semantics obtained "automatically" from their definition as HORN² theories. Section 5 discusses related work, making room for GFOIL in the extensively studied field of general approaches to representing \( \lambda \)-calculi. A short concluding section ends the paper.

We have excised all the proofs into Appendix E.

2. Term-Generic First-Order Logic

We introduce a generic notion of first-order term, axiomatized by means of free variables and substitution, purposely not committing to any concrete syntax for terms. Then we show our first novel result in this paper, namely that the development of first-order logic essentially does not depend on the syntax of terms, but only on the properties of substitution. Additionally, as shown in Section 3, various forms of typed and untyped \( \lambda \)-calculi naturally fall into our framework by properly in stanclining the generic notion of term (and implicitly of free variable and substitution). To keep the discussion notionally simple, we first develop the logic in an unordered form and without equality, an later sketch an ordered extension.

2.1 Term Syntax

DEFINITION 1. Let \( \text{Var} \) be a countably infinite set of variables. A term syntax over \( \text{Var} \) consists of the following data:

(a) A countably infinite set \( \text{Term} \) such that \( \text{Var} \subseteq \text{Term} \), where whose elements are called terms;

(b) A mapping \( \text{FV} : \text{Term} \rightarrow \mathcal{P}(\text{Var}) \); elements of \( \text{FV}(T) \) are called free variables, or simply variables, of \( T \);

(c) A mapping \( \text{Subst} : \text{Term} \times \text{Term}_{\text{var}} \rightarrow \text{Term} \). These are subject to the following requirements (\( x, T, T', \theta, \theta' \) denote a variable, a term, \( \theta \) and \( \theta' \) denote a thinnary variables, terms, and maps in \( \text{Term}_{\text{var}} \) respectively):

\[ \text{Subst}(x, \theta) = \theta(x); \]
\[ \text{Subst}(T, 1_{\text{var}}) = T; \]
\[ \text{if } \theta \in \text{FV}(T) = \theta' \text{, then } \text{Subst}(T, \theta) = \text{Subst}(T, \theta') \]
\[ \text{Subst}(\text{Subst}(T, \theta), \theta') = \text{Subst}(T, \theta') \text{, where for each } x \in \text{Var}, \theta(x) \text{, is by definition, } \text{Subst}(\theta(x), \theta); \]
\[ \text{FV}(x) = \{x\}; \]
\[ \text{FV}(\text{Subst}(T, \theta)) = \{x \in \text{FV}(T) : x \in \text{FV}(\theta(x))\}. \]

From here on we may write a term syntax as a tuple \( \text{Term}, \text{Var}, \text{FV}, \text{Subst} \) or even just \( \text{Term} \) if the other components of the tuple are understood from context. Note that we assume the notion of term coming together with a notion of substitution which is compos (condition (4) above). Therefore, in our examples of calculi with bindings, we shall consider \( \alpha \)-equivalence classes of terms rather than bare terms, a reasonable assumption when working at the logical, and not the implementation, level. For distinct variables \( x_1, \ldots, x_n \), we write \( T[x_1 / x_1, \ldots, T_n / x_n] \) for the function \( \text{Var} \rightarrow \text{Term} \) that maps each \( x_i \) to \( T_i \), for \( i = 1, n \) and all the other variables to themselves, and \( T[T_1 / x_1, \ldots, T_n / x_n] \) for \( \text{Subst}(T, T_1 / x_1, \ldots, T_n / x_n) \).

PROPOSITION 1. The following hold:

1. \( x \notin \text{FV}(T) \) implies \( T[T'/x] = T; \)

\[ 1 \text{Here and elsewhere, by language abuse, we let } 1_{\text{var}} \text{ denote the inclusion mapping of } \text{Var} \text{ into } \text{Term}. \]
2. If $y[T/x] = T$ if $y$ is $x$ and $y[T/x] = y$ otherwise.
   3. $FV(T') = FV(T) \setminus \{x\} \cup FV(T')$.
   4. $T[y/x]_z[y] = T[y/x]_y' \forall y \notin FV(T)$.
   5. $T[y/x]_y = T$ if $y \notin FV(T)$.

2.2 First-Order Logic over a Term Syntax

**Definition 2.** A generic first-order language consists of a term syntax ($\text{Term, Var, FV, Subst}$) together with a countable ranked set $\Pi = (\Pi_i)_{i \in \mathbb{N}}$ of relation symbols.

If $\text{Term}$ is a term syntax as above, then we write generic first-order languages as $(\Pi, \text{Var, FV, Subst})$; if more components of the term syntax are relevant for a given context, then we can also mention them in the tuple, e.g., $(\Pi, \text{Var, FV, Subst})$.

**Definition 3.** A GFOL model is a triple $(\mathcal{A}, (\mathcal{A}_T)_{T \in \text{Term}})$, such that:

(a) $\mathcal{A}$ is a set, called the carrier set.
(b) For each $T \in \text{Term}$, $\mathcal{A}_T$ is a mapping $A^{\text{Var}} \rightarrow A$ such that:
   (i) $A_T(\rho) = \rho(x)$.
   (ii) For each $\rho$ and $\rho'$ such that $\rho|_{\text{FV}(T)} = \rho'|_{\text{FV}(T)}$, it holds $A_T(\rho(x)) = A_T(\rho'(x))$.
   (iii) $A_{\text{Subst}}(T, \theta) = A_T(A_\theta(p))$, where for each $T \in \text{Term}^{\text{Var}}$, $A_T^{\text{Var}} \rightarrow A$ is defined by $A_{\theta(p)}(x) = A_\theta(p)$.
(c) For each $\pi \in \Pi$, $\mathcal{A}_\pi$ is a $\pi$-ary relation on $A$.

Note that, unlike in classical FOL models where the interpretation of terms is built from operations, GFOL models the interpretation of terms is assumed (in the style of Henkin models). However, due to the axioms ruling these interpretations, when instantiated to FOL terms, GFOL yields essentially the same models.

**Term can be organized as a model** $(\text{Term}, (\mathcal{A}_T)_{T \in \text{Term}})_{\mathcal{A} \in \Pi}$, in many ways, corresponding to the choice of relations $\text{Term}_e$, by letting $\text{Term}_e(\rho)$ be $\text{Subst}(T, \rho)$. These are indeed models, since conditions (i)–(iii) of the model definition coincide in this case with (1), (3) and (4) in the term syntax definition. Any such model will be henceforth called a (GFOL) Herbrand model. If one defines model homomorphisms as expected, then one gets that the Herbrand model with all relations empty is free over $X$ in the category of models and model homomorphisms. However, we shall not be interested in such categorical/algebraic aspects here.

Above, and from now on, let us $x, y, z, u, v, t$, etc., range over variables, $T, T_1, T_2$, etc., over terms, $\phi, \rho$, etc., over valuations in $A^{\text{Var}}$, and $\pi, \pi'$, etc., over relation symbols. Formulas are the usual way, starting from atomic formulas $\pi(T_1, \ldots, T_n)$ and applying connectives $\land, \lor$ and quantifier $\forall$. We list $\text{Formula}$ denote the set of formulas. For each formula $\phi$, the set $A_\phi \subseteq A^{\text{Var}}$, of valuations that make $\phi$ true in $A$, is defined recursively on the structure of formulas as follows:

- $\rho \in A_{\phi}(T_1, \ldots, T_n)$ iff $(\mathcal{A}_T(\rho), \ldots, \mathcal{A}_{T_n}(\rho)) \in A_{\phi}$.
- $\rho \in A_{\phi}$ iff $\rho \notin A_{\phi}$.
- $\rho \in A_{\phi \land \phi'}$ iff $\rho \in A_{\phi}$ and $\rho \in A_{\phi'}$.
- $\rho \in A_{\phi \lor \phi'}$ iff $\rho[x \rightarrow a] \in A_{\phi}$ for all $a \in A$.

If $\rho \in A_{\phi}$, we say that $\phi$ satisfies $\phi$ under valuation $\rho$ and write $A \models_{\rho} \phi$. If $A = A^{\text{Var}}$, we say that $\phi$ satisfies $\phi$ and write $A \models \phi$. Given a set of formulas $\Gamma$, $A \models \Gamma$ iff $A \models \phi$ for all $\phi \in \Gamma$. Above, and from now on, let $\phi, \psi, \chi$ range over arbitrary formulas and $A$ over arbitrary models.

**Definition 4.** For each formula $\phi$, the set $FV(\phi)$, of its free variables, is defined recursively as follows:

- $FV(\pi(T_1, \ldots, T_n)) = FV(T_1) \cup \ldots \cup FV(T_n)$.
- $FV(\neg \phi) = FV(\phi)$.
- $FV(\phi \land \psi) = FV(\phi) \cup FV(\psi)$.
- $FV(\forall x. \phi) = FV(\phi) \setminus \{x\}$.

Note that, since $FV(T)$ is finite for each term $T$, $FV(\phi)$ is also finite for each formula $\phi$. A sentence is a closed formula $\phi$, i.e., one with $FV(\phi) = \emptyset$. A theory, or a specification, is a set of sentences.

**Definition 5.** Substitution of terms for variables in formulas, $\text{Subst} : \text{Formula} \times \text{Term}^{\text{Var}} \rightarrow \text{Formula}$, is defined as follows:

- $\text{Subst}(\pi(T_1, \ldots, T_n), \theta) = \pi(\text{Subst}(T_1, \theta), \ldots, \text{Subst}(T_n, \theta))$.
- $\text{Subst}(\neg \phi, \theta) = \neg \text{Subst}(\phi, \theta)$.
- $\text{Subst}(\phi \land \psi, \theta) = \text{Subst}(\phi, \theta) \land \text{Subst}(\psi, \theta)$.
- $\text{Subst}(\forall x. \phi, \theta) = \forall x. \text{Subst}(\phi[x \rightarrow z], \theta)$, where $z$ is the least variable not in $FV(\phi) \cup \bigcup \{\theta(y) : y \in FV(\phi)\}$.

For substitution in formulas we adopt notational conventions similar to the ones about substitution in terms, e.g., $\phi[T/x]$.

**Definition 6.** $\phi$-equivalence of formulas, written $\equiv_{\phi}$, is defined to be the least relation $R \subseteq \text{Formula} \times \text{Formula}$ satisfying:

- $\pi(T_1, \ldots, T_n) \equiv_{\phi} \pi(T_1, \ldots, T_n)$.
- $\phi \equiv_{\phi} \psi$ if $\phi \equiv_{\phi} \psi$.
- $\phi \equiv_{\phi} \psi$, $\psi \equiv_{\phi} \phi$, $\phi \equiv_{\phi} \psi$.
- $\forall x. \phi \equiv_{\phi} \forall y. \psi$ if $\phi[z/x] \equiv_{\phi} \psi[y/z]$ for some $z \notin FV(\phi) \cup FV(\psi)$.

Thus $\text{GFOL}$ is a logic generic only w.r.t. terms - formulae are "concrete" first-order formulas over generic terms, with a "concrete" (and not generic) notion of $\equiv$-equivalence, standardly constructed on top of the identity of terms and using the quantifier bindings; however, in concrete cases involving bindings, the identity of terms is itself $\equiv$-equivalence w.r.t. term bindings.

For the following proposition, recall the definitions, for a model $A$ and two mappings $\theta, \theta' : \text{Var} \rightarrow \text{Term}$ of the composition $\theta \circ \theta' : \text{Var} \rightarrow \text{Term}$ and of the mapping $A_{\theta} : A^{\text{Var}} \rightarrow A^{\text{Var}}$.

**Proposition 2.** The following hold:

1. If $\rho|_{\text{FV}(\phi)} = \rho'|_{\text{FV}(\psi)}$, then $\rho \in A_{\phi}$ iff $\rho' \in A_{\psi}$.
2. $\rho \in A_{\text{Subst}(\phi, \theta)}$ iff $A_{\theta}(\rho) \in A_{\phi}$.
3. $\phi \equiv_{\phi} \psi$ implies $A_{\phi} = A_{\psi}$.
4. $\psi \equiv_{\phi} \phi$ implies $FV(\phi) = FV(\psi)$.
5. $\equiv_{\phi}$ is an equivalence.
6. $\phi \equiv_{\phi} \psi$.
7. $\phi \equiv_{\phi} \psi$.
8. $\phi \equiv_{\phi} \psi$.
9. $\phi \equiv_{\phi} \psi$.
10. $\theta|_{\text{FV}(\phi)} = \theta'|_{\text{FV}(\phi)}$ implies $\text{Subst}(\phi, \theta) \equiv_{\phi} \text{Subst}(\phi, \theta')$.
11. $\text{Subst}(\phi, \theta) \equiv_{\phi} \text{Subst}(\phi, \theta')$.
12. $\phi \equiv_{\phi} \psi$ and $\psi \equiv_{\phi} \phi$ imply $\phi \equiv_{\phi} \psi$.

Thus $\equiv_{\phi}$ is an equivalence, preserves satisfaction and the free variables, and is compatible with substitution and language constructs (points (5), (3), (4), (9), (12) above). Hereafter, we shall identify formulae modulo $\equiv_{\phi}$-equivalence, since mappings $\text{Subst}$, $\text{FV}$, $\text{Subst}$, $\text{A}$ and those that build formulae are well defined on equivalence classes.

\[\text{We may interpret "the least" as "having the least index", where we assume an indexing on the (countable) set of variables; we pick the least variable in order to make a choice - any variable with the mentioned property would do.}\]
Examples

FOL. As expected, classical FOL is an instance of GFO. Indeed, let \((\mathbf{V}, \Sigma, \Pi)\) be a first-order language (possibly with equality - see Subsection 2.3), where \(\mathbf{V}\) is a countably infinite set of variables, and \(\Sigma = \{\Sigma_n\}_{n \in \mathbb{N}}\) and \(\Pi = \{\Pi_n\}_{n \in \mathbb{N}}\) are ranked sets of operation and relation symbols. Let \(\text{Term}\) be the term syntax consisting of ordinary first-order terms over \(\Sigma\) and \(\mathbf{V}\) with \(FV : \text{Term} \to \mathcal{P}_f(\mathbf{V})\) giving all the variables in each term (all variables are free in FOL terms) and \(\text{Subst} : \text{Term} \times \text{Term} \rightarrow \text{Term}\) the normal substitution on FOL terms (no precautions need to be taken here, since there is no variable capture). Define a generic first-order language as \((\text{Term}, \Pi)\). A classical FOL model \((\mathcal{A}, (\mathcal{A}_e)_{e \in \Pi})\) yields a GFO model \((\mathcal{A}, (\mathcal{A}_e)_{e \in \Pi})\) by defining the meaning of terms as derived operations. Conversely, from a GFO model \((\mathcal{A}, (\mathcal{A}_e)_{e \in \Pi})\), one can extract a FOL model by defining \(A_\theta : A^n \rightarrow A\) as \(A_\theta(a_1, \ldots, a_n) = A_{\theta}(a_1, \ldots, a_n)\), where \(x_1, \ldots, x_n\) are distinct variables and \(\theta\) is a valuation that maps each \(x_i\) to \(a_i\). (The definition of \(A_\theta\) does not depend on the \(x_i\).) The two model mappings are mutually inverse and preserve satisfaction. Thus, for this particular choice of terms, GFO yields FOL.

A Formula-Type Logic. GFO is a trivial instance of GFO. However, GFO terms may be arbitrarily exotic. Besides terms of various \(\lambda\)-calculi (that will be discussed in Section 3), one may also have terms that interfere with formulae in non-trivial ways, as shown by the following example, where terms may abduct variables having formulae as types. Let \((\mathbf{V}, \Sigma, \Pi)\) be a first-order language and \(\text{Term}\) and \(\text{Formula}\) be defined mutually recursively:

\[
\text{Term} :: \equiv \mathbf{V} \cup \Sigma \cup \text{Term}, \text{Term} | \text{Term}(\text{Term}) \quad \text{Var} : \text{Term}, \text{Formula} \\
\text{Formula} :: \equiv \text{Term} \cup \Pi(\text{Term}, \ldots, \text{Term}) | \lnot \text{formula} \quad \text{Formula} \land \text{Formula} | \lor \text{Var}, \text{Formula}
\]

where productions \(\text{Term} :: \equiv \Sigma \cup \text{Term}, \ldots, \text{Term} \) and \(\text{Formula} :: \equiv \Pi(\text{formula}, \ldots, \text{formula})\) have the restriction that the number of \(\text{Term}\)'s equals the rank of the corresponding operation in \(\Sigma\) or relation in \(\Pi\). The free variables of terms are defined recursively by \(FV(\lambda x. \varphi. \psi) = (FV(\varphi) \cup FV(\psi)) \setminus \{x\}\) and on the other term constructs as expected, and term \(\alpha\)-equivalence and substitution as expected. It is easy to check that terms up to \(\alpha\)-equivalence form a GFO term syntax. Moreover, even though formulae were defined recursively together with the terms, they are still nothing but first-order formulae over the terms, hence they fall into the framework of GFO. This logic can be seen as an "extremely-typed" \(\lambda\)-calculus, and is itself powerful enough to capture several forms of typed calculus.

2.3 GFO with equality

A generic first-order language with equality is a generic first-order language that has an emphasized binary relation symbol \("=\)", interpreted in all models as the equality relation. All the other concepts remain the same.

GFO with equality is an important variant of GFO and will prove appropriate for conveniently capturing various \(\lambda\)-calculi, where equality plays a central role together with typing/kinding. Since typing will be defined as a binary relation which is implicitly compatible with equality in our framework, type preservation will hold by default in calculi specified in GFO with equality (see Section 3).

2.4 Many-Sorted and Order-Sorted GFO

The notions of a term syntax and term-generic first-order languages have straightforward many-sorted and order-sorted generalizations. We next sketch an order-sorted version of GFO, which also covers the many-sorted case. Order-sorted GFO generalizes order-sorted equational logic [9, 22].

Let \(S = (S, <)\) be a fixed poset. Elements of \(S\) are called sorts, and \(<\) is called the subclass relation. We assume that any two sorts \(s, s'\) having a common subclass \(t\) (i.e., a sort \(s''\) with \(s'' < s\) and \(s'' < s'\)), also have a greatest common subclass, denoted \(s \land s'\). An order-sorted set is a family of sets \(A = (A_s)_{s \in S}\) such that \(A_s \subseteq A_{s'}\) whenever \(s < s'\). Let \(\mathcal{A}\) denote the set \(\bigcup_{s \in S} A_s\). We call \(\mathcal{A}\) unambiguous if \(A_s \cap A_{s'} = A_{s \land s'}\) if \(s < s'\) and have a common subclass, and \(A_s \cap A_{s'} = \emptyset\) otherwise. Note that an unambiguous order-sorted set \(A\) can be recovered from \(\mathcal{A}(A)\) and the relation "has sort" between elements of \(\mathcal{A}(A)\) and \(S\).

