# General Bindings and Alpha-Equivalence in Nominal Isabelle

Christian Urban and Cezary Kaliszyk

TU Munich, Germany

**Abstract.** Nominal Isabelle is a definitional extension of the Isabelle/HOL theorem prover. It provides a proving infrastructure for reasoning about programming language calculi involving named bound variables (as opposed to de-Bruijn indices). In this paper we present an extension of Nominal Isabelle for dealing with general bindings, that means term-constructors where multiple variables are bound at once. Such general bindings are ubiquitous in programming language research and only very poorly supported with single binders, such as lambda-abstractions. Our extension includes new definitions of $\alpha$-equivalence and establishes automatically the reasoning infrastructure for $\alpha$-equated terms. We also prove strong induction principles that have the usual variable convention already built in.

## 1 Introduction

So far, Nominal Isabelle provided a mechanism for constructing $\alpha$-equated terms, for example lambda-terms, $t ::= x \mid t\ t \mid \lambda x.\ t$, where free and bound variables have names. For such $\alpha$-equated terms, Nominal Isabelle derives automatically a reasoning infrastructure that has been used successfully in formalisations of an equivalence checking algorithm for LF [18], Typed Scheme [17], several calculi for concurrency [2] and a strong normalisation result for cut-elimination in classical logic [21]. It has also been used by Pollack for formalisations in the locally-nameless approach to binding [14].

However, Nominal Isabelle has fared less well in a formalisation of the algorithm W [19], where types and type-schemes are, respectively, of the form

$$T ::= x \mid T \to T \qquad S ::= \forall \{x_1, \ldots, x_n\}.\ T \tag{1}$$

and the $\forall$-quantification binds a finite (possibly empty) set of type-variables. While it is possible to implement this kind of more general binders by iterating single binders, this leads to a rather clumsy formalisation of W.

Binding multiple variables has interesting properties that cannot be captured easily by iterating single binders. For example in the case of type-schemes we do not want to make a distinction about the order of the bound variables. Therefore we would like to regard the first pair of type-schemes as $\alpha$-equivalent, but assuming that $x$, $y$ and $z$ are distinct variables, the second pair should *not* be $\alpha$-equivalent:

$$\forall \{x, y\}.\ x \to y\ \approx_\alpha\ \forall \{y, x\}.\ y \to x \qquad \forall \{x, y\}.\ x \to y\ \not\approx_\alpha\ \forall \{z\}.\ z \to z \tag{2}$$

Moreover, we like to regard type-schemes as $\alpha$-equivalent, if they differ only on *vacuous* binders, such as

$$\forall\{x\}.\, x \to y \;\approx_\alpha\; \forall\{x, z\}.\, x \to y \tag{3}$$

where $z$ does not occur freely in the type. In this paper we will give a general binding mechanism and associated notion of $\alpha$-equivalence that can be used to faithfully represent this kind of binding in Nominal Isabelle.

However, the notion of $\alpha$-equivalence that is preserved by vacuous binders is not always wanted. For example in terms like

$$\texttt{let}\; x = 3\; \texttt{and}\; y = 2\; \texttt{in}\; x - y\; \texttt{end} \tag{4}$$

we might not care in which order the assignments $x = 3$ and $y = 2$ are given, but it would be often unusual to regard (4) as $\alpha$-equivalent with

$$\texttt{let}\; x = 3\; \texttt{and}\; y = 2\; \texttt{and}\; z = \mathit{foo}\; \texttt{in}\; x - y\; \texttt{end}$$

Therefore we will also provide a separate binding mechanism for cases in which the order of binders does not matter, but the "cardinality" of the binders has to agree.

However, we found that this is still not sufficient for dealing with language constructs frequently occurring in programming language research. For example in `lets` containing patterns like

$$\texttt{let}\; (x, y) = (3, 2)\; \texttt{in}\; x - y\; \texttt{end} \tag{5}$$

we want to bind all variables from the pattern inside the body of the `let`, but we also care about the order of these variables, since we do not want to regard (5) as $\alpha$-equivalent with

$$\texttt{let}\; (y, x) = (3, 2)\; \texttt{in}\; x - y\; \texttt{end}$$

As a result, we provide three general binding mechanisms each of which binds multiple variables at once, and let the user chose which one is intended in a formalisation.

By providing these general binding mechanisms, however, we have to work around a problem that has been pointed out by Pottier [13] and Cheney [5]: in `let`-constructs of the form

$$\texttt{let}\; x_1 = t_1\; \texttt{and}\; \ldots\; \texttt{and}\; x_n = t_n\; \texttt{in}\; s\; \texttt{end}$$

we care about the information that there are as many bound variables $x_i$ as there are $t_i$. We lose this information if we represent the `let`-constructor by something like

$$\texttt{let}\; (\lambda x_1 \ldots x_n\, .\, s)\;\; [t_1, \ldots, t_n]$$

where the notation $\lambda\_ \ldots \_\, .\, \_$ indicates that the list of $x_i$ becomes bound in $s$. In this representation the term $\texttt{let}\; (\lambda x\, .\, s)\;\; [t_1, t_2]$ is a perfectly legal instance, but the lengths of the two lists do not agree. To exclude such terms, additional predicates about well-formed terms are needed in order to ensure that the two lists are of equal length. This can result in very messy reasoning (see for example [2]). To avoid this, we will allow type specifications for `lets` as follows

$$\begin{array}{rcl} \mathit{trm} & ::= & \ldots \mid \texttt{let}\; \mathit{as}{::}\mathit{assn}\; \mathit{s}{::}\mathit{trm}\;\; \textbf{bind}\; \mathit{bn}(\mathit{as})\; \textbf{in}\; s \\ \mathit{assn} & ::= & \texttt{anil} \mid \texttt{acons}\; \mathit{name}\; \mathit{trm}\; \mathit{assn} \end{array}$$

where *assn* is an auxiliary type representing a list of assignments and *bn* an auxiliary function identifying the variables to be bound by the `let`. This function can be defined by recursion over *assn* as follows

$$bn(\texttt{anil}) = \varnothing \qquad bn(\texttt{acons}\ x\ t\ as) = \{x\} \cup bn(as)$$

The scope of the binding is indicated by labels given to the types, for example $s$::*trm*, and a binding clause, in this case **bind** $bn(as)$ **in** $s$. This binding clause states that all the names the function $bn(as)$ returns should be bound in $s$. This style of specifying terms and bindings is heavily inspired by the syntax of the Ott-tool [16].

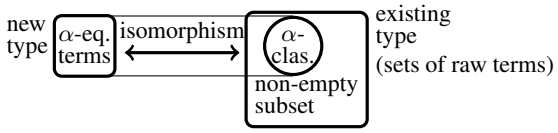
However, we will not be able to cope with all specifications that are allowed by Ott. One reason is that Ott lets the user specify "empty" types like $t ::= t\ t \mid \lambda x.\ t$ where no clause for variables is given. Arguably, such specifications make some sense in the context of Coq's type theory (which Ott supports), but not at all in a HOL-based environment where every datatype must have a non-empty set-theoretic model.

Another reason is that we establish the reasoning infrastructure for $\alpha$-*equated* terms. In contrast, Ott produces a reasoning infrastructure in Isabelle/HOL for *non*-$\alpha$-equated, or "raw", terms. While our $\alpha$-equated terms and the raw terms produced by Ott use names for bound variables, there is a key difference: working with $\alpha$-equated terms means, for example, that the two type-schemes

$$\forall\, \{x\}.\ x \to y\ = \forall\, \{x, z\}.\ x \to y$$

are not just $\alpha$-equal, but actually *equal*! As a result, we can only support specifications that make sense on the level of $\alpha$-equated terms (offending specifications, which for example bind a variable according to a variable bound somewhere else, are not excluded by Ott, but we have to).

Although in informal settings a reasoning infrastructure for $\alpha$-equated terms is nearly always taken for granted, establishing it automatically in Isabelle/HOL is a rather non-trivial task. For every specification we will need to construct type(s) containing as elements the $\alpha$-equated terms. To do so, we use the standard HOL-technique of defining a new type by identifying a non-empty subset of an existing type. The construction we perform in Isabelle/HOL can be illustrated by the following picture:




We take as the starting point a definition of raw terms (defined as a datatype in Isabelle/HOL); then identify the $\alpha$-equivalence classes in the type of sets of raw terms according to our $\alpha$-equivalence relation, and finally define the new type as these $\alpha$-equivalence classes (non-emptiness is satisfied whenever the raw terms are definable as datatype in Isabelle/HOL and our relation for $\alpha$-equivalence is an equivalence relation).

