A Head-to-Head Comparison of de Bruijn Indices and Names

Stefan Berghofer[1] and Christian Urban[2]

Institut für Informatik
Technische Universität München
Boltzmannstraße 3, 85748 Garching, Germany

Abstract

Often debates about pros and cons of various techniques for formalising lambda-calculi rely on subjective arguments, such as de Bruijn indices are hard to read for humans or nominal approaches come close to the style of reasoning employed in informal proofs. In this paper we will compare four formalisations based on de Bruijn indices and on names from the nominal logic work, thus providing some hard facts about the pros and cons of these two formalisation techniques. We conclude that the relative merits of the different approaches, as usual, depend on what task one has at hand and which goals one pursues with a formalisation.

Key words: Proof assistants, lambda-calculi, de Bruijn indices, nominal logic work, Isabelle/HOL.

1 Introduction

When formalising lambda-calculi in a theorem prover, variable-binding and the associated notion of alpha-equivalence can cause some difficult problems. To mitigate these problems several formalisation techniques have been introduced. However, discussions about the merits of these formalisation techniques seem to be governed mainly by personal preference than by facts (see [1]). In this paper, we will study four examples and compare two formalisation techniques—de Bruijn indices [6] and names from nominal logic work [10,15]—in order to shed more light on their respective strengths and weaknesses.

In terms of ease and convenience the standard to which techniques for formalising lambda-calculi have to measure up is, in our opinion, the vast corpus of informal proofs in the existing literature. Even if one can find several works about lambda-calculi containing faulty reasoning, on the whole the informal reasoning

1 Email: `berghofe@in.tum.de`
2 Email: `urbanc@in.tum.de`

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

on "paper" seems to be quite robust, in particular issues arising from binders and alpha-equivalence seem to cause little problems and introduce almost no overhead. (The point of formalising lambda-calculi is to achieve 100% correctness, to provide easy maintenance of proofs and to allow for proofs about languages where a human reasoner is overwhelmed by the sheer number of cases and subtleties to be considered [3].)

When engineering a formal proof in a theorem prover, blindly applying automatic proof tools often leads to a dead end. Usually more successful is the strategy to start with a rough sketch containing a proof idea, and then to try to translate this idea into actual proof steps in the theorem prover. This style of formalising proofs is very much encouraged by the Isar-language of Isabelle [16]. In case of the substitution lemma in the lambda-calculus

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

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

one might start with the following informal proof given by Barendregt [4]:

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. □

In order to translate this informal proof to proof steps in a theorem prover, one has to decide how to encode lambda-terms and how to define the substitution operation. A naïve choice would be to represent the lambda-terms as the datatype

(1) **datatype** $lam = Var\ name \mid App\ lam\ lam \mid Lam\ name\ lam$

where the type $name$ can, for example, be strings or natural numbers. Since the term-constructor Lam has a concrete name, one has to prove the substitution lemma modulo an explicit notion of alpha-equivalence, that is one has to prove

$$M[x := N][y := L] \approx_\alpha M[y := L][x := N[y := L]] \; .$$

For the substitution operation one might follow Church [5] and define

$$
\begin{aligned}
(\mathit{Var}\ y)[x := N] &\stackrel{\text{def}}{=} \begin{cases} N & \text{if } x \equiv y \\ \mathit{Var}\ y & \text{otherwise} \end{cases} \\
(\mathit{App}\ M_1\ M_2)[x := N] &\stackrel{\text{def}}{=} \mathit{App}\ (M_1[x := N])\ (M_2[x := N]) \\
(\mathit{Lam}\ x\ M_1)[x := N] &\stackrel{\text{def}}{=} \mathit{Lam}\ x\ M_1 \\
(\mathit{Lam}\ y\ M_1)[x := N] &\stackrel{\text{def}}{=} \mathit{Lam}\ z\ (M_1[y := z][x := N])
\end{aligned}
\tag{2}
$$

where in the last clause it is assumed that $y \not\equiv x$, and if $x \notin FV(M_1)$ or $y \notin FV(N)$ then $z \equiv y$, otherwise z is the first variable in the sequence $v_0, v_1, v_2, \ldots$ not in M_1 or N.

Unfortunately, with these naïve choices the translation of the informal proof into actual reasoning steps is a nightmare: Already the simple property stating that $L[x := \ldots] \approx_\alpha L$ provided $x \notin FV(L)$ is a tour de force. In nearly all reasoning steps involving Lam one needs the property

$$\text{if } M \approx_\alpha M' \text{ and } N \approx_\alpha N' \text{ then } M[x := N] \approx_\alpha M'[x := N']$$

in order to manually massage the lambda-terms to a suitable form. The "rough sketches" Curry gives for this property extend over 10 pages [5, Pages 94–104]. As can be easily imagined, implementing these sketches results into a rather unpleasant experience with theorem provers—nothing of the sort that makes formalising proofs "addictive in a videogame kind of way" [8, Page 53]. One reason for the difficulties is the fact that Curry's substitution operation is not equivariant—that means is not independent under renamings [10].

