Curry-Style Types for Nominal Terms*

Maribel Fernández and Murdoch J. Gabbay

King's College London
Dept. Computer Science
Strand, London WC2R 2LS, UK
maribel.fernandez@kcl.ac.uk

Heriot-Watt University
Dept. Mathematics and Computer Science
Riccarton, Edinburgh EH14 4AS, UK
murdoch.gabbay@gmail.com

Abstract. We define a rank 1 polymorphic type system for nominal terms, where typing environments type atoms, variables and function symbols. The interaction between type assumptions for atoms and substitution for variables is subtle: substitution does not avoid capture and so can move an atom into multiple different typing contexts. We give typing rules such that principal types exist and are decidable for a fixed typing environment. α-equivalent nominal terms have the same types; a non-trivial result because nominal terms include explicit constructs for renaming atoms. We investigate rule formats to guarantee subject reduction. Our system is in a convenient Curry-style, so the user has no need to explicitly type abstracted atoms.
Keywords: binding, polymorphism, type inference, rewriting.

1 Introduction

Nominal terms are used to specify and reason about formal languages with binding, such as logics or programming languages. Consider denumerable sets of **atoms** $a, b, c, \ldots$, **variables** $X, Y, Z, \ldots$, and **term-formers** or **function symbols** $f, g, \ldots$. Following previous work [10, 21], **nominal terms** t are defined by:

$$s, t ::= a \mid [a]t \mid ft \mid (t_1, \ldots, t_n) \mid \pi{\cdot}X \qquad \pi ::= \mathbf{Id} \mid (a\ b) \circ \pi$$

and called respectively **atoms**, **abstractions**, **function applications**, **tuples**, and **moderated variables** (or just **variables**). We call π a **permutation** and read $(a\ b){\cdot}X$ as '**swap** a **and** b **in** X**'**. We say that permutations **suspend** on variables. X is not a term but $\mathbf{Id}{\cdot}X$ is and we usually write it just as X. Similarly we omit the final **Id** in π, writing $(a\ c){\cdot}X$ instead of $((a\ c) \circ \mathbf{Id}){\cdot}X$.

For example, suppose term-formers lam and app. Then the nominal terms $lam[a]a$ and $app(lam[a]a, a)$ represent λ-terms $\lambda x.x$ and $(\lambda x.x)x$, and $lam[a]X$ and $app(lam[a]X, X)$ represent λ-term 'contexts' $\lambda x.\text{-}$ and $(\lambda x.\text{-})\text{-}$. Note how X occurs under different abstractions. Substitution for X is grafting of syntax trees, it does *not* avoid capture of atoms by abstractions; we may call it **instantiation**.

Nominal terms differ from other treatments of syntax-with-binding because they support a capturing substitution, and the notation, although formal, is

* Research partially supported by the EPSRC (EP/D501016/1 "CANS").

close to standard informal practice; for example β-reduction may be represented simply but explicitly as $app(lam[a]X, Y) \to sub([a]X, Y)$ where $sub([a]X, Y)$ is a term which may be given the behaviour of 'non-capturing substitution' (once we instantiate X and Y) by rewrite rules [10, 12].

Now consider *static* semantics, i.e. types like $\tau ::= \mathbb{N} \mid \tau \to \tau$ where we read $\mathbb{N}$ as **numbers** and $\tau \to \tau$ as **function types**. Assigning types to terms partitions the language into 'numbers', or 'functions between numbers', and so on. Java [16], ML [9], and System F [15] demonstrate how this is commercially useful and theoretically interesting.

Existing static semantics for nominal terms type atoms with a special type of atoms $\mathbb{A}$ [21, 20]. But when we write $lam[a]X$ or $lam[a]a$, our intention is $\lambda x.$- or $\lambda x.x$ and we do not expect a to be forbidden from having any type other than $\mathbb{A}$. We can use explicit casting function symbols to inject $\mathbb{A}$ into other types; however the a in $lam[a]X$ still has type $\mathbb{A}$, so we cannot infer more about a until X is instantiated. This notion of typing can only usefully type terms without variables and in the presence of polymorphism such strategies break down entirely.

We now present a Curry-style system with rank 1 polymorphism (*ML-style polymorphism* or *Hindley-Milner types* [9]). Atoms can inhabit *any* type. We can write $lam[a]X$, or $fix[f]X$, or $forall[a]X$, or $app(lam[a]X, lam[b]X)$, and so on, and expect the type system to make sense of these terms, even though these terms explicitly feature context holes representing unknown terms *and* abstractions over those holes. Different occurrences of X may be under different numbers of abstractions, and for different atoms. This means that, when we instantiate X with t, the atoms in t may move into different type contexts and so receive different types. At the same time, the entire type system is consistent with a functional intuition, so X of type $\mathbb{N}$ manages to simultaneously behave like an 'unknown number' and an 'unknown term'.

We give syntax-directed typing rules and show that every typable term has a *principal type* (one which subsumes all others in a suitable sense) in a given environment. Type inference is decidable and types are compatible with α-equivalence on nominal terms. We give a notion of *typable rewrite rule* such that rewriting preserves types. In future, we plan to extend the system with intersection types, to derive a system with principal typings (a type and a type *environment* which subsume all others). With this system we will study normalisation of nominal terms.

2 Background

We continue the notation from the Introduction. Write $V(t)$ for the variables in t and $A(t)$ for the atoms in t (e.g., $a, b, c \in A([a][b](a\ c)\cdot X)$, $X \in V([a][b](a\ c)\cdot X)$). We define $\pi\cdot t$, the **permutation action** of π on t, inductively by:

$\mathbf{Id}\cdot t \equiv t$ and $((a\ b)\circ\pi)\cdot t \equiv (a\ b)\cdot(\pi\cdot t)$, where

$$(a\ b)\cdot a \equiv b \quad (a\ b)\cdot b \equiv a \quad (a\ b)\cdot c \equiv c \quad (c \not\equiv a,b)$$
$$(a\ b)\cdot(\pi\cdot X) \equiv ((a\ b)\circ\pi)\cdot X \quad (a\ b)\cdot ft \equiv f(a\ b)\cdot t$$
$$(a\ b)\cdot[n]t \equiv [(a\ b)\cdot n](a\ b)\cdot t \quad (a\ b)\cdot(t_1,\ldots,t_n) \equiv ((a\ b)\cdot t_1,\ldots,(a\ b)\cdot t_n)$$

For example $(a\ b)\cdot lam[a](a,b,X) \equiv lam[b](b,a,(a\ b)\cdot X)$.

Define $t[X\mapsto s]$, the **(term-)substitution of X for s in t**, by:

$$(ft)[X\mapsto s] \equiv f(t[X\mapsto s]) \quad ([a]t)[X\mapsto s] \equiv [a](t[X\mapsto s]) \quad (\pi\cdot X)[X\mapsto s] \equiv \pi\cdot s$$
$$a[X\mapsto s] \equiv a \quad (t_1,\ldots)[X\mapsto s] \equiv (t_1[X\mapsto s],\ldots) \quad (\pi\cdot Y)[X\mapsto s] \equiv \pi\cdot Y \quad (X \not\equiv Y)$$

Term-substitutions are defined by $\theta ::= \mathbf{Id} \mid [X\mapsto s]\theta$ and have an action given by $t\mathbf{Id} \equiv t$ and $t([X\mapsto s]\theta) \equiv (t[X\mapsto s])\theta$. We write substitutions postfix, and write $\circ$ for composition of substitutions: $t(\theta\circ\theta') \equiv (t\theta)\theta'$.