The problem with introducing a new type in Isabelle/HOL is that in order to be useful, a reasoning infrastructure needs to be "lifted" from the underlying subset to the new type. This is usually a tricky and arduous task. To ease it, we re-implemented in Isabelle/HOL [8] the quotient package described by Homeier [6] for the HOL4 system. This package allows us to lift definitions and theorems involving raw terms to

definitions and theorems involving $\alpha$-equated terms. For example if we define the free-variable function over raw lambda-terms

$$fv(x) = \{x\} \qquad fv(t_1\ t_2) = fv(t_1) \cup fv(t_2) \qquad fv(\lambda x.t) = fv(t) - \{x\}$$

then with the help of the quotient package we can obtain a function $fv^\alpha$ operating on quotients, or $\alpha$-equivalence classes of lambda-terms. This lifted function is characterised by the equations

$$fv^\alpha(x) = \{x\} \qquad fv^\alpha(t_1\ t_2) = fv^\alpha(t_1) \cup fv^\alpha(t_2) \qquad fv^\alpha(\lambda x.t) = fv^\alpha(t) - \{x\}$$

(Note that this means also the term-constructors for variables, applications and lambda are lifted to the quotient level.) This construction, of course, only works if $\alpha$-equivalence is indeed an equivalence relation, and the "raw" definitions and theorems are respectful w.r.t. $\alpha$-equivalence. To sum up, every lifting of theorems to the quotient level needs proofs of some respectfulness properties (see [6]). In the paper we show that we are able to automate these proofs and as a result can automatically establish a reasoning infrastructure for $\alpha$-equated terms.

**Contributions:** We provide three new definitions for when terms involving general binders are $\alpha$-equivalent. These definitions are inspired by earlier work of Pitts [11]. By means of automatic proofs, we establish a reasoning infrastructure for $\alpha$-equated terms, including properties about support, freshness and equality conditions for $\alpha$-equated terms. We are also able to derive strong induction principles that have the variable convention already built in. The method behind our specification of general binders is taken from the Ott-tool, but we introduce crucial restrictions, and also extensions, so that our specifications make sense for reasoning about $\alpha$-equated terms. The main improvement over Ott is that we introduce three binding modes (only one is present in Ott), provide formalised definitions for $\alpha$-equivalence and for free variables of our terms, and also derive a reasoning infrastructure for our specifications from "first principles".

## 2 A Short Review of the Nominal Logic Work

At its core, Nominal Isabelle is an adaption of the nominal logic work by Pitts [12]. This adaptation for Isabelle/HOL is described in [7] (including proofs). We shall briefly review this work to aid the description of what follows.

Two central notions in the nominal logic work are sorted atoms and sort-respecting permutations of atoms. We will use the letters $a$, $b$, $c$, ... to stand for atoms and $p$, $q$, ... to stand for permutations. The purpose of atoms is to represent variables, be they bound or free. It is assumed that there is an infinite supply of atoms for each sort. In the interest of brevity, we shall restrict ourselves in what follows to only one sort of atoms.

Permutations are bijective functions from atoms to atoms that are the identity everywhere except on a finite number of atoms. There is a two-place permutation operation written $\_ \bullet \_ :: perm \Rightarrow \beta \Rightarrow \beta$ where the generic type $\beta$ is the type of the object over which the permutation acts. In Nominal Isabelle, the identity permutation is written as $0$, the composition of two permutations $p$ and $q$ as $p + q$, and the inverse permutation of $p$ as $-p$. The permutation operation is defined over the type-hierarchy [7]; for example permutations acting on products, lists, sets, functions and booleans are given by:

$p \bullet (x, y) \stackrel{def}{=} (p \bullet x, p \bullet y)$ $\quad p \bullet [] \stackrel{def}{=} []$ $\quad p \bullet X \stackrel{def}{=} \{p \bullet x \mid x \in X\}$
$p \bullet b \stackrel{def}{=} b$ $\quad p \bullet (x::xs) \stackrel{def}{=} (p \bullet x)::(p \bullet xs)$ $\quad p \bullet f \stackrel{def}{=} \lambda x.\, p \bullet (f\ (-p \bullet x))$

Concrete permutations in Nominal Isabelle are built up from swappings, written as $(a\ b)$, which are permutations that behave as follows:

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

The most original aspect of the nominal logic work of Pitts is a general definition for the notion of the "set of free variables of an object $x$". This notion, written *supp* $x$, is general in the sense that it applies not only to lambda-terms ($\alpha$-equated or not), 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:

$$\mathit{supp}\ x \stackrel{def}{=} \{a \mid \mathit{infinite}\ \{b \mid (a\ b) \bullet x \neq x\}\} \tag{6}$$

There is also the derived notion for when an atom $a$ is *fresh* for an $x$, defined as $a \# x \stackrel{def}{=} a \notin \mathit{supp}\ x$. We use for sets of atoms the abbreviation $as \#^* x$, defined as $\forall a \in as.\ a \# x$. 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 $b$, leaves $x$ unchanged, namely if $a \# x$ and $b \# x$ then $(a\ b) \bullet x = x$. While in the older version of Nominal Isabelle, we used extensively this property to rename single binders, it proved too unwieldy for dealing with multiple binders. For such binders the following generalisations turned out to be easier to use.

**Property 1.** *If* $\mathit{supp}\ x \#^* p$ *then* $p \bullet x = x$.

**Property 2.** For a finite set $as$ and a finitely supported $x$ with $as \#^* x$ and also a finitely supported $c$, there exists a permutation $p$ such that $p \bullet as \#^* c$ and $\mathit{supp}\ x \#^* p$.

The idea behind the second property is that given a finite set $as$ of binders (being bound, or fresh, in $x$ is ensured by the assumption $as \#^* x$), then there exists a permutation $p$ such that the renamed binders $p \bullet as$ avoid $c$ (which can be arbitrarily chosen as long as it is finitely supported) and also $p$ does not affect anything in the support of $x$ (that is $\mathit{supp}\ x \#^* p$). The last fact and Property 1 allow us to "rename" just the binders $as$ in $x$, because $p \bullet x = x$.

Most properties given in this section are described in detail in [7] and all are formalised in Isabelle/HOL. In the next sections we will make extensive use of these properties in order to define $\alpha$-equivalence in the presence of multiple binders.

## 3 General Bindings

In Nominal Isabelle, the user is expected to write down a specification of a term-calculus and then a reasoning infrastructure is automatically derived from this specification (remember that Nominal Isabelle is a definitional extension of Isabelle/HOL, which does not introduce any new axioms).

In order to keep our work with deriving the reasoning infrastructure manageable, we will wherever possible state definitions and perform proofs on the "user-level" of

Isabelle/HOL, as opposed to write custom ML-code. To that end, we will consider first pairs $(as, x)$ of type $(\mathit{atom\ set}) \times \beta$. These pairs are intended to represent the abstraction, or binding, of the set of atoms $as$ in the body $x$.

The first question we have to answer is when two pairs $(as, x)$ and $(bs, y)$ are $\alpha$-equivalent? (For the moment we are interested in the notion of $\alpha$-equivalence that is *not* preserved by adding vacuous binders.) To answer this question, we identify four conditions: *(i)* given a free-atom function $fa$ of type $\beta \Rightarrow \mathit{atom\ set}$, then $x$ and $y$ need to have the same set of free atoms; moreover there must be a permutation $p$ such that *(ii)* $p$ leaves the free atoms of $x$ and $y$ unchanged, but *(iii)* "moves" their bound names so that we obtain modulo a relation, say $\_\ R\ \_$, two equivalent terms. We also require that *(iv)* $p$ makes the sets of abstracted atoms $as$ and $bs$ equal. The requirements *(i)* to *(iv)* can be stated formally as the conjunction of:

$$\begin{array}{l}(as, x) \approx^{R, fa, p}_{set} (bs, y) \stackrel{def}{=} \\ \quad \begin{array}{llll} (i) & fa\ x - as = fa\ y - bs & (iii) & (p \bullet x)\ R\ y \\ (ii) & fa\ x - as \#^* p & (iv) & p \bullet as = bs \end{array}\end{array} \tag{7}$$

Note that this relation depends on the permutation $p$; $\alpha$-equivalence between two pairs is then the relation where we existentially quantify over this $p$. Also note that the relation is dependent on a free-atom function $fa$ and a relation $R$. The reason for this extra generality is that we will use $\approx_{set}$ for both "raw" terms and $\alpha$-equated terms. In the latter case, $R$ will be replaced by equality $=$ and we will prove that $fa$ is equal to $supp$.

The definition in (7) does not make any distinction between the order of abstracted atoms. If we want this, then we can define $\alpha$-equivalence for pairs of the form $(as, x)$ with type $(\mathit{atom\ list}) \times \beta$ as follows

$$\begin{array}{l}(as, x) \approx^{R, fa, p}_{list} (bs, y) \stackrel{def}{=} \\ \quad \begin{array}{llll} (i) & fa\ x - set\ as = fa\ y - set\ bs & (iii) & (p \bullet x)\ R\ y \\ (ii) & fa\ x - set\ as \#^* p & (iv) & p \bullet as = bs \end{array}\end{array} \tag{8}$$

where $set$ is the function that coerces a list of atoms into a set of atoms. Now the last clause ensures that the order of the binders matters (since $as$ and $bs$ are lists of atoms).

