Nominal Inversion Principles

Stefan Berghofer and Christian Urban

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

Abstract. When reasoning about inductively defined predicates, such as typing judgements or reduction relations, proofs are often done by inversion, that is by a case analysis on the last rule of a derivation. In HOL and other formal frameworks this case analysis involves solving equational constraints on the arguments of the inductively defined predicates. This is well-understood when the arguments consist of variables or injective term-constructors. However, when alpha-equivalence classes are involved, that is when term-constructors are not injective, these equational constraints give rise to annoying variable renamings. In this paper, we show that more convenient inversion principles can be derived where one does not have to deal with variable renamings. An interesting observation is that our result relies on the fact that inductive predicates must satisfy the variable convention compatibility condition, which was introduced to justify the admissibility of Barendregt's variable convention in rule inductions.

1 Introduction

Inductively defined predicates play an important role in formal methods; they are defined by a set of introduction rules and come equipped with rule induction and inversion principles. A typical example of an inductive predicate is beta-reduction defined by the four rules

$$\frac{}{App\ (Lam\ x.s_1)\ s_2 \longrightarrow_\beta s_1[x{:=}s_2]}\,b_1 \qquad \frac{s_1 \longrightarrow_\beta s_2}{App\ s_1\ t \longrightarrow_\beta App\ s_2\ t}\,b_2$$
$$\frac{s_1 \longrightarrow_\beta s_2}{App\ t\ s_1 \longrightarrow_\beta App\ t\ s_2}\,b_3 \qquad \frac{s_1 \longrightarrow_\beta s_2}{Lam\ x.s_1 \longrightarrow_\beta Lam\ x.s_2}\,b_4 \tag{1}$$

where $\_[\_{:=}\_]$ stands for capture-avoiding substitution. Another is the typing predicate for simply-typed lambda-terms defined by the rules

$$\frac{valid\ \Gamma \quad (x, T) \in \Gamma}{\Gamma \vdash Var\ x : T}\,t_1 \qquad \frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash App\ t_1\ t_2 : T_2}\,t_2$$
$$\frac{(x, T_1){::}\Gamma \vdash t : T_2}{\Gamma \vdash Lam\ x.t : T_1 \to T_2}\,t_3 \tag{2}$$

where the typing contexts Γ are lists of (variable name,type)-pairs, $\in$ stands for list membership and $::$ for list-cons. The premise $valid\ \Gamma$ in the first typing rule is another inductive predicate which states that the typing context must not contain repeated occurrences of a variable name. This can be defined as follows:

$$\frac{}{valid\ []}\,v_1 \qquad \frac{valid\ \Gamma \quad x \# \Gamma}{valid\ ((x, T){::}\Gamma)}\,v_2 \tag{3}$$

where [] stands for the empty typing context and $x \# \Gamma$ states that the variable name x does not occur in Γ.

The rule induction and inversion principles are the main thrust behind these definitions: they provide the infrastructure for convenient reasoning about inductive predicates. This is illustrated by the proof of the following lemma establishing that beta-reduction preserves typing.

Lemma 1 (Type Preservation). *If $\Gamma \vdash u : U$ and $u \longrightarrow_\beta u'$ then $\Gamma \vdash u' : U$.*

Type preservation can be proved by a rule induction on $\Gamma \vdash u : U$. This gives rise to three subgoals:

$$\begin{array}{rl} (i) & \mathit{Var}\ x \longrightarrow_\beta u' \wedge \ldots \Rightarrow \Gamma \vdash u' : T \\ (ii) & \mathit{App}\ t_1\ t_2 \longrightarrow_\beta u' \wedge \ldots \Rightarrow \Gamma \vdash u' : T_2 \\ (iii) & \mathit{Lam}\ x.t \longrightarrow_\beta u' \wedge \ldots \Rightarrow \Gamma \vdash u' : T_1 \rightarrow T_2 \end{array}$$

where we omitted some of the side-assumptions. The proof then proceeds by a case analysis, called *inversion*, of the assumptions about $\longrightarrow_\beta$.

In general, inversion is a reasoning principle that applies to any instance of an inductive predicate occurring in the assumptions; it relies on the observation that this instance must have been derived by at least one of the rules by which the inductive predicate is defined. In informal reasoning one therefore matches the assumption with the conclusion of every rule and tests whether the assumption and conclusion match. We will refer to this kind of informal reasoning as *inversion by matching* and describe it next.

In the case *(i)*, the assumption $\mathit{Var}\ x \longrightarrow_\beta u'$ matches with no conclusion in (1). Therefore this is an impossible case, which implies that the goal $\Gamma \vdash u' : T$ holds trivially.

In the case *(ii)*, the matching of $\mathit{App}\ t_1\ t_2 \longrightarrow_\beta u'$ with the conclusions in (1) succeeds in case of b_1, b_2 and b_3, and therefore three cases need to be considered. Let us first analyse the case corresponding to the rule

$$\frac{s_1 \longrightarrow_\beta s_2}{\mathit{App}\ s_1\ t \longrightarrow_\beta \mathit{App}\ s_2\ t}\,b_2$$

In this case we know for some s_2 that $u' = \mathit{App}\ s_2\ t_2$ (since t_1 matches with s_1, and t with t_2). By induction we can infer that $\Gamma \vdash s_2 : T_1 \rightarrow T_2$ and $\Gamma \vdash t_2 : T_1$ hold. Consequently, $\Gamma \vdash u' : T_2$ holds.

Continuing with our informal reasoning, the case of beta-reduction, i.e. $\mathit{App}\ (\mathit{Lam}\ x.s_1)\ s_2 \longrightarrow_\beta s_1[x{:=}s_2]$, goes as follows: For some term s_1, u' is equal to $s_1[x{:=}t_2]$ and t_1 equal to $\mathit{Lam}\ x.s_1$. The latter equation gives us that $\Gamma \vdash \mathit{Lam}\ x.s_1 : T_1 \rightarrow T_2$ and $\Gamma \vdash t_2 : T_1$ hold. To complete the proof we need the substitutivity lemma:

Lemma 2 (Type Substitutivity).
If $(x, U)::\Gamma \vdash t : T$ and $\Gamma \vdash u : U$ then $\Gamma \vdash t[x{:=}u] : T$.

whose proof we omit. For this lemma to be useful, we have to invert the typing judgement $\Gamma \vdash \mathit{Lam}\ x.s_1 : T_1 \rightarrow T_2$. The informal inversion by matching gives us the desired result: this judgement matches with the conclusion of the rule t_3 and we obtain $(x, T_1)::\Gamma \vdash s_1 : T_2$. So we can conclude in this case by using Lemma 2 (similarly in all remaining cases).