The main point of de Bruijn indices and names from the nominal logic work is to allow for more clever methods of representing binders and to substantially reduce the amount of effort needed to formalise proofs. In Section 2 we illustrate this in the context of the substitution lemma. Section 3 contains a brief sketch of the formalisations for the narrowing and transitivity proof of subtyping from the POPLmark-Challenge [3]. Section 4 draws some conclusions.

2 The Substitution Lemma Formalised

2.1 Version using de Bruijn Indices

De Bruijn indices are sometimes labelled as a *hack* [3] since they are a very useful implementation technique, but are often dismissed as being unfit for consumption of a human reader. Yet six out of the eleven solutions currently submitted for the theorem proving part of the POPLmark-Challenge are based on some form of de Bruijn indices. This indicates that de Bruijn indices are quite respectable amongst

[3] personal communication with N. G. de Bruijn

theorem proving experts. In this section, for the benefit of casual users of theorem provers, we want to study in minutiae detail a formalisation of the substitution lemma using this formalisation technique.

We assume the reader is familiar with the de Bruijn notation of lambda-terms using for example the datatype:

$$\textbf{datatype}\ dB \ = \ \mathit{Var}\ \mathit{nat} \ \mid \ \mathit{App}\ dB\ dB \ \mid \ \mathit{Lam}\ dB$$

One central notion when working with de Bruijn indices is the *lifting* operation, written $\uparrow_k^n$ where n is an offset by which the indices greater or equal than k are incremented; k is the upper bound of indices that are regarded as *locally bound*. This operation can be defined as:

$$\begin{aligned}
\uparrow_k^n (\mathit{Var}\ i) &\stackrel{\text{def}}{=} \begin{cases} \mathit{Var}\ i & \text{if } i < k \\ \mathit{Var}\ (i+n) & \text{otherwise} \end{cases} \\
\uparrow_k^n (\mathit{App}\ M_1\ M_2) &\stackrel{\text{def}}{=} \mathit{App}\ (\uparrow_k^n M_1)\ (\uparrow_k^n M_2) \\
\uparrow_k^n (\mathit{Lam}\ M_1) &\stackrel{\text{def}}{=} \mathit{Lam}\ (\uparrow_{k+1}^n M_1)
\end{aligned}$$

The substitution of a term N for a variable with index k, written as $[k := N]$, can then be defined as follows:

$$\begin{aligned}
(\mathit{Var}\ i)[k := N] &\stackrel{\text{def}}{=} \begin{cases} \mathit{Var}\ i & \text{if } i < k \\ \uparrow_0^k N & \text{if } i = k \\ \mathit{Var}\ (i-1) & \text{if } i > k \end{cases} \\
(\mathit{App}\ M_1\ M_2)[k := N] &\stackrel{\text{def}}{=} \mathit{App}\ (M_1[k := N])\ (M_2[k := N]) \\
(\mathit{Lam}\ M)[k := N] &\stackrel{\text{def}}{=} \mathit{Lam}\ (M[k+1 := N])
\end{aligned}$$

Since the type dB is a completely standard datatype, both definitions can be implemented by primitive recursion. The substitution lemma then takes the following form:

> **Substitution Lemma with de Bruijn Indices:** For all indices i, j, with $i \leq j$ we have that
>
> $$M[i := N][j := L] = M[j+1 := L][i := N[j-i := L]] \ .$$

Note that one proves an equation, rather than an alpha-equivalence. Because equational reasoning is usually much better supported by theorem provers or is even a basic notion in their logics, the de Bruijn indices version avoids the manual massaging of terms with respect to alpha-equivalence needed in the version with concrete names. This fact alone already relieves one of much work when formalising this lemma. Notice also that the condition $i \leq j$ is necessary, otherwise the equation does not hold in general.

Like the informal proof by Barendregt, the formalised proof proceeds by induction on the structure of M. Unlike the informal proof, however, the induction hypothesis needs to be strengthened to quantify over *all* indices i and j. This strengthening is necessary in the de Bruijn version in order to get the Lam-case through. With this strengthening the Lam and App case are completely routine. The *non*-routine case in the de Bruijn version is the Var-case where we have to show that

(3) $$(\mathit{Var}\ n)[i := N][j := L] = (\mathit{Var}\ n)[j+1 := L][i := N[j-i := L]]$$

holds for an arbitrary n. Like in the informal proof, we need to distinguish cases so that we can apply the definition of substitution. There are several ways to order the cases; below we have given the cases as they are suggested by the definition of substitution (namely $n < i$, $n = i$ and $n > i$):

- Case $n < i$: We know by the assumption $i \leq j$ that also $n < j$ and $n < j+1$. Therefore both sides of (3) are equal to $\mathit{Var}\ n$.
- Case $n = i$: The left-hand side of (3) is therefore equal to $(\uparrow_0^i N)[j := L]$ and because we know by the assumption $i \leq j$ that $n < j+1$, the right-hand side is equal to $\uparrow_0^i (N[j-i := L])$. Now we have to show that both terms are equal. For this we prove first the lemma