If we do not want to make any difference between the order of binders *and* also allow vacuous binders, that means *restrict* names, then we keep sets of binders, but drop condition *(iv)* in (7):

$$\begin{array}{l}(as, x) \approx^{R, fa, p}_{set+} (bs, y) \stackrel{def}{=} \\ \quad \begin{array}{llll} (i) & fa\ x - as = fa\ y - bs & (iii) & (p \bullet x)\ R\ y \\ (ii) & fa\ x - as \#^* p & & \end{array}\end{array} \tag{9}$$

It might be useful to consider first some examples how these definitions of $\alpha$-equivalence pan out in practice. For this consider the case of abstracting a set of atoms over types (as in type-schemes). We set $R$ to be the usual equality $=$ and for $fa(T)$ we define

$$fa(x) = \{x\} \qquad fa(T_1 \to T_2) = fa(T_1) \cup fa(T_2)$$

Now recall the examples shown in (2) and (3). It can be easily checked that $(\{x, y\}, x \to y)$ and $(\{y, x\}, y \to x)$ are $\alpha$-equivalent according to $\approx_{set}$ and $\approx_{set+}$ by taking $p$ to be the swapping $(x\,y)$. In case of $x \neq y$, then $([x, y], x \to y) \not\approx_{list} ([y, x], x \to y)$ since there is no permutation that makes the lists $[x, y]$ and $[y, x]$ equal, and also leaves the type $x \to y$ unchanged. Another example is $(\{x\}, x) \approx_{set+} (\{x, y\}, x)$ which holds by taking $p$ to be the identity permutation. However, if $x \neq y$, then $(\{x\}, x) \not\approx_{set} (\{x, y\}, x)$ since there is no permutation that makes the sets $\{x\}$ and $\{x, y\}$ equal (similarly for $\approx_{list}$). It can also relatively easily be shown that all three notions of $\alpha$-equivalence coincide, if we only abstract a single atom.

In the rest of this section we are going to introduce three abstraction types. For this we define

$$(as, x) \approx_{abs\_set} (bs, x) \stackrel{def}{=} \exists p.\ (as, x) \approx^{=, supp, p}_{set} (bs, x) \tag{10}$$

(similarly for $\approx_{abs\_set+}$ and $\approx_{abs\_list}$). We can show that these relations are equivalence relations.

**Lemma 1.** *The relations $\approx_{abs\_set}$, $\approx_{abs\_list}$ and $\approx_{abs\_set+}$ are equivalence relations.*

*Proof.* Reflexivity is by taking $p$ to be $0$. For symmetry we have a permutation $p$ and for the proof obligation take $-p$. In case of transitivity, we have two permutations $p$ and $q$, and for the proof obligation use $q + p$. All conditions are then by simple calculations.

This lemma allows us to use our quotient package for introducing new types $\beta$ *abs_set*, $\beta$ *abs_set+* and $\beta$ *abs_list* representing $\alpha$-equivalence classes of pairs of type (*atom set*) $\times\ \beta$ (in the first two cases) and of type (*atom list*) $\times\ \beta$ (in the third case). The elements in these types will be, respectively, written as $[as]_{set}.x$, $[as]_{set+}.x$ and $[as]_{list}.x$, indicating that a set (or list) of atoms *as* is abstracted in $x$. We will call the types *abstraction types* and their elements *abstractions*. The important property we need to derive is the support of abstractions, namely:

**Theorem 1 (Support of Abstractions).** *Assuming $x$ has finite support, then*

$$\begin{array}{lcl} supp\ [as]_{set}.x & = & supp\ [as]_{set+}.x \ = \ supp\ x - as, \text{ and} \\ supp\ [bs]_{list}.x & = & supp\ x - set\ bs \end{array}$$

This theorem states that the bound names do not appear in the support. For brevity we omit the proof and again refer the reader to our formalisation in Isabelle/HOL.

The method of first considering abstractions of the form $[as]_{set}.x$ etc is motivated by the fact that we can conveniently establish at the Isabelle/HOL level properties about them. It would be laborious to write custom ML-code that derives automatically such properties for every term-constructor that binds some atoms. Also the generality of the definitions for $\alpha$-equivalence will help us in the next sections.

## 4 Specifying General Bindings

Our choice of syntax for specifications is influenced by the existing datatype package of Isabelle/HOL and by the syntax of the Ott-tool [16]. For us a specification of a term-calculus is a collection of (possibly mutual recursive) type declarations, say $ty_1^{\alpha}, \ldots,$

$ty_n^\alpha$, and an associated collection of binding functions, say $bn_1^\alpha, \ldots, bn_m^\alpha$. The syntax in Nominal Isabelle for such specifications is roughly as follows:

$$\begin{array}{ll} \left.\begin{array}{l} \textbf{nominal\_datatype}\ ty_1^\alpha = \ldots \\ \textbf{and}\ ty_2^\alpha = \ldots \\ \ldots \\ \textbf{and}\ ty_n^\alpha = \ldots \end{array}\right\} & \text{type declaration part} \\ \left.\begin{array}{l} \textbf{binder}\ bn_1^\alpha\ \textbf{and}\ \ldots\ \textbf{and}\ bn_m^\alpha \\ \textbf{where} \\ \ldots \end{array}\right\} & \text{binding function part} \end{array} \tag{11}$$

Every type declaration $ty_{1..n}^\alpha$ consists of a collection of term-constructors, each of which comes with a list of labelled types that stand for the types of the arguments of the term-constructor. For example a term-constructor $C^\alpha$ might be specified with

$$C^\alpha\ label_1{::}ty'_1\ \ldots\ label_l{::}ty'_l\quad binding\_clauses$$

whereby some of the $ty'_{1..l}$ can be contained in the collection of $ty_{1..n}^\alpha$ declared in (11). In this case we will call the corresponding argument a *recursive argument* of $C^\alpha$. The labels annotated on the types are optional. Their purpose is to be used in the (possibly empty) list of *binding clauses*, which indicate the binders and their scope in a term-constructor. They come in three *modes*:

**bind** *binders* **in** *bodies*  **bind (set)** *binders* **in** *bodies*  **bind (set+)** *binders* **in** *bodies*

The first mode is for binding lists of atoms (the order of binders matters); the second is for sets of binders (the order does not matter, but the cardinality does) and the last is for sets of binders (with vacuous binders preserving $\alpha$-equivalence). As indicated, the labels in the "**in**-part" of a binding clause will be called *bodies*; the "**bind**-part" will be called *binders*. In contrast to Ott, we allow multiple labels in binders and bodies.

There are also some restrictions we need to impose on our binding clauses in comparison to the ones of Ott. The main idea behind these restrictions is that we obtain a sensible notion of $\alpha$-equivalence where it is ensured that within a given scope an atom occurrence cannot be both bound and free at the same time. The first restriction is that a body can only occur in *one* binding clause of a term constructor (this ensures that the bound atoms of a body cannot be free at the same time by specifying an alternative binder for the same body).

For binders we distinguish between *shallow* and *deep* binders. Shallow binders are just labels. The restriction we need to impose on them is that in case of **bind (set)** and **bind (set+)** the labels must either refer to atom types or to sets of atom types; in case of **bind** the labels must refer to atom types or lists of atom types. Two examples for the use of shallow binders are the specification of lambda-terms, where a single name is bound, and type-schemes, where a finite set of names is bound:

**nominal_datatype** *lam* =
  *Var name*
| *App lam lam*
| *Lam x*::*name t*::*lam*  **bind** *x* **in** *t*

**nominal_datatype** *ty* =
  *TVar name*
| *TFun ty ty*
**and** *tsc* = *All xs*::(*name fset*) *T*::*ty*  **bind (set+)** *xs* **in** *T*

In these specifications *name* refers to an atom type, and *fset* to the type of finite sets. Note that for *lam* it does not matter which binding mode we use. The reason is that we bind only a single *name*. However, having **bind (set)** or **bind** in the second case makes a difference to the semantics of the specification (which we will define in the next section).

A *deep* binder uses an auxiliary binding function that "picks" out the atoms in one argument of the term-constructor, which can be bound in other arguments and also in the same argument (we will call such binders *recursive*, see below). The binding functions are expected to return either a set of atoms (for **bind (set)** and **bind (set+)**) or a list of atoms (for **bind**). They can be defined by recursion over the corresponding type; the equations must be given in the binding function part of the scheme shown in (11). For example a term-calculus containing *Let*s with tuple patterns might be specified as:

$$\begin{array}{l}
\textbf{nominal_datatype}\ trm = \\
\quad\quad Var\ name \\
\quad\mid App\ trm\ trm \\
\quad\mid Lam\ x{::}name\ t{::}trm \quad \textbf{bind}\ x\ \textbf{in}\ t \\
\quad\mid Let\ p{::}pat\ trm\ t{::}trm \quad \textbf{bind}\ bn(p)\ \textbf{in}\ t \\
\textbf{and}\ pat = PNil \mid PVar\ name \mid PTup\ pat\ pat \\
\textbf{binder}\ bn{::}pat \Rightarrow atom\ list \\
\textbf{where}\ bn(PNil) = [] \\
\quad\mid bn(PVar\ x) = [atom\ x] \\
\quad\mid bn(PTup\ p_1\ p_2) = bn(p_1)\ @\ bn(p_2)
\end{array} \tag{12}$$

In this specification the function *bn* determines which atoms of the pattern $p$ are bound in the argument $t$. Note that in the second-last *bn*-clause the function *atom* coerces a name into the generic atom type of Nominal Isabelle [7]. This allows us to treat binders of different atom type uniformly.

