Reasoning about Constants in Nominal Isabelle or How to Formalize the Second Fixed Point Theorem

Cezary Kaliszyk[1] and Henk Barendregt[2]

[1] University of Tsukuba
[2] Radboud University Nijmegen

Abstract. Nominal Isabelle is a framework for reasoning about programming languages with named bound variables (as opposed to de Bruijn indices). It is a definitional extension of the HOL object logic of the Isabelle theorem prover. Nominal Isabelle supports the definition of term languages of calculi with bindings, functions on the terms of these calculi and provides mechanisms that automatically rename binders. Functions defined in Nominal Isabelle can be defined with assumptions: The binders can be assumed fresh for any arguments of the functions. Defining functions is often one of the more complicated part of reasoning with Nominal Isabelle, and together with analysing freshness is the part that differs most from paper proofs. In this paper we show how to define terms from λ-calculus and reason about them without having to carry around the freshness conditions. As a case study we formalize the second fixed point theorem of the λ-calculus.

1 Introduction

Proofs about abstract calculi are often complicated and tedious; which is why they are often done only semi-formally. Nominal Isabelle [13], has been designed to make this kind of proofs easy to formalize. It provides an infrastructure for defining a term language of a calculus (nominal datatypes) together with a reasoning infrastructure about those datatypes. This means a number of lemmas that describe properties of the introduced type, including induction principles that already have the variable convention built in. The framework also provides a number of mechanisms for defining terms and functions over those datatypes. Nominal Isabelle has been used for formalizing proofs about calculi including LF [15], π-calculus [3], and ψ-calculus [2]. Here we will look at formalizing definitions and proofs about terms in the λ-calculus, like pairs, finite sequences, or initial functions [1].

Defining such constants and simple functions can be done in numerous ways. Nominal Isabelle provides *nominal_primrec*, a mechanism for defining functions with primitive recursion [14]. Functions defined with its help can be defined with assumptions: the binders can be assumed fresh for any arguments of the functions. The only proof obligations necessary to fulfill are the FCBs (*fresh*

J.-P. Jouannaud and Z. Shao (Eds.): CPP 2011, LNCS 7086, pp. 87–102, 2011.

condition for binder). This is convenient for functions that analyze term structure, but it does not allow for functions that invent new variables on the right hand side (this is the case for defining constants with λ-abstractions). It is also limited to primitive recursion, but this is not an issue since termination for all the functions that we define here is straightforward.

A different way of defining nominal constants and functions is provided with the *fresh_fun* binder. This construction allows introducing a new variable, that is fresh for the term under the binder. This allows defining more complicated nominal functions, for some CPS translations (the non higher-order ones). Unfortunately reasoning about *fresh_fun* involves reasoning about the term and the freshness obligations *together*, which means that renamings are performed manually. This makes the proofs complicated and very hard to read, especially in the case of iterated applications of *fresh_fun*. In the case of the proofs about the CPS translation, performing them without the help of the Nominal package [9] was less involved than reasoning with *fresh_fun*.

It is also possible to define constants using the Isabelle/HOL function package directly. Given a non-injective datatype, the function package returns a completeness obligation and compatibility obligations. With this approach it is in principle possible to define mutually recursive functions, non-primitive-recursive functions, or even functions on datatypes which abstract multiple binders. In practice fulfilling the proof obligations about the defined function that the function package requires is not feasible with the infrastructure given by the function package (it does not even provide regular induction).

The current implementation of Nominal Isabelle uses quotients [8] to lift the reasoning infrastructure from a raw datatype to a nominal type modulo α-equivalence. One could define constants or functions on the raw datatype and use the same lifting process to obtain a lifted function. This requires showing respectfulness of the raw function, which is true and it is provable. Unfortunately on the raw level we cannot use any of the nominal techniques, and proving such goals does require rederiving the reasoning infrastructure for the raw terms.

This makes defining simple functions or even constants one the more complicated parts of reasoning with Nominal Isabelle. Given that such a function or constant is defined, it normally has freshness obligations for the binders and new variables. When proving properties that involve such obligations with the help of the Nominal package, there is a tendency to leave those freshness obligations to the simplifier. When this does not work, first the user tries to rewrite with a conditional rewrite rule to get from Isabelle the remaining freshness proof obligation. The proof is then changed in such a way that the assumption is available in the goalstate, and the proof can be fully automated with a call to the simplifier. The proof obtained in this way can look similar to a textbook one (with the exception of the step where one specifies the constants to avoid), however this approach has one substantial drawback: the final proof does not reflect the structure in which the proof was written and does not show the reasoning necessary for fulfilling the freshness obligations.

In this paper we look at defining constants. We take an approach different from the ones presented above. Starting with a function that returns different variables, we define constants using those variables and show their convertibility to the forms with freshness obligations. We then look at applied versions of the constants to derive convertibility relations that have no freshness obligations and only add those to the simplifier. This allows reasoning about convertible terms without any freshness assumptions and the formalized proofs resemble textbook proofs. We formalize the second fixed point theorem of the λ-calculus and obtain a formalized proof that resembles a pen and paper one.[1]

The rest of the paper is organized as follows: In Section 2 we introduce the notions from Nominal Logic necessary in later part of the paper. In Section 3 we the introduce the term language and the conversion relation of the λ-calculus, and compare it with classical textbook definition. In Section 4 we show our infrastructure used for defining constants with Nominal Isabelle. Our case-study, the formalization of the second fixed point theorems follows in next two Sections: We show the definitions and the properties of the basic constants in Section 5 and the proof in Section 6. We conclude in Section 7.