Nominal syntax represents systems with binding, closely following informal notation. See [21, 10, 11] for examples and discussions of nominal terms and nominal rewriting. It has the same applications as higher-order systems such as Klop's CRSs, Khasidashvili's ERSs, and Nipkow's HRSs [18, 17, 19]. Intuitively, the distinctive features of nominal syntax are:

- X is an 'unknown term'; the **substitution action** $t[X\mapsto s]$ which does not avoid capture, makes this formal. Therefore X behaves differently from 'free variables' of systems such as HRSs [19] or meta-variables of CRSs [18].
- $[a]X$ denotes 'X with a abstracted in X'. We do *not* work modulo α-equivalence and $[a]X$ and $[b]X$ are not equal in any sense; for example $([a]X)[X\mapsto a] \equiv [a]a$ and $([b]X)[X\mapsto a] \equiv [b]a$, and we certainly expect from the intuition '$\lambda x.x$' and '$\lambda x.y$' that $[a]a$ and $[b]a$ should not be equal. Therefore atoms in nominal terms also behave differently from 'bound variables' of systems such as HRSs, ERSs and CRSs.
  We call occurrences of a **abstracted** when they are in the scope of $[a]$-, and otherwise we may call them **unabstracted**.
- $(a\ b)\cdot X$ represents 'X with a and b swapped'. So $\pi\cdot[a]s \equiv [\pi\cdot a]\pi\cdot s$, and $(a\ b)\cdot[a][b]X \equiv [b][a](a\ b)\cdot X$. Therefore this permutation action is distinct from De Bruijn's transformers and other explicit substitutions, which avoid capture as they distribute under abstractions, and which do not satisfy the same simple algebraic laws [7].

We now come to some of the more technical machinery which gives nominal terms their power.

We call $a\#t$ a **freshness** constraint, and $s \approx_\alpha t$ an **equality** constraint. We will use letters P, Q to range over constraints. We introduce a notion of derivation as follows (below a, b denote two different atoms):

$$\frac{}{a\#b} \quad \frac{a\#s}{a\#fs} \quad \frac{a\#s_i \quad (1 \le i \le n)}{a\#(s_1,\ldots,s_n)} \quad \frac{}{a\#[a]s} \quad \frac{a\#s}{a\#[b]s} \quad \frac{\pi^{-1}(a)\#X}{a\#\pi{\cdot}X} \quad \frac{s{\approx_\alpha}t}{fs{\approx_\alpha}ft}$$

$$\frac{}{a{\approx_\alpha}a} \quad \frac{s{\approx_\alpha}t}{[a]s{\approx_\alpha}[a]t} \quad \frac{s{\approx_\alpha}(a\ b){\cdot}t \quad a\#t}{[a]s{\approx_\alpha}[b]t} \quad \frac{ds(\pi,\pi')\#X}{\pi{\cdot}X{\approx_\alpha}\pi'{\cdot}X} \quad \frac{s_i{\approx_\alpha}t_i \quad (1 \le i \le n)}{(s_1,\ldots,s_n){\approx_\alpha}(t_1,\ldots,t_n)}$$

Here we write π^{-1} (the **inverse** of π) for the permutation obtained by reversing the order of the list of swappings; for example $((a\ b) \circ (c\ d))^{-1} = (c\ d) \circ (a\ b)$. Here $ds(\pi,\pi') \equiv \{n \mid \pi(n) \neq \pi'(n)\}$ is the **difference set**. For example $ds((a\ b),\mathbf{Id}) = \{a,b\}$ so $(a\ b){\cdot}X \approx_\alpha X$ follows from $a\#X$ and $b\#X$. Also $[a]a \approx_\alpha [b]b$ and $[a]c \approx_\alpha [b]c$ but not $[a]c \approx_\alpha [a]a$; this is what we would expect.

Write Δ, ∇ for sets of freshness constraints of the form $a\#X$ and call these **freshness contexts**. We write Pr for an arbitrary set of freshness and equality constraints; we may call Pr a **problem**. Substitutions act on constraints and problems in the natural way. Write $\Delta \vdash P$ when a derivation of P exists using elements of Δ as assumptions, and extend this notation elementwise to $\Delta \vdash Pr$ for deducibility of all $P \in Pr$ (see [21, 11] for algorithms to check constraints). For example, $\Delta \vdash \nabla\theta$ means that the constraints obtained by applying the substitution θ to each term in ∇ can be derived from Δ. We will use this notation in the definition of matching in Section 4.1.

The following result is one of the main technical correctness properties of nominal terms, and we should compare it with Theorem 8.

Theorem 1 *If $\Delta \vdash a\#s$ and $\Delta \vdash s \approx_\alpha t$ then $\Delta \vdash a\#t$.*

Proofs, and further properties of nominal terms, are in [21, 12].

3 Typing

3.1 Types and type schemes

We consider denumerable sets of
- **base data types** (write a typical element δ), e.g. $\mathbb{N}$ is a base data type for numbers;
- **type variables** (write a typical variable α);
- **type-formers** (write a typical element C), e.g. *List* is a type-former.

Definition 1. ***Types*** *τ,* ***type-schemes*** *σ, and* ***type-declarations*** *(or* ***arities****) ρ are defined by:*

$$\tau ::= \delta \mid \alpha \mid \tau_1 \times \ldots \times \tau_n \mid C\ \tau \mid [\tau']\tau \qquad \sigma ::= \forall\overline{\alpha}.\tau \qquad \rho ::= (\tau')\tau$$

$\overline{\alpha}$ denotes any finite set of type variables (if empty we omit the $\forall$ entirely); we call them ***bound*** *in σ and call* ***free*** *any type variables mentioned in σ and not in $\overline{\alpha}$. We write $TV(\tau)$ for the set of type variables in τ, and $\equiv$ for* ***equality*** *modulo α-equivalence for bound variables.*[1]

[1] We could express types too using nominal syntax. We use the standard informal treatment because we focus on the term language in this paper.

We call $\tau_1 \times \ldots \times \tau_n$ a **product** and $[\tau]\tau'$ an **abstraction type**. We say that $C\ \tau$ is a **constructed type**, and we associate a type declaration to each term-former. For example, we can have $succ : (\mathbb{N})\mathbb{N}$ and $0 : ()\mathbb{N}$ (we may write just $0 : \mathbb{N}$ in this case).

Basic type substitution $\tau[\alpha \mapsto \tau']$ is the usual inductive replacement of τ' for every α in τ; base cases are $\alpha[\alpha \mapsto \tau] \equiv \tau$ and $\alpha[\beta \mapsto \tau] \equiv \alpha$. We extend the action elementwise to arities ρ, for example $((\tau)\tau')[\alpha \mapsto \tau''] \equiv \big(\tau[\alpha \mapsto \tau'']\big)(\tau'[\alpha \mapsto \tau''])$, and to type-schemes σ in the usual capture-avoiding manner. For example:

$$([\alpha]\alpha)[\alpha \mapsto \tau] \equiv [\tau]\tau \quad (\forall\beta.[\beta]\alpha)[\alpha \mapsto \beta] \equiv \forall\beta'.[\beta']\beta \quad ((\alpha \times \beta)\alpha)[\alpha \mapsto \beta] \equiv (\beta \times \beta)\beta$$

Type substitutions S, T, U are defined by $S ::= \mathbf{Id} \mid S[\alpha \mapsto \tau]$ where $\mathbf{Id}$ is the **identity** substitution: $\tau\mathbf{Id} \equiv \tau$ by definition (we also use $\mathbf{Id}$ for the identity substitution on terms, but the context will always indicate which one we need). S has an action on types τ, schemes σ, and arities ρ, given by the action of $\mathbf{Id}$ and by extending the basic action of the last paragraph. We write application on the right as τS, and write composition of substitutions just as SS', meaning apply S then apply S'. The **domain** of a substitution S, denoted $dom(S)$, is the set of type variables α such that $\alpha S \neq \alpha$.

Substitutions are partially ordered by instantiation. Write $\mathrm{mgu}(\tau, \tau')$ (**most general unifier**) for a least element S such that $\tau S \equiv \tau' S$ (if one exists). We refer the reader to [1] for detailed definitions and algorithms for calculating mgu.

Write $\sigma \succcurlyeq \tau$ when $\sigma \equiv \forall\overline{\alpha}.\tau'$ and $\tau' S \equiv \tau$ for some S which instantiates only type variables in $\overline{\alpha}$. τ may contain other type variables; only *bound* type variables in σ may be instantiated, for example $\forall\alpha.(\alpha \times \beta) \succcurlyeq (\beta \times \beta)$ but $(\alpha \times \beta) \not\succcurlyeq (\beta \times \beta)$.

