Formal SOS-Proofs for the Lambda-Calculus

Christian Urban, Julien Narboux

Formal SOS-Proofs for the Lambda-Calculus

Christian Urban[1] and Julien Narboux[2]

TU Munich, Germany *University of Strasbourg, France*

Abstract

We describe in this paper formalisations for the properties of weakening, type-substitutivity, subject-reduction and termination of the usual big-step evaluation relation. Our language is the lambda-calculus whose simplicity allows us to show actual theorem-prover code of the formal proofs. The formalisations are done in Nominal Isabelle, a definitional extention of the theorem prover Isabelle/HOL. The point of these formalisations is to be as close as possible to the "pencil-and-paper" proofs for these properties, but of course be completely rigorous. We describe where Nominal Isabelle is of great help with such formalisations and where one has to invest additional effort in order to obtain formal proofs.

Keywords: Structural operational semantics, proof assistants, lambda-calculus, Nominal Isabelle.

1 Introduction

Structural operational semantics (SOS) introduced in 1981 by Plotkin [14] has been very successful in describing what programs are supposed to do. These descriptions can often be used directly in proofs establishing properties about programming languages via induction over the structure of terms or inductions over rules of inductively defined predicates. However, if one wants to formalise such proofs in a theorem prover, then dealing with binders, renaming of bound variables, capture-avoiding substitution, etc., is very often a major problem. Nominal Isabelle [15] is designed to make such proofs easy to formalise: It provides an infrastructure for declaring nominal datatypes (that is α-equivalence classes) and for defining functions over them by structural recursion. It also provides induction principles that have Barendregt's variable convention already built in.

The naïve method of representing binders using abstract syntax-trees is too concrete: it does not take into account α-equivalence where expressions are regarded as equal, if they only differ in the naming of bound variables. As a result one has to deal explicitly with naming issues and has to prove properties modulo α-equivalence. This leads to formal proofs where one has to deal with many details, even if one proves only very simple properties (for an illustrative example see the proof given in [6, Pages 94–104]). Of course one

[1] Email: urbanc@in.tum.de

[2] Email: Julien.Narboux@dpt-info.u-strasbg.fr

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

can reconcile abstract syntax-trees and binders by using de-Bruijn indices. This alleviates the problems about too many details and in some cases leads to very slick proofs. Unfortunately, by using de-Bruijn indices, proofs involve a rather large amount of arithmetic on indices, which is not present in informal descriptions [3]. Another method of representing binders is by using higher-order abstract-syntax (HOAS) where the meta-language provides binding-constructs. The disadvantage we see with HOAS is that one has to encode binders of the object language with variable binders of the meta-language. In practice this means that one does not have direct access anymore to bound variables. This can be a problem if one wants to formalise the classic typing algorithm W presented by Damas and Milner [7]. Recently Aydemir *et al* have reported that a locally nameless representation for terms with binders has been very useful for formalising informal SOS-proofs in Coq [1]. The disadvantage we see with this approach, however, is that one often has to reformulate definitions in order to get through proofs involving bound variables. Also the problem of performing arithmetic over indices, like with "pure" de-Bruijn indices, cannot be completely avoided in the locally nameless representation.

Here we describe Nominal Isabelle, which provides an infrastructure in the theorem prover Isabelle/HOL [11] for representing binders as *named* α-equivalence classes. The paper does not present any new results, rather we describe Nominal Isabelle with some typical proofs from SOS. Our object language will be the lambda-calculus, whose simplicity will allow us to give actual Isar-code [19] for those proofs. Nominal Isabelle adapts ideas from the nominal logic work by Pitts [12]. For example it defines the notion of freshness, written $x \# e$, of a variable x with respect to an expression e.

The paper is organized as follows: Terms and substitutions are defined in Sec. 2, together with a description of strong structural induction principles that have the usual variable convention already built in. Sec. 3 defines types and the typing-judgement for terms Sec. 4 introduces the big-step evaluation relation for terms and in Sec. 5 we show how the proof of the termination property for the evaluation relation proceeds.

2 Terms and Substitutions

We consider here α-equated lambda-terms. For building up these terms we assume the existence of a type *name* for variables. The only property we need to know about *name* is that it consists of infinitely many variables. The terms are then defined by the grammar

Definition 2.1 (Terms) $trm ::= Var\ name \mid App\ trm\ trm \mid Lam\ name.trm$

where in the *Lam*-clause, as usual, a variable is bound. Because Nominal Isabelle allows us to write terms as *Lam x.e*, one might assume that this definition represents "raw", or un-quotient, syntax-trees. However, this is *not* the case: in Nominal Isabelle this definition really represents α-equivalence classes. This can be seen by the fact that the following two terms are *equal*:

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

which would not be the case if our terms were syntax-trees.

The most important operation we need for terms is substitution. In the proofs we present later on it will be necessary to introduce the slightly more complicated notion of simultaneous substitution, which we represent as finite lists of $(name,trm)$-pairs. One reason for this

choice is that it is easier to deal with finite structures in Nominal Isabelle than with infinite ones (a potentially infinite representation of substitutions is, for example, partial maps from *name* to *trm*). The second reason is that it is usually easier to define functions by recursion over lists, than by recursing over sets [10]. Using our list representation we define:

Definition 2.2 (Simultaneous Substitution)

$$\begin{array}{lcll} \theta(\mathit{Var}\ x) & = & \mathit{lookup}\ \theta\ x & \\ \theta(\mathit{App}\ e_1\ e_2) & = & \mathit{App}\ \theta(e_1)\ \theta(e_2) & \\ \theta(\mathit{Lam}\ x.e) & = & \mathit{Lam}\ x.\theta(e) & \mathit{provided}\ x\ \#\ \theta \end{array}$$

where in the first clause we use the auxiliary function *lookup* defined by the clauses:

$$\begin{array}{lcl} \mathit{lookup}\ []\ x & = & \mathit{Var}\ x \\ \mathit{lookup}\ ((y,\ e)::\theta)\ x & = & \mathit{if}\ x = y\ \mathit{then}\ e\ \mathit{else}\ \mathit{lookup}\ \theta\ x \end{array}$$

Single substitutions are a derived concept by defining $e[x::=e'] \stackrel{\text{def}}{=} [(x,\ e')](e)$ where $[(x,\ e')]$ is a singleton list.

