Nominal Techniques in Isabelle/HOL*

Christian Urban[1] and Christine Tasson[2]

[1] Ludwig-Maximilians-University Munich (urban@mathematik.uni-muenchen.de)
[2] ENS Cachan Paris (tasson@dptmaths.ens-cachan.fr)

Abstract. In this paper we define an inductive set that is bijective with the α-equated lambda-terms. Unlike de-Bruijn indices, however, our inductive definition includes names and reasoning about this definition is very similar to informal reasoning on paper. For this we provide a structural induction principle that requires to prove the lambda-case for fresh binders only. The main technical novelty of this work is that it is compatible with the axiom-of-choice (unlike earlier nominal logic work by Pitts *et al*); thus we were able to implement all results in Isabelle/HOL and use them to formalise the standard proofs for Church-Rosser and strong-normalisation.

Keywords. Lambda-calculus, nominal logic, structural induction, theorem-assistants.

1 Introduction

Whenever one wants to formalise proofs about terms involving binders, one faces a problem: how to represent such terms? The "low-level" representations use concrete names for binders (that is they represent terms as abstract syntax trees) or use de-Bruijn indices. However, a brief look in the literature shows that both representations make formal proofs rather strenuous in places (typically lemmas about substitution) that are only loosely concerned with the proof at hand. Three examples from the literature: VanInwegen wrote [19, p. 115]:

> *"Proving theorems about substitutions (and related operations such as alpha-conversion) required far more time and HOL code than any other variety of theorem."*

in her PhD-thesis, which describes a formalisation of SML's subject reduction property based on a "concrete-name" representation for SML-terms. Altenkirch formalised in LEGO a strong normalisation proof for System-F (using a de-Bruijn representation) and concluded [1, p. 26]:

> *"When doing the formalization, I discovered that the core part of the proof... is fairly straightforward and only requires a good understanding of the paper version. However, in completing the proof I observed that in certain places I had to invest much more work than expected, e.g. proving lemmas about substitution and weakening."*

* Published in Proc. of the 20th Int. Conf. on Automated Deduction (CADE 2005). In volume 3632 of Lecture Notes in Computer Science. Springer-Verlag. Pages 38-53.

Hirschkoff made a similar comment in [10, p. 167] about a formalisation of the π-calculus:

> *"Technical work, however, still represents the biggest part of our implementation, mainly due to the managing of de Bruijn indexes...Of our 800 proved lemmas, about 600 are concerned with operators on free names."*

The main point of this paper is to give a representation for α-*equated* lambda-terms that is based on names, is inductive and comes with a structural induction principle where the lambda-case needs to be proved for only fresh binders. In practice this will mean that we come quite close to the informal reasoning using Barendregt's variable convention. Our work is based on the nominal logic work by Pitts *et al* [16, 6]. The main technical novelty is that our work by giving an explicit construction for α-equated lambda-terms is compatible with the axiom of choice. Thus we were able to implement all results in Isabelle/HOL and formalise the simple Church-Rosser proof of Tait and Martin-Löf described in [3], and the standard Tait-style strong normalisation proof for the simply-typed lambda-calculus given, for example, in [7, 17].

The paper is organised as follow: Sec. 2 reviews α-equivalence for lambda-terms. Sec. 3 gives a construction of an inductive set that is bijective with the α-equated lambda-terms and adapts some notions of the nominal logic work for this construction. An induction principle for this set is derived in Sec. 4. Examples of Isabelle/HOL formalisations are given in Sec. 5. Related work is mentioned in Sec. 6, and Sec. 7 concludes.

2 Preliminaries

In order to motivate a design choice later on, we begin with a review of α-equivalence cast in terms of the nominal logic work. The set of lambda-terms is inductively defined by the grammar:

$$\Lambda: \quad t ::= a \mid t\,t \mid \lambda a.t$$

where a is an atom drawn from a countable infinite set, which will in what follows be denoted by $\mathbb{A}$.

The notion of α-equivalence for Λ is often defined as the least congruence of the equation $\lambda a.t =_\alpha \lambda b.t[a := b]$ involving a renaming substitution and a side-condition, namely that b does not occur freely in t. In the nominal logic work, however, atoms are manipulated not by renaming substitutions, but by permutations—bijective mappings from atoms to atoms. While permutations have some technical advantages, for example they preserve α-equivalence which substitutions do not [18], their primary reason in the nominal logic work is that one can use them to define the notion of *support*. This notion generalises what is meant by the set of free atoms of an object, which is usually clear in case the object is an abstract syntax tree, but less so if the object is a function. The generalisation of "free atoms" to functions, however, will play a crucial rôle in our construction of the bijective set.

There are several ways for defining the operation of a permutation acting on a lambda-term. One way [18] that can be easily implemented in Isabelle/HOL is to represent permutations as finite lists whose elements are swappings (i.e., pairs of atoms). We write such permutation as $(a_1\,b_1)(a_2\,b_2)\cdots(a_n\,b_n)$; the empty list $[]$ stands for the identity permutation. The permutation action, written $\pi{\bullet}(-)$, can then be defined on lambda-terms as:

$$
\begin{array}{ll}
[]{\bullet}a \stackrel{\text{def}}{=} a & \\
(a_1\,a_2) :: \pi{\bullet}a \stackrel{\text{def}}{=} \begin{cases} a_2 & \text{if } \pi{\bullet}a = a_1 \\ a_1 & \text{if } \pi{\bullet}a = a_2 \\ \pi{\bullet}a & \text{otherwise} \end{cases} &
\begin{array}{r} \pi{\bullet}(t_1\ t_2) \stackrel{\text{def}}{=} (\pi{\bullet}t_1\ \pi{\bullet}t_2) \\ \pi{\bullet}(\lambda a.t) \stackrel{\text{def}}{=} \lambda(\pi{\bullet}a).(\pi{\bullet}t) \end{array}
\end{array}
\tag{1}
$$

where $(a\,b) :: \pi$ is the composition of a permutation followed by the swapping $(a\,b)$. The composition of π followed by another permutation π' is given by list-concatenation, written as $\pi'@\pi$, and the inverse of a permutation is given by list reversal, written as π^{-1}.

While the representation of permutations based on lists of swappings is convenient for definitions like permutation composition and the inverse of a permutation, this list-representation is not unique; for example the permutation $(a\,a)$ is "equal" to the identity permutation. Therefore some means to identify "equal" permutations is needed.

Definition 1 (Disagreement Set and Permutation Equality). *The* disagreement set *of two permutations, say π_1 and π_2, is the set of atoms on which the permutations disagree, that is* $\mathtt{ds}(\pi_1, \pi_2) \stackrel{def}{=} \{\, a \mid \pi_1{\bullet}a \neq \pi_2{\bullet}a \,\}$. *Two permutations are* equal*, written $\pi_1 \sim \pi_2$, provided* $\mathtt{ds}(\pi_1, \pi_2) = \varnothing$.