As said above, for deep binders we allow binding clauses such as *Bar* $p{::}pat\ t{::}trm$ **bind** $bn(p)$ **in** $p\ t$ where the argument of the deep binder also occurs in the body. We call such binders *recursive*. To see the purpose of such recursive binders, compare "plain" *Let*s and *Let_rec*s in the following specification:

$$\begin{array}{l}
\textbf{nominal_datatype}\ trm = \ldots \\
\quad\mid Let\ as{::}assn\ t{::}trm \quad \textbf{bind}\ bn(as)\ \textbf{in}\ t \\
\quad\mid Let_rec\ as{::}assn\ t{::}trm \quad \textbf{bind}\ bn(as)\ \textbf{in}\ as\ t \\
\textbf{and}\ assn = ANil \mid ACons\ name\ trm\ assn \\
\textbf{binder}\ bn{::}assn \Rightarrow atom\ list \\
\textbf{where}\ bn(ANil) = [] \\
\quad\mid bn(ACons\ a\ t\ as) = [atom\ a]\ @\ bn(as)
\end{array} \tag{13}$$

The difference is that with *Let* we only want to bind the atoms $bn(as)$ in the term $t$, but with *Let_rec* we also want to bind the atoms inside the assignment. This difference has consequences for the associated notions of free-atoms and $\alpha$-equivalence.

To make sure that atoms bound by deep binders cannot be free at the same time, we cannot have more than one binding function for a deep binder. Consequently we exclude specifications such as

$$\begin{array}{ll}
Baz_1\ p{::}pat\ t{::}trm & \textbf{bind}\ bn_1(p)\ bn_2(p)\ \textbf{in}\ t \\
Baz_2\ p{::}pat\ t_1{::}trm\ t_2{::}trm & \textbf{bind}\ bn_1(p)\ \textbf{in}\ t_1, \textbf{bind}\ bn_2(p)\ \textbf{in}\ t_2
\end{array}$$

Otherwise it is possible that $bn_1$ and $bn_2$ pick out different atoms to become bound, respectively be free, in $p$. (Since the Ott-tool does not derive a reasoning infrastructure for $\alpha$-equated terms with deep binders, it can permit such specifications.)

We also need to restrict the form of the binding functions in order to ensure the *bn*-functions can be defined for $\alpha$-equated terms. The main restriction is that we cannot return an atom in a binding function that is also bound in the corresponding term-constructor. That means in (12) that the term-constructors *PVar* and *PTup* may not have a binding clause (all arguments are used to define *bn*). In contrast, in case of (13) the term-constructor *ACons* may have a binding clause involving the argument *trm* (the only one that is *not* used in the definition of the binding function). This restriction is sufficient for lifting the binding function to $\alpha$-equated terms.

In the version of Nominal Isabelle described here, we also adopted the restriction from the Ott-tool that binding functions can only return: the empty set or empty list (as in case *PNil*), a singleton set or singleton list containing an atom (case *PVar*), or unions of atom sets or appended atom lists (case *PTup*). This restriction will simplify some automatic definitions and proofs later on.

In order to simplify our definitions of free atoms and $\alpha$-equivalence, we shall assume specifications of term-calculi are implicitly *completed*. By this we mean that for every argument of a term-constructor that is *not* already part of a binding clause given by the user, we add implicitly a special *empty* binding clause, written **bind** $\varnothing$ **in** *labels*. In case of the lambda-terms, the completion produces

**nominal_datatype** *lam* =
  *Var x*::*name*  **bind** $\varnothing$ **in** *x*
| *App* $t_1$::*lam* $t_2$::*lam*  **bind** $\varnothing$ **in** $t_1$ $t_2$
| *Lam x*::*name t*::*lam*  **bind** *x* **in** *t*

The point of completion is that we can make definitions over the binding clauses and be sure to have captured all arguments of a term constructor.

## 5 Alpha-Equivalence and Free Atoms

Having dealt with all syntax matters, the problem now is how we can turn specifications into actual type definitions in Isabelle/HOL and then establish a reasoning infrastructure for them. As Pottier and Cheney pointed out [13,5], just re-arranging the arguments of term-constructors so that binders and their bodies are next to each other will result in inadequate representations in cases like *Let* $x_1 = t_1 \ldots x_n = t_n$ *in s*. Therefore we will first extract "raw" datatype definitions from the specification and then define explicitly an $\alpha$-equivalence relation over them. We subsequently construct the quotient of the datatypes according to our $\alpha$-equivalence.

The "raw" datatype definition can be obtained by stripping off the binding clauses and the labels from the types. We also have to invent new names for the types $ty^\alpha$ and term-constructors $C^\alpha$ given by the user. In our implementation we just use the affix "*_raw*". But for the purpose of this paper, we use the superscript $\_^\alpha$ to indicate that a notion is given for $\alpha$-equivalence classes and leave it out for the corresponding notion given on the "raw" level. So for example we have $ty^\alpha \mapsto ty$ and $C^\alpha \mapsto C$ where *ty* is

the type used in the quotient construction for $ty^\alpha$ and $C$ is the term-constructor on the "raw" type $ty$.

We subsequently define each of the user-specified binding functions $bn_{1..m}$ by recursion over the corresponding raw datatype. We can also easily define permutation operations by recursion so that for each term constructor $C$ we have that

$$p \bullet (C\ z_1\ \ldots\ z_n) = C\ (p \bullet z_1)\ \ldots\ (p \bullet z_n) \tag{14}$$

The first non-trivial step we have to perform is the generation of free-atom functions from the specification. For the *raw* types $ty_{1..n}$ we define the free-atom functions $fa\_ty_{1..n}$ by recursion. We define these functions together with auxiliary free-atom functions for the binding functions. Given raw binding functions $bn_{1..m}$ we define $fa\_bn_{1..m}$. The reason for this setup is that in a deep binder not all atoms have to be bound, as we saw in the example with "plain" *Let*s. We need therefore a function that calculates those free atoms in a deep binder.

While the idea behind these free-atom functions is clear (they just collect all atoms that are not bound), because of our rather complicated binding mechanisms their definitions are somewhat involved. Given a term-constructor $C$ of type $ty$ and some associated binding clauses $bc_1 \ldots bc_k$, the result of $fa\_ty\ (C\ z_1\ \ldots\ z_n)$ will be the union $fa(bc_1) \cup \ldots \cup fa(bc_k)$ where we will define below what $fa$ for a binding clause means. We only show the details for the mode **bind (set)** (the other modes are similar). Suppose the binding clause $bc_i$ is of the form **bind (set)** $b_1 \ldots b_p$ **in** $d_1 \ldots d_q$ in which the body-labels $d_{1..q}$ refer to types $ty_{1..q}$, and the binders $b_{1..p}$ either refer to labels of atom types (in case of shallow binders) or to binding functions taking a single label as argument (in case of deep binders). Assuming $D$ stands for the set of free atoms of the bodies, $B$ for the set of binding atoms in the binders and $B'$ for the set of free atoms in non-recursive deep binders, then the free atoms of the binding clause $bc_i$ are

$$fa(bc_i) \stackrel{def}{=} (D - B) \cup B'. \tag{15}$$

The set $D$ is formally defined as $D \stackrel{def}{=} fa\_ty_1\ d_1 \cup \ldots \cup fa\_ty_q\ d_q$ where in case $d_i$ refers to one of the raw types $ty_{1..n}$ from the specification, the function $fa\_ty_i$ is the corresponding free-atom function we are defining by recursion; otherwise we set $fa\_ty_i\ d_i = supp\ d_i$.