Also write $\rho \succcurlyeq \rho'$ when $\rho S \equiv \rho'$ for some S. In effect all variables in arities are bound, but since they are *all* bound we do not write the $\forall$. For example $(\alpha \times \alpha)\alpha \succcurlyeq (\beta \times \beta)\beta \succcurlyeq (\mathbb{N} \times \mathbb{N})\mathbb{N}$.

The following useful technical result follows by an easy induction:

Lemma 2 *If $\sigma \succcurlyeq \tau$ then $\sigma[\alpha \mapsto \tau'] \succcurlyeq \tau[\alpha \mapsto \tau']$. Also if $\rho \succcurlyeq \rho'$ then $\rho \succcurlyeq \rho'[\alpha \mapsto \tau']$.*

3.2 Typing judgements

A **typing context** Γ is a set of pairs $(a : \sigma)$ or $(X : \sigma)$ subject to the condition that if $(a : \sigma), (a : \sigma') \in \Gamma$ then $\sigma \equiv \sigma'$, similarly for X.

We write $\Gamma, a\colon \sigma$ for Γ **updated with** $(a\colon \sigma)$, where this means either $\Gamma \cup \{(a\colon \sigma)\}$ or $(\Gamma \setminus \{(a\colon \sigma')\}) \cup \{(a\colon \sigma)\}$ as well-formedness demands. Similarly we write $\Gamma, X : \sigma$. For example, if $\Gamma = a : \alpha$ then $\Gamma, a : \beta$ denotes the context $a : \beta$. We say that a (or rather its association with a type) is **overwritten** in Γ. We write ΓS for the typing context obtained by applying S to the types in Γ. Similarly, $\pi \cdot \Gamma$ denotes the context obtained by applying π to the atoms in Γ. $TV(\Gamma)$ denotes the set of type variables occurring free in Γ.

Definition 2. *A **typing judgement** is a tuple $\Gamma; \Delta \vdash s : \tau$ where Γ is a typing context, Δ a freshness context, s a term and τ a type (when Δ is empty we omit the separating ';').*

We inductively define ***derivable typing judgements*** *as follows:*

$$\frac{\sigma \succcurlyeq \tau}{\Gamma, a:\sigma; \Delta \vdash a : \tau} \qquad \frac{\sigma \succcurlyeq \tau \quad \Gamma; \Delta \vdash \pi{\cdot}X : \diamond}{\Gamma, X:\sigma; \Delta \vdash \pi{\cdot}X : \tau} \qquad \frac{\Gamma, a:\tau; \Delta \vdash t : \tau'}{\Gamma; \Delta \vdash [a]t : [\tau]\tau'}$$

$$\frac{\Gamma; \Delta \vdash t_i : \tau_i \quad (1 \leq i \leq n)}{\Gamma; \Delta \vdash (t_1, \ldots, t_n) : \tau_1 \times \ldots \times \tau_n} \qquad \frac{\Gamma; \Delta \vdash t : \tau' \quad f : \rho \succcurlyeq (\tau')\tau}{\Gamma; \Delta \vdash ft : \tau}$$

Here $\Gamma; \Delta \vdash \pi{\cdot}X : \diamond$ holds if, for any a such that $\pi{\cdot}a \neq a$, $\Delta \vdash a\#X$ or $a{:}\,\sigma$, $\pi{\cdot}a{:}\,\sigma \in \Gamma$ for some σ. The condition $f : \rho \succcurlyeq (\tau')\tau$ is shorthand for $f : \rho$ and $\rho \succcurlyeq (\tau')\tau$. The way we set things up, the arity of f is fixed and $\succcurlyeq$ is independent of Γ. In the rule for abstractions the type environment is updated.

We may write '$\Gamma; \Delta \vdash s : \tau$' for '$\Gamma; \Delta \vdash s : \tau$ is derivable'.

To understand the condition in the second rule, note that $\pi{\cdot}X$ represents an unknown term in which π permutes atoms. Unlike non-nominal α-equivalence, α-equivalence on nominal terms exists in the presence of unknowns X for which substitution does not avoid capture, as $([a]X)[X{\mapsto}s] = [a]s$ demonstrates and as the rules $\dfrac{s \approx_\alpha (a\ b){\cdot}t \quad a\#t}{[a]s \approx_\alpha [b]t}$ and $\dfrac{ds(\pi,\pi')\#X}{\pi{\cdot}X \approx_\alpha \pi'{\cdot}X}$ show. Our typing system is designed to be compatible with this relation and so must be sophisticated in its treatment of permutations acting on unknowns. For concreteness take $\pi = (a\ b)$ and any term t. If $\Gamma \vdash t : \tau$ then $\Gamma \vdash (a\ b){\cdot}t : \tau$ should hold when at least one of the following conditions is satisfied:

1. $(a\ b)\Gamma = \Gamma$ so that Γ cannot 'tell the difference between a and b in t'.
2. If a and b occur in t then they are abstracted, so that what Γ says about a and b gets overwritten.

Given Γ, Δ and s, if there exists τ such that $\Gamma; \Delta \vdash s{:}\tau$ is derivable, then we say $\Gamma; \Delta \vdash s$ is **typable**. Otherwise say $\Gamma; \Delta \vdash s$ is **untypable**.

The following are examples of derivable typing judgements:

$$a : \forall\alpha.\alpha, X : \beta \vdash (a, X) : \alpha \times \beta \qquad a : \forall\alpha.\alpha, X : \beta \vdash (a, X) : \beta \times \beta$$

$$\vdash [a]a : [\alpha]\alpha \qquad a : \beta \vdash [a]a : [\alpha]\alpha \qquad \vdash [a]a : [\zeta]\zeta$$

$$a : \alpha, b : \alpha, X : \tau \vdash (a\ b){\cdot}X : \tau \qquad X : \tau; a\#X, b\#X \vdash (a\ b){\cdot}X : \tau$$

$$X : \tau, a : \alpha, b : \alpha \vdash [a]((a\ b){\cdot}X, b) : [\alpha](\tau \times \alpha)$$

$$a : \alpha, b : \beta \vdash (\ [a](a,b),\ [a][b](a,b),\ a,\ b,\ [a][a](a,b)) : [\alpha](\alpha \times \beta) \times\ [\alpha][\beta](\alpha \times \beta) \times\ \alpha \times\ \beta \times\ [\alpha][\alpha](\alpha \times \beta).$$

$$a : \alpha, b : \beta \vdash (\ [a](a,b),\ [a][b](a,b),\ a,\ b,\ [a]([a](a,b),a)) : [\alpha_1](\alpha_1 \times \beta) \times\ [\alpha_2][\beta_2](\alpha_2 \times \beta_2) \times\ \alpha \times\ \beta \times\ [\alpha_3]([\alpha_4](\alpha_4 \times \beta), \alpha_3).$$

The first line of examples just illustrates some basic typings, and the use of $\succcurlyeq$. The second line illustrates typing abstractions and how this overwrites in Γ. The third line illustrates how permutations are typed. The last three illustrate interactions between typing and (multiple) abstractions and free occurrences of atoms. Note that $a : \alpha, X : \tau \not\vdash (a\ b){\cdot}X : \tau$ and $a : \alpha, X : \tau; b\#X \not\vdash (a\ b){\cdot}X : \tau$.

Lemma 3 *If* $\Gamma; \Delta \vdash t : \tau$ *then* $\Gamma[\alpha{\mapsto}\tau']; \Delta \vdash t : \tau[\alpha{\mapsto}\tau']$.

Proof. By induction on the derivation, using Lemma 2 for the side-conditions.