Let \(\mathcal{A}\) be an order-sorted, sortwise countably infinite \(S\)-sorted set of variables. An order-sorted term syntax over \(\mathcal{A}\) consists of the following data:

(a) An unambiguous, sortwise countably infinite \(S\)-sorted set \(\text{Term}\) such that \(\mathcal{A} \subseteq \text{Term}\);

(b) A mapping \(FV : \text{All(\text{Term})} \rightarrow \mathcal{P}_f(\mathcal{A}(\mathbf{V}))\);

(c) A mapping \(\text{Subst} : \text{All(\text{Term})} \times \text{All(\text{Term})} \rightarrow \text{All(\text{Term})}\) such that, for each \(s \in S, T \in \text{Term}\), and \(\theta \in \text{All(\text{Term})}_S\), \(\text{Subst}(T, \theta) \in \text{Term}_s\),

\[
\begin{align*}
1. & \quad \text{Subst}(x, \theta) = \theta(x) \\
2. & \quad \text{Subst}(T, \lambda x. \varphi. \psi) = T \\
3. & \quad \text{Subst}(T, \theta) = \theta \cdot FV(T), \text{then Subst}(T, \theta) = \text{Subst}(T, \theta) \\
4. & \quad \text{Subst}(T, \theta) = \text{Subst}(T, \theta) \\
5. & \quad \text{FV}(x) = \{x\} \\
6. & \quad \text{FV}(\lambda x. \varphi. \psi)(x) = FV(\varphi) \cup FV(\psi) \setminus \{x\}
\end{align*}
\]

An order-sorted generic first-order language consists of the following: an unambiguous, sortwise countably infinite \(S\)-sorted set \(\mathcal{A}\); an order-sorted term syntax over \(\mathcal{A}\); a countable \(S\)-ranked set \(\Pi = \{\Pi_n\}_{n \in S}\), of relation symbols. A GFO model is a triple \((\mathcal{A}, (\mathcal{A}_e)_{e \in \Pi})\), such that:

(a) \(A\) is an \(S\)-sorted set;

(b) For each \(\pi \in \Pi_{s_1 \cdots s_n}\) \(A_s \subseteq A_{s_1} \times \cdots \times A_{s_n}\);

(c) For each \(T \in \text{All(\text{Term})}, \mathcal{A} T\) is a mapping \(\text{Map}(\mathcal{A}, T) \rightarrow \mathcal{A}\) such that whenever \(T \in \text{Term}_s, \mathcal{A} T(\rho) \in A_s\) for all \(\rho \in \text{Map}(\mathcal{A}, \mathbf{V})\); and:

\[
\begin{align*}
1. & \quad \text{Subst}(x, \theta) = \theta(x) \\
2. & \quad \text{Subst}(T, \lambda x. \varphi. \psi) = T \\
3. & \quad \text{Subst}(T, \theta) = \theta \cdot FV(T), \text{then Subst}(T, \theta) = \text{Subst}(T, \theta) \\
4. & \quad \text{Subst}(T, \theta) = \text{Subst}(T, \theta) \\
5. & \quad \text{FV}(x) = \{x\} \\
6. & \quad \text{FV}(\lambda x. \varphi. \psi)(x) = FV(\varphi) \cup FV(\psi) \setminus \{x\}\)
\end{align*}
\]

Now first-order formulae are defined as usual. All the concepts and results about GFO in this paper, including completeness of
various proof systems for various fragments of the logic, can be easily (but admiteddly somewhat) extended to the many-sorted and outer-sorted cases.

2.5 GFOL Gentzen System and Completeness

Next we show that the axiomatic properties of the generic notions of free variable and substitution in GFOL provide enough infrastructure for proving generic versions of classical FOL results. We are interested in a completeness theorem here, but other model-theoretic results will follow. We shall use the same cut-free Gentzen system as the one usually given in the classical setting [7]. It is worth mentioning that our system rather looks the same than is the same to the classical one, since in the tables below $T$ and $\Delta$ denote generic terms and substitution.

We fix a generic first-order language $\langle \text{Term}, \Pi \rangle$. A sequent is a pair written $\Gamma \vdash \Delta$, with $\Gamma$ and $\Delta$ (at most) countable sets of formulae, called the antecedent and the succedent of the sequent. A sequent $\Gamma \vdash \Delta$ is called tautological if for each model $A, \bigwedge_{\varphi \in \Gamma} A \subseteq \bigcup_{\psi \in \Delta} A$, and falsifiable if it is not tautological. A rule is a pair $\frac{H}{\Delta}$ consisting of a sequent $S$ and a (possibly empty) list of sequents $H$. If $H = \cdot$ (i.e., it is empty) we call $\frac{H}{\Delta}$ an axiom.

The notion of a proof tree for a Gentzen system is defined the usual way - its nodes are labelled with sequents, in a way that is consistent with the rules: if a node is labelled with $\Gamma$, then its descendants, if they exist, are labelled with the elements of $\Gamma$, where $\frac{H}{\Delta}$ is a rule in the Gentzen system. A completed proof tree is one which has all its leaves labelled with axioms. A rule $\frac{H}{\Delta}$ is sound if whenever all sequents in $H$ are tautological, $\Delta$ is tautological too. A sequent is provable in a Gentzen system if it is the root of a completed proof tree. A Gentzen system is sound, if all its provable sequents are tautological, and complete if all tautological sequents are provable. Note that soundness of a Gentzen system is equivalent to soundness of each of its rules.

We consider the Gentzen system, say $\mathcal{G}$, given by the following rule schemes (we write $\Gamma \varphi$ instead of $\Gamma \cup \{\varphi\}$):

\[
\begin{array}{c|c}
\text{Left} & \text{Right} \\
\hline
\Gamma \varphi \Delta & \frac{}{\Gamma \varphi \Delta} \quad (\cdot) \\
\Gamma, \psi \Delta & \frac{}{\Gamma \psi \Delta} \quad (\cdot) \\
\Gamma \varphi \Delta \land \psi \Delta & \frac{}{\Gamma \varphi \land \psi \Delta} \quad (\cdot) \\
\frac{}{\Gamma \varphi \Delta} \quad \text{if } \Gamma \land \Delta \neq \emptyset \quad (Ax)
\end{array}
\]

Given a theory $E$, a sequent $\Gamma \vdash \Delta$ is called $E$-tautological if for each model $A \models E, \bigwedge_{\varphi \in \Gamma} A \subseteq \bigcup_{\psi \in \Delta} A$. The above Gentzen system is meant to entail tautological, i.e., $\varnothing$-tautological, GFOL sequents, and thus is not parameterized by any fixed theory $E$. Note however that, for a countable theory $E$, $\Gamma \vdash \Delta$ is $E$-tautological iff $\Gamma \models E$ is tautological, and thus the case of a fixed countable theory $E$ can be covered by adding $E$ to the antecedent of the desired sequent. Gentzen systems specialized for fixed theories, that apply the axioms of the theory directly as rules, are discussed in Subsection 2.6.

**Theorem 1.** Gentzen system $\mathcal{G}$ is sound and complete for GFOL.

To obtain a Gentzen system for GFOL with equality, we add to $\mathcal{G}$:

\[
\begin{array}{l}
\frac{\Gamma \varphi \Delta T = T}{\Gamma \varphi \Delta} \quad \text{(Inst-Refl)} \\
\frac{\Gamma \varphi \Delta T_1 = T_2 \quad \Gamma \varphi \Delta T_2 = T_3}{\Gamma \varphi \Delta T_1 = T_3} \quad \text{(Inst-Sym)} \\
\frac{\Gamma \varphi \Delta T_1 = T_3 \quad \Gamma \varphi \Delta \Delta = \Delta}{\Gamma \varphi \Delta T_1} \quad \text{(Inst-Trans)} \\
\frac{\Gamma \varphi \Delta \pi(T_1, \ldots, T_n) = T'}{\Gamma \varphi \Delta T_1, \ldots, \Gamma \varphi \Delta T_n = T'} \quad \text{(Inst-Comp)} \\
\frac{\Gamma \varphi \Delta T_1 = T_2 \quad \Gamma T[T_1/x] = T[T_2/x] \varphi \Delta}{\Gamma \varphi \Delta} \quad \text{(Inst-Subst)}
\end{array}
\]

We call these the equality rules. Note that the rules in the above two tables make full sense in our generic framework, since concrete syntax of terms is not required; all that is needed here are abstract notions of term and substitution. In concrete cases, congruence w.r.t. various kinds of operations will be captured as particular cases of (Inst-Subst). Let $\mathcal{G}_e$ be this enriched Gentzen system.

**Theorem 2.** $\mathcal{G}_e$ is sound and complete for GFOL with equality.

The notation used for the equality rules, "(Inst-...)", comes from regarding these rules as instances of some axiom-schemes, namely:

\[
\begin{array}{l}
x = x \\
x \Rightarrow y = y = x \\
x = y \land y = z \Rightarrow x = z \\
(x_1 = y_1 \land \ldots \land x_n = y_n \land \\
\pi(x_1, \ldots, x_n) = \pi(y_1, \ldots, y_n) \\
x = y \Rightarrow T[x \leftarrow x] = T[x \leftarrow y]
\end{array}
\]

Let $\mathcal{Eg}$ be the set of these equality axioms. The relationship between "axioms" and their associated "rules" will be exploited in the following subsection, where we develop a more effective theory-oriented Gentzen system for a fragment of GFOL.

2.6 The Horn^2 Fragment of GFOL

We next consider a fragment of GFOL, called Horn^2 because it only allows formulae which are universally quantified implications whose conditions are themselves universally quantified implications of atomic formulae. All our GFOL specifications of calculi with bindings in Section 3.1 will consist of Horn^2 formulae. As shown in the sequel, we can associate to these theories more natural and intuitive proof systems, which resemble almost identically (modulo syntactic sugar modification and some built-in type preservation properties) the corresponding original proof systems of the calculi.

For convenience, we assume that the language also contains the logical connectives $\top$ (zero-ary, corresponding to "true") and $\Rightarrow$ (binary, the logical implication) and that the Gentzen system $\mathcal{G}$ also contains the rules:

\[
\begin{array}{l}
\frac{\Gamma \varphi \Delta, \Gamma \psi \Delta}{\Gamma \varphi \psi \Delta} \quad \text{(Symm)} \\
\frac{\Gamma \varphi \Delta \psi \varphi \Delta}{\Gamma \varphi \Delta} \quad \text{(Trans)} \\
\frac{\pi(T_1, \ldots, T_n) = T'}{\pi(T_1, \ldots, T_n)} \quad \text{(Comp)} \\
\frac{T[x \leftarrow y]}{\Gamma \varphi \Delta} \quad \text{(Subst)}
\end{array}
\]

$\top$ and $\Rightarrow$ can be treated as derived connectives and the above rules can be derived for them, but we prefer to take them here as primitives. We take the convention that $\top$ is an atomic formula. Clearly, $\mathcal{G}$ is still sound and complete.

In what follows, $\pi$ denotes a tuple $\{x_1, \ldots, x_n\}$ of variables, $T$ a tuple $\{T_1, \ldots, T_n\}$ of terms, and, for a formula $\varphi, \varphi(T)$ indicates that $\varphi$ has all its free variables among $\{x_1, \ldots, x_n\}$, with $\varphi(T)$ denoting $\varphi[T_1/x_1, \ldots, T_n/x_n]$. Because variables are particular terms, we take the liberty to use the notation $\varphi(T)$ with two
different meanings, depending on the context: either to indicate that \( \varphi \) has its variables among \( \{y_1, \ldots, y_n\} \), case in which \( \varphi(y) \) is the same as \( \varphi \), or to denote the formula obtained from \( \varphi \) by substituting the variables \( \bar{\tau} \) assumed indicated previously by writing \( \varphi \) as \( \varphi(\bar{x}) \) with the variables \( \bar{y} \).

Let \( \text{HORN}^n \) be the GFOL formula given by the formula:

\[
\forall \bar{y}. \neg \bigwedge_{i=1}^{n} (a_i(\bar{x}, \bar{y}) \Rightarrow b_i(\bar{x}, \bar{y})) \Rightarrow c(\bar{y}) \quad (*)
\]

where \( a_i, b_i \) are atomic formulae. We call these \( \text{HORN}^n \)-formulas. When \( a_i \) is \( T \) we write only \( b_i(\bar{x}, \bar{y}) \) instead of \( a_i(\bar{x}, \bar{y}) \Rightarrow b_i(\bar{x}, \bar{y}) \), and call the formula extensional; if in addition \( \bar{y} \) has length 0, we obtain Horn formulas. When all \( b_i \)'s are \( T \) or \( n = 0 \), we write \( c(\bar{y}) \) instead of \( (*) \). \( \text{HORN}^n \), extensional, and Horn sentences are by definition universal closures of corresponding types of formulae. We identify formulae with their universal closures in theories. A theory \( E \) is called \( \text{HORN}^n \), extensional, or Horn if it consists of corresponding types of sentences. Besides including equational and Horn logics, \( \text{HORN}^n \) can define any \( \lambda \)-calculus (untyped, typed, polymorphic, etc.) as shown in Section 3.

We shall eventually only consider Horn consequences (in other words, sequents \( \Gamma \vdash d \) with \( \Gamma \) a finite set of atomic formulae and \( d \) an atomic formula) of \( \text{HORN}^n \) specifications, because only this type of consequences are usually relevant for \( \lambda \)-calculus. Moreover, all other \( \text{HORN}^n \) consequences can be deduced from these using (generic forms of) the Constant Lemma and the Deduction Theorem. We first consider slightly more general sequents, namely ones of the form \( \Gamma \supset \Delta \) with \( \Gamma \) and \( \Delta \) finite sets of atomic formulae.

Fix a \( \text{HORN}^n \) theory \( E \). Our goal next is to simplify and specialize with regards to \( E \) the Gentzen system \( \mathcal{G} \) discussed in the previous subsection. We first provide some immediate simplifications, based on the following remarks:

1. There is no need for the rules involving negation, because any provable negation-free sequent has a completed proof tree whose sequents do not contain negation;

2. There is no change in the strength of provability if we accept as axioms only those rules \( \Gamma \vdash a \) such that there exists an atomic sentence in \( \Gamma \cap \Delta \) instead of any sentence, atomic or not; this is because whenever a sequent \( \Gamma \vdash \Delta \) is such that there is a compound formula \( \varphi \) in \( \Gamma \cap \Delta \), then there is a non-axiom rule which can be applied backwards to it such that all its (one or two) upper sequents \( \Gamma \vdash \Delta' \) have a strict subformula of \( \varphi \) in \( \Gamma \cap \Delta' \).

We obtain the following Gentzen system parameterized by \( E \), denoted \( \mathcal{G}_E \), for entailing \( E \)-tautological sequents of the form \( \Gamma \vdash \Delta \), where \( \Gamma \) and \( \Delta \) are finite sets of atomic formulae:

\[
\Gamma \vdash \Delta \quad \text{if} \quad \Gamma \cap \Delta \neq \emptyset \quad \text{(Axiom)}
\]

\[
\Gamma_a(\bar{x}, \bar{T}) \vdash \Delta b_i(\bar{x}, \bar{T}) \text{ for } i = 1, \ldots, m \quad \Gamma(\bar{T}) \vdash \Delta \quad \text{(Inst'-e)}
\]

In the rule \( \text{(Inst'-e)} \) above (the "instance of e" rule), \( e \) is a sentence in \( E \) of the form \( (*) \) thus \( a_i, b_i, c \) are the atomic formulae that build \( e \), \( \bar{T} \) is a fresh tuple of variables with the same length as \( \bar{x} \), and \( \bar{T} \) is a tuple of terms with the same length as \( \bar{y} \). Here as well as in the other similar rules that we shall consider, we implicitly assume that if \( a_i \) is \( T \), then we do not add it to \( \Gamma \), and if \( b_i \) is \( T \), we do not add the sequent \( \Gamma a_i(\bar{x}, \bar{T}) \vdash \Delta b_i(\bar{x}, \bar{T}) \) to the hypotheses. Moreover, notice that if \( n = 0 \), the rule has only one hypothesis, \( \Gamma c(\bar{T}) \vdash \Delta \).

As opposed to \( \mathcal{G} \), the system \( \mathcal{G}_E \) is parameterized by a fixed theory \( E \); the axioms \( e \in E \) do not appear in sequents as such, but lay on the background, yielding "instance" rules (\( \text{Inst}'-e \)). Therefore \( \mathcal{G}_E \), and also all the other Gentzen systems defined later in this subsection, are specialized for \( e \) (a (rather but fixed) theory). According to the discussion above, the following holds:

**Proposition 3.** The Gentzen system \( \mathcal{G}_E \) is (sound and) complete for deducing \( E \)-tautological sequents \( \Gamma \vdash \Delta \), where \( \Gamma \) and \( \Delta \) are finite sets of atomic formulae.

The rule \( \text{(Inst'-e)} \) can be split into a simpler instance rule \( \text{(Inst-e)} \) and a rule \( \text{(Cut)} \) as below:

\[
\Gamma_a(\bar{x}, \bar{T}) \vdash \Delta b_i(\bar{x}, \bar{T}) \text{ for } i = 1, \ldots, m \quad \Gamma \vdash \Delta c(\bar{T}) \quad \text{(Inst-e)}
\]

\[
\Gamma \vdash \Delta \Delta_i, \quad \Gamma \vdash \Delta \
\text{(Cut)}
\]

Indeed, \( \text{(Inst'-e)} \) can immediately be simulated by \( \text{(Inst-e)} \) and \( \text{(Cut)} \); conversely, by completeness of \( \mathcal{G}_E \) and soundness of \( \text{(Inst-e)} \) and \( \text{(Cut)} \), any proof using \( \text{(Inst-e)} \) and \( \text{(Cut)} \) instead of \( \text{(Inst'-e)} \) can be performed in \( \mathcal{G}_E \). Let \( \mathcal{G}_E^{0} \) denote the system consisting of (Axiom), \( \text{(Inst-e)} \) and \( \text{(Cut)} \). We thus obtained the following:

**Proposition 4.** The Gentzen system \( \mathcal{G}_E^{0} \) is (sound and) complete for deducing \( E \)-tautological sequents \( \Gamma \vdash \Delta \), where \( \Gamma \) and \( \Delta \) are finite sets of atomic formulae.

Rules like the above \( \text{(Cut)} \) are usually undesirable for many reasons, among which their non-syntax-driven character due to the appearance of the interpolant d"out of nowhere". For us, this rule is undesirable because it increases the succinct of the sequents, thus not allowing one to only consider singleton antecedents leading to a simpler Gentzen system. Can \( \text{(Cut)} \) be always eliminated from \( \mathcal{G}_E^{0} \)? The answer is negative, as shown by the following "non-intuitionistic" counterexample: \( E \) is \( (a \Rightarrow b) \Rightarrow c \Rightarrow a \Rightarrow c \). Then \( \emptyset \vdash e \) is provable in \( \mathcal{G}_E^{0} \), \( \emptyset \vdash c \) follows by \( \text{(Cut)} \) from \( \emptyset \vdash e \), and \( a \Rightarrow c \) follows by \( \text{(Inst)} \) from \( (\emptyset \vdash e) \Rightarrow c \) from \( \emptyset \vdash a \Rightarrow b \), the latter being an (Axiom); \( a \Rightarrow b \) follows by \( \text{(Inst)} \) from \( \emptyset \vdash e \) from \( \emptyset \vdash a \Rightarrow b \), the latter being an (Axiom). (Note that indeed \( e \) is a semantic consequence of \( E \).) On the other hand, \( \mathcal{G}_E^{0} \) without \( \text{(Cut)} \) cannot prove \( \emptyset \vdash c \), as the reader can easily see. However, some theories \( E \) allow for the elimination of \( \text{(Cut)} \), as shown below. Let \( \mathcal{G}_E^{c} \) be \( \mathcal{G}_E^{0} \) without \( \text{(Cut)} \), and \( \mathcal{G}_E^{c} \) be \( \mathcal{G}_E^{c} \) with \( \text{(Cut)} \) replaced by:

\[
\Gamma \vdash d, \quad \Gamma \vdash \Delta \quad \Gamma \vdash \Delta \quad \text{(Simple-Cut)}
\]

Also consider the following family of rules:

\[
\Gamma(\bar{x}, \bar{T}) \vdash \Delta \quad \Gamma \vdash \Delta \quad \text{(Drop-'}(e, a)\text{)}
\]

where \( e \) is a sentence in \( E \) of the form \( (*) \), \( a(\bar{x}, \bar{y}) \) is one of the \( a_i \)'s, \( \bar{T} \) is a fresh tuple of variables, and \( \bar{T} \) is a tuple of terms of the same size as \( \bar{y} \). We next prove that the simpler-to-check closure of \( \mathcal{G}_E \) under the \( \text{(Drop-'}(e, a)\text{)} \) rules ensures its closure under \( \text{(Cut)} \), hence allows for elimination of the latter rule from \( \mathcal{G}_E^{0} \).

\footnote{Indeed, \( \text{(Cut)} \) could be eliminated if we considered an intuitionistic variant of GFOL; however, we do not get into this issue here.}
LEMMA 1. Assume that $G_E$ is closed under the rules (Drop-(e,a)) if $\Gamma \vdash \Delta_1 \cup \Delta_2$ is derivable in $G_E$, then either $\Gamma \vdash \Delta_1$ or $\Gamma \vdash \Delta_2$ is derivable in $G_E$.

LEMMA 2. $G_E$ and $\mathcal{G}_E$ are equivalent (i.e., (Simple-Cut) can be eliminated from $\mathcal{G}_E$).

PROPOSITION 5. If $G_E$ is closed under the rules (Drop-(e,a)), then it is also closed under (Cut), i.e., then $G_E$ is equivalent to $\mathcal{G}_E$.

