J Autom Reasoning (2008) 40:327–356
DOI 10.1007/s10817-008-9097-2

Nominal Techniques in Isabelle/HOL

Christian Urban

Received: 25 September 2006 / Accepted: 1 January 2008 / Published online: 14 March 2008

Abstract This paper describes a formalisation of the lambda-calculus in a HOL-based theorem prover using nominal techniques. Central to the formalisation is an inductive set that is bijective with the alpha-equated lambda-terms. Unlike de-Bruijn indices, however, this inductive set includes names and reasoning about it is very similar to informal reasoning with "pencil and paper". To show this we provide a structural induction principle that requires to prove the lambda-case for fresh binders only. Furthermore, we adapt work by Pitts providing a recursion combinator for the inductive set. 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, strong-normalisation of beta-reduction, the correctness of the type-inference algorithm W, typical proofs from SOS and much more.

Keywords Lambda-calculus · Nominal logic work · Theorem provers

1 Introduction

> We thank T. Thacher Robinson for showing us on August 19, 1962 by a counterexample the existence of an error in our handling of bound variables.
>
> S. C. Kleene [17, Page 16]

When reasoning informally about syntax, issues with binders and alpha-equivalence are almost universally perceived as unimportant and thus mostly ignored. However,

This paper is a revised and much extended version of Urban and Berghofer [32], and Urban and Tasson [36].

C. Urban (✉)
Technical University Munich, Munich, Germany
e-mail: urbanc@in.tum.de

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. 1 An informal proof of the substitution lemma taken from Barendregt's book [5]. In second case, the variable convention allows him to move the substitutions under the binder, to apply the induction hypothesis and finally to pull the substitutions back out from under the binder

errors *do* arise from these issues as the quotation from Kleene shows. It is therefore desirable to have convenient techniques for formalising informal proofs. In this paper such a technique is described in the context of the lambda-calculus and the theorem prover Isabelle/HOL. However, the techniques generalise to more complex calculi and parts have already been adapted in HOL4, HOL-light and Coq.

The main point of this paper is to give a representation for *alpha-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. Furthermore, we give a structural recursion combinator for defining functions over this set. In practice this will mean that we come quite close to the informal reasoning using Barendregt's variable convention [5]. An illustrative example of such informal reasoning is Barendregt's proof of the substitution lemma shown in Fig. 1. In this paper we describe a reasoning infrastructure for formalising such informal proofs with ease. This reasoning infrastructure has been implemented in Isabelle/HOL as part of the nominal datatype package.[1]

Our work is based on the nominal logic work by Pitts et al. [11, 26]. The main technical novelty is that our work is compatible with the axiom of choice. This is important, because otherwise we would not be able to built in a HOL-based theorem prover a framework for reasoning based on nominal techniques. The reason why the original nominal logic work is incompatible with the axiom of choice has to do with the way how the finite support property is enforced: FM-set theory is defined in [11] so that every set in the FM-set-universe has finite support. In nominal logic [26], the axioms (E3) and (E4) imply that every function symbol and proposition has finite support. However, there are notions in HOL that do *not* have finite support, most notably choice functions (see [27, Example 3.4, Page 470]). Here, we will avoid the incompatibility with the axiom of choice by not a priory restricting our discourse to only finitely supported entities as done previously, rather we will explicitly assume this property whenever it is needed in proofs. One consequence is that we state our

[1] Available from http://isabelle.in.tum.de/nominal.

basic definitions not in terms of nominal sets (as done for example in [27]), but in terms of the weaker notion of permutation types—essentially sets equipped with a "sensible" notion of permutation operation.

The paper is organised as follow: Section 2 introduces the basic notions of the nominal logic work adapted to our Isabelle/HOL setting. Section 3 first reviews alpha-equivalence for lambda-terms and then gives a construction of an inductive set that is bijective with the alpha-equated lambda-terms. Two structural induction principles for this set are derived in Section 4. Recent work by Pitts [27] is adapted in Section 5 to give a structural recursion combinator for defining functions over the bijective set. Section 6 gives examples; related work is mentioned in Section 7 and Section 8 concludes.

2 Atoms, Permutations and Support

In the lambda-calculus there is a single type of bindable names, here denoted by `name`, whose elements in the tradition of the nominal logic work we call *atoms*. While the structure of atoms is immaterial, two properties need to hold for the type `name`: one has to be able to distinguishing different atoms and one needs to know that there are countably infinitely many of them. This can be achieved in Isabelle/HOL by implementing the type `name` as natural numbers or strings.

Permutations are finite bijective mappings from `name` to `name`. They can be represented as finite lists whose elements are swappings (i.e. pairs of atoms). In what follows the type-abbreviation `name prm` will stand for the type of permutations, that is (`name` × `name`) `list`, and we will write permutations as

$$(a_1\,b_1)(a_2\,b_2)\cdots(a_n\,b_n)$$

with the empty list [] standing for the identity permutation. The operation of a permutation π *acting* on an atom a is defined as:

$$\begin{aligned}
[]\bullet a &\overset{\text{def}}{=} a\\
((a_1\,a_2) :: \pi)\bullet a &\overset{\text{def}}{=} \begin{cases} a_2 & \text{if } \pi\bullet a = a_1\\ a_1 & \text{if } \pi\bullet a = a_2\\ \pi\bullet a & \text{otherwise}\end{cases}
\end{aligned} \tag{1}$$

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}.

Our representation of permutations as lists does not give unique representatives: for example, the permutation $(a\,a)$ is "equal" to the identity permutation. We equate the representations of permutations with a relation $\sim$:

Definition 1 (Permutation Equality) Two permutations are *equal*, written $\pi_1 \sim \pi_2$, provided $\pi_1\bullet a = \pi_2\bullet a$ for all atoms a.

To generalise the notion given in (1) of a permutation acting on an atom, we take advantage of the overloading mechanism in Isabelle by declaring a constant, written infix as $(-)\bullet(-)$, with the polymorphic type `name prm` $\Rightarrow \alpha \Rightarrow \alpha$. A definition of the permutation operation can then be given separately for each type-constructor; for lists, products, unit, sets, functions, options and booleans the definitions are as follows:

$$\begin{array}{lrcl}
\alpha\ \texttt{list}: & \pi\bullet[] & \overset{\text{def}}{=} & [] \\
 & \pi\bullet(x :: t) & \overset{\text{def}}{=} & (\pi\bullet x) :: (\pi\bullet t) \\
\alpha_1 \times \alpha_2: & \pi\bullet(x_1, x_2) & \overset{\text{def}}{=} & (\pi\bullet x_1, \pi\bullet x_2) \\
\texttt{unit}: & \pi\bullet() & \overset{\text{def}}{=} & () \\
\alpha\ \texttt{set}: & \pi\bullet X & \overset{\text{def}}{=} & \{\pi\bullet x \mid x \in X\} \\
\alpha_1 \Rightarrow \alpha_2: & \pi\bullet fn & \overset{\text{def}}{=} & \lambda x.\pi\bullet(fn\,(\pi^{-1}\bullet x)) \\
\alpha\ \texttt{option}: & \pi\bullet None & \overset{\text{def}}{=} & None \\
 & \pi\bullet Some(x) & \overset{\text{def}}{=} & Some(\pi\bullet x) \\
\texttt{bool}: & \pi\bullet b & \overset{\text{def}}{=} & b
\end{array} \tag{2}$$

It will save much work later on to *not* establish properties for each of these permutation operations individually, but reason abstractly over them by requiring that every permutation operation satisfies three basic properties:

Definition 2 (Permutation Type) A type α will be referred to as *permutation type*, written pt_α, provided the permutation operation satisfies the following three properties:

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

These properties entail that the permutations operation behaves over permutation types as one expects:

Lemma 1 *Assuming x and y are of permutation type then:*

(i) $\pi^{-1}\bullet(\pi\bullet x) = x$,
(ii) $\pi\bullet x = y$ *if and only if* $x = \pi^{-1}\bullet y$,
(iii) $\pi\bullet x = \pi\bullet y$ *if and only if* $x = y$, *and*
(iv) $\pi\bullet x \in \pi\bullet X$ *if and only if* $x \in X$.

Proof The first property holds by Definition 2(i-iii) since $(\pi^{-1} @ \pi) \sim []$, which can be shown by an induction over the length of π. The second property follows from the first. The third is a consequence of the first and second. For the fourth one has to unwind the definition of the permutation operation for sets and apply the third property. □

Using Isabelle's *axiomatic type-classes* [37], it is very convenient to ensure that a type is a permutation type because most of the routine work can be performed by the type-checking algorithm of Isabelle: one only has to establish that some "base" types, such as `name` and `unit`, are permutation types and that type-constructors, such as products and lists, preserve the property of being a permutation type. More formally we have:

Lemma 2 *Given pt_α, pt_{α_1} and pt_{α_2}, the types* `name`, `unit`, α `list`, α `set`, α `option`, $\alpha_1 \times \alpha_2$, $\alpha_1 \Rightarrow \alpha_2$ *and* `bool` *are also permutation types.*

Proof All properties follow by unwinding the definition of the corresponding permutation operation and routine inductions. The property $pt_{\alpha_1 \Rightarrow \alpha_2}$ uses the fact that $\pi_1 \sim \pi_2$ implies $\pi_1^{-1} \sim \pi_2^{-1}$.

Note that the permutation operation over a function-type, say $\alpha_1 \Rightarrow \alpha_2$ with α_1 being a permutation type, is defined so that for every function *fn* we have the equation

$$\pi \bullet (fn\, x) = (\pi \bullet fn)(\pi \bullet x) \tag{3}$$

in Isabelle/HOL; this is because we have $\pi^{-1} \bullet (\pi \bullet x) = x$ by Lemma 1(i) and $\pi \bullet fn = \lambda x. \pi \bullet (fn\, (\pi^{-1} \bullet x))$ by definition of permutations acting on functions.

The most interesting feature of the nominal logic work is that as soon as one fixes a "sensible" permutation operation for a type, then the *support* for the elements of this type, very roughly speaking their set of free atoms, is fixed as well. The definition of support and the derived notion of freshness is:

Definition 3 (Support and Freshness) The support of x, written $supp(x)$, is the set of atoms defined as:

$$supp(x) \overset{\text{def}}{=} \{a \mid infinite\{b \mid (a\, b) \bullet x \neq x\}\}$$

where $infinite(-)$ means that the set is infinite.[2] An atom a is said to be *fresh* for an x, written $a \,\#\, x$, provided $a \notin supp(x)$.

Intuitively, this definition says that a is fresh for x if and only if $(a\, b) \bullet x = x$ holds for all but finitely many b. Unwinding this definition and the permutation operations given in (2), one can often easily calculate the support for "finitary" permutation types such as:

$$\begin{array}{ll}
\texttt{name}: & supp(a) = \{a\} \\
\alpha\ \texttt{list}: & supp([]) = \varnothing \\
 & supp(x :: xs) = supp(x) \cup supp(xs) \\
\alpha_1 \times \alpha_2: & supp((x_1, x_2)) = supp(x_1) \cup supp(x_2) \\
\texttt{unit}: & supp(()) = \varnothing \\
\alpha\ \texttt{option}: & supp(None) = \varnothing \\
 & supp(Some(x)) = supp(x) \\
\texttt{bool}: & supp(b) = \varnothing
\end{array} \tag{4}$$

[2]In Isabelle/HOL the predicate *infinite* is defined as "not a finite set" with the predicate for a set being finite defined inductively starting with the empty set and by adding elements.

More subtle is the calculation of the support for "infinitary" permutation types such as functions and infinite sets. However, the use of the notion of support, as opposed to the usual notion of free atoms, is crucial for this work: the bijective set we describe in the next section includes some functions, and for those it is far from obvious what the definition of the set of free atoms should be (the obstacle is to find an appropriate definition for free variables of functions with type, say $\alpha_1 \Rightarrow \alpha_2$, in terms of the free variables for elements of the type α_1 and α_2). Contrast this with the definition of permutation for functions given in (2), which is defined in terms of the permutation acting on the domain and co-domain of functions. It will turn out that, albeit slightly unwieldy, Definition 3 coincides exactly with what one intuitively associates with the set of free atoms for the functions we shall use.

