Integrating an Equality Prover into a Software Development System based on Type Theory *

Martin Strecker, Maria Sorea

Universität Ulm
D-89069 Ulm, Germany

Abstract. This paper reports on the integration of an untyped equational prover into a proof system based on an expressive constructive type theory. The proofs returned by the equational prover are effectively verified for type correctness, a proof term can be constructed. The scheme of proof translation described here is illustrated by the integration of the DISCOUNT prover into the software development system TYPELAB.

1 Introduction

In recent years, most effort in automated reasoning research has been expended on improving the performance of systems that support deduction in classical first order logic or fragments thereof. The advances that have been achieved in this area permit to successfully apply automated theorem proving in some branches of mathematics or for problems such as diagnosis.

Because the (practical) expressiveness of first-order logic is quite restricted, most systems that are routinely used in software development or for large-scale hardware verification tasks [OSR93, Pau94, Bar96] are based on a higher-order logic with a more or less elaborate type system. The automation provided by these systems has to a great extent been directly coded into specialized tactics and mostly relies on other techniques than those employed by dedicated proof systems, such as congruence closure for equality reasoning with ground terms in PVS.

The work presented here is part of a larger research programme aiming at making traditional first-order theorem proving techniques available for fragments of more expressive logics. This paper discusses an integration of an untyped first-order equality prover into a software development system based on constructive type theory with dependent types (see Section 2). By the "propositions as types" paradigm, a proposition in type theory can be interpreted as a type whose elements are the proofs of the proposition. A proof problem can then be understood as the problem of constructing a term prf such that the typing judgement $\Gamma \vdash prf : P$ holds, where P is the type corresponding to the proposition to be proved, and Γ is a context containing the hypotheses. The approach taken for integrating an equality prover into a proof assistant for type theory is roughly the following: Given a set of hypotheses Γ and an equality P to be proved, transfer the relevant information to the equality prover (see Section 4), invoke it on the proof problem and read in the trace of the

* This research has partly been supported by the "Deutsche Forschungsgemeinschaft" within the "Schwerpunktprogramm Deduktion"

equality proof. The trace should permit to construct the proof object prf. The details of the steps carried out to obtain the proof object are spelled out in Section 5. The procedure described here has been implemented in the software development system TYPELAB [vHLS97], using DISCOUNT [DKS97] as equality prover.
There are several examples of cooperation between proof systems, most of which, however, are instances of a "black box" paradigm where results returned from the cooperating prover are not checked for validity: KIV [HRS88] uses 3TAP [BHG+96] for automating proof search in predicate logic; ILF [DGH+95] provides an interface to several proof engines (among them Discount , KoMeT, Otter, Setheo and Spass), which are called depending on the problem at hand. PVS [OSR93] invokes an external BDD package for Boolean simplification and a model checker for CTL formulas.
There are only few examples of a cooperation in which results are effectively verified: Harrison [HT93] describes an interface between the HOL theorem prover and a computer algebra system, in which the result returned by the computer algebra system, such as a factorization of a polynomial, is checked for validity. Recently, there are efforts to integrate a proof search procedure for intuitionistic predicate logic [KOS96], based on an extension of the connection method, into the NuPRL system [Con86].
A careful analysis of the proof returned by the equality prover is necessary to ensure that the proof carried out in an untyped logic is valid in a typed logic. Section 6 will provide arguments why a proof returned by the equality prover is typecorrect in most interesting cases. The typing rules presented in Section 5 describe how proof objects can be constructed that permit to verify the validity of every instance of an equality proof. Since proof objects tend to become very large, their construction may not always be desirable. The typing rules can then be understood as safeguards that have to be built in to ensure the correctness of proofs.
Due to lack of space, this paper has to remain cursory. A more detailed description, in particular of implementation aspects, can be found in [Sor96]. An extended version of this paper, containing proofs of some of the statements made below, will be made available on the Web page of the TYPELAB project:
`http://www.informatik.uni-ulm.de/ki/Forschung/Deduktion/index-e.html`

2 Languages

In this section, the language of type theory and the language of the equality prover will be contrasted. Since the language of type theory contains expressions which cannot sensibly be mapped to a first-order equality prover, a sublanguage will be identified which is suitable for stating equality proof problems.