Finally, we are ready to prove the completeness of a particularly simple Gentzen system, for the case of certain HORN theories. We let $K_E$ denote the system obtained from $G_E$ by restricting the succedents to be singletons, i.e., the following system:

- $\Gamma \vdash \Delta$ if $d \in \Gamma$ (Axiom)
- $\Gamma_0, \Delta(\pi, \overline{T}) \vdash \delta_i(\pi, \overline{T})$ for $i = 1, n$ (Inst-e)

THEOREM 3. If $G_E$ is closed under the rules (Drop-(e,a)), then $K_E$ is (sound and) complete for deducting $E$-tautological sequents $\Gamma \vdash \Delta$, with $\Gamma$ finite set of atomic formulae and $\Delta$ atomic formula.

Theorem 3 extends seamlessly to cope with equality, since the equality axioms $Eq_I$ are HORN theories sentences themselves. (As mentioned before, the equality rules are obtained following the same "in simplification" technique from the axioms $Eq_I$).

Let $G_E$ and $\mathcal{G}_E$ denote the systems $\mathcal{G}_E(Eq_I)$ and $\mathcal{G}_E(Eq_I)$, regarded over the language with equality. We obtain the following, as an immediate consequence of Theorem 3:

THEOREM 4. If $G_E$ is closed under the rules (Drop-(e,a)), then $K_E$ is (sound and) complete in deducting, in the logic with equality, $E$-tautological sequents of the form $\Gamma \vdash \Delta$, where $\Gamma$ is a finite set of sentences and $\Delta$ is a sentence.

We call a HORN theory $E$ amenable if it satisfies the hypothesis of Theorem 4 ($G_E$ closed under (Drop-(e,a))).

Notice that, for the purpose of entailing sequents $\Gamma \vdash \Delta$, $K_E$ and $K_E$ are highly optimized Gentzen systems - they function by directly applying the axioms of the theory $E$, and without the need of carrying over intermediate results in the succedents of the sequent. If $E$ is a Horn theory, one obtains the well-known Hilbert system for Horn (and in particular, equational) logic: indeed, the antecedent being fixed in deductions, it can be omitted and thus the Gentzen system becomes a Hilbert system, writing $\frac{\pi}{\overline{\overline{T}}}$ for $\frac{\pi}{\overline{\overline{T}}}$ and keeping an implicit account for the effect of the (Axiom) rule, as any Hilbert system does. Thus, we obtain a derivation of the completeness result for the Horn logic w.r.t. to its simple Hilbert system from the one of $\mathcal{F}_E$ w.r.t. its more involved Gentzen system. More generally, if $E$ is an extensional theory, then there are no (Drop-(e,a)) rules, hence $E$ is also trivially amenable.

COROLLARY 1. If $E$ is an extensional theory, then $K_E$ is complete for entailment $E$-tautological sequents of the form $\Gamma \vdash \Delta$, where $\Gamma$ is a finite set of sentences and $\Delta$ is a sentence.

The above results will be relevant for our calculi specifications, as an amenable Horn theory $E$ of a calculus will recover, in the system $K_E$, the represented calculi itself - see Section 3.2.

3. Specifying Calculi in HORN

We here define several $\lambda$-calculus as HORN theories; thus these calculi fall under HORN in a rather direct way, just like group, field and vector-space theories fall under equational logic. Most of the calculi below were taken from [20, 15, 10].

For untyped $\lambda$-calculus, see [3]; System F was introduced in [8, 23], ML-style polymorphism in [14], and Edinburgh LF in [11]. A polymorphic calculus with units of measurement is studied in [12].

3.1 Specifications of Calculi

Terms below are considered up to their $\alpha$-equivalence; substitution and free variables are standard, substitution acting on terms up to $\alpha$-equivalence, in a capture-avoiding fashion - the term syntax axioms check easily in each case. We work in GFOl with equality.

We next recall some standard $\lambda$ and FOL-like notational conventions that we rely on as well. Since our examples have two kinds of bindings (in terms and in formulæ), we state these conventions explicitly, to avoid further confusion: (1) Both terms and formulæ are considered up to $\alpha$-equivalence. When an expression like $\lambda x.X$ appears in an axiom, $\lambda x$ is assumed to bind any occurrence of $x$ in $X$ (note that $\lambda x.x$ is well-defined on $\alpha$-equivalence classes, because $x \equiv x$). $\lambda x.X$ implies $\lambda x.Y = \lambda x.Y'$ (2) Terms and formulæ- binding operators are assumed to bind as far as they can: thus $\lambda x.x + x$ should be read as $\lambda x.(x + x)$ and $\forall x.(\forall \phi \land \psi)$; the conjunction ($\land$) binds stronger than the implication ($\Rightarrow$). (3) Formulæ are identified with the sentences that are their universal closures, i.e., that quantify universally over all the free variables of the formulæ. Therefore, a more rigorous way to write $x = y \Leftrightarrow x + x$ is $\forall x.(x = y \Leftrightarrow x + x)$ and $\forall \phi.(\forall x.X) = X[y/x]$ where $\forall \phi$ is the tuple of all variables free in $(\forall x.X)$. (4) We use lower-case letters to denote variables and upper-case letters for terms. We call metavariables the symbols that we use to denote variables or terms (e.g., $x, y, X, Y$). Metavariables for variables and for terms are subject to different conventions. When two metavariables $\cdot x$ and $\cdot y$ appear in the same axiom, they denote some fixed, but distinct variables; therefore, e.g., $x = y \cdot x$ denotes a single GFOl sentence, with $x, y \in Var$ and $x$ distinct from $y$ (the choice of $x$ and $y$ is immaterial thanks to $\alpha$-equivalence of formulæ, since we assume an outer universal quantification, by convention 3). On the other hand, a metavariable $\cdot x$ denotes an arbitrary term, thus, e.g., $(\forall x.X)Y = X[y/x]$ is an axiom scheme representing a set of formulæ, one for each pair of terms $(X, Y)$ (for more on axiom schemes see Appendix A); moreover, when $X$ and $Y$ appear in the same axiom scheme (like above), they are not assumed distinct.

When do we use variables and when terms? We use them when there is no way to express the desired axiom as a single sentence. It turns out that axioms like the $\eta$-rule, classically stated as an axiom scheme $(E) = \lambda y.Ey$ with the side condition $y \notin FV(E)$, can be written as a single formulæ $x = \lambda y.x$ in GFOl; indeed, the fact that $x$ is "independent of $y$" is implicit in the way substitution is defined in a capture-avoiding fashion on $\alpha$-equivaleces of terms. On the other hand, the $\beta$-rule cannot be written as a single formulæ, being inherently an axiom scheme.

Untyped $\lambda$-calculus ($\lambda$). An unsorted theory in HORN, with no relations except equality; terms are $\alpha$-equivalence classes over Syntax Term ::= Var | Term Term | Var. Term; axioms are:

- $(\forall x.X = Y) \Rightarrow \lambda x.X = \lambda x.Y$ (3)
- $(\lambda x.X)x = X$ (4)
- $x = \lambda y.x y$ (5)

Remarks: (1) $(\forall x.X = Y)$ is a proper extensional sentence scheme, i.e., it is not equivalent to one in a simpler fragment of GFOl, like $\lambda$-;

2006/7/28
note that $x$ is $\forall$-bound in the left of $\Rightarrow$, and $\lambda$-bound in the right, in order to achieve the desired meaning: if $X$ and $Y$ are equal for an arbitrary $x$, then they are equal as "functions" on $x$. Extensional sentences appear to suffice for untyped calculi.

(2) The very simple syntax of our $\beta$-rule may seem strange at first sight. However, it has a good underlying intuition: a function $\lambda x.X(x)$ applied to a value $y$ yields $X(y)$ - in the latter $X(y)$, the value $x$ has substituted the variable, i.e., the formal parameter $x$. It is common practice to let the same symbol denote both the "formal" and the "actual" parameter. This brings no confusion, since in the equation $(\lambda x.X)x = X$ all occurrences of $x$ in the left $X$ are bound by $\lambda$, while the other occurrences of $x$ are bound by the outer universal quantifier of the equation. Thus if we substitute a term $Y$ for $x$ in this equation we get the more conventional beta-rule $(\lambda x.X)Y = Y$. One may replace the $(\beta)$ axiom by any of its $\beta$-equivalent forms $(\lambda x.X)y = X[y/x]$ and $(\lambda x.X)Y = X[Y/x]$; however, we find the current form of $(\beta)$ quite elegant and compact.

(3) Non-extensional $\lambda$-calculus is obtained by removing $(\eta)$.

(Simply-Typed) $\lambda$-Calculus ($\mathbb{T}\!\lambda\!\text{Const}$). Sorts $\text{type, data}$. Relation symbol $\text{typeOf} : \text{data} \times \text{type}$. $\text{Var} = \{\text{Var}_{\text{type}}, \text{Var}_{\text{Const}}\} = (\text{Var}_{\text{type}}, \text{Var}_{\text{Const}})$ and $\text{Term} = (\text{Term}_{\text{type}}, \text{Term}_{\text{Const}}) = (\text{Term}_{\text{type}}, \text{Term}_{\text{Const}})$, where:

\[
\text{DType} ::=: \text{DVar} | \text{DConst} | \text{Term} \rightarrow \text{Term}
\]

Remarks: (1) One usually so considers, besides the basic types such as $\text{nat}$ and $\text{bool}$ (which are elements of $\text{TConst}$), some data constants, such as $0$, $\text{succ}$, $\text{+, \&}$ if $\text{thenElseth}$, with their assigned types (e.g., $\text{typeOf}(+, \text{nat} \rightarrow \text{nat} \rightarrow \text{nat})$) and defining equations (e.g., $x + \text{succ}(y) = \text{succ}(x+y)$). To save space, we do not include these in our specification, nor shall we include such straightforward items in later specifications.

(2) The axioms (Abs) and (\xi) show that rules changing the typing context can be modelled using proper Horn\$^2$ sentences, in a semantically clean manner, for instance, [Abs] says that we can type $\lambda x : t.X$ to $t' \rightarrow t'$ whenever $X$ has type $t'$ for any value of its "argument" $x$ of type $t$. Syntactically, this approach simplifies the calculus, since it allows one to focus on the actual meaning of axioms rather than on low-level details such as how to deal with typing contexts, free or fresh variables, etc.

(3) We allow type variables and quantify them in formulas, but this fact alone does not bring polymorphism. Since we do not have abstraction over types, we can nevertheless use type variables to reason about types in general, which is not possible in simply-typed $\lambda$-calculus, so the Horn\$^2$ specification above is slightly more powerful than simply-typed $\lambda$-calculus.

(4) Standard definitions of simply-typed $\lambda$-calculus make use of typing contexts. Our Horn\$^2$ definition above does not make typing contexts explicit; they appear implicitly during the derivation process. A typing judgment of the form $x : t_1, \ldots, x_n : t_n \vdash x : T$ in the type-theoretic notation can be seen as syntactic sugar for $\forall x_1, \ldots, x_n. \text{typeOf}(x, t_1) \wedge \text{typeOf}(x_n, t_n) \Rightarrow \text{typeOf}(X, T)$. Because of the built-in equality axioms, we allow equations between data terms which are not necessarily well-typed. For example, $X = X$ holds regardless of whether $X$ is well-typed. However, the conditions in equation $\text{make sure that we cannot deduce any equation } X = Y, \text{ with } X \text{ well-typed and } Y \text{ non-well-typed. A}$

type of the form $x_1 : T_1, \ldots, x_n : T_n \vdash X = Y : T$ can be seen as syntactic sugar for $(\forall x_1, \ldots, x_n. \text{typeOf}(x, T_1) \Rightarrow \text{typeOf}(X,T) \wedge \text{typeOf}(Y, T) \wedge X = Y : T)$. (5) Since compatibility of $\text{typeOf}$ with equality is a built-in property of $\text{GOFOL}$ (and thus Horn\$^2$), our specification enjoys the "$\text{type preservation}" \text{ property (types are preserved by equalities) by default}. That this property can be proved as a theorem in simply-typed $\lambda$-calculus ensures the correctness of our specification. In general, a property like type preservation is seen as a test for a calculus to be sound. If one wants to actually specify the calculus without this property and then prove it as a theorem, then one should use Horn\$^2$ without equality and define := as an ordinary relation.

(6) Here and elsewhere, we state the typing conditions for equations as succinctly as possible; hence the above $(\beta)$-rule - its hypothesis, typeOf$(\lambda x : t.X, t') \Rightarrow t'$ is usually split into two: typeOf$(\lambda x : t.X, t) \Rightarrow t'$ and typeOf$(\lambda x : t.X, t) \Rightarrow t'$. The latter are equivalent with the former, via the $\text{[App]}$-rule, and in fact the compact form gives the essence of the hypothesis: the "$\text{problematic}\"$ term, $(\lambda x : t.X) t'$, is well-typed. Moreover, we specify typing hypotheses for the equations only if needed, i.e., only if the equated terms are susceptible of "$\text{ile-balancing}\"$ typing. This is in the case of $(\beta)$, $X$ is surely well-typed whenever $(\lambda x : t.X) t'$ is so; this way, with minimal precautions, we avoid allowing equalities between a well-typed term and a non-well-typed one. On the other hand, in the $(\xi)$-rule, provided $(\forall x. \text{typeOf}(x, t) \Rightarrow X) = Y$, holds: $\lambda x : t.X$ is well-typed if $\lambda x : t.X t'$ is so, hence there is no need for any typing hypotheses. This succinctness policy, with no spectacular results here, is useful for more complicated calculi, such as Type$\text{Type}$ (see below).

Typed $\lambda$-Calculus with Recursion ($\mathbb{T}\!\lambda\!\text{Rec}$). Extends $\mathbb{T}\!\lambda\!\text{Const}$. $\text{DType} ::=: \text{DVar} | \text{DType} | \text{DConst} | \text{Term} \rightarrow \text{Term}$

\[
\text{DType} := \text{DVar} | \text{DConst} | \text{Term} \rightarrow \text{Term}\]

System $F$ ($\mathbb{S}\!\mathbb{F}$). Extends $\mathbb{T}\!\lambda\!\text{Rec}$. $\text{Type} ::=: \text{DVar} | \text{DConst} | \text{Term} \rightarrow \text{Term}$

\[
\text{Type} := \text{DVar} | \text{DConst} | \text{Term} \rightarrow \text{Term}\]

Typed $\lambda$-Calculus with Subtyping ($\mathbb{T}\!\lambda\!\text{Sub}$). Extends $\mathbb{T}\!\lambda\!\text{Rec}$. Adds a new relation symbol, $\preceq : \text{type} \times \text{type}$.
Subtyping Isorecursive Types. Extend s both $T\lambda T$ and $T\lambda S$.

\[
(\forall t, t' (t \leq t') \Rightarrow T (t \leq t')) \Rightarrow \mu T \leq \mu T'.
\]

(Anchor)

Remark: The above axiom modeling the so-called “Amber rule” is another example of a proper HORR² formula, like the ones for $(\xi)$ and typing of abstraction. Again, the rule makes perfect intuitive sense in this form with universally quantified hypothesis, that avoids considering any typing context.

Typed $\lambda$-Calculus with Type Operators and Binding ($T\lambda\omega$).

Extends $T\lambda$ without $[\text{Abs}]$ (which needs to be modified). Adds new sort, kind, and new relation $\text{kindOf} : \text{type} \times \text{kind}$. Var = $[\text{VarVar}, \text{VarType}, \text{VarDelta}] = (K\text{Var}, T\text{Var}, D\text{Var})$ and Term = $([\text{TermVar}, \text{TermType}, \text{TermDelta}] = (K\text{Term}, T\text{Term}, D\text{Term})$, where:

- $K\text{Term}$ := $\star K\text{Var} [K\text{Term} -> K\text{Term}]
- T\text{Term} := \ldots | X T\text{Var} : K\text{Term} | T\text{Term}

$k, k'$ range over kind variables.

(kindOf(t, t') ∧ (∀x . typeOf(x, t) ⇒ typeOf(X, t')))

⇒ typeOf(λx . X, T, t) = (K-ext)

(kindOf(t, t') ∧ kindOf(t', t'') ⇒ kindOf(t -> t', t''))

⇒ typeOf(λλt . t, T) = (K-ext)

(kindOf(t, t') ⇒ kindOf(λx . k, T, t) = (K-ext)

Remark: Kinding is usually considered together with type polymorphism, to provide simple abbreviations such as $\forall P = λP.λλt'.(t' -> t'')$ as an alternative to parametric abstractions such as $\forall P t' = \Pi t' . t' -> t''$. In our logic, even the former abbreviations (usually kept in the meta-language-see [20]) can be stated at the logical level in a straightforward way:

$\forall P = λP.λλt'.(t' -> t'') = t'' \Rightarrow \ldots$

A polymorphic calculus with units of measurement ($T\lambda U\omega$).

Extends $T\lambda$. Adds a new sort, unit. Var = $[\text{VarVar}, \text{VarType}, \text{VarDelta}] = (U\text{Var}, U\text{Var}, U\text{Var})$ and Term = $([\text{TermVar}, \text{TermType}, \text{TermDelta}] = (U\text{Term}, U\text{Term}, U\text{Term})$, where:

- $U\text{Term} := \ldots | U\text{Var} | U\text{Term} \cdot U\text{Term} | U\text{Term}^{-1}
- T\text{Term} := \ldots | \text{UType} U\text{Term} | \text{UVar} U\text{Term}

$u, u', u''$ range over unit variables and $U$ over unit terms.

(typeOf(λλx . X, U, T, t) ⇒ typeOf(λu . X, u, U, t))

⇒ typeOf(λu . (u, u), T, u) = [U-ext]

(u . u) = u . u = u . (u' . u'')

u . 1 = u

u . u' = 1

(yu, x) = (λu . x, yu, x) = $\text{Assoc}$

λu . x)

Remarks: (1) To make the calculus meaningful, both $k$ and $l$ need to consider basic units of measure, such as $kg$ and meters as $U\text{Term}$. (2) $\text{UType}$ is a set of “quantitative” basic types, such as $\text{nat}$ or $\text{real}$, for which it makes sense to consider units of measurement. Thus, for instance, $\text{m}^2$ is the type of surfaces, while the type $\text{real}$ should be seen as the polymorphic type $\text{Pi} \text{real} u [12]$.

MI-Style Polymorphic $\lambda$-calculus ($\text{MLA}$). Extends $U\lambda X$ the imported sort is called $\text{data}$. Add's two sorts, type and $\text{typeScheme}$, with type $<\text{typeScheme}$ (thus we have an order-sorted setting) and two relations, $\text{typeOf} : \text{data} \times \text{typeScheme}$ and $\text{moreGeneral} : \text{typeScheme} \times \text{typeScheme}$. Var = $\text{Var} \text{VarVar}, \text{VarType}, \text{VarDelta} = (T\text{Var}, T\text{Var}, D\text{Var})$, with $T\text{Var} \leq T\text{Var} \leq T\text{Var}$ and Term = $\text{TermType}, \text{TermType}, \text{TermType}, \text{TermType}, \text{TermType}, \text{TermType}$, where:

- $T\text{Var}$ := $\text{VarVar} \| \text{TermVar} \| \text{TermVar} \| \text{TermVar} \| \text{TermVar} \| \text{TermVar} \| \text{TermVar}$

- $T\text{Var}$ := $\text{TermType} \| \text{TermType} \| \text{TermType} \| \text{TermType} \| \text{TermType} \| \text{TermType} \| \text{TermType}$

- $T\text{Var}$ := $\ldots | \text{let} \text{DVar} \| \text{DVar} \| \text{DVar} \| \text{DVar} \| \text{DVar} \| \text{DVar} \| \text{DVar}$

$x, y / X$ range over data/variables, $t, t', t'' / T$ over type variables/variables, and $s, s', s'' / S$ over type-scheme variables/variables.

Remarks: (1) The typing statement of the “let” construct uses type scheme variables, hence allows for polymorphism. (2) The relation of more general is more defined in a very simple fashion: the rule (MG) says that the type scheme $\Pi S$ is more general that $\Pi$ with any particular choice for the type $t$ appearing in $S$. Recall that the written formulae are meant to express their universal closures, in particular are meant to be universally quantified over $t$, hence any possible occurrence of $t$ in the second $S$ of $\text{moreGeneral}(\Pi S, S_s)$ is in the scope of an universal quantifier; this is precisely what “any particular choice” means. (See also the previous discussion on our $\beta$ axiom.) (3) The typing rule [Gen] says that if we can associate the type (scheme) $S$ to any type $t$, then we can regard the type $S$ of $x$ as polymorphic in $t$.

