Inverting Inductively Defined Relations in LEGO

Conor McBride

Department of Computer Science
University of Edinburgh

1 Introduction

Inverting an inductively defined relation essentially consists of observing that any of its inhabitants must be derivable by at least one of its inference rules.

For example, let us define $\leq$ inductively by the rules

$$\frac{}{\texttt{zero} \leq n} \leq_z \qquad \frac{m \leq n}{\texttt{suc}\; m \leq \texttt{suc}\; n} \leq_s$$

Now suppose we have the hypothesis, $x \leq zero$. Inverting, we see that, since $\texttt{zero}$ cannot equal $\texttt{suc}\; n$, only rule $\leq_z$ could have yielded this conclusion, thus we deduce that $x = \texttt{zero}$. Likewise, if we have $\texttt{suc}\; x \leq y$, inversion shows that only rule $\leq_s$ could have applied, hence $y = \texttt{suc}\, z$ for some z and, moreover $x \leq z$.

In essence, inverting a hypothesis gives us its 'predecessor' premises with respect to each rule, constrained by equations which reflect the validity of the rule instance.

The notion of 'inversion' has a history in logic [Pra65]. It is analogous to Clark's notion of the 'completion' of a logic program, used to give a semantics of 'negation as failure' [Cla78], and also bears considerable resemblance to the idea of 'definitional reflection' given by Hallnäs in [Hal91]. Given a hypothesis, we may gain useful information by asking how it can have come to be true—it is this natural mode of reasoning which inversion captures.

An inversion principle corresponds to case analysis on a derivation, rather than the full recursion inherent in an induction principle. Although it reduces a known inhabitant of the relation to the premises from which it follows, it provides no inductive hypotheses for those premises. However, as Burstall observes in [Bur96], an inversion is often all that is necessary to complete a proof by induction over a related structure. Indeed, rule induction for $\leq$ can be derived from its inversion principle by induction over the natural numbers.

The impetus behind our work is the inversion facility in the Coq system [Coq], originally implemented by Murthy, with subsequent elaboration by Cornes and

Terrasse [CT95]. Our approach differs from theirs in that it is centred on unification. For a given hypothesis to follow from a particular rule, it must unify with that rule's conclusion, and the premises, instantiated by the unifier, must hold.

Where the Coq package proves particular inversion lemmas for particular hypotheses, we prove a generic lemma for each inductively defined relation, internalising the unification problems as systems of equations. We then supply a tactic which solves a certain class of first-order unification problems—this can be applied uniformly to the subgoals generated by an inversion.

We believe that this approach yields a clearer presentation of inversion which is relatively simple to implement. We shall also see that it solves a larger class of problems.

2 Generic Inversion Lemmas

In general, an inductively defined relation $R\,\vec{t}$, where $\vec{t}$ is a sequence of parameters, may be given by a set of inference rules like so:

$$\frac{\vec{P}_1[\vec{x}_1]}{R\,\vec{t}_1[\vec{x}_1]}\,rule_1 \qquad \cdots \qquad \frac{\vec{P}_n[\vec{x}_n]}{R\,\vec{t}_n[\vec{x}_n]}\,rule_n$$

where $\vec{P_i}$ is a sequence of premises (perhaps involving R) and the $\vec{x_i}$ are the schematic variables for $rule_i$.

LEGO's `Inductive` command [Pol94] allows such relations to be represented as Dybjer-style inductive families of types [Dyb91], provided they fall within the positivity restrictions given by Luo in [Luo94]. The induction principle and reductions corresponding to the rules are then generated in accordance with Luo's characterisation and added as assumptions to the local context.

The generic inversion lemma which we shall use is as follows:

$$\frac{R\,\vec{s} \qquad \begin{array}{c} \forall \vec{x}_1 .\, \vec{s} = \vec{t}_1[\vec{x}_1] \rightarrow \vec{P}_1[\vec{x}_1] \rightarrow \Phi \\ \vdots \\ \forall \vec{x}_n .\, \vec{s} = \vec{t}_n[\vec{x}_n] \rightarrow \vec{P}_n[\vec{x}_n] \rightarrow \Phi \end{array}}{\Phi} \qquad (\dagger)$$

This should be recognisable, via currying of the higher order definition of disjunction, as the Clark-style completion lemma:

$$\forall \vec{s}.\ R\,\vec{s} \rightarrow \bigvee_{i=1}^{n} \exists \vec{x_i}.\ (\vec{s} = \vec{t_i} \wedge \vec{P_i}[\vec{x_i}])$$

Provided the equations are well-typed, we can generate this lemma as a type in LEGO and prove it by an easy induction on the derivation of $R\,\vec{s}$. The inductive case for $rule_i$ instantiates $\vec{s}$ to $\vec{t_i}[\vec{x}_i]$ and supplies the $\vec{P_i}[\vec{x}_i]$ as hypotheses, so the corresponding premise of the inversion lemma can be applied to prove Φ.

