Proof Pearl: A New Foundation for Nominal Isabelle

Brian Huffman[1] and Christian Urban[2]

[1] Portland State University
[2] Technical University of Munich

Abstract. Pitts et al introduced a beautiful theory about names and binding based on the notions of permutation and support. The engineering challenge is to smoothly adapt this theory to a theorem prover environment, in our case Isabelle/HOL. We present a formalisation of this work that differs from our earlier approach in two important respects: First, instead of representing permutations as lists of pairs of atoms, we now use a more abstract representation based on functions. Second, whereas the earlier work modeled different sorts of atoms using different types, we now introduce a unified atom type that includes all sorts of atoms. Interestingly, we allow swappings, that is permutations build up by two atoms, to be ill-sorted. As a result of these design changes, we can iron out inconveniences for the user, considerably simplify proofs and also drastically reduce the amount of custom ML-code. Furthermore we can extend the capabilities of Nominal Isabelle to deal with variables that carry additional information. We end up with a pleasing and formalised theory of permutations and support, on which we can build an improved and more powerful version of Nominal Isabelle.

1 Introduction

Nominal Isabelle is a definitional extension of the Isabelle/HOL theorem prover providing a proving infrastructure for convenient reasoning about programming languages. It has been used to formalise an equivalence checking algorithm for LF [11], Typed Scheme [10], several calculi for concurrency [1,2] and a strong normalisation result for cut-elimination in classical logic [13]. It has also been used by Pollack for formalisations in the locally-nameless approach to binding [9].

At its core Nominal Isabelle is based on the nominal logic work of Pitts et al [6,8]. The most basic notion in this work is a sort-respecting permutation operation defined over a countably infinite collection of sorted atoms. The atoms are used for representing variables that might be bound. Multiple sorts are necessary for being able to represent different kinds of variables. For example, in the language Mini-ML there are bound term variables and bound type variables; each kind needs to be represented by a different sort of atoms.

Unfortunately, the type system of Isabelle/HOL is not a good fit for the way atoms and sorts are used in the original formulation of the nominal logic work. Therefore it was decided in earlier versions of Nominal Isabelle to use a separate type for each sort of atoms and let the type system enforce the sort-respecting property of permutations. Inspired by the work on nominal unification [12], it seemed best at the time to also implement permutations concretely as list of pairs of atoms. Thus Nominal Isabelle used the two-place permutation operation with the generic type

$$\_ \bullet \_ \ :: \ (\alpha \times \alpha)\ \mathit{list} \Rightarrow \beta \Rightarrow \beta$$

where α stands for the type of atoms and β for the type of the objects on which the permutation acts. For atoms of type α the permutation operation is defined over the length of lists as follows

$$\begin{aligned} [] \bullet c &= c \\ (a\ b)::\pi \bullet c &= \begin{cases} a & \mathit{if}\ \pi \bullet c = b \\ b & \mathit{if}\ \pi \bullet c = a \\ \pi \bullet c & \mathit{otherwise} \end{cases} \end{aligned} \tag{1}$$

where we write $(a\ b)$ for a swapping of atoms *a* and *b*. For atoms of different type, the permutation operation is defined as $\pi \bullet c \overset{def}{=} c$.

With the list representation of permutations it is impossible to state an "ill-sorted" permutation, since the type system excludes lists containing atoms of different type. Another advantage of the list representation is that the basic operations on permutations are already defined in the list library: composition of two permutations (written $\_ @ \_$) is just list append, and inversion of a permutation (written $\_^{-1}$) is just list reversal. A disadvantage is that permutations do not have unique representations as lists; we had to explicitly identify permutations according to the relation

$$\pi_1 \sim \pi_2 \overset{def}{=} \forall a.\ \pi_1 \bullet a = \pi_2 \bullet a \tag{2}$$

When lifting the permutation operation to other types, for example sets, functions and so on, we needed to ensure that every definition is well-behaved in the sense that it satisfies the following three *permutation properties*:

$$\begin{array}{rl} \mathit{i)} & [] \bullet x = x \\ \mathit{ii)} & (\pi_1 @ \pi_2) \bullet x = \pi_1 \bullet (\pi_2 \bullet x) \\ \mathit{iii)} & \mathit{if}\ \pi_1 \sim \pi_2\ \mathit{then}\ \pi_1 \bullet x = \pi_2 \bullet x \end{array} \tag{3}$$

From these properties we were able to derive most facts about permutations, and the type classes of Isabelle/HOL allowed us to reason abstractly about these three properties, and then let the type system automatically enforce these properties for each type.

The major problem with Isabelle/HOL's type classes, however, is that they support operations with only a single type parameter and the permutation operations $\_ \bullet \_$ used above in the permutation properties contain two! To work around this obstacle, Nominal Isabelle required from the user to declare up-front the collection of *all* atom types, say $\alpha_1,\ldots,\alpha_n$. From this collection it used custom ML-code to generate *n* type classes corresponding to the permutation properties, whereby in these type classes the permutation operation is restricted to

$$\_ \bullet \_ :: (\alpha_i \times \alpha_i)\ \mathit{list} \Rightarrow \beta \Rightarrow \beta$$

This operation has only a single type parameter β (the α_i are the atom types given by the user).