  (4) $$\forall i, j.\ \text{if } i \leq j \text{ and } j \leq i + m \text{ then } \uparrow_j^n (\uparrow_i^m N) = \uparrow_i^{m+n} N$$

  which can be proved by induction on N. (The quantification over i and j is necessary in order to get the Lam-case through.) This lemma helps to prove the next lemma

  (5) $$\forall k, j.\ \text{if } k \leq j \text{ then } \uparrow_k^i (N[j := L]) = (\uparrow_k^i N)[j+i := L]$$

  which too can be proved by induction on N. (Again the quantification are crucial to get the induction through.) We can now instantiate this lemma with $k \mapsto 0$ and $j \mapsto j - i$, which makes the precondition trivially true and thus we obtain the equation

  $$\uparrow_0^i (N[j-i := L]) = (\uparrow_0^i N)[j-i+i := L]\ .$$

  The term $(\uparrow_0^i N)[j-i+i := L]$ is equal to $(\uparrow_0^i N)[j := L]$, as we had to show. However this last step is surprisingly *not* immediate: it depends on the assumption that $i \leq j$. This is because in theorem provers like Isabelle/HOL and Coq subtraction over natural numbers is defined so that $0 - n = 0$ and consequently the equation $j - i + i = j$ does not hold in general!
- Case $n > i$: Since the right-hand side of (3) equals $(Var(n-1))[j := L]$, we distinguish further three subcases (namely $n-1 < j$, $n-1 = j$ and $n-1 > j$):
  - Subcase $n-1 < j$: We therefore know also that $n < j+1$ and thus both sides of (3) are equal to $\mathit{Var}\ (n-1)$.
  - Subcase $n-1 = j$: Taking into account that $n > i$ implies $0 < n$, we have also $n = j+1$ (remember that because of the "quirk" with subtraction, this is not obvious). Hence we can calculate that the left-hand side of (3) equals $\uparrow_0^j L$ and the right-hand side to $(\uparrow_0^{j+1} L)[i := N[j-i := L]]$. To show that these terms are equal we need the lemma

    (6) $$\forall k, i.\ \text{if } k \leq i \text{ and } i < k + (j+1) \text{ then } (\uparrow_k^{j+1} L)[i := P] = \uparrow_k^j L$$

proved by induction on L. Instantiating this lemma with $k \mapsto 0$, $i \mapsto i$ and using the assumption $i \leq j$, we can infer that the preconditions of this lemma hold and thus can conclude that $(\uparrow_0^{j+1} L)[i := N[j - i := L]] = \uparrow_0^j L$.

- Subcase $n - 1 > j$: We therefore also know that $n > j + 1$. These inequalities in turn imply that both sides of (3) are equal to $\mathit{Var}\ (n - 2)$.

This concludes the proof of the substitution lemma. □

In this formalisation considerable ingenuity is needed when inventing the lemmas (4), (5) and (6). Also they are quite "brittle"—in the sense that they seem to go through just in the form stated. To find them can be a daunting task for an inexperienced user of theorem provers (they are only in little part inspired by the facts needed in the main proof). In practice however they seem to cause few problems, because they "carry over" from language to language, and hence one does not need to "invent the wheel" again for a new language. Theorem proving experts just copy these lemmas from existing formalisations. Indeed when submitting his solution of the POPLmark-Challenge, the first author only minimally adapted to System $F_{<:}$ the proofs Nipkow [9] gave in Isabelle/HOL for the lambda-calculus. Nipkow in turn got his collection of lemmas from Rasmussen [12] who worked with Isabelle/ZF. Nipkow wrote [9, Page 57]:

> " *Initially I tried to find and prove these lemmas from scratch but soon decided to steal them from Rasmussen's ZF proofs instead, which has obvious advantages:*
>
> *– I did not have to find this collection of non-obvious lemmas myself…* "

Rasmussen seems to have gotten his lemmas from a formalisation by Huet [7] in Coq.

In light of the subtleties and quirks in the proof based on de Bruijn indices, it might be surprising that one does not end up with a proof script of more than 100 lines of code. In fact the formalised proof by Nipkow consists of only a few lines—similar numbers for the lemmas corresponding to (4), (5) and (6). The reason is that one can "optimise" proof scripts by employing automatic proof tools. Such proof tools can make case distinctions and apply definitions without manual interference. However such optimisations are done *after* one has a formal proof like the one described above. As we mentioned earlier, just blindly attacking a problem with automatic proof tools leads to dead ends, except in the most trivial proofs, and the substitution lemma is already too complicated. This is not surprising considering how much ingenuity one needs to invent the lemmas (4), (5) and (6). However, once one knows how the proof proceeds, one can guide the automatic proof tools by providing explicitly the lemmas that lead to a proof. In case of the de Bruijn indices version of the substitution lemma, however, this kind of post-processing is not without pitfalls. For example it helps if the lemma is stated the other way around, namely as

$$M[j + 1 := L][i := N[j - i := L]] = M[i := N][j := L]$$

otherwise the simplifier can easily loop. As we shall see next, the proof based on names is much more robust in this respect.

