

Nominal Techniques for Reasoning about Formal Languages

Christian Urban
University of Cambridge

30 June 2004

Abstract

When reasoning about formal languages, dealing with binding constructs is often delicate and error-prone, in particular when proving properties by structural induction. This course describes a general framework for representing formal languages involving binders and reasoning modulo renaming of bound names. The framework is based on the operation of swapping of names and on the notion of equivariant predicates (that is predicates that are invariant under the swapping operation); the framework provides rigorous proving techniques that are pleasantly close to informal practice, in contrast with more sophisticated techniques such as de-Brujin indices and higher-order abstract syntax. Examples explaining the techniques will be taken from the lambda-calculus.

Acknowledgements: These overview notes draw heavily upon the papers [2, 3, 5, 6, 7, 8] and several other sources from the *FreshML*-project. I am most grateful to Andrew Pitts for all his help with preparing these notes. The notes also benefited much from the conversations with Christine Tasson. Any errors are my own work.

1 Introduction

Bound variables are often a troublesome aspect of reasoning about formal languages. The reason is that in the presence of binders the fundamental operation of substitution of terms for variables can, if defined incorrectly, easily lead to faulty results. For example (taken from [1]), the following equation is correct

$$I(y) : \int_0^1 x^2 + y \, dx = y + \frac{1}{3} \quad (1)$$

whereas the (naïve) substitution $I(y)[y := x]$ yields the incorrect equation

$$I(y)[y := x] : \int_0^1 x^2 + x \, dx = x + \frac{1}{3} . \quad (2)$$

Similarly, if we naïvely substitute a for b in the lambda-term $\lambda a.(b \, a)$, we receive the incorrect result $\lambda a.(a \, a)$. In the lambda-calculus such problems are usually avoided by identifying lambda-terms that differ only in the names of bound variables. This allows us to choose a fresh name so that binders do not clash with free variables. The convention is then imposed requiring that in any “context” bound and free variables are distinct. This means when we form the substitution $(\lambda a.b \, a)[b := a]$ we have bound and free occurrences of a in one “context”, which is precluded by the convention. When reasoning the problem with this convention is that the “context” might change and what needs to be renamed is determined by subtle side-conditions. This problem worsens if we attempt to formally reason about binders. If we choose to represent the two lambda-terms $\lambda a.(b \, a)$ and $\lambda a.(a \, a)$ using first-order abstract syntax

$$\begin{aligned} & \text{Abs}(a, \text{App}(\text{Var}(b), \text{Var}(a))) \\ & \text{Abs}(a, \text{App}(\text{Var}(a), \text{Var}(a))) \end{aligned} \quad (3)$$

then we need to do all such renamings by “hand”. This results in either proofs that contain a certain amount of informal reasoning or in rather messy formal proofs. Furthermore such renamings can also get in the way when establishing some properties by induction (we shall give an example later).

Another problem, which seems generally to be under-appreciated, is that by just equating lambda-terms that differ in the names of bound variables we receive a *non*-inductive set. Suppose we define the set Λ of lambda-terms using the constructors Abs , App and Var , then we obtain an inductive set. But if we then quotient Λ by α -equivalence, the set $\Lambda_{/\equiv_\alpha}$ we obtain is *not* inductive and consequently useful reasoning principles such as induction on the structure of lambda-terms are *not* applicable. They need to be replaced by, for example, inductions on the height of α -equated lambda-terms, which entails proof obligations to show that the height is preserved by the quotient-construction—which seems overly complicated and unnatural.

Of course all these problems can be avoided if we use a representation of binders based on de-Brujin indices. However, this is not an attractive solution, because de-Brujin indices are hard to read as the following two examples illustrate.

Substitution with de-Brujin

```

type trm =
  Var of int
  | Lam of trm
  | App of trm * trm
;;

let rec shift_by_one trm k =
  match trm with
  | Var(i) -> if i < k then Var(i) else Var(i+1)
  | Lam(t) -> Lam(shift_by_one t (k+1))
  | App(t1,t2) -> App(shift_by_one t1 k, shift_by_one t2 k)
;;

let rec subst trm k trm' =
  match trm with
  | Var(i) -> if i = k then trm' else Var(i)
  | Lam(t) -> Lam(subst t (k+1) (shift_by_one trm' 0))
  | App(t1,t2) -> App(subst t1 k trm', subst t2 k trm')
;;

```

Slide 1: Capture-avoiding substitution with de-Brujin indices in Ocaml.

with names		with de-Brujin indices
$\lambda a. \lambda b. (a b c)$	$\mapsto$	$\lambda \lambda (1\ 0\ 2)$
$\lambda a. \lambda b. (a (\lambda c. c a) b)$	$\mapsto$	$\lambda \lambda (1 (\lambda (0\ 2)) 0)$

Moreover definitions using de-Brujin indices are often too far “away” from informal practise (see Slide 1 for a definition of capture-avoiding substitution based on de-Brujin indices in Ocaml). So there is a point in attempting to do better than de-Brujin indices.

There are numerous techniques for reasoning about binders, but none of them has reached the status of universal acceptance. Here we shall attempt to remain close to informal practise as much as possible, but will be formal. The approach taken here differs in two main points from standard presentations about binders. First, there is a separate syntactic category for bindable entities, which we shall call *atoms*. For example x and y in the integral shown in (1) are atoms— x because it is bound by dx and y because potentially it can be bound; whereas the constants $0..3$ are *not* atoms (writing for example $d1$ does not make sense).

In case of the lambda-calculus, terms are build-up using atoms and two term-constructors for forming applications and abstractions. Hence the set Λ of lambda-terms is given by the grammar:

$t ::=$	a	atoms
	$ \quad tt$	applications
	$ \quad \lambda a. t$	abstractions

Note that the clause for abstractions involves a concrete atom (we are not “anonymising” the names of binders). The reason for having a separate syntactic category for

atoms is that an operation which we shall introduce next operates on the atoms of a given lambda-term—so it will also act on a in $\lambda a.t$. In what follows, we shall still refer to $\lambda a.$ as a binder, even if we treat them rather differently than what the usual meaning of binders would suggest.

The second difference with standard presentations of binders is that for renaming atoms we do *not* use the operation of substitution, but the operation of *swapping* atoms. As we have seen above, the problem with renaming substitutions is that they do not preserve α -equivalence (see also Slide 2). For example, if we start with the two α -equivalent lambda-terms

$$\lambda a.b \quad \text{and} \quad \lambda c.b$$

and apply the renaming substitution $[b := a]$, we get

$$\lambda a.b [b := a] = \lambda a.a \quad \lambda c.b [b := a] = \lambda c.a$$

which are no longer α -equivalent. In contrast, the simple operation of swapping atoms, written as $(ba) \bullet (-)$, preserves α -equivalence. Suppose we apply the swapping (ba) to the two lambda-terms above, we get

$$(ba) \bullet \lambda a.b = \lambda b.a \quad (ba) \bullet \lambda c.b = \lambda c.a$$

which are still α -equivalent (see Slide 3). It is important to notice that the swapping operation acts upon *all* atoms, be they are free, bound or binding. Also notice that swapping is symmetric, that is $(ba) \bullet (-)$ and $(ab) \bullet (-)$ produce the same results, unlike the renaming substitution $[b := a]$, which is clearly different from $[a := b]$.