The Edinburgh LF Calculus with Dependent Types ($\text{LF}$). Sorts $\text{kind}$, $\text{typeFamily}$, object, and relations $\text{typeOf} : \text{object} \times \text{typeFamily}$ and $\text{kindOf} : \text{typeFamily} \times \text{kind}$. Var = $\text{VarVar}, \text{VarType}, \text{VarObject}$ = $\text{VarVar}, \text{VarType}, \text{VarObject}$ = $\text{VarVar}, \text{VarType}, \text{VarObject}$ = $\text{VarVar}, \text{VarType}, \text{VarObject}$ = $\text{VarVar}, \text{VarType}, \text{VarObject}$. Term := $\text{KindTerm} \| \text{TermType} \| \text{TermType} \| \text{TermType} \| \text{TermType} \| \text{TermType} \| \text{TermType}$

Remark: $\text{Var} \text{Var} \| \text{VarVar} \| \text{VarType} \| \text{VarObject} \| \text{VarVar} \| \text{VarType} \| \text{VarObject}$

Rem. (1) The typing statement of “let” uses scheme variables, hence allows for polymorphism. (2) The relation of more general is more defined in a very simple fashion: the rule (MG) says that the type scheme $\Pi S_s$ is more general that $\Pi S$ with any particular choice for the type $t$ appearing in $S$. Recall that the written formulae are meant to express their universal closures, in particular are meant to be universally quantified over $t$, hence any possible occurrence of $t$ in the second $S$ of $\text{moreGeneral}(\Pi S, S_s)$ is in the scope of an universal quantifier; this is precisely what “any particular choice” means. (See also the previous discussion on our $\beta$ axiom.) (3) The typing rule [Gen] says that if we can associate the type (scheme) $S$ to any type $t$, then we can regard the type $S$ of $x$ as polymorphic in $t$.
Remarks: (1) As usual, we omit the signature consisting of type-family- and object-constants of various kinds and types.

(2) Notice again how the process of “instantiating” generic/formal parameters by concrete/actual parameters is handled. For instance, in the axiom $\text{App-O}$ above, $y$ has a type $T'$ formally dependent on $x$ of type $T$, which means that when applied to an actual parameter of the required type, it will yield something of type $T$ with the formal parameter replaced by the actual one also referred to as $x$, but this time not covered by the $\text{El}$-binding.

(3) Especially in intricate situations like this, where kinds, types, and data are combined in various ways, we claim that a HORN$^2$ definition of a calculus is more readable and intuitive than a type-context-based one.

Type-Type $\lambda$-Calculus ($\mathcal{TT}\lambda$). Unsorted. One relation, typeOf:

$$\text{typeOf}(x, t, t', u) \quad \text{range over variables, and } X, T \text{ over terms. We write } x, X \text{ to refer what should be considered as data, } t, t', T \text{ when types are meant, and } u \text{ when the variable denotes either data or types. (These conventions are taken only for readability.)}$$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{typeOf}(\text{type of type})$</td>
<td>$\text{typeOf}(T, t, t', u) \land (\forall u. \text{typeOf}(u, t) \Rightarrow \text{typeOf}(T, u)) \Rightarrow \text{typeOf}(T, t, u)$</td>
</tr>
<tr>
<td>$\text{typeOf}(T, t, t', u)$</td>
<td>$\Rightarrow \text{typeOf}(T(T, u))$</td>
</tr>
<tr>
<td>$\text{typeOf}(\text{typeOf}(T, t, t', u) \land (\forall u. \text{typeOf}(u, t) \Rightarrow \text{typeOf}(T, u)) \Rightarrow \text{typeOf}(\text{typeOf}(T, t, u))$</td>
<td>$\Rightarrow \text{typeOf}(\text{typeOf}(T(\text{typeOf}(T, t, u)), t', u))$</td>
</tr>
<tr>
<td>$\text{typeOf}(\text{typeOf}(T, t, u), t') \land (\lambda u. \text{typeOf}(u, t') \Rightarrow \text{typeOf}(\text{typeOf}(T, t, u), u)) \Rightarrow \text{typeOf}(\lambda u. \text{typeOf}(u, t'), u)$</td>
<td>$\Rightarrow \text{typeOf}(\lambda u. \text{typeOf}(u, t'), u)$</td>
</tr>
<tr>
<td>$\text{typeOf}(\text{typeOf}(T, t, u), t') \land (\lambda u. \text{typeOf}(u, t') \Rightarrow \text{typeOf}(\text{typeOf}(T, t, u), u)) \Rightarrow \text{typeOf}(\lambda u. \text{typeOf}(u, t'), u)$</td>
<td>$\Rightarrow \text{typeOf}(\lambda u. \text{typeOf}(u, t'), u)$</td>
</tr>
</tbody>
</table>

3.2 Recovering the Original Calculi

The above HORN$^2$ specifications of calculi state axioms with a clear intuitive content. One can think of these axioms as either semantically, as properties of the desired GFO models, or syntactically, as constraints over the corresponding term syntax. But what is the precise relationship between our HORN$^2$ specifications and the traditional definitions of these calculi? While intuitively the relationship is very tight - they both follow the same intuitions about functions, typing, subtyping, etc. - this is obviously not precise enough. If we claim to have specified, or “defined”, for instance, System F in HORN$^2$, we should be able to show that, indeed, our specification conforms System F. In other definitional frameworks, this is usually performed by a translation between the original system and its specification, which then needs to be shown adequate. In our case, it turns out that such a translation is not necessary, since the Gentzen system $K_{\mathcal{S}}$ (see Section 2.6) associated to the HORN$^2$ specification of the calculus has already done this.

For example, the next table lists all the “instance” rules given by the axioms of the HORN$^2$ specification $S_{\mathcal{F}}$ of System F, i.e., all the rules of $K_{\mathcal{S}}$. Forth the sake of visual comparison with the traditional System-F definition, we shall use, in an infix style, the symbol “$\Rightarrow$” instead of “typeOf”. Recall that lower-case letters like $T$ and $x$, $y$ denote type and data variables, and corresponding upper-case letters denote type and data terms. (Appendix A further clarifies why the system in the table below is indeed the Gentzen system $K_{\mathcal{S}}$ induced by the theory $S_{\mathcal{F}}$.)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash x : T$</td>
<td>if $(x : T)$ is in $\Gamma$</td>
</tr>
<tr>
<td>$\Gamma \vdash (\lambda x : T : \mathcal{X}) : T$</td>
<td>$\Rightarrow \Gamma \vdash \lambda x : T : \mathcal{X}$</td>
</tr>
<tr>
<td>$\Gamma \vdash x : T$</td>
<td>$\Rightarrow \Gamma \vdash x : T$</td>
</tr>
<tr>
<td>$\Gamma \vdash x : T \Rightarrow \Gamma \vdash x : T$</td>
<td>$\Rightarrow \Gamma \vdash x : T$</td>
</tr>
<tr>
<td>$\Gamma \vdash x : T \Rightarrow \Gamma \vdash x : T$</td>
<td>$\Rightarrow \Gamma \vdash x : T$</td>
</tr>
<tr>
<td>$\Gamma \vdash x : T \Rightarrow \Gamma \vdash x : T$</td>
<td>$\Rightarrow \Gamma \vdash x : T$</td>
</tr>
<tr>
<td>$\Gamma \vdash x : T \Rightarrow \Gamma \vdash x : T$</td>
<td>$\Rightarrow \Gamma \vdash x : T$</td>
</tr>
<tr>
<td>$\Gamma \vdash \lambda x : T : \mathcal{X} : x : T$</td>
<td>$\Rightarrow \Gamma \vdash \lambda x : T : \mathcal{X} : x : T$</td>
</tr>
<tr>
<td>$\Gamma \vdash (\lambda x : T : \mathcal{X}) : x : T$</td>
<td>$\Rightarrow \Gamma \vdash (\lambda x : T : \mathcal{X}) : x : T$</td>
</tr>
<tr>
<td>$\Gamma \vdash (\lambda x : T : \mathcal{X}) : x : T$</td>
<td>$\Rightarrow \Gamma \vdash (\lambda x : T : \mathcal{X}) : x : T$</td>
</tr>
</tbody>
</table>

The side conditions of the above HORN$^2$-Gentzen-system rules turn out to be the familiar ones: at $\text{Inst-Abs}$ and $\text{Inst-}\exists x$, $x$ does not occur (free) in $\Gamma$; at $\text{Inst-}\exists x$ and $\text{Inst-}\exists x$, $x$ does not occur in $\Gamma$; at $\text{Inst-T-App}$, $x \notin FV(\mathcal{X})$; at $\text{Inst-T-App}$, $x \notin FV(\mathcal{X})$. Note that the equality rules are dualized (and labelled $\text{Inst-Refl}$, $\text{Inst-}\exists x$ etc.), since there are two sorts, data and type. We obtained indeed the System-F’s original calculus, modulo a few minor adjustments discussed next.

If we pick any other specification from Section 3.1 and the corresponding original calculus, we shall discover a similar situation as in the case of System F. The Gentzen system underlying a HORN$^2$ theory that defines a calculus coincides with the original-calculus’ definition, modulo the following three adjustments:

- Since we work in a logic with equality where relations such as typing are implicitly compatible with equality, type preservation holds by default; therefore, elimination of the (Inst-Comp.) rule above, i.e., a proof of type preservation, could be seen as an optimization of the HORN$^2$ Gentzen system. If one needs to define a calculus without type preservation, one clearly should
not use HORN$^2$ with equality, but could use HORN$^2$ without equality, treating “=” as an ordinary relation symbol.

- Because of the generality of the term syntax concept, one could not possibly know at the HORN$^2$ level how terms are actually built, e.g., using operations such as the application. Therefore, congruence rules like

$$\Gamma \vdash X = Y \quad \Gamma \vdash X' = Y' \quad \Gamma \vdash XX' = YY'$$  \hspace{1cm} \text{[Cong]}$$

are not available as tautological, i.e., built-in, rules in HORN$^2$. While such rules could indeed be enforced by axioms at the specification level, they are not necessary however, since their actual role would be to state compatibility with substitution, the latter being built in HORN$^2$ with equality - the (Inst-Subs) rule above. In conclusion, the Gentzen system for the HORN$^2$ theory corresponds to a substitution-based (rather than operation-congruence-based) version of the original calculus.\footnote{Actually, most calculi definitions either use directly a substitution rule, or prove it as a derived rule.}

- It is usually the case that some properties that are deducible from the HORN$^2$ definition are not of much interest to the original calculus; for example, in the HORN$^2$ definition of System $F$, one could state and prove that $t = t$ for all types $t$. Moreover, due to the fact that the antecedents $\Gamma \rightarrow \Delta$ in sequents $\Gamma \vdash d$ may contain not only trivial typings $x : T$, associating data values to type variables, but also more involved typings $X : T$ and equalities $X_1 = X_2$ or $T_1 = T_2$, one could also prove things like $(T_1 = T_2) \implies (T_1 = T_2)$ or $(X : T) \implies (T = T)$ for all $T$. However, note that all these extra deducible properties are trivial - indeed, one can easily see that any combination of the rules (Inst-T-Ref), (Inst-T-Symm), (Inst-T-Trans), and (Inst-T-Subs) may only entail trivial equalities between types. Moreover, this extra pseudo-information makes sense for the original calculus itself (and thus it is not “junk”), just that the calculus does not bother to consider it - indeed, if one asks whether type equality is transitive or reflexive, the answer should be positive. For other calculi, such as the Type Type $\lambda$-calculus and Edinburgh LF (see their theories $TTA$ and $LF$ in Subsection 3.1) where one needs to perform more involved deductions with types, the fact that HORN$^2$ allows by default in its sentences variables ranging over all syntactic categories becomes convenient.

Therefore, a HORN$^2$ specification $E$ brings in fact a higher-level notation for the calculus, because, by unfolding $E$ into its afferent Gentzen system $K_E$, one obtains the original calculus itself. Thus, one can think of $K_E$ as the “traditional definition” of the calculus. However, $K_E$ and $E$ have precisely the same expressiveness only if $E$ is amenable (Theorem 4). If $E$ is an extensional theory, as for untyped $\lambda$-calculus, amenability and thus completeness of $K_E$ come for free (Corollary 1), but for proper HORN$^2$ theories one needs to prove amenability. Thus, amenability corresponds to adequacy; note, however, a significant shift of focus: one needs not relate a HORN$^2$ theory to the “external” original definition of a calculus, as in the case of adequacy, but rather prove something only about the theory, namely, that its Gentzen system $K_E$ is closed under the drop rules. Fortunately, like other proofs of drop-rule closures in $\lambda$-calculus [see, e.g., [13]], such proofs tend to be routine.

\textbf{PROPOSITION 6. All the theories $E$ in Subsection 3.1 are amenable, and thus $K_E$ is complete for any of them.}

We believe that any $\lambda$-calculus specification in HORN$^2$, if stated naturally, is amenable. Indeed, amenability means closure of $\mathcal{G}_E$ under the drop rules associated to the proper HORN$^2$ axioms of the specifications. For any proper HORN$^2$ axiom useful for defining calculi that we can imagine, its condition-of-the-condition states that a piece of data $d$ is classified in a certain way, i.e., has a certain type or kind $T$, and thus its associated drop rule states that the information typeOf $(d, T)$ or kindOf $(d, T)$ with $x$ completely fresh, i.e., completely unrelated to the context of the sequent, cannot help deduction. E.g., provided $x$ is fresh for $\Gamma$ and $\Delta$, typeOf $(d, T) \implies \Delta$ is derivable only if $\Gamma \vdash T\Delta$ is so. The information that “something” (referred to as $x$) has a type $T$ (i.e., is inhabited by $x$), does not reveal anything new in a context $\Gamma \vdash \Delta$ that is insensitive to “that something” (i.e., that does not contain $x$).

4. Ad Hoc versus GFOL Models for $\lambda$-Calculi

GFOL provides models in a uniform manner to all its theories, in particular to all those in Section 3.1. We claim that this “general-purpose” GFOL semantics is as good/n as “domain-specific” semantics previously proposed for some of these calculi. Not only it yields a notion of model for the particular calculus that makes deduction complete, but this notion resembles closely the domain-specific ad hoc one. We exemplify this on untyped $\lambda$-calculus and on System F. For the former, the GFOL semantics coincides (up to a carrier-preserving bijection between classes of models) with its ad hoc, set-theoretical semantics from [3]; for the latter, GFOL provides a novel semantics, equivalent to the one given in [5]. Here equivalence means the existence of a bijection between elementary classes of models and will be brought by hand and forth mappings between the classes of models, which preserve and reflect satisfaction; an equivalence brings an isomorphism between the skeletons of two logics [18] - this situation is similar to the one of equivalence of categories [13].

4.1 Untyped $\lambda$-Calculus

The term syntax of Untyped Lambda Calculus (ULC) was already mentioned in Section 3.1, for the specification $\forall \lambda$. We let $\Lambda$ (instead of $\text{Term}$) denote the set of $\lambda$-terms over the countably infinite set $\text{Var}$ of variables, modulo $\alpha$-equivalence (we shall use the same notations as in [3]). In order to ease the presentation, we do not consider constants, but they could have been considered, as well without any further difficulties. (This will be true for System F as well.) We recall from [3] some model-theoretic notions developed around ULC. Let us call prestructure a triple $(\Lambda, \iota, \rho)$, where $\Lambda$ is a set, $\iota$ is a binary operation on $\Lambda$ (i.e., $(\iota, \iota)$ is an applicative structure), and for each $T \in \Lambda(\Lambda)$, $\rho(T) : \Lambda^{\lambda \to} \Lambda$ - $\Lambda$ denotes the set of $\lambda$-terms with constants in $\Lambda$, modulo $\alpha$-equivalence.

Given an equation $T_1 = T_2$ with $T_1$, $T_2$ $\lambda$-terms, one defines $A =_\lambda T_1 = T_2$ as usual, by interpreting $T_1$ as $T_2$ as being implicitly universally quantified - that is, by $A_T(\rho) = A_{T_2}(\rho)$ for all $\rho : \forall \lambda \to \Lambda$. For prestructures, we consider the following properties (where $a, b$ range over elements of $\Lambda$, $x$ over variables, $T, T_1, T_2$ over terms, $\rho, \rho'$ over evaluations, i.e., elements of $\Lambda^{\lambda \to}$):

- (P1) $A_x(\rho) = \rho(x)$;
- (P2) $A_{T_1}(\rho) = A_{T_2}(\rho)(\rho)$;
- (P3) If $\rho[V \rightarrow] = \rho'[V \rightarrow]$ then $A_T(\rho) = A_T(\rho')$;
- (P4) If $A_T(\rho(x \leftarrow a)) = A_T(\rho(x \leftarrow a))$ for all $a \in \Lambda$, then $A_{\lambda x . T}(\rho) = A_{\lambda x . T}(\rho')$;
- (P5) $A_{\lambda x . T}(\rho)(a) = A_T(\rho(x \leftarrow a))$;
- (P6) If $A_c(b) = c(b) = c$ for all $c \in \Lambda$, then $a = b$;
- (P7) $A_\lambda(\rho) = a$.  

\textbf{2006/7/28}
We next simplify the pre-structures slightly, by removing their redundant data given by parameterized terms. A simple pre-structure is a triple \((A, \langle \cdot \rangle, (\Lambda T)_{T \in \mathcal{A}})\) which satisfies all properties of a pre-structure, except (P7). Note that the distinction between simple pre-structures and pre-structures is that only terms in \(A\), and not in \(\Lambda(A)\), are considered. Hence the notion of satisfaction, defined for pre-structures only w.r.t. equations involving terms in \(A\), also makes sense for simple pre-structures. We shall only be interested in pre-structures verifying at least (P1)-(P4). In this case, simple pre-structures and pre-structures are essentially identical:

**Lemma 3.** The forgetful mapping \((A, \langle \cdot \rangle, (\Lambda T)_{T \in \mathcal{A}}) \to (A, \langle \cdot \rangle, (\Lambda T)_{T \in \mathcal{A}})\) is a bijection, preserving satisfaction and each of the properties (P3), (P6), between pre-structures verifying (P1)-(P4) and (P7) and simple pre-structures verifying (P1)-(P4).

This lemma allows us to work with the more manageable simple pre-structures, which we henceforth call "pre-structures", and forget about the more complicated ones, as well as about property (P7).

A syntactical \(\lambda\)-model \([1,2]\) (\(\lambda\)-model for short) is a pre-structure verifying (P1)-(P5). A \(\lambda\)-model is extensional if it verifies (P6).

We now come to the representation of ULC in \(HORN^2\). Consider the generic first-order language \(\forall\, \lambda, \, \theta\), whose models have therefore the form \((A, \langle (\Lambda T)_{T \in \mathcal{A}}\rangle)\). Rather than considering only the \(HORN^2\) theory \(U\lambda\), we prefer to play around with more combinations of \(HORN^2\) formulae (among which \(U\lambda\)), in order to allow a closer look at the relationship between the two types of models. We shall work with the following \(HORN^2\) formulae and schemes of formulae:

\[
(\lambda x.T)T' = T'[T/x] \quad (\lambda x.T)T = T[\lambda x.T/x] \quad (\forall x.T) = T[\lambda x.T/x] \quad (\forall x. T)T = T[\forall x. T/x] \quad (\forall x. T) = (T[\forall x. T/x]) \quad (\forall x. T) = (\forall x. T)[\lambda x.T/x]
\]

Note that we do not use the same notations as in Section 3.1. \(U\lambda\) here consists of \((\beta')\), \((\xi)\), and \((\eta')\).

**Lemma 4.** Each of the schemes of formulae \((\alpha')\), \((\eta)\), \((\eta')\) is semantically equivalent in \(GFO\) to its primed variant.

This lemma points out that side-conditioned axiom schemes like \((\eta)\) and \((\eta')\) are not necessary in \(GFO\), since they are expressible as single sentences, \((\eta')\) and \((\eta')\).

We define a correspondence between pre-structures verifying (P1)-(P4) and \(GFO\) models satisfying \(\xi\) as follows:

- Each pre-structure verifying (P1)-(P4) \(L = (A, \langle \cdot \rangle, (\Lambda T)_{T \in \mathcal{A}})\) is mapped to a \(GFO\) model \(L^0 = (A, (\Lambda T)_{T \in \mathcal{A}})\).
- Each \(GFO\) model \(M = (A, (\Lambda T)_{T \in \mathcal{A}})\) satisfying \(\xi\) is mapped to a pre-structure \(M^0 = (A, \langle \cdot \rangle, (\Lambda T)_{T \in \mathcal{A}})\), where \(\langle \cdot \rangle\) is defined by \(a(b) = A_{xy}(\rho)\), with \(\rho\) taking \(a\) to \(a\) and \(y\) to \(b\).

