A Recursion Combinator for Nominal Datatypes Implemented in Isabelle/HOL

Christian Urban and Stefan Berghofer

Technische Universität München
{urbanc,berghofe}@in.tum.de

Abstract. The nominal datatype package implements an infrastructure in Isabelle/HOL for defining languages involving binders and for reasoning conveniently about alpha-equivalence classes. Pitts stated some general conditions under which functions over alpha-equivalence classes can be defined by a form of structural recursion and gave a clever proof for the existence of a primitive-recursion combinator. We give a version of this proof that works directly over nominal datatypes and does not rely upon auxiliary constructions. We further introduce proving tools and a heuristic that made the automation of our proof tractable. This automation is an essential prerequisite for the nominal datatype package to become useful.

Keywords: Lambda-calculus, proof assistants, nominal logic, primitive recursion

1 Introduction

The infrastructure provided by various datatype packages [2,6] dramatically simplifies the embedding of languages *without* binders inside HOL-based proof assistants [4]. Because such proof assistants emphasise the development of theories by definition rather than axiom postulation, simple tasks like reasoning about lists would be fiendishly complicated without such an infrastructure.

The purpose of the nominal datatype package[1] is to provide an infrastructure in Isabelle/HOL for embedding languages *with* binders and for reasoning conveniently about them. Many ideas for this package originate from the nominal logic work by Pitts ([7], see also [11]). Using this package, the user can define the terms of, for example, the lambda-calculus as follows:

```
atom_decl name
nominal_datatype lam = Var "name"
                     | App "lam" "lam"                    (1)
                     | Lam "«name»lam"
```

where `name` is declared to be the type for variables and where «. . .» indicates that a name is bound in a lambda-term. Despite similarities with the usual datatype declaration of Isabelle/HOL and despite the fact that after this declaration one

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

can, as usual, write $(\texttt{Var}\,a)$, $(\texttt{App}\,t_1\,t_2)$ and $(\texttt{Lam}\,a\,t)$ for the lambda-terms, the code above does *not* define a datatype in the usual sense, but rather defines *alpha-equivalence* classes. Indeed we can show that the *equation*

$$(\texttt{Lam}\,a\,(\texttt{Var}\,a)) = (\texttt{Lam}\,b\,(\texttt{Var}\,b)) \tag{2}$$

holds for the nominal datatype `lam`.

By using alpha-equivalence classes and strong induction principles, that is induction principles which have the usual variable convention already built-in [10,11], one can often formalise with great ease informal proofs about languages involving binders. One example[2] is Barendregt's informal proof of the substitution lemma shown in Fig. 1. This lemma establishes a "commutation-property" for the function of capture-avoiding substitution. This substitution function is usually defined by the three clauses:

$$\begin{aligned}(\texttt{Var}\,a)[b:=t'] &= \texttt{if}\ a=b\ \texttt{then}\ t'\ \texttt{else}\ (\texttt{Var}\,a)\\ (\texttt{App}\,t_1\,t_2)[b:=t'] &= \texttt{App}\,(t_1[b:=t'])\,(t_2[b:=t'])\\ (\texttt{Lam}\,a\,t)[b:=t'] &= \texttt{Lam}\,a\,(t[b:=t'])\end{aligned} \tag{3}$$

where the last clause has the side-constraint that $a \neq b$ and $a \# t'$ (the latter is the nominal logic terminology for a not being free in t'). While it is trivial to define functions by primitive recursion over datatypes, this is not so for nominal datatypes, because there functions need to respect equations such as (2); if *not*, then one can easily prove false in Isabelle/HOL. Consider for example the following two definitions that are intended, respectively, to calculate the set of bound names and to return the set of immediate subterms of a lambda-term:

$$\begin{aligned}bn\,(\texttt{Var}\,a) &= \varnothing & ist\,(\texttt{Var}\,a) &= \varnothing\\ bn\,(\texttt{App}\,t_1\,t_2) &= (bn\,t_1)\cup(bn\,t_2) & ist\,(\texttt{App}\,t_1\,t_2) &= \{t_1,t_2\}\\ bn\,(\texttt{Lam}\,a\,t) &= \{a\}\cup(bn\,t) & ist\,(\texttt{Lam}\,a\,t) &= \{t\}\end{aligned}$$

If *bn* and *ist* were functions, then they must return the same result for the two terms in (2)—that means $\{a\} = \{b\}$ in case of *bn* and $\{\texttt{Var}\,a\} = \{\texttt{Var}\,b\}$ in case of *ist*; however, if we assume $a \neq b$, then both equations lead to contradictions. Pitts gave in [8,9] some general conditions that allow to define the substitution function by the clauses in (3), but exclude definitions such as *bn* and *ist*.