Using the permutation action on lambda-terms, α-equivalence for Λ can be defined in a syntax directed fashion using the relations $(-)\approx(-)$ and $(-) \notin \mathtt{fv}(-)$; see Fig. 1. Because of the "asymmetric" rule $\approx_{\lambda 2}$, it might be surprising, but:

Proposition 1. *$\approx$ is an equivalence relation.*

The proof of this proposition is omitted: it can be found in a more general setting in [18]. (We also omit a proof showing that $\approx$ and $=_\alpha$ coincide). In the following, $[t]_\alpha$ will stand for the α-equivalence class of the lambda-term t, that is $[t]_\alpha \stackrel{\text{def}}{=} \{\, t' \mid t' \approx t \,\}$, and $\Lambda_{/\approx}$ for the set Λ quotient by $\approx$.

3 The Bijective Set

In this section, we will define a set Φ; inside this set we will subsequently identify (inductively) a subset, called Λ_α, that is in bijection with $\Lambda_{/\approx}$. In order to obtain the bijection, Φ needs to be defined so that it contains elements corresponding, roughly speaking, to α-equated atoms, applications and lambda-abstractions—that is to $[a]_\alpha$, $[t_1 t_2]_\alpha$ and $[\lambda a.t]_\alpha$. Whereas this is straightforward for atoms and applications, the lambda-abstractions are non-trivial: for them we shall use some

$$\frac{}{a \approx a}\approx_{\text{var}} \quad \frac{t_1 \approx s_1 \quad t_2 \approx s_2}{t_1\, t_2 \approx s_1\, s_2}\approx_{\text{app}} \quad \frac{t \approx s}{\lambda a.t \approx \lambda a.s}\approx_{\lambda 1} \quad \frac{a \neq b \quad t \approx (a\, b)\bullet s \quad a \notin \mathtt{fv}(s)}{\lambda a.t \approx \lambda b.s}\approx_{\lambda 2}$$

$$\frac{a \neq b}{a \notin \mathtt{fv}(b)}\mathtt{fv}_{\text{var}} \quad \frac{a \notin \mathtt{fv}(t_1) \quad a \notin \mathtt{fv}(t_2)}{a \notin \mathtt{fv}(t_1\, t_2)}\mathtt{fv}_{\text{app}} \quad \frac{}{a \notin \mathtt{fv}(\lambda a.t)}\mathtt{fv}_{\lambda 1} \quad \frac{a \neq b \quad a \notin \mathtt{fv}(t)}{a \notin \mathtt{fv}(\lambda b.t)}\mathtt{fv}_{\lambda 2}$$

Fig. 1. Inductive definitions for $(-) \approx (-)$ and $(-) \notin \mathtt{fv}(-)$.

specific "partial" functions from $\mathbb{A}$ to Φ (by "partial" we mean functions that return "error" for undefined values[1]). Thus the set Φ is defined by the grammar

$$\Phi: \quad t ::= \mathtt{er} \mid \mathtt{am}(a) \mid \mathtt{pr}(t,t) \mid \mathtt{se}(\mathit{fn})$$

where $\mathtt{er}$ stands for "error", a for atoms and fn stands for functions from $\mathbb{A}$ to Φ.[2] This grammar corresponds to the inductive datatype that one might declare in Isabelle/HOL as:

```
datatype phi = er
             | am "atom"
             | pr "phi × phi"
             | se "atom ⇒ phi"
```

where it is presupposed that the type `atom` has been declared. The constructors `am`, `pr` and `se` will be used in Λ_α for representing α-equated atoms, applications and lambda-abstractions. Before the subset Λ_α can be carved out from Φ, however, some terminology from the nominal logic work needs to be adapted. For this we overload the notion of permutation action, that is $\pi\bullet(-)$, and define abstractly sets that come with a notion of permutation:

Definition 2 (PSets). *A set X equipped with a permutation action $\pi\bullet(-)$ is said to be a* pset*, if for all $x \in X$, the permutation action satisfies the following properties:*

(i) $[]\bullet x = x$
(ii) $\pi_1 @ \pi_2 \bullet x = \pi_1 \bullet(\pi_2 \bullet x)$
(iii) *if* $\pi_1 \sim \pi_2$ *then* $\pi_1 \bullet x = \pi_2 \bullet x$

The informal notation $x \in \mathit{pset}$ will be adopted whenever it needs to be indicated that x comes from a pset. The idea behind the permutation action, roughly speaking, is to permute all atoms in a given pset-element. For lists, tuples and sets the permutation action is therefore defined point-wise:

lists:	$\pi\bullet[] \stackrel{\text{def}}{=} []$	tuples:	$\pi\bullet(x_1,\ldots,x_n) \stackrel{\text{def}}{=} (\pi\bullet x_1,\ldots,\pi\bullet x_n)$
	$\pi\bullet(x :: t) \stackrel{\text{def}}{=} (\pi\bullet x) :: (\pi\bullet t)$	sets:	$\pi\bullet X \stackrel{\text{def}}{=} \{\pi\bullet x \mid x \in X\}$

[1] This is one way of dealing with partial functions in Isabelle.
[2] Employing (on the meta-level) a lambda-calculus-like notation for writing such functions, one could in this grammar just as well have written $\lambda a.f$ instead of fn.

The permutation action for Φ is defined over the structure as follows:

$$\begin{array}{llll} \pi\bullet\texttt{er} & \stackrel{\text{def}}{=} \texttt{er} & \pi\bullet\texttt{pr}(t_1,t_2) & \stackrel{\text{def}}{=} \texttt{pr}(\pi\bullet t_1,\pi\bullet t_2) \\ \pi\bullet\texttt{am}(a) & \stackrel{\text{def}}{=} \texttt{am}(\pi\bullet a) & \pi\bullet\texttt{se}(fn) & \stackrel{\text{def}}{=} \texttt{se}(\lambda a.\pi\bullet(fn\ (\pi^{-1}\bullet a)) \end{array}$$

where a lambda-term (on the *meta-level*!) specifies how the permutation acts on the function fn, namely as $\pi\bullet fn \stackrel{\text{def}}{=} \lambda a.\pi\bullet(fn\ (\pi^{-1}\bullet a))$.

When reasoning about Λ_α it will save us some work, if we show that certain sets are psets and then show properties (abstractly) for pset-elements.

Lemma 1. The following sets are psets: $\mathbb{A}$, Λ, Φ, and every set of lists (similarly tuples and sets) containing elements from psets.

