T U M

I N S T I T U T F Ü R I N F O R M A T I K

The 1st Coq Workshop, Proceedings

Hugo Herbelin (Ed.)

TUM-I0919
August 09

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-08-I0919-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

The 1st Coq Workshop

21 August 2009

Munich, Germany

A satellite event of the 22th International Conference on Theorem Proving in Higher Order Logics conference (TPHOLs)

Preface

This first edition of the Coq workshop aimed at gathering the community of Coq users and developers around refereed contributed talks (to be later on re-submitted for publication to the Journal of Formalized Reasoning) and discussions on Coq. The Programme Committee consisted of:

Yves Bertot
Frédéric Blanqui
Jacek Chrząszcz
Eduardo Giménez
Georges Gonthier
Hugo Herbelin (chair)
Greg Morrisett
David Nowak
Benjamin Pierce

Advertised on the Coq Club and TYPES mailing lists, the workshop attracted seven submissions which were given to three members to review, and in a few cases additional reviews were solicited. The papers were discussed by e-mail within a 10-days period and the Program Committee selected six papers for presentation and pre-publication in these proceedings. In addition, the programme of the workshop included an invited talk, by Georges Gonthier (joint invited speaker with the ACM SIGSAM 2009 International Workshop on Programming Languages for Mechanized Mathematics Systems – PLMMS 2009).

I would like to thank all the authors who submitted a paper, and the Program Committee and external reviewers for the quality of their reports and for their feedback to the authors. I would like to thank the Technical University of Munich, the organisation committee of TPHOLs, especially Makarius Wenzel, Stefan Berghofer and Christian Urban, who helped us in making the actual workshop go well and in providing support for publishing these proceedings.

Through the Coq Technological Development Action, INRIA supported the workshop.

Hugo Herbelin

List of papers

Note: The page numbers are relative to the whole document. Each individual paper has its own pagination.

Sets in Coq, Coq in Sets

Bruno Barras

INRIA Saclay - Île de France

The title of this article refers to Werner's "Set in Types, Types in Sets" [13]. Our initial goal was to build formally a model of the Calculus of Inductive Constructions (CIC), the formalism of Coq. In [3], we formalized the syntactic metatheory of CIC and type-checking algorithms, under the assumption that our presentation enjoys the strong normalization property, which is the non-elementary step in proving the consistency of CIC.

The present work can be viewed as a first step towards the formalization of the semantics of CIC, concluding to strong normalization and consistency. Of course, due to Gödel's second incompleteness theorem, this can be fulfilled only under some assumptions that strengthen Coq's theory (unless the formalism is inconsistent). This approach is similar to Harrison's work about verifying HOL Light [8].

It is well-known that the Calculus of Constructions (CC, [4]) admits a finite model that is both classical and proof-irrelevant. The only requirement on such a model is to include booleans and to be closed by arrow type (non-dependent product). No infinite set is involved so we should be able to build a model of CC in the theory of hereditarily finite sets. However simple this description may seem, actually building a model for the common presentation of CC reveals technical traps as illustrated in [10]. The focus will be on the product fragment and on universes of CIC. A complete formalization of inductive types requires a lot of work. To show that our model construction can cope with inductive types, we have built a simple, yet recursive, inductive type: Peano's natural numbers. We have adopted a systematic approach and departed from the usual representation of natural numbers (ordinal ω).

The formal definitions of this article[1] can be organized in three categories: (1) developing a Coq library of common set theoretical notions and facts about pairs, functions, ordinals, transfinite recursion, Grothendieck universes, etc. (the Sets in Coq side), (2) building specific ingredients for models of typed λ-calculi, and (3) building set theoretical models of those theories within Coq (both fall into the Coq in Sets side).

1 Hereditarily finite sets

This is the V_ω set: the set obtained by applying ω times the powerset operation on the empty set. All the basic operations are decidable, so there is no distinction between intuitionistic and classical variants. The type of hereditarily finite sets can be defined as the type of well-founded, finitely branching trees:

```
Inductive hf : Set := HF (elts : list hf).
```

Of course, here we use lists for commodity, but order and repetition of elements in the list is not relevant. We thus need to express the equality as a setoid, in order to have rewriting

[1] See `http://www.lix.polytechnique.fr/Labo/Bruno.Barras/proofs/sets/`.

reasoning on sets. We will use the `let (xl) := x in ...` idiom (destructuring let) to get the list of subsets.

Equality and membership These two notions are mutually recursive: two set are equal if they contain the same elements, and a set is a member of another set if the latter contains an element that is equal to the former. This informal definition cannot be used as-is in Coq because of the strict syntactic guard condition that ensures that recursive definitions are well founded. One solution is to inline the membership definition in the equality. We first define universal and existential quantifiers on the members of a set. Note that they apply only structurally smaller sets to the predicate.

```
Definition forall_elt (P:hf->bool) x := let (xl) := x in List.forallb P xl.
Definition exists_elt (P:hf->bool) x := let (xl) := x in List.existsb P xl.
Fixpoint eq_hf x y {struct x} : bool :=
  forall_elt(fun x' => exists_elt(fun y' => eq_hf x' y') y) x &&
  forall_elt(fun y' => exists_elt(fun x' => eq_hf x' y') x) y.
Definition in_hf x y := exists_elt (fun y' => eq_hf x y') y.
```

We then show the basic facts that `eq_hf` (noted $==$) is an equivalence relation and that membership (noted as usual $\in$) is a morphism for our equality:

$$x == x' \ \wedge \ y == y' \ \wedge x \in y \ \Rightarrow \ x' \in y'.$$

Finite Zermelo-Fraenkel The various operations of HF can be implemented easily:

```
Definition empty    := HF nil.
Definition pair x y := HF(x::y::nil).
Definition union x :=
 HF(fold_set(fun y l => let (yl):=y in yl++l) x nil).
Definition subset x (P:hf->bool) :=
 HF(fold_set(fun y l => if P y then y::l else l) x nil).
Definition power x :=
 HF(fold_set (fun y pow p => pow p ++ pow (y::p)) x
             (fun p => HF (rev p) :: nil) nil).
Definition repl x (f:hf->hf) := let (xl) := x in HF(map f xl).
```

Let us recall that $\bigcup x$ is the union of the all the elements of x, so $a \cup b$ is encoded as $\bigcup\{a; b\}$. `subset` a P denotes $\{x \in a \mid P(x)\}$ where P is decidable. The powerset $\mathcal{P}x$ (`power` x) is the set of all subsets of x.[2] The last operation is the replacement: `repl` a f stands for the informal notation $\{f(x) \mid x \in a\}$, where f is a function of the meta-logic (Coq), so we can actually compute with sets. Iterator `fold_set` has type $\forall X, (\texttt{hf} \to X \to X) \to \texttt{hf} \to X \to X$ and is defined by `fold_set` f $\{x_1; \ldots; x_n\}$ $a = f\, x_1\, (\ldots (f\, x_n\, a) \ldots)$ taking care to cancel repetition of elements. This requirement will be used when defining the dependent product of two sets. A slightly more readable notation for `fold_set` $f\, x\, a$ is $\texttt{fold}_{y \in x}(X \mapsto f(y, X))\, a$.

At this point we can prove that those operators give a model of the Intuitionistic Zermelo-Fraenkel set theory (without the infinite set obviously). The eager reader can have a look at figure 1 for the precise statement of the "axioms" of the theory. From now on, we will not have to consider the actual representation of sets anymore.

[2]The `rev` is only for cosmetic reasons, so the elements of the subsets are displayed in the same order as the original set.

Ordered pairs and functions We follow the common usage to encode the ordered pair `(couple a b)` written (a, b) as $\{\{a\}; \{a; b\}\}$. Function f is coded as the set of couples $(x, f(x))$ where x ranges a given domain set. Typing of functions lead to introduce $\texttt{dep\_func}\, A\, B$ for $B : \texttt{hf} \to \texttt{hf}$, the set of dependent functions from $(x \in A)$ to $B(x)$, that is $\Pi_{x \in A} B(x)$. The formalization is quite standard, so we simply list the definitions and main facts:

$$\begin{aligned}
\texttt{fst}\ p &:= \bigcup\{x \in \bigcup p \mid \{x\} \in p\} \\
\texttt{snd}\ p &:= \bigcup\{y \in \bigcup p \mid \{\texttt{fst}\, p; y\} \in p\} \\
\texttt{lam}\ a\ f &:= \{(x, f(x)) \mid x \in a\} \\
\texttt{app}\ a\ b &:= \texttt{snd} \bigcup\{p \in a \mid \texttt{fst}\, p == b\} \\
\texttt{dep_func}\ A\ B &:= \texttt{fold}_{x \in A}(X \mapsto \{\{(x, y)\} \cup f \mid f \in X,\, y \in B(x)\})\, \{\emptyset\}
\end{aligned}$$

$$\begin{aligned}
\bigcup(a, b) = \{a; b\} \qquad \texttt{fst}\,(a, b) = a \qquad \texttt{snd}\,(a, b) = b \\
x \in a \ \Rightarrow\ \texttt{app}\,(\texttt{lam}\, a\, f)\, x = f(x) \\
(\forall x \in a, f(x) \in B(x)) \ \Rightarrow\ \texttt{lam}\, A\, f \in \texttt{dep_func}\, A\, B \\
f \in \texttt{dep_func}\, A\, B \ \wedge\ x \in A \ \Rightarrow\ \texttt{app}\, f\, x \in B(x) \\
f \in \texttt{dep_func}\, A\, B \ \Rightarrow\ f == \texttt{lam}\, A\, (\lambda x.\, \texttt{app}\, f\, x)
\end{aligned}$$

The fact that $\texttt{fold\_set}\, f\, x\, a$ does not apply the same element of x twice to f is crucial so that we actually build functional relations.

2 Intuitionistic Zermelo-Fraenkel

In this section, we will not proceed as for HF. Our primary goal is to use Coq as a prover for IZF, rather than comparing the theoretical strengths of Coq and IZF. This is why we will first proceed by defining a module interface that gathers the basic operations and axioms of IZF, and build a library of set theoretical constructions together with their properties. To make complex constructions easier, we have chosen a Skolemized presentation. Then, we will try to instantiate the signature. Such attempt to give a model of set theory within Coq has already been formalized by Werner.[3] Here, we recoded this work, and pushed further the study of universes.

2.1 IZF Axiomatization

We assume we have a type `set:Type` equipped with two relations $==$ and $\in$ such that set equality is extensional and membership is a morphism:

$$a == b \iff \forall x.\, x \in a \iff x \in b \qquad \text{and} \qquad a == a' \wedge a \in b \Longrightarrow a' \in b.$$

Next, sets can be constructed using the following constants: the `empty` and `infinity` sets, a binary operation `pair:set->set->set`, two unary operators `union` and `power`, and the replacement operator `repl:set` $\to$ (`set` $\to$ `set` $\to$ `Prop`) $\to$ `set`. They should satisfy the so-called "axioms of ZF" listed in figure 1. Replacement is obviously the one that calls the most for explanations. The introduction of a variable y' equals to y is to deal with cases where R is not a morphism.[4] The side condition (R is functional) does not require R to be total (unlike in HF). So, `repl` a R is the image of a by R, discarding those that are not in the domain of R. This allows to derive the comprehension scheme from replacement.

[3] This formalization is available as the Rocq/ZFC user contribution of Coq.

[4] Assuming we have two extensionally equal sets y and y' but intentionally distinct ($y == y' \wedge \neg y = y'$), then we could show that y belongs to `repl` $\{\emptyset\}$ $(\lambda\_z.\, z = y)$, but y' would not, in contradiction with the fact that $\in$ is a morphism.

$$
\begin{array}{rcl}
x \in \texttt{empty} & \Longrightarrow & \bot \\
x \in \texttt{pair}\, a\, b & \Longleftrightarrow & x == a \ \vee \ x == b \\
x \in \texttt{union}\, a & \Longleftrightarrow & \exists y \in a. x \in y \\
x \in \texttt{power}\, a & \Longleftrightarrow & \forall y \in x. y \in a \\
y \in \texttt{repl}\, a\, R & \Longleftrightarrow & \exists x \in a. \exists y'. y == y' \wedge R(x, y') \\
\multicolumn{3}{r}{(\textsf{if}\ \forall x\, x'\, y\, y'.\, x \in a \wedge R(x,y) \wedge R(x',y') \wedge x == x' \to y == y')} \\
x \in \texttt{infinite} & \Longleftarrow & x == \texttt{empty} \ \vee \ \exists y \in \texttt{infinite}.\, x == y \cup \{y\}
\end{array}
$$

Figure 1: Axioms of Zermelo Fraenkel

Another important remark about replacement is that the relation is a Prop-valued relation, as opposed to a first-order formula. This might strengthen the theory, since we can quantify over proper classes, thanks to the impredicativity of Prop.

2.2 A library of IZF constructions

We are now ready to formalize basic constructions such as pairs, relations and functions mostly in the same way as in HF, so we will resume the formalization at this point.

Disjoint sums The construction of a model for inductive types requires a notion of disjoint sum, in order to ensure that constructors build distinct elements. The definitions and expected properties about typing and elimination are the following:

$$
\begin{array}{rcl}
\texttt{inl}\, a & := & (0,a) \\
\texttt{inr}\, b & := & (1,b)
\end{array}
\qquad
\begin{array}{rcl}
\texttt{inl}\, a == \texttt{inl}\, a' & \Rightarrow & a == a' \\
\texttt{inr}\, b == \texttt{inr}\, b' & \Rightarrow & b == b' \\
\texttt{inl}\, a == \texttt{inr}\, b' & \Rightarrow & \bot
\end{array}
$$

$$
\begin{array}{rcl}
\texttt{sum}\, A\, B & := & \{(0,a) \mid a \in A\} \cup \{(1,b) \mid b \in B\} \\
a \in A & \Rightarrow & \texttt{inl}\, a \in \texttt{sum}\, A\, B \\
b \in B & \Rightarrow & \texttt{inr}\, b \in \texttt{sum}\, A\, B \\
p \in \texttt{sum}\, A\, B & \Rightarrow & (\exists a \in A, p == \texttt{inl}\, a) \ \vee \ (\exists b \in B, p == \texttt{inr}\, b) \\
A \subseteq A' \ \wedge \ B \subseteq B' & \Rightarrow & \texttt{sum}\, A\, B \subseteq \texttt{sum}\, A'\, B'
\end{array}
$$

Ordinals and fixpoints The classical definition of ordinals as hereditarily transitive sets and the successor of x as $x^+ = x \cup \{x\}$ raises problems in an intuitionistic setting. As remarked by Grayson, $y < x^+$ is equivalent to $y = x \vee y < x$, but not to $y \subseteq x$ unless we are classical. Taylor [12] introduced the notion of *plump* ordinals, which fixes that issue. Informally, a set x is a plump ordinal if (1) every element of x is an ordinal, and (2) for all ordinal z such that $z \subseteq y \in x$ for some y, then $z \in x$. Since the term ordinal occurs negatively in condition (2), we define ordinals in two steps. Firstly, `plump` $u\ x$ stands for x is a plump ordinal included in a well-founded set u; this is defined by well-founded induction on u. Secondly, we defined the class `isOrd` of well-founded sets that are plump ordinals bounded by themselves:

$$
\begin{array}{rcl}
\texttt{plump}\ u\ x & := & (\forall y \in u, y \in x \Rightarrow \texttt{plump}\ y\ y) \wedge (\forall z\, y.\, y \in u \wedge \texttt{plump}\ y\ z \wedge z \subseteq y \in x \Rightarrow z \in x) \\
\texttt{isOrd}\ x & := & \texttt{Acc}\ (\in)\ x \wedge \texttt{plump}\ x\ x
\end{array}
$$

The plump successor of x is then the set of plump ordinals included in x. So, $x^+ = \{y \in \mathcal{P}x \mid \texttt{isOrd}\ y\}$. To illustrate the difference, let us consider the ordinal 2: the classical successor of 1 is $\{\emptyset; \{\emptyset\}\}$, which is a boolean algebra. This contrasts with the plump successor of 1, which is the set of all $\{\emptyset \mid P\}$ for some proposition P. This forms a Heyting algebra.

We can define a transfinite operator `TR`. Intuitionistically, we cannot distinguish zero, successor and limit cases, so `TR` is parameterized by a step function $F : (\texttt{set} \to \texttt{set}) \to \texttt{set} \to \texttt{set}$ and the ordinal on which we iterate. Formally, `TR` is defined by replacement using the following relation (defined impredicatively):

$$\texttt{TR\_rel}\ o\ y := \forall P.\,(\forall f\,\alpha.(\forall \beta \in \alpha.P\ \beta\ f(\beta)) \Rightarrow P\ \alpha\ F(f,\alpha)) \Rightarrow P\ o\ y,$$

which is functional on the class of ordinals. This shows clearly the role of F: it produces the intended value for α, given (1) a function collecting all intended values for ordinals $\beta < \alpha$ and (2) α itself.

We define a specialized version of `TR` for the common cases where the limit case corresponds to the union of the previous results: given $F : \texttt{set} \to \texttt{set}$,

$$\texttt{TI}\ F\ o := \texttt{TR}\ (\lambda f\,\alpha.\bigcup\{F(f(\beta)) \mid \beta \in \alpha\})\ o$$

The main property of `TI` is that

$$\texttt{TI}\ F\ \alpha == \bigcup\{F\ (\texttt{TI}\ F\ \beta) \mid \beta < \alpha\}$$

This iterator has several interesting properties when F is monotone w.r.t. set inclusion ($\forall x\,y.\,x \subseteq y \Rightarrow F(x) \subseteq F(y)$): it forms an increasing sequence of sets all included in any pre-fixpoint of F.

$$\alpha \subseteq \beta \Rightarrow \texttt{TI}\ F\ \alpha \subseteq \texttt{TI}\ F\ \beta \qquad \texttt{TI}\ F\ \alpha^{+} == F\ (\texttt{TI}\ F\ \alpha) \qquad F(x) \subseteq x \Rightarrow \texttt{TI}\ F\ \alpha \subseteq x$$

Grothendieck universes The collection Grothendieck universes `grot_univ` is the collection of transitive sets U that are closed under all ZF operators: pairing, powerset, union and replacement (without assuming it contains the empty set or an infinite set). They have been introduced to avoid resorting to proper classes, which can be replaced by subsets of a universe. A set U is a Grothendieck universe if it satisfies the following closure conditions:

$$\begin{array}{rcl} y \in x\ \wedge\ x \in U & \Rightarrow & y \in U \\ x \in U\ \wedge\ y \in U & \Rightarrow & \{x; y\} \in U \\ x \in U & \Rightarrow & \mathcal{P}\,x \in U \\ I \in U\ \wedge\ (\forall x \in U.\,\forall y.\,R(x,y)\ \Rightarrow\ y \in U) & \Rightarrow & \bigcup\{y \mid \exists x \in I.\,R(x,y)\} \in U \\ & & (R \text{ functional}) \end{array}$$

It is straightforward to derive that Grothendieck universes are closed under dependent product, this is the reason why they play an important role in the interpretation of the `Type` hierarchy of CC_ω and CIC.

Grothendieck universes are stable by non-empty intersection, so we can define a functional relation between a universe U and the least universe that contain U, called the successor of U:

$$\texttt{grot\_succ}\,x\,y := \texttt{grot\_univ}\,y \wedge x \in y \wedge (\forall U.\,\texttt{grot\_univ}\,U \wedge x \in U \Rightarrow y \subseteq U)$$

Obviously, the successor universe cannot be built without an extra assumption. The Tarski-Grothendieck set theory (the formalism of Mizar) is ZF where we assume that for any set, there exists a universe that contains it. Clearly, in this theory, the replacement axiom lets us build an infinite sequence of nested universes.

