Dependently Sorted Nominal Signatures

Maribel Fernández
King's College London, UK

Miguel Pagano
Univ. Nac. Córdoba, Argentina

Nora Szasz
Álvaro Tasistro
Universidad ORT Uruguay

We investigate an extension of nominal many-sorted signatures in which abstraction has a form of instantiation, called generalised concretion, as elimination operator (similarly to lambda-calculi). Expressions are then classified using a system of sorts and sort families that respects alpha-conversion (similarly to dependently-typed lambda-calculi) but not allowing names to carry abstraction sorts, thus constituting a first-order dependent sort system. The system can represent forms of judgement and rules of inference of several interesting calculi. We present rules and properties of the system as well as experiments of representation, and discuss how it constitutes a basis on which to build a type theory where raw expressions with alpha-equivalence are given a completely formal treatment.
Keywords: Nominal Terms; Logical Frameworks; Dependent Types.

1 Introduction

We present a generalisation of Pitts' nominal many-sorted signatures [18], where sorts can now depend on terms, yielding a *dependently sorted system* that inherits the distinctive first-order algebraic flavour of nominal signatures. We show that this system can serve as a basis for a logical framework.

Nominal logic programming languages such as α-Prolog [8] provide support for the specification of data structures that include bound names and for the formalisation of their properties. Resolution can be used to prove properties. However, there is no type distinction between solvable/unsolvable goals. Gabbay et al [12, 13, 14] proposed alternative formulations of nominal systems with meta-variables that can be used to represent schematic proofs. These systems, termed one-and-a-halfth-level or two-level calculi, have type systems closer to simply typed lambda calculus. So, for example, one can introduce signatures for First-Order Logic (FOL) and the type system will ensure that the equality of two terms is well-typed only when predicated on terms of the sort corresponding to terms of the object language. On the other hand, again there is no type distinction between theorems and contradictions. To express that ϕ is a theorem, one has to construct not a term, but a derivation of $\phi = \top$ from the axioms of the theory. Notice that this equality corresponds to a different form of judgement. Besides the ample evidence of formalisation of mathematics and computer science in Higher-Order Logic (HOL) and Isabelle/HOL, Bordg, Paulson and Li [4] have shown that sophisticated mathematical constructions can be formalised in simply typed lambda calculus.

With dependent types one can introduce the type (or *sort* as we use in this article) `Form` of well-formed formulas and a family of sorts $\mathscr{D}(\phi)_{\{\phi \in \texttt{Form}\}}$ of proofs of theorems. Now both well-formedness and theoremhood of the object language are represented as typing judgements in the meta-language.

In this paper we aim to demonstrate that a quite simple nominal language with dependent sorts can be used as a logical framework [15]. The nominal foundation introduces a theory of expressions subject to alpha-conversion, on top of which a system of dependent sorts is built that respects alpha-equivalence. To this effect, we let names carry sorts of data —but not abstraction (or "higher-order") sorts. This yields a limited form of computation associated to the elimination of abstractions (concretion), that can be

Chaudhuri, Nantes-Sobrinho (Eds.): International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2025)
EPTCS ??, 2025, pp. 1–17, doi:10.4204/EPTCS.??.??

solved at the level of syntactic meta-definitions. Thus the language becomes a first-order dependent sorts system. We have developed several examples showing its ability to represent binding structures: First Order Logic, Lambda Calculi and Higher-Order Logic. In this paper we show some of these, all treated employing a shallow-encoding strategy in which the object-language substitution is directly implemented with our concretion operation. In addition, a deep encoding of the pure Lambda Calculus is shown to allow the formulation of an alpha-structural induction principle as described and justified in [18]. We deem this example as opening a path for extending the present system with full rules of computation (i.e. possibly recursive definitions) respecting alpha-conversion, thus yielding a dependent type theory à la Martin-Löf with a nominal syntactic foundation providing the treatment of binding at the infrastructural level.

In Section 6 we briefly discuss related work, highlighting the systems that are closer to our approach and comparing with previous works on dependently sorted first-order signatures [5, 20] as well as dependent type systems based on extensions of the λ-calculus with nominal features [7, 19].

2 Preliminaries

2.1 Simple Nominal Signatures

We start with a review of nominal signatures [22, 18]. Here and in the rest of the article we overline symbols or phrases to denote sequences of the species corresponding to the overlined entity.

A signature is a triple $(\mathscr{S},\mathbb{A},\Sigma)$, where $\mathscr{S}$ is a set of *basic sorts* including data sorts and name sorts; $\mathbb{A}$ is a family of countable infinite sets of names (*atoms*), indexed by name sorts. Given $\mathscr{S}$ and $\mathbb{A}$ the following grammar generates *the sorts of terms*:

$$\gamma ::= \mathsf{a} \mid \mathsf{s} \mid \ll\mathsf{a}\gg\overline{\gamma}$$

In this grammar a is a *name* sort, s is a *data* sort, and $\ll\mathsf{a}\gg\overline{\gamma}$ is an *abstraction* sort. Σ is a set of declarations of (term-)*constructors*, each with its *arity*, i.e. $\kappa : \overline{\gamma} \to \mathsf{s}$.

The set of *sorted* terms for the signature $(\mathscr{S},\mathbb{A},\Sigma)$ is given by the following *sorting* rules:

$$\text{(atom)}\ \frac{}{a : \mathsf{a}}\ a \in \mathbb{A}_{\mathsf{a}} \qquad \text{(constr)}\ \frac{\overline{t} : \overline{\gamma}}{\kappa\overline{t} : \mathsf{s}}\ \kappa : \overline{\gamma} \to \mathsf{s} \in \Sigma \qquad \text{(abs)}\ \frac{\overline{t} : \overline{\gamma}}{\ll a\gg\overline{t} : \ll\mathsf{a}\gg\overline{\gamma}}\ a \in \mathbb{A}_{\mathsf{a}}$$

These terms are *ground*; i.e., terms without meta-variables [22, 9]. As usual in a nominal setting one introduces the notion of freshness and identifies terms up to alpha-equivalence. The interesting case of alpha-equivalence is for abstractions, which is defined by means of permutation of the abstracted atom by a fresh enough one. We omit those definitions here, but analogous notions appear in Section 3.

2.2 Dependent Sorts

Sort Families

We begin by introducing *families of sorts indexed over a sort*. To this end, we extend signatures to include *sort constructors*, i.e., symbols declared as: $\mathscr{F} : (\overline{X : \gamma}) \to \mathsf{data}$.

In such a declaration, some sorts γ_j might certainly depend on *preceding* parameters $X_i : \gamma_i$ with $i < j$; this is typical of telescopic structures (e.g., contexts in dependently typed lambda calculi). Each X in the telescope $\overline{X : \gamma}$ is called a *parameter*; we will use capital letters $P, Q, X, Y, \ldots$ to denote parameters, and when a parameter is not used in the successive sorts, we shall simply write an underscore, $\_ : \gamma_i$.

These declarations will as a whole take the place of the category of *data* sorts in the simple setting laid out above. To declare non-dependent sorts we use the empty list for $(\overline{X:\gamma})$ and simply write $\mathscr{F}:\mathtt{data}$. The generality now introduced allows us to have data sorts obtained by instantiating data sort constructors with terms: a sort constructor becomes a sort only when fully applied to arguments of the appropriate sorts as declared in the signature.

Of course, we will still have term constructors with telescopic parameter structures and result sorts, i.e. declarations of the form $f:(\overline{X:\gamma})\rightarrow\ \delta$, with δ a data sort depending on the parameters X_i.

We shall also allow abstraction sorts; in our case, however, the sort of the body of an abstraction term might depend on the abstracted atom. Thus, abstraction sorts carry the name of the atom being abstracted. We let atoms carry sorts of *data*, indicating, in a manner to be explained later, the sort of expressions in which they can be instantiated or *concretised*.

It is worth remarking that the system is intended to generate a language of *ground* terms; i.e., terms with parameters are only used to build declarations, and the targeted judgements are all ground.