**Proposition 7.** The above two mappings are well defined and mutually inverse. Moreover, they preserve satisfaction and they can be restricted and costricted to:

(a) \(\lambda\)-models versus \(GFO\) models satisfying \((\xi), (\beta)\) (i.e., models of \(U\lambda\) without \((\eta')\));

(b) extensional \(\lambda\)-models versus \(GFO\) models satisfying \((\xi), (\beta), (\eta)\) (i.e., models of \(U\lambda\)).

Considering the above satisfaction-preserving bijections, one could say that the syntactic models coincide with the \(GFO\) models for the corresponding theories.

4.2 System \(F\)

The syntactic categories of System \(F\) \([8,23]\) are defined in Section 3.1 (as the two-typed term syntax of theory \(S^F\)). These coincide with those of System \(F\) as defined in the literature, just that we call "data terms" and "type terms" what are traditionally referred to as "terms" and "types". A typing context is a finite set \(\{x_1 : T_1, \ldots, x_n : T_n\}\) where \(x_i\)'s are data variables, \(T_i\)'s are type terms, and no data variable appears twice. Below \(x, y, X, Y\) range over data variables and terms, \(t, t', T, T'\) over type terms and terms, and \(\Gamma, \Delta, \Gamma'\) over typing contexts. The typing system for System \(F\) derives typing judgments, i.e., triples, \(\Gamma \vdash X : T\), and is given by the following rules:

\[
\begin{align*}
\Gamma \vdash x : T & \quad \text{if } (x:T) \in \Gamma \\
\Gamma \vdash (x:T) : X : T' & \quad \Gamma \vdash (\lambda x.T) : X \rightarrow T' \\
\Gamma \vdash \Pi X : T & \quad \Gamma \vdash X : T \\
\Gamma \vdash \Pi X : T & \quad \Gamma \vdash X : T' \\
\Gamma \vdash X : T & \quad \Gamma \vdash X : T' \\
\end{align*}
\]

At the rules [\(\text{SF-Add}\)] and [\(\text{SF-App}\)], \(\Gamma \vdash X : T\) is assumed to be a typing context with \((x:T) \notin \Gamma\), i.e., it is assumed that \(x\) is not free in the left of any pair in \(\Gamma\). At [\(\text{SF-T-Abs}\)], it is assumed that \(t\) is not free in the right of any pair in \(\Gamma\). We let \(\vdash_{\text{SF}} \Gamma \vdash X : T\) denote the fact that \(\Gamma \vdash X : T\) is deducible in the above system.

We relate System \(F\) and the \(HORN^2\) theory \(S^F\) first w.r.t. typing. For each typing context \(\Gamma = \{x_1 : T_1, \ldots, x_n : T_n\}\), we let \(\Gamma^0\) be the \(GFO\) formula \(type((x_1 : T_1) \land \cdots \land type(x_n : T_n))\).

**Proposition 8.** For all typing judgments \(\Gamma \vdash X : T\), \(\vdash_{\text{SF}} \Gamma \vdash X : T' \iff S^F \vdash \Gamma \vdash X : T' \iff type((X,T)) \in \Gamma^0\).

A Henkin model \(H\) for System \(F\) \([5,15]\) is a tuple \((T, F, \tau, \sigma)\), together with a pair \((\Pi \tau X \rightarrow T, \Pi \tau Y \rightarrow T)\), where \(\tau, \sigma \in T\) and \(\tau \neq \sigma\), and \(\Pi \tau X \rightarrow T, \Pi \tau Y \rightarrow T\) are \(S^F\) terms.

We keep a convention similar to that of Section 2.5, that \(\Gamma \vdash (x : T)\) is a notation for \(\Gamma \cup \{(x : T)\}\).
\[(7) \ H_T(x, \tau, \delta) = \delta(x)\; ;
\]
\[(8) \ H_T(x, \tau, \delta) = \text{App}_{H_T(x, \tau, \delta)}(H_T(x, T - t, \tau(\gamma), \delta))(H_T(T, - t, \gamma, \delta))\; ;
\]
\[(9) \ H_T(x, \tau, \delta) = \text{App}_{H_T(\tau, \delta)}(H_T(x, T - t, \tau(\gamma), \delta))(H_T(T, - t, \gamma, \delta))\; ;
\]
\[(10) \ H_{T \times T}(x, \tau, \delta) = \text{Dom}_{H_{T \times T}(x, \tau, \delta)}(H_{T \times T}(x, T - t, \tau(\gamma), \delta))(d) = H_{T \times T}(x, T - t, \tau(\gamma), \delta)(d)\; ;
\]
\[(11) \ H_{T \times T}(x, \tau, \delta) = \text{Dom}_{H_{T \times T}(x, \tau, \delta)}(H_{T \times T}(x, T - t, \tau(\gamma), \delta))(\tau) = H_{T \times T}(x, T - t, \tau(\gamma), \delta)\; ;
\]

Above, \(T\) denotes the sets of typing judgements and \(\bigcup \text{Dom}\) denotes \(\bigcup \text{Dom}(\gamma, \delta)\). \(t\) ranges over typing judgements, \(\sigma, \tau\), and \(d, d'\) over elements of \(T\) and \(\bigcup \text{Dom}\), \(\gamma\) and \(\delta\) over maps in \(T_{\text{var}}\) and \(\bigcup \text{Dom}(\gamma, \delta)\) denotes the function mapping each \(\tau\) to \(H_T(\tau(\gamma), \delta)\). We use slightly different notations than [15]; also, we include interpretations of terms and well-typed terms \(H_T\) and \(H_{T \times T}\) as part of the structure, while [15] equivalently asks that such interpretations exist and then proves them unique.

Satisfaction by Henkin models \(H\) of well-typed equations \(\Gamma \vdash x : T\) and \(\bigcup \text{Dom}\) is defined by \(H \models x : T\) if \(H(x) \in T\). To avoid technical details irrelevant here, we assume non-emptyness of types (without such an assumption, the Henkin models are not complete for System \(F\); but only if one considers richer languages - see [15]). Next we define mappings between System \(F\) Henkin models and GPOI models for \(S F\). Given three sets \(A, B, C\), a mapping \(f: A \times B \rightarrow C\) is called extensional if for all \(a, a' \in A\), if \(f(a,b) = f(a',b)\) for all \(b \in B\) then \(a = a'\). Below, the satisfaction relation for 1-, 2-, 3-, and 4- Henkin models is defined similarly to that for Henkin models.

The first transformations are:

- Consider each \(\text{App}_{\tau, \delta}\) as an injector mapping \(\text{Dom}_{r_{\tau, \delta}} \rightarrow \text{Dom}_{r_{\tau, \delta}}\), but as an extensional mapping \(\text{Dom}_{r_{\tau, \delta}} \times \text{Dom}_{r_{\tau, \delta}}\);

- Consider each \(\text{App}_{\tau, \delta}\) as a mapping \(\text{Dom}_{r_{\tau, \delta}} \rightarrow \bigcup \text{Dom}(\tau, \delta)\), but as an extensional mapping \(\text{Dom}_{r_{\tau, \delta}} \times \bigcup \text{Dom}(\tau, \delta)\), such that for each \((\tau, \delta) \in \text{Dom}_{r_{\tau, \delta}} \times \bigcup \text{Dom}(\tau, \delta)\), \(\text{App}_{\tau, \delta}(\tau, \delta) = \text{Dom}_{r_{\tau, \delta}}\);

- Assume \(\mathcal{F}\) consists only of mappings of the form \(\tau \rightarrow H_T(\tau(\gamma), \delta)\) for \(\gamma, \delta \in T_{\text{var}}\), \(t \in \text{Dom}_{r_{\tau, \delta}}\), only this kind of mappings are used in the Henkin model definition, and thus in the definition of satisfaction;

- Assume all \(\text{Dom}_{\tau, \delta}\) and \(\text{Dom}_{\tau, \delta}\), with \(\tau \neq \sigma\), mutually disjoint; this obviously does not affect the satisfaction relation.

We thus obtain the following equivalent models for System \(F\):

A 1-Henkin model \(H\) is a a tuple \((T, F, \rightarrow, \Pi, \text{App}, \text{Dom}, \rightarrow, \bigcup \text{Dom}(\gamma, \delta)\), \(\text{App}_{\tau, \delta}\), \(\tau, \delta \in T\), \(\Pi \rightarrow T\), \(\Pi \rightarrow T\), \(T \rightarrow T\), \(T \rightarrow T\), \(T \rightarrow T\)), where:

- \((a)\) \(\rightarrow\) \text{App}_{\gamma, \delta}(\text{Dom}_{r_{\gamma, \delta}}) \times \bigcup \text{Dom}(\gamma, \delta)\), where:

- \((b)\) \(\Pi \rightarrow T\),

- \((c)\) \(F \subseteq T^2, \{\tau \rightarrow H_T(\tau(\gamma), \delta) : \tau \in T_{\text{term}}, t \in T_{\text{var}}, \gamma \in T_{\text{var}}\}\),

- \((d)\) \(\text{App}_{\gamma, \delta} : \text{Dom}_{r_{\gamma, \delta}} \times \bigcup \text{Dom}(\gamma, \delta) \rightarrow T\),

for each \(\tau, \sigma \in T\),

- \((e)\) \(\text{App}_{\gamma, \delta} : \text{Dom}_{r_{\gamma, \delta}} \times \bigcup \text{Dom}(\gamma, \delta) \rightarrow T\),

for each \(\tau, \sigma \in T\),

- \((f)\) \(\text{App}_{\gamma, \delta} : \text{Dom}_{r_{\gamma, \delta}} \times \bigcup \text{Dom}(\gamma, \delta) \rightarrow T\),

for each \(\tau, \sigma \in T\),

- \((g)\) \(\text{App}_{\gamma, \delta} : \text{Dom}_{r_{\gamma, \delta}} \times \bigcup \text{Dom}(\gamma, \delta) \rightarrow T\),

for each \(\tau, \sigma \in T\),

- \((h)\) \(\text{App}_{\gamma, \delta} : \text{Dom}_{r_{\gamma, \delta}} \times \bigcup \text{Dom}(\gamma, \delta) \rightarrow T\),

for each \(\tau, \sigma \in T\),
Next we flatten the multi-typed domain \( \text{Dom} \) of 2-Henkin models. The flattening is based on the following:

- The multi-typing of \( \text{Dom} \) can be viewed as a relation \( \text{typeOf} \) between data and types;
- Due to the type-wise disjointness of the domain and the injectivity of \( \rightarrow \), the families of mappings \( (\mathcal{A}_{
abla}, \varepsilon)_{\varepsilon \in \tau} \) and \( (\mathcal{A}_{
abla}, \varepsilon)_{\varepsilon \in \tau} \) can be replaced by two mappings \( \mathcal{A} : \text{Dom} \times \text{Dom} \rightarrow \text{Ddom} \) and \( \mathcal{Tapp} : \text{Dom} \times \text{D} \rightarrow \text{Ddom} \), with postulating the necessary typing restrictions; since \( \mathcal{A} \) and \( \mathcal{Tapp} \) will be total functions, we allow them to be applied outside the areas designated \( (\mathcal{A}_{
abla}, \varepsilon)_{\varepsilon \in \tau} \) and \( (\mathcal{A}_{
abla}, \varepsilon)_{\varepsilon \in \tau} \), but this does not affect the satisfaction relation.

A 3-Henkin model \( H \) is a tuple \( (T, \mathcal{D}, \rightarrow, \mathcal{A}, \mathcal{Tapp}, \mathcal{I}_{\text{type}}, \mathcal{I}) \), \( \text{typeOf} \) together with a pair \( (\mathcal{H}_{\mathcal{T} \in \text{Term}}, (\mathcal{H}_{\mathcal{X}})_{\mathcal{X} \in \text{DOM}}) \), where:

1. \( \rightarrow : T \times T \rightarrow T \) is an injective mapping,
2. \( \mathcal{A} : \mathcal{D} \times \mathcal{D} \rightarrow \mathcal{D} \)
3. \( \mathcal{Tapp} : \mathcal{D} \times T \rightarrow \mathcal{D} \)
4. \( \mathcal{I}_{\text{type}} : \text{Const} \rightarrow T \)
5. \( \text{typeOf} \subseteq \mathcal{D} \times \mathcal{T} \)
6. \( \mathcal{H}_{\mathcal{T} \in \text{Term}} : \forall \mathcal{T} \in \text{Term}, \mathcal{H}_{\mathcal{T}}(\gamma) \rightarrow \mathcal{D} \) for each \( \mathcal{T} \in \text{Term} \),
7. \( \mathcal{H}_{\mathcal{X} \in \text{DOM}} : \forall \mathcal{X} \in \text{DOM}, \mathcal{H}_{\mathcal{X}}(\gamma) \rightarrow \mathcal{D} \) for all \( \mathcal{X} \in \text{DOM} \).

Let \( \mathcal{H} \) be a 3-Henkin model, \( \gamma : \mathcal{T} \rightarrow T \) and \( \delta : \mathcal{D} \rightarrow \mathcal{D} \). Then there for any two pairs \( (\Gamma, T) \) and \( (\Gamma', T') \), such that \( \Gamma \mapsto \Gamma' : T \) and \( \Gamma' \mapsto \Gamma' : T' \). Thus, \( \mathcal{H}_{\mathcal{T} \in \text{Term}} \) and \( \mathcal{H}_{\mathcal{T} \in \text{Term}} \) are defined on \( (\gamma, \delta) \), it holds that \( \mathcal{H}_{\mathcal{T} \in \text{Term}}(\gamma, \delta) \).

Based on the above definition, the fact that satisfaction is not affected by allowing interpretations of data terms that cannot type, we obtain some further equivalent models:

A 4-Henkin model \( H \) is a tuple \( (T, \mathcal{D}, \rightarrow, \mathcal{A}, \mathcal{Tapp}, \mathcal{I}_{\text{type}}, \mathcal{I}, \text{typeOf}) \) together with a pair \( (\mathcal{H}_{\mathcal{T} \in \text{Term}}, (\mathcal{H}_{\mathcal{X}})_{\mathcal{X} \in \text{DOM}}) \), where:

1. \( \rightarrow : T \times T \rightarrow T \) is an injective mapping,
2. \( \mathcal{A} : \mathcal{D} \times \mathcal{D} \rightarrow \mathcal{D} \)
3. \( \mathcal{Tapp} : \mathcal{D} \times T \rightarrow \mathcal{D} \)
4. \( \mathcal{I}_{\text{type}} : \text{Const} \rightarrow T \)
5. \( \text{typeOf} \subseteq \mathcal{D} \times \mathcal{T} \)
6. \( \mathcal{H}_{\mathcal{T} \in \text{Term}} : \forall \mathcal{T} \in \text{Term}, \mathcal{H}_{\mathcal{T}}(\gamma) \rightarrow \mathcal{D} \) for each \( \mathcal{T} \in \text{Term} \),
7. \( \mathcal{H}_{\mathcal{X} \in \text{DOM}} : \forall \mathcal{X} \in \text{DOM}, \mathcal{H}_{\mathcal{X}}(\gamma) \rightarrow \mathcal{D} \) for all \( \mathcal{X} \in \text{DOM} \).
Roughly, \( H^\# \) is obtained from \( H \) by throwing types on the sort type and all data of any type on sort data; the relation typeOf keeps the connection between well-typed terms and types as in \( H \). \( H^\# \) may also contain some additional "junk", of typeless (error) data.

**LEMMA 9.** The mappings \( M_X \) above are well-defined for all \( X \subset DExp \). More specifically, for each \( \gamma , \delta \), \( H^\#_{\Gamma ; X} \tau (\gamma , \delta) \) does not depend on the choice of \( \tau \) and \( T \), so long as \( \Gamma \vdash _{SF} \Gamma \vdash X : T \) and \( H^\#_{\Gamma ; X} \tau \) is defined on \( \gamma , \delta \). Moreover, \( M \) is indeed a GFOIL model and \( M \models _{GFOIL} SF \).

**GFOIL to Henkin:** For each GFOIL model \( M \) satisfying \( SF \), we define a Henkin model \( H = M^\# \) as follows:

(a) \( \Gamma = M_{\text{type}} \);
(b) \( F = \{ \tau \rightarrow M_{\Gamma} (\gamma [t \rightarrow \tau ] : \delta) : \gamma , \delta \text{,} T \text{ arbitrary} \} \);
(c) \( \tau \rightarrow _{\text{str}} \sigma \), if \( \tau \rightarrow _{\text{str}} \sigma \) is the typing rule for \( \sigma \); and \( \tau \rightarrow _{\text{str}} \sigma \) if \( \tau \rightarrow _{\text{str}} \sigma \) is the typing rule for \( \sigma \);
(d) \( M_{\Gamma} (\gamma [t \rightarrow \tau ] : \delta) \); and \( M_{\Gamma} (\gamma [t \rightarrow \tau ] : \delta) \) if \( \gamma [t \rightarrow \tau ] : \delta \) is well-formed.

**LEMMA 10.** All the above mappings are well-defined, i.e.:

- \( M_{\text{type}} (\gamma , \delta) \) does not depend on the choice of \( \gamma , \delta \), so long as \( \gamma (t) = \gamma (t') = \sigma \);
- if for some \( \tau , T', \tau , t', \gamma , \gamma' , \delta , \delta' \), the mappings \( \tau \rightarrow _{\text{str}} M_{\Gamma} (\gamma [t \rightarrow \tau ] : \delta) \) and \( \tau \rightarrow _{\text{str}} M_{\Gamma} (\gamma' [t \rightarrow \tau ] : \delta') \) coincide, then \( M_{\Gamma} (\gamma [t \rightarrow \tau ] : \delta) = M_{\Gamma} (\gamma' [t \rightarrow \tau ] : \delta') \);
- \( M_{\Gamma} (\gamma [t \rightarrow \tau ] : \delta) \) does not depend on the choice of \( \gamma , \delta \), so long as \( \delta (x) = d \) and \( \delta (y) = d' \);
- \( \delta (x) = d \) and \( \delta (y) = d' \), if \( \delta (x) = d \) and \( \delta (y) = d' \), then \( \delta (x) = d \) and \( \delta (y) = d' \).

**PROPOSITION 9.** Assume \( \vdash _{SF} \Gamma \vdash X : T \) and \( \vdash _{SF} \Gamma \vdash Y : T \). Then:

1. \( M_{\text{type}} (\gamma [t \rightarrow \tau ] : \delta) \) does not depend on the choice of \( \tau , \delta \);
2. \( M (\gamma [t \rightarrow \tau ] : \delta) \) does not depend on the choice of \( \gamma , \delta \);
3. \( M (\gamma [t \rightarrow \tau ] : \delta) \) does not depend on the choice of \( \gamma , \delta \);
4. \( M (\gamma [t \rightarrow \tau ] : \delta) \) does not depend on the choice of \( \gamma , \delta \);
5. \( M (\gamma [t \rightarrow \tau ] : \delta) \) does not depend on the choice of \( \gamma , \delta \);
6. \( M (\gamma [t \rightarrow \tau ] : \delta) \) does not depend on the choice of \( \gamma , \delta \);
7. \( M (\gamma [t \rightarrow \tau ] : \delta) \) does not depend on the choice of \( \gamma , \delta \);
8. \( M (\gamma [t \rightarrow \tau ] : \delta) \) does not depend on the choice of \( \gamma , \delta \);
9. \( M (\gamma [t \rightarrow \tau ] : \delta) \) does not depend on the choice of \( \gamma , \delta \);
10. \( M (\gamma [t \rightarrow \tau ] : \delta) \) does not depend on the choice of \( \gamma , \delta \).

**5. Related Work**

To properly distinguish GFOIL from other approaches, a discussion on encodings of formal systems is appropriate. Consider a formal system, consisting of a syntax and deduction rules, say \( \lambda \)-calculus, denoted \( \Lambda \). One can formally mimic the informal definition of \( \Lambda \) inside an axiomatic set theory, say ZF, by setting ZF-formulate.
\text{var}(x), \text{term}(x) and eq(x, y), which say "\(x\) is a (\(\lambda\)-calculus) variable", "\(x\) is a term", and "\(x, y\) are terms and their equality is deducible in \(\Lambda\), etc. Then any proof about \(\Lambda\), including proofs that certain facts are deducible in \(\Lambda\), can be carried on in side ZF. However, as far as the original system \(\Lambda\) is concerned, this ZF "definition" of \(\Lambda\) is not a definition, but an encoding; and to make the encoding rigorous, one should give a mapping from terms in \(\Lambda\) to their \(ZF\) representations.