Proof. By routine inductions. □

The most important notion of a pset-element is that of its *support* (a set of atoms) and derived from this the notion of *freshness*[6]:

Definition 3 (Support and Freshness). *Given an $x \in pset$, its* support *is defined as:*[3]

$$\texttt{supp}(x) \stackrel{def}{=} \{a \mid \texttt{inf}\{b \mid (a\,b)\bullet x \neq x\}\}\ .$$

An atom a is said to be fresh *for such an x, written $a \# x$, provided $a \notin \texttt{supp}(x)$.*

Note that as soon as one fixes the permutation action for elements of a set, the notion of support is fixed as well. That means that Def. 3 defines the support for lists, sets and tuples as long as their elements come from psets. Calculating the support for terms in Λ is simple: $\texttt{supp}(a) = \{a\}$, $\texttt{supp}(t_1\ t_2) = \texttt{supp}(t_1) \cup \texttt{supp}(t_2)$ and $\texttt{supp}(\lambda a.t) = \texttt{supp}(t) \cup \{a\}$. Because of the functions in $\texttt{se}(fn)$, the support for terms in Φ is more subtle. However, later on, we shall see that for terms of the subset Λ_α there is simple structural characterisation for their support, just like for lambda-terms.

First, some properties of support and freshness are established.

Lemma 2. For all $x \in pset$,

(i) $\pi\bullet\texttt{supp}(x) = \texttt{supp}(\pi\bullet x)$, and
(ii) $a \# \pi\bullet x$ if and only if $\pi^{-1}\bullet a \# x$.

Proof. (i) follows from the calculation:

$$\begin{array}{rll} \pi\bullet\texttt{supp}(x) & \stackrel{\text{def}}{=} \pi\bullet\{a \mid \texttt{inf}\{b \mid (a\,b)\bullet x \neq x\}\} & \\ & \stackrel{\text{def}}{=} \{\pi\bullet a \mid \texttt{inf}\{b \mid (a\,b)\bullet x \neq x\}\} & \\ & = \{\pi\bullet a \mid \texttt{inf}\{\pi\bullet b \mid (a\,b)\bullet x \neq x\}\} & (*^1) \\ & = \{a \mid \texttt{inf}\{b \mid (\pi^{-1}\bullet a\ \pi^{-1}\bullet b)\bullet x \neq x\}\} & \\ & = \{a \mid \texttt{inf}\{b \mid \pi\bullet(\pi^{-1}\bullet a\ \pi^{-1}\bullet b)\bullet x \neq \pi\bullet x\}\} & (*^2) \\ & = \{a \mid \texttt{inf}\{b \mid (a\,b)\bullet\pi\bullet x \neq \pi\bullet x\}\} \stackrel{\text{def}}{=} \texttt{supp}(\pi\bullet x) & (*^3) \end{array}$$

[3] The predicate $\texttt{inf}$ will stand for a set being infinite.

where ($*^1$) holds because the sets $\{b| \ldots\}$ and $\{\pi \bullet b| \ldots\}$ have the same number of elements, and where ($*^2$) holds because permutations preserve (in)equalities; ($*^3$) holds because π commutes with the swapping, that is $\pi @ (a\, b) \sim (\pi \bullet a\ \ \pi \bullet b) @ \pi$. (ii): For all π, $a \in \mathtt{supp}(x)$ if and only if $\pi \bullet a \in \pi \bullet \mathtt{supp}(x)$. The property follows then from (i) and $x \in \mathit{pset}$. $\square$

Another important property is the fact that the freshness of two atoms w.r.t. an pset-element means that a permutation swapping those two atoms has no effect:

Lemma 3. For all $x \in \mathit{pset}$, if $a \# x$ and $b \# x$ then $(a\, b) \bullet x = x$.

Proof. The case $a = b$ is clear by Def. $2(i, iii)$. In the other case, the assumption implies that both $\{c \,|\, (c\, a) \bullet x \neq x\}$ and $\{c \,|\, (c\, b) \bullet x \neq x\}$ are finite, and therefore also their union must be finite. Hence the corresponding co-set, that is $\{c \,|\, (c\, a) \bullet x = x \wedge (c\, b) \bullet x = x\}$, is infinite (recall that $\mathbb{A}$ is infinite). If one picks from this co-set one element, which is from now on denoted by c and assumed to be different from a and b, one has $(c\, a) \bullet x = x$ and $(c\, b) \bullet x = x$. Thus $(c\, a) \bullet (c\, b) \bullet (c\, a) \bullet x = x$. The permutations $(c\, a)(c\, b)(c\, a)$ and $(a\, b)$ are equal, since they have an empty disagreement set. Therefore, by using Def. $2(ii, iii)$, one can conclude with $(a\, b) \bullet x = x$. $\square$

A further restriction on psets will filter out all psets containing elements with an infinite support.

Definition 4 (Fs-PSet). *A pset X is said to be an* fs-pset *if every element in X has finite support.*

Lemma 4. The following sets are fs-psets: $\mathbb{A}$, Λ, and every set of lists (similarly tuples and finite sets) containing elements from fs-psets.

Proof. The support of an atom a is $\{a\}$. The support of a lambda-term t is the set of atoms occurring in t. The support of a list is the union of the supports of its elements, and thus finite for fs-pset-elements (ditto tuples and finite sets). $\square$

The set Φ is *not* an fs-pset, because some functions from $\mathbb{A}$ to Φ have an infinite support. Similarly, some infinite sets have infinite support, even if all their elements have finite support. On the other hand, the infinite set $\mathbb{A}$ has *finite* support: $\mathtt{supp}(\mathbb{A}) = \varnothing$ [6]. The main property of elements of fs-psets is that there is always a fresh atom.

Lemma 5. For all $x \in \mathit{fs\text{-}pset}$, there exists an atom a such that $a \# x$.

Proof. Since $\mathbb{A}$ is an infinite set and the support of x is by assumption finite, there must be an $a \notin \mathtt{supp}(x)$. $\square$

We mentioned earlier that we are not going to use all functions from $\mathbb{A}$ to Φ for representing α-equated lambda-abstractions, but some specific functions.[4] The following definition states what properties these functions need to satisfy.

[4] This is in contrast to "weak" and "full" HOAS [15, 4] which use the full function space for representing lambda-abstractions.