Despite the side-condition attached to the *Lam*-clause, the definition above yields a total function, since we work with α-equivalence classes where renamings are always possible. Clearly, if defined over syntax-trees, this definition would be a partial function. While the totality of the substitution operation is rather convenient in a formal proofs, it also means that we must be careful when defining functions over the structure of α-equated terms. This is because we can specify functions over the structure of such terms that lead to inconsistencies. One example is the function that returns the immediate subterms of an α-equated lambda-term, specified by

$$\begin{array}{lcl} \mathit{ist}\ (\mathit{Var}\ x) & = & \varnothing \\ \mathit{ist}\ (\mathit{App}\ e_1\ e_2) & = & \{e_1,\ e_2\} \\ \mathit{ist}\ (\mathit{Lam}\ x.e) & = & \{e\} \end{array}$$

If this function could be defined for α-equivalence classes, then we can prove false. This is because we expect that functions always return the same output for the same input. The problem with the inconsistency can then be seen by considering the α-equivalent terms

$$\mathit{Lam}\ x.(\mathit{Var}\ x) = \mathit{Lam}\ y.(\mathit{Var}\ y)$$

and the two calculations

$$\mathit{ist}\ (\mathit{Lam}\ x.(\mathit{Var}\ x)) = \{\mathit{Var}\ x\} \qquad \mathit{ist}\ (\mathit{Lam}\ y.(\mathit{Var}\ y)) = \{\mathit{Var}\ y\}$$

If we force both right-hand sides to be equal by assuming that *ist* is a function, then we have an inconsistency since $\{\mathit{Var}\ x\} \neq \{\mathit{Var}\ y\}$ in case $x \neq y$.

In order to prevent such inconsistencies, the recursion combinator in Nominal Isabelle only allows to define functions that respect α-equivalence classes [16]. For this we are required in our formalisation to manually check that certain conditions about the clauses in Def. 2.2 are satisfied. To state these condition requires some slightly complicated machinery involving the notion of support of functions (see [13,16]). This notion corresponds roughly to the free variables of an object. In Nominal Isabelle the support is defined not just for functions, but also for pairs, tuples, lists, sets as well as terms. The definition of the latter requires that a permutation operation is defined for terms. This permutation operation, written $\pi \cdot e$, takes a term e and a permutation π, which is a finite list of (*name*, *name*)-pairs and permutes every variable in the term e. We write such permutations as $(a_1$

$b_1)(a_2\ b_2)\cdots(a_n\ b_n)$; the empty list $[]$ stands for the identity permutation. The permutation operation over terms is defined by

Definition 2.3 (Permutations Acting on Terms)

$$\begin{array}{lcl} \pi \cdot \mathit{Var}\ x & = & \mathit{Var}\ (\pi \cdot x) \\ \pi \cdot \mathit{Lam}\ x.e & = & \mathit{Lam}\ (\pi \cdot x).(\pi \cdot e) \\ \pi \cdot \mathit{App}\ e_1\ e_2 & = & \mathit{App}\ (\pi \cdot e_1)\ (\pi \cdot e_2) \end{array}$$

using the auxiliary operation of a permutation acting on a variable

$$[] \cdot a = a$$
$$(a_1\ a_2)::\pi \cdot a = \begin{cases} a_2 & \text{if } \pi \cdot a = a_1 \\ a_1 & \text{if } \pi \cdot a = a_2 \\ \pi \cdot a & \text{otherwise} \end{cases}$$

The support of an object x is then defined as the set of names satisfying

$$\mathit{supp}\ x \stackrel{\text{def}}{=} \{a \mid \mathit{infinite}\ \{b \mid [(a, b)] \cdot x \neq x\}\}$$

which in case of α-equated lambda-terms coincides with the usual notion of free variables.

Using the permutation operation, Nominal Isabelle also defines the notion of α-equivalence for abstractions, which are written $x.e$. This definition distinguishes whether the binders of two abstractions are equal or not:

$$\frac{e_1 = e_2}{x.e_1 = x.e_2} \qquad \frac{x \neq y \qquad e_1 = (x\ y)\cdot e_2 \qquad x\ \#\ e_2}{x.e_1 = y.e_2}$$

In the second rule $x\ \#\ e_2$ stands for x not being in the support of e_2, which, as mentioned above, coincides with x being not a free variable in e_2. Having the notion of α-equivalence for abstractions in place, Nominal Isabelle defines under which conditions two lambda-terms are equal, namely

$$\frac{x = y}{\mathit{Var}\ x = \mathit{Var}\ y} \qquad \frac{x.e_1 = y.e_2}{\mathit{Lam}\ x.e_1 = \mathit{Lam}\ y.e_2} \qquad \frac{e_1 = {e_1}' \qquad e_2 = {e_2}'}{\mathit{App}\ e_1\ e_2 = \mathit{App}\ {e_1}'\ {e_2}'}$$

Equipped with the rules about α-equivalence, we can start to prove properties about terms and substitutions. Later on, for example, we will need the property how a single and a simultaneous substitution interact. For this we prove the following lemma:

Lemma 2.4 *If* $x\ \#\ \theta$ *then* $\theta(e)[x::=e'] = ((x, e')::\theta)(e)$.

whose proof is by induction on the structure of e. For such proofs Nominal Isabelle derives two versions of the structural induction principle—a weak one and a strong one. The weak proves a property $P\ e$ for all terms e provided one establishes for each term-constructor an implication that assumes the property for the arguments and concludes the property for the term-constructor. This pattern follows what Plotkin [14, Page 49] describes as *structural induction for expressions*. As an inference rule the weak induction principle looks as follows:

$$\frac{\begin{array}{l} \forall x.\ P\ (\mathit{Var}\ x) \\ \forall x\ e.\ P\ e \longrightarrow P\ (\mathit{Lam}\ x.e) \\ \forall e_1\ e_2.\ P\ e_1 \wedge P\ e_2 \longrightarrow P\ (\mathit{App}\ e_1\ e_2) \end{array}}{P\ e}$$

Using this principle, the cases in Lem. 2.4 for *Var* and *App* are quite routine, but in the *Lam*-case one has to analyse a binder. We have the induction-hypothesis:

$$\forall x\ \theta\ e'.\ x\ \#\ \theta \longrightarrow \theta(e)[x::=e'] = ((x, e')::\theta)(e)$$

and have to show

$$\theta(Lam\ y.e)[x::=e'] = ((x, e')::\theta)(Lam\ y.e)$$

for arbitrary y and e. However we only know that $x\ \#\ \theta$ holds. In order to apply the definition of substitution and subsequently use the induction hypothesis we need to rename the binder y to a fresh variable z, say. This makes the proof quite clunky and too hard to be found by the automatic search tools available in Isabelle. In informal proofs establishing such properties by induction, one usually ignores the fact that one has to establish the property at hand for an arbitrary bound variable y; rather one employs the convention that binders are always assumed to be suitable fresh (see for example [2]). In the case above this means we have the convention that y is fresh for θ, x and e', that is $y\ \#\ \theta$, $y\ \#\ x$ and $y\ \#\ e'$ hold. With this convention also the case *Lam* is trivial.

