Typed Nominal Rewriting

ELLIOT FAIRWEATHER and MARIBEL FERNÁNDEZ, King's College London, Department of Informatics

Nominal terms extend first-order terms with nominal features and as such constitute a meta-language for reasoning about the named variables of an object language in the presence of meta-level variables. This paper introduces a number of type systems for nominal terms of increasing sophistication and demonstrates their application in the areas of rewriting and equational reasoning. Two simple type systems inspired by Church's simply typed lambda calculus are presented where only well-typed terms are considered to exist, over which α-equivalence is then axiomatised. The first requires atoms to be strictly annotated whilst the second explores the consequences of a more relaxed de Bruijn style approach in the presence of atom-capturing substitution. A final type system of richer ML-like polymorphic types is then given in the style of Curry, in which elements of the term language are deemed typeable or not only subsequent to the definition of alpha-equivalence. Principal types are shown to exist and an inference algorithm given to compute them. This system is then used to define two presentations of typed nominal rewriting, one more expressive and one more efficient, the latter also giving rise to a notion of typed nominal equational reasoning.

CCS Concepts: • **Theory of computation → Equational logic and rewriting**; **Type theory**; **Type structures**; • **Mathematics of computing** → *Lambda calculus*;

Additional Key Words and Phrases: Nominal Syntax, Nominal Rewriting

ACM Reference Format:
ELLIOT FAIRWEATHER and MARIBEL FERNÁNDEZ. 2017. Typed Nominal Rewriting. *ACM Trans. Comput. Logic* 1, 1 (November 2017), 45 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

This paper concerns the intersection of two areas of research, that of syntax with binding and type systems.

The concept of binders is pervasive in the study of formal languages. Nominal techniques [36, 37, 41] offer an approach to reasoning about name binding in formal languages, allowing direct access to binders and named bound variables, and use the properties of permutations of names to simplify reasoning about object-language terms equivalent up to the renaming of bound names. This 'α-equivalence' over terms can be provided as a formal, built-in axiomatisation and allows an approach to structural induction modulo α-equivalence similar to that used in informal practice [58]. The approach has already enjoyed success in both theory and practice. Nominal terms, first introduced in [69], are a nominal extension of first-order terms and as such constitute a meta-language for reasoning about named variables of an object-language in the presence of meta-level variables. They have been studied extensively, particularly in the context of rewriting and equational reasoning [22, 23, 33, 34, 39]. Object-level variables are represented in nominal terms using atoms, which can

Authors' address: ELLIOT FAIRWEATHER, Elliot.Fairweather@kcl.ac.uk; MARIBEL FERNÁNDEZ, Maribel.Fernandez@kcl.ac.uk, King's College London, Department of Informatics.

1529-3785/2017/11-ART $15.00
https://doi.org/0000001.0000001

be abstracted, and meta-level variables are represented by moderated variables (a variable with an atom-permutation suspended over it). The suspended permutation acts when the variable is instantiated by a substitution, which is first-order in nature (that is, not capture-avoiding). For example, the nominal terms lam $[a]\,X$ and app (lam $[a]\,X$, X), where the permutation suspended over X is the identity and has been omitted, represent the lambda term 'contexts' $\lambda x.M$ (or $\lambda x.[-]$) and $(\lambda x.M)\ M$, respectively. Unification and matching of nominal terms has been shown to be decidable [69] and linear time algorithms for matching have been developed [9]. As a result, both logic-based and functional programming languages that integrate nominal techniques have already been implemented [14, 16, 61, 66].

Type systems are another important area within the study of formal languages. First studied as a means to overcome the paradoxes of set theory [64], they were later incorporated into the lambda calculus by Church [1940]. In addition to extensive subsequent research from this theoretical standpoint, type systems were later applied to the area of programming languages; they are intrinsically linked to the specification and verification of program properties through constructivist logics and the Curry-Howard isomorphism. Type systems provide a tractable, syntactic method to define and check specifications of programs, allowing certain of their behaviours to be reasoned about even before execution [56, 57].

This is an area of increasing importance; critical domain software, such as that used in the fields of medicine, telecommunications, transport and defence, often demands mathematical certainty of program correctness. Thus there is a corresponding demand for the development of type systems of increasing power and sophistication for use in certified programming environments. However, thus far, formal research in type systems for nominal abstract syntax, nominal terms and nominal rewriting has been limited (see Related Work below). This paper contributes several type systems for nominal terms and systems of nominal reasoning, in which object-level variables (atoms) may inhabit any type and users may provide arbitrary type declarations for term-formers. Both Church- and Curry-style approaches are considered, the latter in the presence of polymorphic types. Church-style systems are considered in Section 3. In such systems, terms are well-typed by definition and thus the typing of terms precedes the definition of relations on terms such as freshness and alpha-equivalence. The first system which is briefly considered is in strict Church-style [18] whereby atoms are annotated with their types. The second explores de Bruijn's presentation of Church's system [8]. Whilst notationally less verbose than the fully annotated system, this approach, in which types given to abstracted atoms may be added to the typing environment, is more complex due to the possibility of typing conflicts arising from atom-capture when applying variable substitutions to non-linear terms. For example, in the untyped nominal term app (lam $[a]\,X$, X), the first occurrence of the variable X is under the scope of an abstraction for the atom a, but the second occurrence is not. Thus, a substitution $[X \mapsto a]$ applied to this term introduces both an abstracted and a non-abstracted occurrence of a. To ensure safe interaction between capturing substitutions and atom-typing contexts, the type systems designed in this paper check the type compatibility of the abstraction contexts for each occurrence of a moderated variable, taking suspended permutations into account. This is not needed in the strict Church system where each atom has a fixed type.

Two possible directions for further study then arise; one could consider polymorphic or other more powerful types in a Church setting (such as the dependent types presented in [31]), or alternatively, one could define a simple Curry-style system [8, 26], using meta-level type variables.

Here, it was decided to take a step in both directions simultaneously, and a Curry-style, ML-like polymorphic type system is presented in Section 4. Term-formers may now be assigned type schemes and only the types of unabstracted atoms are pre-determined. Elements of the term language are deemed typeable or not only after the definition of alpha-equivalence. As for the de Bruijn system, it is necessary to account for the interaction of capturing substitution but here

the Curry-style typing allows freshness constraints to be taken into consideration and the effect of permutations suspended upon variables to be considered only in the case where non-linear substitutions may occur. Alternative designs are possible, for example by requiring a fixed type for all the occurrences of an atom, as in the strict Church system. The chosen approach, however, permits a more flexible use of atoms, where different occurrences of an atom may have different types, provided a compatibility condition is satisfied. This condition ensures that a well-typed capturing substitution acting on a well-typed term produces a well-typed term. Terms are also shown to have a principal type and an inference algorithm to compute them is given.

This polymorphic system is then studied in the context of both rewriting and equational theories in Section 5. Two formulations of typed nominal rewriting are presented, one more expressive, using equivariant matching over sets of uniform rules and one more efficient, using nominal matching with closed rules, which nevertheless encompasses most scenarios of interest. In both cases, the rewriting relation preserves types. A formulation of typed nominal equational reasoning based on the latter of these approaches is then derived. A wide selection of examples is given.

The work of Section 3 was first presented in [29]. An earlier version of the material presented in Section 4 and Section 5 was published in [30] in collaboration with Murdoch J. Gabbay. It was subsequently improved upon in [29].

Related Work. The subject of syntax with binding is an extremely large area of research, in which many approaches exist. One of the first solutions was that of de Bruijn [1972] used in the development of the theorem prover AUTOMATH, in which variables are nameless and instead represented and manipulated as indices. Although excellent from a mechanical point of view, this approach is far removed from the traditional, human-friendly representation of term languages. An intermediate approach is that of 'locally nameless' systems, such as described in [3, 52]. There, de Bruijn indices are used for bound variables and names for free variables. This approach has been used in the implementation of many modern theorem provers including Coq [17, 24], LEGO [50], and Isabelle [72]. Perhaps the most popular approach in higher-order rewriting [48, 49, 51] is that of 'higher-order abstract syntax' [55]. In such systems, binding in term languages is represented using the lambda calculus and thus, issues of α-equivalence and capture-avoiding substitution are implicitly handled at the meta-level. The undecidable properties of the lambda calculus, such as the undecidability of unification [44], do cause problems here but practical solutions do exist. However, bound names are not directly accessible in such systems and inductive reasoning over expressions containing binders is not easily supported.

For nominal terms, a Curry-style type system is given in [32], later simplified in [30]. Nominal terms are given sorts in [69] where atoms inhabit base atom sorts and term-formers yield terms of base 'data' sorts. The authors are unaware of any other approaches to type systems for nominal terms directly but a number of systems of nominal abstract syntax, a more fundamental approach to the incorporation of the ideas of nominal sets to syntax, have been typed including such as [11, 13, 25, 59, 60, 65]. As with sorted nominal terms, atoms in such systems are either typed with a single type or restricted to a certain set of base types. Abstracted atoms are also subsequently required to inhabit an 'atom' type. These restrictions, whilst sometimes desirable, deny to the nominal parts of the syntax any of the more complex types or features present in the type systems. With the view of developing rewriting applications, in those systems presented here both unabstracted and abstracted atoms may inhabit any type.

The types used here are most similar to those of αProlog [15], a Prolog-like logic programming language employing nominal abstract syntax. This system also gives terms ML-style polymorphic types, built using type variables, type constructors, and abstraction types, and function applications are also given fixed declarations. However, atoms must inhabit atom types, disjoint from the types

of terms, as must the atoms of abstractions, and the result type of function applications is restricted to constructed term types. Whilst implemented, the type system is not formalised.

Urban [2008] describes how the nominal package for Isabelle [72], a higher-order logic theorem proving environment, is typed. Within the Isabelle type system, atoms are given a single type. Permutations are typed simply as lists of pairs of that type. The action of a permutation is an overloaded operator allowing a definition to be given for each built-in 'permutation type'. Such a type is an instance of a type class that guarantees that certain properties of elements of the type are maintained under the permutation operation. However, in later work [45], it is explained that atoms may be given arbitrary sorts ('multi-sorted concrete atoms') at the term level which can be enforced by the Isabelle type-checker by use of sub-typing.

Shinwell [2005] describes the implementation of the language Fresh OCaml, (successor to the FreshML of [67], where atom-abstractions are implemented by dynamically generating fresh names), an extension of OCaml for meta-programming with binding and data structures. Expressions for generating fresh atoms, atom abstraction, swapping atoms and freshness conditions are added to the language. Atoms are given a name type but one that is polymorphic over all types. Abstractions need not necessarily abstract atoms and the abstracted expression need not be of a 'name' type. However this language is not fully formalised. Instead a smaller subset, Mini-FreshML, is studied in which the type system is extended with a single type for names and an abstraction type, of the type of names over types. Pottier [2007] later provided a mechanism for static name control in FreshML.

Type systems for first-order rewriting are studied in [70, 71]. The first-order rewriting language Maude [19, 20] has a rich type system, providing sorts, sub-sorts, operator overloading and more. Type systems for formalisms of higher-order rewriting are less well developed. Higher-order Rewrite Systems [51], use a typed lambda calculus as a meta-language together with restrictions on patterns to ensure the decidability of higher-order unification, however, rewrite rules may only be typed with base types. Richer types for function symbols in this system are considered in [47]. In 'plain' higher-order rewrite systems, the lambda calculus is extended with first-order rewriting together with first-order matching. A mixed approach of standard and plain higher-order rewriting is proposed in [46], using a system of polymorphic types.

Overview. The paper is organised as follows: Section 2 gives an overview of the nominal approach to syntax with binding. Section 3 defines two approaches to Church-style simple types for nominal terms and Section 4 a more sophisticated polymorphic type system in the style of Curry. This system is then used in Section 5 to develop typed nominal equational systems. Section 6 concludes and gives directions for future work.

2 BACKGROUND

Nominal terms [69] are a meta-language designed to specify and reason about formal languages that use names and binding, such as logics or programming languages. Equational reasoning for nominal terms has been extensively studied [22, 23, 33, 34, 39].

2.1 Atoms, Permutations and Terms

Atoms represent names or named variables at the object-language level.

Let there be a countably infinite set of atoms, $\mathbb{A}$, and following the **permutative convention** of [38], let $a,\ b,\ c,\ \ldots$, range over *distinct* atoms in $\mathbb{A}$.

A **permutation**, denoted π, is a bijective function on $\mathbb{A}$, such that $\{a \mid \pi(a) \neq a,\ a \in \mathbb{A}\}$ is a finite set. Call this set the **support** of π and write it as $\mathrm{supp}(\pi)$. Permutations are represented as a list of transpositions or 'swappings' of atoms, where id represents the identity permutation, that is,

the permutation, π, which, for all $a \in \mathbb{A}$, $\pi(a) = a$.

$$\pi ::= \mathsf{id} \mid \pi * (a\ b)$$

Note that permutations are read from right to left and that the final id and 'list cons' operators in the representation of a permutation, π, will most often be omitted; for example, the permutation $(\mathsf{id} * (c\ d)) * (a\ b)$ will be written simply as $(c\ d)\ (a\ b)$.

Below, π ranges over such syntactic representations of permutations as finite lists of pairs of atoms. Due to redundancy in this presentation, a permutation need not have a unique representation; for example, $(a\ b)(b\ c)(a\ b)$ represents the same permutation as $(a\ c)$, and $(a\ a)$, the same as the identity permutation.

Write $\pi'\ @\ \pi$ for the composition of the permutation π followed by the permutation π', which, as a list of transpositions corresponds to the concatenation of π' onto the end of π. Write π^{-1} for the **inverse** of π, which is obtained by reversing the order of the list of transpositions used to represent π; for example $\big((e\ f)(c\ d)(a\ b)\big)^{-1} = (a\ b)(c\ d)(e\ f)$.

The **disagreement set** of two permutations, π and π', written $\mathsf{ds}(\pi,\ \pi')$, is defined as $\{a \mid \pi(a) \neq \pi'(a)\}$; that is, the set of atoms for which π and π' have differing values.

Consider countably infinite, pairwise disjoint sets of **variables**, $X,\ Y,\ Z, \ldots \in \mathbb{X}$ and **term-formers**, $\mathsf{f},\ \mathsf{g},\ \ldots \in \mathbb{F}$, and then following [69], let **nominal terms** be defined using the grammar described in Definition 2.1 immediately below.

Definition 2.1 (Nominal Terms).

$$s,\ t ::= a \mid [a]\,t \mid \mathsf{f}\,t \mid (t_1,\ \ldots,\ t_n) \mid \pi \cdot X$$

Let these constructions be called respectively **atom terms**, **abstractions**, **function applications**, **tuples** and **moderated variables**. The symbol $\equiv$ is used to denote syntactic equality over terms.

Permutations are said to **suspend** on variables to form moderated variables; read $(a\ b) \cdot X$ as 'swap a and b in X'. Whilst a variable, X, is not by itself a nominal term, the moderated variable $\mathsf{id} \cdot X$ is a term; in this situation the identity permutation will most often be omitted and the term be written as just X.

Write $\mathsf{vars}(t)$ for the variables that occur syntactically in t and $\mathsf{atms}(t)$ for the atoms syntactically occurring in t. The definitions are inductive and omitted. Note that in the case of atoms this includes all those mentioned in permutations suspended upon moderated variables. For example, $a,\ b,\ c \in \mathsf{atms}([a]\,[b]\,(a\ c) \cdot X)$ and $X \in \mathsf{vars}([a]\,[b]\,(a\ c) \cdot X)$. These notations are extended to other elements of syntax and tuples of elements in the obvious way.

Say an occurrence of an atom in a term t is **concrete** if it occurs in t directly as an atom subterm. Call concrete occurrences of an atom, a, **abstracted** when they are in the scope of an abstraction on a, $[a]\,t$, and otherwise call them **unabstracted**.

If one supposes term-formers lam and app, then one can build nominal terms that represent terms of the lambda calculus [6].

Example 2.2 (Nominal Terms).

$$\begin{array}{ll} \mathsf{lam}\,[a]\,a & \mathsf{app}\,(\mathsf{lam}\,[a]\,a,\ a) \\ \mathsf{lam}\,[a]\,X & \mathsf{app}\,(\mathsf{lam}\,[a]\,X,\ X) \end{array}$$

In this example the nominal terms in the first line represent, respectively, the lambda terms $\lambda x.x$ and $(\lambda x.x)\ x$, and those in the second line, the lambda term 'contexts' $\lambda x.M$ (or $\lambda x.[-]$) and $(\lambda x.M)\ M$.

The second term in the second line contains two occurrences of the variable X; these represent the *same* unknown part of the term, and each occurs under different abstracted atoms. This is a generalisation of the traditional notion of a lambda term 'context', in which usually only a single unknown part of a term is allowed.

The term app (lam $[a]\, a,\ a$) has three occurrences of the atom a, the rightmost two of which are concrete occurrences, the first abstracted and the second unabstracted. However, whilst the atoms a and b are both in the term lam $[a]\, (a\ b) \cdot X$, none of the occurrences is concrete.

2.2 Permutation Actions

The application of a permutation, π, to an atom, a, is written as a standard function application, $\pi(a)$, and is defined inductively over a list of transpositions as described in Definition 2.3.

Definition 2.3 (Permutation Action on Atoms).

$$\mathsf{id}(a) \triangleq a \qquad (\pi * (a\ b))(a) \triangleq \pi(b)$$
$$(\pi * (a\ b))(b) \triangleq \pi(a) \qquad (\pi * (a\ b))(c) \triangleq \pi(c)$$

Permutations are most commonly applied to terms at the level of the object-language. The **permutation action** of a permutation, π, on a term, t, is written $\pi \cdot t$ and defined inductively, as follows below in Definition 2.4. Note that this is the same notation as that used to represent a permutation suspended upon a variable in a moderated variable; the two ideas are conceptually similar and no ambiguity arises.

Definition 2.4 (Permutation Action on Terms).

$$\pi \cdot a \triangleq \pi(a) \qquad \pi \cdot (\pi' \cdot X) \triangleq (\pi\ @\ \pi') \cdot X$$
$$\pi \cdot [a]\, t \triangleq [\pi(a)]\, \pi \cdot t \qquad \pi \cdot \mathsf{f}\, t \triangleq \mathsf{f}\, (\pi \cdot t)$$
$$\pi \cdot (t_1,\ \ldots,\ t_n) \triangleq (\pi \cdot t_1,\ \ldots,\ \pi \cdot t_n)$$

Example 2.5 (Permutation Action on Terms).

$$(a\ b) \cdot \mathsf{f}\, [a]\, (a,\ b,\ (b\ c) \cdot X) = \mathsf{f}\, [b]\, (b,\ a,\ (a\ b)(b\ c) \cdot X)$$

At times, it is necessary to apply a permutation at the level of nominal terms rather than of an object language; this is called the meta-level permutation action on terms.

Define ${}^{\pi}t$, the **meta-level permutation action** of a permutation, π, on a term, t, inductively, as follows.

Definition 2.6 (Meta-level Permutation Action on Terms).

$${}^{\pi}a \triangleq \pi(a) \qquad {}^{\pi}(\pi' \cdot X) \triangleq (\pi\ @\ \pi'\ @\ \pi^{-1}) \cdot X$$
$${}^{\pi}([a]\, t) \triangleq [\pi(a)]\, {}^{\pi}t \qquad {}^{\pi}(\mathsf{f}\, t) \triangleq \mathsf{f}\, ({}^{\pi}t)$$
$${}^{\pi}(t_1,\ \ldots,\ t_n) \triangleq ({}^{\pi}t_1,\ \ldots,\ {}^{\pi}t_n)$$

Example 2.7 (Meta-level Permutation Action on Terms).

$${}^{(a\ b)}\mathsf{f}\, [a]\, (a,\ b,\ (b\ c) \cdot X) = \mathsf{f}\, [b]\, (b,\ a,\ (a\ c) \cdot X)$$

The permutation $(a\ c)$ in the right-hand side is the same as $(a\ b)(b\ c)(a\ b)$ (as mentioned earlier, permutations do not have a unique syntactical representation).

2.3 Variable Substitutions

Variable substitutions are substitutions that operate at the level of nominal terms.

Definition 2.8 (Variable Substitutions). A **variable substitution** is a mapping from variables to terms, equal to the identity mapping but for finitely many arguments, and written as a set of bindings $[X_1 \mapsto s_1] \ldots [X_n \mapsto s_n]$, such that the variables, $X_1, \ldots, X_n$, are pairwise distinct.

Let θ range over variable substitutions and write the value of a substitution, θ, for a variable, X, as $\theta(X)$. The **domain** of a substitution θ, written $\mathrm{dom}(\theta)$, is the set of variables, $\{X \mid \theta(X) \not\equiv X\}$. id denotes the identity substitution, that is, the substitution whose domain is the empty set.

The action of a variable substitution, θ, upon a term, t, is written as juxtaposition to the right thus, $t\,\theta$, and is defined inductively as follows.

Definition 2.9 (Variable Substitution Action).

$$a\,\theta \triangleq a \qquad (\pi \cdot X)\,\theta \triangleq \pi \cdot (\theta(X))$$
$$([a]\,t)\,\theta \triangleq [a]\,(t\,\theta) \qquad (\mathsf{f}\,t)\,\theta \triangleq \mathsf{f}\,(t\,\theta)$$
$$(t_1, \ldots, t_n)\,\theta \triangleq (t_1\,\theta, \ldots, t_n\,\theta)$$

It is important to note that variable substitutions are grafting onto syntax trees and do *not* avoid the capture of atoms by abstractions.

Example 2.10 (Atom Capture by Variable Substitution).

$$([a]\,X,\ [b]\,X)\,[X \mapsto a] = ([a]\,a,\ [b]\,a)$$

Notice how in the above example, the atom, a, is captured in the first abstraction following the action of the variable substitution.

2.4 Freshness and Alpha-equivalence

The freshness and alpha-equivalence relations now presented are key concepts with regard to nominal terms.

Call $a \# t$ a **freshness constraint** (read 'a fresh for t'). Let $\Delta,\ \nabla$ range over sets of freshness constraints of the form $a \# X$; call such sets **freshness contexts**. The shorthand notation, $\mathcal{A} \# \mathcal{X}$, where $\mathcal{A}$ and $\mathcal{X}$ are arbitrary sets of atoms and variables respectively, may be used to represent the freshness context $\{a \# X \mid a \in \mathcal{A},\ X \in \mathcal{X}\}$. The brackets of singleton sets may be omitted in such notation. The meta-level permutation action is extended to freshness contexts, thus ${}^{\pi}\Delta \triangleq \{\pi(a) \# X \mid a \# X \in \Delta\}$.

Write $\Delta \vdash a \# t$ when a derivation exists using the rules given in Definition 2.11 below.

Definition 2.11 (Freshness Relation).

$$\frac{}{\Delta \vdash a \# b}\,(atm)^{\#} \qquad \frac{\pi^{-1}(a) \# X \in \Delta}{\Delta \vdash a \# \pi \cdot X}\,(var)^{\#}$$

$$\frac{}{\Delta \vdash a \# [a]\,t}\,(abs:aa)^{\#} \qquad \frac{\Delta \vdash a \# t}{\Delta \vdash a \# [b]\,t}\,(abs:ab)^{\#}$$

$$\frac{\Delta \vdash a \# t_1 \ \ldots\ \Delta \vdash a \# t_n}{\Delta \vdash a \# (t_1, \ldots, t_n)}\,(tpl)^{\#} \qquad \frac{\Delta \vdash a \# t}{\Delta \vdash a \# \mathsf{f}\,t}\,(app)^{\#}$$

Call $s \approx_{\alpha} t$ an **alpha-equality constraint** and write $\Delta \vdash s \approx_{\alpha} t$ when a derivation exists using the rules given in Definition 2.12 below.

Definition 2.12 (Alpha-equivalence Relation).

$$\frac{}{\Delta \vdash a \approx_\alpha a}\,(atm)^\alpha \qquad \frac{\forall a \in \mathsf{ds}(\pi,\ \pi') \quad a \# X \in \Delta}{\Delta \vdash \pi \cdot X \approx_\alpha \pi' \cdot X}\,(var)^\alpha$$

$$\frac{\Delta \vdash s \approx_\alpha t}{\Delta \vdash [a]\, s \approx_\alpha [a]\, t}\,(abs:aa)^\alpha \qquad \frac{\Delta \vdash s \approx_\alpha (a\ b) \cdot t \quad \Delta \vdash a \# t}{\Delta \vdash [a]\, s \approx_\alpha [b]\, t}\,(abs:ab)^\alpha$$

$$\frac{\Delta \vdash s_1 \approx_\alpha t_1\ \ldots\ \Delta \vdash s_n \approx_\alpha t_n}{\Delta \vdash (s_1,\ \ldots,\ s_n) \approx_\alpha (t_1,\ \ldots,\ t_n)}\,(tpl)^\alpha \qquad \frac{\Delta \vdash s \approx_\alpha t}{\Delta \vdash \mathsf{f}\, s \approx_\alpha \mathsf{f}\, t}\,(app)^\alpha$$

Example 2.13 (Alpha-equivalence).

$$\varnothing \vdash [a]\, a \approx_\alpha [b]\, b \qquad \varnothing \vdash [a]\, c \approx_\alpha [b]\, c \qquad \varnothing \nvdash [a]\, c \approx_\alpha [a]\, a$$
$$a \# X,\ b \# X \vdash (a\ b) \cdot X \approx_\alpha X$$

Let C range over freshness and alpha-equality constraints; a **constraint problem**, C, is an arbitrary set of such constraints. Extend the above notation for the derivability of constraints element-wise to constraint problems; thus, write $\Delta \vdash \{C_1,\ \ldots,\ C_n\}$ for $\Delta \vdash C_1, \ldots, \Delta \vdash C_n$. The action of variable substitutions extends to constraints and thus constraint problems in the natural way.

If there exists a freshness context Δ such that $\Delta \vdash C$, then C **is valid** in Δ. See [33, 69] for algorithms to validate constraint problems.

The freshness and alpha-equivalence relations are closed under the action of permutations. That is, for any permutation, π, if $\Delta \vdash a \# t$, then $\Delta \vdash \pi(a) \# \pi \cdot t$, and if $\Delta \vdash s \approx_\alpha t$, then $\Delta \vdash \pi \cdot s \approx_\alpha \pi \cdot t$. The following result is one of the main technical correctness properties of nominal terms (see [69] for a proof).

Theorem 2.14. *If $\Delta \vdash a \# s$ and $\Delta \vdash s \approx_\alpha t$ then $\Delta \vdash a \# t$.*

2.5 Nominal Matching

Definition 2.15 (Nominal Matching Problem). Given a constraint problem, C, of the form $\{\ldots,\ s_i \approx_\alpha t_i,\ \ldots\}$, a corresponding nominal matching problem $\{\ldots,\ s_i \stackrel{?}{\approx}_\alpha t_i,\ \ldots\}$, is defined when $\bigcup \mathsf{vars}(s_i) \cap \bigcup \mathsf{vars}(t_i) = \varnothing$.