2.3 An attempt to build a model of IZF

Model of IZF Following Peter Aczel's work [1], (well-founded) sets can be encoded in a tree-like datatype:

```
Inductive set : Type := sup (X:Type) (f:X->set).
Definition idx (x:set) : Type := let (X,f) := x in X.
Definition elts (x:set) : idx x -> set := let (X,f) := x in f.
```

Type X is used to index the direct elements of a set, and `elts` x i is the element of x with index i. The predicativity of inductive types in sort `Type` implies that the sort of X is lower than that of `set` so it is not possible to form the set of all sets by `sup set (fun x=>x)`.

Most of constructions can be implemented straightforwardly: pair $\{x; y\}$ can be coded by `(sup bool (fun b => if b then x else y))`; union of x is a set indexed by a dependent pair formed of an index i of x, and an index of the element of x with index i; powerset of x is a set indexed by predicates over indexes of x, yielding the subset of x which index satisfy the predicate.

We remark that we can define a weaker version of replacement where the relation can be expressed as a function at the meta-level:

$$\texttt{replf:set->(set->set)->set} \qquad x \in \texttt{replf}\, a\, f \iff \exists y \in a.\, x == f\, y$$

The set `replf` a f is indexed by the index set of a, and the element access function is just the composition of f with the access function of a. We remark that all the definitions of the previous paragraphs can be carried out with functional replacement, with the notable exception of ordinals.

The relational replacement is more delicate. Werner resorted to a type theoretical axiom of choice:[5]

```
Axiom choice : forall (A B:Type) (R:A->B->Prop),
  (forall x:A, exists y:B, R x y) -> exists f:A->B, forall x:A, R x (f x).
```

This axiom can transform any relation between sets into an existentially quantified function, than we can feed to `replf`. Thus, we can derive the existential version of replacement:

$$\forall a\, R.R \text{ functional} \Rightarrow \exists z, \forall y, y \in z \iff \exists x \in a, \exists y', y == y' \wedge R(x, y')$$

We slightly refined Werner's result by not requiring the excluded-middle to prove this.

In order to faithfully instantiate our IZF axiomatization, we need to Skolemize the replacement axiom. We have built a functor that, given a model of IZF where constructors are existentially quantified, produces a model of IZF as in section 2.1. Sets of the skolemized signature are predicates over sets of the input signature, that are satisfied by exactly one set.

Universes We are now trying to build a Grothendieck universe. For this we are considering that the `set` of the previous paragraph represents "small sets" and we are going to duplicate this set definition so that we can build a "big set" of all "small sets". We relate both types of sets by defining the copy of any small set at the big set level (which enforces that small sets live in a universe level less than or equal to that of big sets), and finally a big set U that contains a copy of every small set. This latter definition enforces that small sets live in a universe level strictly below that of big sets. The following definition already reflects this constraint.

[5] This axiom, called Type Theoretical Description Axiom by Werner, is weaker than the set theoretical axiom of choice since it does not imply the law of excluded-middle.

```
Inductive bigset : Type := bigsup (X:Type) (f:X->bigset).
Fixpoint copy (x:set) : bigset :=
  match x with
  | sup X f => bigsup X (fun i => copy (f i))
  end.
Definition U : bigset := bigsup set copy.
```

Equality and membership of small sets and their copies coincide. The next step would be to prove that U is a Grothendieck universe. Transitivity of U, closure of U under pair, union and powerset are straightforward. The relational replacement (`repl`) is also an internal operation of U.

One might expect that we can avoid needing the choice axiom by working in IZ + `replf`. Such a theory looks appealing since dependent products and λ-abstractions can be expressed easily. Unfortunately, we failed to prove that Grothendieck universes are closed under functional replacement: given a function that produces big sets and the logical assumption that those big sets are indeed in U, we cannot derive a function producing small sets, which would, by replacement at the small set level, witness that the big set built by replacement indeed belongs to U. Quite ironically, `choice` seems to be the only way to fix this issue.

2.4 Related Work

There already exists several formalizations of set theory in Coq. We already mentioned Werner's work [13]. The focus of this work is to study relationships between set theoretical and type theoretical formalisms. A fragment of plump ordinal theory is also formalized there.

Another related work is Simpson's user contribution Sophia-Antipolis/FunctionsInZFC which features an axiomatization of ZFC, and develops classical set theoretical notions.

Our present work takes from both above cited works, since it reconciles foundational investigations (like Werner's work), and a toolbox to mechanically verify proofs in set theory (like Simpson's work and subsequent formalizations in algebraic geometry). The contribution of this paper is, on the one hand, to study a bit further the encoding of Grothendieck universes (and inaccessible cardinals) in the sets-as-trees paradigm, and on the other hand, to develop as much as possible an intuitionistic set theoretical toolbox usable at a large scale.

3 Set theoretical model of the Calculus of Constructions

This section illustrates how these formalizations can be used to build set theoretical models of the Calculus of Constructions.

3.1 An abstract model of CC

We define an abstract model of the Calculus of Constructions: a structure ($\mathcal{X}$, $==$, $\in$, @, Λ, Π, $*$), with $==$ an equivalence relation, $\in: \mathcal{X} \to \mathcal{X} \to$ `Prop`, $@ : \mathcal{X} \to \mathcal{X} \to \mathcal{X}$, and Λ, Π of type $\mathcal{X} \to (\mathcal{X} \to \mathcal{X}) \to \mathcal{X}$, all should be morphisms. Finally, $* : \mathcal{X}$ will be the set of the propositions of CC. Such a structure is a model of the Calculus of Constructions if it satisfies the properties of figure 2.

3.2 Building the model in HF

In this section, we show that we can build such a model in HF. The first three conditions of an abstract model would suggest we can have @=`app`, Λ=`lam` and Π=`dep_func`, but the last one

$$
\begin{array}{rcll}
(\forall x \in A, f(x) \in B(x)) & \Rightarrow & \Lambda(A,f) \in \Pi(A,B) & (\Pi\text{-I}) \\
x \in \Pi(A,B) \ \wedge \ y \in A & \Rightarrow & @(x,y) \in B(y) & (\Pi\text{-E}) \\
x \in A & \Rightarrow & @(\Lambda(A,f),x) == f(x) & (\beta) \\
(\forall x \in A, B(x) \in *) & \Rightarrow & \Pi(A,B) \in * & (\textsc{Imp})
\end{array}
$$

Figure 2: Abstract model of the Calculus of Constructions

(impredicativity) cannot be satisfied. To turn around this, we use Peter Aczel's encoding of functions [1], that consists of encoding a function f by the set of pairs (x, y) such that y *belongs* (rather than being equal) to $f(x)$. Application is adapted so as to collect all the ys such that (x, y) belongs to the function. This way, the empty set is the function that maps any set to the empty set. Propositions are then either $\emptyset$ or $\{\emptyset\}$, hence the classical and proof-irrelevant nature of this model.

$$
\begin{array}{rcl}
\texttt{cc\_lam}\, A\, f & := & \{(x,y) \mid x \in A, y \in f(x)\} \\
\texttt{cc\_app}\, x\, y & := & \texttt{image}\, \{p \in x \mid \texttt{fst}\, p == y\} \\
\texttt{cc\_prod}\, A\, B & := & \{\texttt{cc\_lam}\ A\ (\lambda x.\, \texttt{app}\, f\, x) \mid f \in \texttt{dep\_func}\, A\, B\} \\
\texttt{props} & := & \mathcal{P}\, \{\emptyset\}
\end{array}
$$

3.3 Building the model in IZF

The abstract model of the Calculus of Constructions can also be instantiated in IZF, using the same definitions as in the previous paragraph. However, in IZF, the same definitions have a totally different meaning. Propositions are not mere booleans, but form a Heyting algebra, reflecting the meta-level propositions up to equivalence.

$$x \in \texttt{props} \iff x \in \mathcal{P}\, \{\emptyset\} \iff x = \{\emptyset \mid P\} \quad \text{for some proposition } P$$

Thus, the model is proof-irrelevant, but not classical.

3.4 Soundness of the abstract model

Here we are going to prove that the abstract model described previously allows to actually build an interpretation of terms and judgements of the Calculus of Constructions that will validate the typing rules of CC. The construction is independent of the way we choose to instantiate the abstract model (either using HF or IZF).

Our approach is to delay the introduction of the syntax as much as possible, so that our model is "open" in the sense that we can check the validity of new constructions or typing rules in a modular way. We will introduce the usual syntax of terms and typing rules only at the end of this section. It is then trivial to translate the syntax into the semantics.

Instead of defining the interpretation function by recursion on the syntax of terms, we represent terms as their interpretation function, that is a function that maps any valuation (assigning a set to every variable) to a set.

Valuations and associated operations (dummy valuation, extension and shift) are defined as:

```
Definition val := nat -> X.
Definition vnil : val := fun _ => props.
Definition vcons (x:X) (i:val) : val :=
  fun k => match k with 0 => x | S n => i n end.
Definition vshift (n:nat) (i:val) : val := fun k => i (n+k).
```

Terms Let us remark that our abstract model gives a way to interpret all kinds (it contains `Prop` and is closed by product), but it does not contain a set to interpret the sort `Kind`, which is not a finite type. So we use the `option` type to represent terms as either `Kind` or a function from valuations to sets. Since we want our interpretation to be a morphism, we get the following definition for terms:

```
Definition term := option {f:val->X|Morphism (eq_val ==> eqX) f}.
```

Terms are viewed either as objects (and they are encoded by an element of $\mathcal{X}$), or as a type (a set of elements of $\mathcal{X}$). The following two definitions reflect that remark (`int` gives the object level interpretation, and `el` the type level interpretation). Observe how `el` encodes that the denotation of `Kind` is the whole model $\mathcal{X}$. The object level interpretation of `Kind` is a dummy value since this sort (like any top sort of a PTS) can never appear in subject position in judgements.

$$\begin{array}{ll} \texttt{int} : \texttt{term} \to \texttt{val} \to \mathcal{X} & \texttt{el} : \texttt{term} \to \texttt{val} \to \mathcal{X} \to \texttt{Prop} \\ \texttt{int}\ (f, \_)\ i \ := \ f\ i & \texttt{el}\ (f, \_)\ i\ x \ := \ x \in f\ i \\ \texttt{int}\ \texttt{None}\ i \ := \ \texttt{props} & \texttt{el}\ \texttt{None}\ i\ x \ := \ \texttt{True} \end{array}$$

We can define the usual term constructors (using de Bruijn notations for variables). We leave out the proof that they are morphisms, which is straightforward.

$$\begin{array}{l} \texttt{prop} := (\lambda\_.\texttt{props},\ \_) \qquad \texttt{kind} := \texttt{None} \qquad \texttt{Ref}\ n := (\lambda i.\, i\ n,\ \_) \\ \quad \texttt{App}\ u\ v \ := \ (\lambda i.\, \texttt{app}\ (\texttt{int}\ u\ i)\ (\texttt{int}\ v\ i),\ \_) \\ \texttt{Abs}\ A\ M \ := \ (\lambda i.\, \texttt{lam}\ (\texttt{int}\ A\ i)\ (\lambda x.\, \texttt{int}\ M\ (\texttt{vcons}\ x\ i)),\ \_) \\ \texttt{Prod}\ A\ B \ := \ (\lambda i.\, \texttt{prod}\ (\texttt{int}\ A\ i)\ (\lambda x.\, \texttt{int}\ B\ (\texttt{vcons}\ x\ i)),\ \_) \end{array}$$

Although we do not have introduced the syntax yet, lifting of de Bruijn variables and substitution can be expressed as operations on the valuation:

$$\begin{array}{ll} \texttt{lift} : \mathbb{N} \to \texttt{term} \to \texttt{term} & \texttt{subst} : \texttt{term} \to \texttt{term} \to \texttt{term} \\ \texttt{lift}\ n\ (f, \_) \ := \ (\lambda i.\, f\,(\texttt{vshift}\, n\, i)\,,\ \_) & \texttt{subst}\ a\ (f, \_) \ := \ (\lambda i.\, f\,(\texttt{vcons}\ (\texttt{int}\, a\, i)\ i)\,,\ \_) \\ \texttt{lift}\ n\ \texttt{None} \ := \ \texttt{None} & \texttt{subst}\ a\ \texttt{None} \ := \ \texttt{None} \end{array}$$

Environments As usual in a de Bruijn setting, environments are lists of types, and they are deemed to denote valuations that map each variable to a value in the denotation of the type associated to this variable.

```
Definition env := list term.
Definition val_ok (e:env) (i:val) := forall n T,
  nth_error e n = value T -> el (lift (S n) T) i (i n).
```

Note that this is slightly more permissive than the typing rules, which generally rule out kind variables (when $T = \texttt{Kind}$).

Judgements We consider two semantic judgements, that intuitively correspond to equality and membership in the model:

- `eq_typ` which corresponds to convertibility. We will discuss later on why this judgement depends on the environment.
- `typ` which expresses typing.

```
Definition eq_typ (e:env) (M M':term) :=
  forall i, val_ok e i -> int M i == int M' i.
Definition typ (e:env) (M T:term) :=
  forall i, val_ok e i -> el T i (int M i).
```

Soundness of the model The goal now is to prove that our model is sound, which means that `eq_typ` admits all the rules of β-conversion (congruent equivalence relation including β-reduction). If we try to prove the admissibility of β-reduction in general, we fail because in a set theoretical model, functions do not behave like a λ-term outside their intended domain. So the property holds only for well-typed terms (condition `eq_typ e N T` below), which requires keeping track of the environment in equality judgements:

```
Lemma eq_typ_beta : forall e T M M' N N',
  T <> kind          -> typ e N T ->
  eq_typ (T::e) M M' -> eq_typ e N N' ->
  eq_typ e (App (Abs T M) N) (subst N' M').
```

This also explains why it is not as easy as expected (see [10]) to build set theoretical models of type systems which consider type convertibility as an untyped relation.

We can now prove that the semantic typing judgement admits all the rules of CC. Finally, by remarking that the denotation of $\forall P.\, P$ is the intersection of all proposition, and using the way we instantiated `props`, we can show that it is empty. This proves the logical consistency of CC. The model does not make use of ordinals or universes, so our consistency proof is axiom-free.

Syntax At this point, we may want to check that some given set of inference rules form a consistent theory. To do so, we just have to write a recursive function that maps syntactic terms to semantic terms (using the term constructors defined previously), prove that syntactical lifting and substitution is equivalent to the semantic operations. A final induction allows to prove that the typing judgements of CC (presented with a judgemental equality) imply the semantic judgements.

As a final remark, we obtain models of the common presentation of CC (with untyped equality) by showing the equivalence between both presentations. Adams [2] has proved this in the case of functional PTSs. We have formalised this proof, that we will not detail here.

Conclusions Several models of the Calculus of Constructions can be found in the literature [5, 10, 11]. We claim that this model is simpler in several respects:

- the definition of the interpretation function is straightforward and does not rely on excluded-middle, thanks to Aczel's trick.
- it does not consider a stratification of terms (as proof-terms, types and kinds), which do not generalize to universes.
- it admits extensions at the `Kind` level by any type whose denotation is a set constructible in IZF, since the denotation of `Kind` is the class of all IZF sets.

Building the model for system in judgemental equality presentation makes the soundness proof *much* easier. The technical difficulties (either resorting to a stratification of terms [5], or introducing some dynamic type-checking in the β-reduction [10]) are restrained to the equivalence proof with the untyped equality presentation.

This model also supports not so trivial extensions like `Prop`$\subset$`Kind`, but we shall remark that Adams' proof does not apply anymore since we have lost type uniqueness. So, the justification of such extension in a presentation of the calculus with untyped conversion remains open, to our knowledge.

4 Model of the Calculus of Constructions with Universes (CC_ω)

An abstract model of the Calculus of Constructions with universes (CC_ω) is an abstract model of CC, extended with a sequence $(u_i)_{i\in\mathbb{N}}$ that satisfies the following properties:

$$\begin{array}{c} * \in u_0 \qquad u_n \in u_{n+1} \qquad u_n \subset u_{n+1} \\ A \in u_n \ \wedge \ (\forall x \in A, B(x) \in u_n) \ \Rightarrow \ \Pi(A,B) \in u_n \\ A \in * \ \wedge \ (\forall x \in A, B(x) \in u_n) \ \Rightarrow \ \Pi(A,B) \in u_n \end{array}$$

CC_ω can be proven consistent in ZF [9]. But if we want a model that can cope with inductive types, there is little hope that we can escape without resorting to an infinite number of inaccessible cardinals, as shown in [13]. We found it easier to reason with Grothendieck universes, which directly gives us a set that is closed under all ZF set constructors, and thus closed under dependent products and inductive types. It is straightforward to prove that if we assume the existence of an infinite sequence of Grothendieck universes, the abstract model of CC_ω signature can be instantiated.

The model construction follows the same steps as for CC, with the difference that CC_ω has no top sort, so our interpretation domain is directly the type $\mathcal{X}$.

If we do not consider cumulativity (inclusion of $\mathtt{Type}_i$ in $\mathtt{Type}_{i+1}$), type uniqueness holds and so does the equivalence between untyped conversion and judgemental equality, and our model construction applies to the presentation with untyped conversion. However, adding cumulativity breaks type uniqueness and if we cannot generalize Adams' result, we will have to find another equivalence proof. Normalization by evaluation techniques might help here.

5 A Model of Natural Numbers based on Size Annotation

In this section we show that a simple inductive type (Peano's natural numbers) can fit into our model construction. We are going to follow a very general scheme so that the method generalizes to arbitrary inductive types. Inductive types are traditionally thought of as the least fixpoint of a (monotonic) type transformer.

Given the inductive structure of `nat`, we consider the type transformer $\mathtt{NATf}\ X := \mathtt{sum}\ \mathtt{UNIT}\ X$ (where `UNIT` is $\{0\}$). Constructor `ZERO` is `inl zero` and `SUCC` x is `inr` x. We can show that `ZERO` belongs to `NATf X` for any X, and if n belongs to X, then `SUCC` n belongs to `NATf X`.

The type of natural numbers with size annotations Termination of functions defined by structural recursion is ensured by type-checking [7], unlike the implementation of Coq which distinguishes the pattern-matching operator from the fixpoint operator, and uses a syntactic criterion to check that recursive calls are made only on structurally smaller terms as described in [6]. In [3], we proposed a variant of [7] where inductive types are annotated by size annotations. Such annotation is intended to denote an ordinal over which we iterate the constructor operator.

Using `TI`, we define `NATi` the transfinite iteration of `NATf`, i.e. it is the function $\alpha \mapsto \mathtt{NATf}^\alpha(\emptyset)$ for any ordinal α. At this point we do not use the fact that $\mathtt{NATi}\,\omega$ is a fixpoint of `NATf`.

Let us assume that P is a morphism and α an ordinal. Pattern-matching on a natural number n of size bounded by α leads to either $n == \mathtt{ZERO}$, or $n == \mathtt{SUCC}\,m$ for some m of size $\beta < \alpha$:

$$P(\mathtt{ZERO}) \ \wedge \ (\forall \beta < \alpha . \forall m \in \mathtt{NATi}\,\beta . P(\mathtt{SUCC}\,m)) \ \Rightarrow \ \forall n \in \mathtt{NATi}\,\alpha . P(n)$$

Replacement can be used to turn this specification into a set that is the denotation of a pattern-matching constant. Constructor discrimination and injection is needed to show that the output of pattern-matching (constructor case and arguments) is uniquely specified.

The fixpoint associated to natural numbers of size bounded by α can be expressed without any reference to the constructors:

$$(\forall\beta \leq \alpha.(\forall\gamma < \beta.\forall m \in \texttt{NATi}\,\gamma.P(m)) \Rightarrow (\forall n \in \texttt{NATi}\,\beta.P(n))) \;\Rightarrow\; \forall n \in \texttt{NATi}\,\alpha.P(n)$$

If we omit the ordinal annotations, this specification looks like a fixpoint operator of type $((\texttt{nat} \to P) \to (\texttt{nat} \to P) \to \texttt{nat} \to P$. Of course, ordinals are the guarantee that recursion terminates. The subtle relation between three sizes

- α the size of initial call to the recursive function,
- $\beta(\leq \alpha)$ the typical size of the input along the recursive calls
- and $\gamma(< \beta)$ the maximum size on which recursive calls are allowed

is a (yet somewhat informal) justification of the typing rules of fixpoints based on size annotation, as in [3].