2 Nominal Logic Preliminaries

In this section we introduce the core notions of nominal logic, that will be used in the remaining part of the paper. Nominal logic has been introduced by Pitts [12] and has been adapted to the Isabelle/HOL formal setting by Urban. The details of the current version of the adaptation including proofs can be found in [7]. Here we shortly introduce only the main notions that will be used further.

The central notions of nominal logic are sorted atoms and sort-respecting permutations. Atoms are supposed to represent variables (both bound and free ones). Sorts can be used, if a calculus has different kinds of variables, such as names and identifiers in LF. Each atom type has an infinite supply of atoms. Further in this paper we will only talk about one kind of atoms; namely the variables of the classical λ-calculus, thus and we will omit the function *atom* which projects concrete atom types to the type of all atoms.

A permutation is a function that swaps a finite number of atoms. We denote permutations with π and an application of a permutation to an object as $\pi \bullet M$. Permutations applied to an object change only the atoms of the object. The permutation application operation, takes a permutation and an object of the general type-class of objects for which permutations are defined. The identity permutation is written as *0* and the composition of two permutations π_1 and π_2 as $\pi_1 + \pi_2$. We denote the inverse permutation of π as $-\pi$. Application of permutations naturally extends to products, lists, functions and datatypes (nominal datatypes) as described in [16].

The smallest non-trivial permutation is called a swapping. We denote a swapping of atoms a and b as $(a\ b)$, and define by:

[1] The source of the formalization is included in the new version of Nominal Isabelle, available from the repository linked at http://isabelle.in.tum.de/nominal/.

$$(a\ b) = \lambda c.\ \textit{if}\ a = c\ \textit{then}\ b\ \textit{else}\ \textit{if}\ b = c\ \textit{then}\ a\ \textit{else}\ c$$

The most important notion of the nominal logic work is a general definition for the "free variables of an object", called *support*. We will denote it *supp x*. This notion applies not only to terms defined in Nominal Logic, but also to lists, pairs, sets or general functions. For the definition to be applicable, the type only needs to have the permutation operation and equality defined:

$$\textit{supp}\ x = \{a.\ \textit{infinite}\ \{b.\ (a\ b) \bullet x \neq x\}\}$$

From the notion of support, the notion of atoms being fresh for an object is derived. Atom a is *fresh* for an object x:

$$a \# x = a \notin \textit{supp}\ x$$

Nominal logic also defines *equivariance*. An object x is equivariant if applying any permutation to the object leaves it unchanged.

$$\forall \pi \bullet x = x$$

All objects that have no atoms (like natural numbers, booleans, ...) are equivariant. Equivariance is especially important for functions, as this is equivalent to the fact that a permutation applied to the application of the function to its arguments *f a1 a2 .. an* is equal to the application of the permutation to all the arguments. This means that the application of any permutations commutes with the application of the function:

$$\forall \pi.\ \pi \bullet (f\ a_1\ a_2\ \ldots\ a_n) = f\ (\pi \bullet a_1)\ (\pi \bullet a_2)\ \ldots\ (\pi \bullet a_n)$$

Finally, if an object is equivariant, any swappings leave the object unchanged so no atoms are in its support. We will use this property often, to show that a term is a closed term.

3 Lambda Terms and Conversion

The definition of the terms of λ-calculus, as done in textbooks, starts with an alphabet of variables, the λ-abstractor and parentheses and continues with an inductive definition of the set of terms. Below we show the definition according to the second author's book [1].

Definition 1. *(i)* Lambda terms *are words over the following alphabet:*

$v_0, v_1, \ldots$ *variables,*
λ *abstractor,*
$(\,,)$ *parentheses.*

(ii) The set of λ-terms Λ is defined inductively as follows:

1. $x \in \Lambda$;
2. $M \in \Lambda \;\Rightarrow\; (\lambda x M) \in \Lambda$;
3. $M, N \in \Lambda \;\Rightarrow\; (MN) \in \Lambda$;

where x in 1 or 2 is an arbitrary variable.

A textbook would next add notations that allow writing λ-terms without the brackets that can be unambiguously removed. To make a similar definition in Isabelle one needs to start with defining the set of variables. This is performed with:

atom-decl var

which introduces a new sort *var* that will be used to name the variables. The command from Nominal Isabelle introduces a new set of variables that is fresh for all existing sets and is shown to be infinite. The fact that it is infinite will be necessary to rename variables.

Defining an inductive type in Isabelle requires giving its cases together with the notations. Additionally Nominal Isabelle lets us specify bindings, here we specify that the abstracted variable is bound in the lambda case. This instructs Nominal Isabelle to derive a type where variables in the λ-abstractions can be renamed:

```
nominal_datatype lam =
  v̄
 | lam · lam
 | λx. lam   bind x in lam
```

In Isabelle, constructors of inductive types embed particular objects of the type into the type itself. This is why we need a constructor that embeds variables into λ-terms. In the above definition the $\overline{v}$ means a lambda terms that is a variable, as opposed to v which represents a variable itself. Application is denoted with an infix left-associative notation, and λ-abstraction is annotated with a binding. With notations set this way, terms can be written in a way that resembles the textbook notation, but are still an inductive type.