2.2 Version using the Nominal Datatype Package

The nominal datatype package [13,15] allows to reason with “named” alpha-equivalent lambda-terms; one can define them by

(7) **nominal_datatype** $lam \;=\; Var\ name \;\mid App\ lam\ lam \;\mid Lam\ \langle\!\langle name\rangle\!\rangle lam$

where $name$ is a type representing atoms [10]—in informal proofs atoms are usually referred to as variables; $\langle\!\langle \ldots \rangle\!\rangle$ indicates that a name is bound in Lam. This definition allows one to write lambda-terms as $Lam\ a\ (Var\ a)$. Unlike the naïve representation mentioned in the Introduction, however, the nominal datatype lam stands for alpha-equivalence classes, that means one has equations such as

$$Lam\ x\ (Var\ x) = Lam\ y\ (Var\ y)\ .$$

When formalising the substitution lemma, this will allow us to reap the benefits of equational reasoning. However, it raises a small obstacle for the definition of the substitution operation. Using the infrastructure of the nominal datatype package one can define this operation as

$$\begin{aligned}
(Var\ y)[x := N] &\stackrel{\text{def}}{=} \begin{cases} N & \text{if } x \equiv y \\ Var\ y & \text{otherwise} \end{cases} \\
(App\ M_1\ M_2)[x := N] &\stackrel{\text{def}}{=} App\ (M_1[x := N])\ (M_2[x := N]) \\
(Lam\ y\ M_1)[x := N] &\stackrel{\text{def}}{=} Lam\ y\ (M_1[x := N]) \qquad \text{provided } y \# (x, N)
\end{aligned}$$

where the side-constraint $y \# (x, N)$ means that $y \neq x$ and y not free in N. However to ensure that one has indeed defined a function, one needs to verify some properties of the clauses by which substitution is defined (see [11,13] for the details). This requires some small proofs that have no counterpart in the informal proof and in the formalisation based on de Bruijn indices. This need of verifying some properties arises whenever a function is defined by recursion over the structure of alpha-equated lambda-terms.

With the definition of the nominal datatype lam comes the following *strong* structural induction principle [14,15]:

$$\frac{\begin{array}{l}\forall c\, x.\ \ P\ (Var\ x)\ c \\ \forall c\, M_1\, M_2.\ \ (\forall d.\ P\ M_1\ d)\ \wedge\ (\forall d.\ P\ M_2\ d)\ \Rightarrow P\ (App\ M_1\ M_2)\ c \\ \forall c\, z\, M.\ \ z \# c\ \wedge\ (\forall d.\ P\ M\ d)\ \Rightarrow\ P\,(Lam\ z\ M)\,c\end{array}}{P\, M\, c}$$

This induction principle states that if one wants to establish a property P for all lambda-terms M, then, as expected, one has to prove it for the constructors Var, App and Lam. It is called *strong* induction principle because it has Barendregt’s variable convention already built in. Barendregt assumes in his informal proof that in the lambda-case the binder z is not equal to x and y, and is not free in N and L. Using the strong induction principle, we will be able to mimic the variable convention by instantiating c, we call this the *context* of the induction, with $c \mapsto (x, y, N, L)$.[4] When it then comes to establishing the Lam-case, we can assume that the binder z is fresh for (x, y, N, L), that means is not equal to x and y, and is not free in N and L. As a result, the induction in the substitution lemma will go through smoothly, just like in Barendregt's informal proof. If the nominal datatype package had *not* provided such strong induction principles, reasoning would be quite inconvenient: one would have to rename binders so that, for example, substitutions can be moved under lambdas.

Despite the excellent notes from Barendregt conveying very well the proof idea, for the formalisation of the substitution lemma we need to supply some details that are left out in his notes. For example in Case 1.2 the details are left out for how to prove the property of

(8) $$x \# L \text{ implies that } L[x := P] = L\ .$$

where $x \# L$ stands for $x \notin FV(L)$. This fact can be proved by an induction over L using the strong induction principle. For this we make the following instantiations:

$$\begin{aligned} P &\mapsto \lambda L.\lambda(x, P).\ \ x \# L \ \Rightarrow\ L[x := P] = L \\ M &\mapsto L \\ c &\mapsto (x, P) \end{aligned}$$

As a result, the variable and application case are completely routine. In the lambda-case we have to show that $x \# (Lam\ z\ L_1)$ implies $(Lam\ z\ L_1)[x := P] = (Lam\ z\ L_1)$ with the assumption that $z \# (x, P)$ and the induction hypothesis

$$\forall x, L.\ x \# L\ \Rightarrow L[x := P] = L\ .$$

From the assumption that z is not equal to x and not free in L, we can infer from $x \# (Lam\ z\ L_1)$ that $x \# L_1$ holds and by applying the definition of substitution that $(Lam\ z\ L_1)[x := P] = Lam\ z\ (L_1[x := P])$. Now we just need to apply the induction hypothesis and are done.