To support this kind of informal reasoning where one does not consider truly arbitrary bound variables, but rather bound variables about which various freshness assumptions are made, Nominal Isabelle derives automatically from the weak induction principle a strong induction principle (see [18]). This strong induction principle looks as inference rule as follows:

$$\frac{\begin{array}{l}\forall c\ x.\ P\ c\ (Var\ x)\\ \forall c\ x\ e.\ x\#c\ \wedge (\forall c.\ P\ c\ e) \longrightarrow P\ c\ (Lam\ x.e)\\ \forall c\ e_1\ e_2.\ (\forall c.\ P\ c\ e_1) \wedge (\forall c.\ P\ c\ e_2) \longrightarrow P\ c\ (App\ e_1\ e_2)\end{array}}{P\ c\ e}$$

The purpose of the parameter c, called the *induction context*, is to accommodate the assumptions we make in informal reasoning about the freshness of the binder. In the *Lam*-case we can then assume that the binder for which the property needs to be established is fresh with respect to this context (see highlighted formula). With these assumptions in place the case for *Lam* is also completely routine: we just have to instantiate the induction context with the tuple (θ, x, e'). The only requirement we have to observe with this instantiations is that the context may only mention finitely many free variables. This holds in our case. We then have the same induction hypothesis as in the weak version

$$\forall x\ \theta\ e'.\ x\ \#\ \theta \longrightarrow \theta(e)[x::=e'] = ((x, e')::\theta)(e)$$

However additionally we have that $y\ \#\ \theta$, $y\ \#\ x$ and $y\ \#\ e'$. These additional assumptions help us in the proof obligation in Lem. 2.4:

$$\theta(Lam\ y.e)[x::=e'] = ((x, e')::\theta)(Lam\ y.e)$$

We can now move θ and the single substitution under the lambda-abstraction on the left-hand side (similarly with $(x, e')::\theta$ on the right-hand side), and then apply the induction hypothesis. As a result *all* cases of Lem. 2.4 are routine and the formal proof is completely automatic, except for setting up the induction and for the need of mentioning two properties about *lookup*, namely:

(i) If $x\ \#\ \theta$ then $lookup\ \theta\ x = Var\ x$

(ii) If $z \# \theta$ and $z \neq x$ then $(\mathit{lookup}\ \theta\ x)[z::=e] = \mathit{lookup}\ \theta\ x$

With these properties, named *lookup-fresh*$_1$ and *lookup-fresh*$_2$ below, the formal proof establishing of Lem. 2.4 is:

```
1 lemma psubst-subst:
2 assumes a: x # θ
3 shows θ(e)[x::=e'] = ((x,e'))::θ)(e)
4 using a by (nominal-induct e avoiding: θ x e' rule: trm.strong-induct)
5            (auto simp add: lookup-fresh1 lookup-fresh2)
```

In line 4, we set up the induction and instantiate the induction context by avoiding θ, x and e'. The argument is then just by calculation, which Isabelle can do automatically.

To sum up this section, Nominal Isabelle derives automatically strong versions of the induction principle for all term-calculi involving single binders, not just the one defined in Def. 2.1. This often makes reasoning by structural induction over α-equivalence classes rather pleasant, because no explicit α-conversions are needed. This is a theme which will reoccur frequently in the proofs we shall describe in the next sections.

3 Typing

Many SOS-proofs involve typing-information for terms. In this section we define types and a typing relation for our terms. The definition of types, for which we use the letter T, consists of type variables and function types:

Definition 3.1 (Types) $ty ::= \mathit{TVar}\ x \mid ty \to ty$

Before we can define a typing-judgement, we need to state what typing contexts are. For them we use lists of $(\mathit{name},\mathit{ty})$-pairs since, as mentioned before, in Nominal Isabelle it is easier to work with finite structures than with infinite ones (if we use sets of $(\mathit{name},\mathit{ty})$-pairs instead, then it is inconvenient to exclude potentially infinitely large typing-contexts). The disadvantage of using lists is, of course, that we distinguish the order of how variables are associated to types. However, in terms of convenience this choice will only cause minor problem in the proofs we shall present.

A typing-context Γ is *valid*, provided it includes only a single association for every variable occurring in Γ. This can be defined inductively by the two rules

$$\frac{}{\mathit{valid}\ []} \qquad \frac{\mathit{valid}\ \Gamma \qquad x \# \Gamma}{\mathit{valid}\ ((x, T)::\Gamma)}$$

where $x \# \Gamma$ stands for x not occurring in Γ. Having the definition of validity at our disposal, the rules for the typing-judgements are relatively standard:

$$\frac{\mathit{valid}\ \Gamma \qquad (x, T) \in \Gamma}{\Gamma \vdash \mathit{Var}\ x : T}\,\textit{t-Var} \qquad \frac{\Gamma \vdash e_1 : T_1 \to T_2 \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash \mathit{App}\ e_1\ e_2 : T_2}\,\textit{t-App}$$

$$\frac{x \# \Gamma \qquad (x, T_1)::\Gamma \vdash e : T_2}{\Gamma \vdash \mathit{Lam}\ x.e : T_1 \to T_2}\,\textit{t-Lam}$$

In rule *t-Var* we use the notation $(x, T) \in \Gamma$ to stand for list-membership. Note the freshness condition in the rule *t-Lam*, which makes this rule sound with respect to the typing-judgements we intend to be derivable.

From the definition of the typing rules, Nominal Isabelle can again derive a stronger induction principle, where in the lambda-case we can assume that the binder satisfies some chosen freshness constraints (this is similar to the stronger structural induction principle for lambda-terms). The ability to choose some freshness constraints is already greatly helpful in proofs of simple properties, for example the weakening lemma. In order to prove this lemma for our typing contexts represented as lists, we first define the notion of a *sub-context* as follows: [3]

Definition 3.2 (Sub-Contexts) $\Gamma_1 \subseteq \Gamma_2 \stackrel{\text{def}}{=} \forall x\, T.\ (x, T) \in \Gamma_1 \longrightarrow (x, T) \in \Gamma_2$.