In earlier versions of the nominal datatype package one could define functions on a case-by-case basis, but this involved some rather non-trivial reasoning—there was no uniform method for defining functions over the structure of nominal datatypes. Pitts gave in [9] two proofs for the existence of a primitive recursion operator that allows one to define functions by stating a clause for each term-constructor. His first proof is fairly complicated and involves auxiliary constructions: for example he does not show the existence directly for alpha-equivalence classes, but indirectly via the existence of primitive recursion for the corresponding "un-quotient" type (in case of the lambda-calculus "un-quotient" means

[2] Other examples such as Church-Rosser and strong normalisation can be found in the distribution of the nominal datatype package.

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. □

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

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

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

Fig. 1. The informal proof shown at the top is taken from Barendregt [1]. In the lambda-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. Using the nominal datatype package one can formalise this proof in Isabelle/HOL by establishing first the lemmas `forget` and `fresh_fact`. Although hidden by the `auto`-tactic, the formal proof follows quite closely Barendregt's reasoning, including his use of the variable convention. One important part of this formalisation is the definition of the function for capture-avoiding substitution.

lambdas are defined having the type `name`$\times$`lam`). Norrish formalised this proof quite faithfully, but needed, despite using the quotient package by Homeier [5] that automated some parts of the proof, approximately 600 lines of extremely dense HOL4-code. It is a fair comment[3] to say that this formalisation and the one included in early versions of the nominal datatype package are far too difficult for an automation.

We present in this paper a formalisation of Pitts second proof whose length is in case of the lambda-calculus only 150 lines of readable Isar-code. In contrast to [9], our proof is a direct proof not relying on any auxiliary constructions; also we prove directly the existence of a recursion combinator and do not make a detour via an iteration combinator. The automation of this proof will be part of the forthcoming release of the nominal datatype package. To ease the automation, we introduce here a heuristic that allowed us to write a tactic for solving some re-occurring proof obligations to do with finite support.

The paper introduces in Sec. 2 the central notions from the nominal logic work and some brief comments on the implementation of the nominal datatype package. Sec. 3 gives the proof of the *structural recursion combinator* for the type `lam`. Some examples are given in Sec. 4. The general case for all nominal datatypes is mentioned very briefly in Sec. 5; Sec. 6 draws conclusions and mentions related work.

2 Preliminaries

As can be seen from the declaration of `lam` shown in (1), there is a single type of variables in the lambda-calculus. We denote this type here by `name` and in the tradition of the nominal logic work call its elements *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.

Permutations are finite bijective mappings from atoms to atoms; as in [11] permutations are implemented as finite lists whose elements are swappings (that is pairs of atoms). We write such permutations as $(a_1\, b_1)(a_2\, b_2)\cdots(a_n\, b_n)$; the empty list $[]$ stands for the identity permutation. A permutation π *acting* on an atom a is defined as:

$$\begin{aligned}
[]\bullet a &\overset{\text{def}}{=} a\\
((a_1\, a_2) :: \pi)\bullet a &\overset{\text{def}}{=} \begin{cases} a_2 & \text{if } \pi\bullet a = a_1\\ a_1 & \text{if } \pi\bullet a = a_2\\ \pi\bullet a & \text{otherwise}\end{cases}
\end{aligned} \tag{4}$$

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

[3] personal communication with Norrish

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 (4) of a permutation acting on an atom, the nominal datatype package takes advantage of the overloading mechanism in Isabelle by declaring a constant, written infix as $(-)\bullet(-)$, with the polymorphic type $(\texttt{name} \times \texttt{name})\,\texttt{list} \Rightarrow \alpha \Rightarrow \alpha$. A definition of the permutation action can then be given separately for each type-constructor; for lists, products, unit, sets and functions the definitions are as follows:

$$\begin{array}{lrcl}
\alpha\ \texttt{list}: & \pi\bullet[] & \overset{\text{def}}{=} & [] \\
 & \pi\bullet(x :: t) & \overset{\text{def}}{=} & (\pi\bullet x) :: (\pi\bullet t) \\
\alpha_1 \times \alpha_2: & \pi\bullet(x_1, x_2) & \overset{\text{def}}{=} & (\pi\bullet x_1, \pi\bullet x_2) \\
\texttt{unit}: & \pi\bullet() & \overset{\text{def}}{=} & () \\
\alpha\ \texttt{set}: & \pi\bullet X & \overset{\text{def}}{=} & \{\pi\bullet x \mid x \in X\} \\
\alpha_1 \Rightarrow \alpha_2: & \pi\bullet fn & \overset{\text{def}}{=} & \lambda x.\pi\bullet(fn\,(\pi^{-1}\bullet x))
\end{array} \tag{5}$$

The nominal datatype package also defines a permutation action for the type `lam`, which behaves as follows:

$$\begin{array}{rcl}
\pi\bullet(\texttt{Var}\,a) & = & \texttt{Var}\,(\pi\bullet a) \\
\pi\bullet(\texttt{App}\,t_1\,t_2) & = & \texttt{App}\,(\pi\bullet t_1)\,(\pi\bullet t_2) \\
\pi\bullet(\texttt{Lam}\,a\,t) & = & \texttt{Lam}\,(\pi\bullet a)\,(\pi\bullet t)
\end{array} \tag{6}$$

(Since we have not yet derived a mechanism for defining functions by structural recursion over nominal datatypes, this permutation action cannot yet be defined directly, but needs to be lifted from the representing type for `lam`.)