For permutation types the notion of support and freshness have good properties: we first show that the support and the permutation operation commute and that permutation preserve freshness.[3]

Lemma 3 *For all x of permutation type:*

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

Proof The first property follows from the calculation:

$$\begin{aligned}
\pi \bullet supp(x) &\overset{\text{def}}{=} \pi \bullet \{a \,|\, infinite\{b \mid (a\,b) \bullet x \neq x\}\} \\
&\overset{\text{def}}{=} \{\pi \bullet a \,|\, infinite\{b \mid (a\,b) \bullet x \neq x\}\} \\
&= \{\pi \bullet a \,|\, infinite\{\pi \bullet b \mid (a\,b) \bullet x \neq x\}\} && (*^1) \\
&= \{a \,|\, infinite\{b \mid (\pi^{-1} \bullet a \;\; \pi^{-1} \bullet b) \bullet x \neq x\}\} \\
&= \{a \,|\, infinite\{b \mid \pi \bullet (\pi^{-1} \bullet a \;\; \pi^{-1} \bullet b) \bullet x \neq \pi \bullet x\}\} && (*^2) \\
&= \{a \,|\, infinite\{b \mid (a\,b) \bullet \pi \bullet x \neq \pi \bullet x\}\} \overset{\text{def}}{=} supp(\pi \bullet x) && (*^3)
\end{aligned}$$

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 by Lemma 1(ii) (in)equalities; $(*^3)$ holds because π commutes with the swapping, that is $\pi @ (c\,d) \sim (\pi \bullet c \;\; \pi \bullet d) @ \pi$ for all atoms c and d. For the second and third property we have by Lemma 1(iv) that $a \in supp(x)$ if and only if $\pi \bullet a \in \pi \bullet supp(x)$; they then follow from *(i)* and Lemma 1(i). □

Another important property of freshness is the fact that if two atoms are fresh w.r.t. an element of a permutation type then the permutation swapping those two atoms in this element has no effect:

Lemma 4 *For all x of permutation type, if $a \,\#\, x$ and $b \,\#\, x$ then $(a\,b) \bullet x = x$.*

[3]Pitts gives in [27] a simpler proof for *(i)*, but in a more restricted setting, namely where x has finite support. Our lemma is more general as we only require x to be of permutation type.

Proof The case $a = b$ is clear by Definition 2(i,iii) and the fact that $(a\,a) \sim []$. In the other case, the assumption implies that both sets $\{c \mid (c\,a)\bullet x \neq x\}$ and $\{c \mid (c\,b)\bullet x \neq x\}$ are finite, and therefore also their union must be finite. Hence the corresponding co-set, that is $\{c \mid (c\,a)\bullet x = x \wedge (c\,b)\bullet x = x\}$, is infinite (recall that there are infinitely many atoms). If one picks from this co-set one element, say c, which can be assumed to be different from a and b, one has $(c\,a)\bullet x = x$ and $(c\,b)\bullet x = x$. Thus $(c\,a)\bullet(c\,b)\bullet(c\,a)\bullet x = x$. Under the assumptions $a \neq c$, $b \neq c$ $a \neq b$, the permutations $(c\,a)(c\,b)(c\,a)$ and $(a\,b)$ are equal. Therefore one can conclude with $(a\,b)\bullet x = x$ by using Definition 2(ii,iii). □

A further restriction on permutation types filters out all those that contain elements with infinite support:

Definition 4 (Finitely Supported Permutation Types) A permutation type α is said to be *finitely supported*, written fs_α, if every element of α has finite support.

We shall write `finite`$(supp(x))$ to indicate that an element x from a permutation type has finite support. The following holds:

Lemma 5 *Given* fs_α, fs_{α_1} *and* fs_{α_2}, *the types* `name`, `unit`, α `list`, α `option`, $\alpha_1 \times \alpha_2$ *and* `bool` *are also finitely supported permutation types.*

Proof Routine proofs using the calculations given in (4).

The crucial property entailed by Definition 4 is that if an element, say x, of a permutation type has finite support, then there must be a fresh atom for x, since there are infinitely many atoms. Therefore we have:

Proposition 1 *If x of permutation type has finite support, then there exists an atom a with $a \# x$.*

As a result, whenever we need to have a fresh atom for an x of permutation type, we have to make sure that x has finite support. This task can be automatically performed by Isabelle's axiomatic type-classes for most constructions occurring in informal proofs: Isabelle has to just examine the types of the construction using Lemma 5.

Proposition 1 also implies that for every finitely supported function a fresh atom exists. However, to determine whether a function has finite support is more subtle, because not all functions are finitely supported, even if their domain and codomain are finitely supported permutation types (see [27, Example 3.4, Page 470]). Introducing a finitely supported function space and blending it well into Isabelle's reasoning infrastructure seems impractical for reasons how Isabelle is implemented. So for functions one has to "manually" ensure finite support, which we shall do in Section 5 by introducing a weaker notion that approximates the support of an element from "above".

3 Constructing a Representation for Alpha-Equated Lambda-Terms

In this section we define an inductive set that is bijective with the set of alpha-equated lambda-terms. In doing so our goal is to give in Isabelle/HOL a formal implementation of the usual convention (from Barendregt [5, Page 26]) employed explicitly or implicitly in many informal proofs:

> CONVENTION. Terms that are α-congruent are identified. So now we write $\lambda x.x \equiv \lambda y.y$, etcetera.

We begin with defining "raw" lambda-terms. They can be defined in Isabelle/HOL with the datatype declaration:

```
datatype lam = Var "name"
             | App "lam × lam"
             | Lam "name × lam"
```
(5)

Given the following permutation operation for lambda-terms

$$\begin{aligned} \pi \bullet \mathtt{Var}(a) &\overset{\text{def}}{=} \mathtt{Var}(\pi \bullet a) \\ \pi \bullet \mathtt{App}(t_1, t_2) &\overset{\text{def}}{=} \mathtt{App}(\pi \bullet t_1, \pi \bullet t_2) \\ \pi \bullet \mathtt{Lam}(a, t) &\overset{\text{def}}{=} \mathtt{Lam}(\pi \bullet a, \pi \bullet t) \end{aligned} \tag{6}$$

the datatype `lam` is a permutation type (routine proof by structural induction). As mentioned earlier, fixing the permutation operation also fixes the notion of support, which in case of `lam` coincides with the set of *all* atoms occurring in a lambda-term. Hence `lam` is a finitely supported permutation type.

The notion of alpha-equivalence for `lam` is usually defined as the least congruence of the equation $\mathtt{Lam}(a, t) =_\alpha \mathtt{Lam}(b, t[a := b])$ 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. This has a number of technical advantages (compare the technical subtleties of Dowek et al. [9] with the approach in Urban et al [35]), because permutations are bijections on atoms, while renaming substitution might identify some atoms. As a consequence of the bijectivity, a renaming based on permutations preserves the binding structure. In contrast, applying naïvely a renaming substitution one might identify an atom that is bound with one that is free.

Using the permutation operation given in (6), alpha-equivalence for `lam` can be defined in a simple and syntax directed fashion using the relations $(-)\approx(-)$ and $(-) \notin \mathtt{fv}(-)$ whose rules are given in Fig. 2. Because of the "asymmetric" rule $\approx_{\mathtt{Lam2}}$, it might be surprising, but:

Proposition 2 *The relation $\approx$ is an equivalence relation.*

The proof of this proposition is omitted: it can be found in a more general setting in Urban et al. [35]. (We also omit a proof showing that $\approx$ and $=_\alpha$ coincide). In the following, $[t]_\alpha$ will stand for the alpha-equivalence class of the lambda-term t, that is $[t]_\alpha \overset{\text{def}}{=} \{\, t' \mid t' \approx t \,\}$, and $\mathtt{lam}_{/\approx}$ for the set of lambda-terms quotient by $\approx$.

Next we will define a set `phi`; inside this set we will subsequently identify (inductively) a subset, called $\mathtt{lam}_\alpha$, that is in bijection with $\mathtt{lam}_{/\approx}$. Since Isabelle/HOL

$$\frac{}{\texttt{Var}(a) \approx \texttt{Var}(a)}\ {\approx_{\texttt{Var}}} \qquad \frac{t_1 \approx s_1 \quad t_2 \approx s_2}{\texttt{App}(t_1, t_2) \approx \texttt{App}(s_1, s_2)}\ {\approx_{\texttt{App}}}$$

$$\frac{t \approx s}{\texttt{Lam}(a, t) \approx \texttt{Lam}(a, s)}\ {\approx_{\texttt{Lam1}}} \qquad \frac{a \neq b \quad t \approx (a\, b) \bullet s \quad a \notin \texttt{fv}(s)}{\texttt{Lam}(a, t) \approx \texttt{Lam}(b, s)}\ {\approx_{\texttt{Lam2}}}$$

$$\frac{a \neq b}{a \notin \texttt{fv}(\texttt{Var}(b))}\ \texttt{fv}_{\texttt{Var}} \qquad \frac{a \notin \texttt{fv}(t_1) \quad a \notin \texttt{fv}(t_2)}{a \notin \texttt{fv}(\texttt{App}(t_1, t_2))}\ \texttt{fv}_{\texttt{App}}$$

$$\frac{}{a \notin \texttt{fv}(\texttt{Lam}(a, t))}\ \texttt{fv}_{\texttt{Lam1}} \qquad \frac{a \neq b \quad a \notin \texttt{fv}(t)}{a \notin \texttt{fv}(\texttt{Lam}(b, t))}\ \texttt{fv}_{\texttt{Lam2}}$$

Fig. 2 Inductive definitions for $(-) \approx (-)$ and $(-) \notin \texttt{fv}(-)$

supports subset types, we can later turn $\texttt{lam}_\alpha$ into a new type. In order to obtain the bijection, `phi` needs to be defined so that it contains elements corresponding, roughly speaking, to alpha-equated variables, applications and lambda-abstractions—that is to $[\texttt{Var}(a)]_\alpha$, $[\texttt{App}(t_1, t_2)]_\alpha$ and $[\texttt{Lam}(a, t)]_\alpha$. Whereas this is straightforward for variables and applications, the lambda-abstractions are non-trivial: for them we shall use some *specific* "partial" functions from `name` to `phi` (by "partial" we mean here functions that return *None* for undefined values and *Some*(x) for defined ones[4]). We therefore define `phi` as the Isabelle/HOL datatype:

```
datatype phi = Am "name"
             | Pr "phi × phi"
             | Se "name ⇒ (phi option)"          (7)
```

where `Am` will be used to encode atoms; `Pr` to encode applications, which are built up by a pair of terms; and `Se` to encode an alpha-equivalence class (that is a set) of terms. The permutation operation for `phi` is defined over the structure as follows:

$$\begin{aligned}\pi \bullet \texttt{Am}(a) &\stackrel{\text{def}}{=} \texttt{Am}(\pi \bullet a)\\ \pi \bullet \texttt{Pr}(t_1, t_2) &\stackrel{\text{def}}{=} \texttt{Pr}(\pi \bullet t_1, \pi \bullet t_2)\\ \pi \bullet \texttt{Se}(fn) &\stackrel{\text{def}}{=} \texttt{Se}(\pi \bullet fn)\end{aligned} \qquad (8)$$

using in the last clause the permutations operation for functions given in (2). It is not hard to show that `phi` is a permutation type (routine induction over the structure of `phi`-terms).

[4]In Urban and Tasson [36] a special error-element was used to stand for undefinedness. However, the approach based on the option-type turned out to be more convenient for building a nominal datatype package in Isabelle/HOL.

We mentioned earlier that we are not going to use all functions from `name` to `phi option` for representing alpha-equated lambda-abstractions, but some specific functions.[5] These functions are of the form:

$$[a].t \overset{\text{def}}{=} \lambda b.\,\texttt{if}\ a = b\ \texttt{then}\ \mathit{Some}(t) \\ \texttt{else if}\ b \,\#\, t\ \texttt{then}\ \mathit{Some}((a\,b)\bullet t)\ \texttt{else}\ \mathit{None} \qquad (9)$$

and we will refer to them as *abstraction functions*; their parameters are an atom and a `phi`-term.