We can then state the weakening-lemma in terms of sub-contexts:

Lemma 3.3 (Weakening) *If* $\Gamma_1 \vdash e : T$ *and valid* Γ_2 *and* $\Gamma_1 \subseteq \Gamma_2$ *then* $\Gamma_2 \vdash e : T$.

In the proof of this lemma, again the *t-Var* and *t-App* cases are routine. Unfortunately not the *t-Lam*-case. This is because the usual (that is weak) induction principle coming with the definition of the typing-rules does not cope well with binders. Consider the naïve attempt of proving the suitably generalised property of weakening, namely

$$\Gamma_1 \vdash e : T \longrightarrow (\forall \Gamma_2.\ \mathit{valid}\ \Gamma_2 \longrightarrow \Gamma_1 \subseteq \Gamma_2 \longrightarrow \Gamma_2 \vdash e : T).$$

Then in the *t-Lam* case we have the induction hypothesis

$$\forall \Gamma_2.\ \mathit{valid}\ \Gamma_2 \longrightarrow (x, T_1)::\Gamma_1 \subseteq \Gamma_2 \longrightarrow \Gamma_2 \vdash e : T_2$$

which we like to use with the instantiation $\Gamma_2 = (x, T_1)::\Gamma_2$. However this will not allow us to make any progress as we cannot obtain $(x, T_1)::\Gamma_2 \vdash e : T$. The reason is that we only know that x is fresh for the smaller typing context Γ_1 and we cannot infer anything for the bigger context Γ_2. Consequently, we cannot ascertain whether *valid* $((x, T_1)::\Gamma_2)$ holds. To get the proof through the naïve way, we have to rename the binder first, at which point the simplicity of the proof disappears (see [8,9]): the inductive hypothesis is much harder to show applicable because it mentions e, but the desired goal is in terms of $e[x::=z]$. This will require a lemma establishing the invariance of the typing-judgement under renamings.

The renaming can be completely avoided if we use the strong version of the induction principle that has the usual variable convention built in. The formal proof is then very close to being straightforward:

```
1  lemma weakening:
2    fixes Γ1 Γ2::(name×ty) list
3    assumes a: Γ1 ⊢ e: T and b: valid Γ2 and c: Γ1 ⊆ Γ2
4    shows Γ2 ⊢ e: T
5  using a b c proof (nominal-induct Γ1 e T avoiding: Γ2 rule: typing.strong-induct)
6    case (t-Lam x Γ1 T1 t T2 Γ2)
7    have vc: x # Γ2 by fact
8    have ih: ⟦valid ((x,T1)::Γ2); (x,T1)::Γ1 ⊆ (x,T1)::Γ2⟧ ⟹ (x,T1)::Γ2 ⊢ t : T2 by fact
9    have valid Γ2 by fact
10   then have valid ((x,T1)::Γ2) using vc by auto
```

[3] This is a neat trick we have learned from Randy Pollack.

11 **moreover**
12 **have** $\Gamma_1 \subseteq \Gamma_2$ **by** *fact*
13 **then have** $(x,T_1)::\Gamma_1 \subseteq (x,T_1)::\Gamma_2$ **by** *simp*
14 **ultimately have** $(x,T_1)::\Gamma_2 \vdash t : T_2$ **using** *ih* **by** *simp*
15 **with** *vc* **show** $\Gamma_2 \vdash Lam\ x.t : T_1 \rightarrow T_2$ **by** *auto*
16 **qed** (*auto*)

Line 5 sets up the induction to avoid Γ_2; therefore we can assume in Line 7 that the binder x is fresh w.r.t. Γ_2. This fact is used in Line 10 to infer validity of *valid* $((x,T_1)::\Gamma_2)$ and in Line 15 to apply the typing rule. Line 8 states the induction hypothesis (which is already instantiated with $(x, T_1)::\Gamma_2$) and the reasoning in Lines 9–13 ensures that the induction hypothesis is applicable. The cases for variables and applications are derived automatically in Line 16. In fact *all* the calculation involved in this lemma can be done automatically by Isabelle as can be seen below:

lemma *weakening*:
fixes $\Gamma_1\ \Gamma_2$::(*name*×*ty*) *list*
assumes *a*: $\Gamma_1 \vdash e$: T **and** *b*: *valid* Γ_2 **and** *c*: $\Gamma_1 \subseteq \Gamma_2$
shows $\Gamma_2 \vdash e$: T
using *a b c* **by** (*nominal-induct* Γ_1 *e T avoiding*: Γ_2 *rule*: *typing.strong-induct*) (*auto*)

Next we will establish the type-substitutivity lemma, which we will be crucial later on when showing the type-preservation property.

Lemma 3.4 (Type-Substitutivity)
If $(x, T')::\Gamma \vdash e : T$ *and* $\Gamma \vdash e' : T'$ *then* $\Gamma \vdash e[x::=e'] : T$.

There are a number of ways to prove this lemma. One is to use the strong induction principle for lambda-terms and perform an induction over the structure of e. This involves a fair amount of straightforward calculations, but they cannot be found by the automated tools of Isabelle. It is more convenient if we first generalise Lem. 3.4 and prove

Lemma 3.5
If $\Delta \mathbin{@} [(x, T')] \mathbin{@} \Gamma \vdash e : T$ *and* $\Gamma \vdash e' : T'$ *then* $\Delta \mathbin{@} \Gamma \vdash e[x::=e'] : T$.