Observe that the effect of applying this lemma to an instance $R\,\vec{s}$ of the family R is to replace a goal Φ by a bunch of subgoals, one for each $rule_i$, like so:

$$\forall \vec{x}_i.\ \vec{s} = \vec{t_i}[\vec{x}_i] \to \vec{P_i}[\vec{x}_i] \to \Phi$$

The $\vec{s} = \vec{t_i}[\vec{x}_i]$ conditions capture the requirement that our hypothesis unifies with the conclusion of $rule_i$, while the $\vec{P_i}[\vec{x}_i]$ conditions force the rule's premises to hold. If there is no unifier, then the subgoal is vacuous. If we can establish a most general unifier, σ, then the subgoal reduces to

$$\sigma\vec{P_i}[\vec{x}_i] \to \sigma\Phi$$

In this light, our inversion lemma bears a strong resemblance to Tamaki and Sato's unfold transformations for logic programs [TS84] and Eriksson's proposed rule of definitional reflection [Eri91]:

$$\frac{\{\sigma\Gamma, \sigma C \vdash \sigma A\ :\ \sigma = \mathrm{mgu}(a,b)\ \text{for definition}\ b \Leftarrow C\}}{\Gamma, a \vdash A}$$

The crucial distinction is that the unification in our treatment is at the object level. In systems such as ALF [Mag95], where the type theory is strengthened by pattern matching, inversion comes for free.

The inversion facility added to LEGO establishes the generic lemma (†) for each R once and for all at the time R is defined. Reducing the subgoals generated for specific R-hypotheses then depends on our ability to solve the unification problems they contain.

The current implementation is limited by the requirement that the equations representing the unification problems be well-typed. This can fail to be the case when there is type dependency between parameters of the relation. For example, a relation $R : \Pi t : \mathtt{SET}.\, (\mathtt{list}\, t) \to \mathtt{Prop}$ would give rise to equations

$$s = t \to ls = lt \to \ldots$$

where ls has type $(\mathtt{list}\, s)$ but lt has type $(\mathtt{list}\, t)$. Inversion principles are not automatically generated in these cases. Similar problems can arise in expressing the injectivity of datatype constructors (see section 4.4).

3 First-Order Unification for Constructor Forms

In their survey of rule-based unification, [JK91], Jouannaud and Kirchner give a first-order unification algorithm, 'Tree-Unify', which is complete for what they call 'constructor forms'. We have adapted it for inductive datatypes in LEGO. A partial implementation of Tree-Unify is the basis of a tactic specifically intended to simplify the subgoals generated by inversion.

Given variables X, we define the constructor forms T of an inductive datatype with constructors C recursively as follows:

$$T ::= X \mid C\, T_1 \ldots T_n$$

Now, motivated by the structure of subgoals left by inversion, let us represent a unification problem by a sequence of equational premises in a goal:

$$s_1 = t_{i1} \rightarrow \ldots \rightarrow s_j = t_{ij} \rightarrow \Phi$$

We present the algorithm Tree-Unify in a refinement style. The backwards-directed transition rules in figure 1 are applied repeatedly until either the goal is proved or no equations remain. The leading equation, $s = t$ determines which rule is appropriate at each step in a syntax-directed way. Figure 2 tabulates this choice according to the form of s (ranging vertically) and t (ranging horizontally).

For example, if we have the equation $x = y$ for distinct variables x and y, then **coalescence** is chosen, removing the superfluous variable; if the two sides have distinct outermost constructors c and c', then **conflict** applies, proving the goal.

It is easily shown that each transition preserves the constructor form property of these unification problems.

$$\textbf{deletion}\quad \frac{\Phi}{x = x \to \Phi}$$

$$\textbf{coalescence}\quad \frac{[y \mapsto x]\Phi}{y = x \to \Phi}\ x, y \text{ distinct}$$

$$\textbf{conflict}\quad \frac{}{c\ \vec{s} = c'\ \vec{t} \to \Phi}\ c \neq c'$$

$$\textbf{injectivity}\quad \frac{\vec{s} = \vec{t} \to \Phi}{c\ \vec{s} = c\ \vec{t} \to \Phi}$$

$$\textbf{checking}\quad \frac{}{c\ \vec{s} = x \to \Phi}\ x \in \mathrm{FV}(\vec{s})$$

$$\textbf{elimination}\quad \frac{[x \mapsto c\ \vec{s}]\Phi}{c\ \vec{s} = x \to \Phi}\ x \notin \mathrm{FV}(\vec{s})$$