2.1 Typed Language

The type theory this work is based on is the *Extended Calculus of Constructions (ECC)* [Luo90], an expressive logic whose language comprises:

- Term constructors such as:
  - Lambda-abstraction $\lambda\, x : T.\, M$ and function application $(M\ N)$
  - Tuple formation $\langle M, N \rangle$ and projection $P.1$, $P.2$
- Type constructors such as:

- Type constants *Prop* (the collection of propositions) and *Type* (the collection of types)
- Dependent function type $\forall x : A.B$. The non-dependent function type $A \rightarrow B$ is an abbreviation for the dependent function type $\forall x : A.B$, where x does not occur in B.
- Dependent record types $\Sigma x : M.\ N$

Details of the structure of the terms of *ECC* are not relevant for the purposes of this paper. Further below we will only make precise what the fragment of the language looks like that is used for stating equality problems.
A typing judgement $\Gamma \vdash t : T$ expresses that term t is of type T in context Γ. A context is a finite sequence of variable declarations of the form $x : A$, where x is a variable and A a type. By the "propositions as types" paradigm, propositions are interpreted as types. Informally, a variable declaration such as $h : t_1 =_T t_2$ can then be understood as a hypothesis labelled with h.
Typing judgements are defined inductively on the structure of terms. Again, the whole set of typing rules is not relevant in our case. We just cite the rules for Lambda abstraction and application:

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda\, x : A.M : \forall\, x : A.B}\ (\lambda)$$

$$\frac{\Gamma \vdash M : \forall\, x : A.B \quad \Gamma \vdash N : A}{\Gamma \vdash (M\ \ N) : B[x := N]}\ (\text{app})$$

Definition 1 Bound and free variables. Suppose t is a term which is well-typed in a context Γ. An occurrence of a variable x will be called *bound* in t if x occurs in the scope of an abstraction in t. It will be called *free* otherwise (then x occurs in Γ).

The notion of a variable being bound or free is relative: In the premiss of the (λ) rule, x occurs free in M (if it occurs in M), but bound in $\lambda\, x : A.M$.
The equality $t_1 =_T t_2$ we will be using in the sequel (t_1 equals t_2, where both terms have type T) is not predefined, but can be encoded by means of Leibniz' principle which says that two terms are equal if they agree in all properties expressible in the language:

$$x =_T y \ := \ \forall\, P : T \rightarrow Prop.(P\ x) \leftrightarrow (P\ y)$$

Leibniz equality is weaker than syntactic equality or convertibility in the sense that convertible terms can be shown to be Leibniz equal, but not vice versa. As a consequence, two terms which are Leibniz equal cannot be arbitrarily substituted for one another without endangering type correctness. For example, if type T contains a term n_1 of type Nat, $n_1 =_{Nat} n_2$ and $\lambda\, x : T[n_1].(f\ \ x)$ is typecorrect, then $\lambda\, x : T[n_2].(f\ \ x)$ is usually not typecorrect.
In order to exclude problematic cases like this one already on the language level, we will in the following identify a sublanguage $\mathcal{L}_T$ of *ECC* in which equality problems will be stated. We will consider *generalized equations* $\forall\, x_1 : T_1, \ldots x_n : T_n.l =_T r$, which are built up from the type constants, a set of variables $\mathcal{V}_T$, and from function

application. The grammar for generalized equations is given by production $\mathcal{E}_T$, defined as follows:

$$\mathcal{T}_T ::= Prop \mid Type \mid \mathcal{V}_T \mid (\mathcal{T}_T\ \mathcal{T}_T)$$

$$\mathcal{E}_T ::= (\mathcal{T}_T = \mathcal{T}_T) \mid \forall\, \mathcal{V}_T : \mathcal{GT}_T .\, \mathcal{E}_T$$

Here, $\mathcal{GT}_T$ denotes the set of general terms of the logic ECC. Note that function application is in curried form, i.e. an application $f(a_1, a_2)$ is here written as $((f\ a_1)\ a_2)$, but will often be abbreviated to $(f\ a_1\ a_2)$.
The following semantic constraints are imposed on the equations that we will consider:

Definition 2 Valid Generalized Equation. A generalized equation E is *valid* in a context Γ if it fulfills the following requirements:

- E is well-typed in context Γ, that is, $\Gamma \vdash E : Prop$ is derivable.
- Each element of $\mathcal{V}_T$ has associated with it a fixed arity, which is the number of arguments which are applied to it.
- Every x_i occurs in l or in r.
- In an application $(f\ a)$, f is not a bound variable.