Assume that the language of ZF has included, via a process of Skolemization, the constants \(\emptyset, succ, app\), and \(\text{lam}\), together with axioms stating their desired meaning: \(\text{succ}\) is the successor function on natural numbers, \(\text{app}\) is the function that takes terms \(t_1, t_2\) to \(t_1t_2\); etc., also assume that, both in \(\Lambda\) and in ZF, one uses natural numbers as \(\lambda\)-calculus variables. The encoding \(\text{Enc}: \lambda\text{-terms} \rightarrow ZF\)-terms can be defined by \(\text{Enc}(n) = succ; \ldots; \text{succ}(0)\) for \(n\) times, \(\text{Enc}(t_1t_2) = \text{app}(\text{Enc}(t_1), \text{Enc}(t_2))\) and \(\text{Enc}(\lambda x. t) = \lambda n(\text{Enc}(n), \text{Enc}(t))\). The encoding is faithful: for \(\lambda\text{-terms} t_1, t_2, t_1t_2 \downarrow \text{ZF} \vdash \text{Enc}(t_1) \equiv \text{Enc}(t_2)\). Thus ZF can be used as a device to prove equalities in \(\Lambda\). Yet, \(ZF\) does not generalize \(\Lambda\), nor is \(\Lambda\) a ZF-theory or anything similar. \(\Lambda\) was only encoded in ZF, so ZF regards it as an object about which it can reason.

What an encoding of a formal system cannot provide is a meaningful model-theoretic semantics for that system, because the encoding was only concerned with representing syntax and deduction. Indeed, given a presupposed model of ZF\(^8\) one cannot claim that it provides a model for the \(\lambda\)-calculus. A \(\lambda\) model would merely provide a universe where the set of \(\lambda\)-terms and the \(\lambda\) deductive system would dwell. It is true that any faithful encoding translates whatever complete model of the meta-theory (here ZF) has into models of the object-theory (here \(\Lambda\)); e.g., by the completeness of FOL where ZF is a theory, \(\vdash_t t_1t_2 \equiv \text{ZF} \vdash \text{Enc}(t_1) \equiv \text{Enc}(t_2)\) for all models \(M\) of ZF. But such a "semantics" would clearly be unacceptable for \(\Lambda\), as a ZF model contains representations of \(\lambda\)-terms, deduction rules, proofs, and the whole syntactic infrastructure of \(\Lambda\); moreover, this "semantics" is complete not because \(\lambda\) equalities are stating facts about these models, as would be desirable with a semantics, but simply because the ZF models contain structure that mimics \(\Lambda\) deductions.

The point of this discussion is independent of \(\Lambda\) and ZF; it applies to any encoding. In particular, whenever one uses a fixed calculus or logic, such as HOL, to encode any other calculi, one cannot claim to provide models for them. Thus, e.g., the fact that HOL admits complete Henkin models and also can encode virtually any formal system, does not mean that its Henkin models work as a uniform semantics for these systems.

\(^8\) The fact that ZF cannot be proven to have models is irrelevant for this discussion - we could have chosen weaker systems instead.

Higher-Order Abstract Syntax (HOAS). In HOAS\(^9\), the \(\lambda\) metacalculus, to encode various other formal systems, such as calculi or deductive systems of logics - let us refer to these as object systems. All the syntactic categories of an object system (terms, formulae, proofs, evaluation relations) become terms in the meta-calculus. Object-system binding are represented by \(\lambda\)-bindings in the meta-calculus. In particular, quite different binding operators in the object system (such as \(\lambda\)-abstraction and universal quantification) are represented uniformly, using a single binding operator - the meta-level \(\lambda\)-abstraction. Consequently, all object-calculus axiom schemes become simple axioms in the meta-calculus, their "semantics" is then automatically handled by a built-in meta-calculus scheme, which does not appear in the specification itself. (To the contrary, GFO\(2\) specifications need to use axiom schemes in order to capture directly the axiom schemes of the defined calculi - see Appendix A.) HOAS' syntactically uniform representation, though very useful for proof-theoretic aspects, cannot be sensitive to the model-theoretic aspects of the object system. In particular, some prescriptive models of the meta-calculus do not provide models for the object calculus; and indeed, HOAS does not attempt to provide such models, being concerned mainly with proof-theoretic adequacy. Take for instance a representation of \(\lambda\)-calculus in Edinburgh LF - such a representation would define the type of terms, that of equations, and the dependent types of proofs. Thus a prescriptive model of this theory (consisting here only of constant declarations) stated in the dependent-type calculus of LF, would be dwelled by elements called terms, equations, and proofs, hence it would be far from being an appropriate model of \(\lambda\)-calculus; the latter should not provide any interpretations for proofs as element in the model (though, as argued in [2], an LF representation, if chosen in a "denotational" manner, could suggest a model).

In conclusion, unlike GFO\(2\), HOAS falls into the encoding-based approaches, even though the encoded object has an extra affinity with its encoding, as they share the same bindings. On the more operational side, one can define any meta-calculi used in HOAS (such as untyped \(\lambda\)-calculus and the Edinburgh LF \(\lambda\)-calculus with dependent types) as HOAR\(2\) theories and use these theories to represent object systems the same way the original calculi do, but this would not be a proper use of HOAR\(2\) - as mentioned, its technique for defining calculus is different from HOAS. Also, a framework such as LF could easily encode any instance of GFO\(2\) and its theories (pretty much like it encodes FOL), whenever the underlying term syntax is encodable.

Nominal Logic (NL)\(^{21}\) is a first-order logic that deals with abstract syntax by means of names, which can be bound in terms just like \(\lambda\)-calculus variables, but they are not variables; names are semantical entities, having syntax-independent meaning and being addressed explicitly by the freshness relation and the swapping operator. GFO\(2) resembles NL in that both are parameterized by a notion of a term (substitution-based for GFO\(2\), binding-based for NL) and both are first-order. However, GFO\(2) differs from NL in several aspects: (1) The NL approach to calculus definitions is even more encoding-based than HOAS, as it explicitly defines substitution and freshness inside its theories. (2) The NL models contain semantic support for substitution, in its more amenable form of swapping/permuation of names; GFO\(2) models are required to interpret syntax in a way that is substitution-consistent, but not to interpret substitution as such. (3) NL is not a complete logic, due to the second-order nature of the restrictions on its models.

Explicitly Closed Families and Binding Algebras. Structures consisting of explicitly closed families and functionals (ECFF) were introduced in [1] and studied as binding algebras in [25]. An ECFF consists of a set \(A\) together with a family of operations on \(A\) and a family of functionals (mappings between functions), such that the set of functions on \(A\) is closed under the functional-based polynomial combinators. The notion of term in this framework forms a term syntax in our sense, and ECFFs are particular cases of models...
in our sense. However, ECFF have a more restricted use than our GFOL models, since they make the commitment that bindings always define functions. To the contrary, the interpretation of terms in GFOL models has a loose character; it does not make any commitment other than what is prescribed by the axioms of the theories. As a consequence, calculi involving types of bindings other than "functional" find a direct representation in GFOL, while in ECFF they could only hope for a functional encoding.

Substitution Algebras [6] treat substitution abstrusely like we do, but require models to account for substitution in a direct way. Therefore they need to work in a preshadow topos different from Set as the underlying universe, in order for elements in models to be families of items, sensitive to the change of context/environment. (Thus they propose a solution for "semantic substitution" different than Nominal Logic.) While substitution algebras are related to our term syntaxes, models differ in that we do not require them to have built-in "abstract syntax" on their carrier sets, but rather to be able to provide interpretations for all syntactic features. As opposed to substitution algebras, HORN$^2$ is directly applicable to typed calculi, with typing judgements that change the typing context being captured by HORN$^2$ formulae.

6. Concluding remarks
We defined a generic first-order logic, GFOL, in which terms are axiomatized by common properties of their free variables and substitution, together with a complete deduction system. A fragment of GFOL with sentences more general than Horn, called HORN$^2$, was shown to admit a more effective complete Gentzen system. Several $\lambda$-calculi were defined as theories in HORN$^2$, following a "higher-level" view that allows one to focus on the specific aspects of the calculus rather than on syntactic or tactical details. This higher-level view brings a complete semantics to the specified calculus in a natural, meaningful, and uniform way.

The kind of semantics that a calculus receives via GFOL is usually called loose, or logical semantics. This means that a calculus does not receive a denotation in a fixed model, but is rather regarded as stating axioms about a whole class of models, just like group theory states axioms that are to hold in a class of models called groups. Completeness of a loose semantics means that a statement in the language of the calculus is derivable in the calculus iff it is true in all models. Thus one might be tempted to say that loose models capture faithfully what the calculus can prove. However, a loose semantics, while convenient for many purposes, is by no means the end of the "semantic story" of the calculus, as sometimes a calculus hides inside more than its deductive system can prove - hence the need for a denotation that would provide further insight into what the calculus actually "means", in particular would discover desired properties that were implicit in the calculus in a way more subtle than by bare deduction consequence.

Regarding a presumptive denotational-semantics methodology developed on top of HORN$^2$, this was not the concern of the present paper, but seems like a promising subject for future research. The key to this would be the study of appropriate free and initial models for certain HORN$^2$ theories - these models would constitute the desired denotation in a similar style with initial models constituting the "desired denotation" for a first-order data-type specification. For example, extensional theories can be shown to admit free models, as well as initial reachable models, along the lines of the interesting results from [16]. To faithfully capture higher-order denotations, partiality might be necessary as a first-class citizen in the models, via incomplete interpretations of terms into models.

Finally, it would be challenging to also study computational, i.e., operational, aspects that could be extracted from the HORN$^2$ theories: can equations and some relations such as typeOf or more-

General be "executed", e.g., via rewriting or some form of general-purpose logic programming technique, and thus obtain a calculus-independent operational semantics methodology?

References
A. The Use of Axiom Schemes

Due to the presence of binding, stating properties such as the \( \beta \)-reduction amounts to giving an infinite number of axioms that follow a certain pattern, i.e., providing an axiom scheme. This situation, present in all lambda calculi, persists in \( \text{HORN}^2 \) theories as well. While the use of axiom schemes (and therefore of infinite recursive axiomatizations rather than finite ones) is indeed a drawback, we consider this that this is the necessary side-effect of the direct way in which \( \text{HORN}^2 \) approaches the definitions of calculi - not by encoding as in HOAS, but by instantiating directly to the desired calculus. In this sense, \( \text{HORN}^2 \) is not a "logical framework" for calculi definitions (as Edinburgh LF is), but rather a generalization for all these calculi (according to Section 3). Moreover, axiom schemes, so long as they rely on side-condition-free term patterns like all the axiom schemes of \( \text{HORN}^2 \) specifications tend to do, are not really a burden for deduction, as we argue below.

Indeed, the \( \text{HORN}^2 \)-Gentzen systems discussed in Section 2.6 use axiom instances obtained from the axioms by substituting variables for terms. Now consider a term-based axiom scheme, such as (T-\( \beta \)) for the System-\( F \) specification \( SF \) in Section 3:  

\[
\text{typeOf}(\lambda t . X , t') \Rightarrow (\lambda t . X) t = X, \quad \text{where } t , t' \text{ denote type variables and } X \text{ an arbitrary type term}.
\]

Recall that the axiom is implicitly universally quantified over all its free variables. For using it inside the \( \text{HORN}^2 \)-Gentzen system, one proceeds as follows: first one picks a term \( X \), obtaining an instance of the axiom scheme; then one substitutes by terms all the free variables in this instance; finally, one places the substituted instance in an appropriate context in the sequents appearing in deduction rules. Note that, according to the above recipe, one needs to instantiate the axiom scheme twice: first by replacing the term metavariable \( X \) with an actual term, and then by substituting the resulting free variables with terms. However, following the usual practice of \( \lambda \)-calculus that do not discriminate term metavariables over variable metavariables, we can collapse these two instantiations into one, building on the intuitively clear (and easily checkable) fact that after replacing a term metavariable such as \( X \) with an actual term, we do not need to further substitute the free variables of this term, as we can take the already substituted term directly. For example, in the \( \text{HORN}^2 \)-Gentzen system \( KG = \) for amenable theories, (T-\( \beta \)) yields the following deduction rule, where \( T, T' \) stand for an arbitrary type terms:

\[
\Gamma \vdash \text{typeOf}(\lambda X . T) \Rightarrow (\lambda X . T) t = X[t/T] \quad \text{(Inst-T-\( \beta \))}
\]

t above is substituted by \( T \) in the right of the succedent of the lower sequent only, since all the other occurrences of \( t \) are term-bound. We obtained the familiar (\( \beta \))-rule for types. When transforming the axioms into their associated "instance" rules, side conditions may arise, as shown by the instance of \( SF \)'s [Abs]:

\[
\begin{array}{c}
\text{typeOf}(x , T) \Rightarrow \text{typeOf}(X , T') \\
\text{typeOf}(\lambda X . T , T) \Rightarrow X[t/T] \\
\end{array}
\quad \text{[Inst-Abs]}
\]

where \( x \) should be a data variable fresh w.r.t. \( \Gamma \). Again we obtain the familiar typing rule for abstraction, with the familiar side-condition that \( x \) not be in \( \Gamma \).

B. The Typing Relations

In the specifications of Section 3, typing was defined by means of a relation \( \text{typeOf} \) between the universe of data and the universe of types. We also spoke about \textit{data terms} and \textit{type terms}. This approach may look non-standard to the readers used to view types as items that are assigned to terms, at parsing or type-checking time, and not to data. We explain it next. Our specification methodology considers types, just like data, to be \textit{semantic items}, populating the models. In this view, it is not the case that terms have types, but data items have types - and while \textit{inferring} types for various data in \textit{all models} one indeed uses terms (data terms and type terms), in a process that looks just like the traditional one of assigning types to terms. The one who wishes to regard typing purely syntactically can define the "syntactic typing relation" between terms (i.e., in our terminology, data terms) and terms (i.e., type terms) as the following meta-relation: \( X \) has type \( T \) in the environment \( \Gamma \) iff the theory infers \( \Gamma \vdash X : T \), i.e., if \( \Gamma \vdash X : T \) is a sequential w.r.t. the theory - this relation is indeed the expected least relation closed under some rules. But again, we regard typing judgements, just like the inferred equality between terms, loosely, as sentences that hold in all models. Just as much as we do not need (and makes no sense) to state in the theory that the equality relation is "the least one" closed under some rules, there is no need to make such meta-statements about typing either, since they hold by the very nature of deduction in any Gentzen system.

C. Sorts versus Types

\( \text{HORN}^2 \) is a many-sorted logic, i.e., has a many-sorted variant. When instantiating it to \( \lambda \)-calculus, depending on the complexity of the calculus and sometimes on mere taste, one may choose between two alternatives:

- To represent the types in the calculus as \( \text{HORN}^2 \) sorts. This way, typing is reguarded \textit{syntactically}, as a parsing, and thus meta-level, issue of \( \text{HORN}^2 \).
- To view types \textit{semantically}, as inhabitants of some universe of types, and define the typing relation within the logic. This way, sorts are reserved for more general classifications of the semantic items, e.g. into \textit{data} and \textit{types}.

The second approach has the advantage of being more flexible, and thus covers the cases of more complex calculi with non-trivial typing and with higher-level classifiers such as kinds. We have pursued this approach in our specifications from Section 3. On the other hand, when possible, the first approach simplifies the structure of the formulæ, since well-typed-ness need not be stated as an extnt condition - indeed, extensional theories seem to suffice here, bringing, via Corollary 1, faithfulness of the representation for free, i.e., without the need to prove closure under the drop rules. We exemplify this first approach on an infinitely-sorted \( \text{HORN}^2 \) definition of simply-typed \( \lambda \)-calculus, alternative to \( TA \) of Section 3.

\textbf{(Simply-)Typed \( \lambda \)-Calculus - Second Version (TA')}

Let \( B \) be a set, of \textit{basic types}.

- The sorts are \( \text{Sort} := B \mid \text{Sort} \to \text{Sort} \) - let us call the sort "types".
- Recall that for each \( t , t' \in \text{Sort} \) \( \text{Var}_t \cap \text{Var}_{t'} = \emptyset \).
- For each \( t , t' \in \text{Sort} \) and \( b \in B \),
  \[
  \text{Term}_{b} := \text{Var}_b \upharpoonright \text{Term}_{b} , \text{Term}_{b} \subseteq B \}
  \[
  \text{Term}_{t \to t'} := \text{Var}_{t \to t'} \cap \text{Term}_{t} \to \text{Term}_{t'} \cap \text{Var}_{t} , \text{Term}_{t} \forall \text{t} , \text{Term}_{t'} \forall \text{t} , \text{Term}_{t} \forall \text{t} \}
  \[
  \text{No relation symbol \( \upharpoonright \)}
\]

We let \( t , t' , s \) range over types, \( x , y \) range over variables of type \( t \), and \( X_t \), \( Y_t \) range over terms of type \( t \) - note that here \text{"term"} means
"well-typed term". The theory is given below:

\[
\begin{align*}
(\forall x_1. X_2 = Y_2) & \Rightarrow \lambda x_1. X_3 = \lambda x_1. Y_4 & (Q) \\
(\forall x. x_1. X_2) & = X_3 & (\beta) \\
\lambda x_1. X_2 & = Y_4 & (\eta)
\end{align*}
\]

The Gentzen system \(K_{\forall}^{\lambda} \Pi^I\) induced by this extensional theory is
the following:

\[
\begin{align*}
\frac{X_1 = X_1}{\text{ Inst-Ref}} & \\
\frac{Y_1 = X_1}{\text{ Inst-Sym}} & \\
\frac{X_1 = Y_1, Y_1 = Z_1}{\text{ Inst-Trans}} & \\
\frac{X_1 = Y_1}{\text{ Inst-Subst}} & \\
\frac{X_2 \backsimeq X_3}{\text{ Inst-\(\subseteq\)}} & \\
\frac{X_1 = Y_1}{\text{ Inst-\(\eta\)}} & \\
\frac{X_1 = Y_1, X_2 = X_3, Y_4}{\text{ Inst-\(\xi\)}} & \\
\frac{\lambda x_1. X_2, Y_1 = X_3, x_1 \rightarrow Y_4}{\text{ Inst-\(\lambda\)}} & \\
\frac{X_1 \rightarrow \lambda y. X_1 \rightarrow Y_1}{\text{ Inst-\(\}(\lambda)\)}} & \\
\end{align*}
\]

The above "low-level" implementation of \(\forall \Pi^I\) is, modulo a
replacement of the congruence rules with a substitution rule, the typed
\(\lambda\)-calculus itself. And its completeness holds immediately by
Corollary 1, since \(\forall \Pi^I\) is an extensional theory. To the contrary,
the previous Gentzen system for the theory \(\forall \lambda\Pi^I\) needs a little
work to be shown complete.

It is worth mentioning that the GFOL models of \(\forall \Pi^I\) are precisely
the Henkin models, also called frame models, of simply-typed
\(\lambda\)-calculus (see [15]).

D. Meta-Reasoning and Inductive Reasoning

Meta-reasoning about a calculus is not possible in its Horn\(^2\) speci-
cification just as much as it is not possible in the calculus itself.
Indeed, as already discussed, a Horn\(^2\) becomes the specified cal-
culus. In particular, one cannot show in Horn\(^2\) that a calculus is
confluent or terminating when equations are regarded as rewrite
rules, neither that a programming language is deterministic. Even
apart from their meta-theoretic aspect, these properties cannot be
captured in an axiomatic approach like ours, where "evaluation"
of a program to a value is only implicit in the deductive system, and
not explicit as in an SOS or other forms of operational semantics.

Even if meta-reasoning is not available in the Horn\(^2\) specifi-
cations (and not meant to be), desired induction principles may be
stated as sentences in an inductive version of GFOL. For instance,
here is how structural induction would look for the theory \(U\lambda\) of
Section 3 (that specifies untyped \(\lambda\)-calculus):

\[
\frac{((\forall X \in \text{Form} \forall y \in FV(X) \setminus \{x\}, (\forall x. \phi(x)) \Rightarrow \phi(x, x)) \wedge
\quad \psi(x, y, \phi(x), \phi(y)) \Rightarrow \phi(y)) \Rightarrow \forall x. \phi(x)}{(\text{Ind})}
\]

Above, \(\phi\) is an arbitrary GFOL formula with a pointed free variable.
Note that this inductive axiom is more manageable than it looks.
It says that if one is able to prove \((\forall x. \phi(x)) \Rightarrow \phi(x, X)\)
for any arbitrary term \(X\) and also to prove \(\phi(x) \wedge \phi(y) \Rightarrow \phi(x, y)\),
then one can infer \(\forall x. \phi(x)\).