Fig. 1. Transition rules for Tree-Unify

$s = t$	t is x ($x \neq y$)	t is $c\ \vec{t}$	t is $c'\ \vec{t}$
s is x	**deletion**	apply symmetry	
s is y	**coalescence**		
s is $c\ \vec{s}$ ($c \neq c'$)	if $x \in \mathrm{FV}(\vec{s})$ then **checking** else **elimination**	**injectivity**	**conflict**

Fig. 2. Tree-Unify is syntax directed by leading equation $s = t$

Theorem 1. *Tree-Unify terminates.*

Proof. Examining the behaviour of the transitions, we find that **conflict** and **checking** cause termination directly.

Further, the side-condition on **elimination** ensures that both it and **coalescence** strictly reduce the number of distinct variables remaining in the problem.

The **deletion** and **injectivity** rules each reduce the total size of the terms in the problem, where the size of a term is simply the total number of variable references and constructor applications therein. Neither rule introduces new variables.

Hence termination follows by a lexicographic induction on the number of distinct variables in the problem, then the total size of the problem. □

The soundness of any implementation of Tree-Unify is ultimately guaranteed by the requirement that the synthesised proofs should typecheck. Nonetheless, it seems appropriate to make an informal argument for the soundness of the algorithm. We postpone discussion of the implementation difficulties until the next section.

Theorem 2. *Tree-Unify is sound.*

Proof. We examine each transition rule.

deletion is trivial. **coalescence** and **elimination** follow from the substitutivity of equality.

conflict holds because the equation $c\vec{s} = c'\vec{t}$ fails to have the Leibnitz property—by case analysis, we can check that $c\,\vec{s}$ has c as its head, while $c'\,\vec{t}$ does not.

injectivity is essentially the observation that equality is preserved by the application of the 'predecessor' functions which map $c\,\vec{t}$ to each t_i. (There is a practical problem constructing these 'predecessor' functions in the presence of dependently-typed arguments, as we shall see in section 4.4.)

This leaves **checking**. For any inductive datatype, we can define the strict subterm ordering $<$ and show by induction that $s < t \rightarrow s \neq t$. If $x \in FV(\vec{s})$ then x is a strict subterm of $c\,\vec{s}$, hence the hypothesis $c\,\vec{s} = x$ is absurd. □

We turn now to the question of completeness.

Theorem 3. *Given goal,*

$$s_1 = t_1 \rightarrow \ldots \rightarrow s_j = t_j \rightarrow \Phi$$

with s_i, t_i in constructor form, Tree-Unify will either prove the goal or reduce it to $\sigma\Phi$ where σ is a most general unifier for the initial j equations.

Proof. We have seen that if the head equation is in constructor form, there is always exactly one transition rule applicable, and that the transition rules preserve the constructor form property of the problem. Since Tree-Unify always terminates, it must either prove the goal or remove the leading j equations, applying a sequence of substitutions σ to the remainder of the goal. It remains, therefore, to show that this σ is a most general unifier for those equations. We shall make an argument by induction over the sequence of transition rules applied.

If Tree-Unify applies no transition rules, then there must be no constructor form equations—the identity substitution is trivially a most general unifier for the empty problem.

Now we must check that each transition rule preserves this completeness property. For **conflict** and **checking** there is nothing to prove, whilst for **deletion** and **injectivity**, the requirement is trivial.

Both **coalescence** and **elimination** may be treated as instances of

$$\frac{[x \mapsto s]\Phi}{s = x \rightarrow \Phi}\; x \notin FV(s)$$

Suppose, inductively, that the substitution sequence σ generated for $[x \mapsto s]\Phi$ is a most general unifier for that problem.

We must show that $\sigma \circ [x \mapsto s]$ is a most general unifier for $s = x \rightarrow \Phi$.

By hypothesis, σ unifies $[x \mapsto s]\Phi$, so $\sigma \circ [x \mapsto s]$ certainly unifies Φ. Clearly, it also unifies x with s. Hence $\sigma \circ [x \mapsto s]$ is certainly a unifier for $s = x \rightarrow \Phi$.

Now suppose some ρ unifies $s = x \rightarrow \Phi$. Necessarily, $\rho = \rho \circ [x \mapsto s]$. Hence, ρ unifies $[x \mapsto s]\Phi$.

Again, our inductive hypothesis tells us that, since σ is a most general unifier for $[x \mapsto s]\Phi$, $\rho = \tau \circ \sigma$ for some τ.