In order to formally define the set $B$ we use the following auxiliary *bn*-functions for atom types to which shallow binders may refer

$$bn_{atom}\ a \stackrel{def}{=} \{atom\ a\} \quad bn_{atom\_set}\ as \stackrel{def}{=} atoms\ as \quad bn_{atom\_list}\ as \stackrel{def}{=} atoms\ (set\ as)$$

Like the function *atom*, the function *atoms* coerces a set of atoms to a set of the generic atom type. The set $B$ is then formally defined as

$$B \stackrel{def}{=} bn\_ty_1\ b_1 \cup \ldots \cup bn\_ty_p\ b_p$$

where we use the auxiliary binding functions for shallow binders. The set $B'$ collects all free atoms in non-recursive deep binders. Let us assume these binders in $bc_i$ are $bn_1\ l_1, \ldots, bn_r\ l_r$ with $l_{1..r} \subseteq b_{1..p}$ and none of the $l_{1..r}$ being among the bodies $d_{1..q}$. The set $B'$ is defined as

$$B' \stackrel{def}{=} fa\_bn_1\ l_1 \cup \ldots \cup fa\_bn_r\ l_r$$

This completes the definition of the free-atom functions $fa\_ty_{1..n}$.

Note that for non-recursive deep binders, we have to add in (15) the set of atoms that are left unbound by the binding functions $bn_{1..m}$. We used for the definition of this set the functions $fa\_bn_{1..m}$, which are also defined by mutual recursion. Assume the user specified a *bn*-clause of the form $bn\ (C\ z_1\ \ldots\ z_s) = rhs$ where the $z_{1..s}$ are of types $ty_{1..s}$. For each of the arguments we calculate the free atoms as follows:

- $fa\_ty_i\ z_i$ provided $z_i$ does not occur in *rhs* (that means nothing is bound in $z_i$ by the binding function),
- $fa\_bn_i\ z_i$ provided $z_i$ occurs in *rhs* with the recursive call $bn_i\ z_i$, and
- $\varnothing$ provided $z_i$ occurs in *rhs*, but without a recursive call.

For defining $fa\_bn\ (C\ z_1\ \ldots\ z_n)$ we just union up all these sets.

To see how these definitions work in practice, let us reconsider the term-constructors *Let* and *Let_rec* shown in (13) together with the term-constructors for assignments *ANil* and *ACons*. Since there is a binding function defined for assignments, we have three free-atom functions, namely $fa_{trm}$, $fa_{assn}$ and $fa_{bn}$ as follows:

$$\begin{array}{ll}
fa_{trm}\ (Let\ as\ t) & = (fa_{trm}\ t - set\ (bn\ as)) \cup fa_{bn}\ as \\
fa_{trm}\ (Let_rec\ as\ t) & = (fa_{assn}\ as \cup fa_{trm}\ t) - set\ (bn\ as) \\
fa_{assn}\ (ANil) & = \varnothing \\
fa_{assn}\ (ACons\ a\ t\ as) & = (supp\ a) \cup (fa_{trm}\ t) \cup (fa_{assn}\ as) \\
fa_{bn}\ (ANil) & = \varnothing \\
fa_{bn}\ (ACons\ a\ t\ as) & = (fa_{trm}\ t) \cup (fa_{bn}\ as)
\end{array}$$

Recall that *ANil* and *ACons* have no binding clause in the specification. The corresponding free-atom function $fa_{assn}$ therefore returns all free atoms of an assignment (in case of *ACons*, they are given in terms of *supp*, $fa_{trm}$ and $fa_{assn}$). The binding only takes place in *Let* and *Let_rec*. In case of *Let*, the binding clause specifies that all atoms given by $set\ (bn\ as)$ have to be bound in *t*. Therefore we have to subtract $set\ (bn\ as)$ from $fa_{trm}\ t$. However, we also need to add all atoms that are free in *as*. This is in contrast with *Let_rec* where we have a recursive binder to bind all occurrences of the atoms in $set\ (bn\ as)$ also inside *as*. Therefore we have to subtract $set\ (bn\ as)$ from both $fa_{trm}\ t$ and $fa_{assn}\ as$.

An interesting point in this example is that a "naked" assignment (*ANil* or *ACons*) does not bind any atoms, even if the binding function is specified over assignments. Only in the context of a *Let* or *Let_rec*, where the binding clauses are given, will some atoms actually become bound. This is a phenomenon that has also been pointed out in [16]. For us this observation is crucial, because we would not be able to lift the *bn*-functions to $\alpha$-equated terms if they act on atoms that are bound. In that case, these functions would *not* respect $\alpha$-equivalence.

Next we define the $\alpha$-equivalence relations for the raw types $ty_{1..n}$ from the specification. We write them as $\approx ty_{1..n}$. Like with the free-atom functions, we also need to define auxiliary $\alpha$-equivalence relations $\approx bn_{1..m}$ for the binding functions $bn_{1..m}$, To simplify our definitions we will use the following abbreviations for *compound equivalence relations* and *compound free-atom functions* acting on tuples.

$$\begin{array}{rcl}
(x_1,\ldots,x_n)\ (R_1,\ldots,R_n)\ (x'_1,\ldots,x'_n) & \stackrel{def}{=} & x_1\ R_1\ x'_1 \wedge \ldots \wedge x_n\ R_n\ x'_n \\
(fa_1,\ldots,fa_n)\ (x_1,\ldots,x_n) & \stackrel{def}{=} & fa_1\ x_1 \cup \ldots \cup fa_n\ x_n
\end{array}$$

The $\alpha$-equivalence relations are defined as inductive predicates having a single clause for each term-constructor. Assuming a term-constructor $C$ is of type $ty$ and has the binding clauses $bc_{1..k}$, then the $\alpha$-equivalence clause has the form

$$\frac{prems(bc_1) \ldots prems(bc_k)}{C\ z_1 \ldots z_n \approx ty\ C\ z'_1 \ldots z'_n}$$

The task below is to specify what the premises of a binding clause are. As a special instance, we first treat the case where $bc_i$ is the empty binding clause of the form

$$\textbf{bind (set)}\ \varnothing\ \textbf{in}\ d_1 \ldots d_q.$$

In this binding clause no atom is bound and we only have to $\alpha$-relate the bodies. For this we build first the tuples $D \stackrel{def}{=} (d_1, \ldots, d_q)$ and $D' \stackrel{def}{=} (d'_1, \ldots, d'_q)$ whereby the labels $d_{1..q}$ refer to arguments $z_{1..n}$ and respectively $d'_{1..q}$ to $z'_{1..n}$. In order to relate two such tuples we define the compound $\alpha$-equivalence relation $R$ as follows

$$R \stackrel{def}{=} (R_1, \ldots, R_q) \tag{16}$$

with $R_i$ being $\approx ty_i$ if the corresponding labels $d_i$ and $d'_i$ refer to a recursive argument of $C$ with type $ty_i$; otherwise we take $R_i$ to be the equality $=$. This lets us define the premise for an empty binding clause succinctly as $prems(bc_i) \stackrel{def}{=} D\ R\ D'$, which can be unfolded to the series of premises $d_1\ R_1\ d'_1\ \ldots\ d_q\ R_q\ d'_q$. We will use the unfolded version in the examples below.

Now suppose the binding clause $bc_i$ is of the general form

$$\textbf{bind (set)}\ b_1 \ldots b_p\ \textbf{in}\ d_1 \ldots d_q. \tag{17}$$

In this case we define a premise $P$ using the relation $\approx_{set}$ given in Section 3 (similarly $\approx_{set+}$ and $\approx_{list}$ for the other binding modes). This premise defines $\alpha$-equivalence of two abstractions involving multiple binders. As above, we first build the tuples $D$ and $D'$ for the bodies $d_{1..q}$, and the corresponding compound $\alpha$-relation $R$ (shown in (16)). For $\approx_{set}$ we also need a compound free-atom function for the bodies defined as

$$fa \stackrel{def}{=} (fa\_ty_1, \ldots, fa\_ty_q)$$

with the assumption that the $d_{1..q}$ refer to arguments of types $ty_{1..q}$. The last ingredient we need are the sets of atoms bound in the bodies. For this we take

$$B \stackrel{def}{=} bn\_ty_1\ b_1 \cup \ldots \cup bn\_ty_p\ b_p\ .$$

Similarly for $B'$ using the labels $b'_{1..p}$. This lets us formally define the premise $P$ for a non-empty binding clause as:

$$P \stackrel{def}{=} \exists p.\ (B, D) \approx_{set}^{R, fa, p} (B', D')\ .$$

This premise accounts for $\alpha$-equivalence of the bodies of the binding clause. However, in case the binders have non-recursive deep binders, this premise is not enough: we also have to "propagate" $\alpha$-equivalence inside the structure of these binders. An example is *Let* where we have to make sure the right-hand sides of assignments are $\alpha$-equivalent.