In the following we shall generalise the notion of swapping of two atoms to permutations of atoms. We shall, however, still represent such permutations as lists of swappings. For example the atom-permutation:

$$\pi = \begin{pmatrix} a \mapsto b \\ b \mapsto a \\ c \mapsto c \end{pmatrix} \tag{4}$$

will be represented as the list of swappings $(cb)(ab)(ac)$, which is read from right to left. In general we have permutation of the form $(a_1 b_1) \cdots (a_n b_n)$. Note that our representation for permutations is *not* unique, but for our purposes that is not a problem. More important is that we can quickly calculate how a permutation π acts upon an atom a , written as $\pi \bullet a$. This operation is inductively defined on Slide 4 together with two auxiliary notions (the composition of two permutations and the inverse of a permutation) on Slide 5. So the example permutation $(cb)(ab)(ac)$ behaves just as the permutation given in (4):

$$\begin{aligned} (cb)(ab)(ac) \bullet a &= b \\ (cb)(ab)(ac) \bullet b &= a \\ (cb)(ab)(ac) \bullet c &= c \end{aligned}$$

Renaming Substitutions

Problem: substitution does not respect α -equivalence, e.g.

$$\begin{array}{ll} \lambda a. b [b := a] & \lambda c. b [b := a] \\ = \lambda a. a & = \lambda c. a \end{array}$$

Traditional Solution: replace $t [b := a]$ by a more complicated, 'capture-avoiding' form of substitution.

Slide 2: The usual operation of renaming is problematic: if defined naïvely, it does not respect α -equivalence. For α -equivalence to be respected we need to use capture-avoiding substitutions.

Swappings

Nice Alternative: use a less complicated operation for renaming

$$(b \ a) \bullet t \stackrel{\text{def}}{=} \text{swap all occurrences of } b \text{ and } a \text{ in } t$$

$$\begin{array}{ll} (b \ a) \bullet \lambda a. b & (b \ a) \bullet \lambda c. b \\ = \lambda b. a & = \lambda c. a \end{array}$$

Unlike for $(-)[b := a]$, for $(b \ a) \bullet (-)$ we do have if $t =_{\alpha} t'$ then $(b \ a) \bullet t =_{\alpha} (b \ a) \bullet t'$.

Slide 3: The operation of swapping atoms can be defined relatively easily (we do not need to know which atoms occur “free” or “bound”) and it preserves α -equivalence.

Permutations on Atoms

$$\begin{aligned} [] \cdot a &\stackrel{\text{def}}{=} a \\ ((a_1 \ a_2) :: \pi) \cdot a &\stackrel{\text{def}}{=} \begin{cases} a_1 & \text{if } \pi \cdot a = a_2 \\ a_2 & \text{if } \pi \cdot a = a_1 \\ \pi \cdot a & \text{otherwise} \end{cases} \end{aligned}$$

- $[]$ stands for the empty list (the identity permutation), and
- $(a_1 \ a_2) :: \pi$ stands for the permutation π followed by the swapping $(a_1 \ a_2)$

Slide 4: Inductive definition of a permutation (a list of swappings) acting on an atom.

Permutations on Atoms (ct.)

- the **composition** of two permutations is given by list-concatenation, written as $\pi' @ \pi$,
- the **inverse** of a permutation is given by list reversal, written as π^{-1} , and
- the **disagreement set** of two permutations π and π' is the set of atoms

$$ds(\pi, \pi') \stackrel{\text{def}}{=} \{a \mid \pi \cdot a \neq \pi' \cdot a\}$$

Slide 5: The composition of two permutations and the inverse of a permutation is given by the corresponding list-operations. The disagreement set is the (finite) set of atoms on which the permutations π and π' differ.

The permutation action has excellent logical properties stemming from the fact that permutations are bijections on atoms. We have the following facts:

Lemma 1.1.

- (i) $\pi^{-1} \cdot (\pi \cdot a) = a$
- (ii) $\pi \cdot a = b$ if and only if $a = \pi^{-1} \cdot b$
- (iii) $\pi_1 \cdot (\pi_2 \cdot a) = \pi_1 @ \pi_2 \cdot a$
- (iv) $\pi \cdot ((a b) \cdot c) = (\pi \cdot a \ \pi \cdot b) \cdot (\pi \cdot c)$

Proof. (i) by induction on π ; (ii) by simple calculation using (i); (iii) by induction on π_1 and (iv) by a case analysis and some calculations. $\square$

One last concept we need about permutations is that of a *disagreement set* between two permutations. It is defined on Slide 5. Note that every disagreement set $ds(\pi, \pi')$ is a subset of the *finite* set of atoms occurring in either of the lists π and π' , because if a does not occur in those lists, then from the definition we get $\pi \cdot a = a = \pi' \cdot a$. For example, the disagreement set of the permutations $(a c)(a b)$ and $(b c)$ is $\{a, c\}$ as the following calculation shows:

$$\begin{array}{ll} (a c)(a b) \cdot a = b & (b c) \cdot a = a \\ (a c)(a b) \cdot b = c & (b c) \cdot b = c \\ (a c)(a b) \cdot c = a & (b c) \cdot c = b \end{array}$$

Next we define how permutations act on lambda-terms. Recall that lambda-terms are defined to be atoms, applications and abstractions (the latter consisting of an atom and a term). A permutation action on a lambda-term is then defined as applying the permutation to *all* atoms that it meets on the way through the structure of a lambda-term. The inductive definition is shown on Slide 6 together with two facts asserting that permutations are bijections on lambda-terms (both can be easily proved by induction on t using the facts established in Lemma 1.1).