Motivating Examples

We begin with FOL as drawn from a natural deduction style presentation. We start by introducing data sorts for *terms* and *formulas*, as follows:

$$\mathtt{Term}:\mathtt{data}\qquad\text{and}\qquad\mathtt{Form}:\mathtt{data}$$

We will for the moment let `Term` further unspecified and concentrate on formulas. Each formula has its potential derivations, which we will represent by introducing a family of sorts:

$$\mathscr{D}:\mathtt{Form}\rightarrow\mathtt{data}$$

A first simple formula is the one representing *falsity*, usually called *bottom* with its elimination rule:

$$\bot:\mathtt{Form}\quad\text{and}\quad\bot_e:(\_:\mathscr{D}(\bot),P:\mathtt{Form})\rightarrow\ \mathscr{D}(P)$$

There are no direct (canonical) derivations of $\bot$, so we do not introduce any term constructor with target sort $\mathscr{D}(\bot)$. The meaning of the declaration for $\bot_e$ is as follows: one can "instantiate" the parameters corresponding to the derivation of falsity and the formula P to get a proof term for the formula instantiating P. A vector of arguments *fits* the telescope of a constructor if each term of the list has the appropriate sort. We now go on to consider *implication*:

$$\supset:(\_:\mathtt{Form},\_:\mathtt{Form})\rightarrow\ \mathtt{Form}$$

Let us consider now its introduction rule; its main parameter must be a derivation of the consequent (say Q) in which an assumption of a derivation of the antecedent (say P) has been discharged, i.e. made local. This phenomenon of discharge/locality is naturally represented in our setting by abstraction. More specifically, the sort of the derivation in question will be an abstraction sort $\ll h{:}\mathscr{D}(P)\gg\mathscr{D}(Q)$. However, a further issue must be considered, namely that the consequent Q can in principle be any term of sort `Form`. It could contain (free) atoms whatsoever, in particular h. And this is a possibility we wish to exclude, because it would result in undesired name capture when forming the abstraction sort above. Hence the declarations in our system have the sort for the constructor and also a set of freshness conditions, constituting what we call a *freshness context*; this is how we indicate that the name (atom) h is to be chosen fresh in the term instantiating Q. We then write:

$$\supset_i:(P:\mathtt{Form},Q:\mathtt{Form},\_:\ll h{:}\mathscr{D}(P)\gg\mathscr{D}(Q))\rightarrow\ \mathscr{D}(\supset\,(P,Q))\ ;\ h\#Q$$

Notice that in the declaration of $\supset$ we omitted the freshness context because it was empty; we will continue with this practice in the rest of the paper.

Let us now show how to actually prove $\varphi \supset \varphi$, for any formula φ. Indeed, $\mathscr{D}(\supset(\varphi,\varphi))$ should be realisable for any term φ of sort `Form`, of course without further assumptions. In an ordinary textbook presentation of FOL, a *schema* of derivations would be provided, actually depicting as many concrete derivations as actual formulas there may be. We proceed in the same way here, i.e. by offering a *schema* of judgements provable in our system, that correspond to the desired derivations. *Internalising* such schemas requires passing from a language of ground terms to one containing meta-variables, a possibility yet to be developed. So let by now suppose we have a (ground) term φ of sort `Form`. Thus, for some appropriate term ?0 we shall actually derive $\vdash ?0 : \mathscr{D}(\supset(\varphi,\varphi))$. A possible realisation of ?0 is via the term: $\supset_i(\varphi,\varphi,\ll a{:}\mathscr{D}(\varphi)\gg a)$. Obviously, given any φ, it should be possible to choose a so as to accomplish the freshness condition indicated above. This is to be checked by the system as explained later.

The elimination rule for implication is now straightforward:

$$\supset_e : (P : \texttt{Form}, Q : \texttt{Form}, \_ : \mathscr{D}(\supset(P,Q)), \_ : \mathscr{D}(P)) \to \mathscr{D}(Q)\ ;$$

We skip the rest of the usual propositional connectives and go straight to the universal quantifier:

$$\forall : (\_ : \ll \_ : \texttt{Term} \gg \texttt{Form}) \to \texttt{Form}$$

This shows another methodological point regarding the encoding of object languages in this system: functions, as e.g. predicates, are uniformly represented as abstractions. We shall have an operation of *atom substitution* and attach to each atom the sort of terms that may be substituted for it. The operation of atom substitution shall be actually subsumed into that of *generalised concretion* (cf. [17]), to be introduced later. An atom shall not in any case have an abstraction sort as its sort, thus making the notion of computation associated to generalised concretion very simple and of a first-order character.

Turning back to the universal quantifier, its introduction rule requires a proof of a generic instance of the predicate. Here comes the first use of the *concretion* operator, in this case in its original, simple form that uses a specific fresh name instead of the (originally mute, unknown) abstracted atom:

$$\forall_i : (P : \ll \_ : \texttt{Term} \gg \texttt{Form}, \_ : \ll x{:}\texttt{Term} \gg \mathscr{D}(P[x])) \to \mathscr{D}(\forall(P))\ ;\ x\#P$$

Finally, we consider the elimination of $\forall$, where $P[T]$ below is an instance of generalised concretion. It corresponds to a simple concretion on a fresh atom followed by a substitution of this atom by T:

$$\forall_e : (P : \ll \_ : \texttt{Term} \gg \texttt{Form}, T : \texttt{Term}, \_ : \mathscr{D}(\forall(P))) \to \mathscr{D}(P[T]).$$

3 Syntax

Building on the above, we propose the following syntax as a basis for a dependently sorted system.

3.1 Grammar

Consider a countably infinite set of name sorts, each one inhabited by a countably infinite set of names (*atoms*). Let a, b, c, range over atoms. Let also there be countably infinite sets of *parameters* X, Y, Z, $\ldots \in$

$\mathbb{X}$; *term constructors*, $f,\ g\ \ldots \in \mathbb{F}$; and *sort constructors* $\mathscr{F},\ \mathscr{G},\ \ldots \in \mathbb{C}$. Following Gabbay's *permutative convention* [11]: a, b range over *distinct atoms*. The notation $\overline{t}$ refers to a vector of terms $t_0,\ldots,t_n$ with $n \geqslant 0$; given a term t' and $\overline{t} = t_0,\ldots,t_n$, we use $\overline{t},t'$ to represent the vector $t_0,\ldots,t_n,t'$.

Sorts γ and terms t are generated by the grammar below. We will use M to stand for either.

$$
\begin{array}{lr}
\gamma ::= \mathscr{F}\,\overline{t} & \textit{data sorts} \\
\quad \mid \ll a : \mathscr{F}\,\overline{t} \gg \gamma & \textit{abstraction sorts} \\
t ::= a & \textit{atom} \\
\quad \mid X[\overline{t}] & \textit{parameter with term concretions} \\
\quad \mid f\,\overline{t} & \textit{application} \\
\quad \mid \ll a : \mathscr{F}\,\overline{t} \gg t & \textit{abstraction}
\end{array}
$$

Sorts are built using sort constructors or abstractions and can depend on terms, which can be atoms, parameters, application of a term constructor to a tuple of terms, or the abstraction of an atom on a term. We say an *expression* (sort or term) is *ground* iff it contains no parameters. When a parameter X has no concretions, we omit the square brackets. As stated earlier, parameters are intended for declarations, as shown in the previous section, while sorting judgments (i.e., the language generated by the system to be given in the next section) involve only ground expressions.

3.2 Operations and Relations

We define the action of permutations on sorts and terms. Here, $\overline{\pi \cdot t}$ denotes the vector $\pi \cdot t_0,\ldots,\pi \cdot t_n$.

Definition 1 (Permutation Action)**.** *A* permutation π *is a bijection on the set of atoms,* $\mathbb{A}$*, with finite domain. We represent permutations as lists of swappings* $(a\ b)$*. The identity permutation is written* id*.*

$$
\begin{array}{ccc}
\pi \cdot a \triangleq \pi(a) & \pi \cdot \ll a : \mathscr{F}\,\overline{s} \gg M \triangleq \ll \pi(a) : \mathscr{F}\,\overline{\pi \cdot s} \gg (\pi \cdot M) & \\
\pi \cdot (X[\overline{t}]) \triangleq X[\overline{\pi \cdot t}] & \pi \cdot f\,\overline{t} \triangleq f\,\overline{\pi \cdot t} & \pi \cdot \mathscr{F}\,\overline{t} \triangleq \mathscr{F}\,\overline{\pi \cdot t}
\end{array}
$$

To define alpha-equivalence, we first introduce the freshness relation. Call $a \# M$ a *freshness constraint*.

Definition 2 (Freshness Relation)**.** *A* freshness judgement *has the form* $\vdash a \# M$*. To derive freshness judgements we use the following rules. A premise* $\vdash a \# \overline{t}$ *is to be expanded as* $\vdash a \# t_0$*, …,* $\vdash a \# t_n$*.*

$$
(atm)^{\#}\ \frac{}{\vdash a \# b} \qquad (cns)^{\#}\ \frac{\vdash a \# \overline{t}}{\vdash a \# \mathscr{F}\,\overline{t}} \qquad (app)^{\#}\ \frac{\vdash a \# \overline{t}}{\vdash a \# f\,\overline{t}}
$$

$$
(ab_{aa})^{\#}\ \frac{\vdash a \# \mathscr{F}\,\overline{t}}{\vdash a \# \ll a : \mathscr{F}\,\overline{t} \gg M} \qquad (ab_{ab})^{\#}\ \frac{\vdash a \# M \qquad \vdash a \# \mathscr{F}\,\overline{t}}{\vdash a \# \ll b : \mathscr{F}\,\overline{t} \gg M} \qquad (var)^{\#}\ \frac{\vdash a \# \overline{t}}{\vdash a \# X[\overline{t}]}
$$

The main difference with respect to the freshness relation for standard nominal terms is the introduction of new rules $(ab_{aa})^{\#}$, $(ab_{ab})^{\#}$, $(cns)^{\#}$, and $(var)^{\#}$, the rule for concretion, which checks freshness only in terms and not in the parameter. As will be commented again later, parameters stand for arbitrary *closed* ground terms of the target language.