For this we use relations $\approx bn_{1..m}$ (which we will formally define shortly). Let us assume the non-recursive deep binders in $bc_i$ are $bn_1\ l_1, \ldots, bn_r\ l_r$. The tuple $L$ is then $(l_1,\ldots,l_r)$ (similarly $L'$) and the compound equivalence relation $R'$ is $(\approx bn_1,\ldots,\approx bn_r)$. All premises for $bc_i$ are then given by

$$prems(bc_i) \stackrel{def}{=} P \ \wedge \ L\,R'\,L'$$

The auxiliary $\alpha$-equivalence relations $\approx bn_{1..m}$ in $R'$ are defined as follows: assuming a *bn*-clause is of the form $bn\ (C\ z_1\ \ldots\ z_s) = rhs$ where the $z_{1..s}$ are of types $ty_{1..s}$, then the corresponding $\alpha$-equivalence clause for $\approx bn$ has the form

$$\frac{z_1\ R_1\ z_1'\ \ldots\ z_s\ R_s\ z_s'}{C\ z_1\ \ldots\ z_s \approx bn\ C\ z_1'\ \ldots\ z_s'}$$

In this clause the relations $R_{1..s}$ are given by

- $z_i \approx ty\ z_i'$ provided $z_i$ does not occur in *rhs* and is a recursive argument of $C$,
- $z_i = z_i'$ provided $z_i$ does not occur in *rhs* and is a non-recursive argument of $C$,
- $z_i \approx bn_i\ z_i'$ provided $z_i$ occurs in *rhs* with the recursive call $bn_i\ x_i$ and
- *True* provided $z_i$ occurs in *rhs* but without a recursive call.

This completes the definition of $\alpha$-equivalence. As a sanity check, we can show that the premises of empty binding clauses are a special case of the clauses for non-empty ones (we just have to unfold the definition of $\approx_{set}$ and take *0* for the existentially quantified permutation).

Again let us take a look at a concrete example for these definitions. For (13) we have three relations $\approx_{trm}$, $\approx_{assn}$ and $\approx_{bn}$ with the following clauses:

$$\frac{\exists p.\ (bn\ as, t) \approx \stackrel{\approx}{list}trm, fa_{trm}, p\ (bn\ as', t') \quad as \approx_{bn} as'}{Let\ as\ t \approx_{trm} Let\ as'\ t'}$$

$$\frac{\exists p.\ (bn\ as, (as, t)) \approx \stackrel{(\approx}{list}assn, \approx_{trm}), (fa_{assn}, fa_{trm}), p\ (bn\ as', (as, t'))}{Let\_rec\ as\ t \approx_{trm} Let\_rec\ as'\ t'}$$

$$\frac{}{ANil \approx_{assn} ANil} \qquad \frac{a = a' \quad t \approx_{trm} t' \quad as \approx_{assn} as'}{ACons\ a\ t\ as \approx_{assn} ACons\ a'\ t'\ as}$$

$$\frac{}{ANil \approx_{bn} ANil} \qquad \frac{t \approx_{trm} t' \quad as \approx_{bn} as'}{ACons\ a\ t\ as \approx_{bn} ACons\ a'\ t'\ as}$$

Note the difference between $\approx_{assn}$ and $\approx_{bn}$: the latter only "tracks" $\alpha$-equivalence of the components in an assignment that are *not* bound. This is needed in the clause for *Let* (which has a non-recursive binder).

## 6 Establishing the Reasoning Infrastructure

Having made all necessary definitions for raw terms, we can start with establishing the reasoning infrastructure for the $\alpha$-equated types $ty^{\alpha}_{1..n}$, that is the types the user

originally specified. We sketch in this section the proofs we need for establishing this infrastructure. One main point of our work is that we have completely automated these proofs in Isabelle/HOL.

First we establish that the $\alpha$-equivalence relations defined in the previous section are equivalence relations.

**Lemma 2.** *Given the raw types $ty_{1..n}$ and binding functions $bn_{1..m}$, the relations $\approx ty_{1..n}$ and $\approx bn_{1..m}$ are equivalence relations.*

*Proof.* The proof is by mutual induction over the definitions. The non-trivial cases involve premises built up by $\approx_{set}$, $\approx_{set+}$ and $\approx_{list}$. They can be dealt with as in Lemma 1.

We can feed this lemma into our quotient package and obtain new types $ty^{\alpha}_{1..n}$ representing $\alpha$-equated terms of types $ty_{1..n}$. We also obtain definitions for the term-constructors $C^{\alpha}_{1..k}$ from the raw term-constructors $C_{1..k}$, and similar definitions for the free-atom functions $fa\_ty^{\alpha}_{1..n}$ and $fa\_bn^{\alpha}_{1..m}$ as well as the binding functions $bn^{\alpha}_{1..m}$. However, these definitions are not really useful to the user, since they are given in terms of the isomorphisms we obtained by creating new types in Isabelle/HOL (recall the picture shown in the Introduction).

The first useful property for the user is the fact that distinct term-constructors are not equal, that is

$$C^{\alpha}\ x_1\ \ldots\ x_r \neq D^{\alpha}\ y_1\ \ldots\ y_s \tag{18}$$

whenever $C^{\alpha} \neq D^{\alpha}$. In order to derive this fact, we use the definition of $\alpha$-equivalence and establish that

$$C\ x_1\ \ldots\ x_r \not\approx ty\ D\ y_1\ \ldots\ y_s \tag{19}$$

holds for the corresponding raw term-constructors. In order to deduce (18) from (19), our quotient package needs to know that the raw term-constructors $C$ and $D$ are *respectful* w.r.t. the $\alpha$-equivalence relations (see [6]). Assuming, for example, $C$ is of type $ty$ with argument types $ty_{1..r}$, respectfulness amounts to showing that

$$C\ x_1\ \ldots\ x_r \approx ty\ C\ x'_1\ \ldots\ x'_r$$

holds under the assumptions that we have $x_i \approx ty_i\ x'_i$ whenever $x_i$ and $x'_i$ are recursive arguments of $C$ and $x_i = x'_i$ whenever they are non-recursive arguments. We can prove this implication by applying the corresponding rule in our $\alpha$-equivalence definition and by establishing the following auxiliary implications

$$\begin{array}{ll} (i)\ x \approx ty_i\ x' \Rightarrow fa\_ty_i\ x = fa\_ty_i\ x' & (iii)\ x \approx ty_j\ x' \Rightarrow bn_j\ x = bn_j\ x' \\ (ii)\ x \approx ty_j\ x' \Rightarrow fa\_bn_j\ x = fa\_bn_j\ x' & (iv)\ x \approx ty_j\ x' \Rightarrow x \approx bn_j\ x' \end{array} \tag{20}$$

They can be established by induction on $\approx ty_{1..n}$. Whereas the first, second and last implication are true by how we stated our definitions, the third *only* holds because of our restriction imposed on the form of the binding functions—namely *not* returning any bound atoms. In Ott, in contrast, the user may define $bn_{1..m}$ so that they return bound atoms and in this case the third implication is *not* true. A result is that the lifing of the corresponding binding functions in Ott to $\alpha$-equated terms is impossible.

Having established respectfulness for the raw term-constructors, the quotient package is able to automatically deduce (18) from (19). Having the facts (20) at our disposal, we can also lift properties that characterise when two raw terms of the form

$$C\ x_1\ \ldots\ x_r \approx ty\ C\ x'_1\ \ldots\ x'_r$$

are $\alpha$-equivalent. This gives us conditions when the corresponding $\alpha$-equated terms are *equal*, namely $C^\alpha\ x_1\ \ldots\ x_r = C^\alpha\ x'_1\ \ldots\ x'_r$. We call these conditions as *quasi-injectivity*. They correspond to the premises in our $\alpha$-equivalence relations.

Next we can lift the permutation operations defined in (14). In order to make this lifting to go through, we have to show that the permutation operations are respectful. This amounts to showing that the $\alpha$-equivalence relations are equivariant [7]. As a result we can add the equations

$$p \bullet (C^\alpha\ x_1\ \ldots\ x_r) = C^\alpha\ (p \bullet x_1)\ \ldots\ (p \bullet x_r) \tag{21}$$

to our infrastructure. In a similar fashion we can lift the defining equations of the free-atom functions $fn\_ty^\alpha_{1..n}$ and $fa\_bn^\alpha_{1..m}$ as well as of the binding functions $bn^\alpha_{1..m}$ and the size functions $size\_ty^\alpha_{1..n}$. The latter are defined automatically for the raw types $ty_{1..n}$ by the datatype package of Isabelle/HOL.

Finally we can add to our infrastructure a cases lemma (explained in the next section) and a structural induction principle for the types $ty^\alpha_{1..n}$. The conclusion of the induction principle is of the form $P_1\ x_1 \wedge \ldots \wedge P_n\ x_n$ whereby the $P_{1..n}$ are predicates and the $x_{1..n}$ have types $ty^\alpha_{1..n}$. This induction principle has for each term constructor $C^\alpha$ a premise of the form

$$\forall x_1 \ldots x_r.\ P_i\ x_i \wedge \ldots \wedge P_j\ x_j \Rightarrow P\ (C^\alpha\ x_1\ \ldots\ x_r) \tag{22}$$

in which the $x_{i..j} \subseteq x_{1..r}$ are the recursive arguments of $C^\alpha$.