Remarks:
The arity of a variable is related to, but not completely determined by its type. For example, a function symbol $f : A \rightarrow B$ might have arity 0 (if only used as a functional constant) or 1 (in applications like $(f\ a)$). The restriction stated in the definition ensures that variable symbols are used in a uniform manner.
In a typed logic, it is not generally valid to conclude from $\forall\, x : A, z : C.(f\ x) = b_1$ and $(f\ a) = b_2$ that $b_1 = b_2$, because no element of type C is given which would permit a correct instantiation of the universal quantifier. The third condition of the definition is thought to prevent such conclusions.
The fourth condition of the definition prohibits generalized equations of the form $\forall f : A \rightarrow B.(f\ a) = b$, which would lead to problems during a translation to a first-order prover.
Note that higher-order abstractions are not disallowed altogether:

$$\begin{array}{l}\forall f : A \rightarrow B, a : A, l : (List\ A).\\ \qquad (map\ f\ (cons\ a\ l)) =_{List\ B} (cons\ (apply\ f\ a)\ (map\ f\ l))\end{array}$$

is a valid equation, however not the (more natural)

$$\begin{array}{l}\forall f : A \rightarrow B, a : A, l : (List\ A).\\ \qquad (map\ f\ (cons\ a\ l)) =_{List\ B} (cons\ (f\ a)\ (map\ f\ l))\end{array}$$

because the bound function symbol f is then used with different arities and in addition occurs in a functional position. In this example, map is assumed to be of type $(A \rightarrow B) \rightarrow A \rightarrow (List\ A) \rightarrow (List\ B)$, where A and B are fixed types. In the fragment of the logic considered here, we could also use a version of the map function with parametric polymorphism. map would then have type $\forall\, X, Y : Type.\ (X \rightarrow Y) \rightarrow \ldots$, and applications of map would be written as: $(map\ A\ B\ f\ l)$.

Definition 3 Valid Proof Problem. A valid proof problem (with respect to a context Γ) consists of a set of *hypotheses* $\{E_1, \ldots E_n\}$, which are generalized equations valid in Γ, and a quantifier-free *goal* equation E_g which is valid in Γ.

The requirement that E_g is a quantifier-free equation is not a genuine restriction, since any universally quantified goal equation can be converted into a quantifier-free one by moving universally quantified variables into the context, thus converting them to constants.

2.2 Language of the Equality Prover

We assume a usual untyped first-order language $\mathcal{L}_E$, whose terms $\mathcal{T}_E$ and equations $\mathcal{E}_E$ are built up from a set of functions and constants $\mathcal{FC}_E$ and a set of variables $\mathcal{V}_E$, according to the following grammar:

$$\mathcal{T}_E ::= \mathcal{V}_E \mid \mathcal{FC}_E(\mathcal{T}_E \ldots \mathcal{T}_E) \qquad\qquad \mathcal{E}_E ::= (\mathcal{T}_E = \mathcal{T}_E)$$

Again, each symbol of $\mathcal{FC}_E$ has associated with it an arity which determines the number of arguments it takes. Functions are here written in the usual un-Curried form.

3 An Example Proof

Before giving a formal presentation of the translation of proof problems and the resulting proofs, the interaction of TYPELAB and DISCOUNT will be illustrated by a standard example from group theory, the problem of showing that $x + (-x) = 0$. The problem to be solved in TYPELAB is displayed in Figure 1.

```
  T:Type
  op:T->T->T
  unit:T
  inv:T->T
  unit_ax_r:all(x:T) op x unit = x
  unit_ax_l:all(x:T) op unit x = x
  left_inv_ax:all(x:T) op (inv x) x = unit
  assoc_ax:all(x,y,z:T) op (op x y) z = op x (op y z)
  x:T
|--------------------------------------
  ?2:op x (inv x) = unit
```