Definition 3 (Alpha-equivalence Relation)**.** *An* α-equivalence judgement *has the form* $\vdash M \approx_{\alpha} N$*, where M and N are ground. We introduce now the rules defining this relation. A premise* $\vdash \overline{s} \approx_{\alpha} \overline{t}$*, where* $\overline{s}$ *and* $\overline{t}$ *must always be of the same size, is to be expanded in an element-wise manner into premises* $\vdash s_i \approx_{\alpha} t_i$*.*

$$(atm)^\alpha\ \frac{}{\vdash a \approx_\alpha a} \qquad (cns)^\alpha\ \frac{\vdash \overline{s} \approx_\alpha \overline{t}}{\vdash \mathscr{F}\,\overline{s} \approx_\alpha \mathscr{F}\,\overline{t}} \qquad (app)^\alpha\ \frac{\vdash \overline{s} \approx_\alpha \overline{t}}{\vdash f\,\overline{s} \approx_\alpha f\,\overline{t}}$$

$$(ab_{aa})^\alpha\ \frac{\vdash \mathscr{F}\,\overline{t} \approx_\alpha \mathscr{F}\,\overline{u} \qquad \vdash M \approx_\alpha M'}{\vdash \ll a\colon \mathscr{F}\,\overline{t} \gg M \approx_\alpha \ll a\colon \mathscr{F}\,\overline{u} \gg M'}$$

$$(ab_{ab})^\alpha\ \frac{\vdash \mathscr{F}\,\overline{t} \approx_\alpha \mathscr{F}\,\overline{u} \qquad \vdash M \approx_\alpha (a\ b)\cdot M' \qquad \vdash a \# M'}{\vdash \ll a\colon \mathscr{F}\,\overline{t} \gg M \approx_\alpha \ll b\colon \mathscr{F}\,\overline{u} \gg M'}$$

This definition of alpha-equivalence generalises the standard one for nominal terms. For simplicity, we omit a rule for parameters, which is not essential but would facilitate the writing of declarations.

Lemma 1 (Equivariance)**.** *If* $\vdash a \# M$ *then* $\vdash \pi\cdot a \# \pi\cdot M$*. Similarly if* $\vdash M \approx_\alpha N$ *then* $\vdash \pi\cdot M \approx_\alpha \pi\cdot N$*.*

Proof. Straightforward induction. □

Freshness is stable by α-equivalence:

Lemma 2. *If* $\vdash a \# M$ *and* $\vdash M \approx_\alpha N$ *then* $\vdash a \# N$*.*

Proof. By induction on the freshness relation. Use equivariance. □

Lemma 3. $\approx_\alpha$ *is a congruence.*

Proof. Induction on the definition of $\approx_\alpha$. □

Definition 4 (Atom Substitution)**.** *We write* $[a \mapsto t]$ *for the operation that substitutes the atom a by the term t. This is defined on expressions as follows:*

$$\begin{aligned}
a\,[a \mapsto t] &\triangleq t & b\,[a \mapsto t] &\triangleq b\\
(f\,\overline{s})\,[a \mapsto t] &\triangleq f\,(\overline{s\,[a \mapsto t]}) & (\mathscr{F}\,\overline{s})\,[a \mapsto t] &\triangleq \mathscr{F}\,(\overline{s\,[a \mapsto t]})\\
(X\overline{[t]})\,[a \mapsto t'] &\triangleq X\overline{[t\,[a \mapsto t']]}\\
(\ll a\colon \mathscr{F}\,\overline{t} \gg M)\,[a \mapsto t] &\triangleq \ll a\colon \mathscr{F}\,\overline{t\,[a \mapsto t]} \gg M\\
(\ll b\colon \mathscr{F}\,\overline{t} \gg M)\,[a \mapsto t] &\triangleq \ll c\colon \mathscr{F}\,\overline{t\,[a \mapsto t]} \gg ((b\ c)\cdot M)\,[a \mapsto t] & (\vdash c \# M, \vdash c \# t).
\end{aligned}$$

Some explanations are in order: to avoid capturing unabstracted atoms, when an atom substitution acts upon an abstraction or abstraction sort (last case above), a suitable alpha-equivalent representative of the latter is first chosen. Any implementation of this definition as a recursive function must accommodate a suitable mechanism for the generation of names; this is most easily achieved by the threading of global state throughout the function or by the use of a global choice function that returns the next available name.

Atom substitutions work uniformly on alpha-equivalence classes.

Lemma 4. *If* $\vdash M \approx_\alpha N$ *and* $\vdash t \approx_\alpha u$ *then* $\vdash M[a \mapsto t] \approx_\alpha N[a \mapsto u]$

Proof. Induction on the derivation of $\vdash M \approx_\alpha N$. □

A concretion $w[t]$ is a partial operation: if w is an abstraction $\ll a : \mathscr{F}\,\overline{s} \gg u$, then its concretion to t evaluates to the body of the abstraction, u, where the abstracted atom is substituted by t. If w is the parameter X (possibly with other concretions suspended in it), the concretion remains "suspended" (until X is instantiated). Under the sorting system of the next section, concretion of a parameter will be well-sorted only if the parameter is of an (appropriate) abstraction sort.

Definition 5 (Concretion)**.** *Concretion is a partial operation:*

$$(\ll a : \mathscr{F}\,\overline{s} \gg u)[t] \triangleq u[a \mapsto t] \qquad\qquad (X[\overline{t}])[t'] \triangleq X[\overline{t}, t']$$

Definition 6 (Parameter Instantiation)**.** *A* parameter instantiation *is a finite mapping from parameters to terms, and it acts on expressions as just grafting (i.e., without a control of capture), subject to the condition that each parameter to be replaced is in the domain of the instantiation.*

4 Sorting judgements

We use five forms of judgements: 1) Well-formedness of signature Σ, formally $\vdash \Sigma$ sig-ok (Fig. 1a); 2) Well-formedness of telescopes $\mathscr{T}$ under a valid signature, $\vdash_\Sigma \mathscr{T}$ tel-ok(Fig. 1b); 3) Well-formedness of contexts of atoms (Fig. 1c), $\mathscr{T} \vdash_\Sigma \Gamma$ ctx-ok; 4) Well-formedness of sorts (Fig. 1d), $\mathscr{T}; \Gamma \vdash_\Sigma \gamma$ sort; and 5) Well-sortedness of terms (Fig. 1e), $\mathscr{T}; \Gamma \vdash_\Sigma t : \gamma$.

As indicated above, the sorts are either *data sorts* or *abstraction sorts*. Data sorts are introduced by *sort constructors* $\mathscr{F}$, and these can only introduce data sorts, never an abstraction sort —the latter being formed exclusively by the binder $\ll\_ : \_ \gg \_$. Similarly, terms of the data sorts are formed by *(term) constructors* f, and terms of abstraction sorts exclusively by the corresponding binder. *Signatures* are sequences of *declarations* of sort and term constructors. As already explained, a declaration specifies the sorts of the corresponding parameters and a freshness context. These parameter declarations are called *telescopes*. The word *context* is reserved for *atom contexts*, Γ, necessary to sort abstractions.

As already stated, the intention is that the system is used for generating well-formed *ground* sorts and terms. The rules given below define well-formed scripts of declarations (i.e. signatures), which involve not only ground expressions but also expressions with parameters.

First, notice the use of *freshness contexts* (Δ) in declarations. They involve conditions of the form $a \# X$, where the atom a is to appear bound in the declaration and X is any parameter of the declaration.

This defines the side condition on well-formedness of the contexts Δ. The rules check the validity of the freshness conditions whenever a declaration is put into use, i.e. in rules (data) and (constr). There the constructor employed must be declared in the signature with a telescope $\mathscr{T}'$ and freshness context Δ, as stated in the side condition. Then a fresh version of $\mathscr{T}'$, as well as of Δ, are created by employing new atoms so as to avoid possible collisions with unabstracted atoms in the expression being checked. We call this new telescope $\mathscr{T}'_\#$, and the new context $\Delta_\#$. Then it is checked that the tuple $\overline{t}$ of arguments *fits* the telescope $\mathscr{T}'_\#$ and at the same time the conditions in $\Delta_\#$ are satisfied, with the mentioned parameters instantiated accordingly by the tuple $\overline{t}$ —which we write $(\Delta_\#)_{\overline{t}}$. That a tuple of terms *fits* a telescope has the (obvious) meaning that: a) The telescopes and the context are well-formed. b) They are of the same length. c) Each term has the sort attached to its corresponding parameter, instantiated on the preceding terms in the tuple.

An equally valid alternative is that the freshness conditions are rather *imposed* by the system, i.e. a freshness declaration is to be interpreted as an *assumption* on part of the user about the employment of names in the (ground) expressions to be generated. The conditions can be imposed by the system by generating in each case a sample chosen among all the alpha-equivalent expressions satisfying the sorting rules that also respects the freshness conditions. For this to work, it is essential that the system is closed under alpha-equivalence —which will be shown presently— and that the freshness conditions are only on bound atoms —which is already imposed in the well-formation of declarations.

In the rule (constr) we use the notation $\overline{u}_{\overline{t}}$, which stands for the instantiation of the parameters of the tuple of terms $\overline{u}$ with the tuple $\overline{t}$.