The point of these calculations is to show that the inversion by matching is very natural and convenient. It is also very typical in programming language research: similar proofs are described for System $F_{<:}$ in the POPLmark challenge (see Appendix of [2]). The contribution of this paper is to make this informal reasoning formal. The problem we have to solve for this arises from the fact that the examples above contain lambda-terms, where the term constructor *Lam* is not *injective*. By this we mean the property that in general one *cannot* infer from the equation

$$Lam\ x.t = Lam\ x'.t'$$

that

$$x = x' \quad \text{and} \quad t = t'$$

hold. This is in contrast to the injective term constructors *Var* and *App* where we have the implications

$$\begin{array}{rcl} Var\ x = Var\ x' & \Rightarrow & x = x' \\ App\ t\ s = App\ t'\ s' & \Rightarrow & t = t' \wedge s = s' \end{array}$$

Why the lack of injectivity leads to problems with formal inversion principles is explained in the next section. Section 3 characterises the form of rules in inductive definitions, Section 4 recalls some notions from the nominal logic work [7, 9] and Section 5 describes the condition for variable-convention compatibility and gives the proof for our main result. Examples are described in Section 6 and Section 7 concludes and mentions related work.

2 Formal Inversion Principles

Unfortunately, the *formal* reasoning in systems such as HOL, Coq and LEGO is subtly different from the informal inversion by matching illustrated in the Introduction: instead of matching two instances of a relation, the formal inversion principles in these systems require equality constraints to be solved.

Consider the inversion principles given in Fig. 1, which are formally derived by Isabelle/HOL for beta-reduction and typing. Both inversion principles can be employed to prove a proposition P from the assumption $u_1 \longrightarrow_\beta u_2$ and $\Delta \vdash u : U$, respectively. Their general structure is as follows: each premise of the inversion rule corresponds to a rule of the inductive predicate. These premises are implications whose right-hand side is the proposition P, and whose left-hand side are conjunctions (note also in each case the outermost universal quantification ranging over the entire implication). The elements of these conjunctions can be divided into two parts: the first part consists of equality constraints expressing the equality between the arguments of the predicate to be inverted and the arguments of each conclusion in the inductive definition; the second part consists of the premises of the corresponding rule.

Returning to our running example of proving the type-preservation lemma, let us analyse how the formally derived inversion principles given in Fig. 1 behave. The case *(i)* in Lemma 1 required us to prove

$$Var\ x \longrightarrow_\beta u' \wedge \ldots \Rightarrow \Gamma \vdash u' : T$$

$$\frac{\begin{array}{l}\forall\, x\, s_2\, s_1.\ u_1 = \mathit{App}\ (\mathit{Lam}\ x.s_1)\ s_2 \wedge u_2 = s_1[x{:=}s_2] \Rightarrow P\\ \forall\, s_1\ s_2\ t.\ u_1 = \mathit{App}\ s_1\ t \wedge u_2 = \mathit{App}\ s_2\ t \wedge s_1 \longrightarrow_\beta s_2 \Rightarrow P\\ \forall\, s_1\ s_2\ t.\ u_1 = \mathit{App}\ t\ s_1 \wedge u_2 = \mathit{App}\ t\ s_2 \wedge s_1 \longrightarrow_\beta s_2 \Rightarrow P\\ \forall\, s_1\ s_2\ x.\ u_1 = \mathit{Lam}\ x.s_1 \wedge u_2 = \mathit{Lam}\ x.s_2 \wedge s_1 \longrightarrow_\beta s_2 \Rightarrow P\end{array}}{u_1 \longrightarrow_\beta u_2 \Rightarrow P} \quad (4)$$

$$\frac{\begin{array}{l}\forall\, \Gamma\ x\ T.\ \Delta = \Gamma \wedge u = \mathit{Var}\ x \wedge U = T \wedge \mathit{valid}\ \Gamma \wedge (x, T) \in \Gamma \Rightarrow P\\ \forall\, t_1\ T_1\ T_2\ t_2.\ \Delta = \Gamma \wedge u = \mathit{App}\ t_1\ t_2 \wedge U = T_2 \wedge \Gamma \vdash t_1 : T_1 \to T_2 \wedge \Gamma \vdash t_2 : T_1 \Rightarrow P\\ \forall\, x\ T_1\ \Gamma\ t\ T_2.\ \Delta = \Gamma \wedge u = \mathit{Lam}\ x.t \wedge U = T_1 \to T_2 \wedge (x, T_1){::}\Gamma \vdash t : T_2 \Rightarrow P\end{array}}{\Delta \vdash u : U \Rightarrow P} \quad (5)$$

Fig. 1. Inversion principles derived by Isabelle/HOL for the inductive predicates beta-reduction and typing.

If we use inversion principle for $\longrightarrow_\beta$ (i.e. (4)) and invert $\mathit{Var}\ x \longrightarrow_\beta u'$, we obtain the following four subgoals:

$$\begin{array}{l}\forall\, x'\, s_2\, s_1.\ \mathit{Var}\ x = \mathit{App}\ (\mathit{Lam}\ x'.s_1)\ s_2 \wedge u' = s_1[x'{:=}s_2] \wedge \ldots \Rightarrow \Gamma \vdash u' : T\\ \forall\, s_1\ s_2\ t.\ \mathit{Var}\ x = \mathit{App}\ s_1\ t \wedge u' = \mathit{App}\ s_2\ t \wedge s_1 \longrightarrow_\beta s_2 \wedge \ldots \Rightarrow \Gamma \vdash u' : T\\ \forall\, s_1\ s_2\ t.\ \mathit{Var}\ x = \mathit{App}\ t\ s_1 \wedge u' = \mathit{App}\ t\ s_2 \wedge s_1 \longrightarrow_\beta s_2 \wedge \ldots \Rightarrow \Gamma \vdash u' : T\\ \forall\, s_1\ s_2\ x'.\ \mathit{Var}\ x = \mathit{Lam}\ x'.s_1 \wedge u' = \mathit{Lam}\ x'.s_2 \wedge s_1 \longrightarrow_\beta s_2 \wedge \ldots \Rightarrow \Gamma \vdash u' : T\end{array}$$

The left-hand sides of these subgoals all reduce to *False* because the term constructors are in conflict (*Var* can never be equal to *App*). Therefore we can quickly, like in the informal reasoning, discharge all subgoals.