by a strong induction on the first typing relation. For this we have to fix the typing context in the induction to be $\Delta \mathbin{@} [(x, T')] \mathbin{@} \Gamma$ and avoid e' and Δ. Fixing the typing context to be $\Delta \mathbin{@} [(x, T')] \mathbin{@} \Gamma$ is a slightly roundabout way of saying that the type association for the variable x occurs somewhere inside the typing context (this is needed in order to get the *Lam*-case trough). The avoiding part will give us the necessary assumptions in order push the substitution under the lambda-abstraction in the *t-Lam*-case: we have in this case the induction hypothesis

$$\forall \Delta\ e'.\ \Gamma \vdash e' : T' \longrightarrow \Delta \mathbin{@} \Gamma \vdash e[x::=e'] : T_2$$

and need to show that

$$\Gamma \vdash e' : T' \longrightarrow \Delta \mathbin{@} \Gamma \vdash Lam\ y.e[x::=e'] : T_1 \rightarrow T_2$$

under the assumption that $y \sharp \Delta$, $y \sharp e'$ and $y \sharp (\Delta \mathbin{@} [(x, T')] \mathbin{@} \Gamma)$ (the former two come from the strong induction and the latter from the premise of the *t-Lam*-rule). Having these assumptions at our disposal, we can move the substitution under the lambda-

abstraction and then apply the *t-Lam*-rule, which is possible since $y \# (\Delta @ [(x, T')] @ \Gamma)$ implies that also $y \# (\Delta @ \Gamma)$ holds (a fact called *fresh-list-append* in the proof below). Finally we can use the induction hypothesis to complete the proof. This reasoning leads to the following quite automatic formal proof. We only have to give the details for the variable case, because there we have to do a case distinction that cannot be found automatically be Isabelle.

```
1  lemma type-substitutivity-aux:
2    assumes a: Δ@[(x,T')]@Γ ⊢ e : T and b: Γ ⊢ e' : T'
3    shows Δ@Γ ⊢ e[x::=e'] : T
4  using a b
5  proof (nominal-induct Γ ≡def (Δ@[(x,T')]@Γ) e T avoiding: e' Δ rule: typing.strong-induct)
6    case (t-Var Γ' y T e' Δ)
7    then have a1: valid (Δ@[(x,T')]@Γ) and  a2: (y,T) ∈ set (Δ@[(x,T')]@Γ)
8          and  a3: Γ ⊢ e' : T' by simp-all
9    from a1 have a4: valid (Δ@Γ) by (rule valid-insert)
10   { assume eq: x=y
11     from a1 a2 have T=T' using eq by (auto intro: context-unique)
12     with a3 have Δ@Γ ⊢ Var y[x::=e'] : T using eq a4 by (auto intro: weakening) }
13   moreover
14   { assume ineq: x≠y
15     from a2 have (y,T) ∈ set (Δ@Γ) using ineq by simp
16     then have Δ@Γ ⊢ Var y[x::=e'] : T using ineq a4 by auto }
17   ultimately show Δ@Γ ⊢ Var y[x::=e'] : T by blast
18 qed (force simp add: fresh-list-append)+
19
```

In line Line 5 we set up the strong induction principle by fixing the typing context and avoiding e' and Δ. Lines 7 and 8 mention the assumption that are available in the *t-Var*-case. Line 9 contains the fact *valid* $(\Delta @ \Gamma)$ which follows from the assumption *valid* $(\Delta @ [(x, T')] @ \Gamma)$. In Lines 10 to 12 we treat the case where the variables x and y are equal. In order to complete this case we have to use weakening. In Lines 14 to 16 we tread the case where $x \neq y$. As a result we can conclude the *t-Var*-case in Line 17. The remaining cases for rules *t-App* and *t-Var* can be found automatically in in Line 18.

Lemma 3.4 is now a simple corollary of Lemma 3.5. Later on we will need the following inversion properties for the typing relation.

Lemma 3.6 (Type-Inversion)

(i) *If* $\Gamma \vdash App\ t_1\ t_2 : T$ *then* $\exists T'.\ \Gamma \vdash t_1 : T' \to T \wedge \Gamma \vdash t_2 : T'$.

(ii) *If* $\Gamma \vdash Lam\ x.t : T$ *and* $x \# \Gamma$ *then* $\exists T_1\ T_2.\ (x, T_1)::\Gamma \vdash t : T_2 \wedge T = T_1 \to T_2$.

Note that the second inversion property needs the precondition $x \# \Gamma$, which means we can only invert the typing relation provided the bound variable x is sufficiently fresh [4].

4 Big-Step Evaluation Relation

In this section we define the usual big-step call-by-value semantics. The inference rules are

$$\frac{x \# (e_1, e_2, e') \qquad e_1 \Downarrow Lam\ x.e \qquad e_2 \Downarrow e_2' \qquad e[x::=e_2'] \Downarrow e'}{App\ e_1\ e_2 \Downarrow e'}\ b\text{-}App$$

$$\frac{}{Lam\ x.e \Downarrow Lam\ x.e}\ b\text{-}Lam$$

In order to take advantage of the automatic facilities in Nominal Isabelle, we have to state the *b-App*-rule so that it includes the freshness constraints $x \# (e_1, e_2, e')$. An important property we can establish for evaluation is that it preserves types.

Lemma 4.1 (Subject Reduction) *If* $e \Downarrow e'$ *and* $\Gamma \vdash e : T$ *then* $\Gamma \vdash e' : T$.

The proof of this lemma is quite routine when we have a strong induction principle for the evaluation relation at our disposal. The only lemmas we need are Lem. 3.4 and the two inversion properties from Lem. 3.6 (Lines 8 and 13).

```
1  lemma subject-reduction:
2    assumes a: e ⇓ e' and b: Γ ⊢ e : T
3    shows Γ ⊢ e' : T
4  using a b proof (nominal-induct avoiding: Γ arbitrary: T rule: big.strong-induct)
5    case (b-App x e1 e2 e' e e2' Γ T)
6    have vc: x # Γ by fact
7    have Γ ⊢ App e1 e2 : T by fact
8    then obtain T' where a1: Γ ⊢ e1 : T'→T and a2: Γ ⊢ e2 : T' by (auto elim: t-App-elim)
9    have ih1: Γ ⊢ e1 : T' → T ⟹ Γ ⊢ Lam x . e : T' → T by fact
10   have ih2: Γ ⊢ e2 : T' ⟹ Γ ⊢ e2' : T' by fact
11   have ih3: Γ ⊢ e[x::=e2'] : T ⟹ Γ ⊢ e' : T by fact
12   have Γ ⊢ Lam x.e : T'→T using ih1 a1 by simp
13   then have (x,T')::Γ ⊢ e : T using vc  by (auto elim: t-Lam-elim)
14   moreover
15   have Γ ⊢ e2': T' using ih2 a2 by simp
16   ultimately have Γ ⊢ e[x::=e2'] : T by (simp add: type-substitutivity)
17   then show Γ ⊢ e' : T using ih3 by simp
18 qed (force)
```

In Line 6 we can assume that $x \# \Gamma$ holds. We invert the assumed typing derivation in Lines 7 and 8. The three induction hypotheses of this case are mentioned in Lines 9 to 11. The first one is used in Line 12 and 13 to infer $(x, T')::\Gamma \vdash e : T$; the second in Line 15 to infer $\Gamma \vdash e_2' : T'$. From these two facts follows that $\Gamma \vdash e[x::=e_2'] : T$ holds. Using the third induction hypothesis in Line 17, we can conclude the *b-App*-case. Since the reasoning is quite routine, Isabelle will be able to find the proof as shown below.

```
lemma subject-reduction:
  assumes a: e ⇓ e' and b: Γ ⊢ e : T
  shows Γ ⊢ e' : T
using a b by (nominal-induct avoiding: Γ arbitrary: T rule: big.strong-induct)
             (force elim: t-App-elim t-Lam-elim simp add: type-substitutivity)+
```

Another important property is that the evaluation relation produces unique results. This can be stated as follows.