A solution to such a problem, if one exists, is a pair, $(\Delta,\ \theta)$, of a freshness context, Δ, and a variable substitution, θ, such that $\mathsf{dom}(\theta) \subseteq \bigcup \mathsf{vars}(s_i)$ and $\Delta \vdash C\,\theta$.

Informally, this says that a nominal matching problem is a constraint problem in which one adds the restriction that the variables in the left-hand sides of alpha-equality constraints are disjoint from the variables in the right-hand sides and that only variables in the left-hand sides of equality constraints may be instantiated.

A solvable matching problem has a unique, minimal solution; nominal matching is decidable [69] and can be solved in linear time if right-hand sides are ground [9].

Definition 2.16 (Terms-in-context). A **term-in-context** is a pair $\Delta \vdash s$ of a freshness context, Δ, and a term, s.

Definition 2.17 (Nominal Pattern Matching Problem). A **nominal pattern matching problem** consists of two terms-in-context, $\nabla \vdash l$ and $\Delta \vdash s$, to be matched, where $\mathsf{vars}(\nabla \vdash l) \cap \mathsf{vars}(\Delta \vdash s) = \varnothing$ and is written: $(\nabla \vdash l) \stackrel{?}{\approx}_\alpha (\Delta \vdash s)$.

A solution to a nominal pattern matching problem $(\nabla \vdash l) \stackrel{?}{\approx}_\alpha (\Delta \vdash s)$, if one exists, is a variable substitution, θ, such that (∇', θ) is a solution to the nominal matching problem $\nabla \cup \{l \stackrel{?}{\approx}_\alpha s\}$ and $\Delta \vdash \nabla'$. The set of solutions is denoted $\mathrm{sol}((\nabla \vdash l) \stackrel{?}{\approx}_\alpha (\Delta \vdash s))$.

Example 2.18 (Nominal Pattern Matching Problems).

$$(a \# X \vdash [a]\, X) \stackrel{?}{\approx}_\alpha (b \# Y \vdash [a]\, (a\ b) \cdot Y)$$
$$(a \# X \vdash [a]\, X) \stackrel{?}{\approx}_\alpha (\varnothing \vdash [a]\, a)$$

In the example above, the first pattern matching problem has the solution $[X \mapsto (a\ b) \cdot Y]$, whereas the second has none.

2.6 Nominal Rewriting

Definition 2.19. A nominal **rewrite rule** is a 3-tuple, consisting of a freshness context, ∇, and two terms, l and r, such that $\mathrm{vars}(\nabla) \cup \mathrm{vars}(r) \subseteq \mathrm{vars}(l)$, and is written, $\nabla \vdash l \to r$.

A nominal term rewriting system or **rewrite theory**, $\mathcal{R}$, is a pair of a set of term-formers and a possibly infinite set of rewrite rules generated from those term-formers.

Expanding upon Example 2.2, consider again the term-formers, lam and app, together with the term-former, sub, where $\mathsf{sub}\,([a]\, M,\ N)$ represents the substitution $M\,[N/a]$ in the lambda calculus. One then has the following rewrite rules defining the β-reduction of lambda terms.

Example 2.20 (Rewrite Theory).

$$\begin{aligned}
\varnothing \vdash \mathsf{app}\,(\mathsf{lam}\,[a]\, X,\ Y) &\to \mathsf{sub}\,([a]\, X,\ Y) \\
a \# X \vdash \mathsf{sub}\,([a]\, X,\ Z) &\to X \\
\varnothing \vdash \mathsf{sub}\,([a]\, a,\ Z) &\to Z \\
\varnothing \vdash \mathsf{sub}\,([a]\,(\mathsf{app}\,(X,\ Y)),\ Z) &\to \mathsf{app}\,(\mathsf{sub}\,([a]\, X,\ Z),\ \mathsf{sub}\,([a]\, Y,\ Z)) \\
b \# Z \vdash \mathsf{sub}\,([a]\,\mathsf{lam}\,[b]\, X,\ Z) &\to \mathsf{lam}\,[b]\,\mathsf{sub}\,([a]\, X,\ Z)
\end{aligned}$$

A rewrite rule, $\nabla \vdash l \to r$, matches a term-in-context, $\Delta \vdash s$, when there exists a solution to the nominal pattern matching problem $(\nabla \vdash l) \stackrel{?}{\approx}_\alpha (\Delta \vdash s)$.

Say that a term, t, has a **position** at the variable X, when t contains only one occurrence of X and the identity permutation, id, is suspended on this one occurrence. Write $t\,[s]$ for $t\,[X \mapsto s]$. One would usually call such a term, t, a 'context' but this terminology is not used below, because it has already been employed for a different purpose, however the standard notation to represent a position, $C\,[s]$, is used. A position could also be defined by a path in the abstract syntax tree of a term as in [35].

Example 2.21 (Positions). The terms X and $[a]\, X$ both have positions at X, whereas $(X,\ X)$ and $(a\ b) \cdot X$ do not.

Definition 2.22 (Nominal Rewriting). The one-step nominal rewriting relation generated by a rewrite theory, $\mathcal{R}$, is the least relation of tuples, $\Delta \vdash s \to_{\mathcal{R}} t$, such that for any rewrite rule $\nabla \vdash l \to r$ belonging to $\mathcal{R}$, freshness context Δ, terms C and s', permutation π and variable substitution θ:

$$\frac{s \equiv C\,[s'] \qquad \Delta \vdash \nabla\,\theta \cup \left\{\pi \cdot (l\,\theta) \approx_\alpha s',\quad C\,[\pi \cdot (r\,\theta)] \approx_\alpha t\right\}}{\Delta \vdash s \to_{\mathcal{R}} t}$$

The **nominal rewriting** relation over a rewrite theory $\mathcal{R}$, written $\Delta \vdash s \twoheadrightarrow_{\mathcal{R}} t$, is the reflexive, transitive closure of the one-step rewriting relation.

The nominal rewriting relation is **equivariant** at the level of nominal terms. The permutation, π, in Definition 2.22, makes it equivalent to the approach in [33] where the set of rules in a rewrite theory is made closed under the action of meta-level permutation. For example, a rewrite rule, $\varnothing \vdash a \rightarrow a$, will rewrite any atom. Equivariant nominal matching is NP-complete [10], and thus the notion of closed rewriting (see Definition 2.30) is often preferred, where possible (see [34] for more details).

Example 2.23 (Nominal Rewriting). Using the rewrite theory of Example 2.20, one has the following rewrite sequences.

$$\mathsf{app}\,(\mathsf{lam}\,[a]\,a,\ \mathsf{lam}\,[b]\,b) \rightarrow \mathsf{sub}\,([a]\,a,\ \mathsf{lam}\,[b]\,b) \rightarrow \mathsf{lam}\,[b]\,b$$
$$\mathsf{app}\,(\mathsf{lam}\,[a]\,b,\ \mathsf{lam}\,[b]\,b) \rightarrow \mathsf{sub}\,([a]\,b,\ \mathsf{lam}\,[b]\,b) \rightarrow b$$
$$\mathsf{app}\,(\mathsf{lam}\,[a]\,\mathsf{lam}\,[b]\,a,\ b) \rightarrow \mathsf{sub}\,([a]\,\mathsf{lam}\,[b]\,a,\ b) \rightarrow \mathsf{lam}\,[b']\,\mathsf{sub}\,([a]\,a,\ b) \rightarrow \mathsf{lam}\,[b']\,b$$

The class of uniform rewrite rules inherits the properties of first-order rewriting [33]. Intuitively, uniform rules preserve the set of fresh atoms of rewritten terms.

Definition 2.24 (Uniform Rewrite Rule). A nominal rewrite rule, $\nabla \vdash l \rightarrow r$, is **uniform** if $\Delta \vdash \nabla,\ a \# l$ implies $\Delta \vdash a \# r$ for any atom a and freshness context Δ.

As shown in [33], it is sufficient to check this condition for each atom mentioned in the rule and one fresh atom.

2.7 Nominal Equational Reasoning

Nominal equational reasoning is to be understood in the sense of a nominal approach to universal algebra, or the study of the logic of equality, as in [39] or [22]. The definitions in this section are taken from [34], where a detailed treatment of the connection between nominal rewriting and nominal equational reasoning can be found.

Definition 2.25. A nominal **axiom** is a 3-tuple, consisting of a freshness context, Δ, and two terms, s and t, and written, $\Delta \vdash s = t$.

An **equational theory**, $\mathcal{E}$, is a pair of a set of term-formers and a possibly infinite set of axioms generated from those term-formers.

Definition 2.26 (Nominal Equational Reasoning). The nominal equality relation generated by an equational theory, $\mathcal{E}$, is the least reflexive, symmetric, transitive relation of tuples, $\Delta \vdash s =_{\mathcal{E}} t$, such that for any axiom $\nabla \vdash l = r$ belonging to $\mathcal{E}$, freshness context Δ, term C, permutation π, variable substitution θ, and freshness context $\Delta^{\#}$ (where, if $a \# X \in \Delta^{\#}$, then $a \notin \mathsf{atms}(\Delta) \cup \mathsf{atms}(s) \cup \mathsf{atms}(t)$):

$$\frac{\Delta \cup \Delta^{\#} \vdash \nabla\,\theta \cup \left\{C\,[\pi \cdot (l\,\theta)] \approx_{\alpha} s,\quad C\,[\pi \cdot (r\,\theta)] \approx_{\alpha} t\right\}}{\Delta \vdash s =_{\mathcal{E}} t}$$

2.8 Newly-freshened Variants and Closed Terms

A **newly-freshened variant** of a term, t, is a term, written t^{ν}, in which all the atoms and variables have been replaced by newly generated atoms and variables with respect to those occurring in t; that is, such that, $\mathsf{atms}(t^{\nu}) \cap \mathsf{atms}(t) = \varnothing$ and $\mathsf{vars}(t^{\nu}) \cap \mathsf{vars}(t) = \varnothing$. An inductive definition is trivial and thus omitted.

The notion of newly-freshened variants is extended to other syntax, such as freshness contexts, rewrite rules and axioms, in the obvious way.

Atoms and variables in a newly-freshened variant may, on occasion, need to be disjoint not only from those atoms and variables occurring in one particular element of syntax itself but also

from those occurring in other elements; in such situations, these extra constraints will be explicitly specified.

Note that in practice the creation of newly-freshened variants is often accomplished by the generation of new atoms and variables with respect to the entire system under consideration.

Example 2.27 (Newly-freshened Variants).

$$([a]\,[b]\,X)^\nu = [a^\nu]\,[b^\nu]\,X^\nu$$
$$(\{a \# X,\ b \# Z\})^\nu = \{a^\nu \# X^\nu,\ b^\nu \# Z^\nu\}$$
$$(c \# Y \vdash [a]\,X \to b)^\nu = c^\nu \# Y^\nu \vdash [a^\nu]\,X^\nu \to b^\nu$$

Intuitively, a closed term has no unabstracted atoms and all occurrences of a variable must appear under the *same* abstracted atoms. Thus, $\varnothing \vdash \mathsf{f}\,([a]\,X,\ X)$ is not closed because the atom a is abstracted in the first occurrence of the variable X and not in the second, but $a \# X \vdash \mathsf{f}\,([a]\,X,\ X)$ is closed, because the freshness context ensures that a cannot occur in X. Closedness was introduced in [35]. Definition 2.28 tests for closedness using newly-freshened variants and nominal matching (see Definition 2.15) but inductive definitions are also possible [21, 38].

Definition 2.28 (Term Closedness Problems). Given a term t, if the nominal matching problem $\left\{t^\nu \stackrel{?}{\approx}_\alpha t\right\}$ has a solution, $(\nabla,\ \theta)$, then $\nabla \setminus (\mathsf{atms}(t^\nu) \# \mathsf{vars}(t))$ is a solution to the term closedness problem for t.

If a term closedness problem has a solution, then there exists a unique, minimal one; this follows from the properties of solutions to nominal matching problems. It is also independent of the newly-freshened variant generated (any freshness constraint mentioning the atoms of t^ν having been removed).

Definition 2.29 (Closed Terms, Rules and Equations). A term t is **closed** if the empty set is a solution to the term closedness problem for t.

A term-in-context, $\Delta \vdash t$, is **closed**, if ∇ is the solution to the term closedness problem for t and $\Delta \vdash \nabla$.

Call a rewrite rule, $\Delta \vdash l \to r$, or an axiom, $\Delta \vdash l = r$, **closed**, if the term-in-context $\Delta \vdash (l,\ r)$ is closed.

2.9 Closed Nominal Rewriting

As mentioned in Section 2.6, for reasons of efficiency, if possible, one should make use of closed nominal rewriting.

Definition 2.30 (Closed Nominal Rewriting). The one-step, closed nominal rewriting relation, generated by a rewrite theory, $\mathcal{R}$, is the least relation of tuples, $\Delta \vdash s \to^C_{\mathcal{R}} t$, such that for every rewrite rule $R \in \mathcal{R}$ of the form $\nabla \vdash l \to r$, term-in-context $\Delta \vdash s$, terms C and s', variable substitution θ, and a newly-freshened variant R^ν of R (fresh for Δ, s and t, in addition to R), where $\Delta^\# = \mathsf{atms}(R^\nu) \# \mathsf{vars}(\Delta) \cup \mathsf{vars}(s) \cup \mathsf{vars}(t)$:

$$\frac{s \equiv C\,[s'] \qquad \Delta \cup \Delta^\# \vdash \nabla^\nu\,\theta \cup \left\{l^\nu\,\theta \approx_\alpha s',\ C\,[r^\nu\,\theta] \approx_\alpha t\right\}}{\Delta \vdash s \to^C_{\mathcal{R}} t}$$

The **closed nominal rewriting** relation, $\Delta \vdash s \twoheadrightarrow^C_{\mathcal{R}} t$ is the reflexive, transitive closure of the one-step, closed nominal rewriting relation.

Closed rewriting is sound and complete for nominal algebra, if all axioms are closed [34].

The notion of closed nominal rewriting is valuable for two reasons. Firstly, consider a rewrite rule, $a \# X \vdash \mathsf{f}\,X \to X$. The term $\mathsf{f}\,Z$ will not rewrite using this rule under nominal rewriting,

because no freshness information for Z is available. Although, intuitively, the rule says only that there exists some atom that is fresh for Z, which is always true. However, $\mathsf{f}\ Z$ will rewrite using closed nominal rewriting, because when matched against a newly-freshened variant of the rule, $a^{\nu} \# X^{\nu} \vdash \mathsf{f}\ X^{\nu} \to X^{\nu}$, the extra freshness constraint, $a^{\nu} \# Z$, may be used.

Secondly, because the use of a newly-freshened rule makes matching against unabstracted atoms impossible, closed nominal rewriting is used with sets of closed rewrite rules, that is, rules that do not contain unabstracted atoms. This means that closed nominal rewriting can use standard nominal matching to match rules to terms; meta-level equivariance need not be considered. This reduces the complexity of the matching problem from exponential time to linear time. Other classes of terms for which equivariant matching is polynomial are described in [12].

3 SIMPLE TYPES FOR NOMINAL TERMS

This section presents two simple type systems for nominal terms, taking inspiration from the simply typed lambda calculus [7]. The systems considered are both Church-style in the sense that atoms are typed ab initio, and the sets of well-typed permutations and well-typed terms are defined before considering the actions of permutations, and the freshness and alpha-equivalence relations. Firstly, a true Church-style system [18], with an annotated syntax for typed atoms, is considered in Section 3.2. The notational burden of Church-style systems is usually alleviated in type systems for the lambda calculus by introducing a typing environment for variables and only annotating variables at the point of binding, a style attributed to de Bruijn [8]. Using typing environments is more difficult when typing nominal terms, because of the presence of variables, which represent unknown parts of terms, and the fact that variable substitutions may capture atoms. Subsection 3.3 considers the consequences of a de Bruijn-style approach for nominal terms.

3.1 Simple Types

This subsection outlines the syntax of simple types. The notion of types built using constructors and that of products of types is standard. This is augmented with a type for the abstraction construction of nominal terms.

3.1.1 Types and Declarations. Consider a countably infinite set of **type-formers**, $\mathbb{C}$, and write a typical element thus, C. For example, `Nat` or `List` might be type-formers.

Define simple **types**, $\sigma,\ \tau$, and type **declarations**, $\rho,\ \xi$, using the grammar described in Definition 3.1.

Definition 3.1 (Types and Declarations).

$$\sigma,\ \tau ::= [\sigma]\,\tau \mid \mathsf{C}\,\tau \mid (\tau_1 \times \ldots \times \tau_n) \qquad \rho,\ \xi ::= \langle \sigma \hookrightarrow \tau \rangle$$

Call $[\sigma]\,\tau$ an **abstraction type**, $\mathsf{C}\,\tau$ a **constructed type** and $(\tau_1 \times \ldots \times \tau_n)$ a **product type**. Base data types may be represented by constructed types built from an empty product type; for example, `Nat` $()$. In such cases, the product type will, more often than not, be omitted; for example, the type `Nat` $()$ will be written simply as `Nat`. Binary type formers are some times written infix, as in `Nat` $\Rightarrow$ `Nat`, where $\Rightarrow$ is a user-defined type-former for function types. Declarations are used below to type term-formers. Let $\sigma \equiv \tau$ denote syntactic equality over types and extend this to type declarations in the obvious fashion.

3.1.2 Environments. A **type association** is a pair of a variable, X, and a type, σ, written $(X\colon \sigma)$. A typing **environment**, Γ, is a set of type associations subject to the condition that if $(X\colon \sigma) \in \Gamma$ and $(X\colon \tau) \in \Gamma$ then $\sigma \equiv \tau$. Let Γ_X be the type **associated** with the variable, X, in the environment, Γ. If there is no association for X in Γ, Γ_X is undefined, written $\bot$. Write $\mathrm{vars}(\Gamma)$ for the set of variables $\{X \mid \Gamma_X \neq \bot\}$.

The equality relation on types is extended to environments in the obvious way. Bracketing in environments may often be omitted for the sake of clarity.

3.1.3 Signatures. A **declaration association** is a pair of a term-former, f, and a declaration, ρ, written $(\mathsf{f}\colon \rho)$; for example, $(0\colon \langle () \hookrightarrow \mathtt{Nat}\rangle)$ and $(\mathsf{suc}\colon \langle \mathtt{Nat} \hookrightarrow \mathtt{Nat}\rangle)$ might be declaration associations. A **signature**, Σ, is a set of declaration associations, again subject to the condition that, if $(\mathsf{f}\colon \rho) \in \Sigma$ and $(\mathsf{f}\colon \xi) \in \Sigma$ then $\rho \equiv \xi$. Σ_{f} denotes the declaration associated with the term-former f in Σ. If no associated declaration for f exists in Σ, then Σ_{f} is undefined, written $\bot$.

3.2 Type System in the Style of Church

Following Church [1940], and in the style of nominal systems using sorted atoms such as [69], this type system considers *typed atoms*, such that for each type there exists a countably infinite set of atoms. This is achieved by extending the traditional syntax of nominal terms by annotating atoms with their type.

3.2.1 Well-typed Terms. Whilst atoms are annotated by types in this system, variables and term-formers are not and so, firstly a grammar for pseudo-terms is described in Definition 3.2 and then the set of well-typed terms is defined in Definition 3.3. Permutations are well-typed by construction.

Definition 3.2 (Permutations and Pseudo-terms).

$$\pi ::= \mathsf{id} \mid \pi * (a_\tau\ b_\tau) \qquad s,\ t ::= a_\sigma \mid [a_\sigma]\, t \mid \mathsf{f}\, t \mid (t_1,\ \ldots,\ t_n) \mid \pi \cdot X$$

Note that this syntax implies that the sets of atoms for each type are disjoint. Thus, if $\sigma \not\equiv \tau$, then $a_\sigma \not\equiv a_\tau$; a_σ and a_τ are *different* atoms. Superficially, this appears to destroy the 'atomic' nature of atoms, namely that atoms are distinguishable by name alone and their internal structure is opaque. However it can easily be seen to be compatible with the notion of typed atoms if for each atom one concatenates a textual representation of its type to its name.

The requirement that each atom in a swapping within a permutation be typed equally may seem stronger than necessary, due to redundancy in the representation of permutations: for instance, $(a\ b)(b\ a)$ is the identity permutation, but in a Church-style system these swappings can only be built if both atoms have the same type. Although for a given permutation, π, it might suffice to check that for each atom a in the support of π, $\Gamma_a \equiv \Gamma_{\pi(a)}$, swappings with atoms of different types, even if redundant, would be at odds with the Church-style nature of the system.

Definition 3.3 (Well-typed Terms). Typing judgements have the form $\Gamma \Vdash_\Sigma t\colon \tau$, where Γ is a typing environment, Σ a signature, t a term and τ a type. If there exists Γ, Σ and τ such that $\Gamma \Vdash_\Sigma t\colon \tau$ may be derived using the following axioms and rules, then t is a well-typed term.

$$\frac{\sigma \equiv \tau}{\Gamma \Vdash_\Sigma a_\sigma\colon \tau}\,(atm)^\tau \qquad \frac{\Gamma_X \equiv \tau}{\Gamma \Vdash_\Sigma \pi \cdot X\colon \tau}\,(var)^\tau \qquad \frac{\Gamma \Vdash_\Sigma t\colon \tau}{\Gamma \Vdash_\Sigma [a_\sigma]\, t\colon [\sigma]\, \tau}\,(abs)^\tau$$

$$\frac{\Gamma \Vdash_\Sigma t_1\colon \tau_1 \ \ldots\ \Gamma \Vdash_\Sigma t_n\colon \tau_n}{\Gamma \Vdash_\Sigma (t_1,\ \ldots,\ t_n)\colon (\tau_1 \times \ldots \times \tau_n)}\,(tpl)^\tau \qquad \frac{\Sigma_{\mathsf{f}} \equiv \langle \sigma \hookrightarrow \tau\rangle \quad \Gamma \Vdash_\Sigma t\colon \sigma}{\Gamma \Vdash_\Sigma \mathsf{f}\, t\colon \tau}\,(app)^\tau$$

Example 3.4 (Well-typed Terms). Consider a formalisation of first-order logic over the domain of the natural numbers. Let $\mathtt{Nat}$ and $\mathtt{o}$ be base types for the natural numbers and formulas respectively. Then, considering a signature, Σ, of the form, $\{0\colon \langle () \hookrightarrow \mathtt{Nat}\rangle$, $\mathsf{suc}\colon \langle \mathtt{Nat} \hookrightarrow \mathtt{Nat}\rangle$, $\mathsf{eql}\colon \langle (\mathtt{Nat} \times \mathtt{Nat}) \hookrightarrow \mathtt{o}\rangle$, $\mathsf{and}\colon \langle (\mathtt{o} \times \mathtt{o}) \hookrightarrow \mathtt{o}\rangle$, $\mathsf{not}\colon \langle \mathtt{o} \hookrightarrow \mathtt{o}\rangle$, $\mathsf{all}\colon \langle [\mathtt{Nat}]\, \mathtt{o} \hookrightarrow \mathtt{o}\rangle\}$, one has the following

examples of well-typed terms.

$$X : \mathsf{Nat} \Vdash_\Sigma \mathsf{suc}\,\mathsf{suc}\,X : \mathsf{Nat}$$
$$P : \mathsf{o} \Vdash_\Sigma \mathsf{all}\,[a_{\mathsf{Nat}}]\,P : \mathsf{o}$$
$$\Vdash_\Sigma \mathsf{not}\,\mathsf{eql}\,(0,\ \mathsf{suc}\,\mathsf{suc}\,a_{\mathsf{Nat}}) : \mathsf{o}$$
$$P : \mathsf{o} \Vdash_\Sigma \mathsf{and}\,(\mathsf{all}\,[a_{\mathsf{Nat}}]\,P,\ \mathsf{not}\,\mathsf{eql}\,(0,\ \mathsf{suc}\,\mathsf{suc}\,a_{\mathsf{Nat}})) : \mathsf{o}$$

3.2.2 Action of Permutations and Variable Substitutions. Now consider the action of permutations over atoms and well-typed terms. Notice that the possibility that atoms may have the same name but be of different types, results in the proliferation of rules required to define the permutation action on atoms in comparison with Definition 2.3.

Definition 3.5 (Permutation Action on Atoms).

$$\mathrm{id}(a_\tau) \triangleq a_\tau \qquad (\pi * (a_\tau\ b_\tau))(c_\sigma) \triangleq \pi(c_\sigma)$$

$$(\pi * (a_\tau\ b_\tau))(a_\sigma) \triangleq \begin{cases} \pi(b_\tau) & \text{if } \sigma \equiv \tau \\ \pi(a_\sigma) & \text{if } \sigma \not\equiv \tau \end{cases} \qquad (\pi * (a_\tau\ b_\tau))(b_\sigma) \triangleq \begin{cases} \pi(a_\tau) & \text{if } \sigma \equiv \tau \\ \pi(b_\sigma) & \text{if } \sigma \not\equiv \tau \end{cases}$$

Definition 3.6 (Permutation Action on Terms).

$$\pi \cdot a_\tau \triangleq \pi(a_\tau) \qquad \pi \cdot (\pi' \cdot X) \triangleq (\pi \mathbin{@} \pi') \cdot X \qquad \pi \cdot [a_\tau]\,t \triangleq [\pi(a_\tau)]\,\pi \cdot t$$

$$\pi \cdot (t_1,\ \ldots,\ t_n) \triangleq (\pi \cdot t_1,\ \ldots,\ \pi \cdot t_n) \qquad \pi \cdot \mathsf{f}\,t \triangleq \mathsf{f}\,(\pi \cdot t)$$

Substitutions act on pseudo-terms and well-typed terms as specified in Definition 2.9; the inductive definition is omitted. Despite variable substitution being atom-capturing, the fact that atoms are directly annotated prohibits capture from affecting the typing of a term. Thus it follows straightforwardly that given a substitution that respects the types of the variables of a term, the type of the term is preserved following instantiation.

3.2.3 Freshness and Alpha-equivalence. Finally, consider the definitions of freshness and alpha-equivalence for well-typed terms of the system. Again notice the increased number of rules in the definition of freshness in comparison to Definition 2.11 resulting from the presence of atoms that may have the same name but different types.