Building the fixpoint We define `NAT` as $\texttt{NATi}\,\omega$. In order to show that `NAT` is a fixpoint of `NATf`, we prove that `sum` is continuous: for any I $(X_i)_{i\in I}$ and $(Y_i)_{i\in I}$,

$$\texttt{sum}\ \bigcup\{X_i \mid i \in I\}\ \bigcup\{Y_i \mid i \in I\} == \bigcup\{\texttt{sum}\ X_i\ Y_i \mid i \in I\}$$

So, $\texttt{NATf NAT} == \bigcup\{\texttt{NATf}\ (\texttt{NATi}\ n^+) \mid n < \omega\} == \bigcup\{\texttt{NATi}\ n^{++} \mid n < \omega\} == \texttt{NAT}$. From this fixpoint equation, it is straightforward to derive the usual eliminator on natural numbers:

$$P(\texttt{ZERO}) \;\wedge\; (\forall m \in \texttt{NAT}.P(\texttt{SUCC}\,m)) \;\Rightarrow\; \forall n \in \texttt{NAT}.P(n)$$

6 Conclusion and Future work

This article shows that formal semantics of expressive type theories are not out of reach anymore. We claim this, although there is a gap between informal and formal semantics that is often overlooked by authors.

This work can be followed in several directions:

- formalizing general inductive types: this requires more support for transfinite recursion and a characterization of the ordinal that makes all positive inductive definitions reach their fixpoint. Such an ordinal can be related to the cardinal of the universe in which the inductive definition lives in. Since a large piece of cardinal theory rely on the (set theoretical) axiom of choice, we might have to axiomatize ZFC.
- studying other models: set theoretical models are good to give an explanation of CIC that is compatible with the intuition of mathematicians; however in a programming language setting, models based on Scott domains are more pertinent; also, it would be interesting to look at how our models (designed initially to prove consistency) can be turned into strong normalization models.

References

[1] P. Aczel. Notes on constructive set theory, 1997.

[2] R. Adams. Pure type systems with judgemental equality. *Journal of Functional Programming*, 16 (2):219–246, 2006.

[3] B. Barras. *Auto-validation d'un système de preuves avec familles inductives.* Thèse de doctorat, Université Paris 7, Nov. 1999.

[4] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3), 1988.

[5] H. Geuvers and M. Stefanova. A simple model construction for the calculus of constructions. In *Types for Proofs and Programs*, pages 249–264. Springer-Verlag LNCS 1158, 1996.

[6] E. Giménez. Codifying guarded definitions with recursive schemes. In *TYPES '94: Selected papers from the International Workshop on Types for Proofs and Programs*, pages 39–59, London, UK, 1995. Springer-Verlag.

[7] E. Giménez. Structural recursive definitions in type theory. In *Proceedings of the International Colloquium on Automata, Languages and Programming*, pages 397–408, Aalborg, Denmark, 1998. Springer-Verlag LNCS 1443.

[8] J. Harrison. Towards self-verification of hol light. In U. Furbach and N. Shankar, editors, *Proceedings of the third International Joint Conference, IJCAR 2006*, volume 4130 of *Lecture Notes in Computer Science*, pages 177–191, Seattle, WA, 2006. Springer-Verlag.

[9] Z. Luo. *An Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1990.

[10] A. Miquel and B. Werner. The not so simple proof-irrelevent model of CC. In *TYPES*, 2002.

[11] M.-O. Stehr. *Programming, Specification, and Interactive Theorem Proving - Towards a Unified Language based on Equational Logic, Rewriting Logic, and Type Theory.* Doctoral thesis, Universität Hamburg, 2002.

[12] P. Taylor. Intuitionistic sets and ordinals. *Journal of Symbolic Logic*, 61:705–744, 1996.

[13] B. Werner. Sets in types, types in sets. In *Proceedings of TACS'97*, pages 530–546. Springer-Verlag, 1997.

A Tactic for Deciding Kleene Algebras

Thomas Braibant
ENS Lyon – INRIA

Damien Pous *
CNRS

Laboratoire d'Informatique de Grenoble, UMR 5217, France

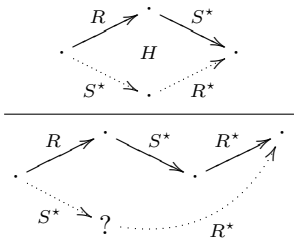
Abstract

We present a Coq reflexive tactic for deciding equalities or inequalities in Kleene algebras. This tactic is part of a larger project, whose aim is to provide tools for reasoning about binary relations in Coq: binary relations form a Kleene algebra, where the *star* operation is the reflexive transitive closure. Our tactic relies on an initiality theorem, whose proof goes by replaying finite automata algorithms in an algebraic way, using matrices.

Motivations

Proof *assistants* like Coq make it possible to leave technical or administrative details to the computer, by defining high-level tactics. For example, one can define tactics in order to solve decidable problems automatically (e.g., `omega` for Presburger arithmetic and `ring` for ring equalities). Here we present a tactic for solving equations and inequalities in Kleene algebras. This corresponds to a broader goal: providing tools (tactics) for working with binary relations. Indeed, Kleene algebras correspond to a non-trivial decidable fragment of binary relations. In the long term, we plan to use these tools for formalising process algebras and concurrency theory results: binary relations play a central role in the corresponding semantics.

A starting point for this work is the following remark: proofs about abstract rewriting (e.g., Newman's Lemma, equivalence between weak confluence and the Church-Rosser property, termination theorems based on commutation properties) are best presented using informal "diagram chasing arguments". This is illustrated by Fig. 1, where the same state of a typical proof is represented three times. Informal diagrams are drawn on the left. The goal listed in the middle corresponds to a naive formalisation where the points related by relations are mentioned explicitly. This is not satisfactory: a lot of variables have to be introduced, the goal is displayed in a rather verbose way, the user has to draw the intuitive diagrams on its own paper sheet. On the contrary, if we move to an algebraic setting (the right-hand side goal),




```
R,S: relation P
H: ∀ p,r,q, R p r → S* r q
    → ∃ s, S* p s & R* s q
p,q,q',s: P
Hpq: R p q
Hqs: S* q s
Hsq': R* s q'
============================
∃ s: P, S* p s & R* s q'
```

```
R,S: X
H: R·S* ⊆ S*·R*
=================
R·(S*·R*) ⊆ S*·R*
```

Figure 1: Diagrammatic, concrete, and abstract presentations of the same state in a proof.

*Work partially funded by the French ANR projet blanc "Curry-Howard pour la Concurrence" CHOCO ANR-07-BLAN-0324

where binary relations are seen as abstract objects, that can be composed using various operators (e.g., union, intersection, relational composition, iteration), statements and Coq's output become rather compact, making the current goal easier to read and to reason about.

Some technology is then required to avoid handling some administrative steps explicitly, which were somehow hidden in concrete proofs contexts. For example, in the right-hand side proof state of Fig. 1, we first need to re-arrange parentheses in order to be able to rewrite the goal using hypothesis H. This drawback is eliminated by defining adequate tactics to work modulo associativity and commutativity.

More importantly, moving to the abstract setting allows us to implement several decision procedures that could hardly be stated with the concrete presentation. For example, once we rewrite H in the right-hand side goal of Fig. 1, we obtain the inclusion $\mathrm{S}^\star{\cdot}\mathrm{R}^\star{\cdot}\mathrm{R}^\star{\subseteq}\mathrm{S}^\star{\cdot}\mathrm{R}^\star$ which is a (straightforward) theorem of Kleene algebras: it can be proved automatically thanks to the tactic we describe in this paper.

Outline. We recall the required mathematical background and we sketch the structure of the tactic in Sect. 1. We give some details about the underlying design choices in Sect. 2. Sect. 3 focuses on the algebraic part of the correctness proof, and the implemented algorithms are described in Sect 4. We conclude with directions for future work in Sect. 5.

1 Deciding equalities in Kleene algebras

Theoretical background. A Kleene algebra [22] is a tuple $\langle X, \cdot, +, 1, 0, \star\rangle$, where $\langle X, \cdot, +, 1, 0\rangle$ is an idempotent non-commutative semiring, and $\star$ is a unary operation on X, satisfying the following axiom and inference rules (where $\leq$ is the preorder defined by $x \leq y \triangleq x + y = y$):

$$1 + a \cdot a^\star \leq a^\star \qquad \frac{a \cdot x \leq x}{a^\star \cdot x \leq x} \qquad \frac{x \cdot a \leq x}{x \cdot a^\star \leq x}$$

Models of Kleene algebras include *regular languages*, where the star operation is language iteration; and *binary relations*, where the product $(\cdot)$ is relational composition, and star is reflexive and transitive closure (in this model, the above rules basically state that $a^\star$ is the least reflexive element which is stable under composition with a).

Thanks to finite automata theory (among others, Kleene [21], Rabin & Scott [29], Nerode [28]), equality of regular languages is decidable:

> *"two regular expressions denote the same regular language if and only if the corresponding minimal automata are isomorphic",*

and minimal automata can be computed as follows: 1) construct a non-deterministic finite automaton with epsilon-transitions (ϵ-NFA), by structural induction on the regular expression; 2) remove epsilon-transitions to obtain a non-deterministic finite automaton (NFA), by computing the closure of epsilon-transitions; 3) determinise the automaton using the accessible subsets construction, to obtain a deterministic finite automaton (DFA); 4) minimise the DFA by merging all states that are equivalent according to Myhill-Nerode's relation.

However, the above theorem is not sufficient to decide equality in all Kleene algebras: it only applies to the regular languages model. We actually need a more recent theorem, by Kozen [22] (also independently proved by Krob [24]):

> *"if two regular expressions α and β denote the same regular language, then $\alpha = \beta$ can be proved in any Kleene algebra".*

In other words, the algebra of regular languages is initial among Kleene algebras: we can use the above decision procedure to solve equations in an arbitrary Kleene algebra $\mathcal{A}$. The main idea of Kozen's proof is to encode automata using matrices over $\mathcal{A}$, and to replay automaton algorithms at this algebraic level. Indeed, a finite automaton with transitions labelled by the elements of $\mathcal{A}$ can be represented with three matrices $(u, M, v) \in \mathcal{M}_{1,n} \times \mathcal{M}_{n,n} \times \mathcal{M}_{n,1}$: n is the number of states of the automaton; u and v are 0-1 vectors respectively coding for the sets of initial and accepting states; and M is the transition matrix: $M_{i,j}$ is non-empty if there is a transition from state i to state j. This corresponds to the definition of an ϵ-NFA; definitions of NFAs and DFAs can easily be recovered by adding conditions on matrices u and M.

We remark that the product $u \cdot M \cdot v$ is a scalar, which can be thought of as the set of one-letter words accepted by the automaton. Therefore, in order to mimic the actual behaviour of a finite automaton, we just need to iterate over the matrix M. This is possible thanks to another theorem, which actually is the crux of the initiality theorem: *"square matrices over a Kleene algebra form a Kleene algebra"*. We hence have a star operation on matrices, and we can interpret an automaton algebraically, by considering the product $u \cdot M^\star \cdot v$. In the regular languages model, this expression actually corresponds to the language recognised by the automaton. We give more details about this proof in Sect. 3.

Overview of our strategy. We define a *reflexive* tactic. This methodology is quite standard: it is described in [2] and it was used by Grégoire and Mahboubi to obtain the current `ring` tactic [15]. Concretely, this means that we implement the decision as a Coq program (Sect. 4), so as to be able to prove its correctness within the proof assistant (Sect. 3):

```
Definition decide_Kleene: regexp → regexp → bool := ...
Theorem Kozen: ∀ a,b: regexp, decide_Kleene a b = true → a ≐ b.
```

The above statement corresponds to Kozen's theorem in the special case of the "free Kleene algebra": `regexp` is the obvious inductive type for regular expressions over a given set of variables, and $\doteq$ is the inductive equality generated by the axioms of Kleene algebras. Using Coq's reification mechanism, this is sufficient for our needs: the result can be lifted to other models using simple tactics (we return to this point in Sect. 2.3).

The equational theory of Kleene algebras is PSPACE-complete [25, 26]. Indeed, the determinisation phase of the algorithm we sketched above can produce automata of exponential size. Although this is not the case on the typical examples we tried, where our tactic runs almost instantaneously, this means that the `decide_Kleene` function must be written with some care, using efficient out-of-the-shelf automaton algorithms. Notably, the matricial representation of automata is not efficient for all stages of the decision procedure. Therefore, we need to work with other data-structures for automata, and to write the corresponding translation functions in order to reason about the algorithms in the uniform setting of matricial automata. We detail and justify our choices about these algorithms and data structures in Sect. 4.

2 Underlying design choices

Before going through Kozen's proof (Sect. 3) and giving details about our implementation of the decision procedure (Sect. 4), we explain the main choices we made about the structure of our development: how to represent the algebraic hierarchy, how to represent matrices, how to manage matrix dimensions, and how to resort to syntactical objects using reification.

2.1 Algebraic hierarchy

The mathematical definition of a Kleene algebra is incremental: it is a non-commutative semiring, which is itself composed of a monoid and a semi-lattice. Moreover, proofs naturally follow this hierarchy: when proving results about semirings, one usually rely on results about both monoids and semi-lattices. In order to structure our development in a similar way, we defined the algebraic hierarchy using Coq's recent *typeclasses* mechanism [30]: we defined several classes, corresponding to the different algebraic structures, so as to obtain the following "sub-typing" relations (these relations are projections, declared as morphisms to the typeclass system):

```
SemiLattice  <:
                 SemiRing <: KleeneAlgebra <: ...
  Monoid     <:
```

The other possibilities were to use *canonical structures* or *modules*. We tried the latter one; it was however quite difficult to organise modules, signatures and functors so as to obtain the desired level of sharing between the various proofs. In particular, when we consider more complex algebraic structures, we can no longer work with syntactical sub-typing between structures (we only have functors from one structure to another) and we lose the ability to directly use theorems, definitions, and tactics from lower structures in higher structures.

Except for some limitations due to the novelty of this feature, typeclasses happen to be much easier to use for our purposes: sharing is obtained in a straightforward way, the code does not need to be written in a monolithic way (as opposed to using functors), and it brings nice solutions for overloading notations (e.g., we can use the same infix symbol for multiplication in a monoid, a semiring, or a matrix semiring). We currently try to compare our strategy with that from [14, 5, 13], which is based on canonical structures. Although the aims of canonical structures and typeclasses are quite close, the underlying mechanisms lead to different constraints.

2.2 Matrices

Coq definition. A matrix can be seen as a partial map from pairs of integers to a given type X, so that a Coq definition of matrices and the sum operation could be the following:

```
Definition MX (n m: nat) := ∀ i j, i<m → j<n → X.
Definition plus n m (M N: MX n m) i j (Hi: i<n) (Hj: j<n) := M i j Hi Hj + N i j Hi Hj.
```

This corresponds to the dependent types approach: a matrix is a map to X from two integers and two proofs that these integers are lower than the bounds of the matrix. Except that they use vectors, this is the approach followed by [5, 13] and [6]. With such a representation, every access to a matrix element must be made by exhibiting two proofs, ensuring that the indices lie within the bounds. For simple operations like the above `plus` function this is not so problematic, this however requires more boilerplate when writing more complex operations like matrix multiplication or block decomposition operations. We actually chose to move these bounds checks to equality proofs only, by working with the following definitions:

```
Definition MX n m := nat → nat → X.
Definition equal n m (M N: MX n m) := ∀ i j, i<n → j<m → M i j == N i j.
Fixpoint sum i k (f: nat → X A B) := match k with 0 ⇒ 0 | S k ⇒ f i + sum (S i) k f end.
Definition dot n m p (A: MX n m) (B: MX m p) := fun i j ⇒ sum 0 m (fun k ⇒ M i k · N k j).
```

Here, a matrix is an infinite function from pairs of integers to X, and equality is restricted to the domain of the matrix. With these definitions, we do not need to manipulate proofs when defining matrix operations (like the above `dot` function), so that these definitions are both easier to write and more efficient to compute with. Bounds checks are required a posteriori only, when proving properties about these matrices operations, e.g., associativity of the product. This is easy in most cases: these proofs are done within the interactive proof mode, and can often be

solved with high level tactics like `omega`. We have not yet found drawbacks to this approach, we do not know whether it scales to more intensive usages like linear algebra [13].

Phantom types. Unfortunately, these definitions allow one to type the following code, where the three additional arguments of `dot` are implicit:

```
Definition ill_dot n p (M: MX n 16) (N: MX 64 p): MX n p := dot M N.
```

This definition is accepted by Coq because of the conversion rule: since `MX n m` is a dependent type that does not mention `n` nor `m` in its body, these type informations can be discarded by the Coq type system, using the conversion rule (`MX n 16 = MX 64 p`). This is not so terrible: such an ill-formed definition will be detected at proof-time. It is however a bit sad not to benefit from the advantages of a strongly typed programming language here. We partially solved this problem by resorting to an inductive singleton definition, reifying bounds in *phantom types*:

```
Inductive MX (n m: nat) := box: (nat → nat → X) → MX n m.
Definition get (n m: nat) (M: MX n m) := match f with box f ⇒ f end.
Definition plus (n m: nat) (M N: MX n m) := box n m (fun i j ⇒ get M i j + get N i j).
```

Coq no longer equates types `MX n 16` and `MX 64 p` with this definition, so that the above `ill_dot` function is rejected, and we can trust inferred implicit arguments (e.g., the `m` argument of `dot`). However, we need to artificially introduce the `box` at each matrix construction – `get` can be declared as a coercion. We still look for a better solution to this problem.

Computation. From a computational point of view, using lazy functions as a representation for matrices is two-edged : on the one hand, if the resulting matrix of a computation is seldom used, then computing the result point-wise, by need, is efficient; on the other hand, making numerous accesses to the same expensive computation may be a burden. Therefore, we have defined a *memoisation* operator that computes every point of a matrix, store the result in an associative map, and returns a function (of the same type) that accesses the associative map instead of recomputing the result. Since this memoisation operator can be proved to be an identity, it can be inserted in our code in a transparent way, at judicious places.

```
Lemma mx_force_id : ∀ n m (M : MX n m), mx_force M == M.
```

2.3 Graded algebras, typed reification

Adding types. *Square* matrices over a semiring form a semiring, and Kozen needed the extension of this folklore result to Kleene algebras [22]. For *rectangular* matrices, the various operations are only partial: dimensions have to agree. Therefore, with naive definitions of the algebraic structures, we are unable to use theorems and tools developed for monoids, semilattices, and semirings to reason about rectangular matrices. To remedy this problem, we

```
X: Type.

dot:  X → X → X.
one:  X.
plus: X → X → X.
zero: X.
star: X → X.

dot_neutral_left:
 ∀ x, dot one x = x.
...
```

```
T: Type.
X: T → T → Type.

dot:  ∀ n m p, X n m → X m p → X n p.
one:  ∀ n,     X n n.
plus: ∀ n m,   X n m → X n m → X n m.
zero: ∀ n m,   X n m.
star: ∀ n,     X n n → X n n.

dot_neutral_left:
 ∀ n m (x: X n m), dot one x = x.
...
```

Figure 2: From Kleene algebras to typed Kleene algebras.

generalised algebraic structures from the beginning, using *types*, which corresponds to working with *graded* algebras. An example is given in Fig. 2: a typical signature for semirings is presented on the left-hand side; we moved to the signature on the right-hand side, where a set `T` of types (or indexes) is used to constrain the various operations. These types can be thought of as matrix dimensions; we can also remark that we actually moved to a categorical setting: `T` is a set of objects, `X n m` is the set of morphisms from `n` to `m`, `one` is the set of identities, and `dot` is composition. As expected, with such definitions, one can form arbitrary matrices over a typed structure, and obtain another instance of this typed structure:

```
Instance mx_SemiRing: SemiRing → SemiRing := ...
Instance mx_KleeneAlgebra: KleeneAlgebra → KleeneAlgebra := ...
```

(The above code relies on our use of maximally inserted implicit arguments for the carrier and operations of the algebraic structures.) Then, thanks to typeclasses, we inherit all theorems, tactics, and notations we defined on generic structures, at the matricial level. Notably, when defining the star operation on matrices over a Kleene algebra, we can benefit from all tools for semirings, monoids, and semi-lattices, at the matricial level. This is quite important since this construction is rather complicated.