The nominal datatype package assumes that every permutation action defined for a type satisfies three basic properties. For this we use the terminology from [11] of a *permutation type*:

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

$$\begin{array}{rl}
(i) & []\bullet x = x \\
(ii) & (\pi_1 @ \pi_2)\bullet x = \pi_1\bullet(\pi_2\bullet x) \\
(iii) & \textit{if } \pi_1 \sim \pi_2 \textit{ then } \pi_1\bullet x = \pi_2\bullet x
\end{array}$$

These properties entail that the permutations action behaves on elements of permutation types as one expects, for example we have $\pi^{-1}\bullet(\pi\bullet x) = x$. We note that:

Lemma 1. *Given pt_α, pt_{α_1} and pt_{α_2}, the types* `name`, `unit`, α `list`, α `set`, $\alpha_1 \times \alpha_2$, $\alpha_1 \Rightarrow \alpha_2$ *and* `lam` *are also permutation types.*

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

The permutation action on 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

$$\pi \bullet (fn\, x) = (\pi \bullet fn)(\pi \bullet x) \tag{7}$$

in Isabelle/HOL; this is because we have $\pi^{-1} \bullet (\pi \bullet x) = x$ for x of type α_1.

The most interesting feature of the nominal logic work is that as soon as one fixes a permutation action for a type, then the *support* for the elements of this type, very roughly speaking their set of free atoms, is fixed as well [3]. 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*

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

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

The advantage of the quite unusual definition of support is that it generalises the notion of a free variable to functions (a fact that will play an important rôle later on). Unwinding this definition and the permutation action given in (5) and (6), one can calculate the support for the types:

$$\begin{array}{ll}
\texttt{name}: & supp(a) = \{a\} \\
\alpha_1 \times \alpha_2: & supp(x_1, x_2) = supp(x_1) \cup supp(x_2) \\
\texttt{unit}: & supp(()) = \varnothing \\
\alpha\,\texttt{list}: & supp([]) = \varnothing \\
 & supp(x :: xs) = supp(x) \cup supp(xs) \\
\texttt{lam}: & supp(\texttt{Var}\,a) = \{a\} \\
 & supp(\texttt{App}\,t_1\,t_2) = supp(t_1) \cup supp(t_2) \\
 & supp(\texttt{Lam}\,a\,t) = supp(t) - \{a\}
\end{array} \tag{8}$$

where the last clause uses the fact that alpha-equivalence for the type `lam` is given by:

$$\texttt{Lam}\,a\,t = \texttt{Lam}\,b\,t' \;\Leftrightarrow\; (a = b \wedge t = t') \;\vee\; (a \neq b \wedge t = (a\,b) \bullet t' \wedge a \# t') \tag{9}$$

For permutation types the notion of support and freshness have very good properties as mentioned next (proofs are in [11]):

$$\pi \bullet a \# \pi \bullet x \text{ if and only if } a \# x \tag{10}$$

$$\text{if } a \# x \text{ and } b \# x \text{ then } (a\,b) \bullet x = x \tag{11}$$

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 $finite(supp(x))/infinite(supp(x))$ to indicate that an element x from a permutation type has finite/infinite support. The following holds:

Lemma 2. *Given fs_α, fs_{α_1} and fs_{α_2}, the types* `name`, `unit`, *α* `list`, *$\alpha_1 \times \alpha_2$ and* `lam` *are also finitely supported permutation types.*

Proof. Routine proofs using the calculations given in (8).

The crucial property entailed by Def. 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 choose 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 [12] for most constructions occurring in informal proofs: Isabelle has to just examine the types of the construction using Lem. 2. Unfortunately, this is more difficult in case of functions, because not all functions have finite support, even if their domain and codomain are finitely supported permutation types (see [9, Example 9]). Therefore we have to establish whether a function has finite support on a case-by-case basis. In order to automate the corresponding proof obligations, we use the auxiliary notion of *supports* [3].