Next, we would like to define convertibility of λ-terms. A formal definition of substitution would often be omitted in textbooks, but Isabelle requires a formal definition of substitution to define β-conversion. Nominal Isabelle allows defining functions where in the clauses the bound variable can be fresh for any other terms. We show this in the standard definition of substitution from Nominal Isabelle:

Definition 2 (Substitution). *Substituting a variable y for a term S in term M is defined by:*

$$T\ [y := S] = \begin{cases} \textit{if } x = y \textit{ then } S \textit{ else } \overline{x} & \textit{if } T = \overline{x} \\ (T_1\ [y := S]) \cdot (T_2\ [y := S]) & \textit{if } T = T_1 \cdot T_2 \\ \lambda x.\ U\ [y := S] & \textit{if } T = \lambda x.\ U \textit{ and } x \ \# \ (y,\, S) \end{cases}$$

A regular definition of substitution would require renaming the variable in the lambda case. Nominal Isabelle extends the definition given only for the case where the binder is fresh for the given arguments to a function on the whole domain. Next we show a number of simple properties of substitution, that we want to use to reason about terms involving substitution and freshness later:

Lemma 1. *Substitution is equivariant, substituting a fresh variable does not change the term and substituting a variable for itself does not change the term.*

$$\begin{gathered}\pi \bullet a\ [aa := ab] = (\pi \bullet a)\ [(\pi \bullet aa) := (\pi \bullet ab)] \\ x \# t \Longrightarrow t\ [x := s] = t \qquad M\ [x := \overline{x}] = M\end{gathered} \tag{1}$$

Proof. By induction, follows by definitions of substitution and freshness.

Closed terms have an empty support, so any variable will be fresh for one. This implies that substitution does not change closed terms:

$$\mathit{supp}\ t = \varnothing \Longrightarrow t\ [x := s] = t$$

Having defined substitution we can return to the definition of convertibility. Again we try to closely follow [1].

Definition 3 (Convertibility). *Convertibility is an inductively defined relation axiomatized by the following:*

$$\begin{aligned}&(\lambda x.\ M) \cdot N \approx M\ [x := N] \\ &M \approx M \\ &M \approx N \Longrightarrow N \approx M \\ &M \approx N \Longrightarrow N \approx L \Longrightarrow M \approx L \\ &M \approx N \Longrightarrow Z \cdot M \approx Z \cdot N \\ &M \approx N \Longrightarrow M \cdot Z \approx N \cdot Z \\ &M \approx N \Longrightarrow (\lambda x.\ M) \approx (\lambda x.\ N)\end{aligned} \tag{2}$$

In this paper, we will use two equivalence relations on terms, namely the Isabelle equality $=$ and the convertibility relation $\approx$ defined above. This is slightly different from the regular textbook approach where convertibility in introduced in addition to syntactic equality. Nominal Isabelle has defined the type of lambda terms in such a way that equality on this type includes α-equivalence, hence it is different from syntactic equality. Nominal Isabelle can automatically derive equivariance for convertibility. Since many of our proofs will talk about chains of convertibilities, we declare convertibility as a transitive relation. This will allow writing $M \approx ... \approx N$ in the formal proofs later.

4 Defining λ-Constants with Nominal Isabelle

In order to define all the constants from the next Section, we will need two mechanisms. First a set of variables that will be fresh in the definitions and second a convenient way to show that concepts defined with those fixed variables can be renamed to any fresh variables and therefore applied without freshness obligations.

Nominal Isabelle ensures that there is an infinite amount of atoms in any of the concrete atom sorts. This means that for any finite set of variables we can find a new one, fresh for this set.

Using this property we can define a function $\nu :: \mathbb{N} \Rightarrow \mathit{var}$ which given a number n returns a variable v that is fresh for the set of the results of the function for all smaller numbers:

$$\nu n \sharp \bigcup_{k=0..(n-1)} \{\nu k\} \tag{3}$$

A direct consequence is, that for any two different natural numbers m and n:

$$\nu m \neq \nu n$$

Given a constant defined with the help of a fixed set of νk, we would like to rename the variables to any variables that satisfy the property 3. This can be done by analyzing the equalities between the variable pairs and applying the definition of swapping atoms together with equivariance. With the help of the Sledgehammer SMT linkup [4] this can be performed fully automatically.

Given a definition that depends on freshness obligations, we cannot apply this definition without an additional step in which we obtain the new variables specifying what are they supposed to be fresh for. This is however not going to be the case for constant terms representing functions in the λ-calculus. Since they need to have a λ-abstraction at the top-level, after the abstraction is applied, the newly invented variable can be forgotten. A simplest possible example is:

$$(\lambda x.\ M) \cdot \overline{x} \approx M \tag{4}$$

We can extend this to arbitrary functions, defined by equations that start with a λ-abstraction:

Lemma 2. *Assuming that function L is defined by the equation*

$$\forall\, a.\ (L = (\lambda a.\ F\ (\overline{a})))$$

and we can substitute variables in F

$$\forall\, x.\ (x \# A \Longrightarrow F\ (\overline{x})\ [x := A] = F\ A)$$

then

$$L \cdot A \approx F\ A$$

Proof. Obtaining a fresh variable a, such that $a \# A$, using the definition of convertibility (2) and properties of substitution (1) the proof is a simple calculation.