While the representation of permutations-as-list solved the "sort-respecting" requirement and the declaration of all atom types up-front solved the problem with Isabelle/HOL's type classes, this setup caused several problems for formalising the nominal logic work: First, Nominal Isabelle had to generate n^2 definitions for the permutation operation over n types of atoms. Second, whenever we need to generalise induction hypotheses by quantifying over permutations, we have to build cumbersome quantifications like

$$\forall \pi_1 \ldots \forall \pi_n. \ldots$$

where the π_i are of type $(\alpha_i \times \alpha_i)$ *list*. The reason is that the permutation operation behaves differently for every α_i. Third, although the notion of support

$$\mathit{supp}\ \_ :: \beta \Rightarrow \alpha\ \mathit{set}$$

which we will define later, has a generic type α *set*, it cannot be used to express the support of an object over *all* atoms. The reason is again that support can behave differently for each α_i. This problem is annoying, because if we need to know in a statement that an object, say x, is finitely supported we end up with having to state premises of the form

$$\mathit{finite}\ ((\mathit{supp}\ x) :: \alpha_1\ \mathit{set})\ , \ldots, \mathit{finite}\ ((\mathit{supp}\ x) :: \alpha_n\ \mathit{set}) \qquad (4)$$

Sometimes we can avoid such premises completely, if x is a member of a *finitely supported type*. However, keeping track of finitely supported types requires another n type classes, and for technical reasons not all types can be shown to be finitely supported.

The real pain of having a separate type for each atom sort arises, however, from another permutation property

$$\textit{iv)} \quad \pi_1 \bullet (\pi_2 \bullet x) = (\pi_1 \bullet \pi_2) \bullet (\pi_1 \bullet x)$$

where permutation π_1 has type $(\alpha \times \alpha)$ *list*, π_2 type $(\alpha' \times \alpha')$ *list* and x type β. This property is needed in order to derive facts about how permutations of different types interact, which is not covered by the permutation properties *i-iii* shown in (3). The problem is that this property involves three type parameters. In order to use again Isabelle/HOL's type class mechanism with only permitting a single type parameter, we have to instantiate the atom types. Consequently we end up with an additional n^2 slightly different type classes for this permutation property.

While the problems and pain can be almost completely hidden from the user in the existing implementation of Nominal Isabelle, the work is *not* pretty. It requires a large amount of custom ML-code and also forces the user to declare up-front all atom-types that are ever going to be used in a formalisation. In this paper we set out to solve the problems with multiple type parameters in the permutation operation, and in this way can dispense with the large amounts of custom ML-code for generating multiple variants for some basic definitions. The result is that we can implement a pleasingly simple formalisation of the nominal logic work.

Contributions of the paper: We use a single atom type for representing atoms of different sorts and use functions for representing permutations. This drastically reduces the number of type classes to just two (permutation types and finitely supported types), which we need in order reason abstractly about properties from the nominal logic work. The novel technical contribution of this paper is a mechanism for dealing with "Church-style" lambda-terms [5] and HOL-based languages [7] where variables and variable binding depend on type annotations.

2 Sorted Atoms and Sort-Respecting Permutations

In the nominal logic work of Pitts, binders and bound variables are represented by *atoms*. As stated above, we need to have different *sorts* of atoms to be able to bind different kinds of variables. A basic requirement is that there must be a countably infinite number of atoms of each sort. Unlike in our earlier work, where we identified each sort with a separate type, we implement here atoms to be

$$\textbf{datatype}\ \mathit{atom} = \mathit{Atom}\ \mathit{string}\ \mathit{nat}$$

whereby the string argument specifies the sort of the atom. (We use type *string* merely for convenience; any countably infinite type would work as well.) We have an auxiliary function *sort* that is defined as $\mathit{sort}\ (\mathit{Atom}\ s\ i) = s$, and we clearly have for every finite set X of atoms and every sort s the property:

Proposition 1. *If finite X then there exists an atom a such that sort $a = s$ and $a \notin X$.*

For implementing sort-respecting permutations, we use functions of type $\mathit{atom} \Rightarrow \mathit{atom}$ that *i*) are bijective; *ii*) are the identity on all atoms, except a finite number of them; and *iii*) map each atom to one of the same sort. These properties can be conveniently stated for a function π as follows:

$$\begin{array}{rl} i) & \mathit{bij}\ \pi \\ ii) & \mathit{finite}\ \{a \mid \pi\ a \neq a\} \\ iii) & \forall a.\ \mathit{sort}\ (\pi\ a) = \mathit{sort}\ a \end{array} \qquad (5)$$

Like all HOL-based theorem provers, Isabelle/HOL allows us to introduce a new type *perm* that includes just those functions satisfying all three properties. For example the identity function, written *id*, is included in *perm*. Also function composition, written $\_ \circ \_$, and function inversion, given by Isabelle/HOL's inverse operator and written $\mathit{inv}\ \_$, preserve the properties *i*-*iii*.

However, a moment of thought is needed about how to construct non-trivial permutations? In the nominal logic work it turned out to be most convenient to work with swappings, written $(a\ b)$. In our setting the type of swappings must be

$$(\_\ \_) :: \mathit{atom} \Rightarrow \mathit{atom} \Rightarrow \mathit{perm}$$

but since permutations are required to respect sorts, we must carefully consider what happens if a user states a swapping of atoms with different sorts. In earlier versions of Nominal Isabelle, we avoided this problem by using different types for different sorts; the type system prevented users from stating ill-sorted swappings. Here, however, definitions such as[3]

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

do not work in general, because the type system does not prevent a and b from having different sorts—in which case the function would violate property *iii*. We could make the definition of swappings partial by adding the precondition $\textit{sort}\ a = \textit{sort}\ b$, which would mean that in case a and b have different sorts, the value of $(a\ b)$ is unspecified. However, this looked like a cumbersome solution, since sort-related side conditions would be required everywhere, even to unfold the definition. It turned out to be more convenient to actually allow the user to state "ill-sorted" swappings but limit their "damage" by defaulting to the identity permutation in the ill-sorted case:

$$\begin{aligned}(a\ b) \overset{def}{=} {} & \textit{if}\ (\textit{sort}\ a = \textit{sort}\ b)\\ & \quad \textit{then}\ \lambda c.\ \textit{if}\ a = c\ \textit{then}\ b\ \textit{else}\ (\textit{if}\ b = c\ \textit{then}\ a\ \textit{else}\ c)\\ & \quad \textit{else}\ \textit{id}\end{aligned} \tag{6}$$

This function is bijective, the identity on all atoms except a and b, and sort respecting. Therefore it is a function in *perm*.