We claim that these functions represent alpha-equivalence classes. To see this, consider $[\texttt{Lam}(a, \texttt{App}(\texttt{Var}(a), \texttt{Var}(b)))]_\alpha$ and the corresponding `phi`-term $\texttt{Se}([a].\texttt{Pr}(\texttt{Am}(a), \texttt{Am}(b)))$. The graph of the abstraction function is as follows: the atom a is mapped to the term $\mathit{Some}(\texttt{Pr}(\texttt{Am}(a), \texttt{Am}(b)))$ since the first `if`-condition is true. For b, the first `if`-condition obviously fails, but also the second one fails, because $\mathit{supp}(\texttt{Pr}(\texttt{Am}(a), \texttt{Am}(b))) = \{a, b\}$; therefore b is mapped to *None*. For all other atoms c, we have $a \neq c$ and $c \,\#\, \texttt{Pr}(\texttt{Am}(a), \texttt{Am}(b))$; consequently these c's are mapped by the abstraction function to $\mathit{Some}((a\,c)\bullet\texttt{Pr}(\texttt{Am}(a), \texttt{Am}(b)))$, which is $\mathit{Some}(\texttt{Pr}(\texttt{Am}(c), \texttt{Am}(b)))$. Clearly, the abstraction function returns *None* whenever the corresponding lambda-term is *not* in the alpha-equivalence class—in this example the lambda-term $\texttt{Lam}(b, \texttt{App}(\texttt{Var}(b), \texttt{Var}(b))) \notin [\texttt{Lam}(a, \texttt{App}(\texttt{Var}(a), \texttt{Var}(b)))]_\alpha$; in all other cases, however, it returns an appropriately "renamed" version of $\texttt{Pr}(\texttt{Am}(a), \texttt{Am}(b))$.

To show formally that abstraction functions represent alpha-equivalence classes, we first establish how the permutation operation behaves on those functions and then establish the conditions under which two such functions are equal:

Lemma 6 *All abstraction functions satisfy:*

(i) $\pi\bullet([a].t) = [\pi\bullet a].(\pi\bullet t)$, *and*
(ii) $[a].t_1 = [b].t_2$ *if and only if either:*

$$a = b \wedge t_1 = t_2 \qquad \textit{or} \qquad a \neq b \wedge t_1 = (a\,b)\bullet t_2 \wedge a \,\#\, t_2\ .$$

Proof The first property follows from the following calculation:

$$\begin{aligned} & \pi\bullet[a].t \\ \overset{\text{def}}{=}\ & \pi\bullet\lambda b.\,\texttt{if}\ a = b\ \texttt{then}\ \mathit{Some}(t) \\ & \qquad \texttt{else if}\ b \,\#\, t\ \texttt{then}\ \mathit{Some}((a\,b)\bullet t)\ \texttt{else}\ \mathit{None} \\ \overset{\text{def}}{=}\ & \lambda b.\,\pi\bullet\texttt{if}\ a = \pi^{-1}\bullet b\ \texttt{then}\ \mathit{Some}(t) \\ & \quad \texttt{else if}\ \pi^{-1}\bullet b \,\#\, t\ \texttt{then}\ \mathit{Some}((a\ \pi^{-1}\bullet b)\bullet t)\ \texttt{else}\ \mathit{None} \\ =\ & \lambda b.\,\texttt{if}\ \pi\bullet(a = \pi^{-1}\bullet b)\ \texttt{then}\ \mathit{Some}(\pi\bullet t) & (*^1) \\ & \quad \texttt{else if}\ \pi\bullet(\pi^{-1}\bullet b \,\#\, t)\ \texttt{then}\ \mathit{Some}(\pi\bullet(a\ \pi^{-1}\bullet b)\bullet t)\ \texttt{else}\ \mathit{None} \end{aligned}$$

[5]This is in contrast to "weak" and "full" HOAS [8, 25] which use the full function space for representing lambda-abstractions.

$$\begin{aligned}
&= \lambda b.\,\texttt{if}\ \pi\bullet(a = \pi^{-1}\bullet b)\ \texttt{then}\ \mathit{Some}(\pi\bullet t) && (*^2)\\
&\qquad \texttt{else if}\ \pi\bullet(\pi^{-1}\bullet b\ \#\ t)\ \texttt{then}\ \mathit{Some}((\pi\bullet a\ b)\bullet\pi\bullet t)\ \texttt{else}\ \mathit{None}\\
&= \lambda b.\,\texttt{if}\ \pi\bullet a = b\ \texttt{then}\ \mathit{Some}(\pi\bullet t) && (*^3)\\
&\qquad \texttt{else if}\ b\ \#\ \pi\bullet t\ \texttt{then}\ \mathit{Some}((\pi\bullet a\ b)\bullet\pi\bullet t)\ \texttt{else}\ \mathit{None}\\
&\overset{\text{def}}{=} [\pi\bullet a].(\pi\bullet t)
\end{aligned}$$

where we use in $(*^1)$ the fact that

$$\pi\bullet\texttt{if}\ldots\texttt{then}\ldots\texttt{else}\ldots = \texttt{if}\,\pi\bullet\ldots\texttt{then}\,\pi\bullet\ldots\texttt{else}\,\pi\bullet\ldots \tag{10}$$

and in $(*^2)$ that $\pi@(a\ \pi^{-1}\bullet b) \sim (\pi\bullet a\ b)@\pi$; for $(*^3)$ the facts that $\pi\bullet(a = \pi^{-1}\bullet b)$ iff $\pi\bullet a = b$ and $\pi\bullet(\pi^{-1}\bullet b\ \#\ t)$ iff $b\ \#\ \pi\bullet t$, which can be easily derived from Lemmas 1(ii) and 3(ii) and the permutation operation on `bool`.

For the second property the case $a = b$ is by a simple calculation using extensionality of functions. In case $a \neq b$ we show first the $\Rightarrow$-direction: the following formula holds then by extensionality of functions:

$$\begin{aligned}
\forall c.\ \ &\texttt{if}\ a = c\ \texttt{then}\ \mathit{Some}(t_1)\\
&\texttt{else if}\ c\ \#\ t_1\ \texttt{then}\ \mathit{Some}((a\,c)\bullet t_1)\ \texttt{else}\ \mathit{None}\\
=\ &\texttt{if}\ b = c\ \texttt{then}\ \mathit{Some}(t_2)\\
&\texttt{else if}\ c\ \#\ t_2\ \texttt{then}\ \mathit{Some}((b\ c)\bullet t_2)\ \texttt{else}\ \mathit{None}
\end{aligned}$$

Instantiating this formula with a yields the equation

$$\mathit{Some}(t_1) = \texttt{if}\ a\ \#\ t_2\ \texttt{then}\ \mathit{Some}((b\ a)\bullet t_2)\ \texttt{else}\ \mathit{None}\ .$$

Next, one distinguishes the cases where $a\ \#\ t_2$ and $\neg\, a\ \#\ t_2$, respectively. In the first case, $\mathit{Some}(t_1) = \mathit{Some}((b\ a)\bullet t_2)$, which by Definition 2(iii) implies $t_1 = (a\,b)\bullet t_2$ since $(a\,b)\sim(b\ a)$; and obviously $a\ \#\ t_2$ by assumption. In the second case $\mathit{Some}(t_1) = \mathit{None}$ which gives a contradiction. The $\Leftarrow$-direction for the case $a \neq b$ is similarly by extensionality and a case-analysis. □

Note that, in *general*, one cannot decide whether two functions from `name` to `phi option` are equal; however for the abstraction functions Lemma 6(ii) provides the means to decide whether $[a].t_1 = [b].t_2$ holds: one just has to consider whether $a = b$, which is just like deciding the alpha-equivalence of two lambda-terms using the relation $(-)\approx(-)$ given in Fig. 2. Now it is also clear why abstraction functions represent alpha-equivalence classes: the condition we derived for the equality between abstraction functions paraphrase the rules $\approx_{\texttt{Lam1}}$ and $\approx_{\texttt{Lam2}}$ defining alpha-equivalence for `lam`.

The properties in Lemma 6 also help us to calculate the support for abstraction functions, provided they "abstract" over a finitely supported `phi`-term.

Lemma 7 *Given $a \neq b$ and t being finitely supported, then*

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

Proof By a simple calculations we have that $\mathit{supp}([b].t) \subseteq \mathit{supp}(b, t)$ because for all c and d we have $\{d \mid (c\,d)\bullet[b].t \neq [b].t\} \subseteq \{d \mid (c\,d)\bullet(b, t) \neq (b, t)\}$. Since b and t are finitely supported, $[b].t$ must be finitely supported. Hence $(a, b, t, [b].t)$ is finitely supported and by Proposition 1 there exists an atom c with $(*)$ $c \,\#\, (a, b, t, [b].t)$.

Now we show the direction *(i ⇒)*: using the assumption $a \,\#\, [b].t$ and the fact that $c \,\#\, [b].t$ (from $*$), Lemma 4 and 6(i) give $[b].t = (c\,a)\bullet[b].t = [(c\,a)\bullet b].((c\,a)\bullet t)$. The right-hand side is $[b].((c\,a)\bullet t)$ because $c \neq b$ (from $*$) and $a \neq b$ by assumption. Hence by Lemma 6(ii) we can infer that $t = (c\,a)\bullet t$. Now $c \,\#\, t$ (from $*$) implies that $c \,\#\, (c\,a)\bullet t$; and moving the permutation to the other side by Lemma 3(ii) gives $a \,\#\, t$. The direction *(i ⇐)* is as follows: from $(*)$, we have that $c \,\#\, [b].t$ and therefore by Lemma 3(iii) also $(a\,c)\bullet c \,\#\, (a\,c)\bullet([b].t)$, which implies by Lemma 6(i) that $a \,\#\, [b].((a\,c)\bullet t)$. From $(*)$ we also have $c \,\#\, t$ and from the assumption $a \,\#\, t$; then Lemma 4 implies that $t = (a\,c)\bullet t$, and we can conclude with $a \,\#\, [b].t$.

The second property follows from the first: we have $c \,\#\, t$ and $c \neq a$ (both from $*$), and can use *(i)* to infer $c \,\#\, [a].t$. Further, from Lemma 3(iii) it holds that $(c\,a)\bullet c \,\#\, (c\,a)\bullet[a].t$. This is $a \,\#\, [c].(c\,a)\bullet t$ by Lemma 6(i). Since $c \neq a$ and $c \,\#\, t$, Lemma 6(ii) implies that $[c].(c\,a)\bullet t = [a].t$. Therefore, $a \,\#\, [a].t$. □

Note that taking both facts of Lemma 7 together implies the following equation for the support of abstraction functions

$$supp([a].t) = supp(t) - \{a\} \tag{11}$$

provided t is finitely supported.

Now everything is in place for defining the subset $\texttt{lam}_\alpha$. It is defined inductively by the three rules:

$$\frac{}{\texttt{Am}(a) \in \texttt{lam}_\alpha} \qquad \frac{t_1 \in \texttt{lam}_\alpha \quad t_2 \in \texttt{lam}_\alpha}{\texttt{Pr}(t_1, t_2) \in \texttt{lam}_\alpha} \qquad \frac{t \in \texttt{lam}_\alpha}{\texttt{Se}([a].t) \in \texttt{lam}_\alpha} \tag{12}$$

using in the third rule the abstraction functions given in (9). We note:

Lemma 8 *For the set* $\texttt{lam}_\alpha$ *we have that:*

(i) *all its elements are finitely supported, and*
(ii) *it is closed under permutations, that is* $t \in \texttt{lam}_\alpha$ *implies* $\pi\bullet t \in \texttt{lam}_\alpha$.