Using the permutation operation, we can give a very simple inductive definition for the α -equivalence relation on lambda-terms (see Slide 7). The interesting rules are the ones involving lambda-abstractions. If we have two lambda-terms with the same binder, say $\lambda a.t$ and $\lambda a.s$, then clearly they are α -equivalent provided t and s are α -equivalent (Rule $\approx$ -lam₁). In case we have two lambda-terms with different binders, say $\lambda a.t$ and $\lambda b.s$, then they are α -equivalent if t is α -equivalent with s in which every b has been renamed to a . For the renaming we use the permutation operation defined earlier. So for $\lambda a.t$ and $\lambda b.s$ being α -equivalent, we require that t and $(a b) \cdot s$ are α -equivalent. However, this rule is only sound if s does not contain any “free” occurrences of a . If there are any such “free” occurrences, then clearly $\lambda a.t$ and $\lambda b.s$ cannot be α -equivalent (consider for example the lambda-terms $\lambda a.a$ and $\lambda b.a$). To define the concept of an atom being not free in a lambda-term, we introduce a *freshness*

Permutations on λ -Terms

$$\begin{aligned} \pi \bullet (a) & \text{ given by the action on atoms} \\ \pi \bullet (t_1 t_2) & \stackrel{\text{def}}{=} (\pi \bullet t_1)(\pi \bullet t_2) \\ \pi \bullet (\lambda a. t) & \stackrel{\text{def}}{=} \lambda(\pi \bullet a)(\pi \bullet t) \end{aligned}$$

We have:

$$\begin{aligned} \blacksquare \pi^{-1} \bullet (\pi \bullet t) &= t \\ \blacksquare t_1 = t_2 & \text{ if and only if } \pi \bullet t_1 = \pi \bullet t_2 \end{aligned}$$

Slide 6: The permutation action on lambda-terms acts on *all* atoms occurring in a term; this operation is involutive and preserves (syntactic) equality.

relation, written as $a \# t$ and read as ‘ a is fresh for t ’. The inference rules defining this relation are also shown on Slide 7. (We leave it to the attentive reader to show that $\approx$ coincides with a standard definition of α -equivalence—depending from which definition you start, this can be rather hairy.)

Given that Rule $\approx\text{-lam}_2$ is defined rather “unsymmetrically”, it might be surprising that $\approx$ is symmetric. We can indeed show that $\approx$ is an equivalence relation and that freshness is preserved by $\approx$.

Lemma 1.2 ($\approx$ -Equivalence).

- (i) $t \approx t$
- (ii) if $t_1 \approx t_2$ and $t_2 \approx t_3$ then $t_1 \approx t_3$
- (iii) if $t_1 \approx t_2$ then $t_2 \approx t_1$
- (iv) if $a \# t_1$ and $t_1 \approx t_2$ then $a \# t_2$

Proof. The details of this proof are rather involved (mainly because of the number of cases we need to distinguish) and we therefore only give some vague hints: (i) is by a simple induction on t ; (ii) by an induction on the height of t_1 (the height is preserved by $\approx$); (iii) by an induction on the definition of $\approx$ using (ii); and (iv) by an induction on the definition of $\approx$. In this proof we make heavy use of the two facts given on Slide 6 and the auxiliary fact that

$$\text{if for all } a \in ds(\pi_1, \pi_2) \text{ we have } a \# t \text{ then } \pi_1 \bullet t \approx \pi_2 \bullet t$$

α-Equivalence & Freshness

$$\begin{array}{c}
 \frac{}{a \approx a} \approx\text{-var} \qquad \frac{t_1 \approx s_1 \quad t_2 \approx s_2}{t_1 t_2 \approx s_1 s_2} \approx\text{-app} \\
 \frac{t \approx s}{\lambda a.t \approx \lambda a.s} \approx\text{-lam}_1 \qquad \frac{t \approx (a b) \bullet s \quad a \# s}{\lambda a.t \approx \lambda b.s} \approx\text{-lam}_2 \\
 \hline
 \frac{}{a \# b} \#\text{-var} \qquad \frac{a \# t_1 \quad a \# t_2}{a \# t_1 t_2} \#\text{-app} \\
 \frac{}{a \# \lambda a.t} \#\text{-lam}_1 \qquad \frac{a \# t}{a \# \lambda b.t} \#\text{-lam}_2 \\
 \text{assuming } a \neq b
 \end{array}$$

Slide 7: An inductive definition of α -equivalence based on permutations and freshness. Notice that binders $\lambda a.$ and $\lambda b.$ have concrete atoms; if we were using the usual method of anonymising binders, then clearly it would make no sense to distinguish Rules $\approx\text{-lam}_1$ and $\approx\text{-lam}_2$.

which we can establish by an induction on t . □

Despite the proofs for showing the above facts being a bit tricky, the relations $\approx$ and $\#$ have their advantages. For example, compare them with the standard definition of α -equivalence usually stated as follows:

least congruence identifying $\lambda a.t$ with $\lambda b.t[a := b]$ if b does not occur at all in t and if $[a := b]$ is the naïve (possibly-capturing) substitution.

One feature of $\approx$ and $\#$ stands out: when deciding whether two lambda-terms are α -equivalent, there is never a need to choose an atom that does not occur in a term, as in the definition above (this feature is not just handy for implementations, but also crucial in the unification algorithm we shall present in Section 3).

Another feature of $\approx$ and $\#$ is noteworthy: both relations are respected by the permutation operation. We have:

Lemma 1.3. For all permutations π

- (i) $t_1 \approx t_2$ if and only if $\pi \bullet t_1 \approx \pi \bullet t_2$
- (ii) $a \# t$ if and only if $\pi \bullet a \# \pi \bullet t$

Proof. The former fact can be proved by induction on the height of t_1 and the latter by an induction on the structure of t . □

Relations (or predicates) that are preserved under permutations are called *equivariant* (see Slide 8). Whenever we define a relation involving terms with binders, equivariance seems to play an important rôle. This point is illustrated by the following example taken from the *simply-typed* lambda-calculus.

One of the most basic properties of the *simply-typed* lambda-calculus is the weakening property. This property asserts that for every typed lambda-term (typed with respect to a typing context Γ), we can extend Γ by a variable of arbitrary type provided this variable is not present in Γ . Let us make this more formal: The typing relation for lambda-terms takes the form $\Gamma \vdash t : \tau$ where types τ are given by the grammar $\tau ::= X \mid \tau \rightarrow \tau$ (with X ranging over an infinite set of type variables) and the typing context Γ is a finite partial function from variables to types. The typing relation is inductively generated by the an axiom and two rules following the structure of the terms t .

$$\frac{a : \tau \in \Gamma}{\Gamma \vdash a : \tau} \quad \frac{\Gamma \vdash t_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1 t_2 : \tau_2} \quad \frac{\Gamma, a : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda a. t : \tau_1 \rightarrow \tau_2} \quad a \notin \text{dom}(\Gamma) \quad (5)$$

The weakening property is then formally stated as the implication:

$$(\forall \Gamma)(\forall t)(\forall \tau) \quad \Gamma \vdash t : \tau \quad \Rightarrow \quad (\forall \tau')(\forall a \notin \text{dom}(\Gamma)) \quad \Gamma, a : \tau' \vdash t : \tau. \quad (6)$$