One advantage of using functions instead of lists as a representation for permutations is that for example the swappings

$$(a\ b) = (b\ a) \qquad\qquad (a\ a) = \textit{id} \tag{7}$$

are *equal*. We do not have to use the equivalence relation shown in (2) to identify them, as we would if they had been represented as lists of pairs. Another advantage of the function representation is that they form an (additive) group provided we define

$$\mathit{0} \overset{def}{=} \textit{id} \qquad \pi_1 + \pi_2 \overset{def}{=} \pi_1 \circ \pi_2 \qquad -\pi \overset{def}{=} \textit{inv}\ \pi \qquad \pi_1 - \pi_2 \overset{def}{=} \pi_1 + -\pi_2$$

and verify the simple properties

$$\pi_1 + \pi_2 + \pi_3 = \pi_1 + (\pi_2 + \pi_3) \qquad \mathit{0} + \pi = \pi \qquad \pi + \mathit{0} = \pi \qquad -\pi + \pi = \mathit{0}$$

Again this is in contrast to the list-of-pairs representation which does not form a group. The technical importance of this fact is that for groups we can rely on Isabelle/HOL's rich simplification infrastructure. This will come in handy when we have to do calculations with permutations.

By formalising permutations abstractly as functions, and using a single type for all atoms, we can now restate the *permutation properties* from (3) as just the two equations

$$\begin{aligned} &\textit{i)} \quad \mathit{0} \bullet x = x\\ &\textit{ii)} \quad (\pi_1 + \pi_2) \bullet x = \pi_1 \bullet \pi_2 \bullet x\end{aligned} \tag{8}$$

[3] To increase legibility, we omit here and in what follows the *Rep_perm* and *Abs_perm* wrappers that are needed in our implementation since we defined permutation not to be the full function space, but only those functions of type *perm* satisfying properties *i-iii*.

in which the permutation operations are of type *perm* $\Rightarrow \beta \Rightarrow \beta$ and so have only a single type parameter. Consequently, these properties are compatible with the one-parameter restriction of Isabelle/HOL's type classes. There is no need to introduce a separate type class instantiated for each sort, like in the old approach.

We call type β a *permutation type* if the permutation properties in (8) are satisfied for every x of type β. This notion allows us to establish generic lemmas, which are applicable to any permutation type. First, it follows from the laws governing groups that a permutation and its inverse cancel each other. That is, for any x of a permutation type:

$$\pi \bullet (-\pi) \bullet x = x \qquad (-\pi) \bullet \pi \bullet x = x \tag{9}$$

Consequently, in a permutation type the permutation operation $\pi \bullet \_$ is bijective, which in turn implies the property

$$\pi \bullet x = \pi \bullet y \ \textit{ if and only if } \ x = y. \tag{10}$$

In order to lift the permutation operation to other types, we can define for:

$$\begin{array}{rl}
\textit{atoms:} & \pi \bullet a \overset{def}{=} \pi\ a \\
\textit{functions:} & \pi \bullet f \overset{def}{=} \lambda x.\ \pi \bullet (f\ ((-\pi) \bullet x)) \\
\textit{permutations:} & \pi \bullet \pi' \overset{def}{=} \pi + \pi' - \pi \\
\textit{sets:} & \pi \bullet X \overset{def}{=} \{\pi \bullet x \mid x \in X\} \\
\textit{booleans:} & \pi \bullet b \overset{def}{=} b
\end{array}
\qquad
\begin{array}{rl}
\textit{lists:} & \pi \bullet [] \overset{def}{=} [] \\
 & \pi \bullet (x{::}xs) \overset{def}{=} (\pi \bullet x){::}(\pi \bullet xs) \\
\textit{products:} & \pi \bullet (x, y) \overset{def}{=} (\pi \bullet x, \pi \bullet y) \\
\textit{nats:} & \pi \bullet n \overset{def}{=} n
\end{array}$$

and then establish:

Theorem 1. *If β, β_1 and β_2 are permutation types, then so are: atom, $\beta_1 \Rightarrow \beta_2$, perm, β set, β list, $\beta_1 \times \beta_2$, bool and nat.*

Proof. All statements are by unfolding the definitions of the permutation operations and simple calculations involving addition and minus. With permutations for example we have

$$\begin{array}{rcl}
0 \bullet \pi' & \overset{def}{=} & 0 + \pi' - 0 = \pi' \\
(\pi_1 + \pi_2) \bullet \pi' & \overset{def}{=} & (\pi_1 + \pi_2) + \pi' - (\pi_1 + \pi_2) \\
 & = & (\pi_1 + \pi_2) + \pi' - \pi_2 - \pi_1 \\
 & = & \pi_1 + (\pi_2 + \pi' - \pi_2) - \pi_1 \\
 & \overset{def}{=} & \pi_1 \bullet \pi_2 \bullet \pi'
\end{array}$$ □

The main point is that the above reasoning blends smoothly with the reasoning infrastructure of Isabelle/HOL; no custom ML-code is necessary and a single type class suffices. We can also show once and for all that the following property—which caused so many headaches in our earlier setup—holds for any permutation type.