Proof Both properties follow by routine inductions over the definition of $\texttt{lam}_\alpha$. For the first induction we use the equations

$$\begin{aligned} supp(\texttt{Am}(a)) &= \{a\} \\ supp(\texttt{Pr}(t_1, t_2)) &= supp(t_1) \cup supp(t_2) \\ supp(\texttt{Se}([a].t)) &= supp(t) - \{a\} \end{aligned} \tag{13}$$

where the last follows from (11)—t is finitely supported by induction hypothesis; for the second we use Lemma 6(i). □

Next, one of the main points of this paper: there is a bijection between $\texttt{lam}_{/\approx}$ and $\texttt{lam}_\alpha$. This is shown using the following mapping from `lam` to $\texttt{lam}_\alpha$:

$$\begin{aligned} q(\texttt{Var}(a)) &\overset{\text{def}}{=} \texttt{Am}(a) \\ q(\texttt{App}(t_1, t_2)) &\overset{\text{def}}{=} \texttt{Pr}(q(t_1), q(t_2)) \\ q(\texttt{Lam}(a, t)) &\overset{\text{def}}{=} \texttt{Se}([a].q(t)) \end{aligned}$$

and the lemma:

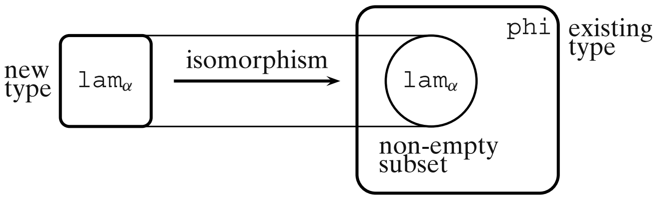
Lemma 9 $t_1 \approx t_2$ *if and only if* $q(t_1) = q(t_2)$.

Proof By routine induction over definition of $\texttt{lam}_\alpha$. □

Theorem 1 *There is a bijection between* $\texttt{lam}_{/\approx}$ *and* $\texttt{lam}_\alpha$.

Proof The mapping q needs to be lifted to alpha-equivalence classes (see Paulson [24]). 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 Lemma 9 this must yield a singleton set. The result of $q'([t]_\alpha)$ is then the singleton. Subjectivity of q' is shown by a routine induction over the definition of $\texttt{lam}_\alpha$. Injectivity of q' follows from Lemma 9 since $[t_1]_\alpha = [t_2]_\alpha$ for all $t_1 \approx t_2$. □

We defined $\texttt{lam}_\alpha$ as an inductive subset of `phi` and showed that there is a bijection with $\texttt{lam}_{/\approx}$. We can now apply standard HOL-techniques and turn the *set* $\texttt{lam}_\alpha$ into a *type* $\texttt{lam}_\alpha$ of HOL (see for example the Isabelle tutorial [21, Section 8.5.2] or Melham [19, 20] for more details). The construction we can perform in HOL is illustrated by the following picture:



We are allowed to introduce the type $\texttt{lam}_\alpha$ by means of identifying a non-empty subset in the existing type `phi` (this type was introduced by the datatype declaration in (7)) and an isomorphism, which we write here as $\ulcorner - \urcorner$. The properties of the type $\texttt{lam}_\alpha$ are then given by the isomorphism and how the subset $\texttt{lam}_\alpha$ is defined. For example we can characterise term-constructors of the type $\texttt{lam}_\alpha$ as follows:

$$\begin{aligned} \ulcorner \texttt{Var}_\alpha(a) \urcorner &\mapsto \texttt{Am}(a) \\ \ulcorner \texttt{App}_\alpha(t_1, t_2) \urcorner &\mapsto \texttt{Pr}(\ulcorner t_1 \urcorner, \ulcorner t_2 \urcorner) \\ \ulcorner \texttt{Lam}_\alpha(a, t) \urcorner &\mapsto \texttt{Se}([a].\ulcorner t \urcorner) \end{aligned} \tag{14}$$

with the following “injection” principles

$$\begin{aligned}\mathtt{Var}_\alpha(a) = \mathtt{Var}_\alpha(b) &\text{ iff } a = b\\ \mathtt{App}_\alpha(t_1, t_2) = \mathtt{App}_\alpha(s_1, s_2) &\text{ iff } t_1 = s_1 \wedge t_2 = s_2\\ \mathtt{Lam}_\alpha(a, t_1) = \mathtt{Lam}_\alpha(b, t_2) &\text{ iff } [a].t_1 = [b].t_2\end{aligned} \tag{15}$$

and the support behaving as follows:

$$\begin{aligned}\mathit{supp}(\mathtt{Var}_\alpha(a)) &= \{a\}\\ \mathit{supp}(\mathtt{App}_\alpha(t_1, t_2)) &= \mathit{supp}(t_1) \cup \mathit{supp}(t_2)\\ \mathit{supp}(\mathtt{Lam}_\alpha(a, t)) &= \mathit{supp}(t) - \{a\}\end{aligned} \tag{16}$$

Since by Lemma 8(ii) the permutation operation is closed on the set $\mathtt{lam}_\alpha$, we can also lift the permutation operation defined over `phi` to the new type so that the following properties hold:

$$\begin{aligned}\pi \bullet \mathtt{Var}_\alpha(a) &= \mathtt{Var}_\alpha(\pi \bullet a)\\ \pi \bullet \mathtt{App}_\alpha(t_1, t_2) &= \mathtt{App}_\alpha(\pi \bullet t_1, \pi \bullet t_2)\\ \pi \bullet \mathtt{Lam}_\alpha(a, t) &= \mathtt{Lam}_\alpha(\pi \bullet a, \pi \bullet t)\end{aligned} \tag{17}$$

We can further show that:

Lemma 10 *The type* $\mathtt{lam}_\alpha$ *is a (i) permutation type and (ii) all its elements are finitely supported.*

Proof By routine induction the over definition of $\mathtt{lam}_\alpha$. For *(i)* we lift the property of `phi` being a permutation type to $\mathtt{lam}_\alpha$ using Lemma 8(ii); for *(ii)* we use (16). □

The crux of constructing the new type $\mathtt{lam}_\alpha$ is that we now have an Isabelle/HOL-type where lambdas are equal provided

$$\begin{aligned}&\mathtt{Lam}_\alpha(a, t_1) = \mathtt{Lam}_\alpha(b, t_2) \quad \text{if and only if either}\\ &\qquad a = b \wedge t_1 = t_2 \qquad \text{or} \qquad a \neq b \wedge t_1 = (a\,b)\bullet t_2 \wedge a \,\#\, t_2\,.\end{aligned} \tag{18}$$

and freshness of a lambda is given by:

$$\begin{aligned}&a \,\#\, \mathtt{Lam}_\alpha(b, t) \quad \text{if and only if either}\\ &\qquad a = b \qquad \text{or} \qquad a \neq b \wedge a \,\#\, t\,.\end{aligned} \tag{19}$$

In effect we have achieved what we set out at the beginning of this section: we have a formal implementation of Barendregt’s convention about identifying alpha-equivalent lambda-terms.

4 Structural Induction Principles

The inductive definition of the set $\texttt{lam}_\alpha$ given in (12) comes with an induction principle. From this induction principle we can derive the following structural induction principle for the type $\texttt{lam}_\alpha$:

$$\frac{\begin{array}{l}\forall a.\;\; P\,(\texttt{Var}_\alpha(a))\\ \forall t_1\,t_2.\;\; P\,t_1 \;\wedge\; P\,t_2 \;\Rightarrow\; P\,(\texttt{App}_\alpha(t_1,t_2))\\ \forall a\,t_1.\;\; P\,t_1 \;\Rightarrow\; P\,(\texttt{Lam}_\alpha(a,t_1))\end{array}}{P\,t} \tag{20}$$

However, this structural induction principle is not very convenient in practice. Consider again Fig. 1 showing a typical informal proof involving lambda-terms. This informal proof establishes the substitution lemma by considering in the lambda-case only binders z that have suitable properties (namely being fresh for x, y, N and L). If one would use for this proof the induction principle given above, then one would need to show the lambda-case for *all* z, not just the ones being suitably fresh. This would mean one has 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. This induction principle is as follows:

$$\frac{\begin{array}{l}\forall c\,a.\;\; P\,(\texttt{Var}_\alpha(a))\;c\\ \forall c\,t_1\,t_2.\;\; (\forall d.\; P\,t_1\;d) \;\wedge\; (\forall d.\; P\,t_2\;d) \;\Rightarrow\; P\,(\texttt{App}_\alpha(t_1,t_2))\;c\\ \forall c\,a\,t_1.\;\; a\,\#\,c \;\wedge\; (\forall d.\; P\,t_1\;d) \;\Rightarrow\; P\,(\texttt{Lam}_\alpha(a,t_1))\;c\end{array}}{P\,t\,c} \tag{21}$$

where the variable t in the conclusion stands for a $\texttt{lam}_\alpha$-term over which the induction is done and the variable c stands for the *context* of the induction. By the context of an induction we mean all free variables of the lemma to be shown by induction, except the variable over which the induction is performed. We also assume that the context is of finitely supported type. In case of the substitution lemma from Fig. 1, for example, we have

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

with M being the variable over which the induction is done. So in this case, the context c would be instantiated with the other free variables in this lemma, namely the tuple (x, y, N, L)—which is of finitely supported type. When it comes to prove the lambda-case, that is

$$P\,(\texttt{Lam}_\alpha(z, M_1))\;(x, y, N, L)$$

one can assume in (21) that the binder z is fresh for (x, y, N, L)—which is equivalent to z not being equal to x and y, and not free in N and L. As we shall see later, with this induction principle one can formalise Barendregt's slick informal proof without difficulties.

In the following we shall establish a slightly more general version of the induction principle given in (21). In the generalised version we require that the induction context is finitely supported, but not necessarily has finitely supported type.

Theorem 2 (Strong Induction Principle) *A property $P\ t\ c$ holds for all t terms of type* $\texttt{lam}_\alpha$*, provided for a given f*

(i) $\forall c.\ \mathit{finite}(\mathit{supp}(f\,c))$,
(ii) $\forall c\,a.\ \ P\,(\texttt{Var}_\alpha(a))\ c$,
(iii) $\forall c\,t_1\,t_2.\ \ (\forall d.\ P\ t_1\ d)\ \wedge\ (\forall d.\ P\ t_2\ d)\ \Rightarrow P\,(\texttt{App}_\alpha(t_1,t_2))\ c$, *and*
(iv) $\forall c\,a\,t_1.\ \ a\ \#\ f\,c\ \wedge\ (\forall d.\ P\ t_1\ d)\ \Rightarrow\ P\,(\texttt{Lam}_\alpha(a,t_1))\ c$

hold.

Proof By induction over t using (20). We strengthen the induction hypothesis by aiming to prove $\forall\pi\ c.\ P\,(\pi\bullet t)\,c$. The cases for $\texttt{Var}_\alpha$ and $\texttt{App}_\alpha$ are routine. The interesting case is $\texttt{Lam}_\alpha$: we need to show that $P\,(\pi\bullet\texttt{Lam}_\alpha(a,t_1))\,c$, where $\pi\bullet\texttt{Lam}_\alpha(a,t_1) = \texttt{Lam}_\alpha(\pi\bullet a,\pi\bullet t_1)$ by (17). Since by *(i)* $f c$ is finitely supported, and by Lemmas 4 and 10 also $\pi\bullet a$ and $\pi\bullet t_1$, we can use Proposition 1 to obtain a b with $b\ \#\ (f\,c,\pi\bullet a,\pi\bullet t_1)$. From this we can infer that $b\neq\pi\bullet a$ and $b\ \#\ \pi\bullet t_1$, which implies by (18) that $(*)$ $\texttt{Lam}_\alpha(b,(b\ \ \pi\bullet a)\bullet(\pi\bullet t_1)) = \texttt{Lam}_\alpha(\pi\bullet a,\pi\bullet t_1)$. From the induction hypothesis, which is $\forall\pi\ c.\ P\,(\pi\bullet t_1)\,c$, we obtain the fact $\forall c.\ P\,(((b\ \ \pi\bullet a)@\pi)\bullet t_1)\,c$. Then we can use the fact $b\ \#\ f\,c$ and *(iv)*, and infer that $P\,(\texttt{Lam}_\alpha(b,((b\ \ \pi\bullet a)@\pi)\bullet t_1))\,c$ holds. Moreover this is by Definition 2(ii) equal to the fact $P\,(\texttt{Lam}_\alpha(b,(b\ \ \pi\bullet a)\bullet(\pi\bullet t_1)))\,c$. By $(*)$ we can conclude with $P\,(\texttt{Lam}_\alpha(\pi\bullet a,\pi\bullet t_1))\,c$. □

If we set in Thoerem 2 f to the identity-function and require that c has finitely supported type, we can discharge condition *(i)* in and obtain the structural induction principle stated in (21). The advantage of (21) is that Isabelle's axiomatic type classes can be used to ensure that the induction context is a finitely supported type, while the induction principle proved in Theorem 2 requires manual reasoning to ensure the finite support property. However, we will need the more general induction principle in the next section where we derive a recursion combinator for $\texttt{lam}_\alpha$.