Finally, let us remark that, as stated in rule (fun-sig) and (cons-tel), valid telescopes and target sorts of term constructors cannot depend on (unabstracted) atoms. Also note that in the rules we omit premises that can be deduced from some explicitly mentioned premise.

$$\text{(empty-sig)}\ \frac{}{\vdash \langle\rangle\ \text{sig-ok}} \qquad \text{(sort-sig)}\ \frac{\vdash_\Sigma \mathscr{T}\ \text{tel-ok}}{\vdash \Sigma, \langle \mathscr{F} \colon \mathscr{T} \to \mathsf{data}\,;\Delta\rangle\ \text{sig-ok}} \begin{cases} \mathscr{F} \notin \mathsf{dom}(\Sigma) \\ \Delta\ \text{well-formed} \end{cases}$$

$$\text{(fun-sig)}\ \frac{\mathscr{T};\cdot \vdash_\Sigma \mathscr{F}\,\overline{t}\ \ \mathsf{sort}}{\vdash \Sigma, \langle f \colon \mathscr{T} \to \mathscr{F}\,\overline{t}\,;\Delta\rangle\ \text{sig-ok}} \begin{cases} f \notin \mathsf{dom}(\Sigma) \\ \Delta\ \text{ well-formed} \end{cases}$$

(a) *Rules for signatures.*

$$\text{(empty-tel)}\ \frac{\vdash \Sigma\ \text{sig-ok}}{\vdash_\Sigma \cdot\ \text{tel-ok}} \qquad \text{(cons-tel)}\ \frac{\mathscr{T};\cdot \vdash_\Sigma \gamma\ \ \mathsf{sort}}{\vdash_\Sigma \mathscr{T}, (X : \gamma)\ \text{tel-ok}}\ X \notin \mathsf{dom}(\mathscr{T})$$

(b) *Rules for telescope formation.*

$$\text{(emp-ctx)}\ \frac{\vdash_\Sigma \mathscr{T}\ \text{tel-ok}}{\mathscr{T} \vdash_\Sigma \cdot\ \text{ctx-ok}} \qquad \text{(cons-ctx)}\ \frac{\mathscr{T};\Gamma \vdash_\Sigma \mathscr{F}\,\overline{t}\ \ \mathsf{sort}}{\mathscr{T} \vdash_\Sigma \Gamma, (a : \mathscr{F}\,\overline{t})\ \text{ctx-ok}}\ a \notin \mathsf{dom}(\Gamma)$$

(c) *Rules for well-formed contexts.*

$$\text{(data)}\ \frac{\mathscr{T};\Gamma \vdash_\Sigma \overline{t}\ \text{fits}\ \mathscr{T}'_{\#}[(\Delta_{\#})_{\overline{t}}]}{\mathscr{T};\Gamma \vdash_\Sigma \mathscr{F}\,\overline{t}\ \ \mathsf{sort}} \begin{cases} \mathscr{F} \in \mathsf{dom}(\Sigma) \\ \Sigma(\mathscr{F}) = \mathscr{T}' \to \mathsf{data};\Delta \end{cases}$$

$$\text{(abs-*)}\ \frac{\mathscr{T};\Gamma \vdash_\Sigma \mathscr{F}\,\overline{t}\ \ \mathsf{sort} \quad \mathscr{T};(\Gamma, b : \mathscr{F}\,\overline{t}) \vdash_\Sigma (a\ b)\cdot\gamma\ \ \mathsf{sort}}{\mathscr{T};\Gamma \vdash_\Sigma \ll a : \mathscr{F}\,\overline{t} \gg \gamma\ \ \mathsf{sort}} \begin{cases} b \notin \mathsf{dom}(\Gamma) \\ b\,\#\,\gamma \end{cases}$$

(d) *Rules for well-formed sorts.*

$$\text{(atm)}\ \frac{\mathscr{T} \vdash_\Sigma \Gamma\ \text{ctx-ok}}{\mathscr{T};\Gamma \vdash_\Sigma a : \Gamma(a)}\ a \in \mathsf{dom}(\Gamma) \qquad \text{(var1)}\ \frac{\mathscr{T} \vdash_\Sigma \Gamma\ \text{ctx-ok}}{\mathscr{T};\Gamma \vdash_\Sigma X : \mathscr{T}(X)}\ X \in \mathsf{dom}(\mathscr{T})$$

$$\text{(var2)}\ \frac{\mathscr{T};\Gamma \vdash_\Sigma X[\overline{t}] : \ll a : \mathscr{F}\,\overline{s} \gg \gamma \qquad \mathscr{T};\Gamma \vdash_\Sigma t' : \mathscr{F}\,\overline{s}}{\mathscr{T};\Gamma \vdash_\Sigma X[\overline{t}, t'] : \gamma[a \mapsto t']}$$

$$\text{(constr)}\ \frac{\mathscr{T};\Gamma \vdash_\Sigma \overline{t}\ \text{fits}\ \mathscr{T}'_{\#}[(\Delta_{\#})_{\overline{t}}]}{\mathscr{T};\Gamma \vdash_\Sigma f\,\overline{t} : \mathscr{F}\,(\overline{u_{\overline{t}}})} \begin{cases} f \in \mathsf{dom}(\Sigma) \\ \Sigma(f) = \mathscr{T}' \to \mathscr{F}\,\overline{u};\Delta \end{cases}$$

$$\text{(abs)}\ \frac{\mathscr{T};\Gamma \vdash_\Sigma \mathscr{F}\,\overline{t}\ \ \mathsf{sort} \qquad \mathscr{T};(\Gamma, b : \mathscr{F}\,\overline{t}) \vdash_\Sigma (a\ b)\cdot t : (a\ b)\cdot\gamma}{\mathscr{T};\Gamma \vdash_\Sigma \ll a : \mathscr{F}\,\overline{t} \gg t : \ll a : \mathscr{F}\,\overline{t} \gg \gamma} \begin{cases} b \notin \mathsf{dom}(\Gamma) \\ b\,\#\,\{t, \gamma\} \end{cases}$$

$$\text{(conv)}\ \frac{\cdot;\Gamma \vdash_\Sigma t : \gamma}{\cdot;\Gamma \vdash_\Sigma t : \gamma'}\ \gamma \approx_\alpha \gamma'$$

(e) *Rules for well-sorted terms.*

Figure 1: Sorting System

Properties of the sorting system

Lemma 5 (Equivariance of sorting)**.** *Let $\mathscr{J}'$ be a permutative variant of the judgement $\mathscr{J}$. If one is derivable, then the other is also derivable.*

Proof. Direct check on the system by simultaneous induction over all forms of judgments. □

Lemma 6 (Closure under alpha conversion)**.**

1. *If* $\cdot;\Gamma \vdash_\Sigma \gamma$ sort *then for every* γ' *such that* $\vdash \gamma' \approx_\alpha \gamma$*, also* $\cdot;\Gamma \vdash_\Sigma \gamma'$ sort.
2. *If* $\cdot;\Gamma \vdash_\Sigma t : \gamma$ *then for every* t' *such that* $\vdash t' \approx_\alpha t$*, also* $\cdot;\Gamma \vdash_\Sigma t' : \gamma$.

Proof. By induction on the sorting system, essentially using rule (conv) and equivariance. □

Since sort inference and alpha-equivalence are decidable, sort-checking is decidable.

Lemma 7 (Decidability)**.** *Given a signature* Σ*, telescope* $\mathscr{T}$ *and context* Γ *valid under* Σ*, it is decidable whether* M *is a sort or a term of some sort, for any* M*.*

5 Representation of calculi

5.1 First Order Logic

Normally one introduces *first-order* languages, each one determined by a choice of function and predicate symbols. This would then call for a specification parameterised on such symbol declarations. While this kind of parameterisation could be incorporated into the framework, we prefer for now to keep matters simple and provide a representation of *first-order arithmetic*. Another choice we make is to represent a *classical* version of the logic.

Syntax

Presentation. In the following x represents a variable taken from a denumerable set.

$$t ::= x \mid 0 \mid S\,t \mid t_1 + t_2 \mid t_1 \times t_2 \qquad \text{Terms}$$
$$\varphi ::= t_1 = t_2 \mid \bot \mid \neg\varphi \mid \varphi_1 \supset \varphi_2 \mid (\forall x)\varphi \qquad \text{Formulæ}$$

The notion of free variable for terms and formulae $fv(t)$ and $fv(\varphi)$ are defined as usual.

Let us call *expressions* (denoted by e) either terms or formulae of the first-order language being considered. We write $\equiv$ for identity on terms and formulas. This is the congruent-closure of α-conversion, which in turn is the diagonal on terms and defined by the following rule on formulae:

$$\frac{\varphi^x_z \equiv \varphi'^y_z}{(\forall x)\varphi \equiv (\forall y)\varphi'}\, z \notin fv(\varphi)$$

Here φ^x_z is the *swapping* of the occurrences of x and z in φ. Notice $\equiv$ is clearly decidable. Now define substitution variable by a term on terms and formulas, in the usual way, using the clause:

$$((\forall x)\varphi)[y := t] \equiv (\forall x)(\varphi[y := t]) \qquad x \notin fv(t \cup \{y\})$$

Justification of well-definedness of substitutions is routine, assuming the clause:

$$\varphi[x := t] \equiv \varphi'[x := t] \text{ if } \varphi \equiv \varphi'^{1}$$

This level of detail is usually not reached in textbook presentations, and actually neither in e.g. [15]. Other alternatives could have been chosen – this one is direct and simple enough.