Definition 3.7 (Freshness Relation). Freshness judgements $\Delta \vdash a_\sigma \# t$, where t is a well-typed term, are derived using the following set of axioms and rules.

$$\frac{\sigma \not\equiv \tau}{\Delta \vdash a_\sigma \# a_\tau}\,(atm : aa)^\# \qquad \frac{}{\Delta \vdash a_\sigma \# b_\tau}\,(atm : ab)^\#$$

$$\frac{\pi^{-1}(a_\sigma) \# X \in \Delta}{\Delta \vdash a_\sigma \# \pi \cdot X}\,(var)^\# \qquad \frac{\Delta \vdash a_\sigma \# t}{\Delta \vdash a_\sigma \# [b_\tau]\,t}\,(abs : ab)^\#$$

$$\frac{\sigma \equiv \tau}{\Delta \vdash a_\sigma \# [a_\tau]\,t}\,(abs : aa)^\# \qquad \frac{\sigma \not\equiv \tau \quad \Delta \vdash a_\sigma \# t}{\Delta \vdash a_\sigma \# [a_\tau]\,t}\,(abs : aa')^\#$$

$$\frac{\Delta \vdash a_\sigma \# t_1 \ \ldots\ \Delta \vdash a_\sigma \# t_n}{\Delta \vdash a_\sigma \# (t_1,\ \ldots,\ t_n)}\,(tpl)^\# \qquad \frac{\Delta \vdash a_\sigma \# t}{\Delta \vdash a_\sigma \# \mathsf{f}\,t}\,(app)^\#$$

Definition 3.8 (Alpha-equivalence Relation). Let s and t be two well-typed terms under the same environment Γ, and same signature Σ, that is, $\Gamma \Vdash_\Sigma s : \sigma$ and $\Gamma \Vdash_\Sigma t : \tau$.

$$\frac{\sigma \equiv \tau}{\Delta \vdash a_\sigma \approx_\alpha a_\tau}\,(atm)^\alpha \qquad \frac{\forall a_\sigma \in \mathsf{ds}(\pi,\ \pi')\ .\ a_\sigma \,\#\, X \in \Delta}{\Delta \vdash \pi \cdot X \approx_\alpha \pi' \cdot X}\,(var)^\alpha$$

$$\frac{\sigma \equiv \tau \quad \Delta \vdash s \approx_\alpha t}{\Delta \vdash [a_\sigma]\, s \approx_\alpha [a_\tau]\, t}\,(abs : aa)^\alpha \qquad \frac{\sigma \equiv \tau \quad \Delta \vdash a_\sigma \,\#\, t \quad \Delta \vdash s \approx_\alpha (a_\sigma\ b_\tau) \cdot t}{\Delta \vdash [a_\sigma]\, s \approx_\alpha [b_\tau]\, t}\,(abs : ab)^\alpha$$

$$\frac{\Delta \vdash s_1 \approx_\alpha t_1 \ \ldots\ \Delta \vdash s_n \approx_\alpha t_n}{\Delta \vdash (s_1,\ \ldots,\ s_n) \approx_\alpha (t_1,\ \ldots,\ t_n)}\,(tpl)^\alpha \qquad \frac{\Delta \vdash s \approx_\alpha t}{\Delta \vdash \mathsf{f}\, s \approx_\alpha \mathsf{f}\, t}\,(app)^\alpha$$

If one considers once again the signature defined in Example 3.4 then one has the following examples of freshness and alpha-equivalence over well-typed terms.

Example 3.9 (Freshness and Alpha-equivalence).

$$\varnothing \vdash a_{\mathsf{Nat}} \,\#\, \mathsf{suc}\, b_{\mathsf{Nat}} \qquad \varnothing \vdash a_{\mathsf{Nat}} \,\#\, \mathsf{suc}\, a_{\mathsf{Nat}\Rightarrow\mathsf{Nat}} \qquad \varnothing \not\vdash a_{\mathsf{Nat}} \,\#\, \mathsf{suc}\, a_{\mathsf{Nat}}$$
$$b_{\mathsf{Nat}} \,\#\, P \vdash \mathsf{all}\, [a_{\mathsf{Nat}}]\, P \approx_\alpha \mathsf{all}\, [b_{\mathsf{Nat}}]\, (a_{\mathsf{Nat}}\ b_{\mathsf{Nat}}) \cdot P$$

This section concludes by stating the main properties of this simple type system.

Theorem 3.10. (1) *$\Gamma \Vdash_\Sigma t : \tau$ implies $\Gamma \Vdash_\Sigma \pi \cdot t : \tau$ for any permutation π.*
(2) *If $\Gamma \Vdash_\Sigma t : \tau$ and θ is a variable substitution such that for any $X \in \mathsf{vars}(\Gamma)$, $\Gamma \Vdash_\Sigma \theta(X) : \Gamma_X$ then $\Gamma \Vdash_\Sigma t\,\theta : \tau$.*
(3) *Given two well-typed terms s and t, such that $\Gamma \Vdash_\Sigma s : \sigma$ and $\Gamma \Vdash_\Sigma t : \tau$, if $\Delta \vdash s \approx_\alpha t$ then $\sigma \equiv \tau$.*

Proof. Parts 1 and 2 are proved by induction on the type derivation for $\Gamma \Vdash_\Sigma t : \tau$ as given in Definition 3.3. Part 2 uses Part 1 in the case of a variable. Part 3 is proved by induction on the derivation of $\Delta \vdash s \approx_\alpha t$, using Part 1 in the case of $(abs : ab)^\alpha$. □

The enforced annotation of all atoms with types makes the syntax of this system rather cumbersome and it could instead be presented using of a fixed typing environment for atoms. Such an approach would have the additional advantage of removing the confusion created by atoms of the same name but of different types. An extra relation would be required to define well-typed permutations but the syntax of pseudo-permutations and pseudo-types could be the same as that for standard nominal terms as given in Definition 2.1. However, instead of pursuing this alternative presentation, the next subsection investigates a more standard approach to alleviating this problem, in which a typing environment for atoms may be updated by local type annotations for abstractions.

3.3 Type System in the Style of de Bruijn

This subsection describes a Church-style type system in the manner of de Bruijn [7]. It is Church-style in that it is based on the notion of typed atoms and only well-typed terms are considered to exist, but it is presented by adopting the practice of a Curry-style system whereby atoms are typed from associations in an accompanying environment. The atom of an abstraction is annotated with a type and the environment is updated with a new association when such a construction is encountered. The atom-capturing nature of variable substitution over nominal terms combined with this updating of type associations, produces a conflict with the desire to type multiple occurrences of a variable in the same environment and thus results in a more complex system than that of the previous subsection or the simply typed lambda calculus.

Definition 3.11 (Pseudo-permutations and Pseudo-terms).

$$\pi ::= \mathsf{id} \mid \pi * (a\ b) \qquad s,\ t ::= a \mid [a\colon \tau]\, t \mid \mathsf{f}\, t \mid (t_1,\ \ldots,\ t_n) \mid \pi \cdot X$$

Let environments now include associations for atoms, $(a\colon \sigma)$, in addition to those for variables and subject to the same condition of uniqueness. Extend the notation, Γ_X, to atoms, Γ_a and write $\mathsf{atms}(\Gamma)$ to denote the set of atoms for which associations exist in Γ, that is, $\{a \mid \Gamma_a \neq \bot\}$. Let an association for an atom, a, now be added to an environment, Γ, written $\Gamma \bowtie (a\colon \tau)$ where this means $\Gamma \cup \{(a\colon \tau)\}$ such that if $a \in \mathsf{atms}(\Gamma)$ then $\Gamma_a \equiv \tau$. This notation is extended element-wise to sets of associations. For example, if Γ is the environment $\{(a\colon \tau_1),\ (c\colon \tau_2)\}$ then $\Gamma \bowtie \{(a\colon \tau_1),\ (b\colon \tau_3)\}$ denotes the environment $\{(a\colon \tau_1),\ (b\colon \tau_3),\ (c\colon \tau_2)\}$. Associations for variables may not be added to an environment. Write ${}^{\pi}\Gamma$ to denote the environment obtained by applying the permutation π, to the atoms in the environment Γ.

The constraint that an association for an atom may not be changed in the environment is a reflection of the Church-style nature of the system and the presence of atom-capturing substitution for variables. In contrast, a simply typed lambda calculus presented in this way can allow types for variables to be updated in Γ, because there substitution is non-capturing.

3.3.1 Well-typed Terms. The well-typedness of a permutation π in an environment Γ is given by the predicate typP and holds when $\Gamma \Vdash \mathsf{typP}(\pi)$ can be derived by the rules in Definition 3.12.

Definition 3.12 (Well-typed Permutations).

$$\frac{}{\Gamma \Vdash \mathsf{typP}(\mathsf{id})}\ (id)^{\tau} \qquad \frac{\Gamma_a \equiv \tau \equiv \Gamma_b \quad \Gamma \Vdash \mathsf{typP}(\pi)}{\Gamma \Vdash \mathsf{typP}(\pi * (a\ b))}\ (prm)^{\tau}$$

Definition 3.12 enforces the same constraint on permutations as that enforced by construction in Definition 3.3 of the previous system, namely that only swappings of atoms of the same type are permitted in permutations.

Note that $\Gamma \Vdash \mathsf{typP}(\pi)$ if and only if $\Gamma \Vdash \mathsf{typP}(\pi^{-1})$.

A **typing judgement** is a tuple, $(\Gamma,\ \Sigma,\ s,\ \tau)$, of an environment Γ, a signature Σ, a pseudo-term s and a type τ, and is written $\Gamma \Vdash_{\Sigma} s\colon \tau$.

Quasi-typing judgements, written $\Gamma \Vdash\!\!\!|_{\Sigma} s\colon \tau$, are inductively defined as follows below (Definition 3.13). The validity of a typing judgement is determined in two stages. First, a derivation for the corresponding quasi-typing judgement is constructed using the rules and axioms given in Definition 3.13, and then a property on the typing of the variables of the term is checked as described in Definition 3.17.

Definition 3.13 (Derivable Quasi-Typing Judgements).

$$\frac{\Gamma_a \equiv \tau}{\Gamma \Vdash\!\!\!|_{\Sigma} a\colon \tau}\ (atm)^{\tau} \qquad \frac{\Gamma_X \equiv \tau \quad \Gamma \Vdash \mathsf{typP}(\pi)}{\Gamma \Vdash\!\!\!|_{\Sigma} \pi \cdot X\colon \tau}\ (var)^{\tau}$$

$$\frac{\Gamma \bowtie (a\colon \sigma) \Vdash\!\!\!|_{\Sigma} t\colon \tau}{\Gamma \Vdash\!\!\!|_{\Sigma} [a\colon \sigma]\, t\colon [\sigma]\, \tau}\ (abs)^{\tau}$$

$$\frac{\Gamma \Vdash\!\!\!|_{\Sigma} t_1\colon \tau_1,\ \ldots,\ \Gamma \Vdash\!\!\!|_{\Sigma} t_n\colon \tau_n}{\Gamma \Vdash\!\!\!|_{\Sigma} (t_1,\ \ldots,\ t_n)\colon (\tau_1 \times \ldots \times \tau_n)}\ (tpl)^{\tau} \qquad \frac{\Sigma_{\mathsf{f}} \equiv \langle \sigma \hookrightarrow \tau \rangle \quad \Gamma \Vdash\!\!\!|_{\Sigma} t\colon \sigma}{\Gamma \Vdash\!\!\!|_{\Sigma} \mathsf{f}\, t\colon \tau}\ (app)^{\tau}$$

Proposition 3.14 (Form of Sub-judgements). *Suppose $\Gamma \Vdash\!\!\!|_{\Sigma} t\colon \tau$. All sub-judgements of the derivation of $\Gamma \Vdash\!\!\!|_{\Sigma} t\colon \tau$ have the form, $\Gamma \bowtie \Gamma' \Vdash\!\!\!|_{\Sigma} t'\colon \tau'$ where t' is a subterm of t and Γ' contains the type associations for abstracted atoms that were added to Γ.*

Proof. By induction on the structure of the derivation. The property holds trivially for atoms and variables; the other cases follow directly by induction. □

Since different occurrences of a variable in a term may occur under the scope of different abstractions, they may be typed in different type environments within a type derivation. The notions of variable judgement and essential environment are introduced to refer to the type environments that are relevant for each variable occurrence.

Definition 3.15 (Variable Judgement). Suppose $\Gamma \Vdash_\Sigma t : \tau$. Call sub-judgements of the form $\Gamma \bowtie \Gamma' \Vdash_\Sigma \pi \cdot X : \tau'$ in the derivation of $\Gamma \Vdash_\Sigma t : \tau$ the **variable judgements** of the derivation.

Note that for a given derivation, there is a unique variable judgement for each occurrence of a variable in a term.

The variable judgements indicate the types of the atoms that will be captured when the variable is instantiated. However, since variables in nominal terms have permutations suspended over them, the inverse of the permutation should be applied to the typing environments of variable judgements in order to compute the actual environments in which the image of a substitution will be typed when the term is instantiated. Such an environment is called an *essential environment*.

Definition 3.16 (Essential Environment). Let $\Gamma' \Vdash_\Sigma \pi \cdot X : \tau'$ be a variable judgement of a derivation of $\Gamma \Vdash_\Sigma t : \tau$. Then, call ${}^{\pi^{-1}}\Gamma'$ the **essential environment** of the variable judgement $\Gamma' \Vdash_\Sigma \pi \cdot X : \tau'$.

A property, called the **compatibility** property, is now defined for derivations of quasi-typing judgements. A quasi-typing judgement that has a derivation for which this property holds is a typing judgement.

Definition 3.17 (Compatibility Property). The **compatibility** property holds for a derivation of $\Gamma \Vdash_\Sigma t : \tau$, if for each variable $X \in \mathsf{vars}(t)$, the essential environments of all variable judgements for X throughout the derivation are equal.

Definition 3.18 (Derivable Typing Judgements). A typing judgement $\Gamma \Vdash_\Sigma t : \tau$ is derivable, if there exists a derivation of the corresponding quasi-typing judgement for which the compatibility property holds.

Informally, a derivable quasi-typing judgement becomes a derivable typing judgement if each occurrence of a variable $X \in \mathsf{vars}(t)$ is typed in the same environment.

Consider again the signature of Example 3.4. The examples of well-typed terms given there, have the following derivable judgements in this de Bruijn-style system. Note that in the final example, given a richer domain, the two abstractions of the atom a could be given different types.

Example 3.19 (Derivable Typing Judgements).

$$X : \mathsf{Nat} \Vdash_\Sigma \mathsf{suc}\,\mathsf{suc}\,X : \mathsf{Nat}$$
$$P : \mathsf{o} \Vdash_\Sigma \mathsf{all}\,[a : \mathsf{Nat}]\,P : \mathsf{o}$$
$$a : \mathsf{Nat} \Vdash_\Sigma \mathsf{not\,eql}\,(0,\ \mathsf{suc}\,\mathsf{suc}\,a) : \mathsf{o}$$
$$a : \mathsf{Nat},\ P : \mathsf{o} \Vdash_\Sigma \mathsf{and}\,(\mathsf{all}\,[a : \mathsf{Nat}]\,P,\ \mathsf{not\,eql}\,(0,\ \mathsf{suc}\,\mathsf{suc}\,a)) : \mathsf{o}$$
$$P : \mathsf{o} \Vdash_\Sigma \mathsf{and}\,(\mathsf{all}\,[a : \mathsf{Nat}]\,P,\ \mathsf{all}\,[a : \mathsf{Nat}]\,P) : \mathsf{o}$$

However, consider the derivable quasi-typing judgement shown in Example 3.20. Here the variable judgements for the two occurrences of the variable P have different environments, the first $\{P : \mathsf{o},\ a : \mathsf{Nat}\}$, and the second just $\{P : \mathsf{o}\}$. A variable substitution for P would thus have to be typed in multiple, inconsistent environments. Thus the compatibility property does not hold and the typing judgement is not derivable.

Example 3.20 (Failure of the Compatibility Property).

$$P\colon \mathsf{o} \Vdash\!\!\!\vdash_{\Sigma} \text{and}\,(\mathsf{all}\,[a\colon \mathtt{Nat}]\,P,\ P)\colon \mathsf{o}$$

The need to check the equivalence of the environments for each separate occurrence of a variable in a derivation is a result of the atom-capturing nature of variable substitution. Not allowing associations for atoms to be changed in Γ is not sufficient to ensure that the environment for all variable judgements for a given variable are equal because they may occur under different abstractions, producing different environments.

The design of the type system, using the compatibility check, permits flexible typing of abstracted atoms (the same atom can be abstracted with different types in different parts of the term) while ensuring that alpha-equivalent terms have the same type. The approach used in the previous Church system, where atoms are typed with a unique type from the outset, also ensures this property but is more restrictive (and not suitable for the Curry-style typing that will be defined in Section 4).

One could also postpone the typing of permutations as described in the polymorphic type system detailed in Section 4. However, it was decided to enforce earlier and stronger checks on permutations in this system to draw attention to its Church-style nature and to the separate causes of the difficulties arising in these systems.

3.3.2 Action of Permutations and Variable Substitutions. Given an environment Γ, the action of a permutation is defined only for permutations well-typed in Γ, upon any atom $a \in \mathrm{atms}(\Gamma)$. The action of permutations upon terms is defined only for well-typed permutations π, upon well-typed terms t under the *same* typing environment Γ. In the rule for an abstraction the fact that the term is well-typed ensures that the type of the abstracted atom is the same for any existing typing for that atom in Γ and that it must therefore be compatible with the typing of the permutation.

Definition 3.21 (Permutation Action on Terms).

$$\pi \cdot a \triangleq \pi(a) \qquad \pi \cdot (\pi' \cdot X) \triangleq (\pi \mathbin{@} \pi') \cdot X \qquad \pi \cdot [a\colon \tau]\, t \triangleq [\pi(a)\colon \tau]\, \pi \cdot t$$

$$\pi \cdot (t_1, \ldots, t_n) \triangleq (\pi \cdot t_1, \ldots, \pi \cdot t_n) \qquad \pi \cdot \mathsf{f}\, t \triangleq \mathsf{f}\,(\pi \cdot t)$$

The following lemma states that the composition of two well-typed permutations in Γ is also well-typed in Γ, and is used in Theorem 3.23 to prove that well-typed permutations preserve types.

Lemma 3.22. *If* $\Gamma \Vdash \mathsf{typP}(\pi)$ *and* $\Gamma \Vdash \mathsf{typP}(\pi')$ *then* $\Gamma \Vdash \mathsf{typP}(\pi \mathbin{@} \pi')$.

Proof. By induction on the number of swappings in $\pi \mathbin{@} \pi'$, using the fact that each swapping $(a\ b)$ in $\pi \mathbin{@} \pi'$ comes either from π or π' and therefore $\Gamma_a \equiv \Gamma_b$ as seen from Definition 3.12. □

Theorem 3.23 (Object-level Equivariance of Typing Judgements). *If* $\Gamma \Vdash\!\!\!\vdash_{\Sigma} t\colon \tau$ *and* $\Gamma \Vdash \mathsf{typP}(\pi)$ *then* ${}^{\pi}\Gamma \Vdash\!\!\!\vdash_{\Sigma} \pi \cdot t\colon \tau$.

Proof. By induction on the derivation for $\Gamma \Vdash\!\!\!\vdash_{\Sigma} t\colon \tau$ for which see Definition 3.13. Distinguish cases depending on the last rule applied. In the case of $(atm)^{\tau}$ and $(abs)^{\tau}$, the result follows from the fact that because $\Gamma \Vdash \mathsf{typP}(\pi)$, by Definition 3.12, π swaps atom of the same type in Γ. The case $(var)^{\tau}$ follows from Lemma 3.22. The other cases follow directly by induction. In the case of a tuple $(t_1, \ldots, t_n)$, if some variable X occurs in different subterms, each variable judgement has the same essential environment E, and the corresponding essential environment for $\pi \cdot X$ in $\pi \cdot t$ is ${}^{\pi}E$. □

Substitutions act on pseudo-terms and well-typed terms in a similar fashion as to that specified in Definition 2.9; the definition is inductive and omitted. The notion of a well-typed substitution

is now defined, the action of which is shown to preserve the types of instantiated terms. Below, for simplicity, consider only idempotent substitutions that instantiate all the variables in a term (a substitution θ is idempotent if $\theta\,\theta = \theta$). This is a standard assumption, always satisfied by the substitutions used in rewriting steps, since rewrite rules assume variables that are different from the variables of the term to be rewritten.

Definition 3.24 (Well-typed Substitution). An idempotent substitution θ, such that $\mathrm{dom}(\theta) \subseteq \mathrm{vars}(t)$, is well-typed in Γ and Σ for $\Phi \Vdash_\Sigma t : \tau$ under the following conditions.

(1) $\Gamma = \Gamma' \bowtie \Phi_{atm}$, where Φ_{atm} denotes the set of declarations for atoms provided in Φ. Thus, Γ contains the associations for atoms provided by Φ and possibly also declarations for other atoms and variables from the image of θ.
(2) For each $X \in \mathrm{vars}(t)$, where $E = E_{atm} \cup E_{var}$ is the essential environment of the variable judgement for X in $\Phi \Vdash_\Sigma t : \tau$ (here E_{atm} contains the declarations for atoms and E_{var} the declarations for variables): $\Gamma' \bowtie E_{atm} \Vdash_\Sigma \theta(X) : \Phi_X$.
(3) The set of type derivations $\{\Gamma' \bowtie E_{atm} \Vdash_\Sigma \theta(X) : \Phi_X \mid X \in \mathrm{vars}(t)\}$ has the compatibility property, that is, for each variable in the image of θ, the essential environments of the variable judgements for that variable throughout all the derivations in the set are equal.

Note that Condition 3 is vacuous if $t\,\theta$ is ground. In that case, Condition 1 and Condition 2 become the standard conditions from the lambda calculus, except that $\theta(X)$ is typed using the essential environment of X, which is not explicit in the typing judgement for t.

Theorem 3.25 (Well-typed Substitutions Preserve Types). *If θ is well-typed in Γ and Σ for $\Phi \Vdash_\Sigma t : \tau$, then $\Gamma \Vdash_\Sigma t\,\theta : \tau$.*

Proof. By induction on the derivation for $\Phi \Vdash_\Sigma t : \tau$ given by Definition 3.13, distinguishing cases on the last rule applied. In the case of a moderated variable $(var)^\tau$, $t \equiv \pi \cdot X$ and the suspended permutation π is well-typed in Φ and also in the environment E of each variable judgement (since $E = {}^\pi\Phi$). Without loss of generality, one can assume that Γ' contains only those atoms of Γ not present in Φ and thus $\pi \cdot (\Gamma' \bowtie E_{atm}) = (\Gamma' \bowtie \Phi_{atm})$. The result then follows from Condition 2 of Definition 3.24 by Theorem 3.23. The case of a tuple is by induction and the assumption of the compatibility property on the variables of the image of θ stated in Condition 3 of Definition 3.24. The other cases follow by induction. □

3.3.3 Freshness and Alpha-equivalence. Freshness is defined only with respect to a single typing environment Γ, that is, for an atom $a \in \mathrm{atms}(\Gamma)$ and a term well-typed in Γ. In the case of rule $(abs : aa)^\#$, a must already have a type association and the fact that the term is well-typed in that same environment ensures that it must be of the same type as that given for the abstracted atom.

Definition 3.26 (Freshness Relation).

$$\frac{}{\Delta \vdash a \# b}\,(atm)^\# \qquad \frac{\pi^{-1}(a) \# X \in \Delta}{\Delta \vdash a \# \pi \cdot X}\,(var)^\#$$

$$\frac{}{\Delta \vdash a \# [a : \tau]\,t}\,(abs : aa)^\# \qquad \frac{\Delta \vdash a \# t}{\Delta \vdash a \# [b : \tau]\,t}\,(abs : ab)^\#$$

$$\frac{\Delta \vdash a \# t_1 \;\ldots\; \Delta \vdash a \# t_n}{\Delta \vdash a \# (t_1, \ldots, t_n)}\,(tpl)^\# \qquad \frac{\Delta \vdash a \# t}{\Delta \vdash a \# \mathrm{f}\,t}\,(app)^\#$$

Let s and t be well-typed terms under the same environment Γ and signature Σ, that is, $\Gamma \Vdash_\Sigma s : \sigma$ and $\Gamma \Vdash_\Sigma t : \tau$. Say s is alpha-equivalent to t if $\Delta \vdash s \approx_\alpha t$ can then be derived using the following axioms and rules.

Definition 3.27 (Alpha-equivalence Relation).

$$\frac{}{\Delta \vdash a \approx_\alpha a}\,(atm)^\alpha \qquad \frac{\forall a \in \mathsf{ds}(\pi,\ \pi')\ .\ a \# X \in \Delta}{\Delta \vdash \pi \cdot X \approx_\alpha \pi' \cdot X}\,(var)^\alpha$$

$$\frac{\sigma \equiv \tau \quad \Delta \vdash s \approx_\alpha t}{\Delta \vdash [a\colon \sigma]\, s \approx_\alpha [a\colon \tau]\, t}\,(abs : aa)^\alpha \qquad \frac{\sigma \equiv \tau \quad \Delta \vdash a \# t \quad \Delta \vdash s \approx_\alpha (a\ b) \cdot t}{\Delta \vdash [a\colon \sigma]\, s \approx_\alpha [b\colon \tau]\, t}\,(abs : ab)^\alpha$$

$$\frac{\Delta \vdash s_1 \approx_\alpha t_1 \ \ldots\ \Delta \vdash s_n \approx_\alpha t_n}{\Delta \vdash (s_1,\ \ldots,\ s_n) \approx_\alpha (t_1,\ \ldots,\ t_n)}\,(tpl)^\alpha \qquad \frac{\Delta \vdash s \approx_\alpha t}{\Delta \vdash \mathsf{f}\, s \approx_\alpha \mathsf{f}\, t}\,(app)^\alpha$$

The following examples of freshness and alpha-equivalence correspond to those given for the Church-style system in Example 3.9.

Example 3.28 (Freshness and Alpha-equivalence). The atoms and terms mentioned below are well-typed in $\Gamma = \{a\colon \mathsf{Nat},\ b\colon \mathsf{Nat},\ P\colon \mathsf{o}\}$. The following judgements are then derivable using the rules in Definition 3.26 and Definition 3.27.

$$\varnothing \vdash a \# \mathsf{suc}\, b \qquad \varnothing \not\vdash a \# \mathsf{suc}\, a$$
$$b \# P \vdash \mathsf{all}\, [a\colon \mathsf{Nat}]\, P \approx_\alpha \mathsf{all}\, [b\colon \mathsf{Nat}]\, (a\ b) \cdot P$$

Theorem 3.29 (Types and Alpha-equivalence). *Given two well-typed terms s and t, such that $\Gamma \Vdash_\Sigma s\colon \sigma$ and $\Gamma \Vdash_\Sigma t\colon \tau$, if $\Delta \vdash s \approx_\alpha t$ then $\sigma \equiv \tau$.*

Proof. By induction on the derivation of $\Delta \vdash s \approx_\alpha t$. The cases of $(atm)^\alpha$ and $(var)^\alpha$ are trivial. In the case of $(abs : ab)^\alpha$ the result follows from Theorem 3.23 and then by induction. The other cases follow directly by induction. □

4 POLYMORPHIC TYPES FOR NOMINAL TERMS

This section describes a polymorphic type system for nominal terms in the style of Curry [7, 26]. It extends the syntax of simple types with type variables and quantified declarations in order to implement a ML-style polymorphic type system [27], in which multiple applications of a term-former within a term may each be given a different type.