Definition 5. *A set S of atoms* supports *an x, 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 3. *If a set S is finite and S supports x, then $supp(x) \subseteq S$.*

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. □

Lem. 3 gives us in many cases some effective 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. For this we use the following heuristic:

Heuristic 1 *Assume an HOL-function, say fn, is given as a lambda-term. The support of the tuple consisting of the free variables of fn supports this function, more formally we have $supp(FV(fn))$ supports fn, where we assume FV is defined as usual, except that we group the free variables in tuples, instead of finite sets.*

This is a heuristic, because it can very likely not be established as a lemma inside Isabelle/HOL, since 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 $\texttt{name} \Rightarrow \alpha$. 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, that means $FV(fn) = (f_1, c)$. 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 (Def. 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:

$$(a\, b)\bullet fn \overset{\text{def}}{=} (a\, b)\bullet(f_1\, c) \overset{\text{by (7)}}{=} ((a\, b)\bullet f_1)\,((a\, b)\bullet c) \overset{(*)}{=} f_1\, c \overset{\text{def}}{=} fn$$

where $(*)$ follows because we know that $a \# f_1$ and $b \# f_1$ and therefore by (11) 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 permutation type). So fn must have finite support. □

Example 2. Given the function $fn' \overset{\text{def}}{=} \lambda\pi.\ f_2\,(r_1\,\pi)\,(r_2\,\pi)$ where we assume that the free variables of fn', namely f_2, r_1 and r_2, are functions with finite support. In order to verify that fn' has finite support we need to verify $(f_1, r_1, r_2)\ supports\ fn'$, that is decide the following equation

$$(a\, b)\bullet(\lambda\pi.\ f_2\,(r_1\,\pi)\,(r_2\,\pi)) = \lambda\pi.\ f_2\,(r_1\,\pi)\,(r_2\,\pi)$$

under the assumptions that $a \# (f_2, r_1, r_2)$ and $b \# (f_2, r_1, r_2)$. Pushing the swapping $(a\, b)$ under the λ and inside the applications using (5) and (7), the swapping will by (11) "vanish" in front of f_1, r_1 and r_2, and we have two identical terms. So fn' has finite support under the given assumptions. □

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 main point is that the decision procedure involving permutations can be relatively easily automated in a special purpose tactic analysing permutations. This seems much more convenient than analysing the support of a function directly.

3 Recursion for the Lambda-Calculus

In this section we derive from an inductively defined relation the existence of a recursion combinator that allows us to define functions over the structure of the type `lam`. This way of introducing a recursion combinator is standard in HOL-based theorem provers.

In contrast with the usual datatypes, such as lists and products, where the term-constructors are always injective, the term-constructors of nominal datatypes are because of the binders in general *not* injective, see equation (2). That means when stating a function definition by characteristic equations like the ones given for capture-avoiding substitution in (3), it is not obvious whether the intended function, roughly speaking, preserves alpha-equivalence—we have seen the counter-examples *bn* and *ist* in the Introduction. Pitts [8,9] stated some general conditions for when functions do preserve alpha-equivalence.

A definition by structural recursion involves in case of the lambda-calculus 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:

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

with α being a permutation type. Then the first condition by Pitts states that f_3—the function for the lambda case—needs to satisfy the following property:[4]

Definition 6 (Freshness Condition for Binders (FCB)). *A function f with type* $\texttt{name} \Rightarrow \texttt{lam} \Rightarrow \alpha \Rightarrow \alpha$ *satisfies the* FCB *provided* $\exists\, a.\ a \# f \wedge \forall\, t\, r.\ a \# f\, a\, t\, r$.

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 have finite support. This condition ensures that we can use Prop. 1 to chose a fresh name.

With these two conditions we can define a recursion combinator, we call it $rfun_{f_1 f_2 f_3}$, with the following properties:

Theorem 1 (Characteristic Equations for Recursion). *If f_1, f_2 and f_3 have finite support and f_3 satisfies the FCB, then:*

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

[4] We slightly adapted the definition of Pitts to apply to our recursion combinator.

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

$$\frac{}{(\texttt{Var}\, a, f_1\, a) \in rec_{f_1 f_2 f_3}}$$

$$\frac{(t_1, r_1) \in rec_{f_1 f_2 f_3} \quad (t_2, r_2) \in rec_{f_1 f_2 f_3}}{(\texttt{App}\, t_1\, t_2, f_2\, t_1\, t_2\, r_1\, r_2) \in rec_{f_1 f_2 f_3}} \tag{12}$$

$$\frac{a \# (f_1, f_2, f_3) \quad (t, r) \in rec_{f_1 f_2 f_3}}{(\texttt{Lam}\, a\, t, f_3\, a\, t\, r) \in rec_{f_1 f_2 f_3}}$$

With this inductive definition comes the following induction principle:

$$\frac{\begin{array}{l}\forall a.\, P\,(\texttt{Var}\, a)\,(f_1\, a)\\ \forall t_1\, t_2\, r_1\, r_2.\, P\, t_1\, r_1 \;\wedge\; P\, t_2\, r_2 \Rightarrow P\,(\texttt{App}\, t_1\, t_2)\,(f_2\, t_1\, t_2\, r_1\, r_2)\\ \forall a\, t\, r.\, a \# (f_1, f_2, f_3) \;\wedge\; P\, t\, r \Rightarrow P\,(\texttt{Lam}\, a\, t)\end{array}}{(t, r) \in rec_{f_1 f_2 f_3} \Rightarrow P\, t\, r} \tag{13}$$

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 obtain a function from `lam` to α.