Although not obvious from first glance, also in Case 2, in the last step of the calculation where the substitution is pulled back from under the binder λz, there are some details missing from Barendregt's informal proof. In order to get from $Lam\ z\ (M_1[y := L][x := N[y := L]])$ to $(Lam\ z\ M_1)[y := L][x := N[y := L]]$, we need the property that:

(9) $$\text{if } z \# N \text{ and } z \# L \text{ then } z \# (N[y := L])\ .$$

where the preconditions are given by his use of the variable convention. This property, too, can be easily proved by strong induction over the structure of N. In this induction we instantiate the induction context with $c \mapsto (z, y, L)$, because then we can in the Lam-case, say instantiated as $(Lam\ x\ N_1)$, move the substitution under

[4] An aspect we do not dwell on here is the fact that the induction context must always be finitely supported, i.e. mentions only finitely many free names, see [10,15].

the binder x and also infer from the assumption $z \# (Lam\ x\ N_1)$ that z is also fresh for N_1 (this reasoning step depends on $z \neq x$). Consequently we can apply the induction hypothesis and infer that $z \# N_1[y := L]$ holds. Again since $z \neq x$, also $z \# (Lam\ x\ N_1[y := L])$ holds and we are done.

The formalisation of the substitution lemma

Substitution Lemma with Names: If $x \neq y$ and $x \# L$ then

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

now follows almost to the word Barendregt's informal proof. The variable-case, say with the instantiation $(Var\ z)$, proceeds by a case-analysis with $z = x$, $z \neq x \wedge z = y$ and $z \neq x \wedge z \neq y$. The calculations involved are routine using in the second case the property in (8). The application case does not need any special attention. The lambda-case, too, is relatively easy: by instantiating the induction context with $c \mapsto (x, y, N, L)$, the strong induction principle allows us to assume that the binder is not equal to x and y, and is not free in $N$and L. Consequently we can reason like Barendregt:

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

where, as mentioned earlier, in the last equation we make use the property in (9).

The resulting formalised proof is quite simple: one only has to manually set up the induction and supply the properties (8) and (9) to the automatic proving tools for which it is a straightforward task to complete the proof (similar for the two side lemmas). We take this as an indicator that the formalised proof using names is "simpler" than the one based on de Bruijn indices.

3 Transitivity and Narrowing for Subtyping

Another proof where we can compare names and de Bruijn indices is the transitivity and narrowing proof for the subtyping relation described in the POPLmark-Challenge. This proof is quite tricky involving a simultaneous outer induction over a type and two inner inductions on the definition of the subtyping relation. The "rough notes" from which we can start the formalisations are given in [3] by the authors of this challenge.

3.1 Version using the Nominal Datatype Package

Using the nominal datatype package the types can be defined as

nominal_datatype $ty\ =\ Tvar\ name\ \mid Top\ \mid Fun\ ty\ ty\ \mid All\ ty\ \langle\!\langle name \rangle\!\rangle ty$

with typing contexts being lists of pairs consisting of a name and a type. A type T is *well-formed* w.r.t. a typing context Γ, written $\Gamma \vdash T$, provided $(supp\ T) \subseteq$

$(dom\ \Gamma)$—that means all free names of T, i.e. its *support* [10], must be included in the domain of the typing context Γ. A *valid* typing context, written $valid\ \Gamma$, is defined inductively by:

$$\frac{}{valid\ []} \qquad \frac{valid\ \Gamma \quad X \# (dom\ \Gamma) \quad \Gamma \vdash T}{valid\ ((X,T)::\Gamma)}$$

The subtyping relation, written $\Gamma \vdash S <: Q$, can then be inductively defined as follows:

$$\frac{valid\ \Gamma \quad \Gamma \vdash S}{\Gamma \vdash S <: Top}\ Top \qquad \frac{valid\ \Gamma \quad X \in (dom\ \Gamma)}{\Gamma \vdash Tvar\ X <: Tvar\ X}\ Refl$$

$$\frac{(X,S) \in \Gamma \quad \Gamma \vdash S <: T}{\Gamma \vdash Tvar\ X <: T}\ Trans \qquad \frac{\Gamma \vdash T_1 <: S_1 \quad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash Fun\ S_1\ S_2 <: Fun\ T_1\ T_2}\ Fun$$

$$\frac{\Gamma \vdash T_1 <: S_1 \quad X \# \Gamma \quad (X,T_1)::\Gamma \vdash S_2 <: T_2}{\Gamma \vdash All\ S_1\ X\ S_2 <: All\ T_1\ X\ T_2}\ All$$

These definitions are quite close to the "rough notes" from the POPLmark-Challenge; the only difference is that we had to ensure validity of the typing contexts in the leaves and to explicitly require that the binder X is fresh for Γ in the All-rule. The transitivity and narrowing lemma can then be stated as

Transitivity and Narrowing with Names: For all Γ, S, T, Δ, X, P, M, N:

- $\Gamma \vdash S <: Q$ and $\Gamma \vdash Q <: T$ implies $\Gamma \vdash S <: T$, and
- $(\Delta @ (X,Q) @ \Gamma) \vdash M <: N$ and $\Gamma \vdash P <: Q$
  implies $(\Delta @ (X,P) @ \Gamma) \vdash M <: N$.