The same mechanism works for functions that require more new variables. The additional assumption that the function makes, is that the variables are distinct, which follows from 3:

Lemma 3. *Assuming that function L is defined by the equation*

$$\forall\, a\ b.\ (a \neq b \Longrightarrow L = (\lambda a.\ \lambda b.\ F\ (\overline{a})\ (\overline{b})))$$

and we can substitute variables in F

$$\begin{aligned}&\forall\, x\ y.\ (x \# (A,\ B) \wedge y \# (A,\ B,\ x) \Longrightarrow\\ &\qquad \Longrightarrow F\ (\overline{x})\ (\overline{y})\ [x := A]\ [y := B]\ =\ F\ A\ B)\end{aligned}$$

then

$$L \cdot A \cdot B \approx F\ A\ B$$

Proof. Obtaining fresh variables a, such that $a \# (A,\ B)$, and b such that $b \# (A,\ B,\ a)$, using freshness for tuples, the definition of convertibility (2) and properties of substitution (1) the proof is a simple calculation.

5 Basic Constants in λ-Calculus

In this Section we start a presentation of a case-study: formalization of the second fixed point theorem of the λ-calculus. The proof relies on an internal coding of lambda terms as normal lambda terms

$$M \longmapsto \ulcorner M \urcorner.$$

We take the coding used in [5] for its elegance, enabling a short proof, but other codings could have been used as well.

We start with the definition of the initial functions U_n^m. We slightly modify the usual textbook definition, by using a different indexing, starting from 0. We define it for all natural numbers, but intend to use it for $m \leq n$.

Definition 4 (initial functions). *For any natural numbers m and n:*

$$U_n^m = \begin{cases} \lambda\nu 0.\ \overline{\nu n} & \textit{provided } m = 0,\\ \lambda\nu m.\ U_n^k & \textit{provided } m = k+1.\end{cases}$$

The terms U_n^m are equivariant and have empty support in the intended domain.

Lemma 4. *Assuming $m \leq n$:*

$$\textit{supp}\ (U_n^m) = \varnothing \qquad\qquad \pi \bullet U_n^m = U_n^m$$

Proof. By induction on m.

When reasoning about the second fixed point theorems, we are going to use only three cases of U_n^m, for $m = 2$ and $n \leq 2$. By expanding the definition for these three cases we can see that it indeed selects the desired elements. We rename the variables to arbitrary but distinct x, y and z.

Assuming $x \neq y$, $y \neq z$ and $z \neq x$:

$$\begin{aligned} U^2_0 &= \lambda x.\ \lambda y.\ \lambda z.\ \overline{z} \\ U^2_1 &= \lambda x.\ \lambda y.\ \lambda z.\ \overline{y} \\ U^2_2 &= \lambda x.\ \lambda y.\ \lambda z.\ \overline{x} \end{aligned}$$

Definition 5. *Assuming $x \neq y$, $x \neq e$ and $y \neq e$:*

$$\begin{aligned} Var &= \lambda x.\ \lambda e.\ \overline{e} \cdot U^2_2 \cdot \overline{x} \cdot \overline{e} \\ App &= \lambda x.\ \lambda y.\ \lambda e.\ \overline{e} \cdot U^2_1 \cdot \overline{x} \cdot \overline{y} \cdot \overline{e} \\ Abs &= \lambda x.\ \lambda e.\ \overline{e} \cdot U^2_0 \cdot \overline{x} \cdot \overline{e} \end{aligned}$$

Using the lemma 3 we can find equations that express convertibility of the applications of *Var*, *App*, *Abs*, that will have no freshness obligations:

$$Var \cdot x \cdot e \approx e \cdot U^2_2 \cdot x \cdot e \tag{5}$$

$$App \cdot x \cdot y \cdot e \approx e \cdot U^2_1 \cdot x \cdot y \cdot e \tag{6}$$

$$Abs \cdot x \cdot e \approx e \cdot U^2_0 \cdot x \cdot e \tag{7}$$

Lemma 5. *The terms Var, App, Abs are closed terms and are equivariant:*

$$\begin{array}{lll} supp\ Var = \varnothing & supp\ App = \varnothing & supp\ Abs = \varnothing \\ \pi \bullet\ Var = Var & \pi \bullet\ App = App & \pi \bullet\ Abs = Abs \end{array}$$

Proof. By definitions of the constants, and equivariance rules the terms are equivariant. Hence they have empty support and are closed terms.

We can now define the function that returns the encoding of a given term. The definition proceeds by induction on λ-terms:

Definition 6. *For a given λ-term t, $\ulcorner t \urcorner$ is is defined by:*

$$\ulcorner t \urcorner = \begin{cases} Var \cdot \overline{x} & provided\ t = \overline{x} \\ App \cdot \ulcorner M \urcorner \cdot \ulcorner N \urcorner & provided\ t = M \cdot N \\ Abs \cdot (\lambda x.\ \ulcorner M \urcorner) & provided\ t = (\lambda x.\ M) \end{cases}$$

Lemma 6. *The encoding function $\ulcorner \ldots \urcorner$ is equivariant. It preserves support and freshness:*

$$\pi \bullet \ulcorner a \urcorner = \ulcorner (\pi \bullet a) \urcorner \qquad supp\ \ulcorner x \urcorner = supp\ x \qquad a \# \ulcorner x \urcorner = a \# x$$

Proof. By induction on x, using the equivariance and freshness lemmas.