We first show that there exists an r for every t. For this we use the following strong structural induction principle [9,10,11] that the nominal datatype package generates for the type `lam`:

$$\frac{\begin{array}{l} \mathit{finite}(S)\\ \forall a.\, P\,(\texttt{Var}\, a)\\ \forall t_1 t_2.\, P\, t_1 \wedge P\, t_2 \Rightarrow P\,(\texttt{App}\, t_1\, t_2)\\ \forall a\, t.\, a \notin S \Rightarrow P\, t \Rightarrow P\,(\texttt{Lam}\, a\, t)\end{array}}{P\, t} \tag{14}$$

This induction principle is called *strong*, because in the lambda-case one does not need to establish the property P for all binders a, but only for binders that are not in the finite set S. With this structural induction principle the proof of the next lemma is routine.

Lemma 4 (Totality). *Provided f_1, f_2 and f_3 have finite support, then for all t there exists an r such that $(t, r) \in rec_{f_1 f_2 f_3}$.*

Proof. By the strong induction principle, where we take S to be $\mathit{supp}(f_1, f_2, f_3)$, which we know by assumption is finite. Then in the lambda-case we can assume that $a \notin \mathit{supp}(f_1, f_2, f_3)$ holds, which is defined to be $a \# (f_1, f_2, f_3)$. All cases are then routine applying the rules in (12).

Next we establish that all r in the relation $rec_{f_1 f_2 f_3}$ have finite support.

Lemma 5 (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 the induction principle give in (13). 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 similar. $\square$

In order to establish the "uniqueness" part of Theorem 1, we need the following two lemmas establishing that $rec_{f_1 f_2 f_3}$ is *equivariant* (see [7]) and that it preserves freshness.

Lemma 6 (Equivariance). *If* $(t, r) \in rec_{f_1 f_2 f_3}$ *then for all* π *also* $(\pi{\bullet}t, \pi{\bullet}r) \in rec_{(\pi\bullet f_1)(\pi\bullet f_2)(\pi\bullet f_3)}$.

Proof. By the induction principle given in (13). All cases are routine by pushing the permutation π into t and r, except in the lambda-case where we have to apply (10) in order to infer $(\pi{\bullet}a) \# (\pi{\bullet}(f_1, f_2, f_3))$ from $a \# (f_1, f_2, f_3)$. $\square$

Lemma 7 (Freshness). *If* f_1*,* f_2 *and* f_3 *have finite support and* f_3 *satisfies the FCB, then assuming* $(t, r) \in rec_{f_1 f_2 f_3}$ *and* $a \# (f_1, f_2, f_3, t)$ *implies* $a \# r$.

Proof. By the induction principle given in (13); non-routine is the lambda-case. In this case, say with the instantiations $(\texttt{Lam}\, a'\, t)$, we have that $a' \# (f_1, f_2, f_3)$. We further have that $a \# (f_1, f_2, f_3, \texttt{Lam}\, a'\, t)$ and have to show that $a \# f_3\, a'\, t\, r$. In case that $a = a'$, we know from the FCB, there exists an a'' such that $a'' \# f_3$ and $\forall\, t\, r.\, a'' \# f_3\, a''\, t\, r$. Using (10) we apply the swapping $(a\, a'')$ to both sides of our goal which gives $a'' \# ((a\, a''){\bullet}f_3)\, a''\, ((a\, a''){\bullet}t)\, ((a\, a''){\bullet}r)$. Since $a \# f_3$ and $a'' \# f_3$ we have by (11) that $(a\, a''){\bullet}f_3 = f_3$ and hence we are done. In case $a \neq a'$ we can infer from $a \# (f_1, f_2, f_3, \texttt{Lam}\, a'\, t)$ that $a \# (f_1, f_2, f_3, t)$ holds and thus apply the induction hypothesis. $\square$

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

Lemma 8 (Uniqueness). *If* f_1*,* f_2 *and* f_3 *have finite support and* f_3 *satisfies the FCB, then* $(t, r) \in rec_{f_1 f_2 f_3}$ *and* $(t, r') \in rec_{f_1 f_2 f_3}$ *implies that* $r = r'$.