Lemma 1. *Given x is of permutation type, then $\pi_1 \bullet (\pi_2 \bullet x) = (\pi_1 \bullet \pi_2) \bullet (\pi_1 \bullet x)$.*

Proof. The proof is as follows:

$$\begin{aligned}\pi_1 \bullet \pi_2 \bullet x &= \pi_1 \bullet \pi_2 \bullet (-\pi_1) \bullet \pi_1 \bullet x && \textit{by } (9)\\ &= (\pi_1 + \pi_2 - \pi_1) \bullet \pi_1 \bullet x && \textit{by } (8.\textit{ii})\\ &\overset{\textit{def}}{=} (\pi_1 \bullet \pi_2) \bullet (\pi_1 \bullet x) && \square\end{aligned}$$

One huge advantage of using bijective permutation functions (as opposed to non-bijective renaming substitutions) is the property of *equivariance* and the fact that most HOL-functions (this includes constants) whose argument and result types are permutation types satisfy this property:

Definition 1. *A function f is* equivariant *if* $\forall \pi.\ \pi \bullet f = f$.

There are a number of equivalent formulations for the equivariance property. For example, assuming f is of type $\alpha \Rightarrow \beta$, then equivariance can also be stated as

$$\forall \pi\, x.\ \ \pi \bullet (f\,x) = f\ (\pi \bullet x) \tag{11}$$

To see that this formulation implies the definition, we just unfold the definition of the permutation operation for functions and simplify with the equation and the cancellation property shown in (9). To see the other direction, we use the fact

$$\pi \bullet (f\,x) = (\pi \bullet f)\ (\pi \bullet x) \tag{12}$$

which follows again directly from the definition of the permutation operation for functions and the cancellation property. Similarly for functions with more than one argument.

Both formulations of equivariance have their advantages and disadvantages: (11) is often easier to establish. For example we can easily show that equality is equivariant

$$\pi \bullet (x = y) = (\pi \bullet x = \pi \bullet y)$$

using the permutation operation on booleans and property (10). Lemma 1 establishes that the permutation operation is equivariant. It is also easy to see that the boolean operators, like $\wedge$, $\vee$ and $\longrightarrow$ are all equivariant. Furthermore a simple calculation will show that our swapping functions are equivariant, that is

$$\pi \bullet (a\ b) = ((\pi \bullet a)\ (\pi \bullet b)) \tag{13}$$

for all a, b and π. These equivariance properties are tremendously helpful later on when we have to push permutations inside terms.

3 Support and Freshness

The most original aspect of the nominal logic work of Pitts et al is a general definition for "the set of free variables of an object x". This definition is general in the sense that it applies not only to lambda-terms, but also to lists, products, sets and even functions. The definition depends only on the permutation operation and on the notion of equality defined for the type of x, namely:

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

(Note that due to the definition of swapping in (6), we do not need to explicitly restrict a and b to have the same sort.) There is also the derived notion for when an atom a is *fresh* for an x, defined as

$$a \# x \overset{\mathit{def}}{=} a \notin \mathit{supp}\ x$$

A striking consequence of these definitions is that we can prove without knowing anything about the structure of x that swapping two fresh atoms, say a and b, leave x unchanged. For the proof we use the following lemma about swappings applied to an x:

Lemma 2. *Assuming x is of permutation type, and a, b and c have the same sort, then $(a\ c) \bullet x = x$ and $(b\ c) \bullet x = x$ imply $(a\ b) \bullet x = x$.*

Proof. The cases where $a = c$ and $b = c$ are immediate. For the remaining case it is, given our assumptions, easy to calculate that the permutations

$$(a\ c) + (b\ c) + (a\ c) = (a\ b)$$

are equal. The lemma is then by application of the second permutation property shown in (8). □

Theorem 2. *Let x be of permutation type. If $a \# x$ and $b \# x$ then $(a\ b) \bullet x = x$.*

Proof. If a and b have different sort, then the swapping is the identity. If they have the same sort, we know by definition of support that both $\mathit{finite}\ \{c \mid (a\ c) \bullet x \neq x\}$ and $\mathit{finite}\ \{c \mid (b\ c) \bullet x \neq x\}$ hold. So the union of these sets is finite too, and we know by Proposition 1 that there is an atom c, with the same sort as a and b, that satisfies $(a\ c) \bullet x = x$ and $(b\ c) \bullet x = x$. Now the theorem follows from Lemma 2. □

Two important properties that need to be established for later calculations is that *supp* and freshness are equivariant. For this we first show that:

Lemma 3. *If x is a permutation type, then $\pi \bullet a \# \pi \bullet x$ if and only if $a \# x$.*

Proof.
$$\begin{array}{lll}
& \pi \bullet a \# \pi \bullet x & \\
\Leftrightarrow & \mathit{finite}\ \{b \mid ((\pi \bullet a)\ b) \bullet \pi \bullet x \neq \pi \bullet x\} & \textit{by definition} \\
\Leftrightarrow & \mathit{finite}\ \{b \mid ((\pi \bullet a)\ (\pi \bullet b)) \bullet \pi \bullet x \neq \pi \bullet x\} & \textit{since } \pi \bullet \_ \textit{ is bijective} \\
\Leftrightarrow & \mathit{finite}\ \{b \mid \pi \bullet (a\ b) \bullet x \neq \pi \bullet x\} & \textit{by (1) and (13)} \\
\Leftrightarrow & \mathit{finite}\ \{b \mid (a\ b) \bullet x \neq x\} & \textit{by (10)} \\
\Leftrightarrow & a \# x & \textit{by definition}
\end{array}$$
□

Together with the definition of the permutation operation on booleans, we can immediately infer equivariance of freshness:

$$\pi \bullet (a \# x) = (\pi \bullet a \# \pi \bullet x)$$

Now equivariance of *supp*, namely

$$\pi \bullet (supp\ S) = supp\ (\pi \bullet S)$$

is by noting that $supp\ x = \{a \mid \neg\ a \ \#\ x\}$ and that freshness and the logical connectives are equivariant.