About the proof of this lemma the POPLmark-paper states:

> " *The two parts are proved simultaneously, by induction on the size of Q. The argument for part (2) assumes that part (1) has been established already for the Q in question; part (1) uses part (2) only for strictly smaller Q.*"

The main point we want to make here is that the formal proof using names proceeds exactly as stated, while as we shall see later this is *not* the case for the de Bruijn indices version. The main inconvenience with the named approach is, however, that the proof then proceeds by two inner inductions on the definition of the subtyping relation and in order to follow the reasoning on "paper" one has to provide *manually* a strong version of the induction principle for subtyping. This strong induction principle has the form (showing only the premise for the All-inference rule):

$$\frac{\begin{array}{ll} \ldots & \\ \forall \Gamma\ X\ S_1\ S_2\ T_1\ T_2\ c. & X \# (c, \Gamma, T_1, S_1)\ \wedge\ \Gamma \vdash T_1 <: S_1\ \wedge \\ & (\forall d.\ P\ \Gamma\ T_1\ S_1\ d)\ \wedge\ \Gamma \vdash S_2 <: T_2\ \wedge\ (\forall d.\ P\ \Gamma\ S_2\ T_2\ d) \\ & \Rightarrow\ P\ \Gamma\ (All\ S_1\ X\ S_2)\ (All\ T_1\ X\ T_2)\ c \end{array}}{\Gamma \vdash S <: T \Rightarrow P\ \Gamma\ S\ T\ c}$$

where we can assume that $X \# (c, \Gamma, S_1, T_1)$. These freshness condition are crucial to get the induction through without the need of renaming binders. Unlike the strong structural induction principle that comes with a nominal datatype definition for "free", establishing the strong induction principle for subtyping is quite a task—something one does not want to burden up to the users of the nominal package. But so far, unfortunately, it is entirely burdened onto them. (This might change however in future versions of the nominal datatype package.)

3.2 Version using de Bruijn Indices

Two out of the three solution currently submitted that solve *all* theorem proving parts of the POPLmark-Challenge use de Bruijn indices.[5] The solution of the first author defines types as:

$$\textbf{datatype}\ \ dbT \ = \ Tvar\ nat \ \mid Top \ \mid Fun\ dbT\ dbT \ \mid All\ dbT\ dbT$$

with the lifting operation given by:

$$\begin{aligned}
\uparrow^n_k (Tvar\ i) &\stackrel{\text{def}}{=} \begin{cases} Tvar\ i & \text{if } i < k \\ Tvar\ (i+n) & \text{otherwise} \end{cases} \\
\uparrow^n_k Top &\stackrel{\text{def}}{=} Top \\
\uparrow^n_k (Fun\ S\ T) &\stackrel{\text{def}}{=} Fun\ (\uparrow^n_k S)(\uparrow^n_k T) \\
\uparrow^n_k (All\ S\ T) &\stackrel{\text{def}}{=} All\ (\uparrow^n_k S)\ (\uparrow^n_{k+1} T)
\end{aligned}$$

Note that the lifting operation preserves the size of a dbT-type. This often allows one to establish facts involving lifting using inductions over the size, if an induction over the structure is not strong enough.

Typing contexts are lists of types and the predicate for valid contexts is defined like in the named variant, except that we do not need freshness constraints when working with de Bruijn indices. One way for defining when a type is well-formed is by using the function

$$\begin{aligned}
frees\ j\ (Tvar\ i) &\stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } i < j \\ \{i-j\} & \text{otherwise} \end{cases} \\
frees\ j\ (Top) &\stackrel{\text{def}}{=} \emptyset \\
frees\ j\ (Fun\ S\ T) &\stackrel{\text{def}}{=} (frees\ j\ S) \cup (frees\ j\ T) \\
frees\ j\ (All\ S\ T) &\stackrel{\text{def}}{=} (frees\ j\ S) \cup (frees\ (j+1)\ T)
\end{aligned}$$

and then define the well-formedness judgement $\Gamma \vdash T$ as the proposition $(\forall i \in (frees\ 0\ T).\ i < |\Gamma|)$ where $|\Gamma|$ stands for the length of the list Γ. The look-up function for typing context is written $\Gamma(i)$ and returns the type on the ith place in

[5] The third uses higher-order abstract syntax in Twelf.