Proof. By the induction principle given in (13); again the only non-routine case is the lambda-case. By assumption we know that $(\texttt{Lam}\, a\, t, f_3\, a\, t\, r) \in rec_{f_1 f_2 f_3}$ from which we can infer that $a \# (f_1, f_2, f_3)$ and $(t, r) \in rec_{f_1 f_2 f_3}$; the induction hypothesis states that for all r', $(t, r') \in rec_{f_1 f_2 f_3}$ implies $r = r'$. Using the second assumption $(\texttt{Lam}\, b\, t', r') \in rec_{f_1 f_2 f_3}$ we need to show that $f_3\, a\, t\, r = f_3\, b\, t'\, r'$ holds for all $\texttt{Lam}\, b\, t'$ such that $b \# (f_1, f_2, f_3)$ and $\texttt{Lam}\, a\, t = \texttt{Lam}\, b\, t'$. The latter implies by (9) that either

$$(a = b \wedge t = t') \text{ or } (a \neq b \wedge t = (a\, b){\bullet}t' \wedge a \# t')\ .$$

The first case is routine because by the induction hypothesis we can infer that $r = r'$. In the second case we have $((a\, b){\bullet}t, r') \in rec_{f_1 f_2 f_3}$ and by Lem. 6 also that $(t, (a\, b){\bullet}r') \in rec_{f_1 f_2 f_3}$ (where we also use (11) and the facts $a \# (f_1, f_2, f_3)$ and $b \# (f_1, f_2, f_3)$). By induction hypothesis we can therefore infer that $r = (a\, b){\bullet}r'$. Hence we have to show that $f_3\, a\, ((a\, b){\bullet}t')\, ((a\, b){\bullet}r') = f_3\, b\, t'\, r'$ holds.

Since we know that $a \# t'$ and $a \# (f_1, f_2, f_3)$, we can use $(t', r') \in rec_{f_1 f_2 f_3}$ and Lem. 7 to show that $a \# r'$ holds. With this and the facts that $a \neq b$, $a \# t'$ and $a \# f_3$, we can infer that $a \# (f_3\, b\, t'\, r')$ (the latter is because (f_3, b, t', r') *supports* $(f_3\, b\, t'\, r')$ and therefore $supp(f_3\, b\, t'\, r') \subseteq (f_3, b, t', r')$).

We now show that also $b \# (f_3\, b\, t'\, r')$. From the FCB we know that there exists a b' such that $b' \# f_3$ and $\forall\, t\, r.\, b' \# f_3\, b'\, t\, r$ holds. If $b = b'$ we are done; otherwise we use (10) and apply the swapping $(b\, b')$ to both sides of $b \# (f_3\, b\, t'\, r')$ which gives $b' \# ((b\, b')\bullet f_3)\, b'\, ((b\, b')\bullet t')\, ((b\, b')\bullet r')$. Since $b \# f_3$ and $b' \# f_3$ we have by (11) that $(b\, b')\bullet f_3 = f_3$ and hence we are done.

Knowing that $a \# (f_3\, b\, t'\, r')$ and $b \# (f_3\, b\, t'\, r')$ hold, we can infer by (11) that $(a\, b)\bullet f_3\, b\, t'\, r' = f_3\, b\, t'\, r'$. The left-hand side of this equation is equal to $f_3\, a\, ((a\, b)\bullet t')\, ((a\, b)\bullet r')$ which is what we had to show. □

To prove our theorem about structural recursion we define $rfun_{f_1 f_2 f_3}\, t$ to be the unique r so that $(t, r) \in rec_{f_1 f_2 f_3}$. This is a standard construction in HOL-based theorem provers. The characteristic equations for $rfun_{f_1 f_2 f_3}$ are given by how the relation $rec_{f_1 f_2 f_3}$ is defined.

4 Examples

We are now going to give examples defining three functions by recursion over the structure of the nominal datatype `lam`. We use the functions:

$$\begin{aligned}
\mathtt{sz}_1 &= \lambda\, a.\, 1\\
\mathtt{sz}_2 &= \lambda\, r_1\, r_2.\, 1 + (\mathbf{max}\, r_1\, r_2)\\
\mathtt{sz}_3 &= \lambda\, a\, r.\, 1 + r
\end{aligned}$$

$$\begin{aligned}
\mathtt{frees}_1 &= \lambda\, a.\, \{a\}\\
\mathtt{frees}_2 &= \lambda\, \_\, \_\, r_1\, r_2.\, r_1 \cup r_2\\
\mathtt{frees}_3 &= \lambda\, a\, \_\, r.\, r - \{a\}
\end{aligned}$$

$$\begin{aligned}
\mathtt{subst}_1\, b\, t' &= \lambda a.\, \mathbf{if}\ a = b\ \mathbf{then}\ t'\ \mathbf{else}\ (\mathtt{Var}\, a)\\
\mathtt{subst}_2\, b\, t' &= \lambda \_\, \_\, r_1\, r_2.\, \mathtt{App}\, r_1\, r_2\\
\mathtt{subst}_3\, b\, t' &= \lambda a\, \_\, r.\, \mathtt{Lam}\, a\, r
\end{aligned}$$