Hence $\rho = \rho \circ [x \mapsto s] = \tau \circ \sigma \circ [x \mapsto s]$.

Hence $\sigma \circ [x \mapsto s]$ is a most general unifier for $s = x \rightarrow \Phi$ as required. □

4 `Qnify`: A Partial Implementation of Tree-Unify

The previous section exhibited the unification algorithm Tree-Unify and some of its properties. In this section, we examine its implementation as the new LEGO tactic, **`Qnify`**[1], and point out some of the pragmatic issues and difficulties which arose in this task.

The representation of a unification problem as a goal with equational premises enables a very simple implementation which makes considerable use of existing data structures and tactics (see [LEGO]). The state of the problem is just the goal itself, viewed as a stack of equations. The transition rules correspond to refinement tactics executed according to a simple syntactic analysis of the head equation. The current implementation performs no reduction on terms and hence fails to detect expressions which convert to constructor applications, although the user can force this computation explicitly beforehand—the possibility of introducing some weak head normalisation into the mechanism is under investigation.

[1] pronounced 'k*unify*' to emphasise it is a unification tactic, and named after the substitution tactic `Qrepl`, on which it heavily relies

`Qnify` handles equations not susceptible to any of the transition rules by introducing them into the context and proceeding with the remainder of the problem. `Qnify` will introduce non-equational premises in the same way, in order to gain access to an equation further down the goal.

4.1 deletion

We remove the initial $x = x$ from the goal, simply by introducing it into the context.

4.2 coalescence

Given an initial $y = x$ premise, we introduce it to the context and use the existing `Qrepl` tactic to substitute one variable for the other in the remainder of the goal.

This begs the pragmatic question of which way round to make the substitution. A generic inversion lemma will often introduce variables which are not required for some applications, and our tactic is geared to eliminate them. We achieve this by choosing to substitute for the variable which has smaller scope. The remaining variable, being 'more global', tends to be more useful.

4.3 conflict

In their treatment of inversion [CT95], Cornes and Terrasse show us how to prove equations $c\,\vec{s} = c'\,\vec{t}$ absurd using the computational power of the calculus and the substitutive property of equality.

For each constructor c of each inductive type T, we construct a discriminator function $T\_\mathtt{is}\_c : T \to \mathtt{Prop}$ which returns $\top$ for any $c\,\vec{s}$ and $\bot$ otherwise.

If c' is any other constructor, we then have

$$\dfrac{\dfrac{c\,\vec{s} = c'\,\vec{t} \qquad \dfrac{\dfrac{}{\top}}{T\_\mathtt{is}\_c\,(c\,\vec{s})}\,equiv}{T\_\mathtt{is}\_c\,(c'\,\vec{t})}\,=_{subst}}{\bot}\,equiv$$

If requested, LEGO will generate the discriminator functions at declaration time for any simple inductive datatype. `Qnify` will then deploy them in **conflict** proofs.

4.4 injectivity

Again, our attempts to prove constructors injective follow the same pattern as those of Cornes and Terrasse, and have the same successes and failures.

If we can define a batch of suitable predecessor functions $c\_\mathtt{pred}_i$ which return t_i given $c\,\vec{t}$ and a suitably typed dummy value otherwise, then the corresponding injectivity proof is given by

$$\dfrac{\dfrac{c\,\vec{s} = c\,\vec{t} \qquad \dfrac{\dfrac{}{s_i = s_i}\,=_{refl}}{s_i = c\_\mathtt{pred}_i\,(c\,\vec{s})}\,equiv}{s_i = c\_\mathtt{pred}_i\,(c\,\vec{t})}\,=_{subst}}{s_i = t_i}\,equiv$$

However, the choice of dummy value is not always straightforward. For simple inductive datatypes, any element will do. If we construct the predecessor functions locally to the injectivity theorem, we can simply use the hypothetical s_i itself. For inductive families, the dummy values required may vary in type, and are frequently far from obvious. Hence injectivity theorems are only generated for simple inductive datatypes.

As with inversion principles, type dependency between the arguments of a constructor may prevent injected equalities from typechecking. For example, if constructor c has type $\Pi n : \mathtt{nat}.\,(\mathtt{vect}\ n) \rightarrow T$, then $c\ m\ u = c\ n\ v$ gives rise to the ill-typed injected equality $u = v$. At present, our implementation detects this problem and does not attempt to prove injectivity for such constructors, issuing a warning, but allowing the rest of the definition to go through.