The next step in the proof will define finite sequences (or tuples). To define it we need a helper definition that given a variable and a list of terms applies all the terms on the list in turn to the variable. We define it by induction on the list l:

Definition 7. *Applying a list of terms l to a variable n is defined by:*

$$\mathit{app\_lst}\ n\ l = \begin{cases} \overline{n} & \text{provided } l \text{ is an empty list} \\ \mathit{app\_lst}\ n\ t \cdot h & \text{provided } l = (h :: t) \end{cases}$$

Lemma 7. *List application is equivariant and preserves support:*

$$\pi \bullet \mathit{app\_lst}\ n\ l = \mathit{app\_lst}\ (\pi \bullet n)\ (\pi \bullet l)$$
$$\mathit{supp}\ (\mathit{app\_lst}\ n\ l) = \{n\} \cup \mathit{supp}\ l$$

Proof. By induction on l, using the equivariance and freshness lemmas.

When defining a finite sequence we can use the nominal function mechanism to write the definition only for the case where the variable is fresh for the list.

Definition 8 (finite sequences). *Given a list of terms l, assuming $x \# l$, we define a finite sequence $\langle\!\langle l \rangle\!\rangle$ by:*

$$\langle\!\langle l \rangle\!\rangle = \lambda x.\ \mathit{app\_lst}\ x\ (\mathit{rev}\ l)$$

Lemma 8. *Finite sequences are equivariant and preserve support. Finite sequences are equal if their underlying lists are equal. Finite sequences commute with substitution:*

$$\pi \bullet \langle\!\langle a \rangle\!\rangle = \langle\!\langle \pi \bullet a \rangle\!\rangle \qquad \mathit{supp}\ \langle\!\langle t \rangle\!\rangle = \mathit{supp}\ t$$
$$\langle\!\langle M \rangle\!\rangle = \langle\!\langle N \rangle\!\rangle \ \text{ if and only if } \ M = N$$
$$\langle\!\langle [M] \rangle\!\rangle\ [x := N] = \langle\!\langle [M\ [x := N]] \rangle\!\rangle$$

Proof. To prove equivariance we obtain a variable fresh for both sides of the equation $y \# (t,\ \pi \bullet t)$, and use the definition with this variable on both sides. To show that equality of finite sequences implies the equality of the underlying term lists we obtain a variable that is fresh for both M and N and use the definition and support equations. To show the support of substitution applied to a sequence, we proceed by induction on the list. For the induction step we use a variable y, fresh for the term, the variable, the substituted term and the result of the substitution: $y \# (M,\ x,\ N,\ M\ [x := N])$. The result follows by a simple computation.

Further in the paper, finite sequences will only refer to lists of one or three elements always written explicitly. Also, we will not use lists for any other purpose. To simplify this notation, we will use the notation $[\ldots]$ in place of $\langle\!\langle [\ldots] \rangle\!\rangle$ to refer to finite sequences from now in the paper.

Since finite sequences preserve support (lemma 8), using the freshness rules for lists we can write out explicitly the freshness conditions for specific finite sequences:

$$x \# [y] = x \# y \qquad x \# [t,\ r,\ s] = x \# (t,\ r,\ s)$$

Applying the lemmas 2 and 3 from the previous section we can get terms convertible to the application of a finite sequence. It is indeed the term applied to the elements of the sequence in order:

$$\begin{gathered}[M] \cdot N \approx N \cdot M \qquad (8)\\ [M,\, N,\, P] \cdot R \approx R \cdot M \cdot N \cdot P\end{gathered}$$

Using the lemmas from Section 4 we can get simpler convertibility rules for finite sequences applied to initial functions. Indeed they return the appropriate elements from the sequences:

$$\begin{aligned}{}[A,\, B,\, C] \cdot U^{2}_{2} &\approx A\\ [A,\, B,\, C] \cdot U^{2}_{1} &\approx B \qquad (9)\\ [A,\, B,\, C] \cdot U^{2}_{0} &\approx C\end{aligned}$$

We can now define the terms F_1, F_2, F_3:

Definition 9. *Assuming $a \neq b$, $b \neq c$ and $c \neq a$:*

$$\begin{aligned}F_1 &= (\lambda a.\ App \cdot \ulcorner Var \urcorner \cdot (Var \cdot \overline{a}))\\ F_2 &= (\lambda a.\ \lambda b.\ \lambda c.\ App \cdot (App \cdot \ulcorner App \urcorner \cdot (\overline{c} \cdot \overline{a})) \cdot (\overline{c} \cdot \overline{b}))\\ F_3 &= (\lambda a.\ \lambda b.\ App \cdot \ulcorner Abs \urcorner \cdot (Abs \cdot (\lambda c.\ \overline{b} \cdot (\overline{a} \cdot \overline{c}))))\end{aligned}$$