To verify the precondition for the function `sz` we need to define $\pi\bullet n = n$ as the permutation action over natural numbers. This definition implies that `nat` is a permutation type; this also implies that the support of $\mathtt{sz}_1$, $\mathtt{sz}_2$ and $\mathtt{sz}_3$ is the empty set. Next we need to show that the FCB-condition, namely $\exists a.\, a \# \mathtt{sz}_3 \wedge \forall t'\, r.\, a \# \mathtt{sz}_3\, a\, r$, holds. For this we can chose any atom a, because $\mathtt{sz}_3$ has empty support and $\mathtt{sz}_3\, a\, r$ is a natural number and so has also empty support.

In order to define the function for the set of free names of a lambda-term in the nominal datatype package, we need to restrict the co-domain of `frees` to finite sets. This is because finite sets, as opposed to arbitrary sets, have much better properties w.r.t. the notion of support. In addition finite sets of names are permutation types. We can verify that $\mathtt{frees}_n$ for $n = 1, 2, 3$ has empty support using our heuristic and the fact that the HOL-functions $\lambda x\, y.\, x \cup y$ and $\lambda x\, y.\, x - y$ have empty support. To verify the FCB-condition, namely $\exists a.\, a \# \texttt{frees}_3 \wedge \forall t'\, r.\, a \# \texttt{frees}_3\, a\, r$, holds. For this we can chose any atom a, because $\texttt{frees}_3$ has empty support; next we have to verify that $\forall r.\, a \# r - \{a\}$ holds, or equivalently $\forall r.\, a \notin \mathit{supp}(r - \{a\})$. Since we restricted the co-domain of $\texttt{frees}$ to finite sets, we know that $r - \{a\}$ is finite for all r and further that $\mathit{supp}(r - \{a\}) = r - \{a\}$. Thus we are done.

For the substitution function we find that $\mathit{supp}(b, t')$ *supports* $\texttt{subst}_n\, b\, t'$ for $n = 1, 2, 3$. The set $\mathit{supp}(b, t')$ is finite, because $\texttt{name}$ and $\texttt{lam}$ are finitely supported permutation types. The FCB-condition of $\texttt{subst}_3$ holds for all atoms c with $c \# (b, t')$. Because $\mathit{supp}(b, t')$ *supports* $\texttt{subst}_n\, b\, t'$, the preconditions of the recursion-combinator in the lambda-case simplify to $a \# (b, t')$ and thus we obtain the characteristic equation

$$\texttt{subst}\, b\, t'\, (\texttt{Lam}\, a\, t) = \texttt{Lam}\, a\, (\texttt{subst}\, b\, t'\, t)$$

with the side-conditions $a \neq b$ and $a \# t'$, as expected.

The "functions" *bn* and *ist* from the Introduction do not satisfy the FCB. In case of *bn* it is never true that $a \# r \cup \{a\}$, and in case of *ist* there does not exists an a such that for all t we have that $a \# \{t\}$ holds—it will fail for example for $t = \texttt{Var}\, a$.

5 General Case

The nominal datatype package supports the declaration of more than one atom type and allows term-constructors to have more than one binder. The notions of support and freshness (see Def. 3) have in the implementation already polymorphic type to take several atom types into account. For the recursion combinator we have to make sure that the function f_i of the characteristic equations have finite support with respect to every atom type that occurs in binding position. By binding position we mean the types occurring inside the «...» that are used in a nominal datatype declaration. For example, given the term-constructor C with the type declaration

$$C\ \texttt{"«atm}_1\texttt{»} \ldots \texttt{«atm}_n\texttt{»}\ \alpha\texttt{"}$$

then we have to consider all atom types $\texttt{atm}_1 \ldots \texttt{atm}_n$.

Similarly the FCB needs to be generalised for all atom types that occur in binding position. To explain the generalisations let us consider first the term-constructor `Let "«name» lam" "lam"`. The type indicates that if we write, say $\texttt{Let}\, a\, t_1\, t_2$, then the scope of the binder a is t_1. Hence the characteristic equation for `Let` is

$$\begin{aligned}\mathit{rfun}_{f_1\, f_2\, f_3\, f_4}\, (\texttt{Let}\, a\, t_1\, t_2) = f_4\, a\, t_1\, t_2\, (\mathit{rfun}_{f_1\, f_2\, f_3\, f_4}\, t_1)\, (\mathit{rfun}_{f_1\, f_2\, f_3\, f_4}\, t_2) \\ \textit{provided}\ a \# (f_1, f_2, f_3, f_4, t_2)\end{aligned}$$

As can be seen, the binder a needs to be fresh for f_1, f_2, f_3 and f_4 (like in the lambda-case), but also for t_2. The general rule is that a needs to be fresh for all terms that are *not* in its scope—in this example, this applies only to t_2. The FCB for `Let` is

$$\exists a.\, a \# f_4 \wedge \forall t_1\, t_2\, r_1\, r_2.\, a \# t_2 \Rightarrow a \# f_4\, a\, t_1\, t_2\, r_1\, r_2$$

where in the second conjunct we may assume that a is fresh for all terms not in its scope.