Clearly, the weakening property does *not* hold for “raw” lambda-terms, that is lambda-terms not identified modulo $\approx$ (try to weaken the context by a for $\lambda a. t$). It *does* hold for α -equated lambda-terms. Even if one usually does not bother to make a notational distinction between “raw” lambda-terms and α -equated lambda-terms, for reasons that will emerge shortly, we make the distinction explicit (see Slide 9) by using the notation $[t]_\alpha$ standing for α -equivalence class of the lambda-term t given by the set

$$[t]_\alpha \stackrel{\text{def}}{=} \{t' \mid t \approx t'\}. \quad (7)$$

Now we can start to prove the property shown in Slide 9: it is natural to try to proceed by rule induction and show that the predicate $\varphi(\Gamma, [t]_\alpha, \tau)$ given by

$$(\forall \tau')(\forall a' \notin \text{dom}(\Gamma)) \quad \Gamma, a' : \tau' \vdash [t]_\alpha : \tau$$

defines a relation that is closed under the axioms and rules. McKinna, Pollack and Pitts [4, 6] note that the induction step for the rule for abstractions is problematic: we have to prove that

$$\varphi(\Gamma, a : \tau_1, [t]_\alpha, \tau_2) \wedge a \notin \text{dom}(\Gamma) \quad \Rightarrow \quad \varphi(\Gamma, [\lambda a. t]_\alpha, \tau_1 \rightarrow \tau_2) \quad (8)$$

i.e. given

$$\varphi(\Gamma, a : \tau_1, [t]_\alpha, \tau_2) \quad \text{and} \quad a \notin \text{dom}(\Gamma) \quad (9)$$

Equivariance of $\approx$ and $\#$

A relation (or predicate) is called **equivariant** provided it is preserved under permutations, that is its validity is invariant under permutations. For example:

$$t_1 \approx t_2 \text{ if and only if } \pi \bullet t_1 \approx \pi \bullet t_2$$

$$a \# t \text{ if and only if } \pi \bullet a \# \pi \bullet t$$

It seems equivariance is an important concept when reasoning about properties involving syntax with binders.

Slide 8: Definition of relations being equivariant.

Typing Relation for $\Lambda_{/\approx}$

$$\frac{a : \tau \in \Gamma}{\Gamma \vdash [a]_\alpha : \tau}$$

$$\frac{\Gamma \vdash [t_1]_\alpha : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash [t_2]_\alpha : \tau_1}{\Gamma \vdash [t_1 t_2]_\alpha : \tau_2}$$

$$\frac{\Gamma, a : \tau_1 \vdash [t]_\alpha : \tau_2}{\Gamma \vdash [\lambda a. t]_\alpha : \tau_1 \rightarrow \tau_2} a \notin \text{dom}(\Gamma)$$

Weakening Property:

$$(\forall \Gamma)(\forall t)(\forall \tau) \quad \Gamma \vdash [t]_\alpha : \tau \Rightarrow (\forall \tau')(\forall a \notin \text{dom}(\Gamma)) \quad \Gamma, a : \tau' \vdash [t]_\alpha : \tau$$

Slide 9: Inference rules of the typing relation for simply-typed lambda-terms. Note that usually one does not bother to make the notational distinction between lambda-terms $t \in \Lambda$ and α -equated lambda-terms $[t]_\alpha \in \Lambda_{/\approx}$.

we have to prove that

$$\Gamma, a' : \tau' \vdash [\lambda a.t]_\alpha : \tau_1 \rightarrow \tau_2$$

holds for **all** $a' \notin \text{dom}(\Gamma)$ —and there is a problem with doing this for the case $a' = a$ because of the side-condition in the typing rule for lambda-abstractions. Usual practice is to ignore this problem by introducing a certain degree of informality into proofs, which of course cannot be done in formal proofs.

An elegant solution for this problem is if we take equivariance into account. We can check that each inference rule defining φ is preserved under permutations—that is, if we have an instance of one rule and apply a permutation to the conclusion and every premise, then we obtain again a valid instance of this rule (this also applies if we permute the types τ_1 and τ_2 , but this does not concern us here). Moreover, we can easily verify that the formula $a \notin \text{dom}(\Gamma)$ is equivariant (as well as formulae build by using $\wedge$ and $\Rightarrow$). Consider for example the typing rule for abstractions: given

$$\frac{\Gamma \vdash [t_1]_\alpha : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash [t_2]_\alpha : \tau_1}{\Gamma \vdash [t_1 t_2]_\alpha : \tau_2}$$

being a valid instance, we have that

$$\frac{\pi \bullet \Gamma \vdash \pi \bullet [t_1]_\alpha : \tau_1 \rightarrow \tau_2 \quad \pi \bullet \Gamma \vdash \pi \bullet [t_2]_\alpha : \tau_1}{\pi \bullet \Gamma \vdash \pi \bullet [t_1 t_2]_\alpha : \tau_2}$$

is a valid instance, where the permutation operation has been appropriately extended to sets of lambda-terms and to typing contexts. To see that this is a valid instance we need to use the fact that $\pi \bullet [t]_\alpha = [\pi \bullet t]_\alpha$ which holds by Lemma 1.3(i) and (7).

If we have satisfied ourself, that the inference rules are preserved under permutations then clearly also the predicate φ is preserved under permutations. We can use this fact to get the induction through. If we know that (9) holds, then also $\varphi(\Gamma, c : \tau_1, [(a c) \bullet t]_\alpha, \tau_2)$ for any *fresh* atom c . So by definition of φ , since $a' \notin \text{dom}(\Gamma, c : \tau_1)$, we have $(\Gamma, c : \tau_1), a' : \tau' \vdash [(a c) \bullet t]_\alpha : \tau_2$. After some calculations, we can apply the typing-rule for abstractions to conclude that $\Gamma, a' : \tau' \vdash [\lambda c.(a c) \bullet t]_\alpha : \tau_2$. But because c is fresh, the lambda-terms $\lambda c.(a c) \bullet t$ and $\lambda a.t$ are α -equivalent, and thus we can conclude that $\varphi(\Gamma, [\lambda a.t]_\alpha, \tau_1 \rightarrow \tau_2)$ holds. This completes the induction step.

2 Nominal Logic

From the considerations in the previous section, we can extract three crucial ideas for reasoning about binders: first permutations behave much better than renaming substitutions, second the notion of freshness should be made explicit and third all relations we define over syntax involving binders need to be equivariant. These ideas are codified in Nominal Logic introduced by Pitts [5].

Nominal logic is a many-sorted first-order logic based on the notions of atoms, swapping and equivariant predicates. Unlike in the previous section where we had to prove that $\approx, \#$ and φ are equivariant, in nominal logic equivariance is part of the mathematical infrastructure, that is there is one axiom asserting that *every* predicate is equivariant—no need to prove it case-by-case.