In case *(ii)* where we invert $\mathit{App}\ t_1\ t_2 \longrightarrow_\beta u'$, we obtain the following four subgoals:

$$\begin{array}{l}\forall\, x\, s_2\, s_1.\ \mathit{App}\ t_1\ t_2 = \mathit{App}\ (\mathit{Lam}\ x.s_1)\ s_2 \wedge u' = s_1[x{:=}s_2] \wedge \ldots \Rightarrow \Gamma \vdash u' : T\\ \forall\, s_1\ s_2\ t.\ \mathit{App}\ t_1\ t_2 = \mathit{App}\ s_1\ t \wedge u' = \mathit{App}\ s_2\ t \wedge s_1 \longrightarrow_\beta s_2 \wedge \ldots \Rightarrow \Gamma \vdash u' : T\\ \forall\, s_1\ s_2\ t.\ \mathit{App}\ t_1\ t_2 = \mathit{App}\ t\ s_1 \wedge u' = \mathit{App}\ t\ s_2 \wedge s_1 \longrightarrow_\beta s_2 \wedge \ldots \Rightarrow \Gamma \vdash u' : T\\ \forall\, s_1\ s_2\ x.\ \mathit{App}\ t_1\ t_2 = \mathit{Lam}\ x.s_1 \wedge u' = \mathit{Lam}\ x.s_2 \wedge s_1 \longrightarrow_\beta s_2 \wedge \ldots \Rightarrow \Gamma \vdash u' : T\end{array}$$

The fourth subgoal can again be discharged because of the conflicting equality between *App* and *Lam*. The reasoning in the second and third is very similar with the informal inversion by matching, because the *App*-term constructor is injective and therefore we can infer

$$\begin{array}{lcl}\mathit{App}\ t_1\ t_2 = \mathit{App}\ s_1\ t & \Rightarrow & t_1 = s_1 \wedge t_2 = t,\ \text{and}\\ \mathit{App}\ t_1\ t_2 = \mathit{App}\ t\ s_1 & \Rightarrow & t_1 = t \wedge t_2 = s_1\end{array} \quad (6)$$

which are the same equations we would have got by the informal inversion by matching.

The first subgoal (corresponding to b_1) is more complicated: although we obtain by injectivity of *App* the equations $t_1 = \mathit{Lam}\ x.s_1$ and $t_2 = s_2$, we will encounter problems with inverting the typing judgement $\Gamma \vdash \mathit{Lam}\ x.s_1 : T_1 \to T_2$. That is, we will not be able to infer that $(x, T_1){::}\Gamma \vdash s_1 : T_2$ holds. This is because *Lam* is not injective and we cannot reason as in (6).

We encounter the same problem with the reasoning in case *(iii)*. There we have to invert the reduction $\mathit{Lam}\ x.t \longrightarrow_\beta u'$ and obtain by using the first inversion principle from (4) the following four subgoals:

$$\forall x'\, s_2\, s_1.\ Lam\ x.t = App\ (Lam\ x'.s_1)\ s_2 \wedge u' = s_1[x'{:=}s_2] \Rightarrow \Gamma \vdash u' : T_1 \to T_2$$
$$\forall s_1\ s_2\ t.\ Lam\ x.t = App\ s_1\ t \wedge u' = App\ s_2\ t \wedge s_1 \longrightarrow_\beta s_2 \Rightarrow \Gamma \vdash u' : T_1 \to T_2$$
$$\forall s_1\ s_2\ t.\ Lam\ x.t = App\ t\ s_1 \wedge u' = App\ t\ s_2 \wedge s_1 \longrightarrow_\beta s_2 \Rightarrow \Gamma \vdash u' : T_1 \to T_2$$
$$\forall s_1\ s_2\ x'.\ Lam\ x.t = Lam\ x'.s_1 \wedge u' = Lam\ x'.s_2 \wedge s_1 \longrightarrow_\beta s_2 \Rightarrow \Gamma \vdash u' : T_1 \to T_2$$

Again the first three cases reduce to *False*. However in the fourth case we end up with solving the equation

$$Lam\ x.t = Lam\ x'.s_1 \tag{7}$$

where the variables x' and s_1 are universally quantified (that is we cannot choose them). Since *Lam* is not injective, the only way to solve this equation is to unfold the definition of alpha-equivalence, which in the Nominal Datatype Package gives us the cases

$$\begin{array}{ll} (i) & x = x' \wedge t = s_1 \quad \textbf{or} \\ (ii) & x \neq x' \wedge t = (x\,x')\bullet s_1 \wedge x \# s_1 \end{array}$$

where $(x\,x')$ is a permutative renaming of x and x', and $x \# s_1$ stands for x not occurring freely in s_1, see [7]. While the first case is easy to deal with (the induction hypothesis is immediately applicable), the second leads to the following proof state:

$$x \neq x' \wedge x \# s_1 \wedge s_1 \longrightarrow_\beta s_2 \wedge \ldots \Rightarrow \Gamma \vdash Lam\ x'.s_2 : T_1 \to T_2$$

with the induction hypothesis

$$\forall s'.\ (x\,x')\bullet s_1 \longrightarrow_\beta s' \Rightarrow (x, T_1)::\Gamma \vdash s' : T_2$$

Here the formal reasoning starts to hurt, as it is much harder than the informal inversion by matching. As one can see, the induction hypothesis is not directly applicable: we know $s_1 \longrightarrow_\beta s_2$ but we need that $(x\,x')\bullet s_1$ reduces to some term. Also the induction hypothesis gives us a typing-judgement involving the variable x, but we need one for x'. The most direct way to complete this case requires the following side lemmas:

Lemma 3.
(i) If $s_1 \longrightarrow_\beta s_2$ *then* $(x\,x')\bullet s_1 \longrightarrow_\beta (x\,x')\bullet s_2$.
(ii) If $x \# s_1$ *and* $s_1 \longrightarrow_\beta s_2$ *then* $x \# s_2$.

where, interestingly, the second is a property specific to beta-reduction.