Definition 5 (Nominal Abstractions). *An operation, written* $[-].(-)$*, taking an atom and a pset-element is said to be a* nominal abstraction*, if it satisfies the following properties (where* $a \neq b$*):*

(i) $\pi \bullet ([a].x) = [\pi \bullet a].(\pi \bullet x)$
(ii) $[a].x_1 = [b].x_2$ *if and only if either:*

$$a = b \wedge x_1 = x_2, \textit{ or}$$
$$a \neq b \wedge x_1 = (a\,b) \bullet x_2 \wedge a \# x_2$$

The first property states that the permutation action needs to commute with nominal abstractions. The second property ensures that nominal abstractions behave, roughly speaking, like lambda-abstractions. To see this reconsider the rules $\approx_{\lambda 1}$ and $\approx_{\lambda 2}$ given in Fig. 1, which can be used to decide when two lambda-terms are α-equivalent. Property (ii) paraphrases these rules for nominal abstractions. The similarities, however, do not end here: given a $[a].x$ with $x \in$ *fs-pset*, then freshness behaves like $(-) \notin \mathtt{fv}(-)$, as shown next:

Lemma 6. Given $a \neq b$ and $x \in$ *fs-pset*, then

(i) $a \# [b].x$ if and only if $a \# x$, and
(ii) $a \# [a].x$

Proof. (i$\Rightarrow$): Since $x \in$ *fs-pset*, $\mathtt{supp}([b].x) \subseteq \mathtt{supp}(x) \cup \{b\}$ and therefore the support of $[a].x$ must be finite. Hence $(a, b, x, [b].x)$ is finitely supported and by Lem. 5 there exists a c with $(*)$ $c \# (a, b, x, [b].x)$. Using the assumption $a \# [b].x$ and the fact that $c \# [b].x$ (from $*$), Lem. 3 and Def. 5(i) give $[b].x = (c\,a)[b].x = [b].(c\,a) \bullet x$. Hence by Def. 5$(ii)$ $x = (c\,a) \bullet x$. Now $c \# x$ (from $*$) implies that $c \# (c\,a) \bullet x$; and moving the permutation to the other side by Lem. 2(ii) gives $a \# x$. (i$\Leftarrow$): From $(*)$, $c \# [b].x$ and therefore by Lem. 2(ii) $(a\,c) \bullet c \# (a\,c).([b].x)$, which implies by Def. 5(i) that $a \# [b].((a\,c) \bullet x)$. From $(*)$ $c \# x$ holds and from the assumption also $a \# x$; then Lem. 3 implies that $x = (a\,c) \bullet x$, and one can conclude with $a \# [b].x$.
(ii): By $c \# x$ and $c \neq a$ (both from $*$) we can use (i) to infer $c \# [a].x$. Further, from Lem. 2(ii) it holds that $(c\,a) \bullet c \# (c\,a) \bullet [a].x$. This is $a \# [c].(c\,a) \bullet x$ using Def. 5(i). Since $c \neq a$, $c \# x$ and $(c\,a) \bullet x = (c\,a) \bullet x$, Def. 5$(ii)$ implies that $[c].(c\,a) \bullet x = [a].x$. Therefore, $a \# [a].x$. ☐

The functions from $\mathbb{A}$ to Φ we identify next satisfy the nominal abstraction properties. Let $[a].t$ be defined as follows

$$[a].t \stackrel{\text{def}}{=} \mathtt{se}(\lambda b.\, \mathtt{if}\ a = b\ \mathtt{then}\ t\ \mathtt{else}\ \mathtt{if}\ b \# t\ \mathtt{then}\ (a\,b) \bullet t\ \mathtt{else}\ \mathtt{er}) \ . \qquad (2)$$

This operation takes two arguments: an $a \in \mathbb{A}$ and a $t \in \Phi$. To see how this operation encodes an α-equivalence class, consider the α-equivalence class $[\lambda a.(a\ b)]_\alpha$ and the corresponding Φ-term $[a].\mathtt{pr}(a, b)$ (for the moment we ignore the term constructor $\mathtt{se}$ and only consider the function given by $[a].\mathtt{pr}(a, b)$). The graph of this function is as follows: the atom a is mapped to $\mathtt{pr}(a, b)$ since the first $\mathtt{if}$-condition is true. For b, the first $\mathtt{if}$-condition obviously fails, but also the second one fails, because $b \in \mathtt{supp}(\mathtt{pr}(a,b))$; therefore b is mapped to $\mathtt{er}$. For all other atoms c, we have $a \neq c$ and $c \# \mathtt{pr}(a,b)$; so the c's are mapped by the function to $(a\,c){\bullet}\mathtt{pr}(a,b)$, which is just $\mathtt{pr}(c,b)$. Clearly, the function returns $\mathtt{er}$ whenever the corresponding lambda-term is *not* in the α-equivalence class—in this example $\lambda b.(b\ b) \notin [\lambda a.(a\ b)]_\alpha$; in all other cases, however, it returns an appropriately "renamed" version of $\mathtt{pr}(a,b)$.

Lemma 7. The operation $[-].(-)$ given for Φ in (2) is a nominal abstraction.

Proof. Def. 5(i) follows from the calculation:

$$
\begin{aligned}
&\pi{\bullet}[a].t \\
\stackrel{\text{def}}{=}\ &\pi{\bullet}\mathtt{se}(\lambda b.\,\mathtt{if}\ a = b\ \mathtt{then}\ t\ \mathtt{else\ if}\ b \# t\ \mathtt{then}\ (a\,b){\bullet}t\ \mathtt{else\ er}) \\
\stackrel{\text{def}}{=}\ &\mathtt{se}(\lambda b.\,\pi{\bullet}\mathtt{if}\ a = \pi^{-1}{\bullet}b\ \mathtt{then}\ t\ \mathtt{else\ if}\ \pi^{-1}{\bullet}b \# t\ \mathtt{then}\ (a\ \pi^{-1}{\bullet}b){\bullet}t\ \mathtt{else\ er}) \\
=\ &\mathtt{se}(\lambda b.\,\mathtt{if}\ a = \pi^{-1}{\bullet}b\ \mathtt{then}\ \pi{\bullet}t\ \mathtt{else\ if}\ b \# \pi{\bullet}t\ \mathtt{then}\ \pi{\bullet}(a\ \pi^{-1}{\bullet}b){\bullet}t\ \mathtt{else\ er})\ (*) \\
=\ &\mathtt{se}(\lambda b.\,\mathtt{if}\ a = \pi^{-1}{\bullet}b\ \mathtt{then}\ \pi{\bullet}t\ \mathtt{else\ if}\ b \# \pi{\bullet}t\ \mathtt{then}\ (\pi{\bullet}a\ b){\bullet}\pi{\bullet}t\ \mathtt{else\ er}) \\
=\ &\mathtt{se}(\lambda b.\,\mathtt{if}\ \pi{\bullet}a = b\ \mathtt{then}\ \pi{\bullet}t\ \mathtt{else\ if}\ b \# \pi{\bullet}t\ \mathtt{then}\ (\pi{\bullet}a\ b){\bullet}\pi{\bullet}t\ \mathtt{else\ er}) \\
\stackrel{\text{def}}{=}\ &[\pi{\bullet}a].(\pi{\bullet}t)
\end{aligned}
$$

where we use in $(*)$ the fact that $\pi{\bullet}\mathtt{if}...\mathtt{then}...\mathtt{else}... = \mathtt{if}...\mathtt{then}\ \pi{\bullet}...\mathtt{else}\ \pi{\bullet}...$ and Lem 2(ii). In case $a = b$, Def. 5(ii) is by a simple calculation using extensionality of functions. In case $a \neq b$ and Def. 5($ii \Rightarrow$), the following formula can be derived from the assumption by extensionality:

$$
\begin{aligned}
\forall c.\ &\mathtt{if}\ a = c\ \mathtt{then}\ t_1\ \mathtt{else\ if}\ c \# t_1\ \mathtt{then}\ (a\,c){\bullet}t_1\ \mathtt{else\ er} = \\
&\mathtt{if}\ b = c\ \mathtt{then}\ t_2\ \mathtt{else\ if}\ c \# t_2\ \mathtt{then}\ (b\,c){\bullet}t_2\ \mathtt{else\ er}
\end{aligned}
$$