Using the lemmas 2 and 3 we can derive the convertibility relations for F_1, F_2, F_3. The first two have no freshness obligations, however in the case of F_3 the internal λ-abstraction is not applied, so the freshness obligations for this variable remain:

$$F_1 \cdot A \approx App \cdot \ulcorner Var \urcorner \cdot (Var \cdot A) \qquad (10)$$

$$F_2 \cdot A \cdot B \cdot C \approx App \cdot (App \cdot \ulcorner App \urcorner \cdot (C \cdot A)) \cdot (C \cdot B) \qquad (11)$$

Assuming $x \# A$ and $x \# B$:

$$F_3 \cdot A \cdot B \approx App \cdot \ulcorner Abs \urcorner \cdot (Abs \cdot (\lambda x.\ B \cdot (A \cdot \overline{x}))) \qquad (12)$$

Lemma 9. *The terms F_1, F_2, F_3 are equivariant and are closed terms:*

$$\begin{array}{lll}\pi \bullet F_1 = F_1 & \pi \bullet F_2 = F_2 & \pi \bullet F_3 = F_3\\ supp\ F_1 = \varnothing & supp\ F_2 = \varnothing & supp\ F_3 = \varnothing\end{array}$$

Proof. Using the equivariance of the subterms and the fact that equivariance implies empty support.

Next, we define the terms A_1, A_2, A_3:

Definition 10. *Assuming $a \neq b$, $b \neq c$ and $c \neq a$:*

$$\begin{aligned}A_1 &= (\lambda a.\ \lambda b.\ F_1 \cdot \overline{a})\\ A_2 &= (\lambda a.\ \lambda b.\ \lambda c.\ F_2 \cdot \overline{a} \cdot \overline{b} \cdot [\overline{c}])\\ A_3 &= (\lambda a.\ \lambda b.\ F_3 \cdot \overline{a} \cdot [\overline{b}])\end{aligned}$$

Using lemma 3 we can derive the convertibility relations for the applied terms A_1, A_2, A_3 that will have no freshness obligations:

$$A_1 \cdot A \cdot B \approx F_1 \cdot A \tag{13}$$

$$A_2 \cdot A \cdot B \cdot C \approx F_2 \cdot A \cdot B \cdot [C] \tag{14}$$

$$A_3 \cdot A \cdot B \approx F_3 \cdot A \cdot [B] \tag{15}$$

Lemma 10. *A_1, A_2 and A_3 are closed terms:*

$$\mathit{supp}\ A_1 = \varnothing \qquad \mathit{supp}\ A_2 = \varnothing \qquad \mathit{supp}\ A_3 = \varnothing$$

Proof. Using lemma 9 and the properties of support.

Finally we define *Num*; the definition follows the idea from the beginning of this section:

Definition 11

$$\mathit{Num} = [[A_1,\ A_2,\ A_3]]$$

From the support of the terms A_1, A_2, A_3 and the fact that finite sequences preserve support, we also get that *Num* is a closed term:

$$\mathit{supp}\ \mathit{Num} = \varnothing$$

6 The Proof of the Second Fixed Point Theorem

In this Section we present to proof of the Second Fixed Point Theorem together with the most interesting lemma that combines all the definitions from the previous Section. Namely, all the constants F_{1-3} and A_{1-3} have been defined in such a way that the encoding of terms is λ-defined by *Num*. Here we will show that *Num* applied to an encoded a term is convertible to its encoding.