4.5 checking

Given an equation $x = t$, substituting $[x \mapsto t]$ without checking that $x \notin \mathrm{FV}(t)$ runs the risk of infinite regress. However, automating the proof of the **checking** rule is far from easy. `Qnify` responds to such cases by warning the user, making no substitution and deleting the offending equation from the problem.

The strict subterm ordering argument presented in the previous section would require the automatic generation of much extra equipment for each inductive datatype. Such facilities would be of benefit to users in a wide variety of applications, but the task is large and has not been done.

In the mean time, a more specific technique for automating the disproof of $c\ \vec{s}[x] = x$ (arising from conversations with Andrew Adams) is to select one occurrence of x within $\vec{s}[x]$ and construct the recursive function `f` (given here in ML style):

$$\begin{array}{l} \texttt{fun f}\ c\ \vec{s}[x] = \texttt{suc}\ (\texttt{f}\ x) \\ \quad \mid \texttt{f}\quad \_ \quad = \texttt{zero} \end{array}$$

Applying `f` to both sides of the offending equation, we get

$$\texttt{suc}\ (\texttt{f}\ x) = \texttt{f}\ x$$

We need merely have proved in advance that $\forall n.\, \texttt{suc}\, n \neq n$ and the result follows at once.

This technique should be implementable easily in Coq, where pattern-matching definition and fixpoint recursion are both supported [Cor96, Gim94], allowing the function to be represented directly at the object level, whatever its depth of recursion. LEGO provides only one-step elimination rules for its inductive datatypes, requiring `f` to be 'compiled' into primitive recursive form. An experimental implementation of this procedure is in progress at time of writing.

4.6 elimination

Given an equation $x = t$, and having checked that $x \notin \mathrm{FV}(t)$, `Qnify` uses `Qrepl` to substitute $[x \mapsto t]$.

`Qnify` does not check that t is in constructor form. However, the occurrence check is sufficient to guarantee termination.

5 Examples and Comparative Study

In this section, we consider three example inversion proofs which show the power of our treatment and also help to justify some of the pragmatic choices made in the implementation. We will also examine similar facilities in other theorem-proving systems.

Our presentation follows the convention of labelling assumptions in the context with identifiers and outstanding proof obligations with ? symbols.

5.1 Induction and Inversion for $\leq$

This simple example shows a proof by natural number induction and $\leq$ inversion, where perhaps one might have expected rule induction. We shall prove

$$? : \forall x.\, \texttt{suc}\, x \leq x \longrightarrow \bot$$

As we have already remarked, rule induction on the $\leq$ premise is not required. Induction on x and inversion of $\leq$ is sufficient.

For the case $x = \texttt{zero}$, inverting the hypothesis $\texttt{suc}\,\texttt{zero} \leq \texttt{zero}$ yields subgoals which zero with a successor. **Qnify** proves both of these:

$$? : \forall n.\, \texttt{suc zero} = \texttt{zero} \longrightarrow \texttt{zero} = n \longrightarrow \bot$$
$$? : \forall m.\, \texttt{suc zero} = \texttt{suc}\, m \longrightarrow \forall n.\, \texttt{zero} = \texttt{suc}\, n \longrightarrow m \leq n \longrightarrow \bot$$

Observe how the implementation has placed the equations as far left as their dependencies permits. LEGO does not distinguish between 'schematic variables' like m and n here, and 'premises' like $m \leq n$. They are merely arguments to a dependently typed constructor function. It seems desirable to position the equations so that **Qnify** applies its substitutions as widely as possible, as we shall see when establishing the inductive step:

$$x\_ih : \texttt{suc}\, x \leq x \longrightarrow \bot$$
$$? : \texttt{suc}\,(\texttt{suc}\, x) \leq \texttt{suc}\, x \longrightarrow \bot$$

The interesting case of the inversion is then

$$? : \forall m.\, \texttt{suc}\,(\texttt{suc}\, x) = \texttt{suc}\, m \longrightarrow \forall n.\, \texttt{suc}\, x = \texttt{suc}\, n \longrightarrow m \leq n \longrightarrow \bot$$

Qnify applies injectivity and substitutes $[m \mapsto \texttt{suc}\, x]$ and $[n \mapsto x]$. Note that the positioning of the equations ensures that these apply to the $m \leq n$ premise, and that it is better to keep x, which occurs in the inductive hypothesis, than n which is local to the inversion. The simplified subgoal follows immediately:

$$? : \texttt{suc}\, x \leq x \longrightarrow \bot$$

5.2 The Advantage of Unification

Coq's inversion facilities (see [CT95]) prove specific inversion lemmas for specific hypotheses. Where our generic lemma has a case for each inference rule, Coq will simplify and possibly eliminate these cases from a given lemma by applying constructor conflict and injectivity results. However, by performing substitution steps within the equational problem, we capitalise on the power of unification where Coq does not.