Lemma 4.2 (Unicity) *If* $e \Downarrow e_1$ *and* $e \Downarrow e_2$ *then* $e_1 = e_2$.

The proof of this lemma is by rule induction over the evaluation relation. The reasoning is similar to Lem. 4.1 and therefore omitted.

A small lemma which is often overlooked in informal reasoning is that freshness is preserved by evaluation.

Lemma 4.3 (Freshness Preservation) *If* $e \Downarrow e'$ *and* $x \# e$ *then* $x \# e'$.

This lemma can in our formalisation be discharged by a completely automatic induction on the evaluation relation. It will play an important rôle when we show in the next section that evaluation terminates for well-typed terms.

5 Termination

The last property we formalise in this paper is that for every typable closed lambda-term evaluates to a value, that means in our context here to a lambda-abstraction.

Theorem 5.1 (Termination) *If* $[] \vdash e : T$ *then* $\exists v.\ e \Downarrow v \wedge val\ v$.

The proof of the this lemma is not straightforward and we cannot expect that the automatic proof search tools of Isabelle are of much help in finding this proof. The proof atually only goes through if one proves a stronger result. For this we use the well-known technique of logical relations. The specific logical relation we use here we will call *valuation*. They are sets of terms and defined as follows:

$$\begin{array}{lcl} V\ (TVar\ x) & = & \{e \mid val\ e\} \\ V\ (T_1 \rightarrow T_2) & = & \{Lam\ x.e \mid \forall v \in V\ T_1.\ \exists v'.\ e[x::=v] \Downarrow v' \wedge v' \in V\ T_2\} \end{array}$$

where the first clause contains the predicate *val*, which only holds for lambda-abstractions, and the second clause includes the standard closure property for lambda-abstractions. In the main lemma we will show that a typable term together with a closing substitution evaluates. In order to define what is meant by a closing substitution we introduce for simultaneous substitutions the notion θ *maps* x *to* e, which ensures that θ contains the association (x, e).

Definition 5.2 θ *maps* x *to* $e \stackrel{\text{def}}{=} lookup\ \theta\ x = e$.

Next, we introduce a notion for when a substitution θ closes a typable term, that means has an assignment for every (x, T)-pair in a typing context Γ, whereby the assignment in θ must come from the valuation $V\ T$.

Definition 5.3 θ *Vcloses* $\Gamma \stackrel{\text{def}}{=} \forall x\ T.\ (x, T) \in \Gamma \longrightarrow (\exists v.\ \theta$ *maps* x *to* $v \wedge v \in V\ T)$.

The first lemma we show is that *Vcloses* is preserved under suitable additional assignments to simultaneous substitutions and typing-contexts. This property is often called the *monotonicity*, or *preservation under weakening* [5].

Lemma 5.4 (Monotonicity) *If* θ *Vcloses* Γ *and* $e \in V\ T$ *and valid* $((x, T)::\Gamma)$ *then* $(x, e)::\theta$ *Vcloses* $(x, T)::\Gamma$.

The proof of this lemma is a routine case-distinction on the extended typing-context and simultaneous substitution. Now we are in a position to give a proof Theorem 5.1, where,

however, we do not prove termination just for closed expressions, but for arbitrary typable terms.

Lemma 5.5 (Termination on open Terms)
If $\Gamma \vdash e : T$ *and* θ *Vcloses* Γ *then* $\exists v.\ \theta(e) \Downarrow v \wedge v \in V\,T$.

Proof. This proof is by a strong structural induction on e, where we generalise over T and set up the induction so that in the lambda-cases we can assume the binders are fresh for Γ and θ. The interesting cases are *App* and *Lam* which we give below.

1 **case** $(App\ e_1\ e_2\ \Gamma\ \theta\ T)$
2 **have** ih_1: $\bigwedge\theta\ \Gamma\ T.\ [\![\theta$ *Vcloses* $\Gamma; \Gamma \vdash e_1 : T]\!] \Longrightarrow \exists v.\ \theta(e_1) \Downarrow v \wedge v \in V\,T$ **by** *fact*
3 **have** ih_2: $\bigwedge\theta\ \Gamma\ T.\ [\![\theta$ *Vcloses* $\Gamma; \Gamma \vdash e_2 : T]\!] \Longrightarrow \exists v.\ \theta(e_2) \Downarrow v \wedge v \in V\,T$ **by** *fact*
4 **have** as_1: θ *Vcloses* Γ **by** *fact*
5 **have** as_2: $\Gamma \vdash App\ e_1\ e_2 : T$ **by** *fact*
6 **then obtain** T' **where** $\Gamma \vdash e_1 : T' \to T$ **and** $\Gamma \vdash e_2 : T'$ **by** $(auto\ elim$: *t-App-elim*$)$
7 **then obtain** $v_1\ v_2$ **where** (i): $\theta(e_1) \Downarrow v_1\ v_1 \in V\ (T' \to T)$
8 **and** (ii): $\theta(e_2) \Downarrow v_2\ v_2 \in V\,T'$ **using** $ih_1\ ih_2\ as_1$ **by** *blast*
9 **from** (i) **obtain** $x\ e'$
10 **where** $v_1 = Lam\ x.e'$
11 **and** (iii): $(\forall v \in (V\,T').\exists\ v'.\ e'[x::=v] \Downarrow v' \wedge v' \in V\,T)$
12 **and** (iv): $\theta(e_1) \Downarrow (Lam\ x.e')$
13 **and** fr: $x \# (\theta,e_1,e_2)$ **by** $(blast\ elim$: *V-arrow-elim-strong*$)$
14 **from** fr **have** fr_1: $x \# \theta(e_1)$ **and** fr_2: $x \# \theta(e_2)$ **by** $(simp\text{-}all\ add$: *fresh-psubst*$)$
15 **from** (ii) (iii) **obtain** v_3 **where** (v): $e'[x::=v_2] \Downarrow v_3 \wedge v_3 \in V\,T$ **by** *auto*
16 **from** fr_2 (ii) **have** $x \# v_2$ **by** $(simp\ add$: *big-preserves-fresh*$)$
17 **then have** $x \# e'[x::=v_2]$ **by** $(simp\ add$: *fresh-subst*$)$
18 **then have** fr_3: $x \# v_3$ **using** (v) **by** $(auto\ simp\ add$: *big-preserves-fresh*$)$
19 **from** $fr_1\ fr_2\ fr_3$ **have** $x \# (\theta(e_1),\theta(e_2),v_3)$ **by** *simp*
20 **with** (iv) (ii) (v) **have** $App\ (\theta(e_1))\ (\theta(e_2)) \Downarrow v_3$ **by** *auto*
21 **then show** $\exists v.\ \theta(App\ e_1\ e_2) \Downarrow v \wedge v \in V\,T$ **using** (v) **by** *auto*