The notion of a principal solution to a typeability problem for a term is defined and an algorithm presented to compute such solutions. This algorithm is shown to be sound and complete with respect to the derivable judgements of the type system.

Such a type system is powerful enough to be of practical interest in programming language design and rewriting. Its polymorphism allows for stronger specifications than those possible using the simple type systems in Section 3, whilst its Curry-style nature and the decidability of type inference are of great importance in terms of the reduction of notational burden. A selection of interesting examples using this system is given in Section 5.

Curry and Church approaches to typing are intrinsically different: in a Church-style system, terms are well typed by construction, whereas in systems à la Curry, terms can be typeable or untypeable — types are assigned to (some) terms a posteriori. This change of perspective has a deep impact: in a Church-style system, the atoms a_σ and a_τ (where σ and τ are different types) are different, so, for example, the term $(a_\sigma,\ a_\tau)$ has two atoms; in a Curry-style system, the term $(a,\ a)$ has only one atom. For this reason, the system defined in this section is *not* a polymorphic extension of the previous simply typed systems. A polymorphic Church system can be defined following the ideas presented in this section, but a Curry style system, being closer to the ML-style of typing, is better for applications in programming language design and rewriting.

4.1 Polymorphic Types

In this subsection, the syntax of simple types is extended with the addition of type variables and quantified declarations for term-formers. The former necessitates a formalisation of type substitutions and the associated action of type instantiation, and the latter a notion of generic type instantiation.

4.1.1 Types and Declarations. Consider a countably infinite set of **type variables**, $\mathbb{T}$, pairwise disjoint from the set of type-formers, $\mathbb{C}$, and let $\alpha,\ \beta,\ \gamma,\ \ldots$, range over elements in $\mathbb{T}$. Extending the syntax of the simple types given in Section 3.1.1, let polymorphic types and type declarations be defined by the grammar described below in Definition 4.1.

Definition 4.1 (Polymorphic Types and Declarations).

$$\sigma,\ \tau ::= \alpha \mid [\sigma]\,\tau \mid \mathsf{C}\,\tau \mid (\tau_1 \times \ldots \times \tau_n) \qquad \rho,\ \xi ::= \forall(\overline{\alpha}).\langle \sigma \hookrightarrow \tau \rangle$$

Call the first, new construction, a **variable type**.

In the rule for a declaration, $\overline{\alpha}$ denotes any finite set of type variables (if empty, the initial quantification may be omitted entirely); these type variables are **generic type variables** and are *bound* throughout the declaration.

Write $\mathrm{tvars}(\sigma)$ for the set of type variables occurring in the type, σ. Extend this notation to declarations, $\mathrm{tvars}(\rho)$, to represent the set of type variables which occur in ρ but are *not* bound; that is, for a declaration $\forall(\overline{\alpha}).\langle \sigma \hookrightarrow \tau \rangle$, the set $(\mathrm{tvars}(\sigma) \cup \mathrm{tvars}(\tau)) \setminus \overline{\alpha}$. Call the elements of this set the **free** type variables of ρ.

Write $\equiv$ for syntactic equality over types and extend this to declarations, modulo the standard notion of α-equivalence of bound type variables. Note that nominal syntax is not used for the definition of declarations (that is, type variables are not modelled as atoms but as conventional meta-level variables).

Example 4.2 (Equality of Declarations).

$$\forall(\alpha).\langle \mathtt{List}\,\alpha \hookrightarrow \mathtt{Nat} \rangle \ \equiv\ \forall(\beta).\langle \mathtt{List}\,\beta \hookrightarrow \mathtt{Nat} \rangle \ \not\equiv\ \forall(\alpha).\langle \mathtt{List}\,\gamma \hookrightarrow \mathtt{Nat} \rangle$$

4.1.2 Type Substitutions and Instantiation.

Definition 4.3 (Type Substitution). A **type substitution**, S, is a mapping from type variables to types, equal to the identity mapping but for finitely many arguments, and written as a set of bindings $[\alpha_1 \mapsto \tau_1] \ldots [\alpha_n \mapsto \tau_n]$, such that the type variables, $\alpha_1,\ \ldots,\ \alpha_n$, are pairwise distinct.

Let $S,\ T,\ U,\ \ldots$ range over type substitutions. The **domain** of a type substitution S, written $\mathrm{dom}(S)$, is the set of type variables, $\{\alpha \mid \alpha\, S \not\equiv \alpha\}$. Let id denote the identity substitution; that is the substitution whose domain is the empty set. Composition of type substitutions is written $S \circ T$, that is, the substitution equivalent to applying S followed by T.

Definition 4.4 (Instantiation). The action of a type substitution, S, upon a type, τ, called **instantiation**, is written to the right thus, $\tau\, S$, and is defined in the usual inductive way as described below. Instantiation is extended to declarations in the usual capture-avoiding manner.

$$\alpha\, S \triangleq S(\alpha) \qquad ([\sigma]\,\tau)\, S \triangleq [\sigma\, S]\,\tau\, S \qquad (\mathsf{C}\,\tau)\, S \triangleq \mathsf{C}\,(\tau\, S)$$

$$(\tau_1 \times \ldots \times \tau_n)\, S \triangleq (\tau_1\, S \times \ldots \times \tau_n\, S)$$

Example 4.5 (Instantiation of Types and Declarations).

$$([\alpha]\,\alpha)\,[\alpha \mapsto \beta] = [\beta]\,\beta$$

$$(\forall(\beta).\langle(\alpha \times \beta) \hookrightarrow \alpha\rangle)\,[\alpha \mapsto \beta] = \forall(\beta').\langle(\beta \times \beta') \hookrightarrow \beta\rangle$$

Substitutions are partially ordered by instantiation. Define the **most general unifier** of two types, written $\mathrm{mgu}(\sigma,\ \tau)$, as a least element, S, if one exists, in an ordering of substitutions, such that, $\sigma\, S \equiv \tau\, S$. [4] is an excellent reference to detailed algorithms for the calculation of most general unifiers.

Extend the notion of the ordering of types by instantiation and the most general unifier of two types to declarations in the obvious way by considering the declaration as a tuple (this ordering on declarations will be used after a generic instantiation has eliminated the quantifiers, see Definition 4.6 below).

4.1.3 Generic Instantiation. Generic instantiation acts only upon generic type variables and therefore only upon declarations.

Definition 4.6 (Generic Instantiation). A declaration, ρ, of the form $\forall(\overline{\alpha}).\langle\sigma \hookrightarrow \tau\rangle$, has a generic instance, written $\rho \succcurlyeq \xi$, where ξ is a declaration of the form $\forall().\langle\sigma' \hookrightarrow \tau'\rangle$ and $\sigma\, S \equiv \sigma'$ and $\tau\, S \equiv \tau'$ for some type substitution, S, where S instantiates *all but only* the generic type variables in ρ.

Example 4.7 (Generic Type Instantiation).

$$\forall(\alpha,\ \beta).\langle(\alpha \times \beta) \hookrightarrow \alpha\rangle \succcurlyeq \langle(\mathtt{Nat} \times \mathtt{Bool}) \hookrightarrow \mathtt{Nat}\rangle$$
$$\forall(\alpha).\langle(\alpha \times \beta) \hookrightarrow \alpha\rangle \not\succcurlyeq \langle(\mathtt{Nat} \times \mathtt{Bool}) \hookrightarrow \mathtt{Nat}\rangle$$
$$\forall(\alpha,\ \beta).\langle(\alpha \times \beta) \hookrightarrow \alpha\rangle \succcurlyeq \langle(\alpha \times \beta) \hookrightarrow \alpha\rangle$$
$$\forall(\alpha).\langle(\alpha \times \beta) \hookrightarrow \alpha\rangle \succcurlyeq \langle(\mathtt{Nat} \times \beta) \hookrightarrow \mathtt{Nat}\rangle$$

Definition 4.8 (Most General Generic Instance). The most general generic instance, ξ, of a declaration, ρ, is the generic instance, such that for any other generic instance, ξ', $\xi \preceq \xi'$.

The following useful technical result follows by an easy induction.

Lemma 4.9. *If $\rho \succcurlyeq \xi$ then $\rho\, S \succcurlyeq \xi\, S$.*

Proof. By induction over the syntax of the pair of types forming the declaration. A similar result is proven in [27]. □

4.2 Type System in the Style of Curry

The following subsection starts by recalling and conflating those concepts relating to environments and signatures given in Section 3 and adding definitions relating to type variables and type substitutions. Following that, a description of the type system is given, together with proofs of a number of properties over derivable typing judgements, the most important of which is that alpha-equivalence respects typeability.

4.2.1 Environments. A **type association** is a pair of a variable, X, and a type, σ, written $(X : \sigma)$, or a pair of an atom, a, and a type, σ, written $(a : \sigma)$.

A typing **environment**, Γ, is a set of type associations subject to the condition that if $(X : \sigma) \in \Gamma$ and $(X : \tau) \in \Gamma$ then $\sigma \equiv \tau$ (and similarly for atom associations).

As usual, Γ_X denotes the type **associated** with the variable X in the environment Γ. If there is no association for X in Γ, Γ_X is undefined, written $\bot$. This notation is extended to atoms.

The equality relation on types is extended to environments in the obvious way. Bracketing in environments may often be omitted for the sake of clarity.

Write ${}^{\pi}\Gamma$ to denote the environment obtained by applying the permutation, π, to the atoms in the environment, Γ.

Let the association for an atom, a, be **updated** in an environment, Γ, written $\Gamma \bowtie (a\colon \tau)$ where this means either $\Gamma \cup \{(a\colon \tau)\}$ or $(\Gamma \setminus \{(a\colon \sigma)\}) \cup \{(a\colon \tau)\}$ as well-formedness demands. This notation is extended element-wise to sets of associations.

Note that, unlike the notion of an update in Section 3, it is now permitted to change the types of atoms, reflecting the Curry-style nature of the system. For example, if Γ is the environment $\{(a\colon \sigma_1),\ (c\colon \sigma_2)\}$ then $\Gamma \bowtie \{(a\colon \tau_1),\ (b\colon \tau_2)\}$ denotes the environment $\{(a\colon \tau_1),\ (b\colon \tau_2),\ (c\colon \sigma_2)\}$.

The associations for variables in an environment may not be updated.

Write $\Gamma\, S$ for the environment obtained by applying the type substitution, S, to the types in the environment, Γ.

Let $\mathrm{tvars}(\Gamma)$ denote the set of type variables occurring in the environment, Γ, and $S\,|_\Gamma$, the restriction of the domain of the type substitution, S, to those type variables occurring in Γ.

4.2.2 Signatures. A **declaration association** is a pair of a term-former, f, and a declaration, ρ, written $(\mathsf{f}\colon \rho)$; for example, $(\mathsf{suc}\colon \langle \mathsf{Nat} \hookrightarrow \mathsf{Nat}\rangle)$ and $(\mathsf{0}\colon \langle () \hookrightarrow \mathsf{Nat}\rangle)$.

A **signature**, Σ, is a set of declaration associations, again subject to the condition that, if $(\mathsf{f}\colon \rho) \in \Sigma$ and $(\mathsf{f}\colon \xi) \in \Sigma$ then $\rho \equiv \xi$. Σ_f denotes the declaration associated with the term-former, f, in Σ. If no associated declaration for f exists in Σ, then Σ_f is undefined, written $\bot$.

All type variables occurring in declarations in a signature are considered to be generic type variables; that is, for every declaration, ρ, $\mathrm{tvars}(\rho) = \varnothing$.

4.2.3 Typing Judgements. A **typing judgement** is a tuple, $(\Gamma,\ \Sigma,\ \Delta,\ s,\ \tau)$, of an environment, Γ, a signature, Σ, a freshness context, Δ, a term, s and a type, τ, and is written $\Gamma \Vdash_\Sigma \Delta \vdash s\colon \tau$.

Quasi-typing judgements, written $\Gamma \Vdash\!\!\!|_\Sigma \Delta \vdash s\colon \tau$, are defined inductively as follows below. The freshness context Δ is not used in quasi-typing derivations, but will be needed later in Definition 4.13.

Definition 4.10 (Derivable Quasi-typing Judgements).

$$\frac{\Gamma_a \equiv \tau}{\Gamma \Vdash\!\!\!|_\Sigma \Delta \vdash a\colon \tau}\,(atm)^\tau \qquad \frac{\Gamma_X \equiv \tau}{\Gamma \Vdash\!\!\!|_\Sigma \Delta \vdash \pi \cdot X\colon \tau}\,(var)^\tau$$

$$\frac{\Sigma_\mathsf{f} \gg \langle \sigma \hookrightarrow \tau\rangle \quad \Gamma \Vdash\!\!\!|_\Sigma \Delta \vdash t\colon \sigma}{\Gamma \Vdash\!\!\!|_\Sigma \Delta \vdash \mathsf{f}\, t\colon \tau}\,(app)^\tau \qquad \frac{\Gamma \bowtie (a\colon \tau) \Vdash\!\!\!|_\Sigma \Delta \vdash t\colon \tau'}{\Gamma \Vdash\!\!\!|_\Sigma \Delta \vdash [a]\, t\colon [\tau]\, \tau'}\,(abs)^\tau$$

$$\frac{\Gamma \Vdash\!\!\!|_\Sigma \Delta \vdash t_1\colon \tau_1 \ \ldots\ \Gamma \Vdash\!\!\!|_\Sigma \Delta \vdash t_n\colon \tau_n}{\Gamma \Vdash\!\!\!|_\Sigma \Delta \vdash (t_1, \ldots, t_n)\colon (\tau_1 \times \ldots \times \tau_n)}\,(tpl)^\tau$$

The rule $(var)^\tau$ differs from that of the same name in Definition 3.13 in that, here, the permutation, π, is not checked as an assumption of the rule. The correctness of the permutation with respect to the typing of variables is delayed until the definition of derivable typing judgements in Definition 4.16. This is true to the Curry-style nature of the system, in that terms involving a single occurrence of a variable need not be peremptorily rejected as untypeable on the basis of an associated suspended permutation. It is only in the case of multiple occurrences of the same variable in a term, that one truly needs to check that the permutation swaps atoms of compatible types. The notion of quasi-typing also allows terms to be typed irrespective of the action of variable substitutions, if so desired.

The following structural property of quasi-typing derivations is used below.

Proposition 4.11. *Suppose a derivation exists for the quasi-typing judgement, $\Gamma \Vdash\!\!\!|_\Sigma \Delta \vdash t\colon \tau$. All the 'sub-judgements' of this derivation have the form $\Gamma' \Vdash\!\!\!|_\Sigma \Delta \vdash t'\colon \tau'$ where t' is a subterm of t and Γ' is Γ updated with associations for abstracted atoms.*

Proof. By induction on the structure of the derivation. □

4.2.4 Essential Environments for Variable Judgements.

Definition 4.12 (Variable Judgement). Suppose a derivation exists for the quasi-typing judgement $\Gamma \Vdash_\Sigma \Delta \vdash t : \tau$. Call leaves of the form $\Gamma' \Vdash_\Sigma \Delta \vdash \pi \cdot X : \tau'$ the **variable judgements** of the derivation of $\Gamma \Vdash_\Sigma \Delta \vdash t : \tau$.

Note that for a given derivation of a quasi-typing judgement, there is one variable judgement for each separate occurrence of a variable in a term and each one is uniquely defined modulo the names of type variables.

Definition 4.13 (Essential Environment). Let $\Gamma' \Vdash_\Sigma \Delta \vdash \pi \cdot X : \tau'$ be a variable judgement of a derivation of $\Gamma \Vdash_\Sigma \Delta \vdash t : \tau$. Then, call ${}^{\pi^{-1}}\Gamma' \setminus \{(a : \sigma) \mid \Delta \vdash a \# X\}$ the **essential environment** of the variable judgement $\Gamma' \Vdash_\Sigma \Delta \vdash \pi \cdot X : \tau'$.

In the above definition, type associations for atoms that are fresh for a variable are removed, because those atoms can not appear in any instantiation of that variable and so need not be typed. This was not done in the Church system (see Definition 3.16) because freshness is defined only for well-typed terms.

Example 4.14 (Essential Environments). The variable judgements of a derivation of the quasi-typing judgement $a : \mathtt{Int},\ b : \mathtt{Int},\ X : \tau \Vdash_\varnothing \Delta \vdash ((a\ b) \cdot X,\ [a]X) : (\tau \times [\mathtt{Bool}]\,\tau)$ are as follows.

$$a : \mathtt{Int},\ b : \mathtt{Int},\ X : \tau \Vdash_\varnothing \Delta \vdash (a\ b) \cdot X : \tau \tag{1}$$

$$a : \mathtt{Bool},\ b : \mathtt{Int},\ X : \tau \Vdash_\varnothing \Delta \vdash \mathsf{id} \cdot X : \tau \tag{2}$$

If one takes the freshness context, Δ to be the empty set, $\varnothing$, then the corresponding essential environments for these variable judgements are as follows.

$$\{b : \mathtt{Int},\ a : \mathtt{Int},\ X : \tau\} \tag{1}$$

$$\{a : \mathtt{Bool},\ b : \mathtt{Int},\ X : \tau\} \tag{2}$$

However, if Δ is taken to be the freshness context, $\{a \# X\}$, the corresponding essential environments are thus.

$$\{b : \mathtt{Int},\ X : \tau\} \tag{1}$$

$$\{b : \mathtt{Int},\ X : \tau\} \tag{2}$$

4.2.5 Typing Derivations. The **compatibility** property is again used to distinguish those quasi-typing derivations where variable judgements are uniform, in the sense that, for each variable, all occurrences are typed in the same essential environment.

Definition 4.15 (Compatibility Property). The compatibility property holds for a derivation of $\Gamma \Vdash_\Sigma \Delta \vdash t : \tau$, if, for each variable $X \in \mathsf{vars}(t)$, the essential environments of all variable judgements for X throughout the derivation are equal.

Definition 4.16 (Derivable Typing Judgements). A typing judgement $\Gamma \Vdash_\Sigma \Delta \vdash t : \tau$ is derivable, if there exists a derivation of the corresponding quasi-typing judgement for which the compatibility property holds.

Informally, a derivable quasi-typing judgement becomes a derivable typing judgement if each occurrence of a variable $X \in \mathsf{vars}(t)$, is typed in the same environment.

$\Gamma \Vdash_\Sigma \Delta \vdash s : \tau$ may sometimes be written to express '$\Gamma \Vdash_\Sigma \Delta \vdash s : \tau$ is derivable'.

The following are examples of derivable typing judgements:

Example 4.17 (Derivable Typing Judgements).

$$a\colon \alpha,\ X\colon \beta \Vdash_{\varnothing} \varnothing \vdash (a,\ X)\colon (\alpha \times \beta) \tag{1}$$
$$\varnothing \Vdash_{\varnothing} \varnothing \vdash [a]\, a\colon [\alpha]\, \alpha \tag{2}$$
$$a\colon \beta \Vdash_{\varnothing} \varnothing \vdash [a]\, a\colon [\alpha]\, \alpha \tag{3}$$
$$\varnothing \Vdash_{\varnothing} \varnothing \vdash [a]\, a\colon [\gamma]\, \gamma \tag{4}$$
$$a\colon \alpha,\ b\colon \beta \Vdash_{\varnothing} \varnothing \vdash ([a]\, [b]\, [b]\, (a,\ b),\ a,\ b)\colon ([\alpha']\, [\beta']\, [\gamma]\, (\alpha' \times \gamma) \times \alpha \times \beta) \tag{5}$$
$$a\colon \tau_1,\ b\colon \tau_2,\ X\colon \tau \Vdash_{\varnothing} \varnothing \vdash (a\ b) \cdot X\colon \tau \tag{6}$$
$$a\colon \tau_1,\ b\colon \tau_1,\ X\colon \tau \Vdash_{\varnothing} \varnothing \vdash ((a\ b) \cdot X,\ \mathrm{id} \cdot X)\colon (\tau \times \tau) \tag{7}$$
$$X\colon \tau \Vdash_{\varnothing} a \# X \vdash ([a]\, \mathrm{id} \cdot X,\ \mathrm{id} \cdot X)\colon (\tau \times \tau) \tag{8}$$
$$a\colon \alpha,\ b\colon \beta,\ X\colon \tau \Vdash_{\varnothing} \varnothing \vdash [a]\, ((a\ b) \cdot X,\ \mathrm{id} \cdot X)\colon [\beta]\, (\tau \times \tau) \tag{9}$$

Notice that for the sixth term above, a type is still derivable, despite the incompatible types of the atoms occurring in the permutation, because there is only one occurrence of the variable, X, whereas the seventh, requires the types of the atoms, a and b, to be equal. For the eighth term, the freshness condition equates the conflicting environments of the two variable judgements, whilst for the ninth, the type association for a is updated in the environment to produce equal environments for the two variable judgements.

4.2.6 Properties of Derivable Typing Judgements. Lemma 4.18 is standard. Lemma 4.20 is a key property for this type system.

Lemma 4.18 (Instantiation of Typing Judgements). *If* $\Gamma \Vdash_{\Sigma} \Delta \vdash t\colon \tau$ *then* $\Gamma\, S \Vdash_{\Sigma} \Delta \vdash t\colon \tau\, S$.

Proof. By induction on the derivation of the corresponding quasi-typing judgement. Again, a similar result is shown in [27]. □

Lemma 4.19 (Meta-level Equivariance of Typing Judgements). *Given a permutation,* π, *if* $\Gamma \Vdash_{\Sigma} \Delta \vdash t\colon \tau$, *then* ${}^{\pi}\Gamma \Vdash_{\Sigma} {}^{\pi}\Delta \vdash {}^{\pi}t\colon \tau$.

Proof. By induction on the derivation of the quasi-typing judgement $\Gamma \Vdash\!\!\!\Vdash_{\Sigma} \Delta \vdash t\colon \tau$.

- For an atom term, a, if $\Gamma \Vdash\!\!\!\Vdash_{\Sigma} \Delta \vdash a\colon \tau$, then $\Gamma_a = \tau$ follows, as does ${}^{\pi}\Gamma_{\pi a} = \tau$ and thus ${}^{\pi}\Gamma \Vdash_{\Sigma} {}^{\pi}\Delta \vdash {}^{\pi}a\colon \tau$.
- For a moderated variable, $\pi' \cdot X$, if $\Gamma \Vdash\!\!\!\Vdash_{\Sigma} \Delta \vdash \pi' \cdot X\colon \tau$, then $\Gamma_X = {}^{\pi}\Gamma_X = \tau$ follows, and thus ${}^{\pi}\Gamma \Vdash_{\Sigma} {}^{\pi}\Delta \vdash {}^{\pi}(\pi' \cdot X)\colon \tau$.
- For a function application, $\mathrm{f}\, t$, if $\Gamma \Vdash_{\Sigma} \Delta \vdash \mathrm{f}\, t\colon \tau$, then by the induction hypothesis, ${}^{\pi}\Gamma \Vdash_{\Sigma} {}^{\pi}\Delta \vdash {}^{\pi}t\colon \tau'$ and because Σ is fixed under permutation, thus ${}^{\pi}\Gamma \Vdash_{\Sigma} {}^{\pi}\Delta \vdash {}^{\pi}(\mathrm{f}\, t)\colon \tau$.
- For a tuple, $(t_1,\ \ldots,\ t_n)$, each element follows from the inductive hypothesis but one must check that the compatibility property still holds. The difficulty is that a variable could occur in more than one t_i. Assume there are n occurrences of a variable X: $(\pi_1 \cdot X,\ \ldots,\ \pi_n \cdot X)$, one knows that the essential environments for each variable judgment $\Gamma_i \Vdash\!\!\!\Vdash_{\Sigma} \Delta \vdash \pi_i \cdot X\colon \tau_i$ are equal, but now one must prove that the essential environments of ${}^{\pi}\Gamma_i \Vdash\!\!\!\Vdash_{\Sigma} {}^{\pi}\Delta \vdash {}^{\pi}(\pi_i \cdot X)\colon \tau_i$ are also equal.
  Observe that ${}^{\pi}(\pi_i \cdot X) = \pi \mathbin{@} \pi_i \mathbin{@} \pi^{-1} \cdot X$ and thus, if ${}^{\pi_i^{-1}}\Gamma_i \setminus \{(a\colon \sigma) \mid \Delta \vdash a \# X\}$ are equal, then, because ${}^{(\pi @ \pi_i @ \pi^{-1})^{-1} @ \pi}\Gamma_i = {}^{\pi @ \pi_i^{-1} @ \pi^{-1} @ \pi}\Gamma_i = {}^{\pi @ \pi_i^{-1}}\Gamma_i$, it follows that ${}^{\pi @ \pi_i^{-1}}\Gamma_i \setminus \{(a\colon \sigma) \mid {}^{\pi}\Delta \vdash a \# X\}$ are all equal.
- For an abstraction, $[a]\, t$, if $\Gamma \Vdash\!\!\!\Vdash_{\Sigma} \Delta \vdash [a]\, t\colon \tau$, then $\tau \equiv [\tau']\, \tau''$ and $\Gamma \bowtie (a\colon \tau') \Vdash\!\!\!\Vdash_{\Sigma} \Delta \vdash t\colon \tau''$. By the induction hypothesis, ${}^{\pi}(\Gamma \bowtie (a\colon \tau')) \Vdash\!\!\!\Vdash_{\Sigma} {}^{\pi}\Delta \vdash {}^{\pi}t\colon \tau''$. Now, ${}^{\pi}(\Gamma \bowtie (a\colon \tau')) \equiv$

${}^{\pi}\Gamma \bowtie ({}^{\pi}a\colon \tau')$ and so ${}^{\pi}\Gamma \bowtie ({}^{\pi}a\colon \tau') \Vdash_{\Sigma} {}^{\pi}\Delta \vdash {}^{\pi}t\colon \tau''$. One can now infer that ${}^{\pi}\Gamma \Vdash_{\Sigma} {}^{\pi}\Delta \vdash [{}^{\pi}a]\,{}^{\pi}t\colon [\tau']\,\tau''$ and so ${}^{\pi}\Gamma \Vdash_{\Sigma} {}^{\pi}\Delta \vdash {}^{\pi}[a]\,t\colon \tau$.

□

Lemma 4.20 (Object-level Equivariance of Typing Judgements). *Given a permutation, π, if $\Gamma \Vdash_{\Sigma} \Delta \vdash t\colon \tau$ then ${}^{\pi}\Gamma \Vdash_{\Sigma} \Delta \vdash \pi \cdot t\colon \tau$.*

Proof. Proceed by induction on the derivation. The cases for an atom term, a moderated variable and a function application proceed as in the proof of Lemma 4.19 for meta-level equivariance.