Removing types. Typed structures not only make it easier to work with matrices, they also give rise to a wider range of models. In particular, we can consider heterogeneous binary relations (between two distinct sets), rather than binary relations on a fixed set. This leads to the following question: can the usual decision procedures (for semi-lattices, semirings, and the one presented here for Kleene algebras) be extended to this more general setting?

Consider for example the equation $a \cdot (b \cdot a)^\star = (a \cdot b)^\star \cdot a$, which is a theorem of typed Kleene algebras as soon as a and b are respectively given types $n \to m$ and $m \to n$, for some n, m; how to make sure that the proof obtained by computing minimal (untyped) automata and concluding using Kozen initiality theorem is actually a valid, well-typed, proof?

For efficiency and practicability reasons, re-defining our decision procedures to work with typed objects is not an option (they are written as reflexive tactics). Instead, we managed to prove the following theorem, which allows one to erase types, i.e., to transform a typed equality goal into an untyped one:

$$\frac{T\Sigma \vdash u = v \qquad \Gamma \vdash u \rhd \alpha : n \to m \qquad \Gamma \vdash v \rhd \beta : n \to m}{\mathcal{A} \vdash \alpha = \beta : n \to m} \qquad (*)$$

Here, $\Gamma \vdash u \rhd \alpha : n \to m$ reads "under the evaluation and typing context Γ, the untyped term u can be evaluated to α, of type $n \to m$"; this predicate can be defined inductively in a straightforward way, for various algebraic structures. The theorem can then be rephrased as follows: "given an untyped equality proof of u and v, and typed interpretations α and β for u and v, we can construct a typed proof of $\alpha = \beta$". We proved it for semi-lattices, monoids, semirings, and Kleene algebras, so that all of our decision tactics apply to the typed setting – and in particular, to matrices. While this theorem is trivial for semi-lattices, and rather simple for monoids, difficulties arise with semirings and Kleene algebras, due to the presence of annihilator elements. Also note that Kozen investigated a similar question [23] and came up with a slightly different solution: he solves the case of the Horn theory rather than equational theory, at the cost of working in a restrained form of Kleene algebras. He moreover relies on model-theoretic arguments, while our considerations are purely proof-theoretic.

Typed reification. The above discussion about types raises another issue: reflexive tactics need to work with syntactical objects. For example, in order to construct an automaton, we

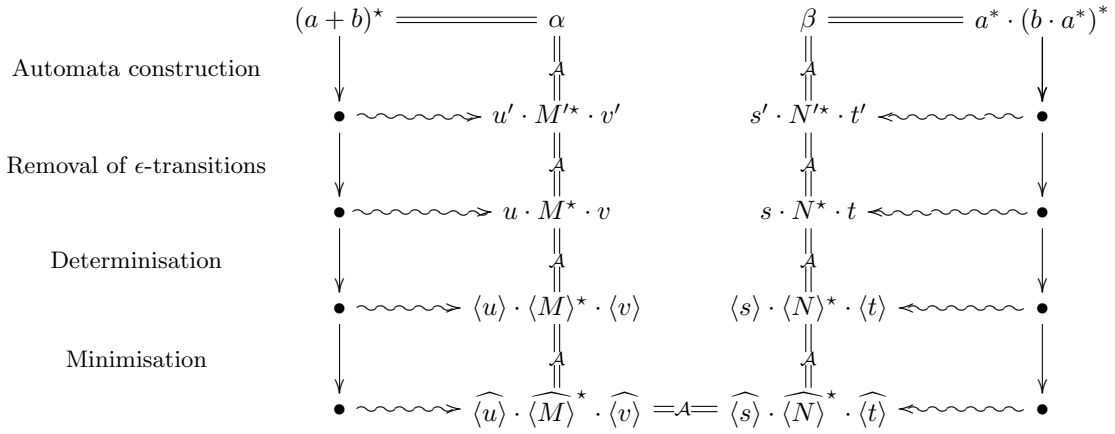


Figure 3: Soundness of `decide_Kleene`.

need to proceed by structural induction on the given expression. This step is commonly achieved by moving to the free algebra of terms, and resorting to Coq's reification mechanism (`quote`). However, this mechanism does not handle typed structures, so that we needed to re-implement it. Since we do not have binders, we were able do this within `Ltac`: it suffices to `eapply` theorem $(*)$ to the current goal, so that we are left with three goals, with holes for u, v and Γ; then by using an adequate representation for Γ, and by exploiting the very simple form of the typing and evaluation predicate, we are able to progressively fill these holes and to close the two goals about evaluation by repeatedly applying constructors and ad-hoc lemmas about environments. Unlike Coq's standard `quote`, which works by conversion and has no impact on the size of the current proof, this "lightweight"-`quote` generates rather large proof-terms. We would like to understand whether this situation can be improved, still remaining within `Ltac`.

3 Kozen's proof

The tactic we describe here relies on Kozen's initiality theorem: to prove that an equality $\alpha = \beta$ holds in any Kleene algebra, it suffices to check that the underlying minimal automata are isomorphic. The overall structure of Kozen's proof is depicted on Fig. 3: bullets represent idealised standard automaton constructions; the proof consists in showing that each construction can be related to a matricial automaton, whose interpretation is provably equal to the initial expression; we finally conclude by transitivity, if the minimal automata coincide. We briefly sketch the inner steps of this proof, i.e., the algebraic part, letting the reader refer to [22] for more details. The algorithms corresponding to the outer arrows are described in Sect. 4.

Building automata. There are several ways of constructing an ϵ-NFA from a regular expression [33]. We chose Thompson's construction [32] because of its simplicity: as described in [22], this is only a matter of block matrix constructions, and we easily show that the ϵ-NFA built from α evaluates to α, using algebraic laws. For example, the automaton for a sum is defined, and proved correct, as follows; the other constructions are obtained in a very similar way.

$$\left[\begin{array}{c|c} u & s \end{array}\right] \cdot \left[\begin{array}{c|c} M & 0 \\ \hline 0 & N \end{array}\right]^\star \cdot \left[\begin{array}{c} v \\ \hline t \end{array}\right] = \left[\begin{array}{c|c} u & s \end{array}\right] \cdot \left[\begin{array}{c|c} M^\star & 0 \\ \hline 0 & N^\star \end{array}\right] \cdot \left[\begin{array}{c} v \\ \hline t \end{array}\right] = \cdots = u \cdot M^\star \cdot v + s \cdot N^\star \cdot t$$

While these constructions are rather simple, they heavily rely on block matrix properties. The fact that we do not use dependent types to represent matrices greatly helps here.

Removing ϵ-transitions. The automata obtained with Thompson's construction may contain ϵ-transitions: their transitions matrices can be written as $M = J + \sum_{a\in\Sigma} a \cdot N_a$, where J and the N_a are 0-1 matrices, and J corresponds to the graph of ϵ-transitions. Removing these transitions to obtain an NFA usually means computing their reflexive and transitive closure, to update the other transitions. This can be done algebraically: thanks to the identity $(a+b)^\star = a^\star \cdot (b \cdot a^\star)^\star$ (a theorem of Kleene algebras), we have $u \cdot (J+N)^\star \cdot v = u \cdot J^\star \cdot (N \cdot J^\star)^\star \cdot v$, and the automaton on the right $(u \cdot J^\star, N \cdot J^\star, v)$ no longer contains ϵ-transitions. Indeed, $J^\star$ corresponds to the reflexive transitive closure of J.

Determinisation. The determinisation algorithm we implemented builds a DFA whose states are sets of states from the initial NFA; it consists in enumerating the set of subsets of states that are accessible from the set of initial states. Starting from a NFA (u, M, v) with n states, this algorithm returns a DFA $(\langle u\rangle, \langle M\rangle, \langle v\rangle)$ with $\langle n\rangle$ states, together with a map ρ from $[1..\langle n\rangle]$ to the subsets of $[1..n]$. We sketch the algebraic part of the correctness proof. By letting X denote the $(\langle n\rangle, n)$ 0-1 matrix defined by $X_{sj} \triangleq j \in \rho(s)$, we prove that the returned automaton satisfies the following commutation properties:

$$\langle M\rangle \cdot X = X \cdot M \quad (1) \qquad \langle u\rangle \cdot X = u \quad (2) \qquad \langle v\rangle = v \cdot X \quad (3)$$

The intuition behind X is that this is a "decoding" matrix: it sends the characteristic vectors of states of the DFA to the characteristic vectors of the corresponding subset of states from the NFA. Therefore, (1) can be read as follows: executing a transition in the DFA and then decoding the result amounts to decoding the given state and executing parallel transitions in the NFA. Similarly, (2) states that the initial state of the DFA corresponds to the set of initial states of the NFA. From (1), we can deduce $\langle M\rangle^\star \cdot X = X \cdot M^\star$ using a theorem of Kleene algebras, and we can conclude with $(2, 3)$: the two automata evaluate to the same value:

$$\langle u\rangle \cdot \langle M\rangle^\star \cdot \langle v\rangle = \langle u\rangle \cdot \langle M\rangle^\star \cdot X \cdot v = \langle u\rangle \cdot X \cdot M^\star \cdot v = u \cdot M^\star \cdot v \ .$$

Minimisation. The algebraic part of the correctness proof for minimisation is similar to that for determinisation. Starting from a DFA (u, M, v), the algorithm computes a partition of states, such that equivalence classes are stable under transitions and refine the partition of states between final and non-final. This partition is computed using Hopcroft's minimisation algorithm, which is described in Sect. 4; it is then converted into a map $[.]$ sending each state of the given DFA to the canonical representant of its equivalence class. This map allows us to define a decoding matrix Y by letting $Y_{ij} \triangleq [i] = j$, and the minimised automaton $(\widehat{u}, \widehat{M}, \widehat{v})$ is defined by:

$$\widehat{M} \triangleq Y^\top \cdot M \cdot Y \qquad \widehat{u} \triangleq u \cdot Y \qquad \widehat{v} \triangleq Y^\top \cdot v \ .$$

We finally prove that $Y \cdot \widehat{M} = M \cdot Y$ and $Y \cdot \widehat{v} = v$ (the first equality means that merging equivalence classes and then computing transitions in the minimised automaton amounts to computing transitions in the initial automaton and then merging the resulting states). As previously, this yields $\widehat{u} \cdot (\widehat{M})^\star \cdot \widehat{v} = u \cdot M^\star \cdot v$: the automata are equivalent.

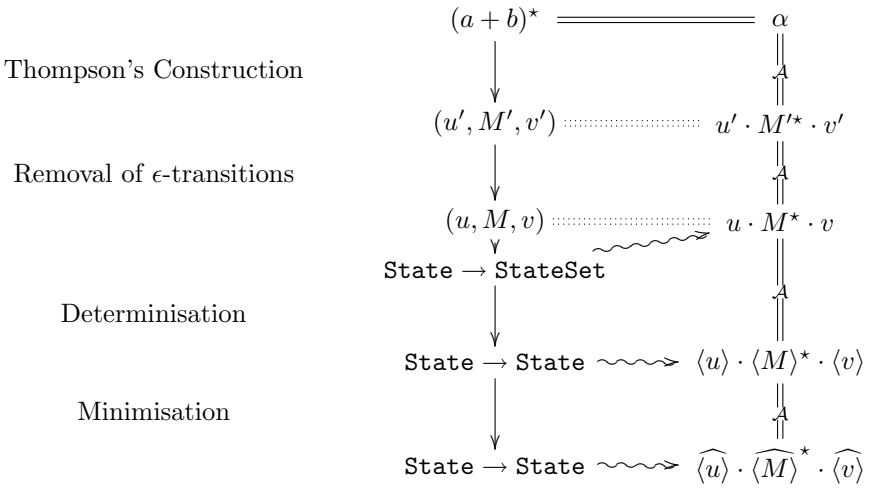


Figure 4: Proof and code relationship.

4 Implementing the decision procedure

We now focus on the external part of Fig. 3, that is, the algorithmic details of our implementation. As explained in the introduction, the equational theory of Kleene algebras being PSPACE-complete, we have to care about efficiency. This drives our choices about both data-structures and algorithms: accessible subset construction for determinisation, and Hopcroft's minimisation algorithm [1].

Boolean matrices. One cannot work with matrices over the free Kleene algebra: there are many terms that will never appear in automata matrices (like $(a+1)^\star$), and we would need to reason modulo the axioms of Kleene algebras, that equate, e.g., $1 + a$ and $0^\star + (1 + 0 \cdot b) \cdot a$. Since Thompson's construction yields automata whose transition matrix can be written as $M = J + \sum_{a \in \Sigma} a \cdot N_a$, where J and the N_a are 0-1 matrices, it actually suffices to work with matrices built upon the Kleene algebra of booleans: $\langle \text{bool}, \text{andb}, \text{orb}, \text{true}, \text{false}, \text{fun } \_ \Rightarrow \text{true} \rangle$. Then, in proofs, we inject these matrices into those built upon the free Kleene algebra (e.g., for evaluating automata formally). This allows us to write optimised functions for boolean matrices: there are only two values to consider and we can exploit laziness. In particular, removing ϵ-transitions can be done easily and efficiently with this representation of automata: as showed in Sect. 3, it suffices to compute the star of a boolean matrix ($J^\star$), and some multiplications ($u \cdot J^\star$ and the $N_a \cdot J^\star$).

FSets. The matricial representation of automata is no longer adequate when it comes to determinisation and minimisation. Therefore, starting from the ϵ-free NFA we built, we convert our matrices to more convenient representations, like transition functions. As can be seen on Fig. 4, we start with non-deterministic transition functions that map states to sets of states, and we later use deterministic transition functions that map states to states. To build efficiently these functions, we use the finite sets and finite maps libraries of Coq to represent state sets, partitions of states, and so on... These libraries being rather complete, this also gives us proper tools for proving the correction of our algorithms, and linking these structures to the matricial representation of automata (the horizontal arrows from Figs. 3 and 4).

```
! a          : label
! i          : state
! p,q,pt,pf : fset state
! P          : fset (fset state)
! L          : fset (label * fset state)

Variables states, finaux: fset state.
Variable  labels: fset label.
Variable  delta: state → label → state.

Definition delta_inv:
  fset state → label → fset state := ....

Definition splittable p a q :=
  let inv := delta_inv a q in
  let (pt,pf) := partition (fun i ⇒ i ∈ inv) p
  in if is_empty pt || is_empty pf
    then None
    else Some (pt,pf).

Definition update_splitters p pf pt L :=
  fold (fun a L ⇒ if (a,p) ∈ L
    then {(a,pf),(a,pt)} ∪ L\(a,p)
    else if cardinal pf < cardinal pt
      then {(a,pf)} ∪ L
      else {(a,pt)} ∪ L
  ) labels L.
```

```
Definition split P L (a,q) :=
  fold (fun p acc ⇒
    match splittable p a q with
      | None ⇒ acc
      | Some (pf,pt) ⇒
        let (P,L) := acc in
          ({pf, pt} ∪ P\p,
           update_splitters p pf pt L))
    end
  ) P (P,L).

Function loop P L {wf RPL (P,L)} :=
  match choose L with
    | None ⇒ P
    | Some x ⇒ loop (split P (L\x) x)
  end.

Definition partition :=
  loop
    {finals, states\finals}
    (labels × {finals}).
```

Figure 5: Coq code for minimisation.

Determinisation. Determinisation is exponential in worst case: this is a power-set construction. However, examples where this bound is reached are rather contrived, and the practical complexity is much better: most subsets of states cannot be reached from the subset of initial states. It is therefore crucial to implement the accessible subset construction, so as to avoid useless computations. We only give a very high level view of our implementation here: the standard algorithm is basically a while loop; that we translate into a tail-recursive fix-point; termination is not structural: it requires us to compute the exponential worst case bound, and we use a standard trick in order to avoid this useless and problematic computation. The proof of the algorithm requires us to find the adequate invariant for the loop; due to tail-recursion, this rather large invariant cannot be defined progressively with Coq's help: it has to be defined by hand, in a monolithic way.

Minimisation. We have to compute the Myhill-Nerode equivalence relation, which equates states sharing the same behaviour, i.e., accepting the same the language. The Coq implementation of Hopcroft's algorithm [17, 1] is sketched in Fig. 5: it consists in a 'while' loop containing two nested 'for' loops, translated using the `fold` operation of finite sets. The termination of the external loop is ensured using a well-founded relation (the algorithm could be rewritten so as to use structural recursion only, we found the resulting code less clear and harder to prove, however).

The idea of the algorithm is to start from an initial partition of states (final and non final states), and to refine this partition whenever one of its elements is `splittable`: i.e., when a move from a set of state can lead to two different sets by a transition with a given label `a`. The implementation of this predicate is made efficient by precomputing the inverse transition graph (`delta_inv`). Hopcroft then uses a set `L` of *splitters*, i.e., pairs (label, state set) w.r.t. which one must attempt to split classes of the partition. The crux of the algorithm is to keep from adding too much redundancy in `L`: if a pair `(a,q)` is not in this set, then either every class of

the partition is already split w.r.t. `(a,q)`, or `L` contains enough pairs to subsume `(a,q)`.

Treading through `L` in the main `loop` function, we dismiss the pairs `(a,q)` that do not split equivalence classes, and we update our partition `P` and the set `L` when `(a,q)` splits an equivalence class `p` into `pf` and `pt`. The update of potential splitters in `L` is based on the following remark: when `p` is split into `pf, pt`, then, for any label `a`, it suffices to split every other class `q` w.r.t. any two of `(a,p)`, `(a,pf)`, and `(a,pt)`. If $\texttt{(a,p)} \in \texttt{L}$, we must add both sub-splitters; if $\texttt{(p,a)} \notin \texttt{L}$, then `L` subsumes `(p,a)` and it suffices to add the smallest of `(a,pf)` and `(a,pt)` to `L`[1].At the end of the algorithm, since `L` is empty, we know that the equivalence classes of `P` cannot be split anymore: `P` is the Myhill-Nerode equivalence relation.

Avoiding automata isomorphism. The languages denoted by two regular expressions are equal if and only if their respective minimised automata are equal up-to isomorphism. By exploring all state permutations, this is sufficient to obtain decidability of regular languages equality. One can do a little better, however: it is not necessary to look for such a permutation. Suppose that languages α and β are represented by two DFAs; minimise the automaton whose set of states is the union of the states of the DFAs (i.e., the sum automaton), and test if the initial states of the two original DFAs are merged: these states are equivalent if and only if the DFAs recognise the same language, i.e., $\alpha = \beta$. This ends our description of the algorithm.

5 Conclusions and directions for future work

We presented a reflexive tactic for deciding Kleene algebra equalities. This tactic belongs to a broader project whose aim is to provide algebraic tools for working with binary relations in Coq; the development can be downloaded from [8]. To our knowledge, this is the first efficient implementation of these algorithms in Coq, and their integration into a generic tactic.

At the time we started this project, Briais formalised decidability of regular languages equalities [9] (but not Kozen's initiality theorem), without taking care about efficiency: determinisation is always exponential; instead of minimising automata, he relies on the 'pumping lemma' to enumerate the finite set of accepted 'small enough' words. As a consequence, even straightforward identities cannot be checked by letting Coq compute. These preliminary results lead us to restart from scratch and to look for a better strategy.

Narboux defined a set of tactics for formalising diagrammatic proofs in Coq [27]. He works in the concrete setting of binary relations, which makes it possible to represent more diagrams, but does not scale to other models. The level of automation is rather low: it basically reduces to a set of hints for the `auto` tactic.