the list Γ. The inductive definition of the subtyping relation with de Bruijn indices takes then the following form:

$$\frac{\mathit{valid}\ \Gamma \quad \Gamma \vdash S}{\Gamma \vdash S <: \mathit{Top}}\ \mathit{Top} \qquad \frac{\mathit{valid}\ \Gamma \quad \Gamma \vdash \mathit{Tvar}\ i}{\Gamma \vdash \mathit{Tvar}\ i <: \mathit{Tvar}\ i}\ \mathit{Tvar}$$

$$\frac{\Gamma(i) = S \quad \Gamma \vdash (\uparrow_0^{i+1} S) <: T}{\Gamma \vdash \mathit{Tvar}\ i <: T}\ \mathit{Trans} \qquad \frac{\Gamma \vdash T_1 <: S_1 \quad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash \mathit{Fun}\ S_1\ S_2 <: \mathit{Fun}\ T_1\ T_2}\ \mathit{Fun}$$

$$\frac{\Gamma \vdash T_1 <: S_1 \quad T_1 :: \Gamma \vdash S_2 <: T_2}{\Gamma \vdash \mathit{All}\ S_1\ S_2 <: \mathit{All}\ T_1\ T_2}\ \mathit{All}$$

Whether these definitions require much ingenuity w.r.t. the informal rules given in the POPLmark-paper is a matter of taste, but an undebatable fact is that the proof for the transitivity and narrowing lemma formulated with de Bruijn indices as follows

Transitivity and Narrowing with de Bruijn Indices: For all Γ, S, T, Δ P, M, N:

- $\Gamma \vdash S <: Q$ and $\Gamma \vdash Q <: T$ implies $\Gamma \vdash S <: T$, and
- $(\Delta @ Q @ \Gamma) \vdash M <: N$ and $\Gamma \vdash P <: Q$ implies $(\Delta @ P @ \Gamma) \vdash M <: N$.

does *not* proceed as stated in the informal proof of the POPLmark-Challenge. Once one has set up the (outer) simultaneous induction over the size of Q, the inner induction for transitivity needs to be strengthened to apply not just for Q, but also for *all* types that have the same size as Q. That means the inner induction does *not* establish the property

$$\forall \Gamma\, S\, T.\ \Gamma \vdash S <: Q\ \wedge\ \Gamma \vdash Q <: T\ \Rightarrow \Gamma \vdash S <: T$$

rather the strengthened property

$$\begin{aligned} \forall Q'\, \Gamma\, S\, T.\ \ & (\mathit{size}\ Q) = (\mathit{size}\ Q')\ \wedge\ \Gamma \vdash S <: Q'\ \wedge\ \Gamma \vdash Q' <: T \\ & \Rightarrow \Gamma \vdash S <: T \end{aligned}$$

This strengthened property is needed in the narrowing part of the lemma where one needs transitivity not for Q, but for a lifted version of Q, where however the lifted version has the same size as Q.

The interesting details for the *Trans*-case in the narrowing proof are as follows: the statement is

$$\forall \Delta\, \Gamma\, M\, N\, P.\ \Delta @ Q @ \Gamma \vdash M <: N \Rightarrow \Gamma \vdash P <: Q \Rightarrow \Delta @ P @ \Gamma \vdash M <: N$$

and its proof procceds by an (inner) induction over the left-most subtyping relation. With the induction infrastructure [17] of Isabelle, we can implement this induction as stated above, without having to introduce "seemingly pointless equalities"[6] that handle syntactic constraints, such as the typing-context being of the form $\Delta @ Q @ \Gamma$. By induction hypothesis we know that $\Delta @ P @ \Gamma \vdash (\uparrow_0^{i+1} S) <: T$

[6] See solutions of the POPLmark-challenge by Chlipala and by Stump in Coq.

and $(\Delta @ Q @ \Gamma)(i) = S$, and we must show that $\Delta @ P @ \Gamma \vdash \mathit{Tvar}\ i <: T$ holds. The non-straightforward case is where $i = |\Delta|$, because then $(\Delta @ P @ \Gamma)(i) = P$ and we can infer that S equals Q. We have $\Gamma \vdash P <: Q$ by assumption and hence $\Delta @ P @ \Gamma \vdash (\uparrow_0^{i+1} P) <: (\uparrow_0^{i+1} Q)$ by weakening. Since $S = Q$ we can now use the transitivity property to infer that $\Delta @ P @ \Gamma \vdash (\uparrow_0^{i+1} P) <: T$. As can be seen, one needs transitivity for $(\uparrow_0^{i+1} Q)$ rather than for Q as stipulated in the informal proof. We then can conclude by applying the Trans-inference rule.

4 Conclusion

We have studied formalisations based on de Bruijn indices and on names from the nominal logic work. The former approach is already well-tested featuring in many formalisations, while the latter is still under heavy development in the nominal datatype package. Extrapolating an amazing amount from the submissions to the POPLmark-Challenge, it seems that all problems occurring in programming meta-theory can, in principle, be solved by theorem proving experts using de Bruijn indices. Further, the reasoning infrastructure needed for de Bruijn indices (mainly arithmetic over natural numbers) has been part of theorem provers, for example Coq and Isabelle/HOL, for a long time. In contrast, the nominal datatype package has been implemented in Isabelle/HOL, only. Except some preliminary work reported in [2], there is little work about replicating our results in non-HOL-based theorem provers.