Fig. 1. TYPELAB proof problem

The declarations of free variables and the hypotheses are listed above the stylized turnstile, the goal formula below. ?2 is a *metavariable* representing the proof term to be constructed. This proof problem is written on file in a format appropriate for DISCOUNT (see Section 4). Upon successful completion of the proof, a trace is returned by DISCOUNT (see Figure 2), which is then interpreted by TYPELAB and ultimately yields the proof term shown in Figure 3. Its structure closely reflects the structure of the proof translation, which will be further discussed in Section 5. In order to avoid name clashes, the variable `x` occuring free in the TYPELAB goal has been renamed to `x4` in DISCOUNT.

```
Consider the following set of axioms:
  Axiom 1: op(x,unit()) = x
  Axiom 2: op(unit(),x) = x
  Axiom 3: op(inv(x),x) = unit()
  Axiom 4: op(op(x,y),z) = op(x,op(y,z))

Proof:
  Lemma 1: y = op(inv(inv(y)),unit())
    y = op(unit(),y) by Axiom 2 RL at e with {x <- y}
      = ...
      = op(inv(inv(y)),unit()) by Axiom 3 LR at 2 with {x <- y}

  Theorem 1: op(x4(),inv(x4())) = unit()
    op(x4(),inv(x4())) = op(op(inv(inv(x4())),unit()),inv(x4()))
                         by Lemma 1 LR at 1 with {y <- x4()}
                       = ...
                       = unit() by Axiom 3 LR ..
```

Fig. 2. DISCOUNT proof (shortened)

```
let (lemma_1:=
 fun(y:T) equal_transitive T y (op unit y) (op (inv (inv y)) unit)
                (equal_symmetric T (op unit y) y (unit_ax_l y))
                (equal_transitive T (op unit y) ...))
in
equal_transitive T (op x (inv x)) (op ... (inv x)) unit
 (lemma_1 x (fun(v:T) op x (inv x) = op v (inv x)) (..))
 (equal_transitive T (op (op (inv (inv x)) unit) (inv x)) ...
   (unit_ax_r (inv (inv x)) ..))
```

Fig. 3. Proof Term (shortened)

4 Translations to the Equality Prover

The translation of *terms and equations* from the typed language $\mathcal{L}_T$ to the untyped language $\mathcal{L}_E$ of the equality prover is carried out as follows:

- The quantifier prefix is discarded.

- Variables of $\mathcal{L}_T$ which are free in the context under consideration are mapped to function symbols or constants, that is, elements of $\mathcal{FC}_E$.
- Variables of $\mathcal{L}_T$ which are bound in the context under consideration are mapped to variables, that is, elements of $\mathcal{V}_E$.
- Function applications are written in un-curried form.

Since only valid equations in the sense of Definition 2 are translated, it can be shown that only correct terms of $\mathcal{L}_E$ are produced by the translation.
As to the translation of an entire *proof problem*, we assume that in the equality prover, proof problems are stated as a list of axioms together with the goal formula to be proved, possibly with some additional information. A proof problem $\Gamma, h_1 : E_1, \dots h_n : E_n \vdash ?n : E_g$ can then be transferred to the equality prover simply by translating every equation $E_1, \dots E_n, E_g$.

5 Translations from the Equality Prover

5.1 Translating Terms

The translation from the untyped language $\mathcal{L}_E$ to the typed language $\mathcal{L}_T$ requires the reconstruction of type information. We assume that untyped terms are translated by a function *trTerm* and untyped equations by *trEquation*. These functions roughly work as follows:

- Since no new constants are generated during an equality proof, elements of $\mathcal{FC}_E$ are contained in the current context, their type can therefore be looked up.
- The type of variables (elements of $\mathcal{V}_E$) is not generally known and therefore has to be inferred. For example, the type of the variable y of the proof in Figure 2 is inferred to be T. Variables can only occur in argument position, so the method for infering argument types, described below, applies.
- An application $f(a_1, \dots a_n)$ is first represented in Curried form. We thus only have to deal with the problem of translating terms $(f\ a)$. Noting that the head term of f can only be a function constant, it can be shown by an inductive argument that the type of f can be completely determined. If f is of type $\forall\, x : A.B$, the argument a has to be evaluated. If a is a variable which has not been assigned a type before, its type is set to be A. Otherwise, evaluation of a returns a type A'. If A and A' are the same, $(f\ a)$ can be typed to $B[x := a]$, otherwise the term is not correctly typed.
- The function *trEquation* evaluates the left and right sides of an equation, compares their types and universally abstracts over all the variables occuring free in the equation.