By working now completely on the $\alpha$-equated level, we can first show that the free-atom functions and binding functions are equivariant, namely

$$p \bullet (fa\_ty^\alpha_i\ x) = fa\_ty^\alpha_i (p \bullet x) \qquad p \bullet (bn^\alpha_j\ x) = bn^\alpha_j\ (p \bullet x)$$
$$p \bullet (fa\_bn^\alpha_j\ x) = fa\_bn^\alpha_j\ (p \bullet x)$$

These properties can be established using the induction principle for the types $ty^\alpha_{1..n}$. Having these equivariant properties established, we can show that the support of term-constructors $C^\alpha$ is included in the support of its arguments, that means

$$supp\ (C^\alpha\ x_1\ \ldots\ x_r) \subseteq (supp\ x_1 \cup \ldots \cup supp\ x_r)$$

holds. This allows us to prove by induction that every $x$ of type $ty^\alpha_{1..n}$ is finitely supported. Lastly, we can show that the support of elements in $ty^\alpha_{1..n}$ is the same as $fa\_ty^\alpha_{1..n}$. This fact is important in a nominal setting, but also provides evidence that our notions of free-atoms and $\alpha$-equivalence are correct.

**Theorem 2.** *For $x_{1..n}$ with type $ty^\alpha_{1..n}$, we have $supp\ x_i = fa\_ty^\alpha_i x_i$.*

*Proof.* The proof is by induction. In each case we unfold the definition of *supp*, move the swapping inside the term-constructors and then use the quasi-injectivity lemmas in order to complete the proof. For the abstraction cases we use the facts derived in Theorem 1.

To sum up this section, we can establish automatically a reasoning infrastructure for the types $ty^\alpha_{1..n}$ by first lifting definitions from the raw level to the quotient level and then by establishing facts about these lifted definitions. All necessary proofs are generated automatically by custom ML-code.

## 7 Strong Induction Principles

In the previous section we derived induction principles for $\alpha$-equated terms. We call such induction principles *weak*, because for a term-constructor $C^\alpha\ x_1 \ldots x_r$ the induction hypothesis requires us to establish the implications (22). The problem with these implications is that in general they are difficult to establish. The reason is that we cannot make any assumption about the bound atoms that might be in $C^\alpha$.

In [20] we introduced a method for automatically strengthening weak induction principles for terms containing single binders. These stronger induction principles allow the user to make additional assumptions about bound atoms. To sketch how this strengthening extends to the case of multiple binders, we use as running example the term-constructors *Lam* and *Let* from example (12). Instead of establishing $P_{trm}\ t \wedge P_{pat}\ p$, the stronger induction principle for (12) establishes properties $P_{trm}\ c\ t \wedge P_{pat}\ c\ p$ where the additional parameter $c$ controls which freshness assumptions the binders should satisfy. For the two term constructors this means that the user has to establish in inductions the implications

$$\forall a\ t\ c.\ \{atom\ a\}\ \#^*\ c \wedge (\forall d.\ P_{trm}\ d\ t) \Rightarrow P_{trm}\ c\ (Lam\ a\ t)$$
$$\forall p\ t\ c.\ (set\ (bn\ p))\ \#^*\ c \wedge (\forall d.\ P_{pat}\ d\ p) \wedge (\forall d.\ P_{trm}\ d\ t) \wedge \Rightarrow P_{trm}\ c\ (Let\ p\ t)$$

In [20] we showed how the weaker induction principles imply the stronger ones. This was done by some quite complicated, nevertheless automated, induction proof. In this paper we simplify this work by leveraging the automated proof methods from the function package of Isabelle/HOL. The reasoning principle these methods employ is well-founded induction. To use them in our setting, we have to discharge two proof obligations: one is that we have well-founded measures (for each type $ty^\alpha_{1..n}$) that decrease in every induction step and the other is that we have covered all cases. As measures we use the size functions $size\_ty^\alpha_{1..n}$, which we lifted in the previous section and which are all well-founded.

What is left to show is that we covered all cases. To do so, we use a *cases lemma* derived for each type. For the terms in (12) this lemma is of the form

$$\frac{\begin{array}{ll}\forall x.\ t = Var\ x \Rightarrow P_{trm} & \forall a\ t'.\ t = Lam\ a\ t' \Rightarrow P_{trm} \\ \forall t_1\ t_2.\ t = App\ t_1\ t_2 \Rightarrow P_{trm} & \forall p\ t'.\ t = Let\ p\ t' \Rightarrow P_{trm}\end{array}}{P_{trm}} \tag{23}$$

where we have a premise for each term-constructor. The idea behind such cases lemmas is that we can conclude with a property $P_{trm}$, provided we can show that this property holds if we substitute for $t$ all possible term-constructors.

The only remaining difficulty is that in order to derive the stronger induction principles conveniently, the cases lemma in (23) is too weak. For this note that in order to apply this lemma, we have to establish $P_{trm}$ for *all Lam*- and *all Let*-terms. What we need instead is a cases lemma where we only have to consider terms that have binders that are fresh w.r.t. a context $c$. This gives the implications

$$\forall a\ t'.\ t = Lam\ a\ t' \wedge \{atom\ a\}\ \#^*\ c \Rightarrow P_{trm}$$
$$\forall p\ t'.\ t = Let\ p\ t' \wedge (set\ (bn\ p))\ \#^*\ c \Rightarrow P_{trm}$$

which however can be relatively easily be derived from the implications in (23) by a renaming using Properties 1 and 2. In the first case we know that $\{atom\ a\}\ \#^*\ Lam\ a$

$t$. Property (2) provides us therefore with a permutation $q$, such that $\{atom\ (q \bullet a)\}\ \#^*\ c$ and $supp\ (Lam\ a\ t)\ \#^*\ q$ hold. By using Property 1, we can infer from the latter that $Lam\ (q \bullet a)\ (q \bullet t) = Lam\ a\ t$ and we are done with this case.

The *Let*-case involving a (non-recursive) deep binder is a bit more complicated. The reason is that the we cannot apply Property 2 to the whole term $Let\ p\ t$, because $p$ might contain names that are bound (by $bn$) and so are free. To solve this problem we have to introduce a permutation function that only permutes names bound by $bn$ and leaves the other names unchanged. We do this again by lifting. For a clause $bn\ (C\ x_1\ \ldots\ x_r) = rhs$, we define

$$p \bullet_{bn} (C\ x_1\ \ldots\ x_r) \stackrel{def}{=} C\ y_1\ \ldots\ y_r \text{ with } \begin{cases} y_i \stackrel{def}{=} x_i \text{ provided } x_i \text{ does not occur in } rhs \\ y_i \stackrel{def}{=} p \bullet_{bn'} x_i \text{ provided } bn'\ x_i \text{ is in } rhs \\ y_i \stackrel{def}{=} p \bullet x_i \text{ otherwise} \end{cases}$$

Now Properties 1 and 2 give us a permutation $q$ such that $(set\ (bn\ (q \bullet_{bn} p))\ \#^*\ c$ holds and such that $[q \bullet_{bn} p]_{list}.(q \bullet t)$ is equal to $[p]_{list}.\ t$. We can also show that $(q \bullet_{bn} p) \approx_{bn} p$. These facts establish that $Let\ (q \bullet_{bn} p)\ (p \bullet t) = Let\ p\ t$, as we need. This completes the non-trivial cases in (12) for strengthening the corresponding induction principle.

## 8 Related Work

To our knowledge the earliest usage of general binders in a theorem prover is described in [10] about a formalisation of the algorithm W. This formalisation implements binding in type-schemes using a de-Bruijn indices representation. Since type-schemes in W contain only a single place where variables are bound, different indices do not refer to different binders (as in the usual de-Bruijn representation), but to different bound variables. A similar idea has been recently explored for general binders in the locally nameless approach to binding [3]. There, de-Bruijn indices consist of two numbers, one referring to the place where a variable is bound, and the other to which variable is bound. The reasoning infrastructure for both representations of bindings comes for free in theorem provers like Isabelle/HOL or Coq, since the corresponding term-calculi can be implemented as "normal" datatypes. However, in both approaches it seems difficult to achieve our fine-grained control over the "semantics" of bindings (i.e. whether the order of binders should matter, or vacuous binders should be taken into account).

Another technique for representing binding is higher-order abstract syntax (HOAS). This technique supports very elegantly many aspects of *single* binding, and impressive work has been done that uses HOAS for mechanising the metatheory of SML [9]. We are, however, not aware how multiple binders of SML are represented in this work. Judging from the submitted Twelf-solution for the POPLmark challenge, HOAS cannot easily deal with binding constructs where the number of bound variables is not fixed. In the second part of this challenge, *Let*s involve patterns that bind multiple variables at once. In such situations, HOAS seems to have to resort to the iterated-single-binders-approach with all the unwanted consequences when reasoning about the resulting terms.