Lemma 4 *If $\Gamma;\Delta \vdash t{:}\tau$ and $a{,}b{:}\sigma \in \Gamma$ for some σ, then $(a\ b)\cdot\Gamma;\Delta \vdash (a\ b)\cdot t{:}\tau$.*

Proof. By induction on the type derivation: The case for atoms is trivial. In the case of a variable, the side condition holds since a and b have the same type in Γ. The other cases follow directly by induction.

3.3 Principal types

Definition 3. *A **typing problem** is a triple (Γ, Δ, s), written: $\Gamma;\Delta \vdash s{:}?$. A **solution** to $\Gamma;\Delta \vdash s :?$ is a pair (S,τ) of a type-substitution S and a type τ such that $\Gamma S;\Delta \vdash s{:}\tau$. We write $S|_\Gamma$ for the restriction of S to $TV(\Gamma)$.*

For example, solutions to $X{:}\alpha, a{:}\beta, b{:}\beta \vdash (a\,b)\cdot X{:}?$ are $(\mathbf{Id}, \alpha)$ and $([\alpha{\mapsto}\mathbb{N}], \mathbb{N})$. Note that a solution may instantiate type-variables in Γ.

Solutions have a natural ordering given by instantiation of substitutions:

$$(S,\tau) \leq (S',\tau') \quad \text{when} \quad \exists S''.S' \equiv SS'' \wedge \tau' \equiv \tau S'';$$

we call (S',τ') an *instance* of (S,τ) using S''. A minimal element in a set of solutions is called a **principal** solution. By our definitions there may be many principal solutions; $(\mathbf{Id}, \alpha)$ and $(\mathbf{Id}, \beta)$ are both principal for $X : \forall\alpha.\alpha \vdash X :?$. As in the case of most general unifiers, principal solutions for a typable $\Gamma;\Delta \vdash s$ are unique modulo renamings of type-variables. In a moment we show that the following algorithm calculates principal solutions:

Definition 4. *The partial function $pt(\Gamma;\Delta \vdash s)$ is defined inductively by:*

- *$pt(\Gamma, a{:}\forall\overline{\alpha}.\tau;\Delta \vdash a) = (\boldsymbol{Id}, \tau)$, where $\alpha \in \overline{\alpha}$ are assumed fresh (not in Γ) without loss of generality.*
- *$pt(\Gamma, X{:}\forall\overline{\alpha}.\tau;\Delta \vdash \pi\cdot X) = (S, \tau S)$ (again $\alpha \in \overline{\alpha}$ are assumed fresh) provided that for each a in π such that $a \neq \pi\cdot a$, we have $\Delta \vdash a\#X$, or otherwise $a{:}\,\sigma, \pi\cdot a{:}\,\sigma' \in \Gamma$ for some σ, σ' that are unifiable. The substitution S is the mgu of those pairs, or **Id** if all such a are fresh for X.*
- *$pt(\Gamma;\Delta \vdash (t_1,\ldots,t_n)) = (S_1 \ldots S_n, \phi_1 S_2 \ldots S_n \times \ldots \times \phi_{n-1} S_n \times \phi_n)$ where $pt(\Gamma;\Delta \vdash t_1) = (S_1,\phi_1)$, $pt(\Gamma S_1;\Delta \vdash t_2) = (S_2,\phi_2)$, $\ldots$, $pt(\Gamma S_1 \ldots S_{n-1};\Delta \vdash t_n) = (S_n,\phi_n)$.*
- *$pt(\Gamma;\Delta \vdash ft) = (SS', \phi S')$ where $pt(\Gamma;\Delta \vdash t) = (S,\tau)$, $f : \rho \equiv (\phi')\phi$ where the type variables in ρ are chosen distinct from those in Γ and τ, and $S' = mgu(\tau,\phi')$.*
- *$pt(\Gamma;\Delta \vdash [a]s) = (S|_\Gamma, [\tau']\tau)$ where $pt(\Gamma, a{:}\alpha;\Delta \vdash s) = (S,\tau)$, α is chosen fresh, $\alpha S = \tau'$.*
  Here $\Gamma, a{:}\alpha$ denotes Γ with any type information about a overwritten to $a{:}\alpha$, as discussed in Subsection 3.2.

$pt(\Gamma;\Delta \vdash s)$ may be undefined; Theorem 5 proves that s is untypable in the environment $\Gamma;\Delta$.

The definition above generalises the Hindley-Milner system [9] for the λ-calculus with arbitrary function symbols f, as is standard in typing algorithms for rewrite systems [2], and with atoms and variables (representing 'unknown terms') with suspended atoms-permutations.

The treatment of typing for atoms, abstraction and moderated variables is new; for example if $\Gamma = X\colon \tau, a\colon \alpha, b\colon \alpha$, then $\Gamma \vdash [a](a\ b)\cdot X\colon [\alpha]\tau$ but not $\Gamma \vdash [a](a\ b)\cdot X\colon [\beta]\tau$. However, $\Gamma \vdash [a]X : [\beta]\tau$ as expected.

$pt(\Gamma; \Delta \vdash s)$ gives a static semantics in the form of a most general type to s, given a typing of atoms and variables mentioned in s, and information about what atoms may be forbidden from occurring in some variables.

Theorem 5 *1. If $pt(\Gamma; \Delta \vdash s)$ is defined and equal to (S, τ) then $\Gamma S; \Delta \vdash s : \tau$ is derivable.*
2. Let U be a substitution such that $dom(U) \subseteq TV(\Gamma)$. If $\Gamma U; \Delta \vdash s : \mu$ is derivable then $pt(\Gamma; \Delta \vdash s)$ is defined and (U, μ) is one of its instances.

That is: (1) "$pt(\Gamma; \Delta \vdash s)$ solves $(\Gamma; \Delta \vdash s{:}?)$", and (2) "any solution to $(\Gamma; \Delta \vdash s{:}?)$ is an instance of $pt(\Gamma; \Delta \vdash s)$".

Proof. The first part is by a routine induction on the derivation of $pt(\Gamma; \Delta \vdash s) = (S, \tau)$ (using Lemma 3), which we omit. The second part is by induction on the syntax of s:

- Suppose $\Gamma U; \Delta \vdash \pi\cdot X : \mu$. Examining the typing rules, we see that $X\colon \forall\overline{\alpha}.\tau \in \Gamma$ and $X : \forall\overline{\alpha}.\tau U \in \Gamma U$, that $\Gamma U; \Delta \vdash \pi\cdot X : \diamond$, and that $\mu = \tau U S$ for some S acting only on $\overline{\alpha}$ (U acts trivially on $\overline{\alpha}$ because we assume $\overline{\alpha}$ was chosen fresh). Hence, for each a such that $a \neq \pi\cdot a$, we have $\Delta \vdash a\#X$, or, for some σ, $a\colon \sigma, \pi\cdot a\colon \sigma \in \Gamma U$, that is, $a\colon \sigma_1, \pi\cdot a\colon \sigma_2 \in \Gamma$. Let V be the mgu of all such pairs σ_1, σ_2, so $U = VS'$ (we take $V = \mathbf{Id}$ if there are no pairs to consider). Thus, $pt(\Gamma; \Delta \vdash \pi\cdot X) = (V, \tau V)$. Therefore, $(U, \mu) = (VS'S, \tau VS'S)$.
- Suppose $\Gamma U; \Delta \vdash a : \mu$. Clearly a is typable in the context $\Gamma; \Delta$ so (examining the typing rules) $a : \forall\overline{\alpha}.\tau$ must occur in Γ and $pt(\Gamma; \Delta \vdash a) = (\mathbf{Id}, \tau)$. It is now not hard to satisfy ourselves that $(\mathbf{Id}, \tau)$ is principal.
- Suppose $\Gamma U; \Delta \vdash [a]t{:}\mu$. This may happen if and only if $\mu \equiv [\mu']\mu''$ and $(U[\alpha \mapsto \mu'], \mu'')$ solves $\Gamma, a : \alpha; \Delta \vdash t{:}?$, where α is fresh for Γ. By inductive hypothesis $(U[\alpha \mapsto \mu'], \mu'')$ is an instance of $(S, \tau) = pt(\Gamma, a : \alpha; \Delta \vdash t)$, that is, there is a substitution U_a such that $U[\alpha \mapsto \mu'] = SU_a$, $\mu'' = \tau U_a$. By definition, $pt(\Gamma; \Delta \vdash [a]t) = (S|_\Gamma, [\tau']\tau)$ where $\tau' = \alpha S$. So $(SU_a)|_\Gamma = U$ and $\alpha SU_a = \mu'$. Therefore $(U, [\mu']\mu'')$ is an instance of $(S|_\Gamma, [\alpha S]\tau)$.
- The cases for $(t_1, \ldots, t_n)$ and ft are long, but routine.