Instantiating this formula once with a and once with b yields the two equations

$$
\begin{aligned}
t_1 &= \mathtt{if}\ a \# t_2\ \mathtt{then}\ (b\,a){\bullet}t_2\ \mathtt{else\ er} \\
t_2 &= \mathtt{if}\ b \# t_1\ \mathtt{then}\ (a\,b){\bullet}t_1\ \mathtt{else\ er}
\end{aligned}
$$

Next, one distinguishes two cases where $a \# t_2$ and $\neg\, a \# t_2$, respectively. In the first case, $t_1 = (b\,a){\bullet}t_2$, which by Lem. 1 and Def. 2(iii) is equal to $(a\,b){\bullet}t_2$; and obviously $a \# t_2$ by assumption. In the second case $t_1 = \mathtt{er}$. This substituted into the second equation gives $t_2 = \mathtt{if}\ b \# \mathtt{er}\ \mathtt{then}\ (a\,b){\bullet}\mathtt{er}\ \mathtt{else\ er}$. Since $\mathtt{supp}(\mathtt{er}) = \varnothing$, $t_2 = (a\,b){\bullet}\mathtt{er} = \mathtt{er}$. Now there is a contradiction with the assumption $\neg\, a \# t_2$, because $a \# \mathtt{er}$. Def. 5($ii \Leftarrow$) for $a \neq b$ is by extensionality and a case-analysis. □

Note that, in *general*, one cannot decide whether two functions from $\mathbb{A}$ to Φ are equal; however Def. 5(ii) provides means to decide whether $[a].t_1 = [b].t_2$ holds: one just has to consider whether $a = b$ and then apply the appropriate property in Def. 5(ii)—just like deciding the α-equivalence of two lambda-terms using $(-)\approx(-)$.

Now everything is in place for defining the subset Λ_α. It is defined inductively by the rules:

$$\frac{a \in \mathbb{A}}{\mathtt{am}(a) \in \Lambda_\alpha} \qquad \frac{t_1 \in \Lambda_\alpha \quad t_2 \in \Lambda_\alpha}{\mathtt{pr}(t_1, t_2) \in \Lambda_\alpha} \qquad \frac{a \in \mathbb{A} \quad t \in \Lambda_\alpha}{[a].t \in \Lambda_\alpha}$$

using in the third inference rule the operation defined in (2). For Λ_α we have:

Lemma 8. Λ_α is:

(i) an fs-pset, and
(ii) closed under permutations, that is if $x \in \Lambda_\alpha$ then $\pi \bullet x \in \Lambda_\alpha$.

Proof. (i): The pset-properties of Φ carry over to Λ_α. The fs-pset property follows by a routine induction on the definition of Λ_α using the fact derived from Lem. 6(i,ii) that for $x \in$ *fs-pset*, $\mathtt{supp}([a].x) = \mathtt{supp}(x) - \{a\}$. (ii) Routine induction over the definition of Λ_α. □

Taking Lem. 8(i) and Lem. 6 together gives us a simple characterisation of the support of elements in Λ_α: $\mathtt{supp}(\mathtt{am}(a)) = \{a\}$, $\mathtt{supp}(\mathtt{pr}(t_1, t_2)) = \mathtt{supp}(t_1) \cup \mathtt{supp}(t_2)$ and $\mathtt{supp}([a].t) = \mathtt{supp}(t) - \{a\}$. In other words it coincides with what one usually means by the free variables of a lambda-term.

Next, one of the main points of this paper: there is a bijection between $\Lambda_{/\approx}$ and Λ_α. This is shown by using the following mapping from Λ to Λ_α:

$$q(a) \stackrel{\text{def}}{=} \mathtt{am}(a) \qquad q(t_1\ t_2) \stackrel{\text{def}}{=} \mathtt{pr}(q(t_1), q(t_2)) \qquad q(\lambda a.t) \stackrel{\text{def}}{=} [a].q(t)$$

and the following lemma:

Lemma 9. $t_1 \approx t_2$ if and only if $q(t_1) = q(t_2)$.

Proof. By routine induction over definition of Λ_α. □

Theorem 1. There is a bijection between $\Lambda_{/\approx}$ and Λ_α.

Proof. The mapping q needs to be lifted to α-equivalence classes (see [14]). For this define $q'([t]_\alpha)$ as follows: apply q to every element of the set $[t]_\alpha$ and build the union of the results. By Lem. 9 this must yield a singleton set. The result of $q'([t]_\alpha)$ is then the singleton. Surjectivity of q' is shown by a routine induction over the definition of Λ_α. Injectivity of q' follows from Lem. 9 since $[t_1]_\alpha = [t_2]_\alpha$ for all $t_1 \approx t_2$. □

4 Structural Induction Principle

The definition of Λ_α provides an induction principle for free. However, this induction principle is not very convenient in practice. Consider Fig. 2 showing a typical informal proof involving lambda-terms—it is Barendregt's proof of the substitution lemma taken from [3]. This informal proof considers in the lambda-case only binders z that have suitable properties (namely being fresh for x, y, N

Substitution Lemma: If $x \not\equiv y$ and $x \notin FV(L)$, then

$$M[x := N][y := L] \equiv M[y := L][x := N[y := L]].$$

Proof: By induction on the structure of M.
Case 1: M is a variable.
Case 1.1. $M \equiv x$. Then both sides equal $N[y := L]$ since $x \not\equiv y$.
Case 1.2. $M \equiv y$. Then both sides equal L, for $x \notin FV(L)$ implies $L[x := \ldots] \equiv L$.
Case 1.3. $M \equiv z \not\equiv x, y$. Then both sides equal z.
Case 2: $M \equiv \lambda z.M_1$. By the variable convention we may assume that $z \not\equiv x, y$ and z is not free in N, L. Then by induction hypothesis

$$\begin{aligned}(\lambda z.M_1)[x := N][y := L] &\equiv \lambda z.(M_1[x := N][y := L])\\ &\equiv \lambda z.(M_1[y := L][x := N[y := L]])\\ &\equiv (\lambda z.M_1)[y := L][x := N[y := L]].\end{aligned}$$

Case 3: $M \equiv M_1 M_2$. The statement follows again from the induction hypothesis. □

Fig. 2. The informal proof of the substitution lemma copied from [3]. In the lambda-case, the variable convention allows Barendregt to move the substitutions under the binder, to apply the induction hypothesis and then to pull out the substitutions.

and L). If we would prove the substitution lemma by induction over the definition of Λ_α, then we would need to show the lambda-case for *all* z, not just the ones being suitably fresh. This would mean we have to rename binders and establish a number of auxiliary lemmas concerning such renamings. In this section we will derive an induction principle which allows a similar convenient reasoning as in Barendregt's informal proof.