The most closely related work to the one presented here is the Ott-tool [16] and the C$\alpha$ml language [13]. Ott is a nifty front-end for creating LaTeX documents from specifications of term-calculi involving general binders. For a subset of the specifications Ott

can also generate theorem prover code using a raw representation of terms, and in Coq also a locally nameless representation. The developers of this tool have also put forward (on paper) a definition for $\alpha$-equivalence of terms that can be specified in Ott. This definition is rather different from ours, not using any nominal techniques. To our knowledge there is no concrete mathematical result concerning this notion of $\alpha$-equivalence. Also the definition for the notion of free variables is work in progress.

Although we were heavily inspired by the syntax of Ott, its definition of $\alpha$-equivalence is unsuitable for our extension of Nominal Isabelle. First, it is far too complicated to be a basis for automated proofs implemented on the ML-level of Isabelle/HOL. Second, it covers cases of binders depending on other binders, which just do not make sense for our $\alpha$-equated terms. Third, it allows empty types that have no meaning in a HOL-based theorem prover. We also had to generalise slightly Ott's binding clauses. In Ott you specify binding clauses with a single body; we allow more than one. We have to do this, because this makes a difference for our notion of $\alpha$-equivalence in case of **bind (set)** and **bind (set+)**. Because of how we set up our definitions, we also had to impose some restrictions (like a single binding function for a deep binder) that are not present in Ott.

Pottier presents in [13] a language, called C$\alpha$ml, for representing terms with general binders inside OCaml. This language is implemented as a front-end that can be translated to OCaml with the help of a library. He presents a type-system in which the scope of general binders can be specified using special markers, written *inner* and *outer*. It seems our and his specifications can be inter-translated as long as ours use the binding mode **bind** only. However, we have not proved this. Pottier gives a definition for $\alpha$-equivalence, which also uses a permutation operation (like ours). Still, this definition is rather different from ours and he only proves that it defines an equivalence relation. A complete reasoning infrastructure is well beyond the purposes of his language. Similar work for Haskell with similar results was reported by Cheney [4].

In a slightly different domain (programming with dependent types), the paper [1] presents a calculus with a notion of $\alpha$-equivalence related to our binding mode **bind (set+)**. The definition in [1] is similar to the one by Pottier, except that it has a more operational flavour and calculates a partial (renaming) map. In this way, the definition can deal with vacuous binders. However, to our best knowledge, no concrete mathematical result concerning this definition of $\alpha$-equivalence has been proved.

## 9 Conclusion

We have presented an extension of Nominal Isabelle for dealing with general binders, that is term-constructors having multiple bound variables. For this extension we introduced new definitions of $\alpha$-equivalence and automated all necessary proofs in Isabelle/HOL. To specify general binders we used the specifications from Ott, but extended them in some places and restricted them in others so that they make sense in the context of $\alpha$-equated terms. We also introduced two binding modes (set and set+) that do not exist in Ott. We have tried out the extension with calculi such as Core-Haskell, type-schemes and approximately a dozen of other typical examples from programming language research [15].

We have left out a discussion about how functions can be defined over $\alpha$-equated terms involving general binders. In earlier versions of Nominal Isabelle this turned out to be a thorny issue. We hope to do better this time by using the function package that has recently been implemented in Isabelle/HOL and also by restricting function definitions to equivariant functions (for them we can provide more automation).

**Acknowledgements:** We thank Peter Sewell for making the informal notes [15] available to us and also for patiently explaining some of the finer points of the Ott-tool.

## References

1. T. Altenkirch, N. A. Danielsson, A. Löh, and N. Oury. PiSigma: Dependent Types Without the Sugar. In *Proc. of the 10th FLOPS Conference*, volume 6009 of *LNCS*, pages 40–55, 2010.
2. J. Bengtson and J. Parrow. Psi-Calculi in Isabelle. In *Proc of the 22nd TPHOLs Conference*, volume 5674 of *LNCS*, pages 99–114, 2009.
3. A. Charguéraud. The Locally Nameless Representation. To appear in J. of Automated Reasoning.
4. J. Cheney. Scrap your Nameplate (Functional Pearl). In *Proc. of the 10th ICFP Conference*, pages 180–191, 2005.
5. J. Cheney. Toward a General Theory of Names: Binding and Scope. In *Proc. of the 3rd MERLIN workshop*, pages 33–40, 2005.
6. P. Homeier. A Design Structure for Higher Order Quotients. In *Proc. of the 18th TPHOLs Conference*, volume 3603 of *LNCS*, pages 130–146, 2005.
7. B. Huffman and C. Urban. Proof Pearl: A New Foundation for Nominal Isabelle. In *Proc. of the 1st ITP Conference*, volume 6172 of *LNCS*, pages 35–50, 2010.
8. C. Kaliszyk and C. Urban. Quotients Revisited for Isabelle/HOL. To appear in the Proc. of the 26th ACM Symposium On Applied Computing, 2011.
9. D. K. Lee, K. Crary, and R. Harper. Towards a Mechanized Metatheory of Standard ML. In *Proc. of the 34th POPL Symposium*, pages 173–184, 2007.
10. W. Naraschewski and T. Nipkow. Type Inference Verified: Algorithm W in Isabelle/HOL. *J. of Automated Reasoning*, 23:299–318, 1999.
11. A. Pitts. Notes on the Restriction Monad for Nominal Sets and Cpos. Unpublished notes for an invited talk given at CTCS, 2004.
12. A. M. Pitts. Nominal Logic, A First Order Theory of Names and Binding. *Information and Computation*, 183:165–193, 2003.
13. F. Pottier. An Overview of C$\alpha$ml. In *ACM Workshop on ML*, volume 148 of *ENTCS*, pages 27–52, 2006.
14. M. Sato and R. Pollack. External and Internal Syntax of the Lambda-Calculus. *J. of Symbolic Computation*, 45:598–616, 2010.
15. P. Sewell. A Binding Bestiary. Unpublished notes.
16. P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: Effective Tool Support for the Working Semanticist. *J. of Functional Programming*, 20(1):70–122, 2010.
17. S. Tobin-Hochstadt and M. Felleisen. The Design and Implementation of Typed Scheme. In *Proc. of the 35rd POPL Symposium*, pages 395–406, 2008.
18. C. Urban, J. Cheney, and S. Berghofer. Mechanizing the Metatheory of LF. In *Proc. of the 23rd LICS Symposium*, pages 45–56, 2008.
19. C. Urban and T. Nipkow. Nominal Verification of Algorithm W. In G. Huet, J.-J. Lévy, and G. Plotkin, editors, *From Semantics to Computer Science. Essays in Honour of Gilles Kahn*, pages 363–382. Cambridge University Press, 2009.
20. C. Urban and C. Tasson. Nominal Techniques in Isabelle/HOL. In *Proc. of the 20th CADE Conference*, volume 3632 of *LNCS*, pages 38–53, 2005.
21. C. Urban and B. Zhu. Revisiting Cut-Elimination: One Difficult Proof is Really a Proof. In *Proc. of the 9th RTA Conference*, volume 5117 of *LNCS*, pages 409–424, 2008.

## Appendix

Details for one case in Theorem 1, which the reader might like to ignore. By definition of the abstraction type *abs_set* we have

$$[as]_{set}.x = [bs]_{set}.y \text{ if and only if } \exists p.\ (as, x) \approx^{op\ =,\ supp,\ p}_{set} (bs, y) \tag{24}$$

and also

$$p \bullet [as]_{set}.x = [p \bullet as]_{set}.(p \bullet x) \tag{25}$$

The second fact derives from the definition of permutations acting on pairs and $\alpha$-equivalence being equivariant. With these two facts at our disposal, we can show the following lemma about swapping two atoms in an abstraction.

**Lemma 3.** *If* $a \notin supp\ x - as$ *and* $b \notin supp\ x - as$ *then* $[as]_{set}.x = [(a\ b) \bullet as]_{set}.((a\ b) \bullet x)$.

*Proof.* This lemma is straightforward using (24) and observing that the assumptions give us $(a\ b) \bullet (supp\ x - as) = supp\ x - as$. Moreover *supp* and set difference are equivariant (see [7]).

Assuming that $x$ has finite support, this lemma together with (25) allows us to show

$$(supp\ x - as)\ supports\ [as]_{set}.x \tag{26}$$

which gives us "one half" of Theorem 1 (the notion of supports is defined in [7]). The "other half" is a bit more involved. To establish it, we use a trick from [11] and first define an auxiliary function *aux*, taking an abstraction as argument: $aux\ ([as]_{set}.x) \stackrel{def}{=} supp\ x - as$.

We can show that *aux* is equivariant (since $p \bullet (supp\ x - as) = supp\ (p \bullet x) - p \bullet as$) and therefore has empty support. This in turn means

$$supp\ (aux\ ([as]_{set}.\ x)) \subseteq supp\ ([as]_{set}\ x)$$

Assuming $supp\ x - as$ is a finite set, we further obtain

$$supp\ x - as \subseteq supp\ [as]_{set}.x \tag{27}$$

since for finite sets of atoms, *bs*, we have $supp\ bs = bs$. Finally, taking (26) and (27) together establishes Theorem 1.