Corollary 6 *$\Gamma; \Delta \vdash s$ is typable if and only if $pt(\Gamma; \Delta \vdash s){=}(\boldsymbol{Id}, \tau)$ for some τ.*

3.4 α-equivalence and types

We now prove that α-equivalence respects types; so our static semantics correctly handles swappings and variables X whose substitution can move atoms into new abstracted (typing) contexts. We need some definitions: Given two type contexts

Γ and Γ', write Γ,Γ' for that context obtained by updating Γ with typings in Γ', overwriting the typings in Γ if necessary. For example if $\Gamma = a\colon \alpha$ and $\Gamma' = a\colon \beta, b\colon \beta$ then $\Gamma,\Gamma' = a\colon \beta, b\colon \beta$. If Γ and Γ' mention disjoint sets of atoms and variables (we say they are **disjoint**) then Γ,Γ' is just a set union.

Lemma 7 *1. If $\Gamma;\Delta \vdash s : \tau$ then $\Gamma,\Gamma';\Delta \vdash s : \tau$, provided that Γ' and Γ are disjoint. Call this **type weakening**.*
*2. If $\Gamma, a : \tau';\Delta \vdash s : \tau$ then $\Gamma;\Delta \vdash s : \tau$ provided that $\Delta \vdash a\#s$. Call this **type strengthening** (for atoms).*
*3. If $\Gamma, X : \tau';\Delta \vdash s : \tau$ then $\Gamma;\Delta \vdash s : \tau$ provided X does not occur in s. Call this **type strengthening** (for variables).*

Proof. By induction on the derivation. For the second part, if a occurs in s under an abstraction, then $a : \tau'$ is overwritten whenever a is used.

Theorem 8 *$\Delta \vdash s \approx_\alpha t$ and $\Gamma;\Delta \vdash s : \tau$ imply $\Gamma;\Delta \vdash t : \tau$.*

Proof. By induction on the size of (s,t). The form of t is rather restricted by our assumption that $\Delta \vdash s \approx_\alpha t$ — for example if $s \equiv \pi{\cdot}X$ then $t \equiv \pi'{\cdot}X$ for some π'. We use this information without commment in the proof.

- Suppose $\Delta \vdash a \approx_\alpha a$ and $\Gamma;\Delta \vdash a : \tau$. There is nothing to prove.
- Suppose $\Delta \vdash [a]s \approx_\alpha [a]t$ and $\Gamma;\Delta \vdash [a]s : \tau$. Then $\Delta \vdash s \approx_\alpha t$, and $\tau \equiv [\tau']\tau''$, and $\Gamma, a : \tau';\Delta \vdash s : \tau''$. We use the inductive hypothesis to deduce that $\Gamma, a : \tau';\Delta \vdash t : \tau''$ and concluce that $\Gamma;\Delta \vdash [a]t : [\tau']\tau''$.
- Suppose $\Delta \vdash [a]s \approx_\alpha [b]t$ and $\Gamma;\Delta \vdash [a]s : \tau$. Then by properties of $\approx_\alpha$ [21, Theorem 2.11] we know $\Delta \vdash s \approx_\alpha (a\ b){\cdot}t, a\#t, (a\ b){\cdot}s \approx_\alpha t, b\#s$. Also $\tau \equiv [\tau']\tau''$ and $\Gamma, a : \tau';\Delta \vdash s : \tau''$. By Lemma 7 (Weakening) also $\Gamma, a : \tau', b : \tau';\Delta \vdash s : \tau''$. By equivariance (Lemma 4) $(a\ b){\cdot}\Gamma, b : \tau', a : \tau';\Delta \vdash (a\ b){\cdot}s : \tau''$. Since $\Delta \vdash (a\ b){\cdot}s \approx_\alpha t$, by inductive hypothesis $(a\ b){\cdot}\Gamma, b : \tau', a : \tau';\Delta \vdash t : \tau''$.
  Now note that $(a\ b){\cdot}\Gamma, b : \tau', a : \tau' = \Gamma, b : \tau', a : \tau'$ (because any data Γ has on a and b is overwritten). So $\Gamma, b : \tau', a : \tau';\Delta \vdash t : \tau''$. We conclude that $\Gamma, b : \tau', a : \tau';\Delta \vdash [b]t : [\tau']\tau''$. Since $\Delta \vdash a\#[b]t$ and $\Delta \vdash b\#[b]t$ by Lemma 7 (strengthening for atoms) we have $\Gamma;\Delta \vdash [b]t : [\tau']\tau''$.
- Suppose $\Delta \vdash \pi{\cdot}X \approx_\alpha \pi'{\cdot}X$ and suppose $\Gamma;\Delta \vdash \pi{\cdot}X : \tau$. Then $\Delta \vdash ds(\pi,\pi')\#X$, and $X : \sigma \in \Gamma$, and $\sigma \succcurlyeq \tau$, and $\Gamma;\Delta \vdash \pi{\cdot}X : \diamond$.
  $\Delta \vdash ds(\pi,\pi')\#X$ so $\Gamma;\Delta \vdash \pi'{\cdot}X : \diamond$ and $\Gamma;\Delta \vdash \pi'{\cdot}X : \tau$ follows.
- The cases of ft and $(t_1,\ldots,t_n)$ follow easily.

Corollary 9 *$\Delta \vdash s \approx_\alpha t$ implies $pt(\Gamma;\Delta \vdash s) = pt(\Gamma;\Delta \vdash t)$ modulo renamings of type variables, for all Γ,Δ such that either side of the equality is defined.*

4 Rewriting

We now consider how to make a notion of nominal rewriting (i.e., a theory of general directed equalities on nominal terms [12]) interact correctly with our type system. We start with some definitions taken from [12].

A **nominal rewrite rule** $R \equiv \nabla \vdash l \to r$ is a tuple of a freshness context ∇, and terms l and r such that $V(r,\nabla) \subseteq V(l)$. Write $R^{(a\,b)}$ for that rule obtained by swapping a and b in R throughout. For example, if $R \equiv b\#X \vdash [a]X \to (b\ a)\cdot X$ then $R^{(a\,b)} \equiv a\#X \vdash [b]X \to (a\ b)\cdot X$. Let $R^{\mathbf{Id}} \equiv R$ and $R^{(a\ b)\circ\pi} = (R^{(a\ b)})^{\pi}$. Let $\mathcal{R}$ range over (possibly infinite) sets of rewrite rules. Call $\mathcal{R}$ **equivariant** when if $R \in \mathcal{R}$ then $R^{(a\ b)} \in \mathcal{R}$ for all distinct atoms a, b. A **nominal rewrite system** is an equivariant set of nominal rewrite rules.

Say a term s has a **position at** X when it mentions X once, with the permutation **Id**. We may call X a **hole**. Write $s[s']$ for $s[X{\mapsto}s']$.[2] We would usually call s a 'context' but we have already used this word so we will avoid it. For example, X and $[a]X$ have positions at X, but (X,X) and $(a\ b)\cdot X$ do not. We may make X nameless and write it just -.