For this we only consider induction hypotheses of the form $P\ t\ x$, where P is the property to be proved; P depends on a variable $t \in \Lambda_\alpha$ (over which the induction is done), and a variable x standing for the "other" variables or *context* of the induction. Since x is allowed to be a tuple, several variables can be encoded. In case of the substitution lemma in Fig. 2 the notation $P\ t\ x$ should be understood as follows: the induction variable t is M, the context x is the tuple (x, y, N, L) and the induction hypothesis P is

$$\lambda M.\ \lambda(x, y, N, L).\ M[x := N][y := L] \equiv M[y := L][x := N[y := L]]$$

where we use Isabelle's convenient tuple-notation for the second lambda-abstraction [11]. So by writing $P\ t\ x$ we just make explicit all the variables involved in the induction.

From the inductive definition of Λ_α we can derive a structural induction principle that requires to prove the lambda-case for binders that are fresh for the context x—this is what the variable convention assumes.

Lemma 10 (Induction Principle). Given an induction hypothesis $P\ t\ x$ with $t \in \Lambda_\alpha$ and $x \in$ *fs-pset*, then proving the following:

- $\forall x\ a.\ P\ \mathtt{am}(a)\ x$
- $\forall x\ t_1\ t_2.\ P\ t_1\ x \wedge P\ t_2\ x\ \Rightarrow P\ \mathtt{pr}(t_1, t_2)\ x$
- $\forall x\ a.\ a \# x\ \Rightarrow\ (\forall t.\ P\ t\ x\ \Rightarrow\ P\ [a].t\ x)$

gives $\forall t\ x.\ P\ t\ x$.

Proof. By induction over the definition of Λ_α. We need to strengthen the induction hypothesis to $\forall t\ \pi\ x.\ P\ (\pi \bullet t)\ x$, that means considering t under all permutations π. Only the case for terms of the form $[a].t$ will be explained. We need to show that $P\ (\pi \bullet [a].t)\ x$, where $\pi \bullet [a].t = [\pi \bullet a].(\pi \bullet t)$ by Def. 5(i). By IH, ($*^1$) $\forall \pi\ x.\ P\ (\pi \bullet t)\ x$ holds. Since $x, \pi \bullet t, \pi \bullet a \in$ *fs-pset* holds, one can derive by Lem. 5 that there is a c such that ($*^2$) $c \# (x, \pi \bullet t, \pi \bullet a)$. From $c \# x$ and the assumption, one can further derive $(\forall t.\ P\ t\ x\ \Rightarrow\ P\ [c].t\ x)$. Given ($*^1$) we have that $P\ ((c\ \ \pi \bullet a) :: \pi \bullet t)\ x$ holds and thus also $P\ ([c].((c\ \ \pi \bullet a) :: \pi \bullet t))\ x$. Because of ($*^2$) $c \neq \pi \bullet a$ and $c \# \pi \bullet t$, and by Def. 5(ii) we have that $[c].((c\ \ \pi \bullet a) :: \pi \bullet t = [\pi \bullet a].(\pi \bullet t)$. Therefore we can conclude with $P\ (\pi \bullet [a].t)\ x$. □

With this we have achieved what we set out in the introduction: we have a representation for α-equivalent lambda-terms based on names (for example $[\lambda a.t]_\alpha$ is represented by $[a].t$) and we have an induction principle where the lambda-case needs to be proved for binders that are fresh w.r.t. the variables in the context of the induction, i.e., we can reason as if we had employed a variable convention.

5 Examples

It is reasonably straightforward to implement the results from Sec. 3 and 4 in Isabelle/HOL: the set Φ is an inductive datatype, the pset and fs-pset properties can be formulated as axiomatic type-classes [20], and the subset Λ_α can be defined using the Isabelle's `typedef`-mechanism. This section focuses on how reasoning over Λ_α pans out in practice.

The first obstacle is that so far Isabelle's datatype package is not general enough to allow a direct definition of functions over Λ_α: although Λ_α contains only terms of the form $\mathtt{am}(a)$, $\mathtt{pr}(t_1, t_2)$ and $[a].t$, pattern-matching in Isabelle requires the injectivity of term-constructors. But clearly, $[a].t$ is *not* injective. Fortunately, one can work around this obstacle by, roughly speaking, defining functions as inductive relations and then use the definite description operator *THE* of Isabelle to turn the relations into functions.

We give an example: capture-avoiding substitution can be defined as a four-place relation (the first argument contains the term into which something is being substituted, the second the variable that is substituted for, the third the term that is substituted, and the last contains the result of the substitution):

```
consts Subst :: "(Λα × 𝔸 × Λα × Λα) set"
inductive Subst
intros
s1: "(am(a),a,t',t')∈Subst"
s2: "a≠b ⟹ (am(b),a,t',am(b))∈Subst"
s3: "⟦(s₁,a,t',s₁')∈Subst; (s₂,a,t',s₂')∈Subst⟧
                                ⟹ (pr(s₁,s₂),a,t',pr(s₁',s₂'))∈Subst"
s4: "⟦b#(a,t');(s,a,t',s')∈Subst⟧ ⟹ ([b].s,a,t',[b].s')∈Subst"
```

While on first sight this relation looks as if it defined a non-total function, one should be careful! Clearly, the lambda-case (i.e. `([b].s,a,t',[b].s')` $\in$ `Subst`) holds only under the precondition `b#(a,s)`—roughly meaning that $a \neq b$ and b cannot occur freely in s. However, `Subst` *does* define a total function, because `Subst` is defined over α-equivalent lambda-terms (more precisely Λ_α), *not* over lambda-terms. We can indeed show "totality":

Lemma 11. *For all* $\mathtt{t}_1$, $\mathtt{a}$, $\mathtt{t}_2$, $\exists \mathtt{t}_3.\ (\mathtt{t}_1, \mathtt{a}, \mathtt{t}_2, \mathtt{t}_3) \in \mathtt{Subst}$.