- For a tuple, $(t_1, \ldots, t_n)$, $\Gamma \Vdash_{\Sigma} \Delta \vdash (t_1, \ldots, t_n)\colon \tau$, again each element follows by the induction hypothesis and one must check that the compatibility property continues to hold. Similarly, in the case where there are n occurrences of a variable X, $(\pi_1 \cdot X, \ldots, \pi_n \cdot X)$, one knows that the essential environments for each variable judgment $\Gamma_i \Vdash_{\Sigma} \Delta \vdash \pi_i \cdot X\colon \tau_i$ are equal, but now one must prove that the essential environments of ${}^{\pi}\Gamma_i \Vdash_{\Sigma} \Delta \vdash \pi \cdot (\pi_i \cdot X)\colon \tau_i$ are equal, which follows from the fact that ${}^{(\pi @ \pi_i)^{-1} @ \pi}\Gamma_i = {}^{\pi_i^{-1} @ \pi^{-1} @ \pi}\Gamma_i = {}^{\pi_i^{-1}}\Gamma_i$.
- For an abstraction, $[a]\,t$, if $\Gamma \Vdash_{\Sigma} \Delta \vdash [a]\,t\colon \tau$, then $\tau \equiv [\tau']\,\tau''$ and $\Gamma \bowtie (a\colon \tau') \Vdash_{\Sigma} \Delta \vdash t\colon \tau''$. By the induction hypothesis, ${}^{\pi}(\Gamma \bowtie (a\colon \tau')) \Vdash_{\Sigma} \Delta \vdash \pi \cdot t\colon \tau''$. One still knows that ${}^{\pi}(\Gamma \bowtie (a\colon \tau')) \equiv {}^{\pi}\Gamma \bowtie ({}^{\pi}a\colon \tau')$ but now ${}^{\pi}\Gamma \bowtie ({}^{\pi}a\colon \tau') \Vdash_{\Sigma} \Delta \vdash \pi \cdot t\colon \tau''$ and ${}^{\pi}\Gamma \bowtie (\pi \cdot a\colon \tau') \Vdash_{\Sigma} \Delta \vdash \pi \cdot t\colon \tau''$ follows from the fact that ${}^{\pi}a = \pi(a) = \pi \cdot a$. One can now infer that ${}^{\pi}\Gamma \Vdash_{\Sigma} \Delta \vdash [\pi \cdot a]\,\pi \cdot t\colon [\tau']\,\tau''$ and so ${}^{\pi}\Gamma \Vdash_{\Sigma} \Delta \vdash \pi \cdot [a]\,t\colon \tau$.

□

The notion of a well-typed substitution is now defined, for which in Theorem 4.23 it is proved that the types of terms are preserved under their action. More precisely, given a typing judgement for a term t, conditions are given that typeable substitutions should satisfy to ensure that the corresponding instance of the term t has the same type as t.

This property is important for rewriting applications, specifically, to ensure that rewriting (using well-typed rules) preserves types. This is not straightforward in a polymorphic Curry-style system due to the fact that atoms, for which types are provided in the environment, may be captured when the substitution is applied.

Below, for simplicity, consider only idempotent substitutions that instantiate all the variables in a term. This is a standard assumption that is always satisfied by the substitutions generated in rewriting steps, because the variables of rewrite rules are taken to be disjoint from those variables of the term to be rewritten.

Definition 4.21 (Well-typed Substitution). An idempotent substitution θ, where $\mathsf{dom}(\theta) \subseteq \mathsf{vars}(t)$, is well-typed in Γ, Σ and Δ for $\Phi \Vdash_{\Sigma} \nabla \vdash t\colon \tau$ under the following conditions.

(1) $\Delta \vdash \nabla\,\theta$
(2) $\Gamma = \Gamma' \bowtie \Phi_{atm}$, where Φ_{atm} denotes the set of declarations for atoms provided in Φ. Thus, Γ contains the associations for atoms provided by Φ and possibly also declarations for other atoms and variables from the image of θ.
(3) For each $X \in \mathsf{vars}(t)$, where E is the essential environment of X in $\Phi \Vdash_{\Sigma} \nabla \vdash t\colon \tau$:
  (a) $\Gamma' \bowtie E_{atm} \Vdash_{\Sigma} \Delta \vdash X\,\theta\colon \Phi_X$, where $E = E_{var} \cup E_{atm}$ such that E_{var} contains associations for variables and E_{atm} associations for atoms;
  (b) $\mathsf{supp}(\pi) \cap (\mathsf{atms}(\Gamma') - \mathsf{atms}(E)) = \varnothing$, for every π such that $\pi \cdot X$ occurs in t.
(4) The set of type derivations $\{\Gamma' \bowtie E_{atm} \Vdash_{\Sigma} \Delta \vdash X\,\theta\colon \Phi_X \mid X \in \mathsf{vars}(t)\}$ has the compatibility property, that is, for each variable in the image of θ, the essential environments for that variable throughout all the derivations in the set are equal.

Condition 2 ensures that Γ types atoms in the same way as Φ: for any atom a, $\Gamma_a = \Phi_a$. This definition is more involved than Definition 3.24: Condition 1 is needed here due to the fact that freshness constraints are taken into account in essential environments in this system, and Condition 3b is not stated in Definition 3.24 because it holds trivially, since all the atoms in $\mathsf{supp}(\pi)$ must be typed in Φ (permutations are well-typed by construction in Church-style systems).

For example, consider part 6 in Example 4.17, and assume $\tau \equiv \tau_1 \equiv \mathtt{Nat}$ and $\tau_2 \equiv \mathtt{Bool}$. The substitution $[X \mapsto a]$ is not well-typed; but $[X \mapsto b]$ is, where $\Gamma' = \varnothing$ and $E = \{b\colon \mathtt{Nat},\ a\colon \mathtt{Bool},\ X\colon \mathtt{Nat}\}$.

It is now proved that well-typed substitutions preserve types. The following lemma must first be stated and proved.

Lemma 4.22 (Type Weakening and Strengthening).

- *If* $\Gamma \Vdash_\Sigma \Delta \vdash t\colon \tau$ *then* $\Gamma \bowtie (a\colon \tau') \Vdash_\Sigma \Delta \vdash t\colon \tau$, *provided that either* $a \notin \mathsf{atms}(t)$ *or* $\Delta \vdash a \# t$. *Call this* ***type weakening****.*
- *If* $\Gamma \Vdash_\Sigma \Delta \vdash t\colon \tau$ *and* $(a\colon \tau') \in \Gamma$ *then* $\Gamma \setminus \{(a\colon \tau')\} \Vdash_\Sigma \Delta \vdash t\colon \tau$ *provided that either* $a \notin \mathsf{atms}(t)$ *or* $\Delta \vdash a \# t$. *Call this* ***type strengthening****.*

Proof. By induction on the typing judgement derivation. Note that, if $a \in \mathsf{atms}(t)$, then, because $\Delta \vdash a \# t$, a must either be abstracted and $(a\colon \tau)$ is updated or occur in a suspended permutation in which case $(a\colon \tau)$ is removed from the essential environment. □

Theorem 4.23 (Well-typed Substitutions Preserve Types). *If* θ *is well-typed in* Γ, Σ *and* Δ *for* $\Phi \Vdash_\Sigma \nabla \vdash t\colon \tau$, *then* $\Gamma \Vdash_\Sigma \Delta \vdash t\,\theta\colon \tau$.

Proof. By induction on the structure of the derivation of the quasi-typing judgement $\Phi \Vdash\!\!\!\!\!\!\sim_\Sigma \nabla \vdash t\colon \tau$, distinguishing cases according to the last rule applied.

- For an atom term, a, if $\Phi \Vdash\!\!\!\!\!\!\sim_\Sigma \nabla \vdash a\colon \tau$, then $\Phi_a = \tau$ follows as does $a\,\theta = a$, and $\Gamma_a = \Phi_a$ by Condition 2 of Definition 4.21 since θ is well-typed in Γ. Thus $\Gamma \Vdash_\Sigma \Delta \vdash a\,\theta\colon \tau$.
- For a moderated variable, $\pi \cdot X$, if $\Phi \Vdash\!\!\!\!\!\!\sim_\Sigma \nabla \vdash \pi \cdot X\colon \tau$, then $\Phi_X = \tau$ and the essential environment of X is $E = {}^{\pi^{-1}}\Phi \setminus \{a\colon \sigma \mid \nabla \vdash a \# X\}$. Due to the fact that θ is well-typed, $\Gamma' \bowtie E_{atm} \Vdash_\Sigma \Delta \vdash \theta(X)\colon \Phi_X$ by Condition 3a of Definition 4.21. Hence, by Lemma 4.20, ${}^{\pi}(\Gamma' \bowtie E_{atm}) \Vdash_\Sigma \Delta \vdash \pi \cdot X\,\theta\colon \Phi_X$. Since ${}^{\pi}E_{atm} \subseteq \Phi_{atm}$ and $\Delta \vdash a \# X\theta$ for any $a \# X \in \nabla$ by Condition 1 of Definition 4.21, by type weakening one obtains ${}^{\pi}\Gamma' \bowtie \Phi_{atm} \Vdash_\Sigma \Delta \vdash \pi \cdot X\,\theta\colon \Phi_X$. By Condition 3b of Definition 4.21, it also follows that $\Gamma' \bowtie \Phi_{atm} \Vdash_\Sigma \Delta \vdash \pi \cdot X\,\theta\colon \Phi_X$, which completes the proof of this case because $\Gamma = \Gamma' \bowtie \Phi_{atm}$ by Condition 2 of Definition 4.21.
- For an application term, the result follows directly by the induction hypothesis.
- For a tuple, $(t_1, \ldots, t_n)$, each element follows from the inductive hypothesis but one must check that the compatibility property still holds. This is a consequence of part 4 in Definition 4.21.
- For an abstraction, $[a]\,t'$, if $\Phi \Vdash\!\!\!\!\!\!\sim_\Sigma \nabla \vdash [a]\,t'\colon \tau$, then $\tau \equiv [\tau']\,\tau''$ and $\Phi \bowtie (a\colon \tau') \Vdash\!\!\!\!\!\!\sim_\Sigma \nabla \vdash t'\colon \tau''$. Note that θ is well-typed in $\Gamma \bowtie (a\colon \tau')$ and Δ for $\Phi \bowtie (a\colon \tau') \Vdash_\Sigma \nabla \vdash t'\colon \tau''$ because t' has the same set of variables and essential environments as t. The result then follows directly from the inductive hypothesis.

□

4.2.7 Types and Alpha-equivalence. It is now proved that alpha-equivalence respects types, or in other words, that the type system correctly deals with alpha-equivalence classes of terms, in the sense that two alpha-equivalent terms have the same types. Most importantly, this requires that the defined static semantics for derivable typing judgements correctly handles the action of suspended permutations and also the possibility that the application of a variable substitution may result in the capture of atoms by abstractions.

Theorem 4.24 (Types and Alpha-equivalence). *If* $\Delta \vdash s \approx_\alpha t$ *and* $\Gamma \Vdash_\Sigma \Delta \vdash s : \tau$ *then* $\Gamma \Vdash_\Sigma \Delta \vdash t : \tau$.

Proof. By induction on the size of the pair, $(s,\ t)$. The form of $(s,\ t)$ is rather restricted by the assumption that $\Delta \vdash s \approx_\alpha t$; for example, if $s \equiv \pi \cdot X$ then $t \equiv \pi' \cdot X$ for some permutation, π'. This information is used without comment in the proof.

- If $\Delta \vdash a \approx_\alpha a$ and $\Gamma \Vdash_\Sigma \Delta \vdash a : \tau$, then there is nothing to prove.
- If $\Delta \vdash \pi \cdot X \approx_\alpha \pi' \cdot X$ and $\Gamma \Vdash_\Sigma \Delta \vdash \pi \cdot X : \tau$, then $\Gamma_X = \tau$ and it follows that $\Gamma \Vdash_\Sigma \Delta \vdash \pi' \cdot X : \tau$.
- If $\Delta \vdash \mathsf{f}\, s \approx_\alpha \mathsf{f}\, t$ and $\Gamma \Vdash_\Sigma \Delta \vdash \mathsf{f}\, s : \tau$, then $\Gamma \Vdash_\Sigma \Delta \vdash s : \tau'$, and by the inductive hypothesis, one deduces that $\Gamma \Vdash_\Sigma \Delta \vdash t : \tau'$ and concludes that $\Gamma \Vdash_\Sigma \Delta \vdash \mathsf{f}\, t : \tau$.
- If $\Delta \vdash (s_1,\ \ldots,\ s_n) \approx_\alpha (t_1,\ \ldots,\ t_n)$ and $\Gamma \Vdash_\Sigma \Delta \vdash (s_1,\ \ldots,\ s_n) : \tau$, then each element again follows by the inductive hypothesis but one must check the compatibility property still holds. Assume there are m occurrences of X, $\pi_1 \cdot X,\ \ldots,\ \pi_m \cdot X$, in s. Since t is alpha-equivalent to s, there are also m occurrences $\pi_1' \cdot X,\ \ldots,\ \pi_m' \cdot X$ in t, at the same positions. Now, $\pi_i \cdot X \approx_\alpha \pi_i' \cdot X$ when $\Delta \vdash \mathrm{ds}(\pi_i,\ \pi_i')\ \#\ X$, and so if the essential environments of each variable judgement in s are equal, then they must also be equal for t.
- If $\Delta \vdash [a]\, s \approx_\alpha [a]\, t$ and $\Gamma \Vdash_\Sigma \Delta \vdash [a]\, s : \tau$, then $\Delta \vdash s \approx_\alpha t$ and $\tau \equiv [\tau']\, \tau''$ and $\Gamma \bowtie (a : \tau') \Vdash_\Sigma \Delta \vdash s : \tau''$. Use the inductive hypothesis to deduce that $\Gamma \bowtie (a : \tau') \Vdash_\Sigma \Delta \vdash t : \tau''$ and conclude that $\Gamma \Vdash_\Sigma \Delta \vdash [a]\, t : [\tau']\, \tau''$.
- If $\Delta \vdash [a]\, s \approx_\alpha [b]\, t$ and $\Gamma \Vdash_\Sigma \Delta \vdash [a]\, s : \tau$, then again, $\tau \equiv [\tau']\, \tau''$ and $\Gamma \bowtie (a : \tau') \Vdash_\Sigma \Delta \vdash s : \tau''$ and by the properties of $\approx_\alpha$ [69, theorem 2.11], $\Delta \vdash \left\{ s \approx_\alpha (a\ b) \cdot t,\ a \# t,\ (a\ b) \cdot s \approx_\alpha t,\ b \# s \right\}$.
  By type weakening, Lemma 4.22, $\Gamma \bowtie \{(a : \tau'),\ (b : \tau')\} \Vdash_\Sigma \Delta \vdash s : \tau''$ and by object-level equivariance, Lemma 4.20, $\Gamma \bowtie \{(a : \tau'),\ (b : \tau')\} \Vdash_\Sigma \Delta \vdash (a\ b) \cdot s : \tau''$.
  From the fact that $\Delta \vdash (a\ b) \cdot s \approx_\alpha t$, deduce by the inductive hypothesis that $\Gamma \bowtie \{(a : \tau'),\ (b : \tau')\} \Vdash_\Sigma \Delta \vdash t : \tau''$ and then that $\Gamma \bowtie \{(a : \tau')\} \Vdash_\Sigma \Delta \vdash [b]\, t : [\tau']\, \tau''$.
  From the fact that $\Delta \vdash a\#[b]\, t$, conclude by type strengthening, Lemma 4.22, that $\Gamma \Vdash_\Sigma \Delta \vdash [b]\, t : [\tau']\, \tau''$.

□

It is to be noted that Theorem 4.24 is stronger than its counterpart in the Church system (see Theorem 3.29) in the following sense: in a Church system, all terms are typed by construction, so to prove that the type system is compatible with the alpha-equivalence relation it need only be shown that alpha-equivalent terms have the same type. In contrast, here it is proved that if one term of an alpha-equivalence class may be given a type, all terms of that class may be given the same type. For this reason, typing judgements in the Curry system also take into consideration the freshness context in which the term exists.

4.3 Inference of Principal Solutions

This subsection first presents the notions of a term typeability problem and a principal solution to such a problem. There then follows a description of an algorithm for the computation of principal solutions and a proof that this algorithm is sound and complete with respect to derivable typing judgements within the system.

4.3.1 *Typeability Problems.*

Definition 4.25 (Typeability Problem). A **typeability problem** is a tuple $(\Gamma,\ \Sigma,\ \Delta,\ t)$, written, $\Gamma \Vdash_\Sigma \Delta \vdash t$, which asks whether the term, t, is typeable using the type and declaration associations

given in the environment, Γ, and signature, Σ, respectively, and the freshness constraints given in the freshness context, Δ.

Given a typeability problem, $\Gamma \Vdash_\Sigma \Delta \vdash t$, if there exists a type, τ, such that $\Gamma \Vdash_\Sigma \Delta \vdash t : \tau$ is derivable, then say that $\Gamma \Vdash_\Sigma \Delta \vdash t$ is **typeable**. Otherwise, say that $\Gamma \Vdash_\Sigma \Delta \vdash t$ is **untypeable**, or say that it is **quasi-typeable** if a derivation for the quasi-typing judgement $\Gamma \Vdash\!\!\Vdash_\Sigma \Delta \vdash t : \tau$ exists for some type τ.

More generally, say that a typeability problem is **solvable** if there exists a type substitution, S, that when applied to the environment, Γ, allows a type, τ, to be derived for the term, t. This is stated formally in Definition 4.26.

Definition 4.26 (Typeability Problem (Quasi-)Solution). A quasi-solution to a typeability problem, $\Gamma \Vdash_\Sigma \Delta \vdash t$, is a pair, (S, τ), consisting of a type substitution, S, and a type, τ, such that the quasi-typing judgement, $\Gamma\, S \Vdash\!\!\Vdash_\Sigma \Delta \vdash t : \tau$, is derivable. If, moreover, the compatibility property holds, then (S, τ) is a solution.

Note that a typeability problem may have a solution but still not be *typeable*. A problem is only typeable when there exists a solution of the form (id, τ).

Example 4.27 (Typeability Problem Solutions). The typeability problem, $a : \mathtt{Bool}$, $b : \mathtt{Bool}$, $X : \alpha \Vdash_\varnothing \varnothing \vdash (a\ b) \cdot X$, has the solutions (id, α) and $([\alpha \mapsto \mathtt{Nat}], \mathtt{Nat})$ amongst others, and is both solvable and typeable.

4.3.2 Principal Solutions and Principal Types. Solutions and quasi-solutions have a natural ordering given by the instantiation of substitutions.

$$(S, \sigma) \leq (T, \tau) \quad \text{when} \quad \exists U.\, \{T = S \circ U \ \wedge\ \tau \equiv \sigma\, U\}$$

A most general (quasi-)solution to a problem is a minimal element in a set of (quasi-)solutions; it is called a **principal (quasi-)solution**. The type component of a principal (quasi-)solution for a given typing problem is a most general (quasi-)type or **principal (quasi-)type** for that problem.

By these definitions, there may be many principal solutions and thus many principal types for a given typeability problem. For example, if one lets Σ be the signature, $\{\mathtt{f} : \forall(\alpha).\langle \mathtt{Nat} \hookrightarrow \alpha \rangle\}$, then both (id, α') and (id, β) are principal solutions to the typeability problem, $X : \mathtt{Nat} \Vdash_\Sigma \varnothing \vdash \mathtt{f}\, X$. However, principal solutions for a solvable typeability problem, $\Gamma \Vdash_\Sigma \Delta \vdash s$, are unique modulo the names of those type variables that are not elements of the set $\mathrm{tvars}(\Gamma)$. In fact, principal types can be viewed as principal schemes, as in the type system described in [27] for the language ML.

4.4 Inference of Principal Types

A partial function, pt, is now presented that given a typeability problem, $\Gamma \Vdash_\Sigma \Delta \vdash t$, computes its principal solution if there is one, and fails if there is no solution.

It is defined in two phases: firstly, given a typeability problem $\Gamma \Vdash_\Sigma \Delta \vdash t$, the function ptQ constructs a most general derivation for $\Gamma \Vdash\!\!\Vdash_\Sigma \Delta \vdash t : \tau$ and then, the function ptE checks the compatibility property for the essential environments of this derivation and outputs a solution, if one exists.

4.4.1 ptQ. Phase ptQ, takes as input a typeability problem and outputs a typeability problem quasi-solution, if one exists, together with the derivation tree constructed for that quasi-solution.

During the computation of ptQ, it is necessary to preserve the full derivation tree constructed in order to compute ptE. The preservation of this tree is verbose and relatively straightforward and is omitted in the outline of the algorithm for ptQ given below, in order to maintain clarity.

The only point of interest is that one must remember to apply the type substitution output to the preserved derivation tree, because typing function applications may instantiate previously computed sub-trees, due to the computation of most general unifiers over types returned in those sub-trees.

Definition 4.28 (Algorithm for ptQ*).* Let $S\,|_\Gamma$ denote the restriction of the substitution S to the variables in Γ.

- $\mathsf{ptQ}(\Gamma \Vdash_\Sigma \Delta \vdash a) = (\mathsf{id},\, \tau)$ where $\Gamma_a = \tau$.
- $\mathsf{ptQ}(\Gamma \Vdash_\Sigma \Delta \vdash \pi \cdot X) = (\mathsf{id},\, \tau)$ where $\Gamma_X = \tau$.
- $\mathsf{ptQ}(\Gamma \Vdash_\Sigma \Delta \vdash [a]\, s) = (S\,|_\Gamma,\, [\tau']\, \tau)$ where $\mathsf{ptQ}(\Gamma \bowtie (a\colon \alpha) \Vdash_\Sigma \Delta \vdash s) = (S,\, \tau)$, the type variable, α, is newly generated and $\tau' = \alpha\, S$.
- $\mathsf{ptQ}(\Gamma \Vdash_\Sigma \Delta \vdash \mathsf{f}\, t) = (SS'\,|_\Gamma,\, \phi\, S')$ where $\mathsf{ptQ}(\Gamma \Vdash_\Sigma \Delta \vdash t) = (S,\, \tau)$, $\Sigma_\mathsf{f} = \rho$ and $\langle \phi' \hookrightarrow \phi \rangle$ is the most general generic instance of ρ, where every type variable, $\alpha \in \mathsf{tvars}(\langle \phi' \hookrightarrow \phi \rangle)$, is newly generated and $S' = \mathsf{mgu}(\phi',\, \tau)$.
- $\mathsf{ptQ}(\Gamma \Vdash_\Sigma \Delta \vdash (t_1,\, \ldots,\, t_n)) = (S_1 \ldots S_n,\, (\tau_1 S_2 \ldots S_n \times \ldots \times \tau_{n-1} S_n \times \tau_n))$ where $\mathsf{ptQ}(\Gamma \Vdash_\Sigma \Delta \vdash t_1) = (S_1,\, \tau_1)$, $\mathsf{ptQ}(\Gamma\, S_1 \Vdash_\Sigma \Delta \vdash t_2) = (S_2,\, \tau_2)$, …, $\mathsf{ptQ}(\Gamma\, S_1 \ldots S_{n-1} \Vdash_\Sigma \Delta \vdash t_n) = (S_n,\, \tau_n)$.

4.4.2 ptE. Phase ptE takes the typeability problem quasi-solution and derivation output by ptQ and, if defined, returns a typeability problem solution, more specific than or equal to the input solution.

The functions to determine variable judgements within a derivation tree and to compute the essential environments that then arise are straightforward.

The function to compute the most general unifier of two types, $\mathsf{mgu}(\sigma,\, \tau)$, is extended to sets of pairs of types in the obvious way, using the transitivity of equality. The most general unifier of two environments, Γ and Υ, is defined such that, for all $a \in \Gamma \cup \Upsilon$, both Γ_a and Υ_a must be defined, and that set of pairs of types be unifiable by mgu. The notion is extended further to sets of environments in the obvious way.

The algorithm for ptE is easily defined using these subcomponents. Given the derivation for the typeability problem, $\Gamma \Vdash_\Sigma \Delta \vdash t$, a most general unifier for the essential environments for each variable $X \in \mathsf{vars}(t)$, is computed and, if defined, this unifier is composed with the substitution component and applied to the type component of the input solution from ptQ to produce the solution to the initial typeability problem. Thus the function pt is defined simply as $\mathsf{pt}(\Gamma \Vdash_\Sigma \Delta \vdash t) = (S \circ S',\, \tau\, S')$ where $\mathsf{ptQ}(\Gamma \Vdash_\Sigma \Delta \vdash t) = (S,\, \tau)$ and $\mathsf{ptE}(\Gamma \Vdash_\Sigma \Delta \vdash t) = S'$.

4.4.3 Soundness and Completeness of pt. Given a typeability problem, P, it must now be shown that $\mathsf{pt}(P)$ is a solution to P, and that every solution to P is an instance of $\mathsf{pt}(P)$. This is now stated and proved formally.

Lemma 4.29 is required for the proof.

Lemma 4.29 (Newly-generated Environments and pt). *If* $\mathsf{pt}(\Gamma \bowtie (a\colon \tau) \Vdash_\Sigma \Delta \vdash t) = (S,\, \sigma)$ *and* α *is a newly-generated type variable with respect to* $\Gamma \bowtie (a\colon \tau)$*, then there exists a type substitution,* T*, and a type,* τ'*, such that* $\mathsf{pt}(\Gamma \bowtie (a\colon \alpha) \Vdash_\Sigma \Delta \vdash t) = (T,\, \tau')$ *and* $(T,\, \tau') \leq (S,\, \sigma)$.

Proof. By induction on the term, t. The only interesting case is that for an atom term, a.

There, $\mathsf{pt}(\Gamma \bowtie (a\colon \tau) \Vdash_\Sigma \Delta \vdash a) = (\mathsf{id},\, \tau)$ and so it follows that $\mathsf{pt}(\Gamma \bowtie (a\colon \alpha) \Vdash_\Sigma \Delta \vdash a) = (\mathsf{id},\, \alpha)$ and, because α is newly-generated, $(\mathsf{id},\, \alpha) \leq (\mathsf{id},\, \tau)$. □

Theorem 4.30 (Soundness and Completeness of pt).

- *Soundness: if* $\mathsf{pt}(\Gamma \Vdash_\Sigma \Delta \vdash t) = (S,\, \tau)$ *then* $\Gamma\, S \Vdash_\Sigma \Delta \vdash t\colon \tau$ *is derivable.*

- *Completeness: let U be a substitution such that* $\mathrm{dom}(U) \subseteq \mathrm{tvars}(\Gamma)$*; if* $\Gamma\, U \Vdash_\Sigma \Delta \vdash t : \sigma$ *is derivable then* $\mathrm{pt}(\Gamma \Vdash_\Sigma \Delta \vdash t)$ *is defined and equal to* (S, τ) *and* $(S, \tau) \leq (U, \sigma)$.