A nominal logic signature is given by some function and relation symbols respecting some sorting constraints. The sorts are defined on Slide 10. We shall use the letter S to range over arbitrary sorts, i.e. atom sorts, data sorts and atom-abstractions sorts. Since a language might contain several kinds of binders, we distinguishing several sorts of atoms, each representing countable finite number of atoms. The atom-abstraction sort is used to generically represent binders. For example, given an atom sort atm and a data sort trm , the lambda-term $\lambda a.t$ will be represented using a function symbol, say abs , with the sort annotation $[atm]trm \rightarrow trm$ as $abs(a.t)$. The terms and formulae of nominal logic are defined on Slide 11. The meaning of the new quantifier, written $\mathbb{N}$, will emerge shortly.

Nominal logic contains the usual axioms (for example excluded middle) and inference rules (for example modus ponens) of first-order logic with equality. In addition, nominal logic contains a number of novel axioms, which we review next. Slide 12 shows the axioms concerning the properties of swappings (from these axioms we can derive the properties of permutations established earlier in Section 1). Slide 13 gives the axioms concerning equivariance. Axiom E3 states that a function is equivariant provided a swapping on the “outside” can be “pushed” to the arguments. Axiom E4 states equivariance for predicates—in effect *all* predicates are equivariant in nominal logic. Slide 14 gives the axioms concerning freshness. Axiom F3 states that an atom of sort A is always fresh for an atom of sort A' (assuming A and A' are distinct atom sorts). Axiom F4 states that for every term in the language there exists an atom that is fresh with respect to this term. Slide 15 shows the axioms concerning atom-abstractions. Axiom A1 asserts the properties when two atom-abstractions are equal (they are defined similarly like Rules $\text{lam}_1 \approx$ and $\text{lam}_2 \approx$ given in Slide 7). Axiom A2 asserts that every term with an atom-abstraction sort *is* an atom-abstraction. The last axiom we need is for the $\mathbb{N}$ -quantifier:

$$(\mathbb{N}a : A)\varphi(a, \vec{x}) \Leftrightarrow (\exists a : A)a \# \vec{x} \wedge \varphi(a, \vec{x})$$

where the $\vec{x}$ stands for all free variables of the formula φ except a .

The main point of nominal logic is that we can now state very simple induction principles for theories about binders. Consider, for example, the following theory formalising the lambda-calculus. This theory consists of a sort atm for atoms and trm for lambda-terms; furthermore it has the three function symbols

$$\begin{aligned} var &: atm \rightarrow trm \\ app &: trm \times trm \rightarrow trm \\ abs &: [atm]trm \rightarrow trm \end{aligned}$$

Nominal Logic: Sorts

- A collection of ground sorts partitioned into two kinds: sorts of **atoms** A and sorts of **data** D . Sorts are then given by:

$$\begin{array}{ll}
 S ::= & A \quad \text{sorts of atoms} \\
 & | \quad D \quad \text{sorts of data} \\
 & | \quad [A]S \quad \text{sorts of atom-abstractions}
 \end{array}$$

- For function symbols f , we write $f : S^n \rightarrow S$ to indicate that f has arguments sorts S^n and result sort S .
- Similarly, for relation symbols R ; we write $R : S^n$ for their argument sorts.

Slide 10: This slides provides the grammar for how sorts are build up and how the sorts for function and relation symbols are determined.

Terms and Formulae

- $x : S$, if x is a variable and S is a sort
- $f(t_1, \dots, t_n)$, if $f : S^n \rightarrow S$ and $t_i : S_i$
- $(a \ b) \bullet t$, if $a, b : A$ and $t : S$
- $a.t : [A]S$, if $a : A$ and $t : S$
- $R(t_1, \dots, t_n)$, if $R : S^n$ and $t_i : S_i$
- $t_1 = t_2, a \# t$ if $a : A$ and $t_1, t_2, t : S$
- $\neg \phi, \psi \wedge \phi, \psi \vee \phi, \psi \Rightarrow \phi, \psi \Leftrightarrow \phi$
- $(\forall x : S)\phi, (\exists x : S)\phi$, if x is of sort S
- $(\mathcal{U}x : A)\phi$, if x is of sort A

Slide 11: The terms of nominal logic are defined in the first four points; the remaining points define the formulae of nominal logic. Nonstandard are swappings (third point), abstractions (fourth point), freshness-formulae (sixth point) and the new-quantifier (last point).

Properties of Swapping

- **S1:**
 $(\forall a : A)(\forall x : S)(a \ a) \bullet x = x$
- **S2:**
 $(\forall a, b : A)(\forall x : S)(a \ b) \bullet (a \ b) \bullet x = x$
- **S3:**
 $(\forall a, b : A)(a \ b) \bullet a = b$

From those axioms we can get back the properties of permutations.

Slide 12: Axioms asserting the properties of swapping.

Properties of Equivariance

- **E1:**
 $(\forall a, a' : A)(\forall b, b' : A')(\forall x : S)$
 $(a \ a') \bullet (b \ b') x = ((a \ a') \bullet b \ (a \ a') \bullet b') \bullet (a \ a') \bullet x$
- **E2:**
 $(\forall a, a' : A)(\forall b : A')(\forall x : S)$
 $b \# x \Rightarrow (a \ a') \bullet b \# (a \ a') \bullet x$
- **E3:**
 $(\forall a, b : A)(\forall \vec{x} : \vec{S})(a \ b) \bullet f(\vec{x}) = f((a \ b) \bullet \vec{x})$
- **E4:**
 $(\forall a, b : A)(\forall \vec{x} : \vec{S}) R(\vec{x}) \Rightarrow R((a \ b) \bullet \vec{x})$
- **E5:**
 $(\forall b, b' : A)(\forall a : A')(\forall x : S)$
 $(b \ b') \bullet (a \ x) = ((b \ b') \bullet a) \bullet ((b \ b') \bullet x)$

Slide 13: Axioms asserting the properties of equivariance.

Properties of Freshness

- **F1:**
 $(\forall a, a' : A)(\forall x : S)$
 $a \# x \wedge a' \# x \Rightarrow (a \ a') \bullet x = x$
- **F2:**
 $(\forall a, a' : A) a \# a' \Leftrightarrow \neg a = a'$
- **F3:**
 $(\forall a : A)(\forall b : A') a \# b$ where $A \neq A'$
- **F4:**
 $(\forall x : S)(\exists a : A) a \# x$

Slide 14: Axioms asserting the properties of freshness.