Clearly, inverting $Lam\ x.t \longrightarrow_\beta u'$ in this way is not very convenient and the same difficulties arise if we try to invert $\Gamma \vdash Lam\ x.s_1 : T_1 \to T_2$ using (5) as needed in the *App*-case above. In contrast, inverting inductive predicates based on the locally nameless approach to binders (see [3]) is much simpler, because there all term constructors are injective—even *Lam*. We show in this paper that we can obtain stronger inversion principles (than given in Fig. 1), where they are stronger in the sense that we can avoid the renaming of the binder, as long as the binder is sufficiently fresh. In this way we can follow quite closely the informal reasoning of inversion by matching an assumption with all rules.

These strong inversion principles will depend on the inductive predicates to satisfy the *variable convention compatibility condition*, short *vc-condition*. The reason for this condition is that the informal reasoning (i.e. inversion by matching) can lead to faulty reasoning when alpha-equivalence classes are involved. Consider the following inductive definition of a two-place predicate (both arguments are alpha-equated lambda-terms)

$$\frac{}{\mathit{Var}\ x \hookrightarrow \mathit{Var}\ x} \qquad \frac{}{\mathit{App}\ t_1\ t_2 \hookrightarrow \mathit{App}\ t_1\ t_2} \qquad \frac{t \hookrightarrow t'}{\mathit{Lam}\ x.t \hookrightarrow t'} \tag{8}$$

Now choose two distinct variables, say x and y with $x \neq y$. A simple calculation shows that $\mathit{Lam}\ x.\mathit{Var}\ x \hookrightarrow \mathit{Var}\ x$ can be derived using the rules above. Therefore we can use it as an assumption. Since we are working with alpha-equated lambda terms, we have that $\mathit{Lam}\ x.\mathit{Var}\ x = \mathit{Lam}\ y.\mathit{Var}\ y$ and therefore also $\mathit{Lam}\ y.\mathit{Var}\ y \hookrightarrow \mathit{Var}\ x$ must hold. Next we apply the inversion principle naively to the latter instance of the relation, i.e. we invert by matching this instance with the conclusions of the rules shown in (8). Only the third rule matches, yielding the fact $\mathit{Var}\ y \hookrightarrow \mathit{Var}\ x$. Next we invert this instance of the relation: the first rule matches, enabling us to infer that $x = y$ holds. This, however, contradicts the assumption that x and y are distinct. The vc-condition will protect us from this kind of faulty reasoning.

3 Inductive predicates

An inductive predicate, say R, is defined by a finite set of rules r_i

$$\frac{B_1}{R\ ts_1}r_1 \qquad \ldots \qquad \frac{B_n}{R\ ts_n}r_n \tag{9}$$

where in the premises the B_i are HOL-formulae possibly containing R and where in the conclusion the ts_i are the arguments of the predicate R. The ts_i are HOL-terms, which for the purposes of this paper we can assume to be either variables or constructed by term constructors. Again for the purposes of this paper HOL-formulae will be the ones given by the grammar

$$B ::= P\ ts \mid B_1 \wedge B_2 \mid B_1 \vee B_2 \mid B_1 \longrightarrow B_2 \mid \neg\, B \mid \forall x.\ B\ x \mid \exists x.\ B\ x$$

where P stands for atomic predicates and ts are the arguments of P. In (9) we have the usual assumption that the premises can contain the predicate R in positive position only (see [1]). However, the B_i can contain other predicates, these are usually called side-conditions. For example our typing rule t_1 has the side-condition concerning $\in$ and *valid* as premise.

In what follows it is convenient to have the notations $t[xs]$, where the xs contain all the variables of t, and $B[ys]$, where ys includes the free variables of B (in B some variables might be bound because of the universal and existential quantifiers). The meaning of a rule in (9) is then the implication

$$\forall xs_i.\ \ B_i[xs_i] \Rightarrow R\ \ ts[xs_i]$$

where each xs_i includes all free variables in r_i. That means every instantiation of the free variables in r_i will result in an instance of this rule. With the rules given in (9) comes the following inversion principle

$$\frac{\begin{array}{ll} \forall\, xs_1.\ ss = ts_1[xs_1] \wedge B_1[xs_1] \Rightarrow P & \text{rule } r_1 \\ \vdots & \\ \forall\, xs_n.\ ss = ts_n[xs_n] \wedge B_n[xs_n] \Rightarrow P & \text{rule } r_n \end{array}}{R\ ss \Rightarrow P} \tag{10}$$

where the ts_i correspond to the arguments in the conclusion of each rule and the B_i to the premises (not also that the xs_i do not include any of the free variables in ss and P). The inversion principles given for $\longrightarrow_\beta$ and the typing rues in Fig. 1 are instances of (10). We refer to this inversion principle as the *weak inversion principle*. As we have shown in Section 2: when applying the weak inversions to cases involving non-injective term constructors, we need to analyse cases involving annoying variable renamings. We will show later that a strong inversion principle can be derived from the weak one and using the strong one we can avoid the renamings.

4 Nominal Logic Work

Before we proceed, we introduce some necessary notions from the nominal logic work [7, 9]. We assume that there are countably infinitely many names, which can be used as binders. We base our description on *permutation actions* and on the notion of *support*. The support of an object will, for the purposes of this paper, coincides with the set of free names of that object. For details and a proper definition of support see [8]. A name a is *fresh* w.r.t. an object, say t, provided that it is not free in t; we write this as $a \# t$. Note that if t has finitely many free variables, then there exists a fresh variable w.r.t. t. We will also use the auxiliary notation $a \# ts$, in which ts stands for a collection of objects $t_1,\ldots,t_n$, to mean $a \# t_1,\ldots, a \# t_n$. We further generalise this notation to a collection of names, namely $as \# ts$, which means $a_1 \# ts,\ldots, a_m \# ts$.