Type reconstruction works for all terms except for "pure" variables not occurring embedded in terms. In an equation $l = x$ or $x = r$, the type of the pure variable x can be determined, respectively, by the type of l or r, unless both sides of an equation are variables. We have here a situation in which a valid equality proof cannot be translated back because type inference fails. Note that this situation can only arise in the case of an equation $x = x$, which is trivial, or an equation $x = y$, which indicates that the equational theory is inconsistent.

5.2 Translating Proofs

The translation of proofs serves the double purpose of checking that the proof found by the equational prover is typecorrect and constructing a proof term. In order to obtain an independent certification that the proof is correct, the proof term can be evaluated and its type can be compared to the theorem it is supposed to prove. Construction of a proof term is not an essential part of the procedure presented below. Since proof terms tend to become quite large, they will usually not be constructed in a realistic scenario.

We make the following assumptions about the representation of equality proofs, which are reasonable for completion-based equality provers:

- An equational proof is a sequence of *lemmas*, where each lemma consists of a statement of the lemma of the form $l = r$ and a subsequent proof of this lemma. The sequence of lemmas may just have a length of one, in which case the main theorem can be proved directly. Otherwise, some intermediate lemmas are used to prove the main theorem. They usually result from critical pairs which are generated during completion.
- The proof of each lemma $l = r$ is in turn composed of a sequence of equations $l \equiv t_0 = t_1 \dots t_n \equiv r$. Each such equation $t_i = t_{i+1}$, together with its justification, is called a *subproof*.
- A *justification* provides evidence for the correctness of an equation $t_i = t_{i+1}$. The justification has to state which axiom or preceding lemma has been applied, at which position of term t_i it is applied, and in which direction and with which instantiations it is applied.

For the constructs cited above, translation functions $tr\dots$ will be given. The behaviour of these translation functions will be specified by inference rules

$$\frac{\mathit{trfun}_1(\dots) \Rightarrow \Gamma_1 \vdash t_1 : T_1 \quad \cdots \quad \mathit{trfun}_n(\dots) \Rightarrow \Gamma_n \vdash t_n : T_n}{\mathit{trfun}_0(\dots) \Rightarrow \Gamma_0 \vdash t_0 : T_0}$$

which have to be read as: If after evaluation of $\mathit{trfun}_i(\dots)$ the judgement $\Gamma_i \vdash t_i : T_i$ holds ($i = 1 \dots n$), it can be concluded that after evaluation of $\mathit{trfun}_0(\dots)$, the judgement $\Gamma_0 \vdash t_0 : T_0$ holds. The correctness of these inference rules can be verified by checking that the corresponding typing rules

$$\frac{\Gamma_1 \vdash t_1 : T_1 \quad \cdots \quad \Gamma_n \vdash t_n : T_n}{\Gamma_0 \vdash t_0 : T_0}$$

are derivable.

Notational convention: Untyped terms resp. equations are primed (such as $l' = r'$), the corresponding typed terms resp. equations have not primes (such as $l =_T r$). For lack of space, we only present the most interesting rules. The reader is invited to compare the translation as suggested by the rules below with the example of Figures 2 and 3.

Translating a proof: A proof is a sequence of lemmas.

$$\frac{\begin{array}{c}\mathit{trLemma}(\Gamma;\ \mathit{AxiomMap};\ \mathit{lemma}) \Rightarrow \Gamma \vdash \mathit{lmp} : \mathit{LM} \\ \mathit{trProof}(\Gamma, d := \mathit{lmp} : \mathit{LM};\ \mathit{NewAxiomMap};\ \mathit{lemlist}) \Rightarrow \Gamma, d := \mathit{lmp} : \mathit{LM} \vdash \mathit{thp} : \mathit{TH}\end{array}}{\mathit{trProof}(\Gamma;\ \mathit{AxiomMap};\ \mathit{lemma} :: \mathit{lemlist}) \Rightarrow \Gamma \vdash \mathrm{let}\ d := \mathit{lmp} : \mathit{LM}\ \mathrm{in}\ \mathit{thp}\ :\ \mathit{TH}}$$