Definition 5. *Suppose $pt(\Phi;\nabla \vdash l) = (\boldsymbol{Id},\tau)$ and suppose that Φ mentions no type-schemes. All the recursive calls involved in calculating $pt(\Phi;\nabla \vdash l)$ have the form $pt(\Phi,\Phi';\nabla \vdash l')$ where l' is a subterm of l and Φ' contains only type assumptions for atoms. We will call recursive calls of the form $pt(\Phi,\Phi';\nabla \vdash \pi\cdot X) = (S,\tau')$ **variable typings** of $\Phi;\nabla \vdash l : \tau$.*

Note that there is one variable typing for each occurrence of a variable in l, and they are uniquely defined modulo renaming of type variables. Also, S may affect Φ' but not Φ since we assume that $pt(\Phi;\nabla \vdash l) = (\mathbf{Id},\tau)$.

Definition 6. *Let $pt(\Phi,\Phi';\nabla \vdash \pi\cdot X) = (S,\tau')$ be a variable typing of $\Phi;\nabla \vdash l : \tau$, and let Φ'' be the subset of Φ' such that $\nabla \vdash A(\Phi'\backslash\Phi'')\#\pi\cdot X$. We call $\Phi,\Phi''S;\nabla \vdash \pi\cdot X : \tau'$ an **essential typing** of $\Phi;\nabla \vdash l : \tau$.*

So the essential typings of $a\colon\alpha, b\colon\alpha, X\colon\tau \vdash ((a\ b)\cdot X,[a]X)\colon\tau\times[\alpha']\tau$ are:

$$a\colon\alpha, b\colon\alpha, X\colon\tau \vdash (a\ b)\cdot X\colon\tau \quad \text{and} \quad b\colon\alpha, a\colon\alpha', X\colon\tau \vdash X\colon\tau.$$

The essential typings of $a\colon\alpha, b\colon\alpha, X\colon\tau; a\#X \vdash ((a\ b)\cdot X,[a]X)\colon\tau\times[\alpha']\tau$ are:

$$b\colon\alpha, X\colon\tau \vdash (a\ b)\cdot X\colon\tau \quad \text{and} \quad b\colon\alpha, X\colon\tau \vdash X\colon\tau.$$

We will talk about the **typing at a position in a term**; for example the typing at Z in $(Z,[a]X)[Z \mapsto (a\ b)\cdot X]$ in the first example above (with hole named Z) is $a : \alpha, b : \alpha, X : \tau \vdash (a\ b)\cdot X : \tau$.

4.1 Matching problems

Definition 7. *A **(typed) matching problem** $(\Phi;\nabla \vdash l) \stackrel{?}{\approx} (\Gamma;\Delta \vdash s)$ is a pair of tuples (Φ and Γ are typing contexts, ∇ and Δ are freshness contexts, l and s are terms) such that the variables and type-variables mentioned on the left-hand side are disjoint from those mentioned in Γ,Δ,s, and such that Φ mentions no atoms or type-schemes.*

[2] Here 'positions' are based on substitution and not paths in the abstract syntax tree as in [10]. The equivalence is immediate since substitution is grafting.

Below, l will be the left-hand side of a rule and s will be a term to reduce. The condition that Φ mentions no atoms or type-schemes may seem strong, but is all we need: we give applications in Section 4.3.

Intuitively, we want to make the term on the left-hand side of the matching problem α-equivalent to the term on the right-hand side. Formally, a **solution** to this matching problem, if it exists, is the least pair (S, θ) of a type- and term-substitution (the ordering on substitutions extends to pairs component-wise) such that:

1. $X\theta \equiv X$ for $X \notin V(\Phi,\nabla,l)$, $\alpha S \equiv \alpha$ for $\alpha \notin TV(\Phi)$[3], $\Delta \vdash l\theta \approx_\alpha s$ and $\Delta \vdash \nabla\theta$.
2. $pt(\Phi; \nabla \vdash l) = (\mathbf{Id}, \tau)$, $pt(\Gamma; \Delta \vdash s) = (\mathbf{Id}, \tau S)$, and for each $\Phi, \Phi'; \nabla \vdash \pi{\cdot}X : \phi'$ an essential typing in $\Phi; \nabla \vdash l : \tau$, we have $\Gamma, (\Phi' S); \Delta \vdash (\pi{\cdot}X)\theta : \phi' S$.

The first condition defines a matching solution for untyped nominal terms (see [21, 12] for more details on untyped nominal matching algorithms). The last condition enforces type consistency: the terms should have compatible types, and the solution should instantiate the variables in a way that is compatible with the typing assumptions. When the latter holds, we say that (S, θ) **respects the essential typings** of $\Phi; \nabla \vdash l : \tau$ in the context $\Gamma; \Delta$.

For example, $(X{:}\alpha \vdash X) \;{}_?{\approx}\; (a{:}\mathbb{B} \vdash a)$ has solution $([\alpha{\mapsto}\mathbb{B}], [X{\mapsto}a])$, whereas $(X{:}\mathbb{B} \vdash X) \;{}_?{\approx}\; (a{:}\alpha \vdash a)$ has no solution — α on the right is too general.

To see why we need to check θ in the second condition, consider the term $g(f\ True)$ where $g : (\alpha)\mathbb{N}$ and $f : (\beta)\mathbb{N}$, that is both functions are polymorphic, and produce a result of type $\mathbb{N}$. Then the untyped matching problem $g(f\ X)) \;{}_?{\approx}\; g(f\ True)$ has a solution $(\mathbf{Id}, \{X \mapsto True\})$, but the typed matching problem $(X : \mathbb{N} \vdash g(f\ X)) \;{}_?{\approx}\; (\vdash g(f\ True))$ has none: $\{X \mapsto True\}$ fails the last condition since X is required to have type $\mathbb{N}$ but it is instantiated with a boolean.

4.2 Typed rewriting

Definition 8. *A **(typed) rewrite rule** $R \equiv \Phi; \nabla \vdash l \to r : \tau$ is a tuple of a type context Φ which only types the variables in l and has no type-schemes (in particular, Φ mentions no atoms), a freshness context ∇, and terms l and r such that*

1. *$V(r, \nabla, \Phi) \subseteq V(l)$,*
2. *$pt(\Phi; \nabla \vdash l) = (\mathbf{Id}, \tau)$ and $\Phi; \nabla \vdash r : \tau$.*
3. *Essential typings of $\Phi; \nabla \vdash r : \tau$ are also essential typings of $\Phi; \nabla \vdash l : \tau$.*

The first condition is standard. The second condition says that l is typable using Φ and ∇, and r is typable with a type *at least* as general. The third condition ensures a consistency best explained by violating it: Let $f : ([\alpha]\mathbb{N})\mathbb{N}$, then $X{:}\,\mathbb{N} \vdash f([a]X) \to X{:}\,\mathbb{N}$ passes the first two conditions, but fails the third because in the right-hand side we have an essential typing $X{:}\,\mathbb{N} \vdash X{:}\,\mathbb{N}$ whereas in the left-hand side we have $X{:}\,\mathbb{N}, a{:}\,\alpha \vdash X{:}\,\mathbb{N}$. For comparison, $X{:}\,\mathbb{N} \vdash g([a]X) \to [a]X{:}\,[\alpha]\mathbb{N}$ with $g : ([\alpha]\mathbb{N})[\alpha]\mathbb{N}$ passes all three conditions and is a valid rewrite rule, as well as $X{:}\,\mathbb{N};\ a\#X \vdash f([a]X) \to X$.

[3] So in particular, by the side-conditions on variables being disjoint between left and right of the problem, $X\theta \equiv X$ for $X \in V(\Gamma, \Delta, s)$ and $\alpha S \equiv \alpha$ for $\alpha \in TV(\Gamma)$.

The rewrite relation is defined on terms-in-context: Take $\Gamma;\Delta \vdash s$ and $\Gamma;\Delta \vdash t$, and a rule $R \equiv \Phi;\nabla \vdash l \to r : \tau$, such that $V(R) \cap V(\Gamma,\Delta,s,t) = \emptyset$, and $TV(R) \cap TV(\Gamma) = \emptyset$ (renaming variables in R if necessary). Assume $\Gamma;\Delta \vdash s$ is typable: $pt(\Gamma;\Delta \vdash s) = (\mathbf{Id},\mu)$, $s \equiv s''[s']$ and $\Gamma';\Delta \vdash s' : \mu'$ is the typing of s' at the corresponding position. We say that s **rewrites with** R **to** t **in the context** $\Gamma;\Delta$ and write $\Gamma;\Delta \vdash s \xrightarrow{R} t$ when:

1. $(\Phi;\nabla \vdash l) \stackrel{?}{\approx} (\Gamma';\Delta \vdash s')$ has solution (S,θ).
2. $\Delta \vdash s''[r\theta] \approx_\alpha t$.