While the abstract properties of support and freshness, particularly Theorem 2, are useful for developing Nominal Isabelle, one often has to calculate the support of some concrete object. This is straightforward for example for booleans, nats, products and lists:

$$\begin{array}{rl} booleans\text{:} & supp\ b = \varnothing \\ nats\text{:} & supp\ n = \varnothing \\ products\text{:} & supp\ (x, y) = supp\ x \cup supp\ y \end{array} \qquad \begin{array}{rl} lists\text{:} & supp\ [] = \varnothing \\ & supp\ (x{::}xs) = supp\ x \cup supp\ xs \end{array}$$

But establishing the support of atoms and permutations in our setup here is a bit trickier. To do so we will use the following notion about a *supporting set*.

Definition 2. *A set S* supports *x if for all atoms a and b not in S we have $(a\ b) \bullet x = x$.*

The main motivation for this notion is that we can characterise $supp\ x$ as the smallest finite set that supports x. For this we prove:

Lemma 4. *Let x be of permutation type.*

i) *If S supports x and finite S then $supp\ x \subseteq S$.*
ii) *$(supp\ x)$ supports x*
iii) *$supp\ x = S$ provided S supports x, finite S and S is the least such set, that means formally:*

$$\textit{for all } S', \textit{ if finite } S' \textit{ and } S' \textit{ supports } x \textit{ then } S \subseteq S'.$$

Proof. For *i*) we derive a contradiction by assuming there is an atom a with $a \in supp\ x$ and $a \notin S$. Using the second fact, the assumption that S *supports* x gives us that S is a superset of $\{b \mid (a\ b) \bullet x \neq x\}$, which is finite by the assumption of S being finite. But this means $a \notin supp\ x$, contradicting our assumption. Property *ii*) is by a direct application of Theorem 2. For the last property, part *i*) proves one "half" of the claimed equation. The other "half" is by property *ii*) and the fact that $supp\ x$ is finite by *i*). □

These are all relatively straightforward proofs adapted from the existing nominal logic work. However for establishing the support of atoms and permutations we found the following "optimised" variant of *iii*) more useful:

Lemma 5. *Let x be of permutation type. Then $supp\ x = S$ provided S supports x, finite S, and for all $a \in S$ and all $b \notin S$, with a and b having the same sort, then $(a\ b) \bullet x \neq x$*

Proof. By Lemma 4.*iii*) we have to show that for every finite set S' that supports x, $S \subseteq S'$ holds. We will assume that there is an atom a that is element of S, but not S' and derive a contradiction. Since both S and S' are finite, we can by Proposition 1 obtain an atom b, which has the same sort as a and for which we know $b \notin S$ and $b \notin S'$. Since we assumed $a \notin S'$ and we have that S' *supports* x, we have on one hand $(a\ b) \bullet x = x$. On the other hand, the fact $a \in S$ and $b \notin S$ imply $(a\ b) \bullet x \neq x$ using the assumed implication. This gives us the contradiction. □

Using this lemma we only have to show the following three proof-obligations

i) $\{c\}$ *supports c*
ii) *finite* $\{c\}$
iii) $\forall\, a \in \{c\}\; b \notin \{c\}.\; \mathit{sort}\; a = \mathit{sort}\; b \longrightarrow (a\; b) \bullet c \neq c$

in order to establish that $\mathit{supp}\; c = \{c\}$ holds. In Isabelle/HOL these proof-obligations can be discharged by easy simplifications. Similar proof-obligations arise for the support of permutations, which is

$$\mathit{supp}\; \pi = \{a \mid \pi \bullet a \neq a\}$$

The only proof-obligation that is interesting is the one where we have to show that

If $\pi \bullet a \neq a$, $\pi \bullet b = b$ *and* $\mathit{sort}\; a = \mathit{sort}\; b$, *then* $(a\; b) \bullet \pi \neq \pi$.

For this we observe that

$(a\; b) \bullet \pi = \pi$ *if and only if* $\pi \bullet (a\; b) = (a\; b)$

holds by a simple calculation using the group properties of permutations. The proof-obligation can then be discharged by analysing the inequality between the permutations $((p \bullet a)\; b)$ and $(a\; b)$.

The main point about support is that whenever an object x has finite support, then Proposition 1 allows us to choose for x a fresh atom with arbitrary sort. This is an important operation in Nominal Isabelle in situations where, for example, a bound variable needs to be renamed. To allow such a choice, we only have to assume *one* premise of the form

$$\mathit{finite}\; (\mathit{supp}\; x)$$

for each x. Compare that with the sequence of premises in our earlier version of Nominal Isabelle (see (4)). For more convenience we can define a type class for types where every element has finite support, and prove that the types *atom*, *perm*, lists, products and booleans are instances of this type class. Then *no* premise is needed, as the type system of Isabelle/HOL can figure out automatically when an object is finitely supported.