Permutations are finite lists of swappings (i.e., pairs of variables). 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, list append (i.e. $\pi_1 @ \pi_2$) for the composition of two permutations and list reversal (i.e. π^{-1}) for the inverse of a permutation. We define the permutation action over the structure of types in HOL. The point of the permutation action is to push permutations inside the structure of every object, renaming names on the way. A permutation acting on names is therefore defined as follows:

$$\begin{aligned} [] \bullet a &= a \\ (a, b)::\pi \bullet c &= \begin{cases} a & \text{if } \pi \bullet c = b \\ b & \text{if } \pi \bullet c = a \\ \pi \bullet c & \text{otherwise} \end{cases} \end{aligned} \tag{11}$$

The permutation action on lists, pairs and booleans is given by

$$\begin{aligned} \pi \bullet [] &= [] \\ \pi \bullet (x::xs) &= \pi \bullet x::\pi \bullet xs \\ \pi \bullet (x, y) &= (\pi \bullet x, \pi \bullet y) \\ \pi \bullet \mathit{True} &= \mathit{True} \\ \pi \bullet \mathit{False} &= \mathit{False} \end{aligned} \tag{12}$$

Notice the last two lines imply the fact that for every HOL-formula B the equality $\pi \bullet B = B$ holds. This is because HOL is a classical logic and every formula is either true or false. For alpha-equated lambda-terms we have

$$\begin{aligned} \pi \bullet \mathit{Var}\ x &= \mathit{Var}\ (\pi \bullet x) \\ \pi \bullet \mathit{App}\ t_1\ t_2 &= \mathit{App}\ (\pi \bullet t_1)\ (\pi \bullet t_2) \\ \pi \bullet \mathit{Lam}\ x.t &= \mathit{Lam}\ (\pi \bullet x).(\pi \bullet t) \end{aligned} \tag{13}$$

We can easily prove that the permutation actions in (11), (12) and (13) satisfy the following three properties:

$$\begin{array}{rl} \textit{(i)} & [] \bullet (\_) = (\_) \\ \textit{(ii)} & (\pi_1 \mathbin{@} \pi_2) \bullet (\_) = \pi_1 \bullet \pi_2 \bullet (\_) \\ \textit{(iii)} & \textit{If } \pi_1 \approx \pi_2 \textit{ then } \pi_1 \bullet (\_) = \pi_2 \bullet (\_). \end{array} \tag{14}$$

where in the last clause equality between two permutations, that is $\pi_1 \approx \pi_2$, is defined by the property that as $\pi_1 \bullet a = \pi_2 \bullet a$ holds for all names a. In the next section we need the following lemma about freshness and the permutation actions in (11), (12) and (13):

Proposition 1. *If* $a \# (\_)$ *and* $b \# (\_)$ *then* $(a\ b)\bullet(\_) = (\_)$.

The notion of *equivariance* is derived from the permutation actions:

Definition 1 (Equivariance [7]). *A HOL-term t, respectively a HOL-formula B, with free variables amongst xs is* equivariant *provided for all* π*, we have* $\pi \bullet t[xs] = t[\pi\bullet xs]$ *and* $\pi \bullet B[xs] = B[\pi\bullet xs]$.

From the definition of their permutation action, pairs, nil and list-cons are equivariant. For HOL-formulae we have:

$$\begin{aligned} \pi \bullet (A \wedge B) &= \pi \bullet A \wedge \pi \bullet B \\ \pi \bullet (A \vee B) &= \pi \bullet A \vee \pi \bullet B \\ \pi \bullet (A \longrightarrow B) &= \pi \bullet A \longrightarrow \pi \bullet B \\ \pi \bullet (\neg A) &= \neg\, \pi \bullet A \\ \pi \bullet (\forall x.\ P\ x) &= \forall x.\ \pi \bullet P\ (\pi^{-1} \bullet x) \\ \pi \bullet (\exists x.\ P\ x) &= \exists x.\ \pi \bullet P\ (\pi^{-1} \bullet x) \end{aligned} \tag{15}$$

Therefore for all the structures we consider in this paper we can move permutations inside the structures until they reach variables, therefore all structures we consider in paper will be equivariant.

For proving our main result in the next section it is convenient to refine our notation $ts[xs]$ and $B[xs]$ for indicating the free variables of ts and B. The reason is that some of these variables stand for names and those names are potentially in *binding positions*. By binding position we mean the x in $\mathit{Lam}\ x.t$. In what follows the notation $ts[as;xs]$ and $B[as;xs]$ will be used to indicate that the variables in binding position of the ts are included in as and the other variables of the ts are either in as or in xs (similarly for HOL-formulae). We extend this notation also to rules: by writing $r[as;xs]$ we mean rules of the form

$$\frac{B[as;xs]}{R\ ts[as;xs]}\, r_i[as;xs]$$

However, unlike in the notation for HOL-terms and HOL-formulae, we mean in $r_i[as;xs]$ that the *as* stand *exactly* for the variables occurring somewhere in r_i in binding position and the *xs* stand for the rest of variables. To see how this notation works out in our examples, reconsider the definitions for the relations given in (1) and (2). Using our notation for these rules, we have

$$\begin{array}{ll} b_1[x;s_1,s_2] & t_1[-;\Gamma,x,T] \\ b_2[-;s_1,s_2,t] & t_2[-;\Gamma,t_1,t_2,T_1,T_2] \\ b_3[-;s_1,s_2,t] & t_3[x;\Gamma,t,T_1,T_2] \\ b_4[x;s_1,s_2] & \end{array}$$

where '$-$' stands for no variable in binding position. An inductive definition for alpha-equivalence between lambda terms includes the two rules:

$$\frac{t_1 = t_2}{Lam\, x.t_1 = Lam\, x.t_2}\,a_1 \qquad \frac{x \neq y \quad t_1 = (x\, y)\bullet t_2 \quad x \# t_2}{Lam\, x.t_1 = Lam\, y.t_2}\,a_2$$

There our notation would be $a_1[x;t_1,t_2]$ and $a_2[x,y;t_1,t_2]$.

5 Strengthening of the Inversion Principle

In this section, we show how the "weak" inversion rules in (10) can be used to derive stronger inversion rules in which the equality constraints are formulated in such a way that they can be solved without having to rename variables.

We have seen in the example about $t \hookrightarrow t'$ from the Introduction that inversion principles involving alpha-equivalence classes require some care. In order to rule out the problematic case (and similar ones), we need to impose a condition on the rules of an inductive definition. It is interesting that the condition we impose is the same as the one introduced in [8] for justifying the admissibility of Barendregt's variable convention in rule inductions.

A rule is said to be *variable convention compatible*, or short *vc-compatible*, provided the following two properties are satisfied:

Definition 2 (Variable Convention Compatibility). *A rule $r[as;xs]$ with conclusion $R\ ts[as;xs]$ and premise $B[as;xs]$ is* vc-compatible *provided that:*

- *all HOL-terms and HOL-formulae occurring in r are equivariant, and*
- *the premise $B[as;xs]$ implies that $as \# ts[as;xs]$ holds and that the as are distinct.*

Note that if rule *r* does not contain any variable in binding position, then the second condition is vacuously true. The first condition ensures that the relation *R* is equivariant. The equivariance property will allow us to push permutations inside HOL-terms and HOL-formulae until they reach free variables.