With $d := lmp : LM$, the name d can be introduced in a context as a definition of term lmp of type LM. Similarly, d can be bound locally within a term by a let-abstraction. Note that d can occur in thp, but not in TH, the theorem to be proved. $AxiomMap$ is a function mapping equations $l' = r'$ to a pair

$$\langle p, \forall\, x_1 : T_1, \ldots x_n : T_n . l =_T r \rangle$$

consisting of the generalized equation corresponding to $l' = r'$ and its proof object p. $NewAxiomMap$ is the obvious extension of $AxiomMap$ by the lemma just proved. For example, referring back to Figure 3, the abstraction `let lemma_1:= fun(y:T) equal_transitive ... in ...` is the result of translating the entire equality proof, by first translating Lemma 1 and binding its result to the identifier `lemma_1` before translating the main theorem.

Translating a lemma: A lemma consists of the statement of an equality and a subproof of the validity of this equality.

$$\frac{\begin{array}{c} trEquation(\Gamma;\ l' = r') \Rightarrow \Gamma \vdash \forall\, x_1 : T_1, \ldots x_n : T_n . l =_T r\ :\ Prop \\ trSubProof(\Gamma, x_1 : T_1, \ldots x_n : T_n;\ AxiomMap;\ subprf) \Rightarrow \\ \Gamma, x_1 : T_1, \ldots x_n : T_n \vdash p : l =_T r \end{array}}{\begin{array}{c} trLemma(\Gamma;\ AxiomMap;\ \langle l' = r', subprf \rangle) \Rightarrow \\ \Gamma \vdash \lambda\, x_1 : T_1, \ldots x_n : T_n . p\ :\ \forall\, x_1 : T_1, \ldots x_n : T_n . l =_T r \end{array}}$$

Note that the subproof is evaluated in a context in which the bound variables of the equation corresponding to the lemma have been moved into the context. Thus, the terms occuring in the subproof are actually ground. For the proof of correctness of this rule, compare with typing rule (λ) of Section 2.1.

Translating a subproof: A subproof of an equation $l =_T r$ consists of a sequence of steps $l \equiv t_0 =_T t_1 \ldots =_T t_n \equiv r$, where each step $t_i =_T t_{i+1}$ is provided with a justification $just_i$ explicating how the equation $t_i =_T t_{i+1}$ can be derived. Essentially, the transitivity of equality is used to concatenate the proof steps.

$$\frac{\begin{array}{c} trJust(\Gamma;\ AxiomMap;\ t_0' = t_1';\ just) \Rightarrow \Gamma \vdash p_0 : t_0 =_T t_1 \\ trSubProof(\Gamma;\ AxiomMap;\ splist) \Rightarrow \Gamma \vdash p_1 : t_1 =_T t_2 \end{array}}{\begin{array}{c} trSubProof(\Gamma;\ AxiomMap;\ \langle t_0' = t_1', just \rangle :: splist) \Rightarrow \\ \Gamma \vdash (equal\_transitive\ T\ t_0\ t_1\ t_2\ p_0\ p_1)\ :\ t_0 =_T t_2 \end{array}}$$

For example, when translating Lemma 1 in Figure 3, the equality `y = op(unit(),y)` is translated first, then (by a repeated application of the transitivity rule) the equality `op(unit(),y) = op(inv(inv(y)),unit())`. Finally, both proof steps are joined to yield the term `equal_transitive T y (op unit y) (op (inv (inv y)) unit) ... ....`

Translating a justification: A justification is translated by first instantiating the referenced axiom or lemma as given by the instantiation list $instlist$, then orienting the resulting equation in the direction specified by dir and finally carrying out the replacement at the term position indicated by $poslist$.

$$\frac{\begin{array}{c} trInst(\Gamma;\ AxiomMap;\ l' = r';\ instlist) \Rightarrow \Gamma \vdash p_i : l_i =_{T_i} r_i \\ trDir(\Gamma;\ dir;\ p_i;\ l_i =_{T_i} r_i) \Rightarrow \Gamma \vdash p_d : l_d =_{T_d} r_d \\ trCong(\Gamma;\ l' = r';\ poslist;\ p_d;\ l_d =_{T_d} r_d) \Rightarrow \Gamma_c : l_c =_{T_c} r_c \end{array}}{trJust(\Gamma;\ AxiomMap;\ l' = r';\ \langle poslist, dir, \langle axiom, instlist \rangle \rangle) \Rightarrow \Gamma \vdash p_c : l_c =_{T_c} r_c}$$

6 Type preservation

One might object against using an untyped logic for carrying out proofs in a typed logic that the untyped proofs will too often include inferences which are not typecorrect, thus making the cooperation between a typed and an untyped prover practically useless. This section provides some evidence which invalidates this objection and shows that, under some reasonable assumptions about the inference mechanisms used by the equality prover, types are preserved in a rewrite proof. Due to lack of space, the arguments have to remain on an informal level.