5 A Recursion Combinator

Before we can formalise Barendregt's proof of the substitution lemma, we need to be able to define the function of capture-avoiding substitution. This can be done by first considering an appropriately defined relation and then showing that this relation behaves like a function. This has been done in Urban and Tasson [36]. However, this way is rather inelegant. More elegant is a definition by structural recursion.

It turns out that defining functions by recursion over the structure of alpha-equated lambda-terms is rather subtle. Let us assume we want to define capture-avoiding substitution by the following three clauses

$$\begin{aligned}
\texttt{Var}_\alpha(x)[y:=t'] &= (\text{if } x=y \text{ then } t' \text{ else } \texttt{Var}_\alpha(x))\\
\texttt{App}_\alpha(t_1,t_2)[y:=t'] &= \texttt{App}_\alpha(t_1[y:=t'],t_2[y:=t'])\\
\texttt{Lam}_\alpha(x,t)[y:=t'] &= \texttt{Lam}_\alpha(x,t[y:=t']) \qquad \text{provided } x\ \#\ (y,t')
\end{aligned}$$

where the side-condition in the lambda-case amounts to the usual condition about $x\neq y$ and x not being a free atom in t'. Then defining it over $\texttt{lam}_\alpha$ results in a total function, while defining it over “raw” lambda-terms of type $\texttt{lam}$ results in a partial function. Furthermore, attempting to define the functions that return the set of bound names and the immediate subterms by the clauses

$$\begin{aligned} bn(\texttt{Var}_\alpha(x)) &= \varnothing & ist(\texttt{Var}_\alpha(x)) &= \varnothing \\ bn(\texttt{App}_\alpha(t_1, t_2)) &= bn(t_1) \cup \mathrm{bn}(t_2) & ist(\texttt{App}_\alpha(t_1, t_2)) &= \{t_1, t_2\} \\ bn(\texttt{Lam}_\alpha(x, t)) &= bn(t) \cup \{x\} & ist(\texttt{Lam}_\alpha(x, t)) &= \{t\} \end{aligned} \tag{22}$$

results in an inconsistency when defined over $\texttt{lam}_\alpha$, while it can be defined without problems over `lam`. The inconsistency with *bn* and *ist* arises by the principle of HOL stating that a function has to return the "same ouput" for the "same input". Since by (18) we have

$$\texttt{Lam}_\alpha(x, \texttt{Var}_\alpha(x)) = \texttt{Lam}_\alpha(y, \texttt{Var}_\alpha(y))$$

for all x and y, we can assume that this equation holds for $x \neq y$. Then $bn(\texttt{Lam}_\alpha(x, \texttt{Var}_\alpha(x)))$ must be equal to $bn(\texttt{Lam}_\alpha(y, \texttt{Var}_\alpha(y)))$, which implies by the clauses in (22) that x must be equal to y giving a contradiction with the assumption $x \neq y$—similar with the function *ist*.

One way around the problem with the inconsistencies is to derive a recursion combinator for $\texttt{lam}_\alpha$ that includes certain preconditions for binders ensuring no inconsistency can be derived. For this we will adapt work by Pitts [27] who introduced such preconditions. We will also adapt his proof establishing the existence of a structural recursion combinator for $\texttt{lam}_\alpha$. The main difference of our proof is that we give here a direct proof for the existence, because in our implementation we do not use anywhere the type `lam` (Pitts uses `lam` to derive a structural induction principle). Another difference is that we derive the recursion combinator without deriving an iteration combinator first.[6]

While in "every-day" formalisation, Lemma 4 is sufficient in nearly all situations to find out when an object has finite support, the reasoning for the recursion combinator includes in several places proof obligations about ensuring that functions have finite support. And for functions one cannot find out whether they have finite support by just looking at their type. In order to automate such proof obligations we use the auxiliary notion of *supports* [11].

Definition 5 A set S of atoms *supports* an x of permutation type, written S *supports* x, provided:

$$\forall a\, b\,.\; a \notin S \wedge b \notin S \;\Rightarrow\; (a\, b) \bullet x = x\,.$$

This notion allows us to approximate the support of an x from "above", because we can show that:

Lemma 11 *If a set S is finite and S supports x, then $supp(x) \subseteq S$.*

[6]The difference between a recursion and an iteration combinator is that in the former we can use directly the arguments of the term constructor, while in the latter this can only be achieved via an encoding of the recursion.

Proof By contradiction we assume $supp(x) \not\subseteq S$, then there exists an atom $a \in supp(x)$ and $a \notin S$. From $S\, supports\, x$ follows that for all $b \notin S$ we have $(a\,b)\bullet x = x$. Hence the set $\{b \mid (a\,b)\bullet x \neq x\}$ is a subset of S, and since S is finite by assumption, also $\{b \mid (a\,b)\bullet x \neq x\}$ must be finite. But this implies that $a \notin supp(x)$ which gives the contradiction. □

Lemma 11 gives us some means to decide relatively easily whether a function has finite support: one only needs to find a finite set of atoms and then verify whether this set supports the function.

If the function is given as a lambda-term on the HOL-level, then for finding a finite set we use the heuristic of considering the support of the free variables of this functions. This is a heuristic, because it cannot be established as a lemma inside Isabelle/HOL—it is a property about HOL-functions. Nevertheless the heuristic is extremely helpful for deciding whether a function has finite support. Consider the following two examples:

Example 1 Given a function $fn \overset{\text{def}}{=} f_1\, c$ where f_1 is a function of type `name` $\Rightarrow \alpha$. We also assume that f_1 has finite support. The question is whether fn has finite support? The free variables of fn are f_1 and c. According to our heuristic we have to verify whether $supp(f_1, c)\; supports\, fn$, which amounts to showing that

$$\forall a\, b\, .\; a \notin supp(f_1, c) \;\wedge\; b \notin supp(f_1, c) \;\Rightarrow\; (a\,b)\bullet fn = fn$$

To do so we can assume by the definition of freshness (Definition 3) that $a \# (f_1, c)$ and $b \# (f_1, c)$ and show that $(a\,b)\bullet fn = fn$. This equation follows from the calculation that pushes the swapping $(a\,b)$ inside fn:

$$(a\,b)\bullet fn \overset{\text{def}}{=} (a\,b)\bullet(f_1\, c) \overset{\text{by (3)}}{=} ((a\,b)\bullet f_1)\,((a\,b)\bullet c) \overset{(*)}{=} f_1\, c \overset{\text{def}}{=} fn$$

where $(*)$ follows because we know that $a \# f_1$ and $b \# f_1$, and therefore by Lemma 4 that $(a\,b)\bullet f_1 = f_1$ (similarly for c).

We can conclude that $supp(fn)$ is a subset of $supp(f_1, c)$, because the latter is finite (since f_1 has finite support by assumption and c is finitely supported because the type `name` is a finitely supported type). So by Lemma 11, fn must have finite support. □

Example 2 Let $fn' \overset{\text{def}}{=} \lambda x.\, \text{if } x = y \text{ then } t' \text{ else } (\texttt{Var}_\alpha(x))$—where x and y are of type `name` and t' a $\texttt{lam}_\alpha$-term. The free variables of this HOL-function are y and t'; so by our heuristic we need to verify whether $supp(y, t')\; supports\, fn'$. This holds by the following calculation:

$$\begin{array}{lll}
 & (a\,b)\bullet(\lambda x.\ \text{if } x = y \text{ then } t' \text{ else } \texttt{Var}_\alpha(x)) & \\
\overset{\text{def}}{=} & \lambda x.\ (a\,b)\bullet(\text{if } (a\,b)^{-1}\bullet x = y \text{ then } t' \text{ else } \texttt{Var}_\alpha((a\,b)^{-1}\bullet x)) & \\
= & \lambda x.\, \text{if } x = (a\,b)\bullet y \text{ then } (a\,b)\bullet t' \text{ then } \texttt{Var}_\alpha(x) & \text{by (10)} \\
= & \lambda x.\ \text{if } x = y \text{ then } t' \text{ else } \texttt{Var}_\alpha(x) & (*)
\end{array}$$

where $(*)$ follows by Lemma 4 and the assumption that $a \# (y, t')$ and $b \# (y, t')$. Since y and t' are finitely supported types, fn' must then have finite support. □

As the examples indicate, by using the heuristic, one can infer from a decision problem involving permutations whether or not a function has finite support. The important point here is that the decision procedure involving permutations can be relatively easily automated with a special purpose tactic analysing permutations. This seems much more convenient than analysing the support of a function directly.

A definition by structural recursion involves in case of the lambda-terms three functions (one for each term-constructor) that specify the behaviour of the function to be defined—let us call these functions f_1, f_2, f_3 for the variable-, application- and lambda-case, respectively, and let us assume they have the types:

$$\begin{aligned} &f_1 : \texttt{name} \Rightarrow \alpha \\ &f_2 : \texttt{lam}_\alpha \Rightarrow \texttt{lam}_\alpha \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \alpha \\ &f_3 : \texttt{name} \Rightarrow \texttt{lam}_\alpha \Rightarrow \alpha \Rightarrow \alpha \end{aligned}$$

with α being a permutation type. Then the first condition Pitts introduced in [27] states that f_3—the function for the lambda case—needs to satisfy the *freshness condition for binders*, or short *FCB*. We formulate this condition as:[7]

Definition 6 (Freshness Condition for Binders)
A function f with type $\texttt{name} \Rightarrow \texttt{lam}_\alpha \Rightarrow \alpha \Rightarrow \alpha$ satisfies the *FCB* provided:

$$\forall a\,t\,r.\ a \,\#\, f \ \wedge\ \mathit{finite}(\mathit{supp}(r)) \ \Rightarrow\ a \,\#\, f\,a\,t\,r\ .$$

As we shall see later on, this condition ensures that the result of f_3 is independent of which particular fresh name one chooses for the binder a. The second condition states that the functions f_1, f_2 and f_3 all must have finite support. This condition ensures that we can use Proposition 1 when choosing a fresh name for the fs.

With these two conditions we can derive a recursion combinator, we call it $\mathit{rfun}_{f_1 f_2 f_3}$, with the following properties:

Theorem 3 (Recursion Combinator) *If f_1, f_2 and f_3 have finite support and f_3 satisfies the FCB, then there exists a recursion combinator $\mathit{rfun}_{f_1 f_2 f_3}$ with the properties:*

$$\begin{aligned} \mathit{rfun}_{f_1 f_2 f_3}\,(\texttt{Var}_\alpha(a)) &= f_1\,a \\ \mathit{rfun}_{f_1 f_2 f_3}\,(\texttt{App}_\alpha(t_1,t_2)) &= f_2\,t_1\,t_2\,(\mathit{rfun}_{f_1 f_2 f_3}\,t_1)\,(\mathit{rfun}_{f_1 f_2 f_3}\,t_2) \\ \mathit{rfun}_{f_1 f_2 f_3}\,(\texttt{Lam}_\alpha(a,t)) &= f_3\,a\,t\,(\mathit{rfun}_{f_1 f_2 f_3}\,t) \\ &\qquad\qquad \textit{provided } a \,\#\, (f_1,\,f_2,\,f_3) \end{aligned}$$

To give a proof of this theorem we start with the following inductive relation, called $\mathit{rec}_{f_1 f_2 f_3}$ and which has type $(\texttt{lam}_\alpha \times \alpha)\ \texttt{set}$ where, like above, α is assumed to be a permutation type:

$$\frac{}{(\texttt{Var}_\alpha(a),\ f_1\,a)\ \in\ \mathit{rec}_{f_1 f_2 f_3}} \qquad \frac{(t_1,r_1)\ \in\ \mathit{rec}_{f_1 f_2 f_3} \quad (t_2,r_2)\ \in\ \mathit{rec}_{f_1 f_2 f_3}}{(\texttt{App}_\alpha(t_1,t_2),\ f_2\,t_1\,t_2\,r_1\,r_2)\ \in\ \mathit{rec}_{f_1 f_2 f_3}}$$
$$\frac{a\ \#\ (f_1,\,f_2,\,f_3) \quad (t,r)\ \in\ \mathit{rec}_{f_1 f_2 f_3}}{(\texttt{Lam}_\alpha(a,t),\ f_3\,a\,t\,r)\ \in\ \mathit{rec}_{f_1 f_2 f_3}} \tag{23}$$

[7]We use a different version of the FCB than actually introduced by Pitts. We shall show later that our version and one that closely resembles his are interderivable.