Another advantage of de Bruijn indices is that they do not introduce any classical reasoning into the formalisation process. In contrast, the nominal datatype package employs in several places classical reasoning principles. It is currently unknown whether a constructive variant of the nominal datatype package that offers the same convenience is attainable. Connected with the aspect of constructivity is the infrastructure to extract programs from proof, which exists in Isabelle for the proofs with de Bruijn indices, but does not exist at all for proofs using the nominal datatype package.

The biggest disadvantage we see with using the nominal datatype package is the amount of infrastructure that needs to be implemented. So far, this package supports only single binders (although iteration is possible and they can occur anywhere in a term-constructor). One can imagine situations where this is not general enough or requires some unpleasant encodings. Unfortunately, if more general binding structures need to be supported, a considerable body of code must be adapted.

One big advantage of the nominal datatype package, we feel, is the relatively small "gap" between an informal proof on "paper" and an actual proof in a theorem prover. An important point we would like to highlight with this paper is that in the context of theorem proving the fact about de Bruijn indices being hard to read for humans is not the worst aspect: the biggest source of grief for us is the substantial amount of ingenuity needed to translate informal proofs to versions using de Bruijn indices. Since we are also the kind of theorem prover users who copied from existing formalisations when doing our own formalisations with de Bruijn indices, we were quite surprised how much reasoning is involved, if one unravels all the steps needed for the substitution lemma. This is an important aspect if one is in the business of educating students about formal proofs in the lambda-calculus: it is not difficult to imagine that a student will give up with great disgust, if one tries to explain the subtleties of de Bruijn indices in the substitution lemma. We hope therefore that the nominal datatype package will make broad inroads in this area. The slickness with which difficult proofs involving Barendregt's variable convention can be formalised in the nominal logic approach is something we cannot live without anymore.

The conclusion we draw from the comparisons is that the decision about favouring de Bruijn indices or names from the nominal logic work very much depends on what task one has at hand. It would be quite desirable to know how the other main formalisation technique—higher order abstract syntax—fares. But alas, we are not (yet) experts in Twelf, where this technique has been extensively employed.

Acknowledgements: The first author received funding via the BMBF project Verisoft. The second author is supported by an Emmy-Noether fellowship from the German Research Council.

References

[1] *POPLmark maling list*, `http://lists.seas.upenn.edu/pipermail/poplmark/`.

[2] Aydemir, B., A. Bohannon and S. Weirich, *Nominal Reasoning Techniques in Coq (Work in Progress)*, in: *Proc. of the International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP)*, Electronic Notes in Theoretical Computer Science, 2006.

[3] Aydemir, B. E., A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich and S. Zdancewic, *Mechanized Metatheory for the Masses: The PoplMark Challenge*, in: *Proc. of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS **3603**, 2005, pp. 50–65.

[4] Barendregt, H., "The Lambda Calculus: Its Syntax and Semantics," Studies in Logic and the Foundations of Mathematics **103**, North-Holland, 1981.

[5] Curry, H. B. and R. Feys, "Combinatory Logic Vol. I," Studies in Logic and the Foundations of Mathematics, North-Holland, 1958.

[6] de Bruijn, N. G., *Lambda-Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem*, Indagationes Math. **34** (1972), pp. 381–392.

[7] Huet, G., *Residual Theory in Lambda-Calculus: A Formal Development*, Journal of Functional Programming **4** (1994), pp. 371–394.

[8] Leroy, X., *Formal Certification of a Compiler Back-End, or: Programming a Compiler with a Proof Assistant*, in: *Proc. of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (2006), pp. 42–54.

[9] Nipkow, T., *More Church-Rosser Proofs (in Isabelle/HOL)*, Journal of Automated Reasoning **26** (2001), pp. 51–66.

[10] Pitts, A. M., *Nominal Logic, A First Order Theory of Names and Binding*, Information and Computation **186** (2003), pp. 165–193.

[11] Pitts, A. M., *Alpha-Structural Recursion and Induction (Extended Abstract)*, in: *Proc. of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS **3603**, 2005, pp. 17–34.

[12] Rasmussen, O., *The Church-Rosser Theorem in Isabelle: A Proof Porting Experiment*, Technical Report 364, Cambridge University (1995).

[13] Urban, C. and S. Berghofer, *A Recursion Combinator for Nominal Datatypes Implemented in Isabelle/HOL*, in: *Proc. of the 3rd International Joint Conference on Automated Reasoning (IJCAR)*, LNAI **4130**, 2006, pp. 498–512.

[14] Urban, C. 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 (MERLIN)*, 2005, pp. 25–32.

[15] Urban, C. and C. Tasson, *Nominal Techniques in Isabelle/HOL*, in: *Proc. of the 20th International Conference on Automated Deduction (CADE)*, LNCS **3632**, 2005, pp. 38–53.

[16] Wenzel, M., *Isar — A Generic Interpretative Approach to Readable Formal Proof Documents*, in: *Proc. of the 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, number 1690 in LNCS, 1999, pp. 167–184.

[17] Wenzel, M., *Structured Induction Proofs in Isabelle/Isar*, in: *Proc. of the 5th International Conference on Mathematical Knowledge Management (MKM)*, LNAI **4108**, 2006.