Consider the datatype of unlabelled binary trees:

$$\frac{}{\texttt{leaf} : \texttt{tree}} \qquad \frac{s, t : \texttt{tree}}{\texttt{node}\, s\, t : \texttt{tree}}$$

The reflexive relation, `refl`, on this datatype can be represented as an inductively defined relation with inference rule:

$$\frac{}{\texttt{refl}\, t\, t}$$

The corresponding inversion principle encapsulates the idea that the two arguments of any inhabitant of this relation must unify. Our approach addresses this question where mere appeals to constructor conflict and injectivity do not. Similarly, repeated instances of the same schematic variable must correspond to terms which unify.

The proof of the following goal shows how our facility addresses both of these issues:

$$? : \forall x.\, \texttt{refl}\, (\texttt{node}\, x\, \texttt{leaf})\, (\texttt{node}\, (\texttt{node}\, \texttt{leaf}\, \texttt{leaf})\, x) \longrightarrow \bot$$

Inverting the `refl` premise gives us

$$? : \forall t.\, (\texttt{node}\, x\, \texttt{leaf}) = t \longrightarrow (\texttt{node}\, (\texttt{node}\, \texttt{leaf}\, \texttt{leaf})\, x) = t \longrightarrow \bot$$

`Qnify` applies **elimination**

$$(\texttt{node}\, (\texttt{node}\, \texttt{leaf}\, \texttt{leaf})\, x) = (\texttt{node}\, x\, \texttt{leaf}) \longrightarrow \bot$$

then **injectivity**,

$$(\texttt{node}\, \texttt{leaf}\, \texttt{leaf}) = x \longrightarrow x = \texttt{leaf} \longrightarrow \bot$$

then **elimination**,

$$(\texttt{node}\, \texttt{leaf}\, \texttt{leaf}) = \texttt{leaf} \longrightarrow \bot$$

and finally proves the goal by **conflict**.

At present, the '`Inversion`' tactic in Coq does not attempt to unify the two instantiations of x, and hence stops short of proving this goal. The extra steps could easily be done by hand or by a user-supplied tactic. It should not be difficult to adapt Coq's inversion package to the same range of problems addressed by `Qnify`.

5.3 Proving an Operational Semantics Deterministic

In this example, we sketch the proof that the operational semantics for a simple 'while' language is deterministic. This example is partly motivated by the similar example given by Camilleri and Melham for their inductive relation package in the HOL system [HOL, CM92].

A command C executed in state s yields state t (denoted $s < C > t$) according to the inductive definition given in figure 3

$$\frac{}{s < \texttt{skip} > s}$$

$$\frac{}{s < V := E > ([V := E]s)}$$

$$\frac{s < C > t \qquad t < C' > u}{s < C; C' > u}$$

$$\frac{s < C' > t}{s < \texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ C' > t}\ B\ s = \texttt{false}$$

$$\frac{s < C > t}{s < \texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ C' > t}\ B\ s = \texttt{true}$$

$$\frac{}{s < \texttt{while}\ B\ \texttt{do}\ C > t}\ B\ s = \texttt{false}$$

$$\frac{s < C > t \qquad t < \texttt{while}\ B\ \texttt{do}\ C > u}{s < \texttt{while}\ B\ \texttt{do}\ C > u}\ B\ s = \texttt{true}$$

Fig. 3. Evaluation Relation for a Simple Imperative Language

Our goal is to show this semantics deterministic, ie:

$? : \forall C. \forall s, t. \forall H : s < C > t. \forall u. \forall H' : s < C > u. t = u$

The proof proceeds by induction on H and inversion of H'. We may use the new features of LEGO to express this plan as a composite tactic, also adding hints to dispose of the trivial cases:

```
Lego> Induction H Then intros Then Invert H' Then Qnify
        Then (Refine Eq_refl Else Immed);
```

`Qnify` eliminates the bulk of the uninteresting cases by **conflict**—those where H and H' take different values of C. Reflexivity of equality is enough for both `skip` and `:=`, whilst the `Immed` catches the trivial inductions—again, the pragmatic positioning of equalities in the inversion lemma and the substitution of older variables for newer ones reduces a number of cases directly to their inductive hypotheses.

The only cases which require user intervention are:

– sequential composition and `while` with $B\ s =$ `true`

  These both have two inductive hypotheses, corresponding to two phases of execution. Using the first inductive hypothesis, the intermediate states resulting from the two executions of C are shown to be the same. Having made this substitution explicitly, the second inductive hypothesis then completes the result.