For the proofs presented in this Section we decide on using the Isabelle rendering of the formal proofs. We use the number of lemmas in the paper as the lemma names in Isabelle). The simplifier is set up with the lemmas about empty support and support preservation shown before in the paper. Given a λ-term *M* we prove the property by induction on the structure of *M*. For each of the three cases the proof proceeds by transforming the left-hand side using the convertibility rules and lemmas proven in the previous section obtaining the right-hand side:

```
lemma num_numeral:
  shows $\mathit{Num} \cdot \ulcorner M \urcorner \approx \ulcorner\ulcorner M \urcorner\urcorner$
proof (induct M)
  case $\overline{n}$
  have $\mathit{Num} \cdot \ulcorner(\overline{n})\urcorner = \mathit{Num} \cdot (\mathit{Var} \cdot \overline{n})$ by simp
  also have $\ldots = [[A_1,\ A_2,\ A_3]] \cdot (\mathit{Var} \cdot \overline{n})$ by simp
  also have $\ldots \approx \mathit{Var} \cdot \overline{n} \cdot [A_1,\ A_2,\ A_3]$ using 8 .
```

```
8     also have ... $\approx [A_1, A_2, A_3] \cdot U^{2}_{2} \cdot \overline{n} \cdot [A_1, A_2, A_3]$ using 5 .
9     also have ... $\approx A_1 \cdot \overline{n} \cdot [A_1, A_2, A_3]$ using 9 by simp
10    also have ... $\approx F_1 \cdot \overline{n}$ using 13 .
11    also have ... $\approx App \cdot \ulcorner Var \urcorner \cdot (Var \cdot \overline{n})$ using 10 .
12    also have ... $= \ulcorner\ulcorner(\overline{n})\urcorner\urcorner$ by simp
13    finally show $Num \cdot \ulcorner(\overline{n})\urcorner \approx \ulcorner\ulcorner(\overline{n})\urcorner\urcorner$ .
14  next
15    case $M \cdot N$
16    assume IH: $Num \cdot \ulcorner M \urcorner \approx \ulcorner\ulcorner M \urcorner\urcorner$  $Num \cdot \ulcorner N \urcorner \approx \ulcorner\ulcorner N \urcorner\urcorner$
17    have $Num \cdot \ulcorner(M \cdot N)\urcorner = Num \cdot (App \cdot \ulcorner M \urcorner \cdot \ulcorner N \urcorner)$ by simp
18    also have ... $= [[A_1, A_2, A_3]] \cdot (App \cdot \ulcorner M \urcorner \cdot \ulcorner N \urcorner)$ by simp
19    also have ... $\approx App \cdot \ulcorner M \urcorner \cdot \ulcorner N \urcorner \cdot [A_1, A_2, A_3]$ using 8 .
20    also have ... $\approx [A_1, A_2, A_3] \cdot U^{2}_{1} \cdot \ulcorner M \urcorner \cdot \ulcorner N \urcorner \cdot [A_1, A_2, A_3]$ using 6 .
21    also have ... $\approx A_2 \cdot \ulcorner M \urcorner \cdot \ulcorner N \urcorner \cdot [A_1, A_2, A_3]$ using 9 by simp
22    also have ... $\approx F_2 \cdot \ulcorner M \urcorner \cdot \ulcorner N \urcorner \cdot Num$ using 14 by simp
23    also have ... $\approx App \cdot (App \cdot \ulcorner App \urcorner \cdot (Num \cdot \ulcorner M \urcorner)) \cdot (Num \cdot \ulcorner N \urcorner)$ using 11 .
24    also have ... $\approx App \cdot (App \cdot \ulcorner App \urcorner \cdot \ulcorner\ulcorner M \urcorner\urcorner) \cdot (Num \cdot \ulcorner N \urcorner)$ using IH by simp
25    also have ... $\approx \ulcorner\ulcorner(M \cdot N)\urcorner\urcorner$ using IH by simp
26    finally show $Num \cdot \ulcorner(M \cdot N)\urcorner \approx \ulcorner\ulcorner(M \cdot N)\urcorner\urcorner$ .
27  next
28    case $\lambda x.\ P$
29    assume IH: $Num \cdot \ulcorner P \urcorner \approx \ulcorner\ulcorner P \urcorner\urcorner$
30    have $Num \cdot \ulcorner(\lambda x.\ P)\urcorner = Num \cdot (Abs \cdot (\lambda x.\ \ulcorner P \urcorner))$ by simp
31    also have ... $= [[A_1, A_2, A_3]] \cdot (Abs \cdot (\lambda x.\ \ulcorner P \urcorner))$ by simp
32    also have ... $\approx Abs \cdot (\lambda x.\ \ulcorner P \urcorner) \cdot [A_1, A_2, A_3]$ using 8 .
33    also have ... $\approx [A_1, A_2, A_3] \cdot U^{2}_{0} \cdot (\lambda x.\ \ulcorner P \urcorner) \cdot [A_1, A_2, A_3]$ using 7 .
34    also have ... $\approx A_3 \cdot (\lambda x.\ \ulcorner P \urcorner) \cdot [A_1, A_2, A_3]$ using 9 by simp
35    also have ... $\approx F_3 \cdot (\lambda x.\ \ulcorner P \urcorner) \cdot [[A_1, A_2, A_3]]$ using 15 .
36    also have ... $= F_3 \cdot (\lambda x.\ \ulcorner P \urcorner) \cdot Num$ by simp
37    also have ... $\approx App \cdot \ulcorner Abs \urcorner \cdot (Abs \cdot (\lambda x.\ Num \cdot ((\lambda x.\ \ulcorner P \urcorner) \cdot \overline{x})))$
38      by (rule 12) simp_all
39    also have ... $\approx App \cdot \ulcorner Abs \urcorner \cdot (Abs \cdot (\lambda x.\ Num \cdot \ulcorner P \urcorner))$ using 4 by simp
40    also have ... $\approx App \cdot \ulcorner Abs \urcorner \cdot (Abs \cdot (\lambda x.\ \ulcorner\ulcorner P \urcorner\urcorner))$ using IH by simp
41    also have ... $= \ulcorner\ulcorner(\lambda x.\ P)\urcorner\urcorner$ by simp
42    finally show $Num \cdot \ulcorner(\lambda x.\ P)\urcorner \approx \ulcorner\ulcorner(\lambda x.\ P)\urcorner\urcorner$ .
43  qed
```

The proof is similar to the text proof formalized, with one exception. In the λ-abstraction case, when transforming the application of F_3 (lines 36-38) a new variable name is necessary. This also comes from lemmas 2, 3, where we cannot remove freshness assumptions about variables introduced inside the term. In the above proof, it is also the only place where we need to use two methods to convince Isabelle that the transformation is correct. The application of F_3 (method rule 12) leaves proof obligations about freshness of the introduced variable. In this case the variable is supposed to be fresh for a closed term, which the simplifier can deduce automatically (method simp_all), however this may not always be the case (and it will not be the case in the final theorem).