We shall show next that the relation $rec_{f_1 f_2 f_3}$ defines a function in the sense that for all lambda-terms t there exists a unique r so that $(t, r) \in rec_{f_1 f_2 f_3}$. From this we can again use standard techniques of HOL to obtain a function from $\mathtt{lam}_\alpha$ to α (see for example Slind [28]). We first show that in $rec_{f_1 f_2 f_3}$ the "result" r has finite support provided the functions f_1, f_2 and f_3 have finite support.

Lemma 12 (Finite Support) *If f_1, f_2 and f_3 have finite support, then $(t, r) \in rec_{f_1 f_2 f_3}$ implies that r has finite support.*

Proof By induction over the relation defined in (23). In the variable-case we have to show that $f_1\, a$ has finite support, which we inferred in Example 1 using our heuristic. The application and lambda-case are by similar calculations. □

In the proof of Theorem 3, we need the following lemma establishing that $rec_{f_1 f_2 f_3}$ is *equivariant* (see Pitts [26]).

Lemma 13 (Equivariance) *If $(t, r) \in rec_{f_1 f_2 f_3}$ holds then for all π, also $(\pi \bullet t, \pi \bullet r) \in rec_{(\pi \bullet f_1)(\pi \bullet f_2)(\pi \bullet f_3)}$ holds.*

Proof By induction over the rules given in (23). All cases are routine by pushing the permutation π into t and r, except in the lambda-case where we have to apply Lemma 3(iii) in order to infer $\pi \bullet a \,\#\, (\pi \bullet f_1, \pi \bullet f_2, \pi \bullet f_3)$ from $a \,\#\, (f_1, f_2, f_3)$. □

Next we can show the crucial lemma about $rec_{f_1 f_2 f_3}$ being a "function".

Lemma 14 (Existence and Uniqueness) *If f_1, f_2 and f_3 have finite support and f_3 satisfies the FCB, then $\exists! r.\ (t, r) \in rec_{f_1 f_2 f_3}$.*

Proof By the induction principle given in Theorem 2, where we set the function f to the constant function $\lambda\_\_.(f_1, f_2, f_3)$ and the induction context c to $\mathtt{unit}$.[8] Condition (i) of Theorem 2 holds because by assumption f_1, f_2 and f_3 have finite support. The only non-routine case then is the lambda-case with showing that $\exists! r.\ (\mathtt{Lam}_\alpha(a, t), r) \in rec_{f_1 f_2 f_3}$ holds. This is difficult, because for lambdas we do not have injectivity (see (18)). The proof in this case proceeds as follows.

The induction principle allows us to assume that $a \,\#\, (f_1, f_2, f_3)$, therefore the "existential" part of the lemma is immediate. In the "uniqueness" part we have to show that if $(\mathtt{Lam}_\alpha(a, t), f_3\, a\, t\, r) \in rec_{f_1 f_2 f_3}$ and also $(\mathtt{Lam}_\alpha(b, t'), f_3\, b\, t'\, r') \in rec_{f_1 f_2 f_3}$ with the equation $\mathtt{Lam}_\alpha(a, t) = \mathtt{Lam}_\alpha(b, t')$, then $f_3\, a\, t\, r = f_3\, b\, t'\, r'$ holds. By rule inversion we can assume that $b \,\#\, (f_1, f_2, f_3)$ and that there exists an r' such that $(t', r') \in rec_{f_1 f_2 f_3}$; further by the induction we know there is a unique r such that $(t, r) \in rec_{f_1 f_2 f_3}$. Now we show the following 6 facts:

(i) From $(t, r) \in rec_{f_1 f_2 f_3}$ and $(t', r') \in rec_{f_1 f_2 f_3}$ we can infer by Lemma 12 that r and r' are finitely supported. Therefore we can apply Proposition 1 to obtain a c with $c \,\#\, (f_1, f_2, f_3, t, t', r, r', a, b)$—all variables in the tuple have finite support.

[8]For this induction we cannot use the more convenient induction principle shown in (21), because functions do not have finitely supported type.

(ii) From (19) we have that $a \# \texttt{Lam}_\alpha(a, t)$ and $b \# \texttt{Lam}_\alpha(b, t')$. With (i) we can further infer that $c \# \texttt{Lam}_\alpha(a, t)$ and $c \# \texttt{Lam}_\alpha(b, t')$. From the assumption $\texttt{Lam}_\alpha(a, t) = \texttt{Lam}_\alpha(b, t')$, we can then use Lemma 4 to derive $(a\, c)\bullet\texttt{Lam}_\alpha(a, t) = (b\, c)\bullet\texttt{Lam}_\alpha(b, t')$, which implies that $\texttt{Lam}_\alpha(c, (a\, c)\bullet t) = \texttt{Lam}_\alpha(c, (a\, c)\bullet t')$; hence by (18) that $(a\, c)\bullet t = (b\, c)\bullet t'$.

(iii) From $(t, r) \in \mathit{rec}_{f_1\, f_2\, f_3}$, $(t', r') \in \mathit{rec}_{f_1\, f_2\, f_3}$ $a \# (f_1, f_2, f_3)$ and $b \# (f_1, f_2, f_3)$, we can infer by Lemma 4 and 13 that $((a\, c)\bullet t, (a\, c)\bullet r) \in \mathit{rec}_{f_1\, f_2\, f_3}$ and $((b\, c)\bullet t', (b\, c)\bullet r') \in \mathit{rec}_{f_1\, f_2\, f_3}$. Since by induction hypothesis $\exists! r.\, (t, r) \in \mathit{rec}_{f_1\, f_2\, f_3}$ we also have the fact that $\exists! r.\, ((a\, c)\bullet t, r) \in \mathit{rec}_{f_1\, f_2\, f_3}$. Thus we can use (ii) to infer that $(a\, c)\bullet r = (b\, c)\bullet r'$.

(iv) Using the FCB for f_3 and knowing that $a \# f_3$ and $b \# f_3$ as well as r and r' are finitely supported (from (i)), we can infer that $a \# f_3\, a\, t\, r$ and $b \# f_3\, b\, t'\, r'$ hold.

(v) Since $\mathit{supp}(f_3, a, t, r)\mathit{supports}(f_3\, a\, t\, r)$ and since $c \# (f_3, a, t, r)$ (from (i)), we know by Lemma 11 that $c \# f_3\, a\, t\, r$ holds. Similarly we can infer that $c \# f_3\, b\, t'\, r'$ holds.

(vi) Finally, in order to show that $f_3\, a\, t\, r = f_3\, b\, t'\, r'$ holds, it suffices by Lemma 4 and the facts derived in (iv) and (v) to show that $(a\, c)\bullet(f_3\, a\, t\, r) = (b\, c)\bullet(f_3\, b\, t'\, r')$ holds. This in turn is by (3) equivalent to $f_3\, c\, ((a\, c)\bullet t)\, ((a\, c)\bullet r) = f_3\, c\, ((b\, c)\bullet t')\, ((b\, c)\bullet r')$. By the facts derived in (ii) and (iii) we have that these terms are indeed equal. □

To prove our theorem about structural recursion we define $\mathit{rfun}_{f_1\, f_2\, f_3}\, t$ to be the unique r so that $(t, r) \in \mathit{rec}_{f_1\, f_2\, f_3}$. This is a standard construction in HOL-based theorem provers; it involves the HOL's definite description operator (see Isabelle's tutorial [21, Section 5.10.1]). The characteristic equations for $\mathit{rfun}_{f_1\, f_2\, f_3}$ are then determined by the definition of $\mathit{rec}_{f_1\, f_2\, f_3}$ given in (23). This completes the proof of Theorem 3.

As mentioned earlier, the FCB we use differs from the one introduced by Pitts. He defines this notion as follows:[9]

Definition 7 (FCB') A function f with type $\texttt{name} \Rightarrow \texttt{lam}_\alpha \Rightarrow \alpha \Rightarrow \alpha$ satisfies the *FCB'* provided:

$$\exists a.\, a \# f \;\wedge\; (\forall t\, r.\, \mathit{finite}(\mathit{supp}(r)) \;\Rightarrow\; a \# f\, a\, t\, r)\,.$$

It can be shown that in all cases where the recursion combinator is applied both versions of the FCB are interderivable.

Lemma 15 *Provided f is finitely supported, then the FCB holds if an only if the FCB' holds.*

Proof ($\Rightarrow$) Since f is finitely supported, we can choose using Proposition 1 an atom a such that $a \# f$. With this we can instantiate the FCB and obtain $\forall t\, r.\, \mathit{finite}(\mathit{supp}(r)) \;\Rightarrow\; a \# f\, a\, t\, r$ as we have to show. ($\Leftarrow$) We have that $a \# f$ and

[9]His definition of the FCB does not actually include $\mathit{finite}(\mathit{supp}(r))$, because he considers only finitely supported objects, and also does not include the quantification over t as he derives an iteration, rather than a recursion combinator.

finite(*supp*(r)) and need to show that $a\ \#\ f\, a\, t\, r$. By the FCB' we have an atom a' such that $a'\ \#\ f$ and $\forall t\, r.\ \mathit{finite}(\mathit{supp}(r)) \Rightarrow a'\ \#\ f a'\, t\, r$. Since *finite*(*supp*($(a\, a')^{-1}\bullet r$)) if an only if *finite*(*supp*(r)), we can infer $a'\ \#\ f a'\ ((a\, a')^{-1}\bullet t)\ ((a\, a')^{-1}\bullet r)$. By Lemma 3(iii) we can apply on both sides of # the swapping $(a\, a')$ and obtain

$$a\ \#\ f a\, ((a\, a')\bullet (a\, a')^{-1}\bullet t)\ ((a\, a')\bullet (a\, a')^{-1}\bullet r)$$

which by Lemma 1(i) is equivalent to $a\ \#\ f\, a\, t\, r$—the fact we had to show. □

The reason that we prefer our version of the FCB is that when establishing a universal quantified formula, Isabelle/HOL will just introduce an eigen-variable and then proceed to prove the "rest". This is in practice easier than generating a fresh atom and then instantiate the existential quantifier in the FCB'.

6 Examples

Finally, we can start to formalise Barendregt's informal proof of the substitution lemma (Fig. 1). All the constructions of the previous 3 sections would, due to their complexity, be of only academic value, *if* we can not automate them and hide the complexities from the user. However, we can! We shall illustrate this next.

The type $\mathtt{lam}_\alpha$ can be defined in Isabelle/HOL using the nominal datatype package by the two declarations:

$$\begin{array}{ll} \mathtt{atom\_decl}\ \mathtt{name} & \\ \mathtt{nominal\_datatype}\ \mathtt{lam}_\alpha = & \mathtt{Var}_\alpha\ \ \texttt{"name"} \\ & \mid \mathtt{App}_\alpha\ \ \texttt{"}\mathtt{lam}_\alpha \times \mathtt{lam}_\alpha\texttt{"} \\ & \mid \mathtt{Lam}_\alpha\ \ \texttt{"«name»}\mathtt{lam}_\alpha\texttt{"} \end{array}$$

where the first declaration establishes the type name with the properties described in Section 2; in the second declaration «...» indicates that a name is bound in $\mathtt{Lam}_\alpha$. With this information the nominal datatype package performs automatically the construction we described in Section 3 and also automatically derives the structural induction principles from Section 4 and the recursion combinator from Section 5 *without* any user interference. Furthermore, this package derives this reasoning infrastructure even for more complicated term-calculi that have more than one binder and binders may have different types.