Unfortunately, this does not work for sets or Isabelle/HOL's function type. There are functions and sets definable in Isabelle/HOL for which the finite support property does not hold. A simple example of a function with infinite support is the function that returns the natural number of an atom

$$\mathit{nat\_of}\; (\mathit{Atom}\; s\; i) \overset{\mathit{def}}{=} i$$

This function's support is the set of *all* atoms. To establish this we show $\neg\, a \;\#\; \mathit{nat\_of}$. This is equivalent to assuming the set $\{b \mid (a\, b) \bullet \mathit{nat\_of} \neq \mathit{nat\_of}\}$ is finite and deriving a contradiction. From the assumption we also know that $\{a\} \cup \{b \mid (a\; b) \bullet \mathit{nat\_of} \neq \mathit{nat\_of}\}$ is finite. Then we can use Proposition 1 to choose an atom c such that $c \neq a$, $\mathit{sort}\; c = \mathit{sort}\; a$ and $(a\; c) \bullet \mathit{nat\_of} = \mathit{nat\_of}$. Now we can reason as follows:

$$\begin{aligned}
\mathit{nat\_of}\ a &= (a\ \ c) \bullet (\mathit{nat\_of}\ a) && \textit{by def. of permutations on nats}\\
&= ((a\ c) \bullet \mathit{nat\_of})\ ((a\ c) \bullet a) && \textit{by } (12)\\
&= \mathit{nat\_of}\ c && \textit{by assumptions on } c \qquad \square
\end{aligned}$$

But this means we have that $\mathit{nat\_of}\ a = \mathit{nat\_of}\ c$ and $\mathit{sort}\ a = \mathit{sort}\ c$. This implies that atoms a and c must be equal, which clashes with our assumption $c \neq a$ about how we chose c. Similar examples for constructions that have infinite support are given in [4].

4 Concrete Atom Types

So far, we have presented a system that uses only a single multi-sorted atom type. This design gives us the flexibility to define operations and prove theorems that are generic with respect to atom sorts. For example, as illustrated above the *supp* function returns a set that includes the free atoms of *all* sorts together; the flexibility offered by the new atom type makes this possible.

However, the single multi-sorted atom type does not make an ideal interface for end-users of Nominal Isabelle. If sorts are not distinguished by Isabelle's type system, users must reason about atom sorts manually. That means subgoals involving sorts must be discharged explicitly within proof scripts, instead of being inferred by Isabelle/HOL's type checker. In other cases, lemmas might require additional side conditions about sorts to be true. For example, swapping a and b in the pair (a, b) will only produce the expected result if we state the lemma in Isabelle/HOL as:

> **lemma**
> **fixes** $a\ b :: \mathit{atom}$
> **assumes** *asm*: $\mathit{sort}\ a = \mathit{sort}\ b$
> **shows** $(a\ \ b) \bullet (a, b) = (b, a)$
> **using** *asm* **by** *simp*

Fortunately, it is possible to regain most of the type-checking automation that is lost by moving to a single atom type. We accomplish this by defining *subtypes* of the generic atom type that only include atoms of a single specific sort. We call such subtypes *concrete atom types*.

The following Isabelle/HOL command defines a concrete atom type called *name*, which consists of atoms whose sort equals the string $''\mathit{name}''$.

> **typedef** $\mathit{name} = \{a \mid \mathit{sort}\ a = {''\mathit{name}''}\}$

This command automatically generates injective functions that map from the concrete atom type into the generic atom type and back, called representation and abstraction functions, respectively. We will write these functions as follows:

$$\lfloor\_\rfloor :: \mathit{name} \Rightarrow \mathit{atom} \qquad \lceil\_\rceil :: \mathit{atom} \Rightarrow \mathit{name}$$

With the definition $\pi \bullet a \stackrel{\mathit{def}}{=} \lceil \pi \bullet \lfloor a \rfloor \rceil$, it is straightforward to verify that the type *name* is a permutation type.

In order to reason uniformly about arbitrary concrete atom types, we define a type class that characterises type *name* and other similarly-defined types. The definition of the concrete atom type class is as follows: First, every concrete atom type must be a permutation type. In addition, the class defines an overloaded function that maps from the concrete type into the generic atom type, which we will write $|\_|$. For each class instance, this function must be injective and equivariant, and its outputs must all have the same sort.

$$\begin{array}{rl} i) & \textit{if } |a| = |b| \textit{ then } a = b \\ ii) & \pi \bullet |a| = |\pi \bullet a| \\ iii) & \textit{sort } |a| = \textit{sort } |b| \end{array} \tag{14}$$

With the definition $|a| \overset{def}{=} \lfloor a \rfloor$ we can show that *name* satisfies all the above requirements of a concrete atom type.

The whole point of defining the concrete atom type class was to let users avoid explicit reasoning about sorts. This benefit is realised by defining a special swapping operation of type $\alpha \Rightarrow \alpha \Rightarrow \textit{perm}$, where α is a concrete atom type:

$$(a \leftrightarrow b) \overset{def}{=} (|a|\ |b|)$$

As a consequence of its type, the $\leftrightarrow$-swapping operation works just like the generic swapping operation, but it does not require any sort-checking side conditions—the sort-correctness is ensured by the types! For $\leftrightarrow$ we can establish the following simplification rule:

$$(a \leftrightarrow b) \bullet c = (\textit{if } c = a \textit{ then } b \textit{ else if } c = b \textit{ then } a \textit{ else } c)$$

If we now want to swap the *concrete* atoms a and b in the pair (a, b) we can establish the lemma as follows:

lemma
  fixes $a\ b$ :: *name*
  shows $(a \leftrightarrow b) \bullet (a, b) = (b, a)$
by *simp*

There is no need to state an explicit premise involving sorts.