Höfner and Struth used the automated first-order theorem prover Prover9 to automatically verify facts about boolean and relation algebras [18]. While these algebras feature the intersection and complementation operators (hence imposing a classical setting), they do not contain the Kleene star operation. Our approaches are quite different: while we implemented a decision procedure, their proposal is based on heuristics and learning techniques, within a resolution/paramodulation based framework.

We conclude this paper with directions for future work.

Optimisations. Even if our tactic works almost instantaneously on simple examples, such as the ones appearing in typical algebraic proofs, there is room for optimisation.

[1]The latter optimisation happened to be rather difficult to prove correct, so that we deactivated it our first release: the inner 'if' statement in the `update_splitters` function is replaced by $\texttt{(a,pf)} \cup \texttt{(a,pt)} \cup \texttt{L}$

- We use unary integers to represent states; this is a drawback when we memoise matrices or make comparisons of state sets. A first step would be to move to Coq's binary natural numbers (`N`); we plan to resort eventually to either n-ary integers [16], or machine integers [31].
- Although the algorithms we implemented for determinisation and minimisation are rather optimal, this is not the case for our construction algorithm (Thompson's one): we could use other algorithms [4, 10, 19], that produce smaller automata. Indeed, the complexity of the determinisation stage being potentially exponential in the size of the starting NFA, producing smaller automata from the beginning would improve the overall complexity.
- We have to implement a better algorithm for elimination of ϵ-transitions, which seems to be the current bottleneck of our tactic. This problem could also be solved by directly constructing ϵ-free automata, like Glushkov's one, or implement a construction like the one proposed in [19], which results in automata that do not contain cycles of ϵ-transitions – yielding to faster transitive closure algorithms.

Richer algebras. Kleene algebras lack several important operations from binary relations: intersection, converse, complement, residuals... We would like to develop tools for the corresponding algebras:

- *Kleene algebras with converse* should be decidable: since the converse operation commutes with all operations, we can imagine to push converses to the leafs of the terms, before applying our tactic for Kleene algebras.
- *Residuated semirings* [20], i.e., semirings with residual operations are decidable thanks to a Gentzen proof system having the sub-formula property. We plan to implement proof search for this proof system, either directly in `Ltac`, or using an external program to produce a trace that would then be reinterpreted as a Coq proof.
- *Allegories* [12] or *relation algebras* have an undecidable equational theory; they however provide means of encoding properties like well-foundedness [11], so that it would be interesting to provide tools for these structures (e.g., for solving decidable fragments).

Rewriting modulo A/AC. As explained in the introduction, some technology is required in order to work implicitly modulo associativity (A) and/or commutativity (C). For example, in the contexts below, we would like to rewrite the goal using hypothesis H without having to manually rearrange the goal first.

```
R,S,U,V: X          R,S,U,V: X          R,S,U,V: X
H: U·V = R          H: U+V = R          H: ∀ T, T·(T+U+V) = T
============       ==========          ====================
(U·U)·V = S         V+S+U = S           (U·R)·(V+R+U) = S
```

For this development, we wrote ad-hoc tactics `ac_rewrite` and `monoid_rewrite` that work in simple cases like the first two examples. However, a more systematic approach is required in order to handle situations like the third one. We plan to pursue Beauquier's work on this topic [3]: we would like to implement algorithms for matching modulo A and AC [7], and to integrate the resulting (external) program with Coq, in order to obtain more satisfying tools for rewriting modulo A and AC.

Acknowledgements. We would like to acknowledge Guilhem Moulin and Sébastien Briais, who worked with us in the early stages of this project. We are also grateful to Assia Mahboubi, Bruno Barras, and Hugo Herbelin for highly stimulating discussions, and to the anonymous referee who gave us very valuable comments.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
[2] S. F. Allen, R. L. Constable, D. J. Howe, and W. E. Aitken. The semantics of reflected proof. In *Proc. LICS '90*, pages 95–105. IEEE Computer Society, 1990.
[3] M. Beauquier. Application du filtrage modulo associativité et commutativité (AC) à la réécriture de sous-termes modulo AC dans Coq. Master's thesis, Master Parisien de Recherche en Informatique, 2008.
[4] G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48(3):117–126, 1986.
[5] Y. Bertot, G. Gonthier, S. Ould Biha, and I. Pasca. Canonical big operators. In *Proc. TPHOL '08*, volume 5170 of *Lecture Notes in Computer Science*, pages 86–101. Springer Verlag, 2008.
[6] F. Blanqui, S. Coupet-Grimal, W. Delobel, and A. Koprowski. CoLoR: a Coq library on rewriting and termination, 2006.
[7] A. Boudet, E. Contejean, and H. Devie. A new AC unification algorithm with an algorithm for solving systems of diophantine equation. In *Proc. LICS '90*, pages 289–299. IEEE Computer Society Press, 1990.
[8] T. Braibant and D. Pous. Coq development: Algebraic tactics for working with binary relations. Avalaible from `http://sardes.inrialpes.fr/~braibant/atwbr/`, May 2009.
[9] S. Briais. Coq development: Finite automata theory. Available from `http://www.prism.uvsq.fr/~bris/tools/Automata_080708.tar.gz`, July 2008.
[10] J.-M. Champarnaud and D. Ziadi. Computing the equation automaton of a regular expression in space and time. In *Proc. CPM*, volume 2089 of *Lecture Notes in Computer Science*, pages 157–168. Springer Verlag, 2001.
[11] H. Doornbos, R. Backhouse, and J. van der Woude. A calculational approach to mathematical induction. *Theoretical Computer Science*, 179(1-2):103–135, 1997.
[12] P. Freyd and A. Scedrov. *Categories, Allegories*. North Holland, 1990.
[13] F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging mathematical structures. In *Proc. TPHOL '09*, Lecture Notes in Computer Science. Springer Verlag, 2009. (To appear).
[14] G. Gonthier, A. Mahboubi, L. Rideau, E. Tassi, and L. Théry. A modular formalisation of finite group theory. In *Proc. TPHOL '07*, volume 4732 of *Lecture Notes in Computer Science*, pages 86–101. Springer Verlag, 2007.
[15] B. Grégoire and A. Mahboubi. Proving equalities in a commutative ring done right in Coq. In *Proc. TPHOL '05*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113. Springer Verlag, 2005.
[16] B. Grégoire and L. Théry. A purely functional library for modular arithmetic and its application for certifying large prime numbers. In *Proc. IJCAR '06*, volume 4130 of *LNAI*, pages 423–437. Springer Verlag, 2006.
[17] D. Gries. Describing an algorithm by Hopcroft. *Acta Informatica*, 2:97–109, 1973.
[18] P. Höfner and G. Struth. On automating the calculus of relations. In *Proc. IJCAR*, volume 5195 of *Lecture Notes in Computer Science*, pages 50–66. Springer Verlag, 2008.
[19] L. Ilie and S. Yu. Follow automata. *Information and Computation*, 186(1):140–162, 2003.
[20] P. Jipsen. From semirings to residuated Kleene lattices. *Studia Logica*, 76(2):291–303, 2004.
[21] S. C. Kleene. Representation of events in nerve nets and finite automata. In *Automata Studies*, pages 3–41. Princeton University Press, 1956.
[22] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994.
[23] D. Kozen. Typed Kleene algebra. Technical Report TR98-1669, Computer Science Department, Cornell University, March 1998.
[24] D. Krob. Complete systems of B-rational identities. *Theoretical Computer Science*, 89(2):207–343, 1991.
[25] A.R. Meyer and L. J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *Proc. SWAT '72*, pages 125–129. IEEE Computer Society, 1972.
[26] A.R. Meyer and L. J. Stockmeyer. Word problems requiring exponential time. In *Proc. STOC '73*, pages 1–9. ACM, 1973.
[27] J. Narboux. *Formalisation et automatisation du raisonnement géométrique en Coq*. PhD thesis, Université Paris Sud, September 2006.
[28] A. Nerode. Linear automaton transformations. In *Proc. of the AMS*, volume 9, pages 541–544, 1958.
[29] M.O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959.
[30] M. Sozeau and N. Oury. First-class type classes. In *Proc. TPHOL '08*, volume 4732 of *Lecture Notes in Computer Science*, pages 278–293. Springer Verlag, 2008.
[31] A. Spiwack. Ajouter des entiers machine à Coq. `http://arnaud.spiwack.free.fr/papers/nativint.pdf`, 2006.
[32] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11:419–422, 1968.
[33] B. W. Watson. A taxonomy of finite automata construction algorithms. Technical report, Computing Science, 1993.

Formalizing a SAT Proof Checker in Coq

Ashish Darbari, Bernd Fischer
School of Electronics and Computer Science
University of Southampton
Southampton, SO17 1BJ, England
email: {*ad06v,b.fischer*}*@ecs.soton.ac.uk*

Joao Marques-Silva
School of Computer Science and Informatics
University College Dublin, Belfield, Dublin 4, Ireland
email: *jpms@ucd.ie*

Abstract

Advances in SAT technology have made it possible for the SAT solvers to solve much bigger instances of problems using fewer resources. Much of the speed of these solvers comes from well-crafted optimizations but these complicate the implementations of the solvers, and make them vulnerable to bugs. However, assurance can be re-gained by use of a checker that validates the outcome of the solver. Two important aspects of this approach are (i) to ensure that the checker program itself is bug free, and (ii) is easy-to-use as a standalone executable.

We have designed and implemented a SAT proof checker using the Coq proof assistant. Our checker is capable of validating a SAT or an UNSAT claim of a SAT solver. In this paper we report on the more interesting aspect of checking the unsatisfiability claims, which have the form of a ground resolution proof. We present our formalization of the checker as a set of definitions within Coq, and characterize and prove its correctness properties. The proofs have been all machine checked in Coq, and an equivalent Ocaml executable program is extracted that can be used independently of the proof-assistant itself. Finally, we present some early evaluation results on industrial benchmarks to illustrate the strength of the extracted checker.

1 Introduction

Advances in SAT technology have made it possible for SAT solvers to be routinely used in the verification of large industrial problems. Moreover, they are now also used as back-end verification engines in several safety-critical domains such as railway systems [1] and avionics [2]. Such applications require some form of formal certification or guarantee that they are correct.

However, much of the performance enhancements in SAT technology come from well-crafted optimizations that make the SAT solvers vulnerable to implementation bugs. At the same time their complexity makes formal proofs of their correctness extremely difficult. For example, Lescuyer et al [3] formalized a SAT solver in the Coq proof assistant and extracted an executable program. The resulting program was mathematically rigourously checked, but its performance suffered, due to the lack of optimizations. Reasoning about these optimizations makes the formal correctness proofs exceedingly hard, as shown by Marić [4], who verified the pseudo-code of the SAT algorithm used in the ARGO-SAT solver but did not verify the solver itself.

An alternative, and more effective approach for ensuring correctness is to not formalize the SAT solver itself, but to instead formalize an independent *checker* in a proof-assistant, and use that checker to validate the outcome of the SAT solver. Weber and Amjad [5] proposed the idea of checking resolution proofs from SAT solvers by re-constructing them in LCF style higher-order logic theorem provers Isabelle, HOL 4, and HOL Light. They imported the proof trace output obtained from the proof-logging versions of Zchaff and Minisat into these theorem provers, and re-played the proofs to check whether they are valid. The benefit of this approach is that they can rely on the trusted LCF style kernel of the theorem provers to check the resolution proof obtained from the trace. However, a problem of this approach is that users need to be able to use these theorem provers in order to use the checker.

Our approach follows the general ideas of Weber and Amjad, but solves the (practical) problem of their work by extracting a stand-alone checker that can be used independently of the proof assistant. We have formalized and implemented a SAT checker called SHRUTI in Coq. Given a CNF description of the problem, and a proof trace obtained from a SAT solver, our checker can determine the validity of the claim made by the solver. Our formalization has two parts, one for checking the satisfiability claim (SAT), and another to validate the unsatisfiability claim (UNSAT). In this paper we present the formalization of the UNSAT part of the checker in the Coq proof assistant. We present some preliminary evaluation results on the industrial benchmarks from the SAT Race competition [6] to illustrate the strength of our approach.

2 Proof Checking Overview

Most SAT solvers can also produce a proof carrying the explanation about why the given problem was unsatisfiable when they produce an UNSAT answer. Any checker should be able to read these proof traces and should come up with a Yes/No answer depending on whether an outcome of the SAT solver is correct or not. In fact many of these solvers such as Zchaff, Minisat, Picosat and Booleforce provide a checker that does just that. However, none of these checkers are formally certified for correctness.

An UNSAT proof trace is a representation of general resolution proofs consisting of the original clauses used during resolution and the intermediate resolvents obtained by resolving the original input clauses. The parts of the proof which are regular input resolutions are called chains. The whole trace thus consists of original clauses and chains. Since a chain is a new proof rule, its input clauses are called 'antecedents' and the final resolvent simply 'resolvent'.

In order to design an efficient checking algorithm we made use of the resolution inference rule [7]. This rule takes a pair of clauses in disjunctive normal form, and produces a union of the two clauses, cancelling any complementary literals present in the two clauses. Of course, it is assumed that the input clauses themselves have no duplicate literals, and have no complementary literals within themselves. It is well known that this inference rule is sound and complete for propositional logic and the proof can be found in [8, 9]. When this inference rule is used to compute a resolution derivation on a set of clauses such that each resolved variable (i.e., the variable that occurs in the pair of complementary literals) is distinct and each clause is either an input clause or a derived clause obtained by the application of the resolution rule, the resolution derivation is called trivial resolution [10]. We often use the term 'trivial resolution' to mean the application of the 'resolution inference rule' since the application of the latter results in a trivial resolution.

The use of resolution rule ensures that the number of resolution steps taken to compute the final resolvent of a chain is linear with respect to the number of antecedents within the chain. Thus the

computation of a final resolvent in a chain begins at one end of the chain (in our case left most end of the chain) and uses each antecedent within the chain only once.

We decided to test our certified checker by reading the proof trace formats generated by Picosat, because it can also generate proof traces readable in ASCII form as compared to some of the other proof logging versions of solvers that only produces binary versions. Picosat [11] was also voted as one of the best SAT solvers in the industrial category of SAT Race 2007. Like many SAT solvers, Picosat reads the problem representation in DIMACS [12] notation. This uses non-zero integers to denote literals. A positive variable is denoted by a positive integer while its complement uses a negative integer. Zeroes are only used as delimiters. As an example consider the following unsatisfiable formula adapted from the `README.tracecheck` file distributed with Booleforce. It consists of all possible binary clauses over the two variables 1 and 2.

```
 1   2  0
-1   2  0
 1  -2  0
-1  -2  0
```

The zeroes at the end of rows are delimiters. A Picosat proof trace consists of such rows representing the input clauses, followed by rows encoding the proof chains. Each "chain row" consists of an asterisk (*) as place-holder for the chain's resolvent,[1] followed by the identifiers of the clauses involved in the chain. Each chain row thus contains at least two clause identifiers, and denotes an application of one or more of the resolution inference rule, describing a trivial resolution derivation. Each row also starts with a non-zero positive integer denoting the identifier for that row's (input or resolvent) clause. In an actual trace there are additional zeroes as delimiters at the end of each row, but we remove these before we start proof checking. The input to our checker thus looks as follows:

```
1   1   2
2  -1   2
3   1  -2
4  -1  -2
5   *   3  1
6   *   4  2  5
```

The first four rows denote the input clauses from the original problem (see above) that are used in the resolution, with their identifiers referring to the original clause numbering, whereas rows 5 and 6 represent the proof chains. In row 5, the clauses with identifiers 3 and 1 are resolved using a single resolution rule, whilst in row 6 first the original clauses with identifier 4 and 2 are resolved and then the resulting clause is resolved against the clause denoted by identifier 5 (i.e., the resolvent from the previous chain), in total using two resolution steps.

The algorithm of checking the validity of the proof trace relies singularly on the repeated use of the resolution rule. Checking begins at the first row of the proof chain (in the above example it would be 5), and the resolution rule is applied to all the antecedent clauses denoted by the identifiers in the chain. The resolvent clause (in this case consisting of $\{1\}$) is stored in a lookup table and is tagged with the key identifier 5. This process is then repeated for the next identifier in the proof chain (in our example it would be 6) and after two resolution rule applications an empty clause is obtained. If the empty clause is obtained then the given problem is UNSAT (i.e., the checker will

[1] This is generated by Picosat; there is another option of generating proof traces from Picosat where instead of the asterisk the actual resolvents are generated delimited by a single zero from the rest of the chain.

return a Yes answer), or else if all proof chain identifers have been checked and the empty clause is not derived, the given problem is not UNSAT (i.e., the checker will return a No answer). Correctness of the checking algorithm depends on the correct implementation of the resolution inference rule. The resolution rule itself is correct if it satisfies the following conditions:

1. All complementary literals are deleted from the given pair of clauses.
2. If the input pair of clauses contains a common literal then only one copy of that literal is made in the resolvent.
3. All unequal literals in the given pair of clauses are retained in the resolvent.

Additionally the resolution rule should produce an empty resolvent for a given pair of clauses that only contain complementary literals. A correctly implemented proof-checking algorithm would produce an empty clause from a proof trace for an unsatisfiable problem provided the trace contains a well-ordered chain of antecedents. If the ordering of the antecedents is not preserved which is the case with the compact resolution trace produced by Picosat, we are likely to introduce a scenario at the time of checking in which each trivial resolution step does not necessarily create a resolvent by cancelling complementary literals using linear number of steps. In other words the antecedents cannot be resolved to form a regular input resolution proof or the trivial proof efficiently. The *tracecheck* program distributed with Picosat can expand the trace output from Picosat and fix the ordering problem of the chains. It takes a resolution proof trace from Picosat as input and creates an extended resolution trace.

In the next section we present our formulation of the UNSAT part of the checker SHRUTI specifically showing the formalization of the resolution inference rule and we characterize its correctness properties by formalizing three main theorems.

3 Formalization of SHRUTI in Coq

3.1 Motivation for using Coq

We wanted to design a certified proof checker that can be formalized and mechanically verified using a proof-assistant to generate a high level of confidence, and at the same time enable the user to use it independently of the proof-assistant. We envision that by not compromising the safety, and enhancing ease-of-use, we can encourage the use of certified checkers as a regular component during the SAT checking flow. We therefore decided to use a proof-assistant in which it would be possible to achieve both our goals and the obvious choice was the Coq proof assistant. Coq has been widely used in several certification projects; most well known is the certification of a C compiler [13].

3.2 Formalizing SHRUTI

At the heart of SHRUTI is the formalization of the UNSAT part of the checker in Coq. The formalization makes use of a shallow embedding of the proof checker inside Coq using the data types and data structures of the Coq meta logic to represent the types and data structures of the proof checker. We then formulate definitions over these, and formally prove inside Coq that these definitions are correct. Once the Coq formalization is complete, Ocaml code is extracted from it through the extraction API that comes with Coq. At the time of extraction, the Coq data types/data structures are mapped to Ocaml data types/data structures. This way, we get the safe, static, one-off characterization in Coq combined with the run-time execution speed of Ocaml. The extracted Ocaml code expects to read its input data from data structures such as tables and lists. Data is stored in these from files containing the CNF description and the proof trace. This is

handled by some extra piece of Ocaml glue that wraps the extracted Ocaml code. The glue code also contains functions for profiling and logging the results in files. The result is then compiled into a native machine code executable that can be run independently of the proof-assistant Coq. A high-level architectural view is shown in Figure 1.

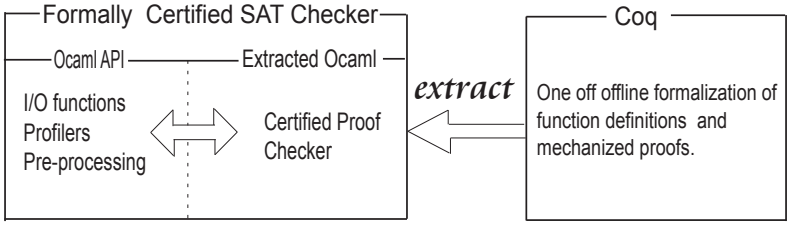


Figure 1: A High Level Architectural View of the Certified Checker.