Albeit not yet supported by the current version of the nominal datatype package, even more interesting is the term-constructor `Letrec "«name»(lam ×` `«name»lam)"` where we have two binders. The characteristic equation for `Letrec` is

$$\begin{aligned} rfun_{f_1\, f_2\, f_3\, f_4\, f_5}\, (\texttt{Letrec}\, a\, t_1\, b\, t_2) = f_5\, a\, t_1\, b\, t_2\, (rfun_{f_1\, ..\, f_5}\, t_1)\, (rfun_{f_1\, ..\, f_5}\, t_2) \\ \textit{provided } a \# (f_1, f_2, f_3, f_4, f_5) \\ \textit{and } b \# (f_1, f_2, f_3, f_4, f_5, t_1) \\ \textit{and } a \neq b \end{aligned}$$

where we need to have $b \# t_1$ since t_1 is not in the scope of the binder b. However, in case we have more than one binder in a term-constructor then we further need to add constraints that make sure every binder is distinct. With these generalisations the proofs we have given in Sec. 3 scale to all nominal datatypes.

6 Conclusion

We presented a structural recursion combinator for nominal datatypes. The details were given for the nominal datatype `lam`; we mentioned briefly the general case—further details are given in [9]. For the presentation we adapted the clever proof given also in [9]. The main difference is that we gave a direct proof for nominal datatypes and did not use auxiliary constructions. There are also a number of other differences: for example Pitts does not need to prove Lem. 5, which is however necessary in Isabelle/HOL, because one cannot conveniently introduce the type of finitely supported functions. In comparison with the formalisation by Norrish, our proof is much shorter—only about 150 lines of readable Isar-code compared to approximately 600 dense lines of HOL4-code. Our use of the heuristic that solves proof obligations to do with finite support made it tractable to automate our proof. The earlier formalisation were far too difficult for such an automation. This work removes the painful obstacle when defining functions over the structure of nominal datatypes using earlier versions of the nominal datatype package. In the future we are aiming at automating the process of verifying the FCB and finite support-conditions required in the recursion combinator.

Acknowledgements: We are very grateful to Andrew Pitts and Michael Norrish for the many discussions with them on the subject of the paper. The first author is supported by a fellowship from the Alexander-von-Humboldt foundation and by a Emmy-Noether fellowship from the German Research Council. The second author received funding via the BMBF project Verisoft.

References

1. H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1981.
2. S. Berghofer and M. Wenzel. Inductive Datatypes in HOL - Lessons Learned in Formal-Logic Engineering. In *Proc. of the 12th International Conference Theorem Proving in Higher Order Logics (TPHOLs)*, number 1690 in LNCS, pages 19–36, 1999.
3. M. J. Gabbay and A. M. Pitts. A New Approach to Abstract Syntax Involving Binders. In *Logic in Computer Science*, pages 214–224. IEEE Computer Society Press, 1999.
4. M. Gordon. From LCF to HOL: a short history. In G. Plotkin, C. P. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction*, pages 169–186. MIT Press, 2000.
5. P. Homeier. A Design Structure for Higher Order Quotients. In *Proc. of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 3603 of *LNCS*, pages 130–146, 2005.
6. T. Melham. Automating Recursive Type Definitions in Higher Order Logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386. Springer-Verlag, 1989.
7. A. M. Pitts. Nominal Logic, A First Order Theory of Names and Binding. *Information and Computation*, 186:165–193, 2003.
8. A. M. Pitts. Alpha-Structural Recursion and Induction (Extended Abstract). In *Proc. of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 3603 of *LNCS*, pages 17–34, 2005.
9. A. M. Pitts. Alpha-Structural Recursion and Induction. *Journal of the ACM*, 200X. to appear.
10. C. Urban and M. Norrish. A Formal Treatment of the Barendregt Variable Convention in Rule Inductions. In *Proc. of the 3rd International ACM Workshop on Mechanized Reasoning about Languages with Variable Binding and Names*, pages 25–32, 2005.
11. C. Urban and C. Tasson. Nominal Techniques in Isabelle/HOL. In *Proc. of the 20th International Conference on Automated Deduction (CADE)*, volume 3632 of *LNCS*, pages 38–53, 2005.
12. M. Wenzel. *Using Axiomatic Type Classes in Isabelle.* Manual in the Isabelle distribution.