Properties of Abstractions

- **A1:**
 $(\forall a, a' : A)(\forall x, x' : S)$
 $a.x = a'.x' \Leftrightarrow$
 $(a = a' \wedge x = x')$
 $\vee$
 $(a \neq a' \wedge x = (a \ a') \bullet x' \wedge a \# x')$
- **A2:**
 $(\forall y : [A]S)(\exists a : A)(\exists x : S)$
 $y = a.x$

Slide 15: Axioms asserting the properties of atom-abstractions.

for representing lambda-terms. With the next seven axioms we formalising the “meaning” of these function symbols.

$$\begin{aligned}
& (\forall a : atm)(\forall t, t' : trm) \neg var(a) = app(t, t') \\
& (\forall a, a' : atm)(\forall t : trm) \neg var(a) = abs(a'.t) \\
& (\forall a : atm)(\forall t, t', t'' : trm) \neg abs(a.t) = app(t', t'') \\
& (\forall t : trm) \quad (\exists a : atm) t = var(a) \\
& \quad \vee (\exists t', t'' : trm) t = app(t', t'') \\
& \quad \vee (\exists a : atm)(\exists t' : trm) t = abs(a.t') \\
& (\forall a, a' : atm) var(a) = var(a') \Rightarrow a = a' \\
& (\forall t, t', s, s' : trm) app(t, t') = app(s, s') \Rightarrow t = s \wedge t' = s' \\
& (\forall a, a' : atm)(\forall t, t' : trm) abs(a.t) = abs(a'.t') \Rightarrow a.t = a'.t'
\end{aligned}$$

The induction principle for lambda-terms is then

$$\begin{aligned}
(\forall \vec{x} : S) \quad & (\forall a : atm) \varphi(atm(a), \vec{x}) \\
& \wedge (\forall t, t' : trm) \varphi(app(t, t'), \vec{x}) \\
& \wedge (\forall a : atm)(\forall t : trm) \varphi(abs(a.t), \vec{x}) \\
& \Rightarrow (\forall t : trm) \varphi(t, \vec{x})
\end{aligned}$$

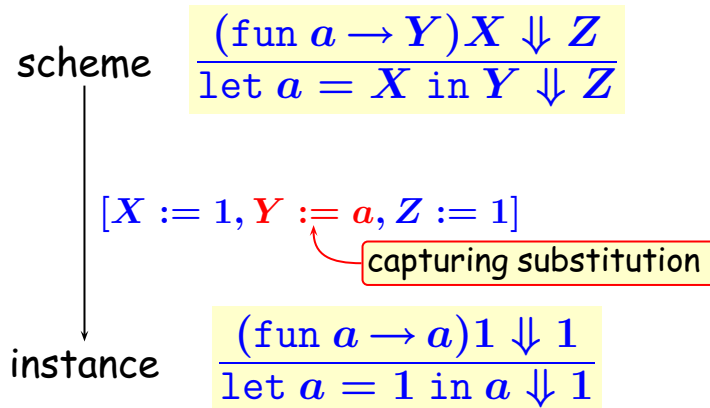
which asserts that a property holds for all lambda-terms provided we can prove this properties for variables, applications and for abstractions—the crux however is that we only need to prove the abstraction case for a “new” atom a . In the lectures we will spend a considerable amount of time justifying this induction principle.

3 Unification of Terms with Binders

It is commonplace when reasoning with formal languages to define schematic inference rules and then to match terms against these rules in order to obtain specific instances of the rules. While this is trivial with inference rules that have no binders (see Slide 7 for an example), there are some subtle issues when binders are involved as can be seen in Slide 16 (see also Slide 17). Using the concepts of permutations and freshness, we shall present in this section an algorithm that solves unification (and matching) problems involving binders.

To do unification we shall extend our term language to include variables which can be substituted by terms. In order to distinguish them from atoms, we shall use capital letters for variables (so far we have build terms using atoms which can be permuted, but not substituted). For reasons that will become clear shortly, we will use actually *suspensions*, that is variables together with permutations. The terms are given on Slide 18.

Schematic Inference Rules



Slide 16: Instantiation of a schematic inference rule requires capturing substitutions when terms with binders are involved. The example shows a typical inference rule for a big-step evaluation relation.

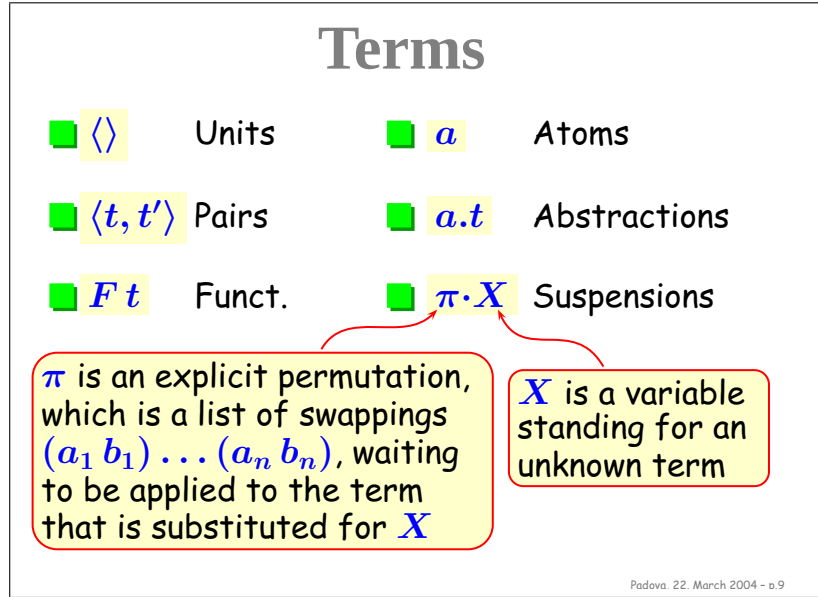
Quiz

Assuming that a and b are distinct variables, is it possible to find λ -terms M_1 to M_7 that make the following pairs α -equivalent?

- $\lambda a. \lambda b. (M_1 \ b)$ and $\lambda b. \lambda a. (a \ M_1)$
- $\lambda a. \lambda b. (M_2 \ b)$ and $\lambda b. \lambda a. (a \ M_3)$
- $\lambda a. \lambda b. (b \ M_4)$ and $\lambda b. \lambda a. (a \ M_5)$
- $\lambda a. \lambda b. (b \ M_6)$ and $\lambda a. \lambda a. (a \ M_7)$

If there is one solution for a pair, can you describe all its solutions?

Slide 17: [Try. The interesting question is, of course, the “describe all solutions” part.]



Slide 18: Terms.