– `if` and `while` with $B\ s =$ `true` $=$ `false`

  $B s$ cannot be `true` in H and `false` in H' or vice versa. However, $B s$ is not in constructor form, so the user must assist in these four cases. A substitution gives `true` $=$ `false`, and `Qnify` finishes the **conflict** proof.

Camilleri and Melham give the full HOL90 source for their treatment of this operational semantics at

`http://www.dcs.glasgow.ac.uk/ tfm/ftp2.html`

Their package, as detailed in [CM92], enables the definition of the evaluation and the automatic derivation of theorems corresponding to the induction and inversion principles generated by LEGO. Other tactics automatically prove conflict and injectivity properties for the constructors of the language syntax.

However, they provide nothing akin to `Qnify` which simplifies inversion subgoals by attacking the unification problems they contain. Progress from the inversion is largely ad hoc, although HOL users have at their disposal a large library of rewriting tactics and the full power of ML for programming with them.

The proof of determinacy is built from the bottom up. Firstly, the inversion principle '`ecases`', the conflict theorems '`distinct`' and the injectivity theorems '`const11`' are combined into a tactic similar to the unification-free inversion tactic in Coq.

```
val SIMPLIFY = REWRITE_RULE (distinct :: const11);

val CASE_TAC = DISCH_THEN
               (STRIP_ASSUME_TAC o
                SIMPLIFY o
                ONCE_REWRITE_RULE[ecases]);
```

Next, `CASE_TAC` is used to build a specific inversion lemma for each inference rule of the evaluation relation. Here is the derivation for `skip`:

```
val SKIP_THM = store_thm("SKIP_THM",
     (--`!s1 s2. EVAL skip s1 s2 = (s1 = s2)`--),
     REPEAT GEN_TAC THEN EQ_TAC THENL
     [CASE_TAC THEN ASM_REWRITE_TAC [],
      DISCH_THEN SUBST1_TAC THEN MAP_FIRST RULE_TAC rules]);
```

Finally, these lemmas are used in the induction itself:

```
val DETERMINISTIC = store_thm ("DETERMINISTIC",
(--`!C st1 st2. EVAL C st1 st2 ==>
          !st3. EVAL C st1 st3 ==> (st2 = st3)`--),
     RULE_INDUCT_TAC THEN REPEAT GEN_TAC THENL
     [REWRITE_TAC [SKIP_THM],
      REWRITE_TAC [ASSIGN_THM],
      PURE_ONCE_REWRITE_TAC [SEQ_THM] THEN STRIP_TAC THEN
        FIRST_ASSUM MATCH_MP_TAC THEN
        RES_TAC THEN ASM_REWRITE_TAC [],
      IMP_RES_TAC IF_T_THM THEN ASM_REWRITE_TAC [],
      IMP_RES_TAC IF_F_THM THEN ASM_REWRITE_TAC [],
      IMP_RES_TAC WHILE_F_THM THEN ASM_REWRITE_TAC [],
      IMP_RES_THEN
        (fn th =>  PURE_ONCE_REWRITE_TAC [th]) WHILE_T_THM THEN
        STRIP_TAC THEN FIRST_ASSUM MATCH_MP_TAC THEN
        RES_TAC THEN ASM_REWRITE_TAC []]);
```

Although it is perhaps a little unfair to compare the raw ML of HOL with the cleaner interfaces of LEGO or Coq, it seems clear from this example that the unification and rewriting preformed by the single LEGO tactic `Qnify` captures a wide variety of cases which must be addressed individually and by hand in HOL.

6 Conclusions and Further Work

This work owes a considerable debt to that of Cornes and Terrasse in Coq ([CT95]). Our formulation and proof of generic inversion principles follow theirs, as do our constructor conflict and injectivity results.

However, the insight

'inversion = predecessor premises + equational constraints'

has led us to a treatment which is intuitive, easy to implement and highly effective in practice. We analyse the equational information separately via the unification algorithm described in this paper. Its specification is clear, it is sound and complete with respect to constructor forms and the tactic `Qnify`, to which it gives rise, has proved independently useful.

For example, recent work extends the inversion facility to cover full induction on the derivation of a relation. A goal of form

$$? : \forall \vec{x}.\, R\, \vec{t}[\vec{x}] \longrightarrow \Phi$$

is first rewritten

$$? : \forall \vec{y}.\, R\, \vec{y} \longrightarrow \forall \vec{x}.\, \vec{y} = \vec{t}[\vec{x}] \longrightarrow \Phi$$

so that the induction principle generated for R is applicable. The equations introduced are ripe for simplification with `Qnify`.