[1] This is the same as identifying formulae up to α-conversion, or working on α-classes.

Encoding. Clearly, we have two *sorts* of expressions:

$$\texttt{Term} : \texttt{data} \qquad \text{and} \qquad \texttt{Form} : \texttt{data}$$

Look now at terms. Each variable x_i will be encoded as a (distinct) atom a_i (that is to carry the sort `Term`). Besides, we introduce constructors and operations:

$$\begin{array}{ll} 0 : \texttt{Term} & S : (\_ : \texttt{Term}) \to \texttt{Term} \\ + : (\_ : \texttt{Term}, \_ : \texttt{Term}) \to \texttt{Term} & \times : (\_ : \texttt{Term}, \_ : \texttt{Term}) \to \texttt{Term} \end{array}$$

As usual, we overload the symbols on the object- and meta-levels. As to formulae:

$$\begin{array}{lll} = : (\_ : \texttt{Term}, \_ : \texttt{Term}) \to \texttt{Form} & \bot : \texttt{Form} & \neg : (\_ : \texttt{Form}) \to \texttt{Form} \\ \supset : (\_ : \texttt{Form}, \_ : \texttt{Form}) \to \texttt{Form} & \forall : (\ll \_ : \texttt{Term} \gg \texttt{Form}) \to \texttt{Form} & \end{array}$$

Notice that binding in the object language is represented using the abstraction construct of the framework.

Adequacy. Let us call Σ the signature just introduced. For any finite set $\mathscr{X}$ of variables $x_1, \ldots, x_n$, let $\hat{\mathscr{X}}$ be a context of our nominal framework (call this NF) containing assumptions $a_i : \mathsf{Term}$ iff $x_i \in \mathscr{X}$.

Lemma 8. *There is a compositional bijection*[2] *between:*

- *Terms t of the FOL-calculus and terms $\hat{t}$ of the NF such that $\cdot ; \widehat{fv(t)} \vdash_\Sigma \hat{t} : \mathsf{Term}$*
- *Formulae φ of the FOL-calculus and terms $\hat{\varphi}$ of the NF such that $\cdot ; \widehat{fv(\varphi)} \vdash_\Sigma \hat{\varphi} : \mathsf{Form}$*

Proof. First we define encoding `enc` as follows, by recursion on FOL-terms (we use $\approx$ for term identity in NF, which includes $\approx_\alpha$):

$$\begin{array}{rlrl} \texttt{enc}(x_i) & \approx a_i & \texttt{enc}(t_1 = t_2) & \approx = (\texttt{enc}(t_1), \texttt{enc}(t_2)) \\ \texttt{enc}(0) & \approx 0 & \texttt{enc}(\bot) & \approx \bot \\ \texttt{enc}(St) & \approx S(\texttt{enc}(t)) & \texttt{enc}(\neg \varphi) & \approx \neg(\texttt{enc}(\varphi)) \\ \texttt{enc}(t_1 + t_2) & \approx +(\texttt{enc}(t_1), \texttt{enc}(t_2)) & \texttt{enc}(\varphi_1 \supset \varphi_2) & \approx \supset (\texttt{enc}(\varphi_1), \texttt{enc}(\varphi_2)) \\ \texttt{enc}(t_1 \times t_2) & \approx \times(\texttt{enc}(t_1), \texttt{enc}(t_2)) & \texttt{enc}((\forall x_i)\varphi) & \approx \forall(\ll a_i : \texttt{Term} \gg \texttt{enc}(\varphi)) \end{array}$$

Next, we show that $e_1 \equiv e_2 \iff \cdot \vdash \texttt{enc}(e_1) \approx \texttt{enc}(e_2)$ (notice that `enc` is ground for every e, so there's no need to consider freshness contexts on the right-hand side).
Define `dec`, the inverse to `enc`, as follows:

$$\begin{array}{rlrl} \texttt{dec}(a_i) & \equiv x_i & \texttt{dec}(= (\hat{t}_1, \hat{t}_2)) & \equiv \texttt{dec}(\hat{t}_1) = \texttt{dec}(\hat{t}_2) \\ \texttt{dec}(0) & \equiv 0 & \texttt{dec}(\bot) & \equiv \bot \\ \texttt{dec}(S(\hat{t})) & \equiv S(\texttt{dec}(\hat{t})) & \texttt{dec}(\supset (\hat{\varphi}_1, \hat{\varphi}_2)) & \equiv \texttt{dec}(\hat{\varphi}_1) \supset \texttt{dec}(\hat{\varphi}_2) \\ \texttt{dec}(+(\hat{t}_1, \hat{t}_2)) & \equiv \texttt{dec}(\hat{t}_1) + \texttt{dec}(\hat{t}_2) & \texttt{dec}(\forall(\ll a_i : \texttt{Term} \gg \hat{\varphi})) & \equiv (\forall x_i)\, \texttt{dec}(\hat{\varphi}) \\ \texttt{dec}(\times(\hat{t}_1, \hat{t}_2)) & \equiv \texttt{dec}(\hat{t}_1) \times \texttt{dec}(\hat{t}_2) & & \end{array}$$

It follows `enc` is a bijection with inverse `dec`. It is straightforward to prove, by induction on terms and formulae, that the sorting judgments hold using the signature given above.

Compositionality is given by: $\texttt{enc}(e[x_i{:=}t]) \approx (\texttt{enc}(e))[a_i \mapsto \texttt{enc}(t)]$ □

[2] Compositional means that substitution commutes with encoding, as introduced in [15].

Derivations

Presentation. We choose Natural Deduction. Let contexts Γ of assumptions be finite sets of formulas. Write Γ, φ for $\Gamma \cup \{\varphi\}$ with $\varphi \notin \Gamma$, and extend fv to contexts in the obvious way.

$$\text{(ass)}\ \frac{}{\Gamma \vdash \varphi}\ \varphi \in \Gamma \qquad (\rho)\ \frac{}{\Gamma \vdash t = t} \qquad (\sigma)\ \frac{\Gamma \vdash t_1 = t_2 \quad \Gamma \vdash \varphi[x_i := t_1]}{\Gamma \vdash \varphi[x_i := t_2]}$$

$$(+_0)\ \frac{}{\Gamma \vdash 0 + t = t} \qquad (+_S)\ \frac{}{\Gamma \vdash s(t_1) + t_2 = s(t_1 + t_2)}$$

$$(\times_0)\ \frac{}{\Gamma \vdash 0 \times t = 0} \qquad (\times_S)\ \frac{}{\Gamma \vdash s(t_1) \times t_2 = (t_1 \times t_2) + t_2}$$

$$\text{(ind)}\ \frac{\Gamma \vdash \varphi[x := 0] \quad \Gamma, \varphi \vdash \varphi[x := sx]}{\Gamma \vdash \varphi[x := t]}\ x \notin FV(\Gamma)$$

$$(\bot_e)\ \frac{\Gamma \vdash \bot}{\Gamma \vdash \varphi}\ \varphi \in \Gamma \qquad (\neg\,\text{i})\ \frac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi} \qquad (\neg\,\text{e})\ \frac{\Gamma \vdash \varphi \quad \Gamma \vdash \neg\varphi}{\Gamma \vdash \bot}$$

$$\text{(RAA)}\ \frac{\Gamma, \neg\varphi \vdash \bot}{\Gamma \vdash \varphi} \qquad (\supset\text{i})\ \frac{\Gamma, \varphi_1 \vdash \varphi_2}{\Gamma \vdash \varphi_1 \supset \varphi_2} \qquad (\supset\text{e})\ \frac{\Gamma \vdash \varphi_1 \supset \varphi_2 \quad \Gamma \vdash \varphi_1}{\Gamma \vdash \varphi_2}$$

$$(\forall\text{i})\ \frac{\Gamma \vdash \varphi}{\Gamma \vdash (\forall x_i)\varphi}\ x_i \notin FV(\Gamma) \qquad (\forall\text{e})\ \frac{\Gamma \vdash (\forall x)\varphi}{\Gamma \vdash \varphi[x := t]}$$

Encoding. Derivations of judgements $\Gamma \vdash \varphi$ can be seen as derivations of judgements φ proceeding from assumptions consisting of single formulae, in such a way that the set of assumptions is included in Γ. We introduce a family of sorts $\mathscr{D} : \texttt{Form} \to \texttt{data}$ for classifying these latter derivations.

So each derivation of the formula φ will be represented as a term in $\mathscr{D}(\varphi)$. The rules of the system shall be term-formers of these various sorts, in the following way: 1) Assumptions: Each assumption of formula φ shall be represented as a distinct atom $h : \mathscr{D}(\varphi)$. These have to be chosen also distinct from the ones representing free variables of the formulae participating in any derivation. 2) Premises: are parameters of the corresponding sorts. 3) Discharge: corresponds to binding, i.e., the discharged assumption(s) are bound to the rule(-occurrence) that discharges it (them). 4) Additional parameters like terms, must be appropriately declared as parameters of the corresponding constructor. One special case is variables chosen to be replaced (i.e. in rule σ or (ind)). These are encoded as atoms possibly appearing in the encoding of the relevant formula. Clearly, in this case the variable is merely a pointer to a place in the formula where to perform the substitution of the relevant term. Accordingly, we encode this phenomenon using our abstraction operator of the framework. That is to say that the relevant formula and the variable in question are encoded as an abstraction.