These conditions are inherited from nominal rewriting [10, 12] and extended with types. Instantiation of X does not avoid capture, so an atom a introduced by a substitution may appear under different abstractions in different parts of a term. We must pay attention to the typing *at the position of the variable* in the rewrite; essential typings do just this. For example if $f : (\tau_1)\tau$, $g : (\tau)[\alpha]\tau$ and $R \equiv X\colon\tau \vdash gX \to [a]X : [\alpha]\tau$ then $pt(X\colon\tau \vdash gX) = (\mathbf{Id},[\alpha]\tau)$ and $X\colon\tau \vdash [a]X : [\alpha]\tau$. R satisfies the first two conditions in the definition of typed rule but fails the third: the only essential typing in the left-hand side is $X\colon\tau \vdash X\colon\tau$, whereas in the right-hand side we have $X\colon\tau, a\colon\alpha \vdash X\colon\tau$. Notice that $a\colon\tau_1 \vdash g(fa) : [\alpha]\tau$ and the typed matching problem $(X\colon\tau \vdash gX) \stackrel{?}{\approx} (a\colon\tau_1 \vdash g(fa))$ has a solution $(\mathbf{Id},\{X \mapsto fa\})$. So, if we ignore the third condition in the definition of typed rule, we have a rewriting step $a\colon\tau_1 \vdash g(fa) \to [a](fa)$ which does not preserve types: $a\colon\tau_1 \vdash g(fa) : [\alpha]\tau$ but $a\colon\tau_1 \not\vdash [a](fa) : [\alpha]\tau$.

We need a lemma to prove Subject Reduction (Theorem 11):

Lemma 10 *Suppose that $\Phi;\nabla \vdash r : \tau$, where Φ types only variables in r (it mentions no atoms) and has no type schemes. Let θ be a substitution instantiating all variables in r, and such that (S,θ) respects the essential typings of $\Phi;\nabla \vdash r : \tau$ in the context Γ,Δ, that is, for each essential typing $\Phi,\Phi';\nabla \vdash \pi\cdot X : \tau'$ of $\Phi;\nabla \vdash r : \tau$, it is the case that $\Gamma,\Phi'S;\Delta \vdash (\pi\cdot X)\theta : \tau'S$. Then $\Gamma;\Delta \vdash r\theta : \tau S$.*

Theorem 11 (Subject Reduction) *Let $R \equiv \Phi;\nabla \vdash l \to r : \tau$. If $\Gamma;\Delta \vdash s : \mu$ and $\Gamma;\Delta \vdash s \xrightarrow{R} t$ then $\Gamma;\Delta \vdash t : \mu$.*

Proof. It suffices to prove that if $pt(\Gamma;\Delta \vdash s) = (\mathbf{Id},\nu)$ and $\Gamma;\Delta \vdash s \xrightarrow{R} t$ then $\Gamma;\Delta \vdash t : \nu$. Suppose $\Gamma;\Delta \vdash s \xrightarrow{R} t$. Then (using the notation in the definition of matching and rewriting above) we know that:

1. $s \equiv s''[s']$, $\Delta \vdash l\theta \approx_\alpha s'$, and $\Delta \vdash \nabla\theta$.
2. θ acts nontrivially only on the variables in R, not those in Γ,Δ,s.
3. Assuming $\Gamma';\Delta \vdash s' : \nu'$ is the typing of s', then $\Gamma',\Phi'S;\Delta \vdash (\pi\cdot X)\theta : \phi'S$ for each essential typing $\Phi,\Phi';\nabla \vdash \pi\cdot X : \phi'$ in $\Phi;\nabla \vdash l : \tau$ (therefore also for each essential typing in $\Phi;\nabla \vdash r : \tau$ since they are a subset).
4. $pt(\Phi;\nabla \vdash l) = (\mathbf{Id},\tau)$ and $pt(\Gamma',\Delta \vdash s') = (\mathbf{Id},\tau S)$ so there is some S' such that $\Gamma'S' = \Gamma'$ and $\tau SS' = \nu'$.
5. $\Delta \vdash s''[r\theta] \approx_\alpha t$.

By Theorem 8, from 3, 4 and 1 we deduce $\Gamma';\Delta \vdash l\theta : \tau SS'$. Since $pt(\Phi;\nabla \vdash l) = (\mathbf{Id},\tau)$, by our assumptions on rewrite rules also $\Phi;\nabla \vdash r : \tau$, and by Lemma 3

also $\Phi SS'; \nabla \vdash r : \tau SS'$. By Lemma 10, $\Gamma'; \Delta \vdash r\theta : \tau S$. Since $\Gamma' S' = \Gamma'$, by Lemma 3 also $\Gamma'; \Delta \vdash r\theta : \tau SS'$. Hence $\Gamma; \Delta \vdash s''[r\theta] : \nu$ as required.

4.3 Examples

Untyped λ-calculus. Suppose a type Λ and term-formers $lam : ([\Lambda]\Lambda)\Lambda$, $app : (\Lambda \times \Lambda)\Lambda$, and $sub : ([\Lambda]\Lambda \times \Lambda)\Lambda$, sugared to $\lambda[a]s$, $s\ t$, and $s[a{\mapsto}t]$. Rewrite rules satisfying the conditions in Definition 8 are:

$$X, Y{:}\Lambda \vdash (\lambda[a]X)Y \to X[a{\mapsto}Y]{:}\Lambda \qquad X, Y{:}\Lambda; a\#X \vdash X[a{\mapsto}Y] \to X{:}\Lambda$$
$$Y{:}\Lambda \vdash a[a{\mapsto}Y] \to Y{:}\Lambda \qquad X, Y{:}\Lambda; b\#Y \vdash (\lambda[b]X)[a{\mapsto}Y] \to \lambda[b](X[a{\mapsto}Y]){:}\Lambda$$
$$X, Y, Z{:}\Lambda \vdash (XY)[a{\mapsto}Z] \to X[a{\mapsto}Z]\,Y[a{\mapsto}Z]{:}\Lambda$$

The freshness conditions are exactly what is needed so that no atom moves across the scope of an abstraction (which might change its type). For instance, in rule $X, Y{:}\Lambda; a\#X \vdash X[a{\mapsto}Y] \to X{:}\Lambda$, X is under an abstraction for a in the left-hand side and not in the right-hand side, but we have $a\#X$.

The typed λ-calculus Suppose a type-former $\Rightarrow$ of arity 2 and term-formers $\lambda : ([\alpha]\beta)(\alpha \Rightarrow \beta)$, $\circ : (\alpha \Rightarrow \beta \times \alpha)\beta$, and $\sigma : ([\alpha]\beta \times \alpha)\beta$. Sugar as in the previous example, so, instead of $\sigma([a]s, t)$ we write $s[a{\mapsto}t]$. Then the following rewrite rules satisfy the conditions in Definition 8:

$$X : \alpha, Y : \beta;\ a\#X \vdash X[a{\mapsto}Y] \to X : \alpha \qquad Y : \gamma \vdash a[a{\mapsto}Y] \to Y : \gamma$$
$$X : \alpha \Rightarrow \beta,\ Y : \alpha,\ Z : \gamma \vdash (XY)[a{\mapsto}Z] \to (X[a{\mapsto}Z])(Y[a{\mapsto}Z]) : \beta$$
$$X : \beta,\ Y : \gamma;\ b\#Y \vdash (\lambda[b]X)[a{\mapsto}Y] \to \lambda[b](X[a{\mapsto}Y]) : \alpha \Rightarrow \beta$$

For the β-reduction rule we mention two variants; they give the same rewrites:

$$X{:}[\alpha]\beta, Y{:}\alpha \vdash (\lambda X)Y \to \sigma(X, Y) : \beta \quad X{:}\beta, Y{:}\alpha \vdash (\lambda[a]X)Y \to \sigma([a]X, Y) : \beta$$

Assume types $\mathbb{B}$ and $\mathbb{N}$. Then $B : \mathbb{B},\ N : \mathbb{N} \vdash ((\lambda[a]a)B, (\lambda[a]a)N) : \mathbb{B} \times \mathbb{N}$ and

$$B : \mathbb{B},\ N : \mathbb{N} \vdash ((\lambda[a]a)B, (\lambda[a]a)N) \to (B, N) : \mathbb{B} \times \mathbb{N}.$$

$\lambda[a]a$ takes different types just like $\lambda x.x$ in the Hindley-Milner type system; $pt(\vdash \lambda[a]a) = (\mathbf{Id}, \alpha \Rightarrow \alpha)$. Our system will not type $B : \mathbb{B},\ N : \mathbb{N} \vdash BN$ or $\lambda[a]aa$ — the system for the *untyped* λ-calculus above, types the second term.