After the declaration, we can then use the recursion combinator to define the capture-avoiding substitution function by stating the following characteristic equations:

$$\begin{array}{rl} \mathtt{Var}_\alpha(x)[y := t'] = & (\mathtt{if}\ x = y\ \mathtt{then}\ t'\ \mathtt{else}\ \mathtt{Var}_\alpha(x)) \\ \mathtt{App}_\alpha(t_1, t_2)[y := t'] = & \mathtt{App}_\alpha(t_1[y := t'], t_2[y := t']) \\ x\ \#\ (y, t') \implies \mathtt{Lam}_\alpha(x, t)[y := t'] = & \mathtt{Lam}_\alpha(x, t[y := t']) \end{array} \tag{24}$$

where in the clause for $\mathtt{Lam}_\alpha$ the precondition $x\ \#\ (y, t')$ corresponds to the usual condition that $x \neq y$ and x is not free in t'. Internally the nominal datatype package extracts the following functions for capture-avoiding substitution:

$$\begin{array}{rcl} s_1\, y\, t' & \stackrel{\text{def}}{=} & \lambda x.\ \mathtt{if}\ x = y\ \mathtt{then}\ t'\ \mathtt{else}\ \mathtt{Var}_\alpha(x) \\ s_2\, y\, t' & \stackrel{\text{def}}{=} & \lambda t_1\, t_2\, r_1\, r_2.\ \mathtt{App}_\alpha(r_2, r_1) \\ s_3\, y\, t' & \stackrel{\text{def}}{=} & \lambda x\, t\, r.\ \mathtt{Lam}_\alpha(x, r) \end{array}$$

In order to apply Theorem 3 with the instantiation $\mathit{rfun}_{(s_1\,y\,t')\,(s_2\,y\,t')\,(s_3\,y\,t')}$, Isabelle first needs to determine whether the result type of the function is a permutation type. Since substitution returns a $\mathtt{lam}_\alpha$-term, it can use Lemma 10(i) and automatically determine this fact. Next Isabelle asks the user to verify the preconditions of Theorem 3 about the functions $(s_1\,y\,t')$, $(s_2\,y\,t')$ and $(s_3\,y\,t')$ having finite support. It turns out that all of them are supported by the set $\mathit{supp}(y,t')$, which is finitely supported because of Lemma 5 (this can be determined automatically by Isabelle). To verify whether $\mathit{supp}(y,t')\,\mathit{supports}\,(s_1\,y\,t')$ holds, the tactic `finite_guess` does automatically the calculations shown in Example 2 and similar ones for the cases $(s_2\,y\,t')$ and $(s_3\,y\,t')$. Next Isabelle asks the user to verify the FCB for $(s_3\,y\,t')$ which amounts to showing that

$$\forall a\,t\,r.\; a \,\#\, (s_3\,y\,t') \;\wedge\; \mathit{finite}(\mathit{supp}(r)) \;\Rightarrow\; a \,\#\, \mathtt{Lam}_\alpha(a,r)$$

holds. This can be done by a simple application of the property given in (19). Last, Isabelle asks the user to verify that the precondition of the recursion combinator in the lambda-case, namely that $x \,\#\, (s_1\,y\,t', s_2\,y\,t', s_3\,y\,t')$ is implied by the precondition $x \,\#\, (y,t')$ given in (24). Since, as indicated earlier, all these functions are supported by $\mathit{supp}(y,t')$, Isabelle can determine this automatically with the help of a tactic. This completes the definition of capture-avoiding substitution. The Isabelle code for this is:

```
consts
  subst :: "lamα ⇒ name ⇒ lamα ⇒ lamα"  ("_[_:=_]"
                                        [100,100,100] 100)

nominal_primrec
  "Varα(x)[y:=t’] = (if x=y then t’ else Varα(x))"
  "Appα(t1,t2)[y:=t’] = Appα(t1[y:=t’],t2[y:=t’])"
  "x#(y,t’) ⟹ Lamα(x,t)[y:=t’] = Lamα(x,t[y:=t’])"
by (finite_guess+,(rule TrueI)+, simp add: abs_fresh,
    fresh_guess+)
```

where in the first two lines we declare the type of the substitution function and introduce nicer syntax for writing this function. The line starting with **by** contains the proof for showing that the characteristic functions of substitution are finitely supported, that the FCB is satisfied and that the precondition $\mathtt{x} \,\#\, (\mathtt{y},\mathtt{t}')$ is sufficient for instantiating the recursion combinator.

Having the substitution function at our disposal, we can now formalise Barendregt's proof of the substitution lemma. First we have to formalise the fact that $x \notin FV(L)$ implies $L[x := P] = L$ whose proof is omitted by Barendregt.

Lemma 16 (Forget) *If* $x \,\#\, L$ *then* $L[x := P] = P$.

Proof The proof proceeds by induction over L using (21) with c instantiated to (x,P). In the variable case we have to show that $\mathtt{Var}_\alpha(y)[x := P] = \mathtt{Var}_\alpha(y)$ under the assumption that $x \,\#\, \mathtt{Var}_\alpha(y)$. This assumption is equivalent to $x \,\#\, y$, which is in turn equivalent to $x \neq y$, allowing us to apply (24) to prove this case. In the lambda-case we have the induction hypothesis $\forall x\,P.\; x \,\#\, L_1 \Rightarrow L_1[x := P] = L_1$ and have to show that $\texttt{Lam}_\alpha(y, L_1)[x := P] = \texttt{Lam}_\alpha(y, L_1)$ under the assumption that $x \# \texttt{Lam}_\alpha(y, L_1)$ holds. The induction in allows us further to assume that $y \# (x, P)$—(x, P) is the induction context and the point of (21) is that we can assume the binder is fresh w.r.t. this context. Therefore we can move the substitution under the binder, namely $\texttt{Lam}_\alpha(y, L_1)[x := P] = \texttt{Lam}_\alpha(y, L_1[x := P])$, and also infer by (19) that $x \# L_1$. This allows us to apply the induction hypothesis and we are done. The application case is trivial. □

Using Isabelle's automatic proof-tools one can formalise this proof with:

```
lemma forget:
  assumes a: "x#L"
  shows "L[x:=P] = L"
using a by (nominal_induct L avoiding: x P rule: lamα.induct)
           (auto simp add: abs_fresh fresh_atm)
```

where `abs_fresh` corresponds to the property given in (19) and the lemma `fresh_atm` to the fact that for atoms x and y, $x \# y$ holds if and only if $x \neq y$. The method `nominal_induct` (see Wenzel [38]) brings the induction principle, called $\texttt{lam}_\alpha\texttt{.induct}$, automatically to the form needed in (21)—we only have to state over which variable the induction is done and what the induction context is, that is the variables to avoid.

Next we need to show a lemma whose need is not immediately apparent by looking at Barendregt's informal proof. However, in the lambda-case where Barendregt pulls out a substitution from under the binder, namely in the step

$$\lambda z.(M_1[y := L][x := N[y := L]]) \equiv (\lambda z.M_1)[y := L][x := N[y := L]]$$

we need to know that z is not free in $N[y := L]$. But by the variable convention we only know that z is not free in $N$and L. In a formalisation, this fact needs to be established explicitly. It can be done in Isabelle with

```
lemma fresh_fact:
  fixes z::"name"
  assumes a: "z#N" "z#L"
  shows "z#N[y:=L]"
using a by (nominal_induct N avoiding: z y L rule: lamα.induct)
           (auto simp add: abs_fresh fresh_atm)
```

where z needs to be given an explicit type-annotation so that Isabelle can determine its type. The substitution lemma can now be formalised with:

```
lemma substitution_lemma:
  assumes a: "x≠y" "x#L"
  shows "M[x:=N][y:=L] = M[y:=L][x:=N[y:=L]]"
using a by (nominal_induct M avoiding: x y N L rule: lamα.induct)
           (auto simp add: fresh_fact forget)                    (25)
```

A formalised proof of this lemma mentioning much more details is shown in Fig. 3.

Other proofs we formalised in a similar fashion are the Church-Rosser proof from Barendregt [5, pp. 60–62] and [29], the strong normalisation proof given in

```
lemma substitution_lemma:
  assumes a: "x≠y" "x ♯ L"
  shows "M[x:=N][y:=L] = M[y:=L][x:=N[y:=L]]"
using a
proof (nominal_induct M avoiding: x y N L rule: lam_α.induct)
  case (Var_α z)                                                   (Case 1: variables)
  show "Var_α(z)[x:=N][y:=L] = Var_α(z)[y:=L][x:=N[y:=L]]" (is "?lhs=?rhs")
  proof -
    { assume  "z=x"                                                        (Case 1.1)
      have 1: "?lhs = N[y:=L]" using ‘z=x‘ by simp
      have 2: "?rhs = N[y:=L]" using ‘z=x‘ ‘x≠y‘ by simp
      from 1 2 have "?lhs = ?rhs"  by simp
    }
    moreover
    { assume "z=y" and "z≠x"                                               (Case 1.2)
      have 1: "?lhs = L"              using ‘z≠x‘ ‘z=y‘ by simp
      have 2: "?rhs = L[x:=N[y:=L]]" using ‘z=y‘ by simp
      have 3: "L[x:=N[y:=L]] = L"    using ‘x ♯ L‘ by (simp add: forget)
      from 1 2 3 have "?lhs = ?rhs" by simp
    }
    moreover
    { assume "z≠x" and "z≠y"                                               (Case 1.3)
      have 1: "?lhs = Var_α z" using ‘z≠x‘ ‘z≠y‘ by simp
      have 2: "?rhs = Var_α z" using ‘z≠x‘ ‘z≠y‘ by simp
      from 1 2 have "?lhs = ?rhs" by simp
    }
    ultimately show "?lhs = ?rhs" by blast
  qed
next
  case (Lam_α z M1)                                                 (Case 2: lambdas)
  have ih: "⟦x≠y; x ♯ L⟧ ⟹ M1[x:=N][y:=L] = M1[y:=L][x:=N[y:=L]]" by fact
  have vc: "z ♯ x" "z ♯ y" "z ♯ N" "z ♯ L" by fact                (variable convention)
  hence "z ♯ N[y:=L]" by (simp add: fresh_fact)
  show "Lam_α(z,M1)[x:=N][y:=L] = Lam_α(z,M1)[y:=L][x:=N[y:=L]]" (is "?lhs=?rhs")
  proof -
    have "?lhs = Lam_α(z,M1[x:=N][y:=L])" using vc by simp
    also have "... = Lam_α(z,M1[y:=L][x:=N[y:=L]])" using ih ‘x≠y‘ ‘x ♯ L‘ by simp
    also have "... = Lam_α(z,M1[y:=L])[x:=N[y:=L]]" using vc ‘z ♯ N[y:=L]‘ by simp
    also have "... = ?rhs" using  vc by simp
    finally show "?lhs = ?rhs" by simp
  qed
next
  case (App_α M1 M2)                                            (Case 3: applications)
  thus "App_α(M1,M2)[x:=N][y:=L] = App_α(M1,M2)[y:=L][x:=N[y:=L]]" by simp
qed
```

Fig. 3 A formalised proof of Barendregt's substitution lemma using the Isabelle's Isar language. This proof contains all reasoning steps given in extreme detail. An automated version of this proof, given in (25), is only 5 lines long. The crucial point in both proofs, however, is that in the lambda-case we have the assumptions labelled with vc available. They allow us to easily formalise Barendregt's slick informal proof, shown in Fig. 1, which uses the variable convention

Girard et al. [12, pp. 42–46], the strong normalisation proof for cut-elimination from Urban [31], the correctness proof of the type-inference algorithm W from Leroy [18, pp. 26–31] and the logical relation proof for algorithmic equality between simply-typed lambda-terms given in Crary [7, pp. 223–244] and between LF-terms given by Harper and Pfenning in [15]. These proofs are more complicated than the proofs we have given above and need some manual reasoning. All proofs are included in the distribution of the nominal datatype package available from http://isabelle.in.tum.de/nominal/

7 Related Work

There are many approaches to formal treatments of binders; this section describes the ones from which we have drawn inspiration and also work reported in Ambler et al. [1], Aydemir et al. [2] and Homeier [16].