In the following encoding, we collapse declarations of parameters of the same sort (so, instead of writing $t_1 : \texttt{Term}, t_2 : \texttt{Term}$ we write $t_1, t_2 : \texttt{Term}$).

$$\rho : (t : \texttt{Term}) \to \mathscr{D}(= (t,t))$$

$$+_0 : (t : \texttt{Term}) \to \mathscr{D}(= (+(0,t),t)) \qquad +_S : (t_1, t_2 : \texttt{Term}) \to \mathscr{D}(= (+(S(t_1),t_2)), S(+(t_1,t_2)))$$

$$\times_0 : (t : \texttt{Term}) \to \mathscr{D}(= (\times(0,t),0)) \qquad \times_S : (t_1, t_2 : \texttt{Term}) \to \mathscr{D}(= (\times(S(t_1),t_2)), (+(\times(t_1,t_2),t2)))$$

$$\sigma : (\varphi{:}\ \ll\_{:}\texttt{Term}\gg \texttt{Form}, t_1,\ t_2 : \texttt{Term}, \_ : \mathscr{D}(= (t_1,t_2)), \_ : \mathscr{D}(\varphi[t_1])) \to \mathscr{D}(\varphi[t_2])$$

$$\text{ind} : (\varphi{:}\ \ll\_{:}\texttt{Term}\gg \texttt{Form}, P0 : \mathscr{D}(\varphi[0]),$$
$$PS{:}\ \ll x{:}\texttt{Term}\gg\ll\_{:}\mathscr{D}(\varphi[x])\gg \mathscr{D}(\varphi[S(x)]), t{:}\texttt{Term}) \to \mathscr{D}(\varphi[t])\ ;\ x\#\varphi$$

$$\bot_e : (\varphi : \texttt{Form}, \_ : \mathscr{D}(\bot)) \to \mathscr{D}(\varphi) \qquad \neg_i : (\varphi{:}\texttt{Form}, \_ : \ll\_{:}\mathscr{D}(\varphi)\gg \mathscr{D}(\bot)) \to \mathscr{D}(\neg(\varphi))$$

$\neg_e : (\varphi : \mathtt{Form}, \_ : \mathscr{D}(\varphi), \_ : \mathscr{D}(\neg(\varphi))) \to \mathscr{D}(\bot)$
$\supset_i : (\varphi_1, \varphi_2 : \mathtt{Form}, \_ : \ll \_ : \mathscr{D}(\varphi 1) \gg \mathscr{D}(\varphi 2)) \to \mathscr{D}(\supset(\varphi_1, \varphi_2))$
$\supset_e : (\varphi_1, \varphi_2 : \mathtt{Form}, \_ : \mathscr{D}(\supset(\varphi_1, \varphi_2)), \_ : \mathscr{D}(\varphi_1)) \to \mathscr{D}(\varphi_2)$
$\forall_i : (\varphi : \ll \_ : \mathtt{Term} \gg \mathtt{Form}, \_ : \ll x : \mathtt{Term} \gg \mathscr{D}(\varphi[x])) \to \mathscr{D}(\forall(\varphi))\ ;\ x \# \varphi$
$\forall_e : (\varphi : \ll \_ : \mathtt{Term} \gg \mathtt{Form}, t : \mathtt{Term}, \_ : \mathscr{D}(\forall(\varphi))) \to \mathscr{D}(\varphi[t])$

Adequacy. To prove adequacy of derivations we need to map derivation trees of judgements $\Gamma \vdash \varphi$ to terms of sort $\mathscr{D}(\hat{\varphi})$ in the framework. These terms are to depend on contexts containing declarations for atoms of sort `Term` in $\hat{\varphi}$ (which correspond to free variables in φ) and for atoms corresponding to the undischarged assumptions of the derivation. These, in turn, must be preceded by declarations of atoms of sort `Term` that correspond to free variables of the formulas in Γ. Besides, the atoms corresponding to the assumptions can be chosen somewhat arbitrarily, as long as they are different enough to ensure well-formation of the context in question. Thus, for each derivation of judgement $\Gamma \vdash \varphi$ and set of atoms H of size equal to Γ, we can form a context $\widehat{fv(\Gamma)}, \Gamma_H$, where $\widehat{fv(\Gamma)}$ is as before (adequacy of terms) and Γ_H associates to each $h \in H$ the sort $\mathscr{D}(\hat{\varphi})$ for φ a different formula in Γ. We require H disjoint from the set of atoms declared in $\widehat{fv(\Gamma)}$. Now we can formulate:

Lemma 9. *For each set H of atoms such that $|H| = |\Gamma|$, there exists a bijective correspondence between derivations δ of judgements $\Gamma \vdash \varphi$ in FOL and terms $\hat{\delta}$ of NF such that: $\cdot ; \widehat{fv(\Gamma)} \vdash \hat{\delta} : \mathscr{D}(\hat{\varphi})$.*

Let us make two remarks. First, we are still considering only ground terms of the framework. Second, we consider identity of derivations in FOL to be given freely by the rules-as-constructors.

Proof. By induction on derivations. We define $\hat{\delta}$ as $\delta\mathtt{enc}$ with parameter H as follows:

Derivation, δ	Encoding, $\delta\mathtt{enc}_H(\delta)$	Comment
(ass) $\dfrac{}{\Gamma \vdash \varphi}$	h where $(h : \mathscr{D}(\hat{\varphi}) \in \Gamma_H)$	
$(\rho)\ \dfrac{}{\Gamma \vdash t = t}$	$\rho(\hat{t})$	
$(\sigma)\ \dfrac{\Gamma \vdash t_1 \overset{\delta_1}{=} t_2 \quad \Gamma \vdash \varphi[x_i := t_1] \overset{\delta_2}{}}{\Gamma \vdash \varphi[x_i := t_2]}$	$\sigma(\ll x : \mathtt{Term} \gg \hat{\varphi},\ \hat{t_1},\ \hat{t_2}, \delta\mathtt{enc}_H(\delta_1), \delta\mathtt{enc}_H(\delta_2))$	
$(+_0)\ \dfrac{}{\Gamma \vdash 0 + t = t}$	$+_0(\hat{t})$	similarly for $+_S, \times_0, \times_S$
$(\mathrm{ind})\ \dfrac{\Gamma \vdash \varphi[x := 0] \overset{\delta_0}{} \quad \Gamma, \varphi \vdash \varphi[x := S(x)] \overset{\delta_1}{}}{\Gamma \vdash \varphi[x := t]}$	$\mathtt{ind}(\ll a_i : \mathtt{Term} \gg \hat{\varphi}, \delta\mathtt{enc}_H(\delta_0), \ll a_i : \mathtt{Term} \gg \ll h : \mathscr{D}(\hat{\varphi}) \gg \delta\mathtt{enc}_{H \cup \{h\}}(\delta_1), t)$	$h \notin H$
$(\bot_e)\ \dfrac{\Gamma \vdash \bot \overset{\delta}{}}{\Gamma \vdash_\delta \varphi}$	$\bot_e(\hat{\varphi}, \delta\mathtt{enc}_H(\delta))$	similarly for $\neg_e, \supset_e$
$(\neg_i)\ \dfrac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi}$	$\neg_i(\hat{\varphi}, \ll h : \mathscr{D}(\varphi) \gg \delta\mathtt{enc}_{H \cup \{h\}}(\delta))$	$h \notin H$ similarly for RAA, $\supset_i$
$(\forall_i)\ \dfrac{\Gamma \vdash \varphi \overset{\delta}{}}{\Gamma \vdash (\forall x_i)\varphi}\ x_i \notin fv(\Gamma)$	$\forall_i(\ll a_i : \mathtt{Term} \gg \hat{\varphi}, \ll a_i : \mathtt{Term} \gg \delta\mathtt{enc}_H(\delta))$	
$(\forall_e)\ \dfrac{\Gamma \vdash (\forall x_i)\varphi \overset{\delta}{}}{\Gamma \vdash \varphi[x_i := t]}$	$\forall_e(\ll a_i : \mathtt{Term} \gg \hat{\varphi}, \hat{t}, \delta\mathtt{enc}_H(\delta))$	

Now we show that the typing restriction holds in the framework. We do it for two interesting cases, namely assumption and induction:

(ass) We need to show $\widehat{fv(\Gamma)},\Gamma_H \vdash h : \mathscr{D}(\hat{\varphi})$. But $h : \mathscr{D}(\hat{\varphi})$ is in Γ_H by construction, and the whole context is well-formed by construction too.