We assume familiarity with the quantifiers ($\forall$, $\exists$) and logical connectives such as *and* ($\wedge$), *or* ($\vee$), *not* ($\neg$) and implication. We distinguish implication over propositions by $\supset$ and over types with $\rightarrow$ for presentation clarity, though inside Coq they are exactly the same. The notation $\Rightarrow$ is used during pattern matching (using match-with-end) as in other functional languages. For type annotation we use :, and for the cons operation on lists we use ::. Empty list is denoted by *nil*. The set of integers is denoted by Z, the type of polymorphic list by *list* and the type of list of integers by *list* Z. List containment is represented by $\in$ and its negation by $\notin$. The function *Zabs* computes the absolute value of an integer. We use Definition to denote the Coq function definitions. Main data structures that we have used in Coq formalization are lists, and finite maps. Finite maps or simply maps are functionally similar to hashtables with integer keys and polymorphic bindings although they are implemented using balanced binary trees.

To define the resolution function we make use of an auxillary function *union* which is defined below. This function takes as input a pair of clauses represented as a list of integers and an accumulator, and performs the functionality of the resolution operation.

Definition $union\ (c_1\ c_2 : list\ Z)(acc : list\ Z) =$
match c_1, c_2 with
| $nil, c_2 \Rightarrow app\ (rev\ acc)\ c_2$
| $c_1, nil \Rightarrow app\ (rev\ acc)\ c_1$
| $x :: xs,\ y :: ys \Rightarrow$ if $(x + y = 0)$ then $union\ xs\ ys\ acc$
else if $(Zabs\ x) < (Zabs\ y)$ then $union\ xs\ (y :: ys)\ (x :: acc)$
else if $(Zabs\ y) < (Zabs\ x)$ then $union\ (x :: xs)\ ys\ (y :: acc)$
else $union\ xs\ ys\ (x :: acc)$

The key feature of this function is that it expects the input clauses to be sorted by absolute value and the resolvent produced is also sorted by absolute value. This has the benefit of keeping the efficiency of the resolution operation linear in the size of the input clauses.

Definition $sorted =$
Inductive $sorted : list\ Z \rightarrow Prop :=$
| $sorted_0 : sorted\ nil$
| $sorted_1 : \forall z : Z.\ sorted\ (z :: nil)$
| $sorted_2 : \forall z_1\ z_2 : Z\ \ell : list\ Z.\ (Zabs\ z_1 \leq Zabs\ z_2)\ \supset sorted\ (z_2 :: \ell)\ \supset sorted\ (z_1 :: z_2 :: \ell)$

Note that in this predicate we do not enforce the constraint that an element has to be strictly less than the other, as we use the $\leq$ relation. However, when it comes to the proofs later on, this constraint is automatically enforced by stating that the clauses cannot contain duplicates or complementary literals.

Once the input clauses are sorted by absolute value, at the time of resolution each integer in the clause is compared pointwise. If the integers are complementary to one another then neither is added to the accumulator else the smaller (in terms of absolute value) of the two is added. If the integers are equal, one of them is stored in the accumulator. Once a single run of any of the clauses is finished, the accumulator's contents are sorted and then merged with the other, longer clause. Sorting is done by simply reversing the accumulator. This is because integers are added to the front of the list using the (::) operation, and the resulting accumulator has the final elements in descending order.

The actual binary resolution function is defined below. It is denoted by $\bowtie$ and makes use of the *union* function.

Definition $c_1 \bowtie c_2 = (union\ c_1\ c_2\ nil)$

To ensure that the formalization of our checker is correct we need to check that the resolution ($\bowtie$) function is defined correctly. What it means is that it should preserve the basic properties of the binary resolution function which are enumerated below:

1. Any pair of complementary literals is deleted in the resolvent obtained from resolving a given pair of clauses (Theorem 1).
2. All non-complementary literals that are pairwise unequal are retained in the resolvent (Theorem 2).
3. For a given pair of clauses, if there are no duplicate literals within each clause, then for a literal that exists in both the clauses of the pair, only one copy of the literal is retained in the resolvent (Theorem 3).

We have proven these properties in Coq. The actual proof, including several lemmas, comprises in total about 2000 lines of proof script in Coq.

For the sake of clarity in presentation we do not detail all the assumptions but we need to assume that the following assumptions hold for the three main theorems that we present later on.

1. No duplicates are allowed in each of the clauses c_1 and c_2.
2. There exists no mutually complementary pair of literals within each of the clauses c_1 and c_2.

These assumptions are essentially the constraints imposed on input clauses when the resolution function is applied in practice.

Since we have developed the machine checked proofs[2] we will not show the proof of these theorems in this paper. The general stratgey is to use structural induction on clauses c_1 and c_2. For each theorem, this results in four main goals, three of which are proven by contradiction since for all elements ℓ_1, $\ell_1 \notin nil$. For the remaining goal a case-split is done on if-then-else, thereby producing 8 sub-goals, some of whom are proven from induction hypotheses, and some from conflicting assumptions arising from the case-split. For others we employ a collection of special properties. Some of these are about integers and their reationship with their absolute values and the fact that these values appear in sorted lists without duplicates. Some are about counting

[2]Available at http://users.ecs.soton.ac.uk/ad06v/papers/coqwkshp09/

an element and its relationship with the $\bowtie$-function. The interested reader is referred to the online proof script. We will point out some of the main properties used in the proof of the theorems when we present the theorem.

Theorem 1. *All complementary literals are deleted:*

$$\forall c_1\ c_2.\ sorted\ c_1\ \supset\ sorted\ c_2\ \supset \\ \forall \ell_1\ \ell_2.\ (\ell_1 \in c_1)\ \supset\ (\ell_2 \in c_2)\ \supset\ (\ell_1 + \ell_2 = 0) \supset \\ (\ell_1 \notin (c_1 \bowtie c_2)) \wedge (\ell_2\ \notin (c_1 \bowtie c_2))$$

We make use of two important properties to prove two of the sub-goals arising in the proof of this theorem. The first property states that if an element is not present in either of c_1, c_2 or acc then it cannot be present in the resolvent of c_1 and c_2. The other important property states that if an element is already in acc then it exists in the resolvent of c_1 and c_2.

For the following theorem we need to assert in the assumption that for any literal in one clause there exists no literal in the other clause such that the sum of two literals is 0. This is defined by the predicate *NoMutualComp*.

Theorem 2. *All non-complementary, unequal pair of literals is retained:*

$$\forall c_1\ c_2.\ sorted\ c_1\ \supset\ sorted\ c_2\ \supset \\ \forall \ell_1\ \ell_2.\ (\ell_1 \in c_1)\ \supset\ (\ell_2 \in c_2)\ \supset (\ell_1 \neq \ell_2)\ \supset \\ (NoMutualComp\ \ell_1\ c_2) \supset (NoMutualComp\ \ell_2\ c_1)\ \supset \\ (\ell_1 \in (c_1 \bowtie c_2)) \wedge (\ell_2\ \in (c_1 \bowtie c_2))$$

For the proof we make use of an important property that states if an element is in clause c_1 and is not in clause c_2 and provided that its complement also does not exist in either c_1 or c_2 we will get that element in the resolvent of c_1 and c_2.

Theorem 3. *(Factoring) Only one copy of equal literal is retained:*

$$\forall c_1\ c_2.\ sorted\ c_1\ \supset\ sorted\ c_2\ \supset \\ \forall \ell_1\ \ell_2.\ (\ell_1 \in c_1)\ \supset\ (\ell_2 \in c_2)\ \supset\ (\ell_1 = \ell_2) \supset \\ ((\ell_1 \in (c_1 \bowtie c_2)) \wedge (count\ \ell_1\ (c_1 \bowtie c_2) = 1))$$

The proof of this theorem makes use of an important counting property. It states that if an element occurs in the accumulator acc once, and it exists in $union\ c_1\ c_2\ acc$ but it does not exist in c_1 or c_2, then it must only occur once in $union\ c_1\ c_2\ acc$.

In order to check the resolution steps for each row, we have to collect the actual clauses corresponding to their identifiers and this is done by the *findClause* function.

```
Definition findClause acc ctbl rtbl dlst =
  match dlst with
   | nil ⇒ (List.rev acc,true)
   | (x :: xs) ⇒
   match (find x rtbl) with
   | Some a ⇒
     findClause (a :: acc) ctbl rtbl xs
   | None ⇒
     match (find id ctbl) with
        | None ⇒ (acc,false)
        | Some a ⇒ findClause (a :: acc) ctbl rtbl xs
```

The function *findClause* takes a list of clause identifiers (*dlst*), an accumulator (*acc*) to collect the list of clauses, and requires as input a table that has the information about all the input clauses (*ctbl*). It also takes another table (*rtbl*) as an argument which is the table that contains the processed resolvents. Whenever a clause id is processed, then its resolvent clause is first looked up in the resolvent table, if that contains no entry for the given clause identifier, the clause is obtained from *ctbl*. If there is no entry in either of the tables, an error is signalled. It means there is a clause id for which there is no clause. This could be because there is an input/output problem with the proof trace file.

We then prove some sanity-checking properties about the maps. An obvious property that follows from the finite map implementation itself is that if a key is inserted in a table it will return some binding on being queried. We prove that if an entry is not found in the clause table and the resolvent table then the false flag is raised.

The function that uses the $\bowtie$ function recursively on a list of input clause chain is called *hyperResolution* and it simply folds the $\bowtie$ function from left to right for every row in the proof part of the proof trace file.

```
Definition hyperResolution lst =
  match (lst : list (list Z)) with
    | nil ⇒ nil
    | (x :: xs) ⇒ List.fold_left (⋈) xs x
```

The function *findAndResolve* is our last function defined in Coq world for UNSAT checking and provides a wrapper on other functions. The proof traces obtained from Picosat contains the proof chains specifying the clause identifiers used to derive the conflict, and the actual clauses that are used to generate the conflict. At the time of proof checking the pre-processed (trailing 0s removed) input proof trace is scanned and for each line in the trace it is either stored into a clause table (*ctbl*) since it represents an input clause, or in the trace table (*ttbl*) because it denotes a proof chain. The function *findAndResolve* then starts the checking process by first snarfing all the antecedents (identifiers for clauses) in a chain from the trace table, and then for each antecedent, obtains the actual clause either from the clause table or from the resolvent table by using the function *findClause*.

```
Definition findAndResolve ctbl ttbl rtbl id =
  let dlst = (find id ttbl) in
    match dlst with
      | None ⇒ (add id (0 :: nil) rtbl)
      | Some a ⇒
        let (cls,flag) =
        findClause nil ctbl rtbl a in
        match flag with
        | false ⇒ add id (0::(0::nil)) rtbl
        | true ⇒ add id (hyperResolution cls) rtbl
```

Once all the clauses are obtained for a single chain, the function *hyperResolution* is called and applied on the list of clauses for all proof chains. For each chain, the resolvent is stored in a separate resolvent table and tagged with the chain identifier from the trace table. It then checks whether the resolvent for the identifier of the last chain is an empty clause (i.e., empty list), and returns Yes (meaning that the solver's UNSAT claim is valid) if it finds one, else No.

We prove that if there is no binding for a given identifier in the trace table then a list with single zero is inserted in the resolvent table corresponding to this identifier. Similarly we prove that if the *findClause* function returns an error (flag is set to false) then a list with two zeroes is inserted in the resolvent table.

Since the proof trace obtained from Picosat contains proof chains that are a trivial resolution derivation, and since they are well-ordered, it is guaranteed that at the time of proof checking each application of the resolution inference rule will resolve at least one complementary pair of literals, thereby decreasing the count of total literals in the resolvent. For an UNSAT problem there would be enough proof chains and enough antecedents in each chain so that finite amount of resolution inference rule applications would eventually produce a pair of clauses with equal number of literals that are complementary to one another, and thus the final application of resolution rule would produce an empty clause. Our implementation of the resolution inference rule guarantees (due to the three main theorems presented) that this will happen provided the input proof trace is well-ordered and represents a chain of trivial resolution derivations.

We also check whether the given proof trace is a legal proof trace, i.e., whether any input clause used in any proof chain in the given trace is contained in the original problem. If the trace is not legal then the user gets a message and the checker aborts. However, this is provided as an option to the user at runtime, and if invoked, adds about 1-2% time overhead. This feature is currently optional because the uncertified checkers currently do not do this, and for comparing the runtime performances of our checker with uncertified ones we would like to disable this option at runtime to keep the comparison fair. Comparison results for these are still in process.

4 Results and Discussion

Benchmark	Proof Steps	SHRUTI (opt.)	SHRUTI (orig.)
een-tip-uns-nusmv-t5.B	122816	0.91	5.82
een-pico-prop01-75	246430	1.29	10.14
ibm-2004-26-k25	1132	0.004	0.02
ibm-2002-26r-k45	1105	0.001	0.02
ibm-2004-3_02_1-k95	114794	0.63	3.87
ibm-2004-6_02_3-k100	126873	0.76	4.92
ibm-2004-1_11-k25	254544	1.86	11.29
ibm-2002-07r-k100	255159	1.38	9.37
ibm-2004-2_14-k45	701430	6.59	51.51
manol-pipe-c10nidw_s	458042	2.82	19.51
manol-pipe-f6bi	1058871	10.32	97.85

Table 1: Results showing the times taken by our extracted checker on a sample of industrial benchmark problems from the SAT Race Competition. We show the number of proof steps obtained from Picosat/tracecheck in the second column. The third column shows the time taken by our optimized version SHRUTI (opt.) whilst the last one shows the timings obtained from the originally extracted, un-optimized version SHRUTI (orig.). The compiled binaries in both the cases were executed on a server running Red Hat Linux with Intel Xeon CPU 3 GHz, and 4GB memory. All times include resolution checking time, I/O and pre-processing times.

Our checker was tested on a chosen set of industrial benchmarks from SAT Race. The results

are summarized in Table 1. We pre-processed the input trace file to remove trailing zeroes using Ocaml routines. We do not enforce the check for duplicates in the input trace. If they are present in any line of the trace, they will "ripple out" in the resolvent, and an empty clause cannot be derived using the resolution based proof-checking. The input trace then no longer represents an UNSAT problem and the checker will simply return a No verdict.

We experimented with the extraction process and optimized the extracted Ocaml functions for efficiency. In our first implementation we only mapped the Coq lists to Ocaml lists. The resulting implementation (shown as SHRUTI (orig.) in the table) was more than one order of magnitude slower.

The Coq Zs were replaced with Ocaml integers and we replaced the Coq functions on Zs with the equivalent Ocaml functions. We also replaced the Coq finite maps with Ocaml finite maps and together with this change noticed a significant improvement.

Replacing Coq Zs with Ocaml integers and the maps gave a performace boost by a factor of 5-10. This can be perhaps attributed to the reduced overhead when dealing with Ocaml integer keys (in the Ocaml maps) directly, without having to convert between Coq Zs and Ocaml integers, and that all integer operations were now done on Ocaml integers.

A substantial bottle-neck in performance was the Ocaml garbage collector that unwittingly kicked in each time the number of inference steps exceed one million. The effect of this was almost an exponential drop in performance. We therefore changed the runtime settings of the garbage collector, by specifying large initial sizes for major and minor heaps and controlling the `space_overhead` and `max_overhead` settings such that minimal amount of garbage collection takes place. This enabled us to have much better execution times that scaled linearly with the number of inferences and we are now able to check proof traces with up to 15 million inferences. The results for this are shown as SHRUTI (opt.) in the table. Our extraction process only mapped Coq data structures to Ocaml data structures to enhance efficiency which is a standard practice in any program extraction based development in Coq. An important point to note here is that the core logic and functionality of the checker program is not compromised by program extraction in Coq.

5 Related Work

Lescuyer et al. [3] formalized a SAT solver in the Coq proof assistant and extracted an executable program. The resulting program was mathematically rigourously checked, but its performance suffered, due to the lack of optimizations. Recently Marić [4] proposed to verify a SAT solver with the low-level optimizations. He formalized the ARGO-SAT solver in Isabelle/HOL by modelling the solver and its low-level optimizations at an abstract level (pseduo code). One major difficulty is that it is difficult to formalize low-level optimizations (that work on real code used in a SAT solver) at a sufficiently abstract level without loosing sight of the low-level details. Even though optimizations were formalized, they were done for the pseudo-code, not the actual code that is used in the solver which still leaves the gap between what is formalized and what is used at runtime. Moreover, its practically not very useful (it took Marić one man year) to verify a solver, since it has to be done for each new solver. Instead its more efficient to verify a checker correct (since checkers are small and relatively straight-forward), and use it to validate outputs of any solver that produces proof certificates that were previously agreed.

Weber and Amjad [5] proposed the idea of checking resolution proofs from SAT solvers by re-constructing them in higher-order logic based theorem provers Isabelle, HOL 4 and HOL Light. They imported the proof trace output obtained from the proof-logging versions of Zchaff and Minisat

into these theorem provers, and re-played the proofs to check if they are valid.

A key difference between our checker and Weber and Amjad's is in the design and usage. In order to use Weber and Amjad's checker one has to have the different theorem provers installed, and more importantly the knowledge of using each one of them becomes paramount. In our case, we provide an executable binary that can be run independently of the Coq theorem prover or any other for that matter. Thus usability is considerably enhanced in our case. Weber and Amjad mostly reported their performance results on pigeon-hole problems and not much on industrial benchmarks. Pigeon hole problems though somewhat hard are also artificially created and thus share a common structure to them, so we personally decided not to calibrate our checker on these problems and instead chose to test ours on industrial benchmarks from the SAT Race Competition. We are investigating if we can get Weber and Amjad's checkers results' on the industrial benchmarks, and provide some comparison with our's. Bulwahn et al. [14] experimented with the idea of doing reflective theorem proving in Isabelle and suggested that it can be used for designing a SAT checker. In this sense their work is closest to our's. They proposed to enhance the functional core of Isabelle with imperative data structures for efficiency. However we have not seen the complete formalization of their SAT checker, and no benchmark results have been reported to the best of our knowledge. Recently there has been some work done in certifying SMT solvers notable amongst them are the work done by Moskal [15] and de Moura [16].

In a recent development related to Coq, there has been an emergence of a tool called Ynot [17] that can deal with arrays, pointers and file related I/O in a Hoare Type Theory. Future work in certification using Coq should definitely investigate the usage of this.

6 Conclusion

We presented the formalization of a SAT checker in the Coq proof assistant and presented some of the early benchmarking results. We observed that by using Coq we could do a one-off offline formalization of the checker and machine check all the proofs in Coq, while at the same when we extract an ocaml program, we obtain a fast executable binary, that can be used for checking industrial benchmarks as demonstrated by some of our results.

Acknowledgement

We thank Yves Bertot, Pierre Letouzey, and many more people on the Coq mailing list who helped us with Coq questions. We especially thank Tjark Weber and Hasan Amjad for answering our questions on their work. This work was funded by EPSRC, Grant # EP/E012973/1.

References

[1] M. Penicka, "Formal Approach to Railway Applications," in *Formal Methods and Hybrid Real-Time Systems*, Lecture Notes in Computer Science 4700, pp. 504–520. Springer, 2007.

[2] J. Hammarberg and S. Nadjm-Tehrani, "Formal Verification of Fault Tolerance in Safety-Critical Reconfigurable Modules," *International Journal on Software Tools for Technology Transfer* 7(3), pp. 268–279, 2005.

[3] S. Lescuyer and S. Conchon, "A Reflexive Formalization of a SAT Solver in Coq," in *Emerging Trends of the 21st International Conference on Theorem Proving in Higher Order Logics*, 2008.

[4] F. Marić, "Formalization and Implementation of Modern SAT Solvers," *Journal of Automated Reasoning* 43(1), pp. 81–119, 2009.