We can automate the process of creating concrete atom types, so that users can define a new one simply by issuing the command

atom_decl *name*

This command can be implemented using less than 100 lines of custom ML-code. In comparison, the old version of Nominal Isabelle included more than 1000 lines of ML-code for creating concrete atom types, and for defining various type classes and instantiating generic lemmas for them. In addition to simplifying the ML-code, the setup here also offers user-visible improvements: Now concrete atoms can be declared at any point of a formalisation, and theories that separately declare different atom types can be merged together—it is no longer required to collect all atom declarations in one place.

5 Multi-Sorted Concrete Atoms

The formalisation presented so far allows us to streamline proofs and reduce the amount of custom ML-code in the existing implementation of Nominal Isabelle. In this section we describe a mechanism that extends the capabilities of Nominal Isabelle. This mechanism is about variables with additional information, for example typing constraints. While we leave a detailed treatment of binders and binding of variables for a later paper, we will have a look here at how such variables can be represented by concrete atoms.

In the previous section we considered concrete atoms that can be used in simple binders like $\lambda x.\ x$. Such concrete atoms do not carry any information beyond their identities—comparing for equality is really the only way to analyse ordinary concrete atoms. However, in "Church-style" lambda-terms [5] and in the terms underlying HOL-systems [7] binders and bound variables have a more complicated structure. For example in the "Church-style" lambda-term

$$\lambda x_\alpha.\ x_\alpha\ x_\beta \tag{15}$$

both variables and binders include typing information indicated by α and β. In this setting, we treat x_α and x_β as distinct variables (assuming $\alpha \neq \beta$) so that the variable x_α is bound by the lambda-abstraction, but not x_β.

To illustrate how we can deal with this phenomenon, let us represent object types like α and β by the datatype

datatype *ty* = *TVar string* | *ty* → *ty*

If we attempt to encode a variable naively as a pair of a *name* and a *ty*, we have the problem that a swapping, say $(x \leftrightarrow y)$, applied to the pair

$$((x, \alpha), (x, \beta))$$

will always permute *both* occurrences of x, even if the types α and β are different. This is unwanted, because it will eventually mean that both occurrences of x will become bound by a corresponding binder.

Another attempt might be to define variables as an instance of the concrete atom type class, where a *ty* is somehow encoded within each variable. Remember we defined atoms as the datatype:

datatype *atom* = *Atom string nat*

Considering our method of defining concrete atom types, the usage of a string for the sort of atoms seems a natural choice. However, none of the results so far depend on this choice and we are free to change it. One possibility is to encode types or any other information by making the sort argument parametric as follows:

datatype *'a atom* = *Atom 'a nat*

The problem with this possibility is that we are then back in the old situation where our permutation operation is parametric in two types and this would require to work around Isabelle/HOL's restriction on type classes. Fortunately, encoding the types in a separate parameter is not necessary for what we want to achieve, as we only have to know when two types are equal or not. The solution is to use a different sort for each object type. Then we can use the fact that permutations respect *sorts* to ensure that permutations also respect *object types*. In order to do this, we must define an injective function *sort_ty* mapping from object types to sorts. For defining functions like *sort_ty*, it is more convenient to use a tree datatype for sorts. Therefore we define

datatype $sort = Sort\ string\ (sort\ list)$
datatype $atom = Atom\ sort\ nat$

With this definition, the sorts we considered so far can be encoded just as *Sort s* []. The point, however, is that we can now define the function *sort_ty* simply as

$$\begin{aligned} &sort\_ty\ (TVar\ s) = Sort\ ''TVar''\ [Sort\ s\ []] \\ &sort\_ty\ (\tau_1 \to \tau_2) = Sort\ ''Fun''\ [sort\_ty\ \tau_1, sort\_ty\ \tau_2] \end{aligned} \tag{16}$$

which can easily be shown to be injective.

Having settled on what the the sorts should be for "Church-like" atoms, we have to give a subtype definition for concrete atoms. Previously we identified a subtype consisting of atoms of only one specified sort. This must be generalised to all sorts the function *sort_ty* might produce, i.e. the range of *sort_ty*. Therefore we define

typedef $var = \{a \mid sort\ a \in range\ sort\_ty\}$

This command gives us again injective representation and abstraction functions. We will write them also as $\lfloor\_\rfloor :: var \Rightarrow atom$ and $\lceil\_\rceil :: atom \Rightarrow var$, respectively.

We can define the permutation operation for *var* as $\pi \bullet a \overset{def}{=} \lceil \pi \bullet \lfloor a \rfloor \rceil$ and the injective function to type *atom* as $|a| \overset{def}{=} \lfloor a \rfloor$. Finally, we can define a constructor function that makes a *var* from a variable name and an object type:

$$Var\ x\ \alpha \overset{def}{=} \lceil Atom\ (sort\_ty\ \alpha)\ x \rceil$$

With these definitions we can verify all the properties for concrete atom types except Property 14.*iii*), which requires every atom to have the same sort. This last property is clearly not true for type *var*. This fact is slightly unfortunate since this property allowed us to use the type-checker in order to shield the user from all sort-constraints. But this failure is expected here, because we cannot burden the type-system of Isabelle/HOL with the task of deciding when two object types are equal. This means we sometimes need to explicitly state sort constraints or explicitly discharge them, but as we will see in the lemma below this seems a natural price to pay in these circumstances.