Surjective pairing Rewrites for surjective pairing cannot be implemented by a compositional translation to λ-calculus terms [4]. Our system deals with rules defining surjective pairing directly; assume $fst : (\alpha \times \beta)\alpha$ and $snd : (\alpha \times \beta)\beta$:

$$X : \alpha,\ Y : \beta \vdash fst(X, Y) \to X : \alpha \qquad X : \alpha,\ Y : \beta \vdash snd(X, Y) \to Y : \beta$$
$$X : \alpha \times \beta \vdash (fst(X), snd(X)) \to X : \alpha \times \beta$$

Higher-order logic. Extend the typed λ-calculus above with a type $Prop$ and term-formers $\top : Prop$, $\bot : Prop$, $= : (\alpha \times \alpha)Prop$, $\forall : ([\alpha]Prop)Prop$, $\wedge$: $(Prop \times Prop)Prop$, and so on. Rewrite rules include:

$$X : \alpha \vdash X = X \to \top : Prop \qquad X : Prop; a\#X \vdash \forall[a]X \to X : Prop$$
$$X, Y : Prop \vdash \forall[a](X \wedge Y) \to \forall[a]X \wedge \forall[a]Y : Prop$$

Arithmetic Extend further with a type $\mathbb{N}$, term-formers $0 : \mathbb{N}$, $succ : (\mathbb{N})\mathbb{N}$, $+ : (\mathbb{N} \times \mathbb{N})\mathbb{N}$ and $=: (\alpha \times \alpha)Prop$. Observe that $\lambda[a]succ(a)$ has principal type $\mathbb{N} \Rightarrow \mathbb{N}$ whereas $\lambda[a]0$ has principal type $\alpha \Rightarrow \mathbb{N}$. Likewise, $\forall[a](a = 0)$ is typable (with type $Prop$) whereas $\forall[a](\lambda[a]succ(a) = 0)$ is not typable.

5 Conclusions and future work

We have defined a syntax-directed type inference algorithm for nominal terms. It smoothly resolves the tension between the denotational intuition of a type as a set, and the syntactic intuition of a variable in nominal terms as a term which may mention atoms. The algorithm delivers principal types (our function pt). The types produced resemble the Hindley-Milner polymorphic type system for the λ-calculus, but are acting on nominal terms which include variables X representing context holes as well as atoms a representing program variables, and such that the same atom may occur in many different abstraction contexts and thus may acquire different types in different parts of the term.

Theorem 8 proves our types compatible with the powerful notion of α-equivalence inherited from nominal terms [21]. Theorem 11 shows that a notion of typed nominal rewrite rule exists which guarantees preservation of types.

Our system is in Curry style; type annotations on terms are not required. We do rely on type declarations for function symbols (arities) and in future we may investigate inferring the type of a function from its rewrite rules.

Type inference is well-studied for the λ-calculus and Curry-style systems also exist for first-order rewriting systems [2] and algebraic λ-calculi (which combine term rewriting and λ-calculus) [3]. We know of no type assignment system for the standard higher-order rewriting formats (HRSs use a typed metalanguage, and restrict rewrite rules to base types).

Our type system has only rank 1 polymorphism (type-variables are quantified, if at all, only at the top level of the type). It should be possible to consider more powerful systems, for instance using rank 2 polymorphic types, or intersection types [6]. The latter have been successfully used to provide characterisations of normalisation properties of λ-terms. Normalisation of nominal rewriting using type systems is itself a subject for future work, and one of our long-term goals in this work is to come closer to applying logical semantics such as intersection types, to nominal rewriting.

References

1. F. Baader and W. Snyder, Unification Theory, *Handbook of Automated Reasoning*, volume I, chapter 8, 445–532. A. Robinson and A. Voronkov (eds.), Elsevier Science, 2001.
2. S. van Bakel and M. Fernández. Normalization results for typable rewrite systems. *Information and Computation*, 133(2):73–116, 1997.
3. F. Barbanera and M. Fernández. Intersection type assignment systems with higher-order algebraic rewriting. *Theoretical Computer Science*, 170:173–207, 1996.
4. H. P. Barendregt, *Pairing without conventional constraints*, Zeitschrift für mathematischen Logik und Grundlagen der Mathematik **20** (1974), 289–306.
5. H. P. Barendregt, Lambda Calculi With Types, *Handbook of Logic in Computer Science*, Oxford University Press, S. Abramsky, D. Gabbay and T. S. E. Maibaum (eds.), 1992.
6. H. P. Barendregt, M. Coppo and M. Dezani-Ciancaglini, A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.
7. S. Berghofer and C. Urban, A Head-to-Head Comparison of de Bruijn Indices and Names. *LFMTP'06*, 46–59, 2006.
8. H. B. Curry and R. Feys, *Combinatory Logic*, volume 1. North Holland,1958.
9. L. M. M. Damas and R. Milner, *Principal Type Schemes for Functional programs*, Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, 1982.
10. M. Fernández, M. J. Gabbay and I. Mackie, *Nominal Rewriting Systems*, ACM Symposium on Principles and Practice of Declarative Programming (PPDP'04), ACM Press, 2004.
11. M. Fernández and M. J. Gabbay, *Nominal Rewriting with Name Generation: Abstraction vs. Locality*, ACM Symposium on Principles and Practice of Declarative Programming (PPDP'05), ACM Press, 2005.
12. M. Fernández and M. J. Gabbay, Nominal Rewriting, *Information and Computation*, to appear, available from `http://dx.doi.org/10.1016/j.ic.2006.12.002`.
13. M. J. Gabbay and A. M. Pitts, *A New Approach to Abstract Syntax with Variable Binding*. Formal Aspects of Computing, vol. 13, pp. 341–363, 2002.
14. M. J. Gabbay. *A Theory of Inductive Definitions with Alpha-Equivalence*. PhD Thesis, Cambridge University, 2000.
15. J-Y. Girard, The System F of Variable Types, Fifteen Years Later, *Theoretical Computer Science*, 45, 1986.
16. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996.
17. Z. Khasidashvili, *Expression reduction systems*, Proceedings of I.Vekua Institute of Applied Mathematics (Tbisili), vol. 36, 1990, pp. 200–220.
18. J.-W. Klop, V. van Oostrom, and F. van Raamsdonk, Combinatory reduction systems, introduction and survey, *Theoretical Computer Science* **121** (1993), 279–308.
19. R. Mayr and T. Nipkow, Higher-order rewrite systems and their confluence, *Theoretical Computer Science* **192** (1998), 3–29.
20. M. R. Shinwell, A. M. Pitts, and Murdoch Gabbay, *FreshML: Programming with binders made simple*, ICFP 2003, pp. 263–274.
21. C. Urban, A. M. Pitts, and M. J. Gabbay, Nominal unification, *Theoretical Computer Science*, 323, pp. 473–497, 2004.