We now need to extend the notion of α -equivalence to terms with variables. For this we shall equate the following two terms:

$$\lambda a.X \approx \lambda b.(a b) \bullet X$$

where $(a b) \bullet X$ stands for a swapping suspended in front of the variable X . This equation will remain valid under substitution as long as we do *not* substitute for X a term that contains free occurrences of b . Therefore our notion of α -equivalence will be here not just an equation between two terms (as on Slide 7), but an equation under some freshness constraints. The inference rule defining this generalised notion of α -equivalence and freshness are given in Figure 1.

The permutation action is defined similarly as given on Slide 6, that is permutation are applied to all atoms it meets on the way, except that permutations “piled up” in front of variables giving what we have called *suspensions*. In case the permutation π in a suspension is the empty list, we just write X for $\pi \bullet X$. The main reason for using suspensions in the syntax of terms is to enable a definition of *substitution of terms for variables* that allows capture of free atoms by atom-abstractions while still respecting α -equivalence. This substitution is defined as follows:

Definition 3.1 (Substitution). A substitution σ is a function from variables to terms with the property that $\sigma(X) = X$ for all but finitely many variables X . We shall write $\text{dom}(\sigma)$ for the finite set of variables X satisfying $\sigma(X) \neq X$. If $\text{dom}(\sigma)$ consists of distinct variables $X_1, \dots, X_n$ and $\sigma(X_i) = t_i$ for $i = 1..n$, we shall sometimes write σ as

$$\sigma = [X_1 := t_1, \dots, X_n := t_n]. \quad (10)$$

We apply substitutions to terms, written $\sigma(t)$, as follows:

$$\begin{array}{c}
\overline{\nabla \vdash \langle \rangle \approx \langle \rangle} \text{ } (\approx\text{-unit}) \quad \frac{\nabla \vdash t_1 \approx t'_1 \quad \nabla \vdash t_2 \approx t'_2}{\nabla \vdash \langle t_1, t_2 \rangle \approx \langle t'_1, t'_2 \rangle} (\approx\text{-pair}) \\
\\
\frac{\nabla \vdash t \approx t'}{\nabla \vdash a.t \approx a.t'} (\approx\text{-abstraction-1}) \quad \frac{\nabla \vdash t \approx t'}{\nabla \vdash f t \approx f t'} (\approx\text{-function symbol}) \\
\\
\frac{a \neq a' \quad \nabla \vdash t \approx (a a') \cdot t' \quad \nabla \vdash a \# t'}{\nabla \vdash a.t \approx a'.t'} (\approx\text{-abstraction-2}) \\
\\
\overline{\nabla \vdash a \approx a} (\approx\text{-atom}) \quad \frac{(a \# X) \in \nabla \text{ for all } a \in ds(\pi, \pi')}{\nabla \vdash \pi.X \approx \pi'.X} (\approx\text{-suspension}) \\
\\
\overline{\nabla \vdash a \# \langle \rangle} (\# \text{-unit}) \quad \frac{\nabla \vdash a \# t}{\nabla \vdash a \# f t} (\# \text{-function symbol}) \\
\\
\frac{\nabla \vdash a \# t_1 \quad \nabla \vdash a \# t_2}{\nabla \vdash a \# \langle t_1, t_2 \rangle} (\# \text{-pair}) \\
\\
\overline{\nabla \vdash a \# a.t} (\# \text{-abstraction-1}) \quad \frac{a \neq a' \quad \nabla \vdash a \# t}{\nabla \vdash a \# a'.t} (\# \text{-abstraction-2}) \\
\\
\frac{a \neq a'}{\nabla \vdash a \# a'} (\# \text{-atom}) \quad \frac{(\pi^{-1} \cdot a \# X) \in \nabla}{\nabla \vdash a \# \pi.X} (\# \text{-suspension})
\end{array}$$

Figure 1: Inductive definition of $\approx$ and $\#$.

$$\begin{array}{ll}
\sigma(\langle \rangle) & \stackrel{\text{def}}{=} \langle \rangle \\
\sigma(a) & \stackrel{\text{def}}{=} a \\
\sigma(\langle t_1, t_2 \rangle) & \stackrel{\text{def}}{=} \langle \sigma(t_1), \sigma(t_2) \rangle \\
\sigma(f t) & \stackrel{\text{def}}{=} f \sigma(t) \\
\sigma(a.t) & \stackrel{\text{def}}{=} a.\sigma(t) \\
\sigma(\pi.X) & \stackrel{\text{def}}{=} \pi \cdot \sigma(X)
\end{array}$$

where in the last clause a permutation action needs to be performed. Given substitutions σ and σ' , and freshness environments ∇ and ∇' , we write

$$(a) \quad \nabla' \vdash \sigma(\nabla) \quad \text{and} \quad (b) \quad \nabla \vdash \sigma \approx \sigma' \quad (11)$$

to mean that (for a) $\nabla' \vdash a \# \sigma(X)$ holds for each $(a \# X) \in \nabla$ and (for b) $\nabla \vdash \sigma(X) \approx \sigma'(X)$ holds for all $X \in dom(\sigma) \cup dom(\sigma')$.

Suppose we have the terms t and t' , can we decide whether or not there is a substitution of terms for the variables in t and t' that makes them equal in the sense of the relation $\approx$ introduced in Figure 1? Since instances of $\approx$ are established modulo freshness constraints, it makes more sense to ask whether or not there is both a substitution σ and a freshness environment ∇ for which $\nabla \vdash \sigma(t) \approx \sigma(t')$ holds. As for ordinary first-order unification, solving such an equational problem may throw up *several* equational subproblems; but an added complication here is that because of rule ($\approx$ -abstraction-2) in Figure 1, equational problems may generate *freshness* problems, i.e. ones involving the relation $\#$. We are thus led to the unification problems stated on Slide 19 and freshness problem stated on Slide 20.

A *solution* for a unification problem P consists of a pair (∇, σ) where ∇ is a freshness environment and σ is a substitution satisfying

- $\nabla \vdash a \# \sigma(t)$ for each $(a \#? t) \in P$ and
- $\nabla \vdash \sigma(t) \approx \sigma(t')$ for each $(t \approx? t') \in P$.

We write $\mathcal{U}(P)$ for the set of all solutions of a problem P . $(\nabla, \sigma) \in \mathcal{U}(P)$ is a *most general* solution for P if given any other solution $(\nabla', \sigma') \in \mathcal{U}(P)$, then there is a substitution σ'' satisfying $\nabla' \vdash \sigma''(\nabla)$ and $\nabla' \vdash \sigma'' \circ \sigma \approx \sigma'$ (see Slide 21)

We now describe an algorithm which, given any unification problem, decides whether or not it has a solution and if it does, returns a most general solution. The algorithm uses labelled transformations, directly generalising the presentation of first-order unification. We use two types of labelled transformation between unification problems, namely

$$P \xRightarrow{\sigma} P' \quad \text{and} \quad P \xRightarrow{\nabla} P'$$

where the substitution σ is either the identity ε , or a single replacement $[X := t]$; and where the freshness environment ∇ is either empty $\emptyset$, or a singleton $\{a \# X\}$. The legal transformations are given in Figure 2. This figure uses the notation $P \uplus P'$ to indicate the union of problems P and P' that are disjoint ($P \cap P' = \emptyset$); and the notation σP to indicate the problem resulting from applying the substitution σ to all the terms occurring in the problem P .