Proof. The proof in Isabelle/HOL uses the induction principle derived in Thm. 10. It is as follows:

```
proof (nominal_induct t₁)
   case (1 b) (* variable case *)
   show "∃t₃. (am(b),a,t₂,t₃)∈Subst" by (cases "b=a") (force+)
next
   case (2 s₁ s₂) (* application case *)
   thus "∃t₃. (pr(s₁,s₂),a,t₂,t₃)∈Subst" by force
next
   case (3 b s) (* lambda case *)
   thus "∃t₃. ([b].s,a,t₂,t₃)∈Subst" by force
qed
```

The induction method `nominal_induct` brings the induction hypothesis automatically into the form

$$\underbrace{(\lambda \mathtt{t}_1\ \lambda(\mathtt{a}, \mathtt{t}_2).\ \exists \mathtt{t}_3.(\mathtt{t}_1, \mathtt{a}, \mathtt{t}_2, \mathtt{t}_3) \in \mathtt{Subst})}_{P}\ \underbrace{\mathtt{t}_1}_{t}\ \underbrace{(\mathtt{a}, \mathtt{t}_2)}_{x}$$

by collecting all free variables in the goal, and then it applies Thm. 10. This results in three cases to be proved—variable case, application case and lambda-case. The requirement that the context $(\mathtt{a}, \mathtt{t}_2)$ is a *fs-pset*-element is enforced by using axiomatic type-classes and relying on Isabelle's type-system. Note that in the lambda-case it is important to know that the binder `b` is fresh for `a` and $\mathtt{t}_2$. The proof obligation in this case is:

$$\mathtt{b} \# (\mathtt{a}, \mathtt{t}_2) \wedge \exists \mathtt{t}_3.(\mathtt{s}, \mathtt{a}, \mathtt{t}_2, \mathtt{t}_3) \ \text{ implies } \ \exists \mathtt{t}_3.([\mathtt{b}].\mathtt{s}, \mathtt{a}, \mathtt{t}_2, \mathtt{t}_3)$$

which can be easily be shown by rule `s4`. As a result, the only case in which we really need to manually "interfere" is in the variable case where we have to give Isabelle the hint to distinguish the cases $b = a$ and $b \neq a$. □

Together with a uniqueness-lemma (whose proof we omit) asserting that

$$\forall \mathtt{s}_1 \mathtt{s}_2. (\mathtt{t}_1, \mathtt{a}, \mathtt{t}_2, \mathtt{s}_1) \in \mathtt{Subst} \wedge (\mathtt{t}_1, \mathtt{a}, \mathtt{t}_2, \mathtt{s}_2) \in \mathtt{Subst} \Rightarrow \mathtt{s}_1 = \mathtt{s}_2 \tag{3}$$

one can prove the stronger totality-property, namely for all $\mathtt{t}_1$, $\mathtt{a}$, $\mathtt{t}_2$:

$$\exists! \mathtt{t}_3.\ (\mathtt{t}_1, \mathtt{a}, \mathtt{t}_2, \mathtt{t}_3) \in \mathtt{Subst}\ . \tag{4}$$

Having this at our disposal, we can use Isabelle's definite description operator *THE* and turn capture-avoiding substitution into a function; we write this function as $(-)[(-) := (-)]$, and establish the equations:

$$\begin{array}{rll} \mathtt{am}(\mathtt{a})[\mathtt{a} := \mathtt{t}] = \mathtt{t} & & \\ \mathtt{am}(\mathtt{b})[\mathtt{a} := \mathtt{t}] = \mathtt{am}(\mathtt{b}) & & \text{provided } \mathtt{a} \neq \mathtt{b} \\ \mathtt{pr}(\mathtt{s}_1, \mathtt{s}_2)[\mathtt{a} := \mathtt{t}] = \mathtt{pr}(\mathtt{s}_1[\mathtt{a} := \mathtt{t}], \mathtt{s}_2[\mathtt{a} := \mathtt{t}]) & & \\ ([\mathtt{b}].\mathtt{s})[\mathtt{a} := \mathtt{t}] = [\mathtt{b}].(\mathtt{s}[\mathtt{a} := \mathtt{t}]) & & \text{provided } \mathtt{b} \# (\mathtt{a}, \mathtt{t}) \end{array} \tag{5}$$

These equations can be supplied to Isabelle's simplifier and one can reason about substitution "just like on paper". For this we give in Fig. 3 one *simple* example as evidence—giving the whole formalised Church-Rosser proof from [3, p. 60–62] would be beyond the space constraints of this paper. The complete formalisations of all the results, the Church-Rosser and strong normalisation proof is at `http://www.mathematik.uni-muenchen.de/~urban/nominal/` .

6 Related Work

There are many approaches to formal treatments of binders; this section describes the ones from which we have drawn inspiration.

Our work uses many ideas from the nominal logic work by Pitts *et al* [16, 6]. The main difference is that by constructing, so to say, an explicit model of the α-equated lambda-terms based on functions, we have no problem with the axiom-of-choice. This is important. For consider the alternative: if the axiom-of-choice causes inconsistencies, then one cannot build a framework for binding on top of Isabelle/HOL with its rich reasoning infrastructure. One would have to interface on a lower level and has to redo the effort that has been spend to develop Isabelle/HOL. This was attempted in [5], but the attempt was later abandoned.

Closely related to our work is [9] by Gordon and Melham; it has been applied and further developed by Norrish [13]. This work states five axioms characterising α-equivalence and then shows that a model based on de-Bruijn indices satisfies the axioms. This is somewhat similar to our approach where we construct explicitly the set Λ_α. In [9] they give an induction principle that requires in the lambda-case to prove (using their notation)

$$\forall\, x\, t.\ (\forall\, v.\ P\,(t[x := \mathit{VAR}\ v])) \Longrightarrow P\,(\mathit{LAM}\ x\ t)$$

That means they have to prove $P(\mathit{LAM}\ x\ t)$ for a variable x for which nothing can be assumed; explicit α-renamings are then necessary in order to get the

```
lemma substitution_lemma:
assumes a1: "x ≠ y"
    and a2: "x # L"
shows "M[x:=N][y:=L] = M[y:=L][x:=N[y:=L]]"
proof (nominal_induct M)
 case (1 z) (* case 1: variables *)
 have "z=x ∨ (z≠x ∧ z=y) ∨ (z≠x ∧ z≠y)" by force
 thus "am(z)[x:=N][y:=L] = am(z)[y:=L][x:=N[y:=L]]"
   using a1 a2 forget by force
next
 case (2 z M1) (* case 2: lambdas *)
 assume ih: "M1[x:=N][y:=L] = M1[y:=L][x:=N[y:=L]]"
 assume f1: "z # (L,N,x,y)"
 from f1 fresh_fact1 have f2: "z # N[y:=L]" by simp
 show "([z].M1)[x:=N][y:=L]=([z].M1)[y:=][x:=N[y:=L]]" (is "?LHS=?RHS")
 proof -
   have "?LHS = [z].(M1[x:=N][y:=L])" using f1 by simp
   also have "...= [z].(M1[y:=L][x:=N[y:=L]])" using ih by simp
   also have "...= ([z].(M1[y:=L]))[x:=N[y:=L]]" using f1 f2 by simp
   also have "...= ?RHS" using f1 by simp
   finally show "?LHS = ?RHS" by simp
 qed
next
 case (3 M1 M2) (* case 3: applications *)
 thus "pr(M1,M2)[x:=N][y:=L]=pr(M1,M2)[y:=L][x:=N[y:=L]]" by simp
qed
```