If every introduction rule in an inductive definition satisfies these conditions, then the inversion principle can be strengthened. The strengthened version looks as follows

$$\frac{\begin{array}{ll}\forall\, xs_1.\ (bs_1 \# ss \wedge distinct(bs_1) \Rightarrow ss = ts_1[bs_1;xs_1] \wedge B_1[bs_1;xs_1]) \Rightarrow P & \text{rule } r_1\\ \vdots & \\ \forall\, xs_n.\ (bs_n \# ss \wedge distinct(bs_n) \Rightarrow ss = ts_n[bs_n;xs_n] \wedge B_n[bs_n;xs_n]) \Rightarrow P & \text{rule } r_n\end{array}}{R\ ss \Rightarrow P} \tag{16}$$

where for every rule $r_1,\ldots,r_n$ we have a case to analyse. In our notation the rules have the form $r_1[bs_1;xs_1],\ldots,r_n[bs_n;xs_n]$ where the bs_i are the variables in binding position. Note that in contrast to (10) the variables bs_i are no longer universally quantified, meaning that we are free to choose the names bs_i when we want to invoke the strong inversion principle. The only constraints we have is that the preconditions $bs_i \# ss \wedge distinct(bs_i)$ need to be satisfied. This will be the case if the bs_i are sufficiently fresh.

We now prove the main result of this paper: if the rules of an inductive definition are vc-compatible, then the strong inversion principle in (16) holds.

Theorem 1. *For an inductive definition of the predicate R, involving vc-compatible rules only, a strong inversion principle exists deriving the implication $R\ ss \Rightarrow P$.*

Proof. We need to establish $R\ ss \Rightarrow P$ using the implications indicated in (16). To do so we will use the weak inversion rule from (10). For each rule $r_i[as_i;xs_i]$ of the form

$$\frac{B[as_i;xs_i]}{R\ ts_i[as_i;xs_i]}$$

we have to analyse one case of the form

$$\forall\, as_i\ xs_i.\ ss = ts_i[as_i;xs_i] \wedge B_i[as_i;xs_i] \Rightarrow P$$

To show P in these cases we have available the fact from (16), namely

$$\forall\, xs_i.\ (bs_i \# ss \wedge distinct(bs_i) \Rightarrow ss = ts_i[bs_i;xs_i] \wedge B_i[bs_i;xs_i]) \Rightarrow P \tag{17}$$

We first assume that

$$ss = ts_i[as_i;xs_i] \tag{18}$$

$$B_i[as_i;xs_i] \tag{19}$$

hold. Since $r_i[as_i;xs_i]$ is assumed to be vc-compatible, we further have that

$$(a)\ as_i \# ts_i[as_i;xs_i] \qquad \text{and} \qquad (b)\ distinct(as_i) \tag{20}$$

hold. The proof then proceeds by choosing for every name a in as_i a fresh name c such that for all the cs_i the following hold (cs_i is the collection of all those c):

$$(a)\ cs_i \# ss \quad (b)\ cs_i \neq as_i \quad (c)\ cs_i \neq bs_i \quad (d)\ distinct(cs_i) \tag{21}$$

Such a sequence cs_i always exists: the first three properties can be obtained since the terms ss, as_i and bs_i stand for finitely supported objects—so a free variable always exists; the last can be obtained by choosing the c one after another avoiding the ones that have already been chosen. We now build the permutation

$$\pi \stackrel{\text{def}}{=} (b_n\ c_n) \ldots (b_1\ c_1)\ (a_n\ c_n) \ldots (a_1\ c_1)$$

The point of π is that when applied to the as_i we get $\pi \bullet as_i = bs_i$. This follows from the properties in (20.b), (21.b-d) and the fact that we can assume $distinct(bs_i)$ holds (see below). We next instantiate in (17) the xs_i with $\pi \bullet xs_i$ giving us

$$(bs_i \# ss \wedge distinct(bs_i) \Rightarrow ss = ts_i[bs_i;\pi \bullet xs_i] \wedge B_i[bs_i;\pi \bullet xs_i]) \Rightarrow P$$

So in order to show P, it suffices to prove

$$ss = ts_i[bs_i;\pi \bullet xs_i] \wedge B_i[bs_i;\pi \bullet xs_i] \tag{22}$$

under the assumptions

$$(a)\ bs_i \# ss \qquad \text{and} \qquad (b)\ distinct(bs_i) \tag{23}$$

From (23.a) and (18) we obtain $bs_i \# ts_i[as_i;xs_i]$. Using this, (20.a) and Lemma 1, we have that $\pi \bullet ts_i[as_i;xs_i] = ts_i[as_i;xs_i]$. Since the rule is equivariant we have that $\pi \bullet ts_i[as_i;xs_i] = ts_i[bs_i;\pi \bullet xs_i]$ and thus also the first conjunct of (22). The reasoning for the other conjunct is as follows: using (19) and the fact that B_i is a boolean we have that $\pi \bullet B_i[as_i;xs_i]$ holds. Again by equivariance of the rule, we can move the permutation inside to obtain $B_i[bs_i;\pi \bullet xs_i]$—the second conjunct of (22). This concludes the proof. □

Let us next describe how the stronger inversion principles simplify the formal reasoning in the type preservation lemma.

6 Examples

To use the strong inversion rules, we first have to make sure that the beta-reduction and typing relation are equivariant. For this we only have to observe that all constants (that is term constructors and functions) in the rules of $\longrightarrow_\beta$, typing and *valid* are equivariant. This follows either from the definition of the permutation action or is by a simple induction over the predicates (in our implementation Isabelle will infer this automatically). To show that the second condition in Definition 2 is satisfied we have to show that the binders are fresh w.r.t. the conclusions of the rule they appear in. That is a simple calculation for the rules

$$\frac{(x, T_1)::\Gamma \vdash t : T_2}{\Gamma \vdash Lam\ x.t : T_1 \to T_2}t_3 \qquad \frac{s_1 \longrightarrow_\beta s_2}{Lam\ x.s_1 \longrightarrow_\beta Lam\ x.s_2}b_4$$