Our work uses many ideas from the nominal logic work by Pitts et al. [11, 26, 27]. 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 base the implementation on a lower level and would have to redo the effort that has been spend to develop Isabelle/HOL. This was attempted in Gabbay [10], but the attempt was quickly abandoned.

Closely related to our work is Gordon and Melham [14], which has been applied and much further developed by Norrish [22, 23]. Gordon and Melham's work states five axioms characterising α-equivalence and then shows that a model based on de-Bruijn indices satisfies these axioms. This is somewhat similar to our approach where we construct explicitly the set $\mathtt{lam}_\alpha$. In [14] Gordon and Melham give an induction principle that requires in the lambda-case to prove (using their notation)

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

That means they have to prove $P(LAM\ x\ t)$ for a variable x for which nothing can be assumed; explicit α-renamings are then often necessary in order to get proofs through. This inconvenience has been alleviated by the version of structural induction given in [13] and [23], where the lambda-case is as follows

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

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 where we have to specify an induction context, but we claim that our version based on freshness fits better with informal practice (recall Fig. 1 where Barendregt states that z is fresh w.r.t. x, y, N and L) and can make better use of the automatic infrastructure of Isabelle (namely the axiomatic type-classes enforce the finite-support property).

Gordon and Melham [14] do not consider the case of rule inductions over inductively defined predicates. This has been done in [33, 34]. It turns out that while the variable convention can be built into every structural induction principle, like our Theorem 2, this is not the case for rule induction principles. In [33] the authors give an example where the variable convention can lead to faulty reasoning. The nominal datatype package prevents this by introducing conditions for when an inductive definition is compatible with the variable convention and only derives a strong rule induction principle for those that satisfy these conditions.

Like our $\texttt{lam}_\alpha$, HOAS uses functions to encode lambda-abstractions; it comes in two flavours: *weak* HOAS [8] and *full* HOAS [25]. 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 making 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 proofs based on logical relation arguments), which in full HOAS seem rather difficult to encode when one at the same time wants to reap the benefits of a HOAS-representation. Another advantage we see is that by inductively defining $\texttt{lam}_\alpha$, one has induction for "free", whereas induction requires considerable effort in full HOAS. The work by Ambler et al. [1] on the Hybrid-system provides full HOAS on top of Isabelle/HOL. For this they use a de-Bruijn encoding and construct a type corresponding to full HOAS. This construction is somewhat similar to our subset-construction from Section 3. However, their construction is done manually and only for one datatype, while we have automatic support to do the subset construction for any nominal datatype.

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.

Recently, Homeier [16] introduced a quotient package for HOL4 that helps with defining alpha-equivalence classes (this package supports quotients by any equivalence relation) and with lifting theorems from the "raw" version of the datatype to the quotient. Norrish makes use of this package in [23]. This package would help us with the construction of $\texttt{lam}_\alpha$, but would have only little impact on obtaining the strong induction principles and the recursion combinator. Nevertheless we look forward to a port of Homeier's package to Isabelle/HOL. It will simplify our work when we consider more complicated binding structures.

Aydemir et al. [2] reported work in progress for providing nominal reasoning techniques in Coq. Essentially, they derive more or less automatically from a specification of a nominal datatype an axiomatisation of nominal concepts in Coq and in case of the lambda-calculus use a Gordon-Melham representation to justify their axiomatisation. However, this justification needs to be done manually, while with our constructions we provide the justification completely automatically. Judging from recent work, the authors seem to have "abandoned" this work in favour of working with a locally nameless representation of α-equated lambda-terms [3].

8 Conclusion

The paper [4], 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. The nominal datatype package has no problems with generalising the results reported here to more complicated term-calculi. For example, there is already work by Bengtson using the nominal datatype package for formalising the π-calculus [6]; Tobin-Hochstadt and Felleisen used it to verify their work on Typed Scheme [30].

There has also been work on extending strong induction principles to rule inductions [33, 34]. The real challenge has been and still is to generalise all the necessary reasoning infrastructure to more general binding structures. While there is no problem in the nominal datatype package with iterated binders, as in `Foo` «name»«name»__, and binders of different type, as in `Bar` «name»__ «coname»__, it is not yet possible to have, for example, a finite set of binders in a term-constructor. A typical example where such a generalisation is very helpful is the Hindley-Milner typing-algorithm where one has type-schemes of the form $\forall\{a_1, \ldots, a_n\}.ty$. Such type-schemes can at the moment only be represented by encoding them as an iterated list of single binders. To work out the details for the generalisation of binding structures and to implement them is future work. Future work also includes the generalisation of our recursion combinator to work with varying parameters. This has been treated in [23, 27], but it seems difficult to adapt their results to our setting.

Acknowledgements I am very grateful to Andy Pitts and Michael Norrish for the many discussions with them on the subject of the paper. Stefan Berghofer and Markus Wenzel have been helpful *beyond measure* with implementing the work reported here. Christine Tasson helped with the early parts of the work. Julien Narboux provided helpful comments.

References

1. Ambler, S.J., Crole, R.L., Momigliano, A.: Combining higher order abstract syntax with tactical theorem proving and (co)induction. In: Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs). LNCS, vol. 2410, pp. 13–30. Hampton, 20–23 August 2002
2. Aydemir, B., Bohannon, A., Weihrich, S.: Nominal reasoning techniques in Coq (work in progress). In: Proceedings of the International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP). ENTCS, pp. 60–68. Seattle, 16 August 2006
3. Aydemir, B., Charguéraud, A., Pierce, B.C., Pollack, R., Weirich, S.: Engineering formal metatheory. In: Proceedings of the 35rd Symposium on Principles of Programming Languages (POPL), pp. 3–15. ACM, New York (2008)
4. Aydemir, B.E., Bohannon, A., Fairbairn, M., Foster, J.N., Pierce, B.C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., Zdancewic, S.: Mechanized metatheory for the masses: the Poplmark challenge. In: Proceedings of the 18th International Conference on Theorem Proving in Higher-Order Logics (TPHOLs). LNCS, vol. 3603, pp. 50–65. Oxford, 22–25 August 2005
5. Barendregt, H.: The Lambda Calculus: Its Syntax and Semantics. Studies in Logic and the Foundations of Mathematics, vol. 103. North-Holland, Amsterdam (1981)
6. Bengtson, J., Parrow, J.: Formalising the pi-Calculus using nominal logic. In: Proceedings of the 10th International Conference on Foundations of Software Science and Computation Structures (FOSSACS). LNCS, vol. 4423, pp. 63–77. Braga, March 2007
7. Crary, K.: Logical relations and a case study in equivalence checking. In: Pierce, B.C. (ed.) Advanced Topics in Types and Programming Languages, pp. 223–244. MIT, Cambridge (2005)
8. Despeyroux, J., Felty, A., Hirschowitz, A.: Higher-order abstract syntax in Coq. In: Proceedings of the 2nd International Conference on Typed Lambda Calculi and Applications (TLCA). LNCS, vol. 902, pp. 124–138. Springer, New York (1995)
9. Dowek, G., Hardin, T., Kirchner, C.: Higher-order unification via explicit substitutions. Inf. Comput. **157**, 183–235 (2000)

10. Gabbay, M.J.: A theory of inductive definitions with α-equivalence. PhD thesis, University of Cambridge (2001)
11. Gabbay, M.J., Pitts, A.M.: A new approach to abstract syntax with variable binding. Form. Asp. Comput. **13**, 341–363 (2001)
12. Girard, J.-Y., Lafont, Y., Taylor, P.: Proofs and Types. Cambridge Tracts in Theoretical Computer Science, vol. 7. Cambridge University Press, Cambridge (1989)
13. Gordon, A.D.: A mechanisation of name-carrying syntax up to alpha-conversion. In: Proceedings of the 6th International Workshop on Higher-order Logic Theorem Proving and its Applications (HUG). LNCS, vol. 780, pp. 414–426. Vancouver, 11–13 August 1994
14. Gordon, A.D., Melham, T.: Five axioms of alpha conversion. In: Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics (TPHOLs). LNCS, vol. 1125, pp. 173–190. Turku, 26–30 August 1996
15. Harper, R., Pfenning, F.: On equivalence and canonical forms in the LF type theory. ACM Trans. Comput. Log. **6**(1), 61–101 (2005)
16. Homeier, P.: A design structure for higher order quotients. In: Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs). LNCS, vol. 3603, pp. 130–146. Oxford, 22–25 August 2005
17. Kleene, S.C.: Disjunction and existence under implication in elementary intuitionistic formalisms. J. Symb. Log. **27**(1), 11–18 (1962)
18. Leroy, X.: Polymorphic typing of an algorithmic language. Ph.D. thesis, University Paris 7, INRIA Research Report, No 1778 (1992)
19. Melham, T.: Automating recursive type definitions in higher order logic. Technical Report 146, Computer Laboratory, University of Cambridge, September (1988)
20. Melham, T.: Automating recursive type definitions in higher order logic. In: Current Trends in Hardware Verification and Automated Theorem Proving, pp. 341–386. Springer, New York (1989)
21. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle HOL: A proof assistant for higher-order logic. LNCS, vol. 2283. Springer, New York (2002)
22. Norrish, M.: Recursive function definition for types with binders. In: Proceedings of the 17th International Conference Theorem Proving in Higher Order Logics (TPHOLs). LNCS, vol. 3223, pp. 241–256. Park City, 14–17 September 2004
23. Norrish, M.: Mechanising λ-calculus using a classical first order theory of terms with permutation. High. Order Symb. Comput. **19**, 169–195 (2006)
24. Paulson, L.: Defining functions on equivalence classes. ACM Trans. Comput. Log. **7**(4), 658–675 (2006)
25. Pfenning, F., Elliott, C.: Higher-order abstract syntax. In: Proceedings of the 10th Conference on Conference on Programming Language Design and Implementation (PLDI), pp. 199–208. ACM, New York (1989)
26. Pitts, A.M.: Nominal logic, a first order theory of names and binding. Inf. Comput. **186**, 165–193 (2003)
27. Pitts, A.M.: Alpha-structural recursion and induction. J. ACM **53**, 459–506 (2006)
28. Slind, K.: Wellfounded schematic definitions. In: Proceedings of the 17th International Conference on Automated Deduction (CADE). LNCS, vol. 1831, pp. 45–63. Pittsburgh, 17–20 June 2000
29. Takahashi, M.: Parallel reductions in lambda-calculus. Inf. Comput. **118**(1), 120–127 (1995)
30. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of typed scheme. In: Proceedings of the 35rd Symposium on Principles of Programming Languages (POPL), pp. 395–406. ACM, New York (2008)
31. Urban, C.: Classical logic and computation. Ph.D. thesis, Cambridge University, October (2000)
32. Urban, C., Berghofer, S.: A recursion combinator for nominal datatypes implemented in Isabelle/HOL. In: Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR). LNAI, vol. 4130, pp. 498–512. Seattle, 17–20 August 2006
33. Urban, C., Berghofer, S., Norrish, M.: Barendregt's variable convention in rule inductions. In: Proceedings of the 21st International Conference on Automated Deduction (CADE). LNAI, vol. 4603, pp. 35–50. Bremen, 17–20 July 2007
34. Urban, C., Norrish, M.: A formal treatment of the Barendregt variable convention in rule inductions. In: Proceedings of the 3rd International ACM Workshop on Mechanized Reasoning about Languages with Variable Binding and Names, pp. 25–32. ACM, New York (2005)
35. Urban, C., Pitts, A.M., Gabbay, M.J.: Nominal unification. Theor. Comp. Sci. **323**(1–2), 473–497 (2004)

36. Urban, C., Tasson, C.: Nominal techniques in Isabelle/HOL. In: Proceedings of the 20th International Conference on Automated Deduction (CADE). LNCS, vol. 3632, pp. 38–53. Tallinn, 22–27 July 2005
37. Wenzel, M.: Using axiomatic type classes in Isabelle. Manual in the Isabelle distribution. http://isabelle.in.tum.de/doc/axclass.pdf (2000)
38. Wenzel, M.: Structured induction proofs in Isabelle/Isar. In: Proceedings of the 5th International Conference on Mathematical Knowledge Management (MKM). LNAI, vol. 4108, pp. 17–30. Wokingham, 11–12 August 2006