Proof. The argument for soundness is by a routine induction on the definition of $\mathrm{pt}(\Gamma \Vdash_\Sigma \Delta \vdash t) = (S, \tau)$, using Lemma 4.18 and is omitted.

That for completeness is divided into two parts, corresponding to the two phases of pt. The first part is thus proving completeness of ptQ for quasi-solutions, and is by induction on the abstract syntax tree of the term, t, with the help of Lemma 4.29. Distinguish the following cases:

- If $\Gamma\, U \Vdash_\Sigma \Delta \vdash a : \sigma$ then $\sigma = (\Gamma\, U)_a = \Gamma_a\, U$. Let $\Gamma_a = \tau$, then by definition $\mathrm{ptQ}(\Gamma \Vdash_\Sigma \Delta \vdash a) = (\mathrm{id}, \tau)$, which is clearly the principal solution.
- The case for a moderated variable follows that for an atom term.
- If $\Gamma\, U \Vdash_\Sigma \Delta \vdash [a]\, t : \sigma$, then $\sigma \equiv [\sigma']\, \sigma''$, and $\Gamma\, U \bowtie (a : \sigma') \Vdash_\Sigma \Delta \vdash t : \sigma''$.
  By the inductive hypothesis, $\mathrm{ptQ}(\Gamma \bowtie (a : \sigma') \Vdash_\Sigma \Delta \vdash t) = (S, \tau'')$, of which, (U, σ''), is an instance; that is, there is a substitution, T, such that $U = S \circ T$ and $\sigma'' \equiv \tau''\, T$.
  Now also, by definition, $\mathrm{ptQ}(\Gamma \Vdash_\Sigma \Delta \vdash [a]\, t) = (V|_\Gamma, [\mu']\, \mu'')$, where $\mu' = \alpha\, V$ and $\mathrm{ptQ}(\Gamma \bowtie (a : \alpha) \Vdash_\Sigma \Delta \vdash t) = (V, \mu'')$, thus, by Lemma 4.29, there exists a substitution, W, such that, $V \circ W = S$ and $\mu''\, W \equiv \tau''$.
  However, because α is newly generated, it follows that for all $\beta \in \mathrm{tvars}(\Gamma)$ then $\beta\, V|_\Gamma \equiv \beta\, S$. The fact that $\alpha \notin \mathrm{dom}(S)$ gives $\mu' \equiv \alpha\, V \equiv \sigma'$ and $\alpha \notin \mathrm{tvars}(\mu'')$, gives $\mu'' \equiv \tau''$. Thus, $(U, [\sigma']\, \sigma'')$ is an instance of $(V|_\Gamma, [\mu']\, \mu'')$.
- If $\Gamma\, U \Vdash_\Sigma \Delta \vdash \mathsf{f}\, t : \sigma$, then $\Gamma\, U \Vdash_\Sigma \Delta \vdash t : \sigma'$, $\Sigma_\mathsf{f} = \rho \equiv \langle \rho' \hookrightarrow \rho'' \rangle$ and $\langle \rho' \hookrightarrow \rho'' \rangle \succcurlyeq \langle \sigma' \hookrightarrow \sigma \rangle$.
  By the inductive hypothesis, $\mathrm{ptQ}(\Gamma \Vdash_\Sigma \Delta \vdash t) = (S, \tau')$, of which, (U, σ'), is an instance; that is, there is a substitution, T, such that $U = S\, T$, $\sigma' \equiv \tau'\, T$.
  Thus by definition, $\mathrm{ptQ}(\Gamma \Vdash_\Sigma \Delta \vdash \mathsf{f}\, t) = ((S \circ V)|_\Gamma, \phi''\, V)$, where $\langle \phi' \hookrightarrow \phi'' \rangle$, is a newly generated most general generic instance of ρ, and $V = \mathrm{mgu}(\phi', \tau')$.
  Now, because ϕ' is part of the most general generic instance of ρ, there exists a substitution W, such that $\phi'\, W \equiv \sigma' \equiv \tau'\, T$ but $\tau'\, V \equiv \phi'\, V$, thus $V \leq W \circ T$, otherwise V would not be the mgu of ϕ' and τ'.
- If $\Gamma\, U \Vdash_\Sigma \Delta \vdash (t_1, \ldots, t_n) : (\sigma_1 \times \ldots \times \sigma_n)$, then $\Gamma\, U \Vdash_\Sigma \Delta \vdash t_1 : \sigma_1, \ldots, \Gamma\, U \Vdash_\Sigma \Delta \vdash t_n : \sigma_n$.
  By definition:
  $\mathrm{ptQ}(\Gamma \Vdash_\Sigma \Delta \vdash (t_1, \ldots, t_n)) = (S_1 \ldots S_n, (\tau_1\, S_2 \ldots S_n \times \ldots \times \tau_{n-1}\, S_n \times \tau_n))$ where $\mathrm{ptQ}(\Gamma \Vdash_\Sigma \Delta \vdash t_1) = (S_1, \tau_1)$, $\mathrm{ptQ}(\Gamma\, S_1 \Vdash_\Sigma \Delta \vdash t_2) = (S_2, \tau_2)$, …, $\mathrm{ptQ}(\Gamma\, S_1 \ldots S_{n-1} \Vdash_\Sigma \Delta \vdash t_n) = (S_n, \tau_n)$.
  This is equivalent to computing, first, $\mathrm{ptQ}(\Gamma \Vdash_\Sigma \Delta \vdash t_1) = (S_1, \tau_1)$, …, $\mathrm{ptQ}(\Gamma \Vdash_\Sigma \Delta \vdash t_n) = (S_n, \tau_n)$, and then computing the most general unifier of these solutions.
  Element-wise each is provable by induction and $((t_1, \ldots, t_n), (\sigma_1 \times \ldots \times \sigma_n))$, must be an instance of the most general unifier of the solutions.

This completes the proof of completeness of ptQ with respect to quasi-solutions: if a typeability problem has a quasi-solution, then ptQ finds a principal one.

Now consider the second phase, ptE.

If $\Gamma\, U \Vdash_\Sigma \Delta \vdash t : \sigma$ is derivable then there exists a derivation for the quasi-typing judgement $\Gamma\, U \Vdash\!\!\!\Vdash_\Sigma \Delta \vdash t : \sigma$ in which the essential environments of the variable judgements are equal for each variable, $X \in \mathrm{vars}(t)$. Given the completeness of ptQ, shown immediately above, then $\mathrm{ptQ}(\Gamma \Vdash_\Sigma \Delta \vdash t)$ is defined and equal to (S, τ) and $(S, \tau) \leq (U, \sigma)$. By the fact that ptE computes a most general unifier and it is known that one exists, then $\mathrm{pt}(\Gamma \Vdash_\Sigma \Delta \vdash t)$ is defined and must remain more general than (U, σ). □

The following property follows as a corollary of Theorem 4.30.

Corollary 4.31 (Empty Type Substitution). *A typeability problem,* $\Gamma \Vdash_\Sigma \Delta \vdash t$*, is typeable, if and only if* $\mathsf{pt}(\Gamma \Vdash_\Sigma \Delta \vdash t) = (\mathsf{id}, \tau)$ *for some type,* τ*.*

The following property follows as a corollary of Theorem 4.30 and Theorem 4.24.

Corollary 4.32 (Alpha-equivalence and Principal Solutions). $\Delta \vdash s \approx_\alpha t$ *implies* $\mathsf{pt}(\Gamma \Vdash_\Sigma \Delta \vdash s) = \mathsf{pt}(\Gamma \Vdash_\Sigma \Delta \vdash t)$ *modulo the names of type variables, for any environment,* Γ*, and signature,* Σ*, such that* pt *is defined for either* s *or* t*.*

5 TYPED NOMINAL EQUATIONAL THEORIES

This section explores the application of the polymorphic type system of Section 4 to nominal equational theories.

A formalisation of typed nominal rewriting is given by first defining the notion of a typeable rewrite rule and then providing conditions upon nominal rewriting sufficient to guarantee the preservation of types. A simplification, using the more efficient approach of closed nominal rewriting, over typeable terms, is then investigated and a corresponding system of typed nominal algebra defined.

5.1 Typed Nominal Rewriting

This subsection begins by defining a typeable rewrite rule. This is followed by conditions upon nominal rewriting that ensure the property of subject reduction. These conditions are formulated in Section 5.1.2, as typed nominal pattern matching.

5.1.1 *Typeable Rewrite Rules.*

Definition 5.1 (Typeable Rewrite Rule). A **typeable rewrite rule**, written $\Phi \Vdash_\Sigma \nabla \vdash l \to r : \tau$, is a tuple of an environment, Φ, a signature, Σ, a freshness context, ∇, two terms, l and r and a type, τ, such that the following conditions hold.

(1) $\nabla \vdash l \to r$ is a uniform rule;
(2) $\mathsf{pt}(\Phi \Vdash_\Sigma \nabla \vdash l) = (\mathsf{id}, \tau)$ and $\Phi \Vdash_\Sigma \nabla \vdash (l, r) : (\tau \times \tau)$.

The first condition requires the rewrite rule to be uniform; intuitively, this means that reductions do not generate new atoms.

The second says that l and r are both typeable with the principal type of l in the context of ∇, Φ and Σ. Most importantly, this condition is so framed to ensure that the essential environments of both sides of the rule are the same, which is key to proving the property of subject reduction (that is, the rewriting relation generated by typeable rules preserves types).

Note that by the unicity of principal types, given ∇ and Φ there exists at most one τ such that $\Phi \Vdash_\Sigma \nabla \vdash l \to r : \tau$ is typeable.

Let Σ be the signature, $\{\mathsf{lam} : \forall(\alpha, \beta).\langle [\alpha]\,\beta \hookrightarrow \alpha \Rightarrow \beta\rangle,\ \mathsf{app} : \forall(\alpha, \beta).\langle(\alpha \Rightarrow \beta \times \alpha) \hookrightarrow \beta\rangle,\ \mathsf{sub} : \forall(\alpha, \beta).\langle([\alpha]\,\beta \times \alpha) \hookrightarrow \beta\rangle\}$, where $\Rightarrow$, is a type-former, written infix, to construct function types. One then has the following typeable rules representing $\beta\eta$-reduction for a polymorphically typed lambda calculus.

Example 5.2 (Typeable Rewrite Rules).

$$X\colon \alpha,\ Y\colon \beta \Vdash_{\Sigma} \varnothing \vdash \mathsf{app}\,((\mathsf{lam}\,[a]\,X),\ Y) \to \mathsf{sub}\,([a]\,X,\ Y)\colon \alpha$$

$$X\colon \alpha \Rightarrow \beta \Vdash_{\Sigma} a \# X \vdash \mathsf{lam}\,[a]\,(\mathsf{app}\,(X,\ a)) \to X\colon \alpha \Rightarrow \beta$$

$$X\colon \alpha,\ Z\colon \gamma \Vdash_{\Sigma} a \# X \vdash \mathsf{sub}\,([a]\,X,\ Z) \to X\colon \alpha$$

$$Z\colon \gamma \Vdash_{\Sigma} \varnothing \vdash \mathsf{sub}\,([a]\,a,\ Z) \to Z\colon \gamma$$

$$X\colon \beta \Rightarrow \alpha,\ Y\colon \beta,\ Z\colon \gamma \Vdash_{\Sigma} \varnothing \vdash \mathsf{sub}\,([a]\,(\mathsf{app}\,(X,\ Y)),\ Z) \to \mathsf{app}\,(\mathsf{sub}\,([a]\,X,\ Z),\ \mathsf{sub}\,([a]\,Y,\ Z))\colon \alpha$$

$$X\colon \alpha,\ Z\colon \gamma \Vdash_{\Sigma} b \# Z \vdash \mathsf{sub}\,([a]\,(\mathsf{lam}\,[b]\,X),\ Z) \to \mathsf{lam}\,[b]\,(\mathsf{sub}\,([a]\,X,\ Z))\colon \alpha' \Rightarrow \alpha$$

5.1.2 Typed Nominal Pattern Matching. Given a typeable rewrite rule and a typeable term for which a solution to an equivariant nominal pattern matching problem exists, it does not necessarily hold that their respective types are compatible (see Example 5.4 below).

Thus, this section introduces the concept of typed nominal pattern matching.

Typed nominal pattern matching, as defined below, is an equivariant matching.

Definition 5.3 (Typed Nominal Pattern Matching). Let $\Phi \Vdash_{\Sigma} \nabla \vdash l \to r\colon \tau$ be a typeable rule. A **typed nominal pattern matching problem**, $(\Phi \Vdash_{\Sigma} \nabla \vdash l) \stackrel{?}{\approx}_{\alpha} (\Gamma \Vdash_{\Sigma} \Delta \vdash s)$ is a pair of tuples, where Φ and Γ are environments, Σ is a signature, ∇ and Δ are freshness contexts, and l and s are terms, such that the variables and type variables occurring on the left-hand side are disjoint from those occurring on the right-hand side, that is, $\mathsf{vars}(\Phi \Vdash_{\Sigma} \nabla \vdash l) \cap \mathsf{vars}(\Gamma \Vdash_{\Sigma} \Delta \vdash s) = \varnothing$ and $\mathsf{tvars}(\Phi) \cap \mathsf{tvars}(\Gamma) = \varnothing$.

The **solution to a typed nominal pattern matching problem**, if it exists, is the least 3-tuple, $(S,\ \theta,\ \pi)$, of a type substitution, a variable substitution and a permutation such that the following conditions are met.

(1) $\Delta \vdash \left\{\nabla\,\theta,\ \pi \cdot (l\,\theta) \approx_{\alpha} s\right\}$
(2) θ is well-typed in Γ, Σ and Δ for $({}^{\pi}\Phi)\,S \Vdash_{\Sigma} \nabla \vdash \pi \cdot l\colon \tau\,S$.

Condition 1 defines a solution, to an untyped *equivariant* nominal matching problem and determines the substitution, θ, and the permutation, π, of the solution to the typed nominal matching problem.

Condition 2 checks that the variable substitution θ is well-typed for an instance (given by the type substitution S) of the typing judgement for the permuted pattern. It ensures that the term substituted for each variable is typeable in the essential environment for that variable in the pattern with the addition of any extra associations in the term's environment. Note that the second condition entails that associations for atoms in Γ are not updated with incompatible types. Note also that θ is idempotent due to the fact that variables in the left-hand side of a pattern matching problem are disjoint from those in the right-hand side.

To see why it is necessary to check the variable substitution θ, consider the following example.

Example 5.4. Consider a signature Σ, such that $\Sigma_{\mathsf{f}} = \forall(\alpha).\langle \alpha \hookrightarrow \mathtt{Nat}\rangle$ and $\Sigma_{\mathsf{true}} = \langle() \hookrightarrow \mathtt{Bool}\rangle$. Given the pattern $\mathsf{f}\,X$, and the term $\mathsf{f}\,\mathsf{true}$, then the untyped pattern matching problem $\varnothing \vdash \mathsf{f}\,X \stackrel{?}{\approx}_{\alpha} \varnothing \vdash \mathsf{f}\,\mathsf{true}$ has a solution $[X \mapsto \mathsf{true}]$, but the typed pattern matching problem $(X\colon \mathtt{Nat} \Vdash_{\Sigma} \varnothing \vdash \mathsf{f}\,X) \stackrel{?}{\approx}_{\alpha} (\varnothing \Vdash_{\Sigma} \varnothing \vdash \mathsf{f}\,\mathsf{true})$ has none: the variable substitution $[X \mapsto \mathsf{true}]$ is not well-typed, because X is required to have the type `Nat`, but it is instantiated with a term of type `Bool`.

The following proof shows that given a typeable pattern, then a typeability problem that is solved by typed nominal pattern matching is typeable with an instance of the type of the pattern.

Theorem 5.5 (Typed Nominal Pattern Matching). *If* $\Phi \Vdash_\Sigma \nabla \vdash l : \tau$, *and* $(\Phi \Vdash_\Sigma \nabla \vdash l) \stackrel{?}{\approx}_\alpha (\Gamma \Vdash_\Sigma \Delta \vdash s) = (S,\ \theta,\ \pi)$, *then* $\Gamma \Vdash_\Sigma \Delta \vdash \pi \cdot (l\,\theta) : \tau\, S$.

Proof. By Lemma 4.20, $({}^{\pi}\Phi)\, S \Vdash_\Sigma \nabla \vdash \pi \cdot l : \tau S$. By Theorem 4.23, $\Gamma \Vdash_\Sigma \Delta \vdash \pi \cdot (l\,\theta) : \tau\, S$. □

5.1.3 Typed Nominal Rewriting. The conditions of typed nominal rewriting are inherited from nominal rewriting (see section 2.6 and [33, 35]) and extended with types.

Definition 5.6 (Typed Nominal Rewriting). Given a rewrite theory, $\mathcal{R}$, of which, R is a typeable rewrite rule of the form $\Phi \Vdash_\Sigma \nabla \vdash l \to r : \sigma$, a typeability problem, $\Gamma \Vdash_\Sigma \Delta \vdash s$, and a term, t, such that $\mathsf{vars}(R) \cap \Big(\mathsf{vars}(\Gamma \Vdash_\Sigma \Delta \vdash s) \cup \mathsf{vars}(t)\Big) = \varnothing$ and $\mathsf{tvars}(R) \cap \mathsf{tvars}(\Gamma) = \varnothing$ (renaming variables or type variables in R, if necessary), then the one-step typed nominal rewriting relation, $\Delta \vdash s \to^T_{\mathcal{R}} t$, is the least relation generated over the theory $\mathcal{R}$ such that the following conditions are met.

(1) $s \equiv C\,[s']$
(2) $\Gamma \Vdash_\Sigma \Delta \vdash s : \tau$, with a subderivation $\Gamma' \Vdash_\Sigma \Delta \vdash s' : \tau'$.
(3) $(\Phi \Vdash_\Sigma \nabla \vdash l) \stackrel{?}{\approx}_\alpha (\Gamma' \Vdash_\Sigma \Delta \vdash s')$ has the solution $(S,\ \theta,\ \pi)$.
(4) $\Delta \vdash C\,[\pi \cdot (r\,\theta)] \approx_\alpha t$.

The **typed nominal rewriting** relation over a rewrite theory, $\mathcal{R}$, written $\Delta \vdash s \twoheadrightarrow^T_{\mathcal{R}} t$, is the reflexive, transitive closure of the one-step typed nominal rewriting relation.

5.1.4 Subject Reduction. It is now proved that typed nominal rewriting enjoys the property of subject reduction; that is, given typeable rules, the type of a term is preserved under rewriting.

Theorem 5.7 (Subject Reduction). *Given a typeable rewrite rule,* $\Phi \Vdash_\Sigma \nabla \vdash l \to r : \sigma$, *let* $\mathcal{R}$ *be the rewrite theory* $(\Sigma, \{\nabla \vdash l \to r\})$; *if* $\Gamma \Vdash_\Sigma \Delta \vdash s : \tau$ *and* $\Delta \vdash s \twoheadrightarrow^T_{\mathcal{R}} t$ *then* $\Gamma \Vdash_\Sigma \Delta \vdash t : \tau$.

Proof. By Corollary 4.31, it suffices to prove that, if $\mathsf{pt}(\Gamma \Vdash_\Sigma \Delta \vdash s) = (\mathsf{id},\ \tau)$ and $\Delta \vdash s \twoheadrightarrow^T_{\mathcal{R}} t$ then $\Gamma \Vdash_\Sigma \Delta \vdash t : \tau$.

The proof proceeds by induction on the length of the finite rewrite sequence, $\Delta \vdash s \twoheadrightarrow^T_{\mathcal{R}} t$ and thus it also suffices to prove that subject reduction holds for the one-step nominal rewriting relation, $\Delta \vdash s \to^T_{\mathcal{R}} t$.

From this base case, it is known that $s \equiv C\,[s']$ and that $\Delta \vdash \nabla\, \theta \cup \Big\{\pi \cdot (l\,\theta) \approx_\alpha s',\ C\,[\pi \cdot (r\,\theta)] \approx_\alpha t\Big\}$.

From the assumption that the term, s, is typeable, one deduces that there exists a subderivation such that $\Gamma' \Vdash_\Sigma \Delta \vdash s' : \tau'$, for the subterm s' (Definition 5.6, part 2).

However, $\mathsf{pt}(\Phi \Vdash_\Sigma \nabla \vdash l) = (\mathsf{id},\ \sigma)$ and by Theorem 5.5 and Theorem 4.24, then $\Gamma' \Vdash_\Sigma \Delta \vdash s' : \sigma\, S$ and therefore there exists some type substitution, T, such that $\Gamma'\, T \equiv \Gamma'$ and $\sigma\, S\, T \equiv \tau'$.

By Theorem 4.24, that is, that alpha-equivalence respects types, it follows now that $\Gamma' \Vdash_\Sigma \Delta \vdash \pi \cdot l\,\theta : \sigma\, S\, T$.

By the assumptions on typeable rewrite rules, $\Phi \Vdash_\Sigma \nabla \vdash r : \sigma$, and by Lemma 4.18, the instantiation of types, it follows that $\Phi\, S\, T \Vdash_\Sigma \nabla \vdash r : \sigma\, S\, T$. By Lemma 4.20, ${}^{\pi}\Phi\, S\, T \Vdash_\Sigma \nabla \vdash \pi \cdot r : \sigma\, S\, T$.

It is now proved that θ is well-typed in Γ', Δ for ${}^{\pi}\Phi\, S \Vdash_\Sigma \nabla \vdash \pi \cdot r : \sigma\, S$, that is, it satisfies the conditions in Definition 4.21. Condition 2 and Condition 1 follow directly from the fact that θ is a solution to a typed pattern matching problem, and hence well-typed for the pattern under the same type and freshness environments. Condition 3a and Condition 4 follow from the second condition of Definition 5.1, namely that l has the same essential environments for each variable judgement as those that occur in r, and condition 3b follows from the fact that typeable rules are uniform.

Hence $\Gamma' \Vdash_\Sigma \Delta \vdash \pi \cdot r\,\theta : \sigma\,S$. $\Gamma'\,T \equiv \Gamma'$, and so again by Lemma 4.18, it also holds that $\Gamma' \Vdash_\Sigma \Delta \vdash \pi \cdot (r\,\theta) : \sigma\,S\,T$.

Hence, $\Gamma \Vdash_\Sigma \Delta \vdash C\,[\pi \cdot (r\,\theta)] : \tau$, and then, again by Theorem 4.24, $\Gamma \Vdash_\Sigma \Delta \vdash t : \tau$, as required. □

5.2 Typed Closed Nominal Rewriting

As previously noted, closed nominal rewriting, Definition 2.30, is markedly more efficient than the equivariant matching required in Definition 5.3 (see [9, 10, 12]). Thus a system of typed closed nominal rewriting is highly desirable.

In addition, when using typed nominal pattern matching (Definition 5.3), each attempted matching is type-checked and this represents a significant overhead when used at each rewrite step. Thus, the system presented here also adds conditions upon the typing of function applications in rewrite rules to ensure that untyped pattern matching may be safely used to generate rewriting steps — safely in the sense that rewriting preserves types. Rewrite rules may then be typed statically, whilst still ensuring subject reduction.

5.2.1 *Typeable-closed Rewrite Rules.*

Definition 5.8 (Typeable-closed Rewrite Rule). A **typeable-closed rewrite rule**, R, written, $\Phi \Vdash_\Sigma \nabla \vdash l \to r : \tau$ is a tuple of a freshness context, ∇, an environment, Φ, which has type associations for variables only, a signature, Σ, two terms, l and r and a type, τ such that the following conditions are met.

(1) The rule, $\nabla \vdash l \to r$, is closed.
(2) $\mathrm{pt}(\Phi \Vdash_\Sigma \nabla \vdash l) = (\mathsf{id},\ \tau)$ and $\Phi \Vdash_\Sigma \nabla \vdash (l,\ r) : (\tau \times \tau)$.
(3) Every variable in l has an occurrence within a function application $\mathsf{f}\,t$, and for every subderivation $\Gamma' \Vdash_\Sigma \Delta \vdash \mathsf{f}\,t : \tau'$ in l where t is not ground, if $\Sigma_\mathsf{f} = \forall(\overline{\alpha}).\langle \sigma \hookrightarrow \tau \rangle$, then the type of t is as general as σ.

Condition 1 in Definition 5.8 is stronger than Condition 1 in Definition 5.1 since every closed rule is uniform. Condition 2 is the same as in Definition 5.1. Therefore, every typeable-closed rule is a typeable rewrite rule.

Condition 3 in Definition 5.8 says that the subterm that forms the argument to an occurrence of a term-former has the most general type possible. This imposes the rigidity of generic type variables in rewrite rules using function applications, (that is, if f is declared to be polymorphic, its rewrite rules should be generic) and ensures that for every variable $X \in \mathsf{vars}(l)$, Φ_X is the most general type possible. Thus, the problem regarding different instantiations of generic type variables when typing matching substitutions discussed in Example 5.4 does not arise. To see why variables should occur under function applications in l, consider the pathological example: $X : \mathtt{Nat} \Vdash_\Sigma \varnothing \vdash (X,\ X) \to (\mathsf{suc}\,X,\ X) : \mathtt{Nat} \times \mathtt{Nat}$. Then the typeable term $(\mathsf{true},\ \mathsf{true})$, where $\Sigma_{\mathsf{true}} = \mathtt{Bool}$ would rewrite to the untypeable term $(\mathsf{suc}\,\mathsf{true},\ \mathsf{true})$).

Example 5.9 (Typeable-closed Rewrite Rules). Assume a signature, Σ, of the following form, $\{\mathsf{fst} : \forall(\alpha\,\beta).\langle(\alpha \times \beta) \hookrightarrow \alpha\rangle,\ \mathsf{snd} : \forall(\alpha\,\beta).\langle(\alpha \times \beta) \hookrightarrow \beta\rangle\}$. One then has the following set of typeable-closed rules, defining surjective pairing (which cannot be implemented by a compositional translation to the lambda calculus, [5]).

$$X : \alpha,\ Y : \beta \Vdash_\Sigma \varnothing \vdash \mathsf{fst}\,(X,\ Y) \to X : \alpha$$
$$X : \alpha,\ Y : \beta \Vdash_\Sigma \varnothing \vdash \mathsf{snd}\,(X,\ Y) \to Y : \beta$$
$$Z : (\alpha \times \beta) \Vdash_\Sigma \varnothing \vdash (\mathsf{fst}\,Z,\ \mathsf{snd}\,Z) \to Z : (\alpha \times \beta)$$

Note that rewrite rules may be closed but not typeable-closed. The conditions on typeable-closed rules, given in Definition 5.8 are such that rewrite rules can be typed statically and typed nominal

pattern matching not be required. A closed rule that is not typeable-closed may still be typeable according to Definition 5.1.