We assume that the equality prover is initialized with a proof problem which is the result of translating a valid typed problem (in the sense of Def. 3). Thus, the terms and equations the equality prover starts with are typeable, and the question that has to be examined is whether inferences of the equality prover can produce terms or equations that are not typeable. We further assume that the basic inference mechanisms of an equality prover are rewriting a term (resp. an equation) and completion, i.e. the generation of lemmas from critical pairs. Under this assumption, the above question can be restated:

- Does rewriting a typeable term t (a typeable equation $t_l = t_r$) with a typecorrect rule $l \rightarrow r$ always produce a term t' (an equation $t'_l = t'_r$) which is typeable?
- Does the completion procedure always introduce rewrite rules as lemmas which are typeable?

If these questions can be affirmed, a simple inductive argument shows that the untyped equality proof retains its validity in a typed framework. Unfortunately, this is not so in general. Consider the problem of proving from the equation $\forall x : A.x =_A a$ that $b_1 =_B b_2$, where b_1, b_2 are constants of type $B \neq A$. If type information is discarded, the proof succeeds by showing that $b_1 = a = b_2$. Of course, the resulting equations cannot be typed any more. In a dependently typed calculus, still more arcane effects can arise due to the interaction between the value of one term and the type of another term.

We will argue that, with "interesting" equations as defined in the following, a kind of proof as in the above example cannot be carried out in a simply-typed framework. This argument can be extended to a polymorphic calculus under the assumption that there are only equations between values and no equations between types.

Definition 4 Interesting equation.
A valid equation $\forall x_1 : T_1, \dots x_n : T_n.l =_T r$ is defined to be *interesting* if l being a variable x implies that r is a term unequal x, but containing x.

All equations of the form $\forall x_1 : T_1, \dots x_n : T_n.f(t_1, \dots, t_k) =_T r$ (with $k \geq 0$) are interesting in this sense, also equations of the form $\forall x_1 : T_1, \dots x_n : T_n.x =_T f(\dots, x, \dots)$. Under usual variable assumptions ($Var(r) \subseteq Var(l)$ or vice versa) only equations of the form $\forall x : A.x =_A a$ are excluded, which express that type A consists of just a single element. Note that any reasonable equality prover based on rewriting applies an equality $x = f(\dots, x, \dots)$ in the direction $f(\dots, x, \dots) \rightarrow x$.

The important feature of interesting equations is that the outermost function symbol provides a cover which ensures that the equation is only applied at a term of the same type. More precisely, assume that the term $t := f(t_1, \dots t_n)$ can be typed with type T_1 and equation $f(l_1, \dots l_n) = r$ can be typed with type T_2. T_1 and T_2 then

have to agree. If t and $f(l_1, \ldots l_n)$ can be matched, that is, there is a substitution σ such that $t \equiv f(l_1, \ldots l_n)\sigma$, then both $f(l_1, \ldots l_n)\sigma$ and $r\sigma$ are typeable with T_1, so that the inference $t = r\sigma$ is typecorrect. A similar argument holds for critical pairs generated during completion.
Summarizing the above argument, it can be shown that an untyped equality prover produces a proof which is typeable under the following assumptions:

- The initial set of equations consists of "interesting" equations, in the above sense.
- During completion, only interesting equations are generated.
- The basic inferences of the equational prover consist of replacing equal terms for equals, subject to usual variable requirements, and of generating lemmas from critical pairs.