To sum up this section, the encoding of type-information into atoms allows us to form the swapping $(Var\ x\ \alpha \leftrightarrow Var\ y\ \alpha)$ and to prove the following lemma

lemma
  assumes *asm*: $\alpha \neq \beta$
  shows $(Var\ x\ \alpha \leftrightarrow Var\ y\ \alpha) \bullet (Var\ x\ \alpha, Var\ x\ \beta) = (Var\ y\ \alpha, Var\ x\ \beta)$
  using *asm* **by** *simp*

As we expect, the atom *Var x* β is left unchanged by the swapping. With this we can faithfully represent bindings in languages involving "Church-style" terms and bindings as shown in (15). We expect that the creation of such atoms can be easily automated so that the user just needs to specify

atom_decl *var* (*ty*)

where the argument, or arguments, are datatypes for which we can automatically define an injective function like *sort_ty* (see (16)). Our hope is that with this approach the authors of [3] can make headway with formalising their results about simple type theory. Because of its limitations, they did not attempt this with the old version of Nominal Isabelle. We also hope we can make progress with formalisations of HOL-based languages.

6 Conclusion

This proof pearl describes a new formalisation of the nominal logic work by Pitts et al. With the definitions we presented here, the formal reasoning blends smoothly with the infrastructure of the Isabelle/HOL theorem prover. Therefore the formalisation will be the underlying theory for a new version of Nominal Isabelle.

The main difference of this paper with respect to existing work on Nominal Isabelle is the representation of atoms and permutations. First, we used a single type for sorted atoms. This design choice means for a term *t*, say, that its support is completely characterised by *supp t*, even if the term contains different kinds of atoms. Also, whenever we have to generalise an induction so that a property *P* is not just established for all *t*, but for all *t and* under all permutations π, then we only have to state $\forall\,\pi.\ P\ (\pi \bullet t)$. The reason is that permutations can now consist of multiple swapping each of which can swap different kinds of atoms. This simplifies considerably the reasoning involved in building Nominal Isabelle.

Second, we represented permutation as functions so that the associated permutation operation has only a single type parameter. From this we derive most benefits because the abstract reasoning about permutations fit cleanly with Isabelle/HOL's type classes. No custom ML-code is required to work around rough edges. Moreover, by establishing that our permutations-as-functions representation satisfy the group properties, we were able to use extensively Isabelle/HOL's reasoning infrastructure for groups. This often reduced proofs to simple calculations over $+$, $-$ and *0*. An interesting point is that we defined the swapping operation so that a swapping of two atoms with different sorts is *not* excluded, like in our older work on Nominal Isabelle, but there is no "effect" of such a swapping (it is defined as the identity). This is a crucial insight in order to make the approach based on a single type of sorted atoms to work.

We noticed only one disadvantage of the permutations-as-functions: Over lists we can easily perform inductions. For permutation made up from functions, we have to manually derive an appropriate induction principle. We can establish such a principle, but we have no experience yet whether ours is the most useful principle: such an induction principle was not needed in any of the reasoning we ported from the old Nominal Isabelle.

Finally, our implementation of sorted atoms turned out powerful enough to use it for representing variables that carry additional information, for example typing annotations. This information is encoded into the sorts. With this we can represent conveniently binding in "Church-style" lambda-terms and HOL-based languages. We are not aware of any other approach proposed for language formalisations that can deal conveniently with such binders.

The formalisation presented here will eventually become part of the Isabelle distribution, but for the moment it can be downloaded from the Mercurial repository linked at http://isabelle.in.tum.de/nominal/download.

Acknowledgements: We are very grateful to Jesper Bengtson, Stefan Berghofer and Cezary Kaliszyk for their comments on earlier versions of this paper.

References

1. J. Bengtson and J. Parrow. Formalising the pi-Calculus using Nominal Logic. In *Proc. of the 10th International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*, volume 4423 of *LNCS*, pages 63–77, 2007.
2. J. Bengtson and J. Parrow. Psi-Calculi in Isabelle. In *Proc of the 22nd Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *LNCS*, pages 99–114, 2009.
3. C. Benzmüller and L. C. Paulson. *Quantified Multimodal Logics in Simple Type Theory*. SEKI Report SR–2009–02 (ISSN 1437-4447). SEKI Publications, 2009. http://arxiv.org/abs/0905.2435.
4. J. Cheney. Completeness and Herbrand theorems for Nominal Logic. *Journal of Symbolic Logic*, 71(1):299–320, 2006.
5. A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(2):56–68, 1940.
6. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002.
7. A. M. Pitts. *Syntax and Semantics*. Part of the documentation for the HOL4 system.
8. A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 183:165–193, 2003.
9. M. Sato and R. Pollack. External and Internal Syntax of the Lambda-Calculus. To appear in *Journal of Symbolic Computation*.
10. S. Tobin-Hochstadt and M. Felleisen. The Design and Implementation of Typed Scheme. In *Proc. of the 35rd Symposium on Principles of Programming Languages (POPL)*, pages 395–406. ACM, 2008.
11. C. Urban, J. Cheney, and S. Berghofer. Mechanizing the Metatheory of LF. In *Proc. of the 23rd IEEE Symposium on Logic in Computer Science (LICS)*, pages 45–56, 2008.
12. C. Urban, A. Pitts, and M. Gabbay. Nominal Unification. *Theoretical Computer Science*, 323(1-3):473–497, 2004.
13. C. Urban and B. Zhu. Revisiting Cut-Elimination: One Difficult Proof is Really a Proof. In *Proc. of the 9th International Conference on Rewriting Techniques and Applications (RTA)*, volume 5117 of *LNCS*, pages 409–424, 2008.