$$\frac{\begin{array}{l}\forall\, s_2\; s_1.\; (y \# (u_1, u_2) \Rightarrow u_1 = App\;(Lam\; y.s_1)\; s_2 \wedge u_2 = s_1[y{:=}s_2] \wedge y \# s_2) \Rightarrow P\\ \forall\, s_1\; s_2\; t.\; u_1 = App\; s_1\; t \wedge u_2 = App\; s_2\; t \wedge s_1 \longrightarrow_\beta s_2 \Rightarrow P\\ \forall\, s_1\; s_2\; t.\; u_1 = App\; t\; s_1 \wedge u_2 = App\; t\; s_2 \wedge s_1 \longrightarrow_\beta s_2 \Rightarrow P\\ \forall\, s_1\; s_2.\; (x \# (u_1, u_2) \Rightarrow u_1 = Lam\; x.s_1 \wedge u_2 = Lam\; x.s_2 \wedge s_1 \longrightarrow_\beta s_2) \Rightarrow P\end{array}}{u_1 \longrightarrow_\beta u_2 \Rightarrow P} \qquad (24)$$

$$\frac{\begin{array}{l}\forall\, \Gamma\; x\; T.\; \Delta = \Gamma \wedge u = Var\; x \wedge U = T \wedge valid\; \Gamma \wedge (x, T) \in \Gamma \Rightarrow P\\ \forall\, t_1\; T_1\; T_2\; t_2.\; \Delta = \Gamma \wedge u = App\; t_1\; t_2 \wedge U = T_2 \wedge \Gamma \vdash t_1 : T_1 \to T_2 \wedge \Gamma \vdash t_2 : T_1 \Rightarrow P\\ \forall\, T_1\; \Gamma\; t\; T_2.\; (x \# (\Delta, u, U) \Rightarrow \Delta = \Gamma \wedge u = Lam\; x.t \wedge U = T_1 \to T_2 \wedge (x, T_1){::}\Gamma \vdash t : T_2) \Rightarrow P\end{array}}{\Delta \vdash u : U \Rightarrow P} \qquad (25)$$

Fig. 2. Strong inversion principles derived by the Nominal Datatype Package for the inductive predicates for beta reduction and typing.

In the first case we have to show that $x \# (\Gamma, Lam\; x.t, T_1 \to T_2)$ holds under the assumption that $(x, T_1){::}\Gamma \vdash t : T_2$. Since we can show by a routine induction that typing judgements only include *valid* contexts, we have that *valid* $((x, T_1){::}\Gamma)$ holds. From this we can infer that $x \# \Gamma$. We also know that $x \# Lam\; x.t$ (since x is abstracted) and that $x \# T_1 \to T_2$ (since types in the simply-typed lambda-calculus do not contain any variables). We can discharge the conditions in the other rule by similar arguments. However the condition will fail for the rule

$$\frac{}{App\;(Lam\; x.s_1)\; s_2 \longrightarrow_\beta s_1[x{:=}s_2]}\,b_1 \qquad (26)$$

because we cannot determine whether $x \# s_2$. However we can show that this beta-reduction rule is equivalent to the following more restricted rule

$$\frac{x \# s_2}{App\;(Lam\; x.s_1)\; s_2 \longrightarrow_\beta s_1[x{:=}s_2]}\,b_1' \qquad (27)$$

This is because we can choose a y such that $y \# (s_1, s_2)$ and alpha-rename $App\;(Lam\; x.s_1)\; s_2$ to $App\;(Lam\; y.(y\; x){\bullet}s_1)\; s_2$. Then apply the restricted rule to this term in order to obtain the reduct $((y\; x){\bullet}s_1)[y{:=}s_2]$. By a structural induction over s_1, we can show that this term is equal to $s_1[x{:=}s_2]$ as desired. The point of this "manoeuvre" is that we can show that the restricted rule for beta-reduction does satisfy the vc-condition.

The result of these calculations is that there are strengthened inversion rules for beta-reduction and the typing-relation. They are given in Fig. 2. Using them for the type preservation lemma, the second and third case are the same as with the weak inversion rule (4). In the first and fourth case, however, the user does not need to show the claim for an arbitrary variable x', but for a sufficiently freshly chosen one (it has to be fresh w.r.t. (u_1, u_2)). In the strong inversion for the typing rule we have that the cases for variables and applications are the same as with the weak inversion rule (5). In the case of lambda abstractions, the user can choose a x so that $x \# (\Delta, u, U)$. These choices will hugely simplify the formal reasoning. To give an impression of this fact we show next three lemmas in Isabelle/HOL proving special instances of inversion principles.

lemma *Ty-Lam-inversion*:
assumes *ty*: $\Gamma \vdash Lam\ x.t : T$ **and** *fc*: $x\#\Gamma$
shows $\exists\, T_1\ T_2.\ T = T_1 \to T_2 \wedge (x,T_1)::\Gamma \vdash t : T_2$
using *ty fc* **by** (*cases rule*: *typing.strong-cases*) (*auto simp add*: *alpha*)

lemma *Beta-Lam-inversion*:
assumes *red*: $Lam\ x.t \longrightarrow_\beta s$ **and** *fc*: $x\#s$
shows $\exists\, t'.\ s = Lam\ x.t' \wedge t \longrightarrow_\beta t'$
using *red fc* **by** (*cases rule*: *Beta.strong-cases*) (*auto simp add*: *alpha*)

lemma *Beta-App-inversion*:
assumes *red*: $App\ (Lam\ x.t)\ s \longrightarrow_\beta r$ **and** *fc*: $x\#(s,r)$
shows $(\exists\, t'.\ r = App\ (Lam\ x.t')\ s \wedge t \longrightarrow_\beta t') \vee$
$(\exists\, s'.\ r = App\ (Lam\ x.t)\ s' \wedge s \longrightarrow_\beta s') \vee (r = t[x{:=}s])$
using *red fc*
by (*cases rule*: *Beta.strong-cases*) (*auto dest*: *Beta-Lam-inversion simp add*: *alpha*)

These lemmas are needed frequently in proofs about structural operational semantics. As seen in Section 2, it would have been quite painful to derive them using the weak inversion principles. We use the *alpha*-rule in the proofs above in order to rewrite the trivial alpha-equivalence $Lam\ x.t = Lam\ x.s$ to $t = s$.

The Isar-proof of the complete type preservation lemma is given in Fig. 3. Lines 6 and 7 show the variable case. Lines 9-21 contain the steps for the case where a beta-reduction occurs (the other cases are automatic in Line 22). We first chose a fresh name x (Line 10); invert $App\ t_1\ t_2 \longrightarrow_\beta u'$ in Line 12 using the fresh x. In the only interesting case, we have that $\Gamma \vdash Lam\ x.s_1 : T_1 \to T_2$ holds (Line 15), which we can invert to $(x,\ T_1)::\Gamma \vdash s_1 : T_2$. To this we can apply the Lemma 2 (Line 20). In the lambda-case (Lines 24-31), we invert $Lam\ x.t \longrightarrow_\beta u'$. We know that x is fresh for u' by the strong induction (Line 5). We can apply the induction hypothesis in Line 28 and use the typing rule to conclude (Lines 30 and 31).