This new `Induction` tactic deals with the problem of type dependency within parameters yielding ill-typed equations by packaging the related parameters in tuples which share the same Σ-type. A similar repair could be made to both inversion and injectivity theorems, extending the class of definition for which they can be generated.

It would seem both worthwhile and tractable to extend all of these new facilities to encompass mutual inductive definitions. This work may be carried out in the near future.

At present, work is in progress implementing **checking** proofs. The compilation of the necessary functions does appear both systematic and tractable, without recourse to fixpoints.

All the new facilities described in this paper, together with documentation describing their usage is available on the web at

`http://www.dcs.ed.ac.uk/home/lego/html/alpha/`

Acknowledgements The author would like to thank Cristina Cornes for providing the model for this work and much useful advice. Much gratitude is due also to Rod Burstall, James McKinna and Alan Smaill for their patience, guidance and support.

References

[Bur96] R. M. Burstall. Inductively Defined Relations: A Brief Tutorial. Extended Abstract. In Haveraan, M., and Owe, O., and Dahl, O.-J., editors, Recent Trends in Data Types Specification. Springer LNCS 1130, pp14–17. 1996.

[CM92] J. Camilleri and T. Melham. Reasoning with Inductively Defined Relations in the HOL Theorem Prover. Technical Report No. 265 University of Cambridge Computer Laboratory. 1992.

[Cla78] K. Clark. Negation as Failure. pp293-322 of Logic and Data Bases, edited by H. Gallaire and J. Minker. Plenum Press. 1978.

[Coq] C. Cornes, J. Courant, J.F. Fillaître, G. Huet, C. Murthy, C. Parent, C. Paulin, B. Werner. The Coq Proof Assistant Reference Manual, Version 5.10. Projet Coq, Inria-Rocquencourt and CNRS-ENS Lyon, France.

[Cor96] C. Cornes Compilation du Filtrage avec Types Dépendants dans le Système Coq. Actes de la réunion du pôle Spécification et Preuves du GDR Programmation. Orleans, Novembre 1996.

[CT95] C. Cornes, D. Terrasse. Automating Inversion of Inductive Predicates in Coq. In BRA Workshop on Types for Proofs and Programs, Turin, June 1995. To appear in LNCS series.

[Eri91] L.-H. Eriksson. A finitary version of the calculus of partial inductive definitions. In: L.-H. Eriksson, L. Hallnäs & P. Schroeder-Heister (editors), Extensions of Logic Programming. Second International Workshop, ELP-91, Stockholm. Springer LNCS 596, pp89–134. 1992.

[Dyb91] P. Dybjer. Inductive Sets and Families in Martin-Löf's Type Theory. pp280–306 of Logical Frameworks, edited by G. Huet and G. Plotkin. CUP 1991.

[Gim94] E. Giminez. Codifying guarded definitions with recursive schemes. Proceedings of Types 94, pp39–59.

[Hal91] L. Hallnäs. Partial Inductive Definitions. Theoretical Computer Science. Vol. 87. pp115–142. 1991.

[HOL] Introduction to HOL; A theorem proving environment for higher order logic. Edited by M.J.C. Gordon and T.F. Melham. CUP 1993.

[JK91] Jean-Pierre Jouannaud and Claude Kirchner. Solving Equations in Abstract Algebras: A Rule-Based Survey of Unification. pp257–321 of Computational Logic: Essays in Honor of Alan Robinson, edited by Jean-Louis Lassez and Gordon Plotkin, MIT Press, 1991.

[Luo94] Zhaohui Luo. Computation and Reasoning: A Type Theory for Computer Science. OUP 1994.

[LEGO] Zhaohui Luo, Randy Pollack. LEGO Proof Development System: User Manual. Technical Note, 1992.

[Mag95] Lena Magnusson. The Implementation of ALF. PhD Thesis. Chalmers University of Technology and University of Göteborg, Sweden. January 1995.

[Pau87] L. Paulson. Logic and Computation: Interactive Proof with Cambridge LCF. Cambridge Tracts in Theoretical Computer Science 2. CUP 1987.

[Pol94] Randy Pollack. Incremental Changes in LEGO: Technical Note, 1994.

[Pra65] Prawitz, D. Natural Deduction: A Proof-Theoretical Study. Almqvist & Wiksell. Stockholm, 1965.

[TS84] H. Tamaki, T. Sato. Unfold/Fold Transformation of Logic Programs. Proceedings of Second International Logic Programming Conference. pp127–138. Uppsala, 1984.

This article was processed using the LaTeX macro package with LLNCS style