Finally, desired existential properties, most notably existence of
fixed points, are expressible and potentially provable in GFOL.
For instance, a consequence in GFOL of \(U\lambda\) is: \(\forall x. \exists y. x = y\),
meaning that every item has, regarded as a function, a fixed point.

Existence of "programs" performing desired tasks (i.e., conforming
certain "specification" stated as formulae) may also be expressed
and proved directly in GFOL - a handy example is the existence
of fixpoint operators: \(\exists y. \forall x. x = y \Rightarrow y\), a stronger version
of the existence of fixed points.

E. Proofs

Here we give proof sketches for the results stated in the
paper.

PROPOSITION 1. The following hold:

1. If \(x \notin FV(T)\) implies \(T[T/x] = T\);
2. \(y[T/x] = T\) if \(y = x\) and \(y[T/x] = y\) otherwise;
3. \(FV(T[T/x]) = FV(T) \setminus \{x\} \cup FV(T'')\);
4. \(y[T/x] = T\) if \(y \notin FV(T)\);
5. \(T[y/x] = T\) if \(y \notin FV(T)\).

Proof: We shall tacitly use properties (1)-(6) in Definition 1.
1. Assume \(x \notin FV(T)\). Since \([T/x] FV(y) = 1_{\text{var}}[FV(T)]\), we
obtain \(T[T/x] = \text{Subst}(T, 1_{\text{var}}) = T\).
2. If \(y = x\) then \(y[T/x] = \text{Subst}(x, x[T/x]) = T\) if \(x \notin T\).
3. \(y \notin FV(T)\), \text{Subst}(y, 1_{\text{var}}) = y\).
4. \(y[T'/x] = FV(\text{Subst}(T, T'/x)) = \bigcup\{FV(T'/x) : y \in FV(T')\} = \bigcup\{FV(T'/x) : x \in FV(T') \cup FV(T'/x) : y \in FV(T') \cup FV(T) \setminus \{x\}\}.

Above, we also applied point (2) of the current proposition.
4. We have that \(T[y/x] = T\) if \(y \notin FV(T)\), then \(\exists y \in Var\),
so we have that:

\[
\begin{aligned}
\sqrt{(y/x)}^{(y/x)}(u) &= \text{Subst}_{(y/x)}(y) \cup \{u, y\} \\
&= \{u, y\} \cup \{u, y\} \\
&= \{u, y\} \cup \{u, y\}
\end{aligned}
\]

Hence, since \(y \notin FV(T)\), it follows that \(FV(T) = \text{Subst}(T, (y/x)) = \text{Subst}(T, (y/x))\).
5. It follows by point (4), since \(\text{Subst}(T, (y/x)) = T\).

PROPOSITION 2. The following hold:

1. If \(\rho \in FV(\phi)\), then \(\rho \in A_{\phi} \iff \rho' \in A_{\phi'}\);
2. \(\rho \in A_{\text{Subst}(\phi, \theta)} \iff A_{\phi} \in A_{\phi'}\);
3. \(\phi \equiv \psi\) implies \(A_{\phi} = A_{\psi}\);
4. \(\phi \equiv \psi\) implies \(FV(\phi) = FV(\psi)\);
5. \(\equiv_\alpha\) is an equivalence;
6. \(\phi \equiv_\alpha \text{Subst}(\phi, 1_{\text{var}})\);
7. \(y \notin FV(\phi)\) implies \(\phi(y/x)[y/x] \equiv_\alpha \phi(z/x)\);
8. \(x \notin FV(\phi)\) implies \(\phi[T/x] = \phi\);
9. \(\phi \equiv_\alpha \psi\) implies \(\text{Subst}(\phi, \theta) \equiv_\alpha \text{Subst}(\psi, \theta)\);
10. \(\theta \in FV(\phi) \Rightarrow \theta \in FV(\phi) \equiv_\alpha \text{Subst}(\phi, \theta) \equiv_\alpha \text{Subst}(\phi, \theta')\);
11. \(\text{Subst}(\phi, \theta) \equiv_\alpha \text{Subst}(\phi, \theta) \equiv_\alpha \text{Subst}(\phi, \theta)';
12. \phi \equiv_\alpha \phi' \wedge \psi \equiv_\alpha \psi' \Rightarrow \phi \wedge \psi \equiv_\alpha \psi' \wedge \phi' \wedge \psi \equiv_\alpha \forall x. \phi \wedge \psi

Proof: We shall tacitly use properties (1)-(6) in the definition of a
term syntax and properties (c)-(i-iii) in the definition of models.
All proofs, except the one of point (12), will be performed by induction.
either on the structure of formulas, or on the structure of $\equiv_{\varphi}$ "IH" will stand for the "Induction Hypothesis". Each time, we shall skip the case of logical connectors $\land, \lor, \Rightarrow$, since the induction step is trivial for them.

We prove (1) and (2) by induction on the structure of $\varphi$.

(1) Base case. $\rho_{\forall \varphi} = \rho_{\forall \varphi}^T$ for each $i \in \{1, \ldots, n\}$, which implies $A_\varphi(i) = A_{\forall \varphi}(i)$ for each $i \in \{1, \ldots, n\}$, which implies that $\rho \in A_\varphi^T$, if $\rho \in A^T$.

Induction step. Assume $\rho_{\forall \psi} = \rho_{\forall \psi}$ for $\forall \psi$. Then $\rho_{\forall \psi}^T \approx \rho_{\forall \psi}^T$, hence for all $a \in A$, $\rho[a] = \rho_{\forall \psi}^T[a]$ for all $a \in A$, as defined by induction. By (IH), we get that for all $a \in A$, $\rho[a] \in A$, $\rho[a] = \rho_{\forall \psi}^T[a]$, in particular that $\rho \in A_{\forall \psi}^T$, if $\rho \in A_{\forall \psi}^T$.

(2) Base case. We have the following equivalences:

$\rho \in A_{\forall \psi} \iff \rho \in A_{\forall \psi}^T \iff \rho \in A_{\forall \psi} \iff \rho \in A_{\forall \psi}^T$

Induction step. We have the following equivalences:

$\rho \in A_{\forall \psi} \iff \rho \in A_{\forall \psi}^T \iff \rho \in A_{\forall \psi} \iff \rho \in A_{\forall \psi}^T$

It remains to prove the promised equivalence. For it, we would suffice that $A_{\varphi[a]}(\rho[z \leftarrow a]) \rho_{\forall \psi} = A_{\forall \psi}(\rho) \rho_{\forall \psi}$ for all $a \in A$, as defined by induction. By (IH), $\rho[a] \in A_{\forall \psi}$, for all $a \in A$, if (as will be proved shortly)

$A_{\forall \psi}(\rho[a] \leftarrow a) \in A_{\forall \psi}$, for all $a \in A$, if $\rho \in A_{\forall \psi}$.

To prove the latter, let $y \in FV(\varphi)$. Then

$\rho[x \leftarrow a](y) = \begin{cases} a, & \text{if } y = x \\ \rho(y), & \text{if } y \neq x \end{cases}$

On the other hand,

$A_{\varphi[a]}(\rho[z \leftarrow a])(y) = A_{\varphi[a]}(\rho[z \leftarrow a])$

And since $z \notin FV(\varphi)$, $\rho(y) = \rho[z \leftarrow a](y)$.

And (4): Base case. Obvious, since here $\equiv_{\varphi}$ coincides with equality.

Induction step. Assume $\forall \psi \equiv \exists \varphi$, i.e., that $\psi[y/z] \equiv_{\varphi}$ for some $z \notin FV(\varphi) \cup \exists \psi$. By (IH), $FV(\varphi[z/x]) = FV(\varphi[y/z])$, hence, by Proposition 1, $FV(\varphi) \subseteq \{x \} \cup \{z\}$, because $z \notin FV(\varphi) \cup \exists \psi$, this implies $FV(\varphi) \subseteq \{x \} \cup \{z\}$, i.e., $FV(\varphi) \cup \exists \psi = FV(\varphi[y/x])$.

Points (5)-(11) shall be proved together. In the case of (5), we prove by induction two properties - reflexivity and transitivity - since symmetry holds by definition.

Base case.

(6) Reflexivity and transitivity follow from the corresponding properties of equality.

(7) and (8): Follow similarly to (6), but also using Proposition 1, points (4) and (1), respectively.

(9): Obvious, since here $\equiv_{\varphi}$ is the equality.

(10): Obvious.

(11): Follow similarly to (6), using properties (3) and (9) in the definition of a term syntax.

Induction step:

(5): For reflexivity, note that $\forall x \varphi \equiv \forall x \varphi \equiv \varphi$ holds because, by (IH) for point (5), $\varphi[x/z] \equiv_{\varphi} \varphi[z/x]$. In order to prove transitivity, assume that $\forall x \varphi \equiv \forall x \varphi \equiv \varphi$ for some $z \notin FV(\varphi) \cup \exists \psi$, and $\forall x \varphi \equiv \forall x \varphi$ for some $z \notin FV(\varphi) \cup \exists \psi$. Then for some $z \notin FV(\varphi) \cup \exists \psi$, it holds that $\varphi[x/z] \equiv_{\varphi} \varphi[z/x]$ and $\varphi[z/x] \equiv_{\varphi} \varphi[x/z]$. From $\varphi[z/x] \equiv_{\varphi} \varphi[x/z]$, we get $\varphi[z/x] \equiv_{\varphi} \varphi[x/z]$, and $\varphi[x/z] \equiv_{\varphi} \varphi[z/x]$. Finally, note that $\varphi[z/x] \equiv_{\varphi} \varphi[z/x]$ and $\varphi[z/x] \equiv_{\varphi} \varphi[z/x]$. This is true by the choice of $\varphi$, $\varphi \notin FV(\varphi[y/x])$. Points (3)-(11) will be proved by induction on the structure of $\equiv_{\varphi}$.

(3): Base case. Obvious, since here $\equiv_{\varphi}$ coincides with equality.

Induction step. Assume $\forall x \varphi \equiv \forall x \varphi \equiv \varphi$ for some $z \notin FV(\varphi) \cup \exists \psi$. We have the following equivalences:

$\rho \in A_{\forall \varphi} \iff \rho \in A_{\forall \varphi}^T \iff \rho \in A_{\forall \varphi} \iff \rho \in A_{\forall \varphi}^T$

Finally, we need to argue that $\rho[a] \leftarrow a \in A_{\forall \varphi}$, which is true by the choice of $\varphi$, $\varphi \notin FV(\varphi[y/x])$.

Points (7)-(11) will be proved by induction on the structure of $\equiv_{\varphi}$.

(7) and (8): Follow similarly to (6), using properties (3) and (9) in the definition of a term syntax.

Induction step:

(5): For reflexivity, note that $\forall x \varphi \equiv \forall x \varphi \equiv \varphi$ holds because, by (IH) for point (5), $\varphi[x/z] \equiv_{\varphi} \varphi[z/x]$. In order to prove transitivity, assume that $\forall x \varphi \equiv \forall x \varphi \equiv \varphi$ for some $z \notin FV(\varphi) \cup \exists \psi$, and $\forall x \varphi \equiv \forall x \varphi$ for some $z \notin FV(\varphi) \cup \exists \psi$. Then for some $z \notin FV(\varphi) \cup \exists \psi$, it holds that $\varphi[x/z] \equiv_{\varphi} \varphi[z/x]$ and $\varphi[z/x] \equiv_{\varphi} \varphi[x/z]$. From $\varphi[z/x] \equiv_{\varphi} \varphi[x/z]$, we get $\varphi[z/x] \equiv_{\varphi} \varphi[x/z]$, and $\varphi[x/z] \equiv_{\varphi} \varphi[z/x]$. Finally, note that $\varphi[z/x] \equiv_{\varphi} \varphi[z/x]$ and $\varphi[z/x] \equiv_{\varphi} \varphi[z/x]$. This is true by the choice of $\varphi$, $\varphi \notin FV(\varphi[y/x])$. Points (3)-(11) will be proved by induction on the structure of $\equiv_{\varphi}$.

(3): Base case. Obvious, since here $\equiv_{\varphi}$ coincides with equality.

Induction step. Assume $\forall x \varphi \equiv \forall x \varphi \equiv \varphi$ for some $z \notin FV(\varphi) \cup \exists \psi$. We have the following equivalences:

$\rho \in A_{\forall \varphi} \iff \rho \in A_{\forall \varphi}^T \iff \rho \in A_{\forall \varphi} \iff \rho \in A_{\forall \varphi}^T$

Finally, we need to argue that $\rho[a] \leftarrow a \in A_{\forall \varphi}$, which is true by the choice of $\varphi$, $\varphi \notin FV(\varphi[y/x])$. Points (7)-(11) will be proved by induction on the structure of $\equiv_{\varphi}$.

(7) and (8): Follow similarly to (6), using properties (3) and (9) in the definition of a term syntax.
Subst(ϕ, [z′/z]). The latter follows by (IH) for point (10), since

\[ \forall x. \phi \rightarrow \phi \] for some \( \psi \notin FV(\phi) \) and \( \forall y. \psi \). In order to prove \( \text{Subst}(\forall x. \phi, \theta) \equiv \alpha \), we take \( z' \) as in the definition of substitution and show that

\[ \forall z. \text{Subst}(\phi, \theta[x/z]) \equiv \forall z'. \text{Subst}(\phi, \theta[z'/z]) \]

For proving the latter, using (IH) for point (11), we take \( z' \notin FV(\phi) \cup FV(\psi) \) and show that \( \text{Subst}(\phi, \theta[x/z]) \equiv \alpha \), i.e., that \( \text{Subst}(\phi, \theta[z'/z]) \equiv \alpha \). Since \( [\phi/z'] \equiv \phi[z'/z] \) and \( [\theta[z'/z]] \equiv \theta[z'/z] \), we reduced the desired equivalence to 

\[ \text{Subst}(\phi[z'/z]; \theta[x/z]) \equiv \alpha \]

Now, by (IH) for point (11), we have that \( \phi[z'/z] \equiv \phi[z'/z] \) and \( \theta[z'/z] \equiv \theta[z'/z] \); moreover, by (IH) for point (10), from \( \theta[x/z] \equiv \theta[z'/z] \), we get

\[ \text{Subst}(\phi[z'/z]; \theta[x/z]) \equiv \alpha \]

(10) Assume \( \theta_{FV(\psi)} \equiv \theta'_{FV(\psi)} \). In order to prove that \( \text{Subst}(\forall x. \phi, \theta) \equiv \forall x. \text{Subst}(\phi, \theta[x/z]) \equiv \forall x. \text{Subst}(\phi, \theta[z'/z]) \), note that the variable \( z \) in the definition of substitution is the same in the two cases, and we need to show \( \forall z. \text{Subst}(\phi, \theta[x/z]) \equiv \forall z. \text{Subst}(\phi, \theta[z'/z]) \), i.e., by (IH) for point (11), that \( \text{Subst}(\phi, \theta[x/z]) \equiv \text{Subst}(\phi, \theta[z'/z]) \). The latter is true by (IH) for point (10), since \( \theta[x/z] \equiv \theta[z'/z] \).

(11) In order to prove that \( \text{Subst}(\forall x. \phi, \theta; \theta') \equiv \alpha \), let \( z, z' \) as in the definition of substitution (for each of the three involved substitutions). We need to show that \( \forall z. \text{Subst}(\phi, \theta; \theta'[x/z]) \equiv \forall z. \text{Subst}(\phi, \theta; \theta'[z'/z]) \), i.e., that

\[ \text{Subst}(\phi, \theta; \theta'[x/z]) \equiv \text{Subst}(\phi, \theta; \theta'[z'/z]) \]

Indeed, using (IH) for point (11) and the freshness of \( z, z', z'' \), we have the following chain of \( \alpha \)-equivalences and equalities:

\[ \text{Subst}(\phi, \theta; \theta'[z'/z]) \equiv \alpha \]

\[ \text{Subst}(\phi, \theta; \theta'[x/z]) \equiv \alpha \]

By which (IH) for point (5) yield the desired result.

(12) The cases of logical connectives are obvious. Assume now \( \varphi \equiv \varphi' \). Then, by point (9), \( \varphi[x/z] \equiv \varphi'[x/z] \) for any \( x \) and \( z \), in particular \( \forall x. \varphi \equiv \forall x. \varphi' \).

\( \square \)

Theorem 1. The Gentzen system \( \mathbb{G} \) is sound and complete for generic first-order logic.

Proof: Soundness: We need to check that the rules are sound. We only consider the quantifier rules, since the soundness of the others follows standardly. Let L be a model. For soundness of (∀Left), it suffices that \( A_{x.\varphi} \subseteq A_{\varphi[y/z]} \), which is true because of the following: \( \rho \in A_{x.\varphi} \) is equivalent to \( \rho[x/a] \in A_{\varphi[y/z]} \) for all \( a \in A \), which implies \( \rho(x - A_{\varphi}) \in A_{\varphi} \), which in turn is equivalent, by Proposition 2.2, to \( \rho \in A_{\varphi[y/z]} \).

For (∀Right), we shall tacitly use Proposition 2.1.2 several times. Assume \( \bigcap_{\Delta \subseteq A \subseteq A_{\varphi[y/z]}} \), where \( y \) is not free in \( \Gamma, \Delta, \forall x. \varphi \). We need to show \( \bigcap_{\Delta \subseteq A \subseteq A_{\varphi[y/z]}}, \forall x. \varphi \). For this, let \( \rho \in \bigcap_{\Delta \subseteq A \subseteq A_{\varphi[y/z]}} \), and let us show that \( \rho \in A_{\varphi[y/z]} \), i.e., that \( \rho(x - A_{\varphi}) \in A_{\varphi[y/z]} \).

\[ \forall a \in A. \text{Let } \rho(a - A_{\varphi}) \in A_{\varphi[y/z]} \text{ for each } \chi \in \Gamma \cup \Delta \text{ and substitution } \psi \notin FV(\chi) \text{ and } \forall y. \psi \text{ and } \forall y. \varphi \text{ for each } \chi \in \Gamma \cup \Delta \text{ and } \text{substitution } \psi \notin FV(\chi) \text{ and } \forall y. \psi \text{ respectively.} \]

\[ \forall a \in A. \text{Let } \rho(a - A_{\varphi}) \in A_{\varphi[y/z]} \text{ for each } \chi \in \Gamma \cup \Delta \text{ and } \forall y. \varphi \text{ and } \forall y. \psi \text{ for each } \chi \in \Gamma \cup \Delta \text{ and } \forall y. \varphi \text{ respectively.} \]

\[ \forall a \in A. \text{Let } \rho(a - A_{\varphi}) \in A_{\varphi[y/z]} \text{ for each } \chi \in \Gamma \cup \Delta \text{ and } \forall y. \varphi \text{ and } \forall y. \psi \text{ for each } \chi \in \Gamma \cup \Delta \text{ and } \forall y. \varphi \text{ respectively.} \]

\[ \forall a \in A. \text{Let } \rho(a - A_{\varphi}) \in A_{\varphi[y/z]} \text{ for each } \chi \in \Gamma \cup \Delta \text{ and } \forall y. \varphi \text{ and } \forall y. \psi \text{ for each } \chi \in \Gamma \cup \Delta \text{ and } \forall y. \varphi \text{ respectively.} \]
Moreover, because the considered infinite path does not contain
(nodes labelled with) axioms and because both the $Γ_i$'s and the $Δ_i$'s are
totally ordered by inclusion w.r.t. the atomic formula, it holds that $H_{left} \cap H_{right} \cap Atomic\ Formulae = \emptyset$.