7 Conclusion and Related Work

As long as one is dealing with injective term constructors, the weak (or standard) inversion rules provided by Isabelle/HOL work similarly to the informal inversion by matching an assumption over the conclusions of inference rules. However, non-injective term constructors, such as *Lam* in the lambda-calculus, give rise to annoying variable renamings, and formal reasoning is quite different from and much more inconvenient than the informal inversion by matching. This was observed in [3], because in their locally nameless representation of binders, all term constructors are injective.

We have shown in this paper that if a binder is fresh with respect to the conclusion of the rule where the binder appears and the inductive predicate satisfies the vc-condition, then one can avoid the renamings. As a result the formal inversion principles are again as convenient the informal reasoning of inversion by matching—though the strong inversion principles only apply to vc-compatible inductive relations. In (8) we have shown that the informal inversion by matching can lead to faulty reasoning when the vc-condition is not satisfied. In our implementation this kind of faulty reasoning is

```
lemma type-preservation:
 assumes ty: Γ ⊢ u : U and red: u ⟶β u′
 shows Γ ⊢ u′ : U
using ty red
proof (nominal-induct avoiding: u′ rule: typing.strong-induct)
 case (ty-Var Γ x T)
 from ‹Var x ⟶β u′› show Γ ⊢ u′ : T by (cases) (simp-all)
next
 case (ty-App Γ t1 T1 T2 t2)
 obtain x::name where fc: x # (Γ, App t1 t2, u′) by (rule exists-fresh-var)
 from ‹App t1 t2 ⟶β u′› show Γ ⊢ u′ : T2 using fc
 proof (cases rule: Beta.strong-cases[where x=x and xa=x])
  case (Beta s2 s1)
  then have eqs: t1 = Lam x.s1  t2 = s2  u′ = s1[x:=s2] using fc by (simp-all)
  from ‹Γ ⊢ t1 : T1 → T2› have Γ ⊢ Lam x.s1 : T1 → T2 using eqs by simp
  then have (x,T1)::Γ ⊢ s1 : T2 using fc
   by (cases rule: typing.strong-cases) (auto simp add: alpha)
  moreover
  from ‹Γ ⊢ t2 : T1› have Γ ⊢ s2 : T1 using eqs by simp
  ultimately have Γ ⊢ s1[x:=s2] : T2 by (rule type-substitutivity)
  then show Γ ⊢ u′ : T2 using eqs by simp
 qed (auto intro: ty-App)
next
 case (ty-Lam x T1 Γ t T2)
 from ‹Lam x.t ⟶β u′› ‹x # u′›
 obtain s2 where t-red: t ⟶β s2 and eq: u′ = Lam x.s2
  by (cases rule: Beta.strong-cases) (auto simp add: alpha)
 have ih: t ⟶β s2 ⟹ (x,T1)::Γ ⊢ s2 : T2 by fact
 with t-red have (x,T1)::Γ ⊢ s2 : T2 by simp
 then have Γ ⊢ Lam x.s2 : T1 → T2 by (rule typing.ty-Lam)
 with eq show Γ ⊢ u′ : T1 → T2 by simp
qed
```

Fig. 3. An Isar-proof of the type preservation lemma in Isabelle/HOL.

prevented because the strong inversion principles are derived only when the user has verified the second part of the vc-condition (see Def. 2); the first part of that condition is verified automatically by observing that equivariant inductive predicates must be composed of equivariant components only.

What was surprising to us is that the strong inversion principles depend on the vc-condition that we introduced in previous work [8]. There, this condition was used to make sure that the variable convention in proofs by rule induction does not lead to faulty lemmas. An disadvantage of our approach is that in case of beta-reduction we have to use rule b_1' shown in (27) and so far we have no automatic method to derive from it the usual rule b_1 shown in (26).

The most closely related work to the one presented here is our own [8], where we study strong induction principles. Here we were concerned with inversion principles, which in our setting with non-injective term constructors are *not* a degenerated form of induction (as is usually the case). In contrast with that work [8], we also deal here with the case where rules include quantifiers. In the context of type theory, inversion principles have been studied by Cornes and Terrasse for the Coq proof assistant [4] and by McBride for the LEGO system [5]. McBride's implementation in LEGO uses an algorithm for solving equality constraints based on unification. The derivation of inversion principles for inductive sets in Isabelle's object logic HOL and ZF was first described by Paulson [6].

References

1. P. Aczel. An Introduction to Inductive Definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. Elsevier, 1977.
2. B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized Metatheory for the Masses: The POPLMARK Challenge. In T. Melham and J. Hurd, editors, *Theorem Proving in Higher Order Logics: TPHOLs 2005*, LNCS. Springer-Verlag, 2005. Available electronically at http://www.cis.upenn.edu/∼plclub/wiki-static/poplmark.pdf.
3. B. E. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In G. C. Necula and P. Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 3–15. ACM, 2008.
4. C. Cornes and D. Terrasse. Automating Inversion of Inductive Predicates in Coq. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers*, volume 1158 of *Lecture Notes in Computer Science*, pages 85–104. Springer, 1996.
5. C. McBride. Inverting Inductively Defined Relations in LEGO. In E. Giménez and C. Paulin-Mohring, editors, *Types for Proofs and Programs, International Workshop TYPES'96, Aussois, France, December 15-19, 1996, Selected Papers*, volume 1512 of *Lecture Notes in Computer Science*, pages 236–253. Springer, 1998.
6. L. C. Paulson. A fixedpoint approach to (co)inductive and (co)datatype definitions. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honor of Robin Milner*, pages 187–211. MIT Press, 2000.
7. A. M. Pitts. Nominal Logic, A First Order Theory of Names and Binding. *Information and Computation*, 186:165–193, 2003.
8. C. Urban, S. Berghofer, and M. Norrish. Barendregt's Variable Convention in Rule Inductions. In F. Pfenning, editor, *21st International Conference on Automated Deduction (CADE-21)*, volume 4603 of *Lecture Notes in Artificial Intelligence*, pages 35–50. Springer-Verlag, 2007.
9. 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.