7 Conclusions

This paper has described the integration of an untyped equality prover into a system based on a constructive type theory. As opposed to most other examples of prover cooperation, a rigorous verification of the proof returned by the equality prover is carried out. This verification comprises the reconstruction of necessary type information and, if desired, also the construction of a proof term which, through typechecking, permits the direct validation of the proof.
It has been shown that proof verification fails for valid equational proofs only in rare cases. In a similar vein, it has been argued that for equational problems which are "interesting" (in a precise sense), omitting type information does not lead to type-incorrect proofs.
A sublanguage of type theory in which proof problems can be stated has been identified. In some applications involving refinements between abstract data types or theory morphisms, a more expressive language is required. In this case, the translation between the typed and untyped language could assign uninterpreted symbols to terms not forming part of the language identified in Section 2 (such as Lambda-abstractions, tuples or projections). It remains a topic of future research to precisely define these abstracting translations and to clarify their semantics and pragmatics. Also, the role of weaker equalities than Leibniz equality, for example the congruence relations often employed in specifications of abstract data types, has to be investigated. In principle, the verification that congruence properties of function symbols have been used correctly could be carried out in an *a posteriori* fashion, as described above for Leibniz equality. However, it may be wondered whether it is possible to show, in analogy to the results of Section 6, that there is no perceptible mismatch between the actual outcome of an equality proof and the expected result, as defined by the semantics of the equality relation involved.

Acknowledgments

The TYPELAB system is a joint development by the first author and Marko Luther. Suggestions by Friedrich von Henke, Holger Pfeifer, Harald Rueß, Detlef Schwier and Matthias Wagner have influenced the system design. We are grateful to Jörg

Denzinger and Matthias Fuchs for making the DISCOUNT system available and for providing assistance with the installation. Wolfgang Reif and Gerhard Schellhorn have patiently answered our questions about the integration between KIV and 3TAP.

References

[Bar96] Bruno Barras et al. *The Coq Proof Assistant Reference Manual, Version 6.1.* INRIA Rocquencourt and CNRS-ENS Lyon, November 1996.

[BHG+96] Bernhard Beckert, Reiner Hähnle, Karla Geiß, Peter Oel, Christian Pape, and Martin Sulzmann. The many-valued tableau-based theorem prover $_3T^AP$, version 4.0. Interner Bericht 3/96, Universität Karlsruhe, Fakultät für Informatik, 1996.

[Con86] R.L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[DGH+95] Bernd I. Dahn, Jürgen Gehne, Thomas Honigmann, Lutz Walther, and Andreas Wolf. *Integrating Logical Functions with ILF.* Humboldt-Universität Berlin, 1995.

[DKS97] J. Denzinger, M. Kronenburg, and S. Schulz. Discount - a distributed and learning equational prover. *Journal of Automated Reasoning*, 1997. To appear.

[HRS88] M. Heisel, W. Reif, and W. Stephan. Implementing verification strategies in the KIV-System. In *Proc. 9th CADE*, pages 131–140. Springer LNCS 310, 1988.

[HT93] John Harrison and Laurent Théry. Extending the HOL theorem prover with a computer algebra system to reason about the reals. In Jeffrey J. Joyce and Carl Seger, editors, *Proceedings of the 1993 International Workshop on the HOL theorem proving system and its applications*, volume 780 of *Lecture Notes in Computer Science*, pages 174–184, UBC, Vancouver, Canada, 1993. Springer-Verlag.

[KOS96] Christoph Kreitz, Jens Otten, and Stephan Schmitt. Guiding program development systems by a connection based proof strategy. In *LoPSTr'95: Proceedings of the 5th International Workshop on Logic Program Synthesis and Transformation*, pages 137–151. Springer LNCS 1048, 1996.

[Luo90] Zhaohui Luo. *An Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, July 1990.

[OSR93] S. Owre, N. Shankar, and J.M. Rushby. *The PVS Specification Language.* Computer Science Lab, SRI International, Menlo Park CA 94025, March 1993.

[Pau94] Larry Paulson. *Isabelle - a generic theorem prover.* Springer LNCS 828, 1994.

[Sor96] Maria Sorea. Integration von Gleichheitsbeweisen in einen typentheoretischen Beweiser. Master's thesis, Universität Ulm, 1996.

[vHLS97] F.W. von Henke, M. Luther, and M. Strecker. TYPELAB: An environment for modular program development. In M. Dauchet M. Bidoit, editor, *Proceedings TAPSOFT'97*, pages 851–854. Springer LNCS 1214, 1997.

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