Also note that condition 1 in Definition 5.8 cannot be deduced from condition 2: the fact that all the essential environments for a given variable are the same throughout the type derivation does not mean that the rule is closed, as the following example shows: $X : \alpha \Vdash_{\Sigma} \varnothing \vdash \mathsf{f}\,(X) \to (a\ b) \cdot X : \alpha$.

5.2.2 Typed Closed Nominal Rewriting Relation. The conditions given in Definition 5.8 allow typed closed nominal rewriting to be defined using nominal matching and closed nominal rewriting; that is, *neither equivariant matching nor typed matching is required.*

However typed closed nominal rewriting still has the property of subject reduction; that is, given typeable-closed rules, the type of a typeable term is preserved under rewriting.

Theorem 5.10 (Subject Reduction). *Given a typeable-closed rewrite rule,* $\Phi \Vdash_{\Sigma} \nabla \vdash l \to r : \sigma$, *let* $\mathcal{R}$ *be the rewrite theory* $(\Sigma, \{\nabla \vdash l \to r\})$; *if* $\Gamma \Vdash_{\Sigma} \Delta \vdash s : \tau$ *and* $\Delta \vdash s \twoheadrightarrow^{C}_{\mathcal{R}} t$ *then* $\Gamma \Vdash_{\Sigma} \Delta \vdash t : \tau$.

Proof. By induction on the length of the rewriting sequence. The base case, where t is obtained from s in one closed-rewriting step is proved below.

Closed nominal rewriting reduces to nominal rewriting: by Definition 2.30, if $\Delta \vdash s \to^{C}_{\mathcal{R}} t$, then there exists Δ' (containing freshness assumptions for atoms that do not occur in s or t) such that $\Delta, \Delta' \vdash s \to_{\mathcal{R}} t$. Also, typeable-closed rules are typeable rewrite rules, since the conditions in Definition 5.1 are satisfied.

Therefore, in order to prove subject reduction for typed closed nominal rewriting, it is sufficient to show that the conditions on typeable-closed rewrite rules ensure the conditions in Definition 5.3 of typed nominal pattern matching, that is, that there is a typed nominal rewriting step $\Delta, \Delta' \vdash s \to^{T}_{\mathcal{R}} t$. One may then proceed as in Theorem 5.7 in Section 5.1.4.

Since $\Delta, \Delta' \vdash s \to_{\mathcal{R}} t$, by definition of nominal rewriting (see Definition 2.22), there exists a subterm s' of s and a permutation π such that the nominal pattern matching problem $(\nabla \vdash \pi \cdot l) \stackrel{?}{\approx}_{\alpha} (\Delta, \Delta' \vdash s')$ has solution θ.

To complete the proof, it is sufficient to show that there exists a type substitution S such that $(S,\ \theta,\ \pi)$ is a solution of the corresponding typed nominal pattern matching problem.

Condition 1 of Definition 5.3 is satisfied by the definition of nominal rewriting (see Definitions 2.22 and 2.17):

Condition 2 of Definition 5.3 requires checking the conditions stated in Definition 4.21. Recall that the variables in the rule are newly generated for each rewrite step, so all the variables in l are in $\mathrm{dom}(\theta)$, and θ is idempotent.

Condition 1 of Definition 4.21 holds by definition of rewriting.

Since the rewrite rule must be closed, and for the type environment, Φ, of a typeable-closed rewrite rule, it may be considered that $\mathrm{atms}(\Phi) = \varnothing$, condition 2 in Definition 4.21 holds.

Condition 3a (that the variable substitution is well-typed with the types provided in ΦS for the variables in l) and Condition 4 are here a consequence of *the fact that only typeable terms are subject to rewriting*, using Condition 3 of Definition 5.8 that ensures that each occurrence of a variable in l is under a term-former that is typed with the most general type possible. Indeed, it can be proved by induction on the structure of l that there is a type derivation for l using ΦS, for a type substitution S, such that if E is the essential environment of X in $\Phi S \Vdash_{\Sigma} \nabla \vdash l : \sigma S$ then $\Gamma \bowtie E_{atm} \Vdash_{\Sigma} \Delta \vdash \theta(X) : (\Phi S)_X$.

Condition 3b of Definition 4.21 holds because closed rewriting steps are generated using freshened versions of rules, therefore the support of the permutations occurring in the rule is always disjoint with respect to the set of atoms occurring in the term to be rewritten. □

Consider again the set of rewrite rules for $\beta\eta$-reduction for a polymorphically typed lambda calculus given in Example 5.2. Whilst all the rules in this set are typeable rewrite rules, not all are typeable-closed. Specifically, the rule for substitution over a lambda abstraction does not comply to Condition 3 of Definition 5.8: in the subterm $\mathsf{sub}\,([a]\,(\mathsf{lam}\,[b]\,X),\ Z)$, the first argument has type $[\gamma]\,(\alpha' \Rightarrow \alpha)$ which is too specific an 'argument' type for the declaration for the substitution term-former sub. This brings to attention the fact that this rule does not exhibit genuine generic polymorphism but is rather making use of an ad-hoc polymorphism where a rewrite depends on the type of the term being rewritten. It also highlights a conflict between the types for the lambda calculus being modelled and the type system for nominal terms explored here that specifies the syntactic representations of the lambda terms. The rules for substitution give behaviour to the syntax trees defined for lambda terms and do not require the information provided by the lambda calculus types being represented.

Subject reduction for both typed nominal rewriting and typed closed nominal rewriting should allow a hybrid approach where the greater efficiency of closed nominal rewriting is harnessed whenever it is possible to do so.

The authors also conjecture that techniques borrowed from research into polymorphically typed logic programs [42, 53] and generalised algebraic data types (GADTs) [54], may allow such ad-hoc polymorphism in rules to be made less problematic.

5.3 Typed Nominal Equational Reasoning

Closed nominal rewriting is complete for nominal equational reasoning when using closed axioms, [34]. Thus it is straightforward to extend the result in Section 5.2 to define a system of typed nominal equational reasoning in which types are preserved.

The definition of a typeable-closed axiom follows closely that for a typeable closed rewrite rule, given in Definition 5.8.

Definition 5.11 (Typeable-closed Axiom). A **typeable-closed axiom** A, written $\Phi \Vdash_\Sigma \nabla \vdash l = r : \tau$, is a tuple of a freshness context ∇, an environment Φ, containing type associations for variables only, a signature Σ, two terms l and r and a type τ, such that the following conditions are met.

(1) The axiom $\nabla \vdash l = r$ is closed.
(2) $\mathsf{pt}(\Phi \Vdash_\Sigma \nabla \vdash (l,\ r)) = (\mathsf{id},\ (\tau \times \tau))$
(3) Every variable in l and r has an occurrence within a function application $\mathsf{f}\,t$, and for every subderivation $\Gamma' \Vdash_\Sigma \Delta \vdash \mathsf{f}\,t : \tau'$ in either l or r, where t is not ground, if $\Sigma_\mathsf{f} = \forall(\overline{\alpha}).\langle \sigma \hookrightarrow \tau \rangle$, then the type of t is as general as σ.

From the aforementioned result [34, Theorem 5.19], that closed nominal rewriting is complete for nominal equational reasoning when using closed axioms, it is again straightforward to prove that typed nominal equational reasoning also preserves types, by demonstrating that the conditions on typeable-closed axiom given in Definition 5.11 are sufficient to guarantee typed nominal pattern matching (Definition 5.5).

The following axioms defining quantifier exchange for a fragment of first-order logic provide a good example of typeable-closed axioms.

Assume Σ to be the signature, $\{\mathsf{and} : \langle(\mathsf{o}\times\mathsf{o}) \hookrightarrow \mathsf{o}\rangle,\ \mathsf{or} : \langle(\mathsf{o}\times\mathsf{o}) \hookrightarrow \mathsf{o}\rangle,\ \mathsf{not} : \langle \mathsf{o} \hookrightarrow \mathsf{o}\rangle,\ \mathsf{all} : \langle[\mathtt{i}]\,\mathsf{o} \hookrightarrow \mathsf{o}\rangle,\ \mathsf{some} : \langle[\mathtt{i}]\,\mathsf{o} \hookrightarrow \mathsf{o}\rangle\}$, where the base types, $\mathtt{i}$ and o, represent individuals and propositions respectively.

Example 5.12 (Typeable-closed Axioms).

$$P\colon \mathsf{o},\ Q\colon \mathsf{o} \Vdash_\Sigma a \# P \vdash \mathsf{and}\,(P,\ \mathsf{all}\,[a]\,Q) = \mathsf{all}\,[a]\,\mathsf{and}\,(P,\ Q)\colon \mathsf{o}$$
$$P\colon \mathsf{o},\ Q\colon \mathsf{o} \Vdash_\Sigma a \# P \vdash \mathsf{or}\,(P,\ \mathsf{all}\,[a]\,Q) = \mathsf{all}\,[a]\,\mathsf{or}\,(P,\ Q)\colon \mathsf{o}$$
$$P\colon \mathsf{o},\ Q\colon \mathsf{o} \Vdash_\Sigma a \# P \vdash \mathsf{and}\,(\mathsf{all}\,[a]\,Q,\ P) = \mathsf{all}\,[a]\,\mathsf{and}\,(Q,\ P)\colon \mathsf{o}$$
$$P\colon \mathsf{o},\ Q\colon \mathsf{o} \Vdash_\Sigma a \# P \vdash \mathsf{or}\,(\mathsf{all}\,[a]\,Q,\ P) = \mathsf{all}\,[a]\,\mathsf{or}\,(Q,\ P)\colon \mathsf{o}$$

$$P\colon \mathsf{o},\ Q\colon \mathsf{o} \Vdash_\Sigma a \# P \vdash \mathsf{and}\,(P,\ \mathsf{some}\,[a]\,Q) = \mathsf{some}\,[a]\,\mathsf{and}\,(P,\ Q)\colon \mathsf{o}$$
$$P\colon \mathsf{o},\ Q\colon \mathsf{o} \Vdash_\Sigma a \# P \vdash \mathsf{or}\,(P,\ \mathsf{some}\,[a]\,Q) = \mathsf{some}\,[a]\,\mathsf{or}\,(P,\ Q)\colon \mathsf{o}$$
$$P\colon \mathsf{o},\ Q\colon \mathsf{o} \Vdash_\Sigma a \# P \vdash \mathsf{and}\,(\mathsf{some}\,[a]\,Q,\ P) = \mathsf{some}\,[a]\,\mathsf{and}\,(Q,\ P)\colon \mathsf{o}$$
$$P\colon \mathsf{o},\ Q\colon \mathsf{o} \Vdash_\Sigma a \# P \vdash \mathsf{or}\,(\mathsf{some}\,[a]\,Q,\ P) = \mathsf{some}\,[a]\,\mathsf{or}\,(Q,\ P)\colon \mathsf{o}$$

$$P\colon \mathsf{o} \Vdash_\Sigma \varnothing \vdash \mathsf{not}\,\mathsf{all}\,[a]\,P = \mathsf{all}\,[a]\,\mathsf{not}\,P\colon \mathsf{o}$$
$$P\colon \mathsf{o} \Vdash_\Sigma \varnothing \vdash \mathsf{not}\,\mathsf{some}\,[a]\,P = \mathsf{some}\,[a]\,\mathsf{not}\,P\colon \mathsf{o}$$

Recall again Example 5.2. One may also axiomatise $\beta\eta$-reduction for that polymorphically typed lambda calculus as follows, [40].

Example 5.13 (Axiomatised Polymorphic Lambda Calculus).

$$X\colon \alpha,\ Y\colon \beta \Vdash_\Sigma \varnothing \vdash \mathsf{app}\,((\mathsf{lam}\,[a]\,X),\ Y) = \mathsf{sub}\,([a]\,X,\ Y)\colon \alpha$$
$$X\colon \alpha \Rightarrow \beta \Vdash_\Sigma a \# X \vdash \mathsf{lam}\,[a]\,(\mathsf{app}\,(X,\ a)) = X\colon \alpha \Rightarrow \beta$$

$$X\colon \alpha,\ Z\colon \gamma \Vdash_\Sigma a \# X \vdash \mathsf{sub}\,([a]\,X,\ Z) = X\colon \alpha$$

$$Z\colon \gamma \Vdash_\Sigma \varnothing \vdash \mathsf{sub}\,([a]\,a,\ Z) = Z\colon \gamma$$

$$X\colon \beta \Rightarrow \alpha,\ Y\colon \beta,\ Z\colon \gamma \Vdash_\Sigma \varnothing \vdash \mathsf{sub}\,([a]\,(\mathsf{app}\,(X,\ Y)),\ Z)$$
$$= \mathsf{app}\,(\mathsf{sub}\,([a]\,X,\ Z),\ \mathsf{sub}\,([a]\,Y,\ Z))\colon \alpha$$

$$X\colon \alpha,\ Z\colon \gamma \Vdash_\Sigma b \# Z \vdash \mathsf{sub}\,([a]\,(\mathsf{lam}\,[b]\,X),\ Z)$$
$$= \mathsf{lam}\,[b]\,(\mathsf{sub}\,([a]\,X,\ Z))\colon \alpha' \Rightarrow \alpha$$

$$X\colon \alpha,\ a\colon \beta \Vdash_\Sigma \varnothing \vdash \mathsf{sub}\,([a]\,X,\ a) = X\colon \alpha$$

The axioms for substitution over abstractions and applications are not typeable-closed for the same reasons as those outlined above in Section 5.2.2. However, in addition, the axiom, $X\colon \alpha,\ a\colon \beta \Vdash_\Sigma \varnothing \vdash \mathsf{sub}\,([a]\,X,\ a) = X\colon \alpha$ is not typeable-closed because it violates Condition 1 of Definition 5.11. The remaining axioms are typeable-closed.

Non-closed rules are not common (and are not accepted by the standard higher-order rewriting formalisms), but interesting examples do exist, such as for models of the π-calculus; here, in this case, the rule is needed for completeness with respect to the models, [40]. A formalism of non-closed typed nominal equational reasoning, following that for typed nominal rewriting, given in Section 5.1, using typed nominal pattern matching would allow this axiom to be typed.

5.4 Further Examples

This subsection gives a number of further examples of sets of rewrite rules and axioms that are typeable or typeable-closed.

5.4.1 Arithmetic. Given a base type for natural numbers, Nat, and a signature, Σ, thus, $\{0\colon\langle()\hookrightarrow\mathsf{Nat}\rangle,\ \mathsf{suc}\colon\langle\mathsf{Nat}\hookrightarrow\mathsf{Nat}\rangle,\ \mathsf{add}\colon\langle(\mathsf{Nat}\times\mathsf{Nat})\hookrightarrow\mathsf{Nat}\rangle,\ \mathsf{mul}\colon\langle(\mathsf{Nat}\times\mathsf{Nat})\hookrightarrow\mathsf{Nat}\rangle,\ \mathsf{ack}\colon\langle(\mathsf{Nat}\times\mathsf{Nat})\hookrightarrow\mathsf{Nat}\rangle\}$, then one has the following typeable-closed rewrite rules for arithmetic.

$$X\colon\mathsf{Nat}\Vdash_\Sigma\varnothing\vdash\mathsf{add}\,(X,\ 0)\to X\colon\mathsf{Nat}$$
$$X\colon\mathsf{Nat},\ Y\colon\mathsf{Nat}\Vdash_\Sigma\varnothing\vdash\mathsf{add}\,(X,\ \mathsf{suc}\,Y)\to\mathsf{suc}\,(\mathsf{add}\,(X,\ Y))\colon\mathsf{Nat}$$

$$X\colon\mathsf{Nat}\Vdash_\Sigma\varnothing\vdash\mathsf{mul}\,(X,\ 0)\to 0\colon\mathsf{Nat}$$
$$X\colon\mathsf{Nat},\ Y\colon\mathsf{Nat}\Vdash_\Sigma\varnothing\vdash\mathsf{mul}\,(X,\ \mathsf{suc}\,Y)\to\mathsf{add}\,(X,\ \mathsf{mul}\,(X,\ Y))\colon\mathsf{Nat}$$

$$Y\colon\mathsf{Nat}\Vdash_\Sigma\varnothing\vdash\mathsf{ack}\,(0,\ Y)\to\mathsf{suc}\,Y\colon\mathsf{Nat}$$
$$X\colon\mathsf{Nat}\Vdash_\Sigma\varnothing\vdash\mathsf{ack}\,(\mathsf{suc}\,X,\ 0)\to\mathsf{ack}\,(X,\ \mathsf{suc}\,0)\colon\mathsf{Nat}$$
$$X\colon\mathsf{Nat},\ Y\colon\mathsf{Nat}\Vdash_\Sigma\varnothing\vdash\mathsf{ack}\,(\mathsf{suc}\,X,\ \mathsf{suc}\,Y)\to\mathsf{ack}\,(X,\ \mathsf{ack}\,(\mathsf{suc}\,X,\ Y))\colon\mathsf{Nat}$$

In general, any many-sorted first-order term rewriting system over a signature consisting of a set $\mathcal{S}$ of sorts s, and a set $\mathcal{F}$ of function symbols $\mathsf{f}_{s_1\ldots s_n\to s}$, can be specified as a typeable closed nominal rewriting system as follows. Define a type-former s for each sort $s\in\mathcal{S}$ (that is, sorts are base types) and a signature $\Sigma=\{\mathsf{f}\colon\langle(s_1\ldots s_n)\hookrightarrow s\rangle\mid\mathsf{f}_{s_1\ldots s_n\to s}\in\mathcal{F}\}$. Any first-order term built over $\mathcal{F}$ and a set of variables $\mathcal{X}$ may now be represented as a nominal term by replacing each variable $X\in\mathcal{X}$ with a moderated variable $\mathsf{id}\cdot X$, and first-order rewrite rules can be represented as nominal rewrite rules in an empty freshness context. Since there are no atoms in first-order rules (and hence no abstractions or permutations) and no polymorphism, all rules are typeable closed, and the nominal rewriting relation coincides with the first-order rewriting relation.

5.4.2 Higher-order Logic. Let o, be a base type representing propositions and Σ be the signature, $\{\top\colon\langle()\hookrightarrow\mathsf{o}\rangle,\ \bot\colon\langle()\hookrightarrow\mathsf{o}\rangle,\ \wedge\colon\langle(\mathsf{o}\times\mathsf{o})\hookrightarrow\mathsf{o}\rangle,\ \Rightarrow\colon\langle(\mathsf{o}\times\mathsf{o})\hookrightarrow\mathsf{o}\rangle,\ \Leftrightarrow\colon\langle(\mathsf{o}\times\mathsf{o})\hookrightarrow\mathsf{o}\rangle,\ \approx\colon\forall(\alpha).\langle(\alpha\times\alpha)\hookrightarrow\mathsf{o}\rangle,\ \forall\colon\forall(\alpha).\langle[\alpha]\,\mathsf{o}\hookrightarrow\mathsf{o}\rangle,\ \sigma\colon\forall(\alpha).\langle([\alpha]\,\mathsf{o}\times\alpha)\hookrightarrow\mathsf{o}\rangle\}$.

Write function applications for substitution, $\sigma\,([a]\,s,\ t)$, as $s\,[a/t]$, and use infix notation for all the binary type-formers.

The set of rewrite rules defined below, representing equality and simplification for (a fragment of) higher-order logic, are typeable-closed.

$$Z\colon\alpha\Vdash_\Sigma\varnothing\vdash Z\approx Z\to\top\colon\mathsf{o}$$
$$X\colon\mathsf{o}\Vdash_\Sigma a\,\#\,X\vdash\forall\,[a]\,X\to X\colon\mathsf{o}$$
$$X\colon\mathsf{o},\ Y\colon\mathsf{o}\Vdash_\Sigma\varnothing\vdash\forall[a]\,(X\wedge Y)\to\forall\,[a]\,X\wedge\forall\,[a]\,Y\colon\mathsf{o}$$

One also has the following typeable-closed axioms.

$$P\colon\mathsf{o},\ R\colon\alpha\Vdash_\Sigma\varnothing\vdash\forall\,[a]\,P\Rightarrow P\,[a/R]=\top\colon\mathsf{o}$$
$$P\colon\mathsf{o},\ Q\colon\mathsf{o}\Vdash_\Sigma a\,\#\,P\vdash\forall\,[a]\,(P\Rightarrow Q)\Leftrightarrow(P\Rightarrow\forall\,[a]\,Q)=\top\colon\mathsf{o}$$
$$P\colon\mathsf{o},\ R\colon\alpha,\ S\colon\alpha\Vdash_\Sigma\varnothing\vdash R\approx S\wedge P\,[a/R]\Rightarrow P\,[a/S]=\top\colon\mathsf{o}$$

5.4.3 Gödel's System T. Consider an extension of Example 5.2 for the polymorphically typed lambda calculus to represent Gödel's System T of total higher-order functions over the natural numbers.

Let Nat be a base type for the natural numbers and let $\Rightarrow$, again be a type-former, written infix, for the construction of function types.

Given the extended signature, Σ, of the form, $\{\mathsf{lam}\colon \forall(\alpha,\ \beta).\langle[\alpha]\,\beta \hookrightarrow \alpha \Rightarrow \beta\rangle,\ \mathsf{app}\colon \forall(\alpha,\ \beta).\langle(\alpha \Rightarrow \beta \times \alpha) \hookrightarrow \beta\rangle,\ \mathsf{sub}\colon \forall(\alpha,\ \beta).\langle([\alpha]\,\beta \times \alpha) \hookrightarrow \beta\rangle,\ 0\colon \langle() \hookrightarrow \mathsf{Nat}\rangle,\ \mathsf{suc}\colon \langle\mathsf{Nat} \hookrightarrow \mathsf{Nat}\rangle,\ \mathsf{rec}\colon \forall(\alpha).\langle(\alpha \times \alpha \Rightarrow (\mathsf{Nat} \Rightarrow \alpha) \times \mathsf{Nat}) \hookrightarrow \alpha\rangle\}$ then to define System T the following rules are added to those in Example 5.2.

$$Z\colon \gamma \Vdash_{\Sigma} \varnothing \vdash \mathsf{sub}\,([a]\,0,\ Z) \to 0\colon \mathsf{Nat}$$

$$X\colon \mathsf{Nat},\ Z\colon \gamma \Vdash_{\Sigma} \varnothing \vdash \mathsf{sub}\,([a]\,(\mathsf{suc}\,X),\ Z) \to \mathsf{suc}\,(\mathsf{sub}\,([a]\,X,\ Z))\colon \mathsf{Nat}$$

$$\begin{aligned} U\colon \alpha,\ V\colon \alpha \Rightarrow (\mathsf{Nat} \Rightarrow \alpha),\ X\colon \mathsf{Nat},\ Z\colon \gamma \\ \Vdash_{\Sigma} \varnothing \vdash \mathsf{sub}\,([a]\,(\mathsf{rec}\,(U,\ V,\ X)),\ Z) \\ \to \mathsf{rec}\,(\mathsf{sub}\,([a]\,U,\ Z),\ \mathsf{sub}\,([a]\,V,\ Z),\ \mathsf{sub}\,([a]\,X,\ Z))\colon \alpha \end{aligned}$$

$$U\colon \alpha,\ V\colon \alpha \Rightarrow (\mathsf{Nat} \Rightarrow \alpha) \Vdash_{\Sigma} \varnothing \vdash \mathsf{rec}\,(U,\ V,\ 0) \to U\colon \alpha$$

$$\begin{aligned} U\colon \alpha,\ V\colon \alpha \Rightarrow (\mathsf{Nat} \Rightarrow \alpha),\ X\colon \mathsf{Nat} \Vdash_{\Sigma} \varnothing \vdash \mathsf{rec}\,(U,\ V,\ \mathsf{suc}\,X) \\ \to \mathsf{app}\,(\mathsf{app}\,(V,\ \mathsf{rec}\,(U,\ V,\ X)),\ X)\colon \alpha \end{aligned}$$

The specification of the declarations for the term-formers 0, suc and rec, results in the violation of Condition 3 of Definition 5.8, with respect to the declarations for sub in all cases except for the rules for rec, which are typeable-closed.

One can define a lambda term for addition (and for the other arithmetic functions in Section 5.4.1) over Church numerals, thus:

$$\varnothing \Vdash_{\Sigma} \varnothing \vdash \mathsf{lam}\,[m]\,\mathsf{lam}\,[n]\,\mathsf{rec}\,(m,\ \mathsf{lam}\,[a]\,\mathsf{lam}\,[b]\,\mathsf{suc}\,a,\ n)\colon \mathsf{Nat} \Rightarrow (\mathsf{Nat} \Rightarrow \mathsf{Nat})$$

5.4.4 Untyped Lambda Calculus. Suppose a base type, Lam, representing terms of an untyped lambda calculus and let Σ be the signature, $\{\mathsf{lam}\colon \langle[\mathsf{Lam}]\,\mathsf{Lam} \hookrightarrow \mathsf{Lam}\rangle,\ \mathsf{app}\colon \langle(\mathsf{Lam} \times \mathsf{Lam}) \hookrightarrow \mathsf{Lam}\rangle,\ \mathsf{sub}\colon \langle([\mathsf{Lam}]\,\mathsf{Lam} \times \mathsf{Lam}) \hookrightarrow \mathsf{Lam}\rangle\}$.

A set of rewrite rules, which define $\beta\eta$-reduction for the untyped lambda calculus, and all of which satisfy the conditions given for a typeable-closed rule in Definition 5.8, are given below.

$$X\colon \mathsf{Lam},\ Y\colon \mathsf{Lam} \Vdash_{\Sigma} \varnothing \vdash \mathsf{app}\,((\mathsf{lam}\,[a]\,X),\ Y) \to \mathsf{sub}\,([a]\,X,\ Y)\colon \mathsf{Lam}$$

$$X\colon \mathsf{Lam} \Vdash_{\Sigma} a\#X \vdash \mathsf{lam}\,[a]\,(\mathsf{app}\,(X,\ a)) \to X\colon \mathsf{Lam}$$

$$X\colon \mathsf{Lam},\ Z\colon \mathsf{Lam} \Vdash_{\Sigma} a \# X \vdash \mathsf{sub}\,([a]\,X,\ Z) \to X\colon \mathsf{Lam}$$

$$Z\colon \mathsf{Lam} \Vdash_{\Sigma} \varnothing \vdash \mathsf{sub}\,([a]\,a,\ Z) \to Z\colon \mathsf{Lam}$$

$$\begin{aligned} X\colon \mathsf{Lam},\ Y\colon \mathsf{Lam},\ Z\colon \mathsf{Lam} \Vdash_{\Sigma} \varnothing \vdash \mathsf{sub}\,([a]\,(\mathsf{app}\,(X,\ Y)),\ Z) \\ \to \mathsf{app}\,(\mathsf{sub}\,([a]\,X,\ Z),\ \mathsf{sub}\,([a]\,Y,\ Z))\colon \mathsf{Lam} \end{aligned}$$

$$\begin{aligned} X\colon \mathsf{Lam},\ Z\colon \mathsf{Lam} \Vdash_{\Sigma} b \# Z \vdash \mathsf{sub}\,([a]\,(\lambda b.X),\ Z) \\ \to \mathsf{lam}\,[b]\,(\mathsf{sub}\,([a]\,X,\ Z))\colon \mathsf{Lam} \end{aligned}$$

In general, any untyped closed nominal rewriting system can be defined as a typeable closed system with only one base type of terms.