The second fixed point theorems states, that for any λ-term F, there exists a term X, such that X is convertible to the application of F to the term representing the encoding of X. The proof defines the term X explicitly and shows that is has the desired property by transforming the term using convertibility rules and properties shown before:

```
44 theorem second_fixed_point_theorem:
45   fixes F :: lam
46   shows ∃X. X ≈ F · ⌜X⌝
47 proof -
48   obtain x :: var where x ♯ F using obtain_fresh by blast
49   def W = λx. F · (App · x̄ · (Num · x̄))
50   def X = W · ⌜W⌝
51   have a: X = W · ⌜W⌝ unfolding X_def ..
52   also have ... = (λx. F · (App · x̄ · (Num · x̄))) · ⌜W⌝ unfolding W_def ..
53   also have ... ≈ F · (App · ⌜W⌝ · (Num · ⌜W⌝)) by simp
54   also have ... ≈ F · (App · ⌜W⌝ · ⌜⌜W⌝⌝) by simp
55   also have ... ≈ F · ⌜(W · ⌜W⌝)⌝ by simp
56   also have ... = F · ⌜X⌝ unfolding X_def ..
57   finally show X ≈ F · ⌜X⌝ ..
58 qed
```

The proof is again similar to a textbook one, however as the first step of the proof we need to add line 48, even before the definitions of W and X. The term W introduces a new abstraction together with a new variable x. In the formal proof we assume that this variable is fresh for the original term F. This is indeed necessary, if x was bound in F then the first step of convertibility reasoning would leave the term $F[x := W]$ on the left side of the application until the end of the proof, and the property would not hold.

7 Conclusion

We compared a number of approaches for defining terms in the λ-calculus and showed a convenient way of defining constants. We start with a function that returns different atoms, define constants using those atoms and show their convertibility to the forms with freshness obligations. We use the simplifier only for convertibility equations, obtaining proofs where there are very few freshness obligations making them similar to paper proofs.

We formalized the second fixed point theorem of the λ-calculus, using Nominal Isabelle. We showed how to define constants and their properties, so that the proof resembles a paper proof. The main difference is obtaining a variable fresh for the given term. We show that if the variable is not fresh for the given term, the usual textbook proof does not hold.

In the proof we give the term that satisfies the second fixed point theorem explicitly; it remains to be seen how a meta-proof that talks about λ-definability could be formalized. The approach presented here could be compared with using standard combinator terms used by Norrish [11], this could perheapes simplify the equations for definitions with internal abstractions, in our case study this could be preferable in case of property 12.

7.1 Related Work

A number of proofs about the λ-calculus have already been performed with Nominal Isabelle. Examples include the Church-Rosser property, strong normalization. More examples are described in [13].

Norrish [10] derives an infrastructure for the λ-calculus manually in the HOL4 system, using properties from Nominal Logic. He proves a number of properties about the λ-calculus following [1] and the book by Hankin [6].

References

1. Barendregt, H.P.: The Lambda Calculus: Its Syntax and Semantics. Studies in Logic and the Foundations of Mathematics, vol. 103. Elsevier (2001)
2. Bengtson, J., Johansson, M., Parrow, J., Victor, B.: Psi-calculi: a framework for mobile processes with nominal data and logic. Logical Methods in Computer Science 7(1) (2011)
3. Bengtson, J., Parrow, J.: Formalising the pi-calculus using nominal logic. Logical Methods in Computer Science 5(2) (2008)
4. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending sledgehammer with smt solvers. Automated Deduction (2011) (accepted)
5. Böhm, C., Piperno, A., Guerrini, S.: Lambda-definition of function(al)s by normal forms. In: Sannella, D. (ed.) ESOP 1994. LNCS, vol. 788, pp. 135–149. Springer, Heidelberg (1994)
6. Hankin, C.: Lambda Calculi: A Guide for Computer Scientists. Graduate Texts in Computer Science, vol. 3. Clarendon Press (1993)
7. Huffman, B., Urban, C.: A New Foundation for Nominal Isabelle. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 35–50. Springer, Heidelberg (2010)
8. Kaliszyk, C., Urban, C.: Quotients revisited for Isabelle/HOL. In: Chu, W.C., Wong, W.E., Palakal, M.J., Hung, C.-C. (eds.) Proc. of the 26th ACM Symposium on Applied Computing (SAC 2011), pp. 1639–1644. ACM (2011)
9. Minamide, Y., Okuma, K.: Verifying CPS transformations in Isabelle/HOL. In: Proc. of the Workshop on Mechanized reasoning about languages with variable binding (MERLIN 2003). ACM (2003)
10. Norrish, M.: Mechanising λ-calculus using a classical first order theory of terms with permutations. Higher-Order and Symbolic Computation 19, 169–195 (2006)
11. Norrish, M.: Mechanised Computability Theory. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 297–311. Springer, Heidelberg (2011)
12. Pitts, A.M.: Nominal Logic, A First Order Theory of Names and Binding. Information and Computation 183, 165–193 (2003)

13. Urban, C., Tasson, C.: Nominal Techniques in Isabelle/HOL. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 38–53. Springer, Heidelberg (2005)
14. Urban, C., Berghofer, S.: A Recursion Combinator for Nominal Datatypes Implemented in Isabelle/HOL. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 498–512. Springer, Heidelberg (2006)
15. Urban, C., Cheney, J., Berghofer, S.: Mechanizing the Metatheory of LF. In: Proc. of the 23rd LICS Symposium, pp. 45–56 (2008)
16. Urban, C., Kaliszyk, C.: General Bindings and Alpha-Equivalence in Nominal Isabelle. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 480–500. Springer, Heidelberg (2011)