In order to falsify $Γ \vdash Δ$, it suffices to falsify $H_{left} \triangleright H_{right}$. We
define a Herbrand model $A$ by letting $(T_1, \ldots, T_n) \in A_v$ iff
$\pi(T_1, \ldots, T_n) \in H_{left}$, and take the valuation $ρ : Var \rightarrow A$ to be
again $1_{var}$. By structural induction on $ϕ$ the following statement:
$ϕ \in H_{right}$ implies $A \models ϕ$ and $ϕ \in H_{left}$ implies $A \not\models ϕ$. If $ϕ$ has the form $π(T_1, \ldots, T_n)$, then by the definition of $A$, $A \models π(T_1, \ldots, T_n)$ iff $π(T_1, \ldots, T_n) \in H_{right}$. In particular, $π(T_1, \ldots, T_n) \in H_{left}$ implies $A \models π(T_1, \ldots, T_n)$. Moreover, since $π(T_1, \ldots, T_n)$ is atomic and because of $H_{left} \cap H_{right} \cap Atomic\ Formulae = \emptyset$, it cannot happen that $π(T_1, \ldots, T_n) \in H_{left}$ and $A \not\models π(T_1, \ldots, T_n)$, thus $π(T_1, \ldots, T_n) \in H_{right}$ implies $A \not\models π(T_1, \ldots, T_n)$. The case of the logical connectives is straightforward. Assume now $ϕ$ has the form $∀x.ψ$. From $∀x.ψ \in H_{left}$ we infer that $ψ[T/x] \in H_{left}$ for each term $T$ (i.e., for each element $T$ of $A$), and furthermore, by the induction hypotheses, that $A \models ψ[T/x]$, i.e., $ρ \in A_{ψ[T/x]}$, i.e., by Proposition 2.9 ($ρ \leftarrow τ, A \models ψ[T/x]$), furthermore, $ρ \not\models T \in A_v$, for each $T \in A$; thus $A \models ψ[T/x]$ for each $∀x.ψ \in H_{left}$ implies $A \not\models ∀x.ψ$ can be proved similarly.

Thus $Γ \vdash Δ$ is falsifiable and the proof is finished.

THEOREM 2. The Gentzen system $G_2$ is sound and complete for
generic first-order languages with equality.

Proof: Soundness: We only check soundness of the rule regarding
substitution. Let $A$ and $ρ$ such that $A_{τ} (ρ) = A_{τ'} (ρ)$ for each $τ \in \{ 1, a \}$.
Then by point (ii) in the definition of $A_{τ} (ρ)$, $A_{τ} (ρ) (l) = A_{τ'} (ρ) (l)$ for each $l \in A_v$. By $τ \not\models A_{τ} (ρ)$, we have $τ \not\models A_{τ'} (ρ)$.

Completeness: One can see that the effect of adding the equality
rules amounts to adding the axioms $EqI$ in the antecedent of sequents. Thus $Γ \vdash Δ$ is provable in $G_2$, iff $Γ \cup \Gamma \vdash Δ$ is provable in $G_2$.

Now, given any model $A$ in the language without equality (i.e.,
in the language that contains $\%\%$", but treats this symbol as just an
ordinary binary relation symbol) that satisfies $EqI$, one defines the
relation $≡$ by $a ≡ b$ iff $A \models x = y$ for some $ρ$ with $ρ(x) = a$ and
$ρ(y) = b$. Due to satisfaction of $EqI$ by $A$, $≡$ is an equivalence
relation compatible with the relations $A_{τ}$ and with the substitution,
the latter in the sense that whenever $A_{τ} (ρ) = A_{τ'} (ρ)$, it holds that
$A_{τ} (ρ) (l) = A_{τ'} (ρ) (l)$ for each $l \in A_v$. Thus we can speak about a quotient
model $A_{τ}$ and define, for each $ρ : Var \rightarrow A_{τ}$, $A_{τ} (ρ) \models ϕ$ by $ρ (x) = ρ (x')$. Then a simple induction on $ϕ$ shows that
$A_{τ} (ρ) \not\models ϕ$ if $A \not\models ϕ$. It follows that $Γ \vdash Δ$ is tautological in the logic with equality iff $Γ \cup Γ \vdash Δ$ is tautological in the logic
without equality.

Completeness of $G_2$ now follows from completeness of $G_2$.

Below, by "proof tree" we mean "completed proof tree".

LEMMA 1. Assume that $G_E$ is closed under the rules (Drop-
(e, a)) If $Γ \vdash Δ_1 \cup Δ_2$ is derivable in $G_E$, then either $Γ \vdash Δ_1$ or $Γ \vdash Δ_2$ is derivable in $G_E$.

Proof: We prove the statement by induction on the size of a
minimal proof tree for $Γ \vdash Δ_1 \cup Δ_2$. Let $Γ \vdash Δ_1 \cup Δ_2$ be an axis, then $Γ \cup Δ_1 \not\models \emptyset \cup \emptyset \models Δ_1 \not\models \emptyset$, making $Γ \vdash Δ_1$ or $Γ \vdash Δ_2$ an axis.

If $Γ \vdash Δ_1 \cup Δ_2$ followed by an application of an (Inst-e) rule,
then assume w.l.o.g. that $Δ_2 = Δ_2 \cup \{ τ (T) \}$ and $Γ \vdash Δ_2 \cup Δ_2$
followed from $Γ \cup \{ τ (T) \}$ by $Δ_1 \cup Δ_2 \cup \{ b (T, T) \}$, with $i \in \{ 1, \ldots, n \}$. By the induction hypothesis, for each $i$, either

$Γ_a (τ, T) \triangleright Δ_1$, or $Γ_a (τ, T) \triangleright Δ_2 \triangleright Δ_1 \triangleright Δ_2$. If the former is the case for at least one $i$, then since $G_E$ is closed under (Drop-
(e, a)), we get that $Γ \triangleright Δ_1$ is derivable. Otherwise $Γ_a (τ, T) \triangleright Δ_2 \triangleright Δ_1 \triangleright Δ_2$, for all $i$, hence, by the (Inst-e) rule, $Γ \triangleright Δ_2 \triangleright Δ_1 \triangleright Δ_2$ is derivable, □

LEMMA 2. $G_E$ and $G_E^2$ are equivalent (i.e., (Simple-Cut) can be eliminated from $G_E^2$).

Proof: We show that for each proof tree for $Γ \vdash Δ$ having a
proof tree for $Γ \vdash Δ$. Then we have a proof tree for $Γ \vdash Δ$ that does not use this rule at all. Assume such a proof tree $T$ starting with an application of (Simple-Cut)- then $Γ \vdash Δ$ followed from $Γ \vdash d' \vdash d$ and $Γ \vdash \Gamma' \vdash Δ$.

If $Γ \vdash Δ$ is an axis, then either $Δ \cap Δ \not\models \emptyset$, meaning $Γ \vdash Δ$ is an axis, or $d \models Δ$, meaning that the proof tree of $Γ \vdash d$, which does not use (Simple-Cut) can be made into a proof tree of $Γ \vdash Δ$ that does not use (Simple-Cut) either.

Assume now that $Δ \not\models Λ \cup \{ τ (T) \}$ and $Γ \vdash Δ$. We obtain a proof tree $T'$ for $Γ \vdash Δ$ by switching the applications of (Simple-Cut) and (Inst-e). More precisely, we derive $Γ \vdash Δ$ from $Γ a_1 (τ, T) \triangleright Δ_1 \triangleright Δ_2$, and $Γ a_1 (τ, T) \triangleright Δ_2 \triangleright Δ_3$, where the proof of $Γ a_1 (τ, T) \triangleright d$ is copied from the one of $Γ \vdash d$. Applying the induction hypothesis for the proof trees (strictly smaller than $T'$) of $Γ a_1 (τ, T) \triangleright Δ_1 \triangleright Δ_2$, we can assume that they do not use (Simple-Cut), and we are done.

PROPOSITION 5. If $G_E$ is closed under the rules (Drop(e, a)), then it is also closed under (Cut), i.e., then $G_E$ is equivalent to $G_E^0$.

Proof: Consider a proof tree $Γ \vdash Δ$ in $G_E$, such that the (Cut) rule was applied only ones, at the root. Then $Γ \vdash Δ$ followed from $Γ \vdash Δ \vdash Δ$, where both latter sequents are derivable in $G_E$. By Lemma 1, either $Γ \vdash Δ$ or $Γ \vdash d$ is derivable in $G_E$. The former case is precisely what we need to prove; in the latter case, $Γ \vdash d$, and thus $Γ \vdash Δ$, are derivable in $G_E$, i.e., by Lemma 1 in $G_E$. □

THEOREM 3. If $G_E$ is closed under the rules (Drop(e, a)), then $K_E$ is complete for deducting E-tautological sequents $Γ \vdash d$, with $Γ$ finite set of atomic formule and $d$ atomic formula.

Proof: Follows at once from Proposition 5, noticing that in the system $G_E$, if the sequent $Γ \vdash d$ is derivable, then all the sequents $Γ \vdash Δ \vdash d$ in the proof tree have $Δ$ a singleton - in other words, $G_E$ is a conservative extension of $K_E$. □

PROPOSITION 6. All the theories $E$ in Subsection 3.1 are amenable, and thus $K_E$ is complete for any of them.

Proof: We only sketch a proof for the specification $S F$ of System $F$. The cases of the other specifications, including their versions that contain axioms of various kinds or kinds and equations that define their behavior, can be treated similarly, as they all need to "drop" atomic formulae like typeOf(x, T) or kindOf(x, K), with x and t fresh variables. The trick used in the below proof of restricting the "troubling rules" works in all these cases, because the troubling rules for all these specifications are the same, namely the ones that come from the equality axioms of transitivity, comparability and substitution. After the restriction, a straightforward induction for proving closure under the drop rules works fine in each case.

Thus let us prove amenability of $S F$. The only type of drop rules Drop(e, a) come from being [Abs] or (Σ) and (a, T) being typeOf(x, T), with x fresh. Thus we need to prove closure under...
The side-conditions of the above rules are the following:

- At (Inst-abs) and (Inst-ξ), x does not occur (free) in Γ ∪ Δ;
- At (Inst-T-abs) and (Inst-T-ξ), t does not occur in Γ ∪ Δ;
- At (Inst-η), y \not\in FV(X);
- At (Inst-T-η), t \not\in FV(X).

As expected, the above system of rules is obtained from \( \text{K}_{S,P} \) of Section 3.2 by adding finite sets \( \Delta \) of atomic formulae to the succedents (here we write "\( \text{type}(\cdot) \)" rather than "\( \vdash \cdot \)" though).

Now, an argument for "\( \Delta \vdash \text{type}(z), \Gamma \vdash \Delta \) (with \( z \) fresh)" derivable implies \( \Gamma \vdash \Delta \) derivable, inductive on the structure of a completed proof tree of \( \Gamma \vdash \text{type}(z), \Gamma \vdash \Delta \). almost works. For example, the induction step for rule (Inst-abs) goes as follows: Assume \( \Gamma \vdash \text{type}(z), \Gamma \vdash \Delta \). was derived via \( \Gamma \vdash \text{type}(z), \Gamma \vdash \Delta \). Therefore, \( \Gamma \vdash \text{type}(z) \), and thus \( z \) is different from \( x \). Moreover, since \( \text{FV}(\text{type}(X,T')) \setminus \text{FV}(\text{type}(\lambda x : T.x)), T \rightarrow T' \) is a \( \Gamma \vdash \Delta \), the \( \Delta \) is fresh for \( \text{type}(X,T') \), therefore, \( \Gamma \vdash \Delta \) derivable, as desired.

The only problems are caused by the rules (Inst-trans), (Inst-comp-type0), and (Inst-subst) - as they are, these rules cannot be considered in a straightforward induction, because they might have more data variables in their hypotheses than in their conclusion. However, these rules can be replaced by apparently weaker rules, obtained from them by adding the following side-conditions:

- At (Inst-trans) \( \Gamma \vdash F(Y) \subseteq F(X) \cup F(Z) \cup F(\Gamma) \);
- At (Inst-comp-type0) \( \Gamma \vdash F(Y) \subseteq F(X) \cup F(\Gamma) \);
- At (Inst-subst) \( \Delta[x \mapsto X] \in \Delta[x \mapsto Y] \) are syntactically different.

Now the induction step works smoothly for these rules as well.

**Lemma 3.** The forgetful mapping \( (A, \Delta, \lambda\cdot) \mapsto (\Delta, \lambda\cdot) \) is a bijection, preserving satisfaction and each of the properties (P5), (P6), between pre-structures verifying (P1)-(P4) and (P7) and simple pre-structures verifying (P1)-(P4).

**Proof:** The inverse of the forgetful function maps simple pre-structures \( (A, \Delta, \lambda\cdot) \mapsto (\Delta, \lambda\cdot) \) verifying (P1)-(P4) to \( (A, \Delta, \lambda\cdot) \), where for each term \( T \) in \( A \), sequence of elements \( a_1, \ldots, a_n \) in \( A \) and sequence of distinct variables \( x_1, \ldots, x_n \), \( A_{[x_1/a_1, \ldots, x_n/a_n]}(T) \) is, by definition, \( A_{[x_1/a_1, \ldots, x_n/a_n]}(T) \). This definition is correct, because any term in \( A(\lambda\cdot) \) has the form \( T[x_1/a_1, \ldots, x_n/a_n] \) for some \( T \) and \( A_{[x_1/a_1, \ldots, x_n/a_n]}(T) \) does not depend on the choice of \( T \). A simple induction on the structure of \( A(\lambda\cdot) \) terms shows that \( A_{[x_1/a_1, \ldots, x_n/a_n]}(T) \) is indeed equal to \( A_{[x_1/a_1, \ldots, x_n/a_n]}(T) \) in any pre-structure, and thus the two mappings are mutually inverse.

These mappings preserve satisfaction, since satisfaction is basically the same in each two structures related by these mappings - note also that in pre-structures only satisfaction of \( \lambda\cdot \)-term equalities is defined. As for properties (P5) and (P6), one needs another induction to prove that in a pre-structure, verifying these properties w.r.t. \( \lambda\cdot \)-terms is sufficient for them to hold w.r.t. \( \lambda\cdot \)-terms; here one uses again the equality \( A_{[x_1/a_1, \ldots, x_n/a_n]}(T) = A_{[x_1/a_1, \ldots, x_n/a_n]}(T) \).

**Lemma 4.** Each of the schemes of formulae \( \beta(\cdot), \eta(\cdot), \text{ext}(\cdot) \) is semantically equivalent in GFO\(\cdot\) to its primed variant.

**Proof:** Since \( \beta(\cdot), \eta(\cdot) \) and \( \text{ext}(\cdot) \) are instances of the schemes \( \beta(\cdot), \eta(\cdot) \) and \( \text{ext}(\cdot) \), all we need to show is that the latter follow from the former; and this simply holds because in our logic it is sound to
infer \( \varphi(T) \) from \( \forall y \varphi(y) \), and by the way substitution in formulae was defined.

**Proposition 7.** The above two mappings are well defined and mutually inverse. Moreover, they preserve satisfaction and they can be restricted and restricted to:

(a) \( \lambda \)-models versus GFOL models satisfying \((\xi), (\beta)\);
(b) extensional \( \lambda \)-models versus GFOL models satisfying \((\xi), (\beta), (\eta)\).

**Proof:** We show that \( L^A \) is a GFOL model. Two of the GFOL model axioms, \((e), (ii)\) and \((e), (iii)\), are precisely \((P1)\) and \((P3)\). The remaining axiom, \((e), (iii)\), cannot be written as \( A_{r_1 T_1/x_1 \ldots \ldots x_n/x_n}(\rho) = A_T(\rho[x_1 \mapsto A_T(x_1) \ldots \ldots A_T(x_n)]) \). We check this by lexicographic induction on two criteria: the depth of \( T \), and then the number \( n \). The cases with \( T \) variable and \( T \) of the form \( T' \) are simple, and they use \((P1)\) and \((P2)\). Assume now that \( T \) has the form \( \lambda x T' \). Since we work modulo \( \alpha \)-equivalence, we can assume that \( x \) is not free in any of \( T_1, \ldots, T_n \).

- If \( x \notin \{x_1, \ldots, x_n\} \), then \( T[T/x] = \lambda x (T'[T/x]) \). Thus we need to check \( A_{x, T'[T/x]}(\rho) = A_{\lambda x, T'}(\rho[\tau \mapsto A_T(\rho)]) \). By \((P4)\), it is sufficient to consider \( a \in A \) and prove \( A_{x, T'[T/x]}(\rho[x \mapsto a]) = A_{\lambda x, T'}(\rho[\tau \mapsto A_T(\rho)]) \). Assume \( \lambda x \rightarrow \alpha \).

- If \( x \in \{x_1, \ldots, x_n\} \), then \( T[T/x] = T[T_1/x_1, \ldots, T_n/x_n] \) and by the induction hypothesis (second criterion), \( T[T_1/x_1, \ldots, T_n/x_n] = A_T(\rho[x_1 \mapsto A_T(x_1) \ldots \ldots A_T(x_n)]) \). Finally, \( x \notin \{x_1, \ldots, x_n\} \) and \( \rho[\tau \mapsto A_T(\rho)] \) and \( \rho_2 = \rho[x \mapsto A_T(x)] \) coincide on \( T[T/x] \), hence, by \((P3)\), \( A_T(\rho_1) = A_T(\rho_2) \).

(Above, we used the obvious tuple notations \( \mathcal{T} \) for \( \{x_1, \ldots, x_n\}, \mathcal{T} \) for \( \{T_1, \ldots, T_n\} \), \( L^A \) satisfies \((\xi)\) because \((P4)\) is nothing else but a semantic statement of \( \xi \).

We show that \( \tau^A \) is a \( \lambda \)-model. Properties \((P1)\) and \((P3)\) are required for generic models as well, hence they hold. \((P2)\) holds as an instance of the axioms \((e), (ii)\) of GFOL models: \( A_{T_1 T_2}(\rho) = A_{T_1 T_2}(\rho[x_1 \mapsto A_T(x_1), \ldots \ldots x_n \mapsto A_T(x_n)]) \).

That \# and \$ are mutually inverse follows from the fact that \( (a|b) = A_{r_1 r_2}(\rho) \) with \( r_1 \mapsto a \) and \( r_2 \mapsto b \), holds in any pre-structure verifying \((P1)-(P4)\).

In order to see that the pre-structure is a \( \lambda \)-structure (i.e., it also verifies \( \varphi(T) \)) iff the corresponding GFOL model satisfies \((\beta)\) (note that \((P5)\) is just a semantic statement of \((\beta)\), which is equivalent to \((\beta)\) by Lemma 4). Similarly, extensional \( \lambda \)-models correspond to \((\beta) \cup (\eta)\) GFOL models because of the following:

- under \((\beta)\), \((\xi)\) assumptions, \( (\eta) \) is equivalent to \((\xi)\);  
- \((\xi)\) is equivalent to \((\xi)\) by \((\xi)\);  
- \((\xi)\) is just a semantic statement of \((\xi)\).

Finally, both \# and \$ preserve satisfaction since it has the same definition for equations in pre-structures and generic models.

**Proposition 8.** For all typing judgements \( \Gamma \triangleright X : T \),\( \vdash_{SF} \Gamma \triangleright X : T \iff \vdash_{SF} \Gamma \triangleright \text{typeOf}(X, T) \).

**Proof:** Let \( SF \) denote the typing fragment of \( SF \), i.e., the ones consisting of axioms whose labels were enclosed into brackets: \([\text{Abs}], [\text{Abs}], [\text{App}], [\text{App}]\). \( SF \) was proved amenable."
in the definition of 3-Henkin models. Everything else remains the same.

Note that \( \alpha_3^2 \circ \alpha_3^2 \) is the identity mapping, thus 2-Henkin models are somehow more "concise" than 3-Henkin models. Again, there is nothing to prove about preservation an reflection of satisfaction, since again the satisfaction relation is the same in two corresponding models. \( \Box \)

**Lemma 8.** Let \( H \) be a 3-Henkin model, \( \gamma : \text{Var} \to T \) and \( \delta : \text{Var} \to D \). Then there for any two pairs \( (\Gamma, T) \) and \( (\Gamma', T') \) such that \( \vdash \Gamma : T \) and \( \vdash \Gamma' : T' \) such that \( H_{\Gamma} \models_{X:T} \) and \( H_{\Gamma'} \models_{X:T} \) are defined on \( (\gamma, \delta) \), it holds that \( H_{\Gamma} \models_{X:T}(\gamma, \delta) = H_{\Gamma'} \models_{X:T}(\gamma, \delta) \).

**Proof:** Easy induction on the derivation of typing judgements, the idea is that all the information needed for interpreting \( X \) is already in the valuation \( (\gamma, \delta) \), and \( \Gamma \) and \( T \) can only "confirm" this information. \( \Box \)