(ind) First of all, $\ll a_i{:}\mathtt{Term}\gg \hat{\varphi}$ is of sort $\ll a_i{:}\mathtt{Term}\gg \mathtt{Form}$. Also, by induction hypothesis, $\delta\mathtt{enc}_H(\delta_0)$ is of sort $\mathscr{D}(\widehat{\varphi[x_i := 0]})$ under $\widehat{fv(\Gamma)},\Gamma_H$. We get $\widehat{\varphi[x := 0]} \approx \hat{\varphi}[a \mapsto 0] \approx (\ll a_i{:}\mathtt{Term}\gg \hat{\varphi})[0]$ by Lemma 8. Therefore the second argument for the constructor `ind` is well-typed. Next, let us check that $\ll a_i{:}\mathtt{Term}\gg\ll h{:}\mathscr{D}(\hat{\varphi})\gg \delta\mathtt{enc}_{H\cup\{h\}}(\delta_1)$, has sort $\ll a_i{:}\mathtt{Term}\gg\ll h{:}\mathscr{D}(\hat{\varphi})\gg \mathscr{D}(\widehat{\varphi[x_i := S(x_i)]})$. Then we know both $\hat{\varphi} \approx (\ll a_i{:}\mathtt{Term}\gg \hat{\varphi})[a_i]$ and $\widehat{\varphi[x_i := S(x_i)]} \approx \hat{\varphi}[a_i \mapsto S(a_i)] \approx (\ll a_i{:}\mathtt{Term}\gg \hat{\varphi})[S(a_i)]$, as desired.

Clearly, the mapping is one-to-one. The converse is shown by defining a decoding mapping. The idea is to map a term $\hat{\delta}$ such that $\cdot;\widehat{fv(\Gamma)} \vdash \hat{\delta} : \mathscr{D}(\hat{\varphi})$ for some $\hat{\varphi}$ such that $\cdot;\hat{\Gamma} \vdash \hat{\varphi} : \mathtt{Form}$, to a derivation of $\Gamma \vdash \varphi$ for appropriate Γ. □

5.2 Lambda Calculi

The Pure Calculus

Here we give a shallow encoding (see the deep version below). We have to introduce a sort $\Lambda : \mathtt{data}$ for terms, after which we get the atoms, which in the present version will represent the usual variables of the calculus. Then it remains to declare:

$$@ : (\_ : \Lambda,\ \_ : \Lambda) \to \Lambda \ \text{(for application)} \ \text{ and } \ \lambda : (\_ : \ll \_ : \Lambda \gg \Lambda) \to \Lambda \ \text{(for functional abstraction).}$$

Note that β-contraction of redexes, usually denoted by $\triangleright$, is a binary relation to be encoded as:

$$\triangleright : (\_ : \Lambda,\ \_ : \Lambda) \to \mathtt{data}$$

with one rule: $$\beta : (B : \ll \_ : \Lambda \gg \Lambda, N : \Lambda) \to \triangleright(@(\lambda(B), N), B[N]),$$
using the generalised concretion of the framework.

Adequacy. To show the correctness of our encoding of the λ-calculus, we need to prove that we can map β-reductions $s \to_\beta t$ in the λ-calculus to terms of type $\triangleright(\mathtt{enc}(s), \mathtt{enc}(t))$ in our framework.

Lemma 10. *If s,t are λ-terms such that $s \to_\beta t$, i.e., $s \equiv (\lambda x.s_1)s_2$ and $t \equiv s_1[x := s_2]$ then*
$\vdash \beta(\ll a_i{:}\Lambda \gg \mathit{enc}(s_1), \mathit{enc}(s_2)) : \triangleright(@(\lambda(\ll a_i{:}\Lambda \gg \mathit{enc}(s_1)), \mathit{enc}(s_2)), (\ll a_i{:}\Lambda \gg \mathit{enc}(s_1))[\mathit{enc}(s_2)]$.

The Simply Typed Calculi

We give a system in Church's monomorphic style. We introduce a sort $\Lambda^{\to}\mathtt{Type} : \mathtt{data}$ for types with constructors:

$$\iota : \Lambda^{\to}\mathtt{Type} \qquad \text{— any ground type}$$
$$\Rightarrow : (\_ : \Lambda^{\to}\mathtt{Type}, \_ : \Lambda^{\to}\mathtt{Type}) \to \Lambda^{\to}\mathtt{Type} \qquad \text{— the functional types}$$

Typed terms is a sort family $\Lambda\mathtt{Term} : (\_ : \Lambda^{\to}\mathtt{Type}) \to \mathtt{data}$ indexed by types, with constructors:

$$@ : (\alpha : \Lambda^{\to}\mathtt{Type}, \beta : \Lambda^{\to}\mathtt{Type}, \_ : \Lambda\mathtt{Term}(\Rightarrow(\alpha,\beta)), \_ : \Lambda\mathtt{Term}(\alpha)) \to \Lambda\mathtt{Term}(\beta)$$

$$\lambda : (\alpha : \Lambda^{\rightarrow}\mathtt{Type}, \beta : \Lambda^{\rightarrow}\mathtt{Type}, \_ : \ll \_ : \Lambda\mathtt{Term}(\alpha) \gg \Lambda\mathtt{Term}(\beta)) \rightarrow \Lambda\mathtt{Term}(\Rightarrow(\alpha, \beta))$$

Contraction gets typed (because generalized concretion of the framework preserves typing):

$$\begin{array}{l} \triangleright : (\beta : \Lambda^{\rightarrow}\mathtt{Type}, \_ : \Lambda\mathtt{Term}(\beta), \_ : \Lambda\mathtt{Term}(\beta)) \rightarrow \mathtt{data} \\ \beta : (\alpha : \Lambda^{\rightarrow}\mathtt{Type}, \beta : \Lambda^{\rightarrow}\mathtt{Type}, B : \ll \_ : \Lambda\mathtt{Term}(\alpha) \gg \Lambda\mathtt{Term}(\beta), N : \Lambda\mathtt{Term}(\alpha)) \\ \quad \rightarrow \triangleright(\beta, @(\alpha, \beta, \lambda(\alpha, \beta, B), N), B[N]) \end{array}$$

There is a difficulty with encoding Curry's style system — which has to do with the representation of contexts assigning types to variables. Indeed, since variables in our encoding are atoms of sort Λ in the framework context, we would need to somehow zip the context to a list of $\Lambda^{\rightarrow}\mathtt{Types}$. This seems to call for a deeper embedding.

Given the encoding of first-order logic and the lambda-calculus, it is not surprising that the system can also encode Higher-Order Logic (omitted due to lack of space). It is worth mentioning that also versions of dependently typed lambda calculi can be represented.

Deep Embeddings

We show an alternative encoding of the pure lambda-calculus, which corresponds to a so-called *deep* embedding, in contrast to the one given at the beginning of this section. The difference has to do fundamentally with the status of *substitution*. Above we have used the atom substitution in the concretion operator of the framework to directly implement the object language substitution, whereas in the approach to be now considered, the latter is given an alpha-recursive characterisation.

First we define the set of names without constructors, i.e., only inhabited by atoms, and the set of terms:

$$V : \mathsf{sort} \quad \text{and} \quad \Lambda : \mathsf{sort}$$

The constructors are defined by:

$$\mathsf{Var} : (\_ : V) \rightarrow \Lambda\,; \qquad @ : (\_ : \Lambda, \_ : \Lambda) \rightarrow \Lambda\,; \qquad \lambda : (\_ : \ll x{:}V \gg \Lambda) \rightarrow \Lambda\,;$$

Induction over lambda-terms is declared as follows:

$$\begin{array}{l} \Lambda_{\mathsf{ind}} : (P : \ll t{:}\Lambda \gg \mathtt{Form}, \\ \qquad \_ : \ll x{:}V \gg \mathscr{D}(P[\mathsf{Var}(x)]), \\ \qquad \_ : \ll m{:}\Lambda \gg \ll n{:}\Lambda \gg \ll h_1{:}\mathscr{D}(P[m]) \gg \ll h_2{:}\mathscr{D}(P[n]) \gg P[@(m,n)], \\ \qquad \_ : \ll m{:}\Lambda \gg \ll h{:}\mathscr{D}(P[m]) \gg \mathscr{D}(P[\lambda(\ll z{:}V \gg m)]), \\ \qquad M : \Lambda \\ \qquad) \rightarrow \mathscr{D}(P[M])\,;\; z\#P \end{array}$$

Notice the analogy with the principle of alpha-structural induction formulated in [18]. In a similar manner, substitution can be declared as:

$$\Lambda_{\mathsf{subst}} : (\_ : \Lambda, \_ : V, \_ : \Lambda) \rightarrow \Lambda\,;$$

and axiomatised by the following equations (where $\Lambda_{\mathsf{subst}}(M,x,N)$ stands for $M[x := N]$):

$$\begin{array}{l} \Lambda_{\mathsf{subst}-\mathsf{v}} : (X : V, N : \Lambda) \rightarrow \mathscr{D}(\forall(\ll x{:}V \gg = (\Lambda_{\mathsf{subst}}(\mathsf{Var}(x), X, N), \mathsf{if}(==(x,X), N, \mathsf{Var}(x))))\,; \\ \Lambda_{\mathsf{subst}-@} : (M_1 : \Lambda, M_2 : \Lambda, X : V, N : \Lambda) \rightarrow \mathscr{D}(= (\Lambda_{\mathsf{subst}}(@(M_1,M_2), X, N), @(\Lambda_{\mathsf{subst}}(M_1,X,N), \Lambda_{\mathsf{subst}}(M_2,X,N))))\,; \\ \Lambda_{\mathsf{subst}-\lambda} : (M : \Lambda, X : V, N : \Lambda) \rightarrow \mathscr{D}(= (\Lambda_{\mathsf{subst}}(\lambda(\ll x{:}V \gg M), X, N), \lambda(\ll x{:}V \gg \Lambda_{\mathsf{subst}}(M,X,N))\,;\; x\#\{N,X\}. \end{array}$$

Here an operator $==$ is used to compare names in V, which yields a boolean in the obvious manner. The set of booleans with the operator if is introduced as usual. With these declarations, we have been able to construct a derivation for the substitution lemma

$(M[x := N])[y := P] = (M[y := P])[[x := N[y := P]]$ if $x \neq y$ and $x\#P$,

which proceeds by (alpha-structural) induction on M in very much the same way as a pencil-and-paper proof utilising Barendregt's variable convention [2].