Given a unification problem P , the algorithm proceeds in two phases. In the first phase it applies as many $\xRightarrow{\sigma}$ transformations as possible (non-deterministically). If this results in a problem containing no equational subproblems, then it proceeds to the second phase; otherwise it halts signalling failure. In the second phase it applies as many $\xRightarrow{\nabla}$ transformations as possible (non-deterministically). If this does not result in the empty problem, then it halts signalling failure; otherwise overall it has constructed a transformation sequence of the form

$$P \xRightarrow{\sigma_1} \dots \xRightarrow{\sigma_n} P' \xRightarrow{\nabla_1} \dots \xRightarrow{\nabla_m} \emptyset \tag{12}$$

Equational Problems

An equational problem

$$t \approx? t'$$

is **solved** by

- a substitution σ (terms for variables)
- and a set of freshness assumptions ∇

so that $\nabla \vdash \sigma(t) \approx \sigma(t')$.

Munich, 20 January 2003 - p.15

Slide 19: Equational problems.

Freshness Problems

A freshness problem

$$a \#? t$$

is **solved** by

- a substitution σ
- and a set of freshness assumptions ∇

so that $\nabla \vdash a \# \sigma(t)$.

Munich, 20 January 2003 - p.17

Slide 20: Freshness problems.

Most General Unifiers

Definition: for a unification problem P , a solution (σ_1, ∇_1) is **more general** than another solution (σ_2, ∇_2) , iff. there exists a substitution σ with

$$\nabla_2 \vdash \sigma(\nabla_1)$$

$$\nabla_2 \vdash \sigma_2 \approx \sigma \circ \sigma_1$$

Munich, 20 January 2003 - s.21

Slide 21: Definition of most general unifiers.

(where P' does not contain any equational subproblems) and the algorithm returns the solution $(\nabla_1 \cup \dots \cup \nabla_m, \sigma_n \circ \dots \circ \sigma_1)$.

A machine-checked proof of the correctness of this unification algorithm using the theorem prover Isabelle can be found at

<http://www.cl.cam.ac.uk/users/cu200/Unification>.

This unification algorithm is used in the logic programming language called α Prolog in which one can implement the typing rule given on Slide 9 *correctly* as:

```
type Gamma (var X) A :- mem (pair X A) Gamma.
type Gamma (app M N) B :- type Gamma M (arrow A B),
                           type Gamma N A.
type Gamma (lam x.M) (arrow A B) / x#Gamma :-
                           type (pair x A)::Gamma M B.

mem A A::Tail.
mem A B::Tail :- mem A Tail.
```

I am currently working on a theorem prover based on this unification algorithm. The aim is to allow the user to conveniently, but formally, verify properties about syntax with binders, which is still fiendishly difficult with state-of-the-art theorem provers.

($\approx?$ -unit)	$\{\langle \rangle \approx? \langle \rangle\} \uplus P \xRightarrow{\varepsilon} P$
($\approx?$ -pair)	$\{\langle t_1, t_2 \rangle \approx? \langle t'_1, t'_2 \rangle\} \uplus P \xRightarrow{\varepsilon} \{t_1 \approx? t'_1, t_2 \approx? t'_2\} \cup P$
($\approx?$ -function symbol)	$\{f t \approx? f t'\} \uplus P \xRightarrow{\varepsilon} \{t \approx? t'\} \cup P$
($\approx?$ -abstraction-1)	$\{a.t \approx? a'.t'\} \uplus P \xRightarrow{\varepsilon} \{t \approx? t'\} \cup P$
($\approx?$ -abstraction-2)	$\{a.t \approx? a'.t'\} \uplus P \xRightarrow{\varepsilon} \{t \approx? (a a') \bullet t', a \#? t'\} \cup P$ provided $a \neq a'$
($\approx?$ -atom)	$\{a \approx? a\} \uplus P \xRightarrow{\varepsilon} P$
($\approx?$ -suspension)	$\{\pi \bullet X \approx? \pi' \bullet X\} \uplus P \xRightarrow{\varepsilon} \{a \#? X \mid a \in ds(\pi, \pi')\} \cup P$
($\approx?$ -variable)	$\left\{ \begin{array}{l} \{t \approx? \pi \bullet X\} \uplus P \\ \{\pi \bullet X \approx? t\} \uplus P \end{array} \right\} \xRightarrow{\sigma} \sigma P \quad \text{with } \sigma = [X := \pi^{-1} \bullet t]$ provided X does not occur in t
($\#?$ -unit)	$\{a \#? \langle \rangle\} \uplus P \xRightarrow{\emptyset} P$
($\#?$ -pair)	$\{a \#? \langle t_1, t_2 \rangle\} \uplus P \xRightarrow{\emptyset} \{a \#? t_1, a \#? t_2\} \cup P$
($\#?$ -function symbol)	$\{a \#? f t\} \uplus P \xRightarrow{\emptyset} \{a \#? t\} \cup P$
($\#?$ -abstraction-1)	$\{a \#? a.t\} \uplus P \xRightarrow{\emptyset} P$
($\#?$ -abstraction-2)	$\{a \#? a'.t\} \uplus P \xRightarrow{\emptyset} \{a \#? t\} \cup P$ provided $a \neq a'$
($\#?$ -atom)	$\{a \#? a'\} \uplus P \xRightarrow{\emptyset} P$ provided $a \neq a'$
($\#?$ -suspension)	$\{a \#? \pi \bullet X\} \uplus P \xRightarrow{\nabla} P \quad \text{with } \nabla = \{\pi^{-1} \bullet a \# X\}$

Figure 2: Labelled transformations.

References

- [1] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.
- [2] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. In *14th Symposium of LICS*, pages 214–224. IEEE Computer Society Press, 1999.
- [3] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002.
- [4] J. McKinna and R. Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3–4), Nov. 1999.
- [5] A. M. Pitts. Nominal logic: A first order theory of names and binding. In *Proc. of TACS 2001*, volume 2215 of *LNCS*, pages 219–242. Springer-Verlag, 2001.
- [6] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.
- [7] C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. to appear in *TCS*.

- [8] C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. In M. Baaz, editor, *Computer Science Logic and 8th Kurt Godel Colloquium (CSL'03 & KGC)*, volume 2803 of *LNCS*, pages 513–527, 2003.