Fig. 3. An Isabelle proof using the Isar language for the substitution lemma shown in Fig. 2. It uses the following auxiliary lemmas: `forget` which states that `x # L` implies `L[x:=T]=L`, needed in the variable case. This case proceeds by stating the three subcases to be considered and then proving them automatically using the assumptions `a1` and `a2`. The lemma `fresh_fact1` in the lambda-case shows from `z # (L,N,x,y)` that `z # N[x:=L]` holds. This lemma is not explicitly mentioned in Barendregt's informal proof, but it is necessary to pull out the substitution from under the binder `z`. This case proceeds as follows: the substitutions on left-hand side of the equation can be moved under the binder `z`; then one can apply the induction hypothesis; after this one can pull out the second substitution using `z # N[y:=L]` and finally move out the first substitution using `z # (L,N,x,y)`. This gives the right-hand side of the equation.

proof through. This inconvenience has been alleviated by the version of structural induction given in [8] and [12], which is as follows

$$\exists X.\, \texttt{FINITE}\ X \wedge (\forall\, x\, t.\, x \notin X \wedge P\ t \Longrightarrow P\ (LAM\ x\ t))$$

For this principle one has to provide a finite set X and then has to show the lambda-case for all binders not in this set. This is very similar to our induction principle, but we claim that our version based on freshness fits better with informal practise and can make use of the infrastructure of Isabelle (namely the axiomatic type-classes enforce the finite-support property).

Like our Λ_α, HOAS uses functions to encode lambda-abstractions; it comes in two flavours: *weak* HOAS [4] and *full* HOAS [15]. The advantage of full HOAS over our work is that notions such as capture-avoiding substitution come for free. We, on the other hand, load the work of such definitions onto the user. The advantage of our work is that we have no difficulties with notions such as simultaneous-substitution (a crucial notion in the usual strong normalisation proof), which in full HOAS seem rather difficult to encode. Another advantage we see is that by inductively defining Λ_α one has induction for "free", whereas induction requires considerable effort in full HOAS. The main difference of our work with weak HOAS is that we use *some* specific functions to represent lambda-abstractions; in contrast, weak HOAS uses the *full* function space. This causes problems known by the term "exotic terms"—essentially junk in the model.

7 Conclusion

The paper [2], which sets out some challenges for automated proof assistants, claims that theorem proving technologies have almost reached the threshold where they can be used *by the masses* for formal reasoning about programming languages. We hope to have pushed with this paper the boundary of the state-of-the-art in formal reasoning closer to this threshold. We showed all our results for the lambda-calculus. But the lambda-calculus is only *one* example. We envisage no problems generalising our results to other term-calculi. In fact, there is already work by Bengtson adapting our results to the π-calculus. We also do not envisage problems with providing a general framework for reasoning about binders based on our results. The real (implementation) challenge is to integrate these results into Isabelle's datatype package so that the user does not see any of the tedious details through which we had to go. For example one would like that the subset construction from a bigger set is done completely behind the scenes. Deriving an induction principle should also be done automatically. Ideally, a user just defines an inductive datatype and indicates where binders are—the rest of the infrastructure should be provided by the theorem prover. This is future work.

Acknowledgements: The first author is very grateful to Andrew Pitts and Michael Norrish for the many discussions with them on the subject of the paper. We thank James Cheney, Aaron Bohannon, Daniel Wang and one anonymous referee for their suggestions. The first author's interest in this work was sparked by an email-discussion with Frank Pfenning and by a question from Neil Ghani at the spring school of Midland Graduate School. The Alexander-von-Humboldt Foundation funded the first author.

References

1. T. Altenkirch. A Formalization of the Strong Normalisation Proof for System F in LEGO. In *Proc. of TLCA*, volume 664 of *LNCS*, pages 13–28, 1993.
2. B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized Metatheory for the Masses: The PoplMark Challenge. accepted at tphol 05.
3. H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1981.
4. J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-Order Abstract Syntax in Coq. In *Proc. of TLCA*, volume 902 of *LNCS*, pages 124–138, 1995.
5. M. J. Gabbay. *A Theory of Inductive Definitions With α-equivalence*. PhD thesis, University of Cambridge, 2000.
6. M. J. Gabbay and A. M. Pitts. A New Approach to Abstract Syntax with Variable Binding. *Formal Aspects of Computing*, 13:341–363, 2001.
7. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.
8. A. D. Gordon. A Mechanisation of Name-Carrying Syntax up to Alpha-Conversion. In *Proc. of Higher-order logic theorem proving and its applications*, volume 780 of *LNCS*, pages 414–426, 1993.
9. A. D. Gordon and T. Melham. Five Axioms of Alpha-Conversion. In *Proc. of TPHOL*, volume 1125 of *LNCS*, pages 173–190, 1996.
10. D. Hirschkoff. A Full Formalisation of π-Calculus Theory in the Calculus of Constructions. In *Proc. of TPHOL*, volume 1275 of *LNCS*, pages 153–169, 1997.
11. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.
12. M. Norrish. Mechanising λ-calculus using a Classical First Order Theory of Terms with Permutations, forthcoming.
13. M. Norrish. Recursive Function Definition for Types with Binders. In *Proc. of TPHOL*, volume 3223 of *LNCS*, pages 241–256, 2004.
14. L. Paulson. Defining Functions on Equivalence Classes. To appear in ACM Transactions on Computational Logic.
15. F. Pfenning and C. Elliott. Higher-Order Abstract Syntax. In *Proc. of the ACM SIGPLAN Conference PLDI*, pages 199–208. ACM Press, 1989.
16. A. M. Pitts. Nominal Logic, A First Order Theory of Names and Binding. *Information and Computation*, 186:165–193, 2003.
17. A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*, volume 43 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2000.
18. C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal Unification. *Theoretical Computer Science*, 323(1-2):473–497, 2004.
19. M. VanInwegen. *The Machine-Assisted Proof of Programming Language Properties*. PhD thesis, University of Pennsylvania, 1996. Available as MS-CIS-96-31.
20. M. Wenzel. *Using Axiomatic Type Classes in Isabelle*. Manual in the Isabelle distribution.