6 Conclusions and Related Work

One of the best known examples of logical frameworks is LF [15], based on a typed λ-calculus with dependent types. Several proof assistants based on the use of Higher-Order Abstract Syntax to encode binders have been implemented (e.g., Beluga [3]). Nominal type theory as a basis for logical frameworks has been investigated independently by Cheney [6, 7] and Pitts [19] as extensions of a typed λ-calculus with names, name-abstraction and concretion operators, and name-abstraction types. Although the extension with nominal features of a dependently typed λ-calculus yields a powerful type theory, the interaction between name abstraction and functional abstraction is a source of difficulties (see [6] for a detailed discussion).

We have shown that despite its first-order character, our dependently sorted system can yield a logical framework where standard languages with binders can be defined and reasoned about. For example, we have shown how to define an induction principle for λ-terms, taking into account the α-equivalence relation. Also, the first-order character of the language, which is a consequence of the restriction for atoms to carry only data sorts, permits a simple definition of computation, actually consisting in simple syntactic definition at the meta-level. This alleviates somewhat the notions and proofs of adequacy, as compared e.g. to [15], just as happens with the lambda-free frameworks [1].

Cartmell's Generalised Algebraic Theories (GAT) [5, 20] also include dependent sorts but lack any intrinsic binding structure. To facilitate the specification of languages with binders second-order versions of GATs have been proposed [10, 21, 16], which incorporate binding and capture-avoiding substitution using free algebras with substitution structure as a model. We adopt a nominal approach: our dependently sorted system can be seen as an extension of GATs with nominal features, such as notions of fresh names and name abstraction, as well as name permutations and capture-avoiding substitutions.

The sorting system has some limitations that we will address in future work, such as the fact that in declarations for term and type constructors the variables cannot have sorts that depend on atoms, which would be useful to define recursor operators on λ-terms. Allowing for variables depending on terms will also permit to use them to represent goals to be solved in incomplete terms, as well as schematic derivations. We also assign great importance to the goal of extending the present framework with recursive definitions to be used as rules of computation, as in Martin-Löf's type theory. This would conduct us to a version of this theory fully founded upon a nominal syntax, with the issues brought about by binding solved at an infrastructure level.

Acknowledgments

We thank three anonymous referees for their comments that helped us to improve this final version. This work was partially funded by Agencia Nacional de Investigación e Innovación (ANII), of Uruguay. Miguel Pagano was partially funded by a research grant from SECyT-UNC.

References

[1] Robin Adams (2008): *Lambda-Free Logical Frameworks*. *CoRR* abs/0804.1879, doi:10.48550/arXiv.0804.1879. Available at `http://arxiv.org/abs/0804.1879`.

[2] Henk P. Barendregt (1984): *The Lambda Calculus: its Syntax and Semantics (revised ed.)*. *Studies in Logic and the Foundations of Mathematics* 103, North-Holland, doi:10.1016/B978-0-444-87508-2.50006-X.

[3] Olivier Savary Bélanger, Stefan Monnier & Brigitte Pientka (2013): *Programming Type-Safe Transformations Using Higher-Order Abstract Syntax*. In Georges Gonthier & Michael Norrish, editors: *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, *Lecture Notes in Computer Science* 8307, Springer, pp. 243–258, doi:10.1007/978-3-319-03545-1_16.

[4] Anthony Bordg, Lawrence C. Paulson & Wenda Li (2022): *Simple Type Theory is not too Simple: Grothendieck's Schemes Without Dependent Types*. *Exp. Math.* 31(2), pp. 364–382, doi:10.1080/10586458.2022.2062073.

[5] John Cartmell (1986): *Generalised algebraic theories and contextual categories*. *Ann. Pure Appl. Log.* 32, pp. 209–243, doi:10.1016/0168-0072(86)90053-9.

[6] James Cheney (2009): *A Simple Nominal Type Theory*. *Electr. Notes Theor. Comput. Sci.* 228, pp. 37–52. Available at `http://dx.doi.org/10.1016/j.entcs.2008.12.115`.

[7] James Cheney (2012): *A dependent nominal type theory*. *Logical Methods in Computer Science* 8(1), doi:10.2168/LMCS-8(1:8)2012.

[8] James Cheney & Christian Urban (2008): *Nominal logic programming*. *ACM Trans. Program. Lang. Syst.* 30(5), pp. 26:1–26:47, doi:10.1145/1387673.1387675.

[9] Elliot Fairweather, Maribel Fernández, Nora Szasz & Alvaro Tasistro (2015): *Dependent Types for Nominal Terms with Atom Substitutions*. In Thorsten Altenkirch, editor: *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland*, *LIPIcs* 38, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 180–195, doi:10.4230/LIPICS.TLCA.2015.180.

[10] Marcelo P. Fiore (2008): *Second-Order and Dependently-Sorted Abstract Syntax*. In: *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*, IEEE Computer Society, pp. 57–68, doi:10.1109/LICS.2008.38.

[11] Murdoch J. Gabbay (2011): *Nominal terms and nominal logics: from foundations to meta-mathematics*. In: *Handbook of Philosphical Logic*, 17, Kluwer, pp. 79–178, doi:10.1007/978-94-007-6600-6_2.

[12] Murdoch J. Gabbay & Aad Mathijssen (2008): *Capture-Avoiding Substitution as a Nominal Algebra*. *Formal Aspects of Computing* 20(4–5), pp. 451–479, doi:10.1007/11921240_14.

[13] Murdoch J. Gabbay & Aad Mathijssen (2008): *One-and-a-halfth-order Logic*. *Journal of Logic and Computation* 18(4), pp. 521–562, doi:10.1093/logcom/exm064.

[14] Murdoch J. Gabbay & Dominic P. Mulligan (2008): *One-and-a-halfth Order Terms: Curry-Howard for Incomplete Derivations*. In: *Proceedings of 15th Workshop on Logic, Language and Information in Computation (WoLLIC 2008)*, *Lecture Notes in Artificial Intelligence* 5110, Springer, pp. 180–194, doi:10.1007/978-3-540-69937-8_16.

[15] Robert Harper, Furio Honsell & Gordon Plotkin (1987): *A Framework for Defining Logics*. In: *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science (LICS 1987)*, IEEE Computer Society Press, New York, pp. 194–204, doi:10.1145/138027.138060.

[16] Ambrus Kaposi & Szumi Xie (2024): *Second-Order Generalised Algebraic Theories: Signatures and First-Order Semantics*. In Jakob Rehof, editor: *9th International Conference on Formal Structures for Computation and Deduction, FSCD 2024, July 10-13, 2024, Tallinn, Estonia*, *LIPIcs* 299, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 10:1–10:24, doi:10.4230/LIPICS.FSCD.2024.10.

[17] Andrew M. Pitts (2003): *Nominal Logic, A First Order Theory of Names and Binding*. *Information and Computation* 186(2), pp. 165–193, doi:10.1016/S0890-5401(03)00138-X.

[18] Andrew M. Pitts (2006): *Alpha-structural recursion and induction*. Journal of the ACM 53(3), pp. 459–506, doi:10.1145/1147954.1147961.

[19] Andrew M. Pitts, Justus Matthiesen & Jasper Derikx (2014): *A Dependent Type Theory with Abstractable Names*. In Mauricio Ayala-Rincón & Ian Mackie, editors: Ninth Workshop on Logical and Semantic Frameworks, with Applications, LSFA 2014, Brasília, Brazil, September 8-9, 2014, Electronic Notes in Theoretical Computer Science 312, Elsevier, pp. 19–50, doi:10.1016/J.ENTCS.2015.04.003.

[20] Jonathan Sterling (2019): *Algebraic Type Theory and Universe Hierarchies*. CoRR abs/1902.08848, doi:10.48550/arXiv.1902.08848.

[21] Taichi Uemura (2021): *Abstract and concrete type theories*. Ph.D. thesis, Universiteit van Amsterdam, Institute for Logic, Language and Computation. Available at `https://eprints.illc.uva.nl/id/eprint/2195`.

[22] Christian Urban, Andrew M. Pitts & Murdoch J. Gabbay (2004): *Nominal Unification*. Theoretical Computer Science 323(1–3), pp. 473–497, doi:10.1016/j.tcs.2004.06.016.