By induction hypothesis (Lines 2 and 3) we know that:

$$(\text{ih}_1) \quad \forall \theta\ \Gamma\ T.\ \theta\ Vcloses\ \Gamma \wedge \Gamma \vdash e_1 : T \longrightarrow (\exists v.\ \theta(e_1) \Downarrow v \wedge v \in V\,T)$$

$$(\text{ih}_2) \quad \forall \theta\ \Gamma\ T.\ \theta\ Vcloses\ \Gamma \wedge \Gamma \vdash e_2 : T \longrightarrow (\exists v.\ \theta(e_2) \Downarrow v \wedge v \in V\,T)$$

By assumption (Lines 4 and 5) we know

$$(\text{as}_1) \quad \theta\ Vcloses\ \Gamma \qquad \text{and} \qquad (\text{as}_2) \quad \Gamma \vdash App\ e_1\ e_2 : T$$

From the second assumption we can derive that $\Gamma \vdash e_1 : T' \to T$ and $\Gamma \vdash e_2 : T'$ hold by inversion of *t-App* for some type T' (Line 6). Using the induction hypotheses and the first assumption (Lines 7 and 8) we can derive that there exists a v_1 and v_2 such that:

$$(i) \quad \theta(e_1) \Downarrow v_1 \quad \text{and} \quad v_1 \in V\ (T' \to T)$$

$$(ii) \quad \theta(e_2) \Downarrow v_2 \quad \text{and} \quad v_2 \in V\,T'$$

From the first fact, we obtain by definition of V that v_1 must be of the form $Lam\ x.e'$ (Line 10) whereby x can be assumed to be fresh for θ, e_1 and e_2 (Line 13; in this step we need

a strong elimination rule for the function V). This also implies that x is fresh for $\theta(e_1)$ and $\theta(e_2)$ (Line 14). We can further infer from the definition of V that (Lines 11 and 12):

$$(iii) \quad \forall v{\in}V\,T'.\ \exists v'.\ e'[x::=v] \Downarrow v' \wedge v' \in V\,T \text{ and}$$

$$(iv) \quad \theta(e_1) \Downarrow Lam\ x.e'$$

Now we combine (ii) and (iii) to obtain a v_3 (Line 15) such that

$$(v) \quad e'[x::=v_2] \Downarrow v_3 \ \text{ and } \ v_3 \in V\,T$$

holds. Since x is fresh for $\theta(e_2)$ and freshness is preserved under evaluation (see Lem. 4.3), we have by (ii) that x is fresh for v_2 (Line 16). In turn this means that x is fresh for $e'[x::=v_2]$ (Line 17), and hence by (v) also for v_3 (Line 18). Now (Line 19) we have $x \# (\theta(e_1), \theta(e_2), v_3)$ which we can combine with (iv), (ii) and (v) to obtain by rule *b-App* that (Line 20)

$$App\ \theta(e_1)\ \theta(e_2) \Downarrow v_3$$

holds. Using v_3 we can conclude (Line 21) that there exists a v such that:

$$\theta(App\ e_1\ e_2) \Downarrow v \ \text{ and } \ v \in V\,T.$$

This completes the application-case. The lambda-case is as follows:

1 **case** $(Lam\ x\ e\ \Gamma\ \theta\ T)$
2 **have** *ih*: $\bigwedge\theta\ \Gamma\ T.\ [\![\theta\ Vcloses\ \Gamma; \Gamma \vdash e : T]\!] \Longrightarrow \exists v.\ \theta(e) \Downarrow v \wedge v \in V\,T$ **by** *fact*
3 **have** as_1: $\theta\ Vcloses\ \Gamma$ **by** *fact*
4 **have** as_2: $\Gamma \vdash Lam\ x.e : T$ **by** *fact*
5 **have** *fs*: $x \#\ \Gamma\ x \#\ \theta$ **by** *fact*
6 **from** as_2 *fs* **obtain** $T_1\ T_2$
7 **where** (i): $(x,T_1)::\Gamma \vdash e{:}T_2$ **and** (ii): $T = T_1 \to T_2$ **using** *fs* **by** $(auto\ elim{:}\ t\text{-}Lam\text{-}elim)$
8 **from** (i) **have** (iii): $valid\ ((x,T_1)::\Gamma)$ **by** $(simp\ add{:}\ typing\text{-}implies\text{-}valid)$
9 **have** $\forall v \in (V\,T_1).\ \exists v'.\ (\theta(e))[x::=v] \Downarrow v' \wedge v' \in V\,T_2$
10 **proof**
11 **fix** v
12 **assume** $v \in (V\,T_1)$
13 **with** (iii) as_1 **have** $(x,v)::\theta\ Vcloses\ (x,T_1)::\Gamma$ **using** *monotonicity* **by** *auto*
14 **with** *ih* (i) **obtain** v' **where** $((x,v)::\theta)(e) \Downarrow v' \wedge v' \in V\,T_2$ **by** *blast*
15 **then have** $\theta(e)[x::=v] \Downarrow v' \wedge v' \in V\,T_2$ **using** *fs* **by** $(simp\ add{:}\ psubst\text{-}subst\text{-}psubst)$
16 **then show** $\exists v'.\ \theta(e)[x::=v] \Downarrow v' \wedge v' \in V\,T_2$ **by** *auto*
17 **qed**
18 **then have** $Lam\ x.\theta(e) \in V\ (T_1 \to T_2)$ **by** *auto*
19 **then have** $\theta(Lam\ x.e) \Downarrow (Lam\ x.\theta(e)) \wedge Lam\ x.\theta(e) \in V\ (T_1{\to}T_2)$ **using** *fs* **by** *auto*
20 **then show** $\exists v.\ \theta(Lam\ x.e) \Downarrow v \wedge v \in V\,T$ **using** (ii) **by** *auto*