[5] T. Weber and H. Amjad, "Efficiently Checking Propositional Refutations in HOL Theorem Provers," *Journal of Applied Logic* 7(1), pp. 26–40, 2009.

[6] "SAT Race Competition," 2008. [Online]. Available: http://baldur.iti.uka.de/sat-race-2008/index.html

[7] J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle," *Journal of ACM* 12(1), pp. 23–41, 1965.

[8] J. A.Robinson, *Logic: Form and function - The Mechanization of Deductive Reasoning.* Elsevier, 1980.

[9] C.-L. Chang and R. C.-T. Lee, *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, 1997.

[10] P. Beame, H. Kautz, and A. Sabharwal, "Towards Understanding and Harnessing the Potential of Clause Learning," *Journal of Artificial Intelligence Research* 22, pp. 319–351, 2004.

[11] A. Biere, "PicoSAT Essentials," *Journal on Satisfiability, Boolean Modeling and Computation* 4, pp. 75–97, 2008.

[12] "Satisfiability Suggested Format," 1993. [Online]. Available: ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc/

[13] X. Leroy and S. Blazy, "Formal Verification of a C-like Memory Model and its uses for Verifying Program Transformations," *Journal of Automated Reasoning* 41(1), pp. 1–31, 2008.

[14] L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök, and J. Matthews, "Imperative Functional Programming with Isabelle/HOL," in *Proc 21st International Conference on Theorem Proving in Higher Order Logics*. Lecture Notes in Computer Science 5170, pp. 134–149. Springer, 2008.

[15] M. Moskal, "Rocket-Fast Proof Checking for SMT Solvers," in *Proc. 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 4963, pp. 486–500. Springer, 2008.

[16] L. M. de Moura and N. Bjørner, "Proofs and Refutations, and Z3," in *Proc. 7th International Workshop on the Implementation of Logics*, CEUR Workshop Proceedings 418, 2008.

[17] A. Nanevski, G. Morrisett, A. Shinnar, and P. Govereau, "Ynot: Reasoning with the Awkward Squad," *In Proc. 13th International Conference on Functional Programming*, pp. 229–240. ACM Press, 2008.

Proof Objects for Logical Translations

Andrew Polonsky Marc Bezem

Department of Informatics
University of Bergen
Norway

Abstract

One principal obstacle to integrating into Coq the full power of automated theorem provers is translation of the input into the prover's internal logic. Many translations introduce new predicates which actually change the logical meaning, making it difficult to recover a proof object of the input problem from the solution to the translated problem. We present a novel method for translating back proofs by reinterpreting the new predicates. The method is described for a particular prover logic, but is in principle generic. We illustrate the idea by a few examples in Coq. Since Coq allows abstraction over the new predicates, the proofs can be obtained with minimal overhead.

1 Introduction

Coq is a great interactive theorem prover. Yet as of today, complete formalization of complex results requires considerable amount of effort. Many small steps which would be left out of a published proof have to be explicitly justified by the user. In contrast, automated theorem provers can solve small problems completely autonomously, but are inherently incapable of dealing with deep mathematical results. If it were possible to integrate the automated First-Order Logic (FOL) provers into the Coq system, the amount of detail to be spelled out in formalizing a theorem would be greatly reduced.

The major obstacle to this enterprise is the fact that the resolution method, which is the basis of most high-performance FOL provers today, necessarily involves a normalization step converting the original problem into a Clause Normal Form, see [7, pp.19–99,273–333]. This step is difficult to undo at the level of proof objects. Skolemization in particular makes it difficult to lift the proof of the translated problem to a lambda term inhabiting the type of the original problem.

We present a translation with proof objects for Coherent Logic (CL), which extends resolution with existential quantifiers, making skolemization unnecessary. The Geo2007 prover is an example of an automated theorem prover based on Coherent Logic competing in CASC [4]. Our central result is that the proof objects for the input problem can be recovered from the proof of the translation with virtually no effort.

2 Coherent Logic

The general form of a coherent formula is

$$A_1 \wedge \cdots \wedge A_m \rightarrow B_1 \vee \cdots \vee B_n \tag{1}$$

with A_i atomic and B_i of the form

$$\exists \vec{y}. D_1 \wedge \cdots \wedge D_k$$

with D_i atomic. In contrast with resolution, where B_i must also be atoms, coherent logic allows B_i to be existentially quantified conjunctions of atoms. If the clause (1) contains free variables, these are implicitly universally quantified. Thus coherent clauses are interpreted as universal closures of (1).

One of the attractive features of Coherent Logic is that, in a sense, it is its own proof theory. Specifically, a set of coherent clauses serves both as a set of axioms as well as a complete set of deduction rules which generate all logical consequences of these axioms.

Definition 1. Let Γ be a set of facts (atomic formulas). Γ can be viewed as a coherent theory by taking the set

$$\{\top \rightarrow A\}_{A \in \Gamma}$$

Generally, $\top$ will be written on the left of the arrow in (1) to denote that $m = 0$. If $n = 0$, we will write $\bot$ on the right.

Definition 2. Let Γ be a set of closed facts, $\mathcal{T}$ a coherent theory, ϕ a fact. Let $\mathsf{dom}(\Gamma)$ denote the set of terms which occur in Γ. The relation $\Gamma \vdash_{\mathcal{T}} \phi$ is defined by induction:

Base $\Gamma \vdash_{\mathcal{T}} \phi$ if $\phi \in \Gamma$.

Induction Let $C = \bigwedge A_i \rightarrow \bigvee B_j$ be a clause in $\mathcal{T}$ and $\sigma : \mathsf{FV}(C) \rightarrow \mathsf{dom}(\Gamma)$ a substitution with $\{A_i^\sigma\}_{1 \leq i \leq m} \subseteq \Gamma$. If for each $1 \leq j \leq n$, $B_j = \exists \vec{y}. D_1 \wedge \cdots \wedge D_k$ we have $\Gamma \cup \{D_i^\sigma\}_{1 \leq i \leq k} \vdash \phi$, then $\Gamma \vdash \phi$.

With no loss of completeness, we will restrict our attention to *ground* reasoning. This means that all of the facts in Γ are closed and in the induction step the existentials introduce fresh constants into the D_i^σ.

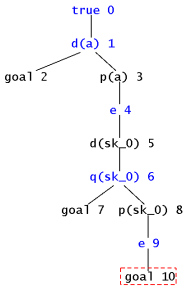


Figure 1: A derivation of *goal*

$$
\begin{array}{rcl}
\neg\neg\phi & \longrightarrow & \phi \\
\neg \bigwedge \phi_i & \longrightarrow & \bigvee \neg\phi_i \\
\neg \bigvee \phi_i & \longrightarrow & \bigwedge \neg\phi_i \\
\neg\exists\vec{x}\phi & \longrightarrow & \forall\vec{x}\neg\phi \\
\neg\forall\vec{x}\phi & \longrightarrow & \exists\vec{x}\neg\phi \\
(\phi \rightarrow \psi) & \longrightarrow & (\neg\phi \vee \psi) \\
(\phi \leftrightarrow \psi) & \longrightarrow & (\phi \wedge \psi) \vee (\neg\phi \wedge \neg\psi)
\end{array}
$$

Figure 2: NNF transformation

Example 3. Coherent derivations are usually visualized as trees. For example, consider the theory

$$
\mathcal{T} = \left\{
\begin{array}{rrcl}
 & \top & \rightarrow & d(a) \\
\forall x. & d(x) & \rightarrow & goal \vee (p(x) \wedge E) \\
 & E & \rightarrow & \exists y.(q(y) \wedge d(y)) \\
\forall x. & p(x) \wedge q(x) & \rightarrow & goal
\end{array}
\right.
$$

Then $\emptyset \vdash_{\mathcal{T}} goal$. The derivation as per Definition 2 is illustrated in Figure 1. Every node in the tree corresponds to a state Γ consisting of all the facts occurring on the path from the root to the node. Inference nodes (0,1,4,6, and 9) correspond to applications of the induction case of $\vdash_{\mathcal{T}}$, while the leaves correspond to the base case. See [6] for a detailed discussion. John Fisher implemented the "visual" prover which we used to generate the trees.

Fact 4 (Completeness of Coherent Logic, [1])**.** Let $\mathcal{T}$ be a coherent theory, ϕ a closed atomic formula. Then

$$\emptyset \vdash_{\mathcal{T}} \phi \iff \mathcal{T} \models \phi$$

where $\models$ is the Tarskian entailment of $\mathcal{T}$ as a first-order theory.

3 The Canonical Translation

The basic idea for a translation from FOL to CL was suggested in [1]. Essentially it consists of encoding the semantic tableaux proof method [7, pp.100–178] into a set of coherent clauses. A refutation of the given formula by a tableaux system corresponds to a derivation of $\bot$ from the translated theory by a CL prover.

Although general tableau rules for logical operators come in both polarities, we will find it convenient to restrict ourselves only to the positive rules. As such, we will consider *negation normal forms* (NNFs) as the basic forms of input, where negations only occur in front of atomic formulae and the only propositional connectives are $\wedge$ and $\vee$. A general FOL formula is converted into a NNF by normalizing it under the rules in Figure 2.

We now define the canonical translation from FOL to CL.

Definition 5. Let P be a formula in NNF.

- For an atomic predicate A occuring in P, let
  - T_A, F_A be fresh predicate symbols of the same arity as A,
  - C_A be the coherent clause

$$\forall \vec{x}. \quad T_A(\vec{x}) \wedge F_A(\vec{x}) \to \bot$$

- For a literal $L \subseteq P$, define the formula

$$L^t = \begin{cases} T_A(\vec{t}\,) & \text{if } L = A(\vec{t}\,) \\ F_A(\vec{t}\,) & \text{if } L = \neg A(\vec{t}\,) \end{cases}$$

- For a compound (non-literal) subformula $Q \subseteq P$, let
  - T_Q be a fresh predicate of arity $|\mathsf{FV}(Q)|$,
  - Q^t be the formula $T_Q(\vec{x})$, where $\vec{x} = \mathsf{FV}(Q)$,
  - C_Q be the coherent clause defined according to the top connective in Q, with implicit universal quantification over all free variables:

Q	$C_Q = \forall \vec{x}.$_____
$Q_1 \wedge Q_2 \wedge \cdots \wedge Q_n$	$Q^t \to Q_1^t \wedge Q_2^t \wedge \cdots \wedge Q_n^t$
$Q_1 \vee Q_2 \vee \cdots \vee Q_n$	$Q^t \to Q_1^t \vee Q_2^t \vee \cdots \vee Q_n^t$
$\exists \vec{y} R$	$Q^t \to \exists \vec{y} R^t$
$\forall \vec{y} R$	$Q^t \to R^t$

- The *canonical coherent theory of* P is the set

$$\mathcal{T}_P = \{\top \to P^t\} \ \cup \ \{C_A\}_{A \text{ atomic predicate in } P} \ \cup \ \{C_Q\}_{Q \subseteq P}$$

Definition 6. Let Γ be a first-order theory. The *canonical translation* of Γ is

$$\mathcal{T}_\Gamma = \bigcup_{\phi \in \Gamma} \mathcal{T}_{\mathsf{NNF}(\phi)}$$

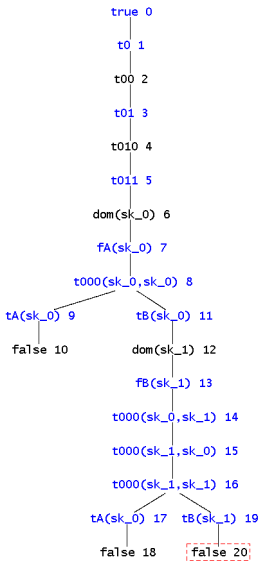


Figure 3: A derivation of **false**

Example 7. Suppose we wish to find a coherent theory equisatisfiable with the formula

$$\phi = \neg((\forall xy.Ax \vee By) \rightarrow (\forall x.Ax) \vee (\forall x.By))$$

The first step is to compute the negation normal form of ϕ:

$$\mathsf{NNF}(\phi) = (\forall xy.Ax \vee By) \wedge (\exists x.\neg Ax \wedge \exists y.\neg By)$$

Now the new predicates T_ψ for every $\psi \subseteq \mathsf{NNF}(\phi)$ are introduced: $T_{(\forall xy.Ax \vee By) \wedge (\exists x.\neg Ax \wedge \exists y.\neg By)}$, $T_{\forall xy.Ax \vee By}$, $T_{\exists x.\neg Ax \wedge \exists y.\neg By}$, $T_{Ax \vee By}(x,y)$, $T_{\exists x.\neg Ax}$, $T_{\exists y.\neg By}$, $T_A(x)$, $F_A(x)$, $T_B(x)$, $F_B(x)$.

Finally, one constructs the clauses as per Definition 5:

$$\mathcal{T}_\phi = \begin{cases} \top & \rightarrow & T_{(\forall xy.Ax \vee By) \wedge (\exists x.\neg Ax \wedge \exists y.\neg By)} \\ T_{(\forall xy.Ax \vee By) \wedge (\exists x.\neg Ax \wedge \exists y.\neg By)} & \rightarrow & T_{\forall xy.Ax \vee By} \wedge T_{\exists x.\neg Ax \wedge \exists y.\neg By} \\ T_{\forall xy.Ax \vee By} & \rightarrow & T_{Ax \vee By}(x,y) \\ T_{Ax \vee By}(x,y) & \rightarrow & T_A(x) \vee T_B(y) \\ T_{\exists x.\neg Ax \wedge \exists y.\neg By} & \rightarrow & T_{\exists x.\neg Ax} \wedge T_{\exists y.\neg By} \\ T_{\exists x.\neg Ax} & \rightarrow & \exists x.F_A(x) \\ T_{\exists y.\neg By} & \rightarrow & \exists y.F_B(y) \\ T_A(v) \wedge F_A(v) & \rightarrow & \bot \\ T_B(v) \wedge F_B(v) & \rightarrow & \bot \end{cases}$$

As the reader may have been expecting, the set of clauses above is inconsistent. The derivation of $\bot$ from $\mathcal{T}_\phi$ is shown in Figure 3, where *dom* is the domain-predicate and sk_i are fresh constants substituted for existentials.

4 Proof Objects

The refutation of a coherent theory consists, as we have seen, of applications of its clauses organized into a tree. Thus if ϕ is a FOL formula and $\mathcal{T}_{\neg\phi} = \{C_1, C_2, ..., C_n\}$ is the canonical translation of its negation, then the derivation of $\bot$ from this translation will give us a proof object of type

$$C_1 \to C_2 \to \cdots \to C_n \to \bot$$

At the same time, a proof of the original problem can be obtained by applying the classical `NNPP` term to the type

$$(\phi \to \bot) \to \bot$$

Thus the problem of proof reconstruction consists of inhabiting the C_i's in the context $Ax = \{s : \phi \to \bot\}$.

4.1 Erasing the ^{t} s

The key observation we wish to convey is that the coherent clauses C_Q become one-step tautologies if the formulas Q^t are replaced by Q. That is, if the fresh predicates T_Q are instantiated by the formulas Q. Indeed, the superscript ^{t} acts very much like a quote of a formula. The quick way to recover the proof is then to "unquote" all formulas. For example, let $Q = Q_1 \wedge Q_2$ be a subformula of P. Then the clause

$$C_Q = \forall \vec{x}.Q^t \to Q_1^t \wedge Q_2^t$$

would, after "erasing the quotes", become

$$\forall \vec{x}.Q_1 \wedge Q_2 \to Q_1 \wedge Q_2$$

which is a trivial tautology that can be easily solved by the `tauto` tactic. Generally, the types C_i corresponding to the coherent clauses C_Q are inhabited explicitly by small terms.

Inhabitation of the finish rules C_A for A atomic is just as easy. In accordance with instantiating the literals L^t with the formula L, the predicates F_A should be interpreted as negations $\neg A$. Thus the "bottoming out" clauses

$$C_A = \forall \vec{x}.T_A(\vec{x}) \wedge F_A(\vec{x}) \to \bot$$

are changed into the form

$$\forall \vec{x}.A(\vec{x}) \wedge (A(\vec{x}) \to \bot) \to \bot$$

These types too can be inhabited by explicit terms.

Finally, it remains to find the inhabitant of the start rule $\top \to (\neg\phi)$. But this is exactly the content of the term s provided in the context Ax of the prover. The start rule is inhabited by $\lambda o : \top.s$.

We have thus found an inhabitant of every type required by the prover. The recovery of the proof object for the input problem is complete.

The translation, proof search, and proof object recovery can all be fully automated to the point that given a (first-order) goal, one can actually get the Coq term inhabiting its type if the prover succeeds. Since the proof produced by the prover is generalized over all predicates and domains which are afterwards bound to explicit terms, the context is never polluted by any new variables. Thus we have a clean, complete, automated proof procedure for first-order tautologies. This process is illustrated in the following.

4.2 An Example

Example 8. Let P be the first-order tautology

$$P = (\forall xy.Ax \vee By) \to (\forall x.Ax) \vee (\forall x.By)$$

The negation normal form of $\neg P$ was computed in Example 7. Automation of translation and proof search yields a Coq term `p` of the following type (compare figure 3):

```
Coq < Check p.
p
     : forall (dom : Set) (goal : Prop) (fA fB : dom -> Prop)
         (t0 t00 : Prop) (t000 : dom -> dom -> Prop)
         (t01 t010 t011 : Prop) (tA tB : dom -> Prop),
       t0 ->
       (forall A : dom, tA A /\ fA A -> False) ->
       (forall A : dom, tB A /\ fB A -> False) ->
       (t0 -> t00 /\ t01) ->
       (forall A B : dom, t00 -> t000 A B) ->
       (forall A B : dom, t000 A B -> tA A \/ tB B) ->
       (t01 -> t010 /\ t011) ->
       (t010 -> exists A : dom, fA A) ->
       (t011 -> exists A : dom, fB A) -> goal
```

The actual term is rather large, and due to the space requirements we are not able to include it here. Notice however, that it is abstract not only in the predicates which occur in the input formula, but also the domain(s) of discourse and the goal of the prover, which is `False` in our case.

In order to get the proof of P, it remains to do the following.

1. Bind the newly introduced predicates to the formulas they represent.

```
Let tA := A.
Let fA := ∼ A.
Let tB := B.
Let fB := ∼ B.
Let t000(X,Y) := tA(X) \/ tB(Y).
Let t00 := (forall X Y : D, t000(X,Y)).
Let t010 := (exists X, fA X).
Let t011 := (exists Y, fB Y).
Let t01 := t010 /\ t011.
Let t0 := t00 /\ t01.
```

2. Construct proof objects for the coherent clauses.

```
Lemma ax1 V0 : tA V0 /\ fA V0 -> False.
intros v0 H; elim H. auto. Qed.
Lemma ax2 V0 : tB V0 /\ fB V0 -> False.
intros v0 H; elim H. auto. Qed.
Lemma ax3 : t0 -> t00 /\ t01.
Proof. trivial. Qed.
Lemma ax4 : forall X Y : D,  t00 -> t000 X Y.
Proof. trivial. Qed.
Lemma ax5 : forall X Y : D,  t000 X Y -> tA X \/ tB Y.
Proof. trivial. Qed.
Lemma ax6 : t01 -> t010 /\ t011.
Proof. trivial. Qed.
Lemma ax7 : t010 -> (exists X, fA X).
Proof. trivial. Qed.
Lemma ax8 : t011 -> (exists Y, fB Y).
Proof. trivial. Qed.
```

3. Apply the proof object of the translated theory to the translated clauses.

```
Theorem nnP: (forall X Y : D, (A X) \/ (B Y)) /\
((exists X : D, ∼(A X)) /\ (exists Y : D, ∼(B Y))) -> False.
Proof.
intro s. apply (p D False fA fB t0 t00 t000 t01 t010 t011 tA tB
s ax1 ax2 ax3 ax4 ax5 ax6 ax7 ax8). Qed.
```

As a result, we get the proof object of $\mathsf{NNF}(\neg P) \to \bot$, from which it is routine to obtain a lambda term inhabiting the type P.