5.4.5 Untyped Linear Lambda Calculus with Unbounded Recursion. The final example in this section specifies a nominal representation of an untyped linear lambda calculus with unbounded recursion, and an abstract machine implementing the *closed reduction strategy*, where β-redexes are reduced only if the argument is closed [1].

Suppose, again, a base type, Lam, representing untyped lambda terms and add to the signature, Σ, the following declaration associations: $0 : \langle () \hookrightarrow \mathsf{Lam} \rangle$, $\mathsf{suc} : \langle \mathsf{Lam} \hookrightarrow \mathsf{Lam} \rangle$, $\mathsf{let} : \langle ([\mathsf{Lam}]\,[\mathsf{Lam}]\,\mathsf{Lam} \times \mathsf{Lam}) \hookrightarrow \mathsf{Lam} \rangle$, $\mathsf{par} : \langle (\mathsf{Lam} \times \mathsf{Lam}) \hookrightarrow \mathsf{Lam} \rangle$ and $\mathsf{rec} : \langle (\mathsf{Lam} \times \mathsf{Lam} \times \mathsf{Lam} \times \mathsf{Lam}) \hookrightarrow \mathsf{Lam} \rangle$. One can now represent the *closed reduction strategy* using the following typeable-closed rewrite rules. Below, for simplicity, $\mathsf{lam}\,[a]\,s$, $\mathsf{app}\,(s,\ t)$ and $\mathsf{sub}\,([a]\,s,\ t)$ will be written as $\lambda a.s$, $s\ t$, and $s\,[t/a]$, respectively.

$$X : \mathsf{Lam},\ Y : \mathsf{Lam} \Vdash_\Sigma \varnothing \vdash (\lambda a.X)\ Y \to X\,[Y/a] : \mathsf{Lam}$$

$$X : \mathsf{Lam} \Vdash_\Sigma a \# X \vdash \lambda a.(X\ a) \to X : \mathsf{Lam}$$

$$X : \mathsf{Lam},\ Z : \mathsf{Lam} \Vdash_\Sigma a \# X \vdash X\,[Z/a] \to X : \mathsf{Lam}$$

$$Z : \mathsf{Lam} \Vdash_\Sigma \varnothing \vdash a\,[Z/a] \to Z : \mathsf{Lam}$$

$$X : \mathsf{Lam},\ Y : \mathsf{Lam},\ Z : \mathsf{Lam} \Vdash_\Sigma \varnothing \vdash (X\ Y)\,[Z/a] \to X\,[Z/a]\ Y\,[Z/a] : \mathsf{Lam}$$

$$X : \mathsf{Lam},\ Z : \mathsf{Lam} \Vdash_\Sigma b \# Z \vdash (\lambda b.X)\,[Z/a] \to \lambda b.(X\,[Z/a]) : \mathsf{Lam}$$

$$Z : \mathsf{Lam} \Vdash_\Sigma \varnothing \vdash 0\,[Z/a] \to 0 : \mathsf{Lam}$$

$$X : \mathsf{Lam},\ Z : \mathsf{Lam} \Vdash_\Sigma \varnothing \vdash (\mathsf{suc}\,X)\,[Z/a] \to \mathsf{suc}\,(X\,[Z/a]) : \mathsf{Lam}$$

$$X : \mathsf{Lam},\ Y : \mathsf{Lam},\ Z : \mathsf{Lam} \Vdash_\Sigma b \# Z,\ c \# Z \vdash (\mathsf{let}\,([b]\,[c]\,X,\ Y))\,[Z/a] \to \mathsf{let}\,([b]\,[c]\,(X\,[Z/a]),\ Y\,[Z/a]) : \mathsf{Lam}$$

$$X : \mathsf{Lam},\ Y : \mathsf{Lam},\ Z : \mathsf{Lam} \Vdash_\Sigma \varnothing \vdash (\mathsf{par}\,(X,\ Y))\,[Z/a] \to \mathsf{par}\,(X\,[Z/a],\ Y\,[Z/a]) : \mathsf{Lam}$$

$$V : \mathsf{Lam},\ W : \mathsf{Lam},\ X : \mathsf{Lam},\ Y : \mathsf{Lam},\ Z : \mathsf{Lam} \Vdash_\Sigma \varnothing \vdash (\mathsf{rec}\,(V,\ W,\ X,\ Y))\,[Z/a] \to \mathsf{rec}\,(V\,[Z/a],\ W\,[Z/a],\ X\,[Z/a],\ Y\,[Z/a]) : \mathsf{Lam}$$

$$V : \mathsf{Lam},\ W : \mathsf{Lam},\ X : \mathsf{Lam},\ Y : \mathsf{Lam} \Vdash_\Sigma \varnothing \vdash \mathsf{rec}\,(\mathsf{par}\,(0,\ V),\ W,\ X,\ Y) \to W : \mathsf{Lam}$$

$$U : \mathsf{Lam},\ V : \mathsf{Lam},\ W : \mathsf{Lam},\ X : \mathsf{Lam},\ Y : \mathsf{Lam} \Vdash_\Sigma \varnothing \vdash \mathsf{rec}\,(\mathsf{par}\,(\mathsf{suc}\,U,\ V),\ W,\ X,\ Y) \to X\ \mathsf{rec}\,(\mathsf{Y}\,\mathsf{par}\,(U,\ V),\ W,\ X,\ Y) : \mathsf{Lam}$$

Extend this model again with base types, $\mathtt{stkT}$, $\mathtt{elemT}$ and $\mathtt{cfgT}$, representing stacks, stack elements and configurations, respectively, and let Σ' be the signature, $\Sigma \cup \{\mathtt{cfg} : \langle (\mathsf{Lam} \times \mathtt{stkT}) \hookrightarrow \mathtt{cfgT} \rangle,\ \mathtt{elem} : \langle \mathsf{Lam} \hookrightarrow \mathtt{elemT} \rangle,\ \mathtt{cns} : \langle (\mathtt{elemT} \times \mathtt{stkT}) \hookrightarrow \mathtt{stkT} \rangle,\ \mathtt{recS1} : \langle (\mathsf{Lam} \times \mathsf{Lam} \times \mathsf{Lam}) \hookrightarrow \mathtt{elemT} \rangle,\ \mathtt{recS2} : \langle (\mathsf{Lam} \times \mathsf{Lam} \times \mathsf{Lam} \times \mathsf{Lam}) \hookrightarrow \mathtt{elemT} \rangle,\ \mathtt{letS} : \langle [\mathsf{Lam}]\,[\mathsf{Lam}]\,\mathsf{Lam} \hookrightarrow \mathtt{elemT} \rangle\}$.

One now has the following typeable-closed rewrite rules to describe the stack machine for the extended lambda calculus defined above, as detailed in [1, Table 6].

$$M\colon \mathsf{Lam},\ N\colon \mathsf{Lam},\ S\colon \mathsf{stkT} \Vdash_{\Sigma'} \varnothing \vdash \mathsf{cfg}\,((M\ N),\ S) \to \mathsf{cfg}\,(M,\ \mathsf{cns}\,(\mathsf{elem}\,N,\ S))\colon \mathsf{cfgT}$$

$$M\colon \mathsf{Lam},\ N\colon \mathsf{Lam},\ S\colon \mathsf{stkT} \Vdash_{\Sigma'} \varnothing \vdash \mathsf{cfg}\,(\lambda a.M,\ \mathsf{cns}\,(\mathsf{elem}\,N,\ S)) \to \mathsf{cfg}\,(M\,[a/N],\ S)\colon \mathsf{cfgT}$$

$$M\colon \mathsf{Lam},\ N\colon \mathsf{Lam},\ S\colon \mathsf{stkT} \Vdash_{\Sigma'} \varnothing \vdash \mathsf{cfg}\,(\mathsf{let}\,([a]\,[b]\,M,\ N),\ S) \to \mathsf{cfg}\,(N,\ \mathsf{cns}\,(\mathsf{letS}\,[a]\,[b]\,M,\ S))\colon \mathsf{cfgT}$$

$$N1\colon \mathsf{Lam},\ N2\colon \mathsf{Lam},\ M\colon \mathsf{Lam},\ S\colon \mathsf{stkT} \Vdash_{\Sigma'} b\ \#\ N1 \vdash \mathsf{cfg}\,(\mathsf{par}\,(N1,\ N2),\ \mathsf{cns}\,(\mathsf{letS}\,[a]\,[b]\,M,\ S)) \to \mathsf{cfg}\,((M\,[a/N1])\,[b/N2],\ S)\colon \mathsf{cfgT}$$

$$N\colon \mathsf{Lam},\ U\colon \mathsf{Lam},\ V\colon \mathsf{Lam},\ W\colon \mathsf{Lam},\ S\colon \mathsf{stkT} \Vdash_{\Sigma'} \varnothing \vdash \mathsf{cfg}\,(\mathsf{rec}\,(N,\ U,\ V,\ W),\ S) \to \mathsf{cfg}\,(N,\ \mathsf{cns}\,(\mathsf{recS1}\,(U,\ V,\ W),\ S))\colon \mathsf{cfgT}$$

$$N1\colon \mathsf{Lam},\ N2\colon \mathsf{Lam},\ U\colon \mathsf{Lam},\ V\colon \mathsf{Lam},\ W\colon \mathsf{Lam},\ S\colon \mathsf{stkT} \Vdash_{\Sigma'} \varnothing \vdash \mathsf{cfg}\,(\mathsf{par}\,(N1,\ N2),\ \mathsf{cns}\,(\mathsf{recS1}\,(U,\ V,\ W),\ S)) \to \mathsf{cfg}\,(N1,\ \mathsf{cns}\,(\mathsf{recS2}\,(N2,\ U,\ V,\ W),S))\colon \mathsf{cfgT}$$

$$T\colon \mathsf{Lam},\ U\colon \mathsf{Lam},\ V\colon \mathsf{Lam},\ W\colon \mathsf{Lam},\ S\colon \mathsf{stkT} \Vdash_{\Sigma'} \varnothing \vdash \mathsf{cfg}\,(0,\ \mathsf{cns}\,(\mathsf{recS2}\,(T,\ U,\ V,\ W),\ S)) \to \mathsf{cfg}\,(U,\ S)\colon \mathsf{cfgT}$$

$$N\colon \mathsf{Lam},\ T\colon \mathsf{Lam},\ U\colon \mathsf{Lam},\ V\colon \mathsf{Lam},\ W\colon \mathsf{Lam},\ S\colon \mathsf{stkT} \Vdash_{\Sigma'} \varnothing \vdash \mathsf{cfg}\,(\mathsf{suc}\,N,\ \mathsf{cns}\,(\mathsf{recS2}\,(T,\ U,\ V,\ W),\ S)) \to \mathsf{cfg}\,(V,\ \mathsf{cns}\,(\mathsf{elem\ rec}\,((W\ \mathsf{par}\,(N,\ T)),\ U,\ V,\ W),\ S))\colon \mathsf{cfgT}$$

The translation of the rules for this stack machine to a system of typeable-closed nominal rewrite rules and their subsequent machine-checking by the first author's implementation identified a previously implicit freshness condition needed to ensure closedness in one of the rules for the reduction of a let-expression ($b\ \#\ N1$).

6 CONCLUSIONS AND FUTURE WORK

This paper has presented two simple type systems for nominal terms in the style of Church. The second of these, the de Bruijn style presentation, could be developed to derive a further simply typed system, in the style of Curry. Such a system was not explored here; it was preferred to demonstrate the Curry-style approach using more powerful, ML-like polymorphic types in Section 4. For that system it has been proved that the derivation of typeable terms respects the notion of alpha-equivalence. That is, that the type system gives derivations compatible with the action of suspended permutations on variables and the atom-capturing nature of variable substitution. It has also been shown to have principal solutions modulo the renaming of those type variables not occurring in

the environment. An algorithm to compute such principal solutions for a given term typeability problem has been given and its soundness and completeness with respect to the derivable typing judgements of the type system proven. Based on this polymorphic type system for nominal terms, two systems of typed nominal rewriting and one of typed nominal equational reasoning have been defined. The first, typed nominal rewriting, is a general formulation that uses an equivariant nominal matching extended with conditions on types. Subject reduction is ensured and it allows non-closed (but uniform) rules to be typed. The second, closed typed nominal rewriting is more efficient, allowing rules to be statically typed and uses standard nominal matching but is more restrictive as to which rules are considered typeable. A formulation of typed nominal equational reasoning follows from this second system as a corollary.

The work of Section 3 on Church-style systems is being drawn upon in the development of a dependent type system for nominal terms to be used in a future nominal logical framework [31]. This work is similar in spirit to that of the Edinburgh Logical Framework (LF) [43], itself a dependently typed extension of the simply typed lambda calculus in the style of Church. The work developed in Section 4 and Section 5 is hoped to be extended to more advanced systems of equational reasoning such as that of [2] in which nominal unification in the presence of a given equational theory is used to define a theory of nominal narrowing. It is hoped to integrate the present work on Curry-style types with this theory and further work on AC-unification to allow a future nominal extension to a modelling and specification language such as Maude [19] or $\mathbb{K}$ [63].

REFERENCES

[1] Sandra Alves, Maribel Fernández, Mário Florido, and Ian Mackie. 2011. Linearity and Recursion in a Typed Lambda-calculus. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming (PPDP '11)*. ACM, 173–182.

[2] Mauricio Ayala-Rincón, Maribel Fernández, and Daniele Nantes-Sobrinho. 2016. Nominal Narrowing. In *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[3] Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering Formal Metatheory. In *POPL*. 3–15.

[4] Franz Baader and Wayne Snyder. 2001. Unification Theory. In *Handbook of Automated Reasoning*, Alan Robinson and Andrei Voronkov (Eds.). Vol. 1. Elsevier, Chapter 8, 445–532.

[5] Henk P. Barendregt. 1974. Pairing without conventional constraints. *Zeitschrift für mathematischen Logik und Grundlagen der Mathematik* 20 (1974), 289–306.

[6] Henk P. Barendregt. 1984. *The Lambda Calculus: its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics, Vol. 103. North-Holland.

[7] Henk P. Barendregt. 2000. Lambda Calculi With Types. In *Handbook of Logic in Computer Science*, S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum (Eds.). Vol. 2. Oxford University Press, 117–309.

[8] Henk P. Barendregt, Will Dekkers, and Richard Statman. 2013. *Lambda Calculus With Types*. Cambridge University Press.

[9] Christophe Calvès and Maribel Fernández. 2010. Matching and Alpha-equivalence Check for Nominal Terms. *Journal of Computer System Sciences* 76, 5 (2010), 283–301.

[10] James Cheney. 2004. The Complexity of Equivariant Unification. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP 2004) (Lecture Notes in Computer Science)*, Vol. 3142. Springer, 332–344.

[11] James Cheney. 2009. A Simple Nominal Type Theory. *Electronic Notes in Theoretical Computer Science* 228 (2009), 37–52.

[12] James Cheney. 2010. Equivariant Unification. *J. Autom. Reasoning* 45, 3 (2010), 267–300. DOI: http://dx.doi.org/10.1007/s10817-009-9164-3

[13] James Cheney. 2012. A Dependent Nominal Type Theory. *Logical Methods in Computer Science* 8, 1 (2012).

[14] James Cheney and Christian Urban. 2003. System Description: Alpha-Prolog, a Fresh Approach to Logic Programming Modulo Alpha-Equivalence. In *UNIF'03*. Universidad Politécnica de Valencia, 15–19.

[15] James Cheney and Christian Urban. 2004. Alpha-Prolog: A Logic Programming Language with Names, Binding and Alpha-Equivalence. In *Proceedings of the 20th International Conference on Logic Programming (ICLP 2004) (Lecture Notes in Computer Science)*, Bart Demoen and Vladimir Lifschitz (Eds.). Springer, 269–283.

[16] James Cheney and Christian Urban. 2008. Nominal logic programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30, 5 (2008), 1–47. DOI: http://dx.doi.org/10.1145/1387673.1387675
[17] Adam Chlipala. 2011. *Certified Programming with Dependent Types.* MIT Press.
[18] Alonzo Church. 1940. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic* 5, 2 (1940), 56–68.
[19] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. 2002. Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science* 285, 2 (Aug. 2002), 187–243.
[20] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. 2011. *Maude Manual.* http://maude.cs.uiuc.edu/maude2-manual/maude-manual.pdf
[21] Ranald A. Clouston. 2007. Closed terms (unpublished notes). (2007). http://users.cecs.anu.edu.au/~rclouston/closedterms.pdf.
[22] Ranald A. Clouston. 2010. *Equational Logic for Names and Binding.* Ph.D. Dissertation. University of Cambridge.
[23] Ranald A. Clouston and Andrew M. Pitts. 2007. Nominal Equational Logic. *Electronic Notes in Theoretical Computer Science* 172 (2007), 223–257. DOI: http://dx.doi.org/10.1016/j.entcs.2007.02.009
[24] Coq Development Team. 2012. *The Coq Proof Assistant: Reference Manual.* http://coq.inria.fr/distrib/current/refman/
[25] Roy L. Crole and Frank Nebel. 2013. Nominal Lambda Calculus: An Internal Language for FM-Cartesian Closed Categories. *Electr. Notes Theor. Comput. Sci.* 298 (2013), 93–117. DOI: http://dx.doi.org/10.1016/j.entcs.2013.09.009
[26] Haskell B. Curry and Robert Feys. 1958. *Combinatory Logic.* Vol. 1. North-Holland.
[27] Luis Damas and Robin Milner. 1982. Principal Type-schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 82).* ACM, 207–212.
[28] Nicolaas G. de Bruijn. 1972. Lambda Calculus Notation With Nameless Dummies, A Tool For Automatic Formula Manipulation, With Application To The Church-Rosser Theorem. *Indagationes Mathematicae* 5, 34 (1972), 381–392.
[29] Elliot Fairweather. 2014. *Type Systems for Nominal Terms.* Ph.D. Dissertation. King's College London.
[30] Elliot Fairweather, Maribel Fernández, and Murdoch J. Gabbay. 2011. Principal Types for Nominal Theories. In *Proceedings of the 18th International Symposium on Fundamentals of Computation Theory (FCT 2011), Oslo, August 2011 (Lecture Notes in Computer Science).* Springer.
[31] Elliot Fairweather, Maribel Fernández, Nora Szasz, and Alvaro Tasistro. 2015. Dependent types for nominal terms with atom substitutions. In *13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015).* Schloss Dagstuhl–Leibniz-Zentrum für Informatik.
[32] Maribel Fernández and Murdoch J. Gabbay. 2007a. Curry-style Types for Nominal Terms. In *Types for Proofs and Programs (TYPES 06).* Lecture Notes in Computer Science, Vol. 4502. Springer, 125–139.
[33] Maribel Fernández and Murdoch J. Gabbay. 2007b. Nominal Rewriting. *Information and Computation* 205, 6 (June 2007), 917–965.
[34] Maribel Fernández and Murdoch J. Gabbay. 2010. Closed Nominal Rewriting and Efficiently Computable Nominal Algebra Equality. In *Proceedings of the 5th International Workshop on Logical Frameworks and Meta-Languages (LFMTP 2010).*
[35] Maribel Fernández, Murdoch J. Gabbay, and Ian Mackie. 2004. Nominal Rewriting Systems. In *Proceedings of the 6th ACM SIGPLAN symposium on Principles and Practice of Declarative Programming (PPDP 2004).* ACM Press, 108–119.
[36] Murdoch Gabbay and Andrew M. Pitts. 2002. A New Approach to Abstract Syntax with Variable Binding. *Formal Asp. Comput.* 13, 3-5 (2002), 341–363. DOI: http://dx.doi.org/10.1007/s001650200016
[37] Murdoch J. Gabbay. 2011a. Foundations of nominal techniques: logic and semantics of variables in abstract syntax. *Bulletin of Symbolic Logic* (2011).
[38] Murdoch J. Gabbay. 2011b. Nominal Terms and Nominal Logics: From Foundations to Meta-mathematics. In *Handbook of Philosphical Logic.* Vol. 17. Kluwer.
[39] Murdoch J. Gabbay and Aad Mathijssen. 2009. Nominal Universal Algebra: Equational Logic with Names and Binding. *Journal of Logic and Computation* 19, 6 (Dec. 2009), 1455–1508.
[40] Murdoch J. Gabbay and Aad Mathijssen. 2010. A nominal axiomatisation of the lambda-calculus. *Journal of Logic and Computation* 20, 2 (April 2010), 501–531.
[41] Murdoch J. Gabbay and Andrew M. Pitts. 1999. A New Approach to Abstract Syntax Involving Binders. In *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS 1999).* IEEE Computer Society Press, 214–224.
[42] Michael Hanus. 1991. Horn Clause Programs with Polymorphic Types: Semantics and Resolution. *Theor. Comput. Sci.* 89, 1 (1991), 63–106. DOI: http://dx.doi.org/10.1016/0304-3975(90)90107-S
[43] Robert Harper, Furio Honsell, and 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, 194–204.
[44] Gérard P. Huet. 1973. The Undecidability of Unification in Third Order Logic. *Information and Control* 22, 3 (1973), 257–267.
[45] Brian Huffman and Christian Urban. 2010. Proof Pearl: A New Foundation for Nominal Isabelle. In *In Proceedings of*

the 1st Conference on Interactive Theorem Proving (ITP 2010) (Lecture Notes in Computer Science), Vol. 6172. 35–50.
[46] Jean-Pierre Jouannaud. 2005. Higher-Order Rewriting: Framework, Confluence and Termination. In *Processes, Terms and Cycles: Steps on the Road to Infinity*, Aart Middeldorp, Vincent Oostrom, Femke Raamsdonk, and Roel Vrijer (Eds.). Springer-Verlag, 224–250.
[47] Jean-Pierre Jouannaud, Femke van Raamsdonk, and Albert Rubio. 2005. Higher-Order Rewriting with Types and Arities. (2005). http://www.lix.polytechnique.fr/~jouannaud/articles/horta.pdf Submitted.
[48] Jan-Willem Klop. 1980. *Combinatory Reduction Systems.* Mathematical Centre Tracts, Vol. 127. Mathematischen Centrum.
[49] Jan-Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. 1993. Combinatory Reduction Systems: Introduction and Survey. *Theoretical Computer Science* 121 (1993), 279–308.
[50] Zhaohui Luo and Randy Pollack. 1992. *LEGO Proof Development System: User's Manual.* Technical Report ECS-LFCS-92-211. University of Edinburgh.
[51] Richard Mayr and Tobias Nipkow. 1998. Higher-order Rewrite Systems and their Confluence. *Theoretical Computer Science* 192 (1998), 3–29.
[52] James McKinna and Robert Pollack. 1999. Some Lambda Calculus and Type Theory Formalized. *Journal of Automated Reasoning* 23, 3-4 (1999), 373–409. citeseer.nj.nec.com/mckinna98some.html
[53] Alan Mycroft and Richard A. O'Keefe. 1984. A Polymorphic Type System for Prolog. *Artif. Intell.* 23, 3 (1984), 295–307. DOI: http://dx.doi.org/10.1016/0004-3702(84)90017-1
[54] Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple Unification-based Type Inference for GADTs. *SIGPLAN Notices* 41, 9 (Sept. 2006), 50–61.
[55] Frank Pfenning and Conal Elliott. 1988. Higher-order Abstract Syntax. In *PLDI (Programming Language Design and Implementation)*. ACM Press, 199–208.
[56] Benjamin C. Pierce. 2002. *Types and Programming Languages.* MIT Press.
[57] Benjamin C. Pierce. 2004. *Advanced Topics in Types and Programming Languages.* MIT Press.
[58] Andrew M. Pitts. 2006. Alpha-structural Recursion and Induction. *J. ACM* 53, 3 (2006), 459–506.
[59] Andrew M. Pitts. 2010. Nominal System T. In *Proceedings of the 37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 2010)*. ACM Press, 159–170.
[60] Andrew M. Pitts, Justus Matthiesen, and Jasper Derikx. 2015. A Dependent Type Theory with Abstractable Names. *Electr. Notes Theor. Comput. Sci.* 312 (2015), 19–50. DOI: http://dx.doi.org/10.1016/j.entcs.2015.04.003
[61] FranÃğois Pottier. 2006. An Overview of Cα ml. In *ACM Workshop on ML (Electronic Notes in Theoretical Computer Science)*, Vol. 148. 27–52.
[62] François Pottier. 2007. Static Name Control for FreshML. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*. IEEE Computer Society, 356–365. DOI: http://dx.doi.org/10.1109/LICS.2007.44
[63] Grigore Rosu and Traian Florin Serbanuta. 2014. K Overview and SIMPLE Case Study. In *Proceedings of International K Workshop (K'11)*. Elsevier.
[64] Bertrand Russell and Alfred North Whitehead. 1910, 1912, 1913. *Principia Mathematica.* Cambidge University Press.
[65] Ulrich Schöpp and Ian Stark. 2004. A Dependent Type Theory with Names and Binding. In *CSL (Lecture Notes in Computer Science)*, Vol. 3210. 235–249.
[66] Mark Shinwell. 2005. *The Fresh Approach: Functional Programming With Names and Binders.* Technical Report 618. University of Cambridge.
[67] Mark R. Shinwell, Andrew M. Pitts, and Murdoch J. Gabbay. 2003. FreshML: Programming with Binders Made Simple, In Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP 2003). *SIGPLAN Notices* 38 (Aug. 2003), 263–274.
[68] Christian Urban. 2008. Nominal Techniques in Isabelle/HOL. *Journal of Automated Reasoning* 40, 4 (May 2008), 327–356.
[69] Christian Urban, Andrew M. Pitts, and Murdoch J. Gabbay. 2004. Nominal Unification. *Theoretical Computer Science* 323, 1–3 (Sept. 2004), 473–497.
[70] Steffen van Bakel and Maribel Fernández. 1997. Normalization Results for Typeable Rewrite Systems. *Information and Computation* 133, 2 (1997), 73–116.
[71] Steffen van Bakel and Maribel Fernández. 2003. Normalization, Approximation and Semantics for Combinator Systems. *Theoretical Computer Science* 290, 1 (2003), 975–1019.
[72] Makarius Wenzel. 2013. *The Isabelle Reference Manual.* http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle2013-2/doc/isar-ref.pdf