By induction hypothesis (Line 2) we know that:

$$\text{(ih)} \quad \forall\theta\ \Gamma\ T.\ \theta\ Vcloses\ \Gamma \wedge \Gamma \vdash e : T \longrightarrow (\exists v.\ \theta(e) \Downarrow v \wedge v \in V\,T)$$

By assumption (Lines 3 and 4) we know

$$(\text{as}_1) \quad \theta\ Vcloses\ \Gamma \qquad \text{and} \qquad (\text{as}_2) \quad \Gamma \vdash Lam\ x.e : T$$

Since we use a strong induction principle we know further the freshness conditions that $x \# \Gamma$ and $x \# \theta$ (Line 5). We can use them and the second assumption to infer (Lines 6–8) that

$$(i) \quad (x, T_1)::\Gamma \vdash e : T_2 \qquad (ii) \quad T = T_1 \to T_2 \qquad (iii) \quad \mathit{valid}\ ((x, T_1)::\Gamma)$$

Where (*iii*) follows from (*i*) since the judgment $(x, T_1)::\Gamma \vdash e : T_2$ implies that $(x, T_1)::\Gamma$ must be valid.

Next we are going to show (Lines 9–18) that $\mathit{Lam}\ x.\theta(e) \in V\ (T_1 \to T_2)$. By definition of V, it therefore suffices to show that (Line 9)

$$\exists v'.\ \theta(e)[x::=v] \Downarrow v' \wedge v' \in V\ T_2$$

holds for all $v \in V\ T_1$. We can use Lemma 5.4, (*iii*) and the first assumption to infer (Line 13) that $(x, v)::\theta\ \mathit{Vcloses}\ (x, T_1)::\Gamma$. We can use this and (*i*) to instantiate the induction hypothesis, which gives us a v' such that (Line 14)

$$((x, v)::\theta)(e) \Downarrow v' \wedge v' \in V\ T_2$$

holds. We know by Lem. 2.4 that this is equivalent to $\theta(e)[x::=v] \Downarrow v' \wedge v' \in V\ T_2$, since x is fresh for θ (Line 15). This means we have shown that $\mathit{Lam}\ x.\theta(e) \in V\ (T_1 \to T_2)$ holds (Line 18) and we also know by the freshness of x that $\theta(\mathit{Lam}\ x.e)$ is equal to $\mathit{Lam}\ x.\theta(e)$ and evaluates to $\mathit{Lam}\ x.\theta(e)$ (Line 19). Using this and (*ii*) we can take v to be $\mathit{Lam}\ x.\theta(e)$ and conclude (Line 20) with

$$\theta(\mathit{Lam}\ x.e) \Downarrow v \wedge v \in V\ T.$$

□

The proof for Theorem 5.1 is now by instantiating θ to be the identity substitution $[]$ and the facts that $[]\ \mathit{Vcloses}\ []$ and that $V\ T$ is a set of values (the latter can be shown by a simple induction over the type T).

6 Conclusion

We have described a formalisation of some very typical proofs from SOS. The main point we want to convey is that such proofs can be done relatively easily using Nominal Isabelle. This must however be qualified insofar as Nominal Isabelle only supports languages involving simple, lambda-calculus-like binders. Although they can be of different type and can be iterated (see [17]), more complicated binding structures, such as binding a finite set of variables, are not yet supported. One can encode such general binders using the simple binding, but this makes proofs quite complicated. The second qualification we must mention is that even though we based our formalisation on α-equivalence classes, reasoning about them can be quite subtle. Many of the complications can be hidden from the user, for example by automatically providing strong versions of the induction principles, but they cannot be hidden completely. Most notably issues about α-equivalence show up in definitions of functions by structural recursion. On "paper" one is usually not concerned with questions about whether a function is compatible with α-equivalence classes or whether it leads to an inconsistency. In Nominal Isabelle, conditions need to be verified which guarantee the compatibility with α-equivalence classes. Nevertheless most presented formal proofs really proceed like the corresponding informal proofs done with "pencil-and-paper".

Acknowledgements: To do this formalisation we have been inspired by Adam Chlipala who mailed some of the problems as challenge on the PoplMark list. We are very grateful to Nick Benton who outlined to us the proof for the termination property described in Sec. 5.

References

[1] B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering Formal Metatheory. In *Proc. of the 35rd Symposium on Principles of Programming Languages (POPL)*, pages 3–15. ACM, 2008.

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

[3] S. Berghofer and C. Urban. A Head-to-Head Comparison of de Bruijn Indices and Names. In *Proc. of the International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP)*, ENTCS, pages 46–59, 2006.

[4] S. Berghofer and C. Urban. Nominal Inversion Principles. In *Proc. of the 21th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5170 of *LNCS*, pages 71–85, 2008.

[5] K. Crary. *Advanced Topics in Types and Programming Languages*, chapter on Logical Relations and a Case Study in Equivalence Checking, pages 139–160. MIT Press, 2005.

[6] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1958.

[7] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th ACM Symp. Principles of Programming Languages*, pages 207–212, 1982.

[8] J. H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Harper & Row, 1986.

[9] J. McKinna and R. Pollack. Some Type Theory and Lambda Calculus Formalised. *Journal of Automated Reasoning*, 23(1-4), 1999.

[10] T. Nipkow and L. C. Paulson. Proof Pearl: Defining Functions Over Finite Sets. volume 3603 of *LNCS*, pages 385–396. Springer Verlag, 2005.

[11] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

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

[13] A. M. Pitts. Alpha-Structural Recursion and Induction. *Journal of the ACM*, 53:459–506, 2006.

[14] G. Plotkin. A Structural Approach to Operational Semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, 2004.

[15] C. Urban. Nominal Techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008.

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

[17] 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.

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

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