5 Improved Translation

The canonical translation described above has the advantage of conceptual elegance and is simple to implement, but unfortunately the theories it produces are of very poor quality for the purposes of proof search. As such, we found it necessary to considerably extend the algorithm in order to overcome its previous shortcomings. The fact that proof objects could still be recovered with little effort demonstrates the flexibility and generality of the method.

The problems with the naive translation are illustrated by the formula

$$\forall xyz. \quad p(x,y,z) \to q(x,y,z).$$

Although this formula is already coherent, its translation is

$$\forall xyz. \quad \top \to F_P(x,y,z) \vee T_Q(x,y,z).$$

Since all implications are rewritten as disjunctions, any universal quantification over them will yield, during the forward ground proof search, an exponential tower of proof splits. If the clause was ever reached by the prover, it would give rise to a branch *for every sequence of triples* of elements from the domain.

Our solution was to allow disjunctions $\bigvee Q_i$ to generate clauses where some Q_i's occur in negative positions:

$$Q^t \wedge Q_i^f \to \bigvee_{j \neq i} Q_j^t$$

The general way to do this (allowing several Q_is to move to the left while containing the immense number of possible translations) is beyond the scope of this paper. The final algorithm produced superior theories without the problems of the naive approach, but at the cost of added complexity. Remarkably, recovery of proof objects remained virtually effortless. One way is to instantiate Q_i^f with $\neg Q_i$ and inhabitate

$$Q \wedge (Q_i \to \bot) \to (Q_1 \vee \cdots \vee Q_{i-1} \vee Q_{i+1} \vee \cdots \vee Q_n)$$

A simpler approach is to instantiate Q_i^f with $\neg Q_i^t$, and, before the Q^ts are instantiated, to apply a term of type

$$\begin{aligned}&(Q^t \to Q_1^t \vee \cdots \vee Q_n^t) \to \\ &Q^t \wedge \neg Q_i^t \to Q_1^t \vee \cdots \vee Q_{i-1}^t \vee Q_{i+1}^t \vee \cdots \vee Q_n^t\end{aligned}$$

to the required clause. Of course, all such types are inhabitated mechanically.

6 Function Symbols and Equality

So far we paid no special attention to the role of functions and/or equality in translations. However, the Geo system, developed by Hans de Nivelle, supports equality reasoning and uses a different translation scheme. Our proof reconstruction technique still applies, albeit with certain adaptations, all of which are very straightforward.

Geo2007 is based on geometric logic and participates in the CADE ATP System Competition [4]. In the category FOF (first-order formulas) it solved 31% of the problems, down from 48% in 2006 (for unknown reasons). In the category FNT (model finding for first-order formulas), Geo2007 performed very well: 81% of the models were found, only marginally behind the winner Paradox, which solved 85%.

The translation from FOL to CL used by Geo has one remarkable feature, namely that function symbols are completely eliminated. Every n-ary function symbol is eliminated by introducing a new $(n+1)$-ary predicate symbol representing the graph of the function. Even constants are eliminated, using unary predicates. For example, constant a is eliminated from $p(a)$ by introducing a new unary predicate A and by replacing $p(a)$ by $\forall x.\ A(x) \rightarrow p(x)$, under the addition of the axiom $\exists x.\ A(x)$. No axioms postulating uniqueness are needed. The function f in, for example, $\forall x.\ p(f(x))$ is eliminated by introducing a new binary predicate F and by replacing $\forall x.\ p(f(x))$ by $\forall xy.\ F(x,y) \rightarrow p(y)$, under the addition of the axiom $\forall x\ \exists y.\ F(x,y)$. Again, uniqueness of y for any given x is not needed. The translated problems can be shown to be equisatisfiable with the original problems, see [3], but are of course far from equivalent with the original ones. This makes this translation into an interesting challenge for our approach. We will elaborate one example, but the method works in general. Consider the following theorem-to-prove in Coq.

```
Parameter dom: Set.
Parameter c:dom.
Parameter f: dom -> dom.
Parameter p: dom -> Prop.
Parameter q: dom -> Prop.
Theorem x: (forall A: dom, p A -> q (f (f A))) ->
           (forall A: dom, q A -> p (f A)) ->
           (forall A: dom, p A \/ q A) ->
            exists B: dom, p B /\ q B.
```

Although a nice exercise (which, by the way, could be varied by taking different combinations of iterations `(f ... (f A)...)`; not all of them are provable!), we prefer to mechanize the proof. However, the current version of Geo2007 does not generate Coq proof objects. Therefore we use the Prolog prototype coherent prover [2], which does generate Coq proof objects.

The above theorem is already in coherent format, but uses a function. We essentially use the Geo approach for eliminating this function, but the particular way of refuting the existential conclusion is in the style of [2]. For reasons of uniformity we maintain Coq syntax. We now state the translated problem, extending the Coq script above.

```
Parameter goal: Prop.
Parameter gf: dom -> dom -> Prop.
Theorem y:
        (forall A B C: dom, p A /\ gf A B /\ gf B C -> q C) ->
        (forall A B: dom, q A /\ gf A B -> p B) ->
        (forall A: dom, p A \/ q A) ->
        (forall A: dom, p A /\ q A -> goal) ->
        (forall A: dom, exists B: dom, gf A B) ->        goal.
```

The system CL from [2], promptly and diligently, generates the proof object `y`. Actually, as the translated problem is dealt with in a different section, the proof object obtained abstracts from the context. This means that actually a higher-order version has been proved: for all domains, constants, functions, propositions and predicates we have the result. This of course reflects the freedom one has in Tarskian semantics to interpret the syntax *ad libitum*. Let us denote the abstracted proof object for `y` by `abstract_y`. We now use the freedom of interpretation to reinterpret `goal` and `gf` in order to obtain a proof object for `x`. To this end we should bind `goal` to the conclusion `exists B: dom, p B /\ q B` and `gf` to the graph of `f`, that is, to `fun X Y: dom => f X = Y`. This amounts to the following application term:

```
apply (abstract_y dom c p q (fun X Y: dom =>  f X = Y)
                            (exists X:dom, p X /\ q X)).
```

(there may be some variation in the order in which the arguments are abstracted).

Of course we do not get the conclusion for free, and the costs are five proof obligations, one for each assumption in `Theorem y`. However, the first three follow trivially from the corresponding assumptions in `Theorem x`. Let's take the first, the other two are even easier. As `gf` is bound to `fun X Y: dom =>  f X = Y` this proof obligation boils down to:

```
forall A B C: dom, p A /\ f A = B /\ f B = C -> q C)
```

which can be automatically inferred from

```
forall A: dom, p A -> q (f (f A)))
```

As `goal` is bound to `exists X:dom, p X /\ q X`, the fourth proof obligation is simply existential introduction. The fifth proof obligation is a trivial consequence of the binding of `gf` and the reflexivity of equality. All this can be mechanized in a simple and general way.

References

[1] M. Bezem and T. Coquand. Automating Coherent Logic. In G. Sutcliffe and A. Voronkov, editors, *Proceedings LPAR-12*, number 3825 in Lecture Notes in Computer Science, pages 246–260, Berlin, 2005. Springer-Verlag.

[2] M. Bezem and D. Hendriks. Web page including CL tool, input files, Coq files. `http://www.cs.vu.nl/~diem/research/ht/`.

[3] H. de Nivelle and J. Meng. Geometric Resolution: A Proof Procedure Based on Finite Model Search. In J. Harrison, U. Furbach, and N. Shankar, editors, *International Joint Conference on Automated Reasoning 2006*, page 15 pages, Seattle, USA, August 2006. Springer.

[4] G. Sutcliffe and C. Suttner. The State of CASC. *AI Communications*, 19(1):35–48, 2006.

[5] The Coq Development Team. *The Coq Proof Assistant Reference Manual, version 8.1*, 2006. `http://coq.inria.fr/`.

[6] J.R. Fisher and M.A. Bezem. Skolem Machines. *Fundamenta Informatica*, 91(1):79–103, 2009.

[7] J.A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

A New Look at Generalized Rewriting in Type Theory

Matthieu Sozeau

Harvard University
mattam@eecs.harvard.edu

Abstract. Rewriting is an essential tool for computer-based reasoning, both automated and assisted. It is so because rewriting is a general notion that permits to model a wide range of problems and provides means to effectively solve them. In a proof assistant, rewriting can be used to replace terms in arbitrary contexts, generalizing the usual equational reasoning to reasoning modulo arbitrary relations. This can be done provided the necessary proofs that functions appearing in goals are congruent with respect to specific relations. We present a new implementation of generalized rewriting in the Coq proof assistant, making essential use of the expressive power of dependent types and a recently implemented type class mechanism. The tactic improves on and generalizes previous versions by supporting natively higher-order functions, polymorphism and subrelations. The type class system inspired from Haskell provides a perfect interface between the user and such tactics, making them easily extensible.

1 Introduction

By generalized rewriting we mean the ability to replace a subterm t of an expression by another term t' when they are related by a relation R. When the relation is Leibniz equality, this reduces to standard equational reasoning which is allowed in essentially any context[1]. However, when the relation is different, the contexts in which a replacement can occur are restricted and one needs to prove a compatibility lemma about the specific context involved to show that the replacement is valid. Luckily, one can prove that rewriting in a context is allowed compositionally by combining compatibility lemmas for each constant.

By using an automatic tactic to find this proof, we can completely forget this boring part of reasoning and use generalized rewriting to support seemingly **extensional** reasoning with formulas, setoids and any other user-defined relations like enriched logical operators (e.g. separation logic connectives as in [1]).

We will first introduce and review related work in this area (§2) before presenting our new system (§3) and analysing it (§4), finally concluding in §5.

2 Related Work

Generalized rewriting is a notion that appears in many forms in the literature. For example, it is at the core of "window inferencing" systems like the one described in [2], that permits to prove goals by refinement steps each of which being an application of a lemma of the form $t\ \mathcal{R}\ t'$ or by recursively opening "windows", new subgoals that refine a given subterm of the current goal. We review the approaches to integrate this idea in type theory.

[1] Except if the rewrite involves capture of binders in an intensional type theory like Coq

LCF The idea of combining rewriting tactics appears in the Boyer-Moore and LCF systems, in particular in Lawrence Paulson's work [3] in Cambdrige LCF. He designs a set of so-called "conversions", higher-order rewriting tactics that can be used to implement custom rewriting strategies. He first focuses on the primitive rewriting tools of the system, β-reduction and Leibniz equality; then shows how to extend the technique to logical formulae using logical equivalence as the rewriting relation. In this system, it's still the user who combines the tactics himself to create a strategy.

NuPRL The step further was to automatically infer the combination of proofs needed to show that a rewriting is allowed, given compatibility lemmas on the constants involved. This was done by Basin [4] in NuPRL who also generalizes on the relations involved. He supposes given a set of lemmas showing the compatibility of operators with respect to some relations and combines them automatically to build the appropriate proof term when the user tries a rewrite step. In this setting, it is possible to give multiple *signatures* to a single constant, for example addition can be given the signatures:

$$+ : \{x = x' \to y = y' \to x + y = x' + y'\} \quad + : \{x < x' \to y \leq y' \to x + y < x' + y'\}$$
$$+ : \{x \leq x' \to y < y' \to x + y < x' + y'\} \quad + : \{x \leq x' \to y = y' \to x + y \leq x' + y'\}$$
$$+ : \{x = x' \to y \leq y' \to x + y \leq x' + y'\}$$

The first declares that addition is congruent for equality (actually, all objects are) and the later show that it is monotone for the various combinations of $=$, $<$ and $\leq$ on its arguments. The algorithm must sometimes choose one of these proofs during proof search, as the output relation is generally not known in advance, and the obvious combinatorial explosion in this setting led the author to find a heuristic for this choice and implement a partial search. This heuristic is based on user-provided information on the subrelation property of these relations: $=$ and $<$ are incomparable here, and both are stronger (smaller) that $\leq$. Choosing the strongest signature gives the best experimental results, hence choosing one of the first three signatures over the last three (implication is covariant for strongness on the right).

The implementation is based again on a set of tactics that are composed on-the-fly to produce a deterministic rewriting step that makes the goal progress.

Coq Finally, the `setoid_rewrite` tactic developed by Claudio Sacerdoti Coen [5] in Coq (after an initial implementation by Samuel Boutin already improved upon by Clément Renard) is slightly different. It differs from Basin's approach in a number of ways:

- The tactic is *complete*: instead of using a heuristic when multiple signatures can be selected, the algorithm tries all possibilities. The rationale for this choice is that goals are not deep enough in general to warrant a more efficient implementation that avoids the exponential factor. The tactic does not support subrelations hence it could not use Basin's heuristic.
- The tactic is *semi-reflexive*, which means it is separated in two parts, one meta part (written in ML) that builds a trace for the rewrite using a database of user lemmas and another part (in Coq) which proves a general theorem showing that any trace gives rise to a correct rewrite. The trace consists of the applied user lemmas along with information on variance.
- The tactic supports *variance* natively for asymmetric relations. Signatures are written point-free (without explicit mention of the objects) from the algebra (deeply embedded, as an inductive definition) of terms for atomic relations and the combinators $++\!\!>$, $\longrightarrow$, $\Longrightarrow$ for respectively covariant, contravariant and equivariant relations on arrow types. Symmetry is treated natively

(arguably for simplicity and user-friendliness) when using the contravariant and equivariant arrows, as each signature defines an opposite signature which has the same set of associated morphisms and one need to write only one of them. In comparison, our implementation does not treat the equivariant arrow but supports the automatic inference of opposite signatures (§3.4).
– The tactic also supports *non-reflexive* relations, generating subgoals for reflexivity on unchanged arguments when needed.

Sacerdoti Coen [5] indicates some possible optimizations on the proof search algorithm which is entangled with the recursive search for rewrites. However, in practice, it is not sufficient to speed up the trace creation process when the goal is very deep. This system was also somewhat limited due to the deep embedding in supporting polymorphic or dependent relations and functions.

Rewriting with Leibniz equality Finally, our work can be compared with the existing support for rewriting with the native equality of the system, for which every construction is congruent (except when capturing binders). That is, the standard rewrite works in contexts involving let-binders, pattern-matching or fixpoints and it allows substitution when type dependencies are involved, none of which is handled here. The current setup of the `rewrite` tactic is to use the standard rewrite when rewriting with a Leibniz equality, and use generalized rewriting for other relations.

3 A new tactic for Generalized Rewriting

We will present a new, generalized implementation of generalized rewriting in Coq that blends better with the system, directly supporting polymorphism, higher-order functions and rewriting under binders. Our algorithm is a mix of Basin's and Sacerdoti Coen's work.

We will split the problem in two parts to get a clearer view on the whole system: a constraint generation procedure (in ML) and a customizable proof search that is also at the meta-level (in $\mathcal{L}_{\text{tac}}$), based on type classes [6]. This simplification follows a current trend in the design of proof search algorithms (e.g. for type inference [7]) to make them more modular: it allows the study and more practically the independent modification of each part.

The resulting system allows to experiment more efficient proof-search strategies and supports all the previously-mentioned features, some of which are implemented solely using the extensibility capabilities of type classes. The tactic uses a set of general-purpose definitions on relations that we will present now.

3.1 Relations

We will begin by defining a number of standard concepts around relations and introduce a few useful type classes. We first introduce combinators on relations. We recall that in Coq a homogeneous binary relation R on a given type A is represented as a function of type $A \to A \to \texttt{Prop}$ into the propositions. The inverse (or converse) relation, noted R^{-1}, is easily obtained by flipping the order of arguments using the flip combinator:

```
Definition inverse {A} (R : relation A) : relation A := flip R.
Notation " R ^-1 " := (inverse R) (at level 2) : relation_scope.
```

The complementary relation is classicaly defined as a negation:

```
Definition complement {A} (R : relation A) : relation A := λ x y, R x y → False.
```

We can define the pointwise extension of a relation on B to $A \to B$:

```
Definition pointwise_relation {A B} (R : relation B) : relation (A → B) :=
  λ f g, ∀ x : A, R (f x) (g x).
```

We use the combinator all to represent universal quantification as a constant application.

```
Definition all {A} (P : A → Prop) : Prop := ∀ x : A, P x.
```

Properties We also introduce type classes that formalize the usual notions of reflexivity, symmetry, transitivity and their duals.

```
Class Reflexive {A} (R : relation A) := reflexivity : ∀ x, R x x.
Class Irreflexive {A} (R : relation A) := irreflexivity :> Reflexive (complement R).
Class Symmetric {A} (R : relation A) := symmetry : ∀ {x y}, R x y → R⁻¹ x y.
Class Transitive {A} (R : relation A) := transitivity : ∀ {x y z}, R x y → R y z → R x z.
```

These class declarations introduce overloaded methods that can be used to refer to arbitrary reflexivity, symmetry or transitivity proofs afterwards. Note that these classes are all indexed by a type and a value, making essential use of dependent types. See [6] for an introduction to type classes.

Standard Instances We can already populate the instance database with easy proofs by duality. All these properties are preserved by inversion, for example:

```
Instance flip_Reflexive '(Reflexive A R) : Reflexive R⁻¹ := reflexivity (R := R).
```

Finaly we define some instances for the standard logical operators. We use the Program extension [8] to define these instances. In this mode, each undefined field is turned into an obligation that is automatically proved using a default tactic including `firstorder` reasoning which is enough in this case. Implication impl = `fun` $A\ B$: `Prop` $\Rightarrow A \to B$ is reflexive and transitive:

```
Program Instance impl_Reflexive : Reflexive impl.
Program Instance impl_Transitive : Transitive impl.
```

Both logical equivalence (defined as double implication and denoted by iff or $\leftrightarrow$) and Leibniz equality form equivalences, so they both have **Reflexive**, **Symmetric** and **Transitive** instances.

Subrelations The last interesting concept we introduce is that of subrelations: the inclusion order on relations. We make it a class so that we can declare logic clauses to dynamically prove it on given relations.

```
Class subrelation {A : Type} (R R' : relation A) : Prop :=
  is_subrelation : Π x y, R x y → R' x y.
```

An essential property of the **subrelation** relation is its reflexivity:

```
Instance subrelation_refl : @subrelation A R R.
```

We declare the two following subrelation instances by default:

```
Instance iff_impl_subrelation : subrelation iff impl.
Instance iff_inverse_impl_subrelation : subrelation iff (inverse impl).
```

3.2 Signatures and morphisms

Contrary to Sacerdoti Coen's tactic, we chose a shallow embedding of signatures in the dependent type theory. This has the disadvantage that we cannot write algorithms on the signatures in Coq itself but we can always do so using the $\mathcal{L}_{\text{tac}}$ system. This choice also makes sense because the unification procedure that is needed later when trying to find constants having a given signature cannot be deeply embedded easily, nor is it really desirable for efficiency. Another obvious advantage is that one can design and support new constructions in signatures easily.

Morphisms The central notion of the tactic is that of being a morphism for a given relation R. We say that an object m is a morphism for a relation R when $R\ m\ m$, that is m is in the kernel of R, or m is a **Proper** element of R, using PER terminology[2]. Note that this definition is very general and not in any way specialized for functions, i.e. objects of arrow types which we will speak of as morphisms instead of proper elements, following the terminology used in previous work.

```
Class Proper {A} (R : relation A) (m : A) : Prop := proper : R m m.
```

We make this notion a class, hence users can easily add new **Proper** instances to the type class database. We make **Proper**'s type an implicit argument as it can always be infered from the signature R or the object itself.

Clearly, any element in a type accompanied by a reflexive relation is a proper element for it. We basically add a new logic clause for the **Proper** $R\ x$ theorem saying it is enough to find a proof of **Reflexive** $A\ R$ to solve it.

```
Instance reflexive_proper '(Reflexive A R) (x : A) : Proper R x.
```

Signatures We declare a new parsing scope for relations seen as signatures so that the notations we use later can be given other meanings in different contexts.

```
Delimit Scope signature_scope with signature.
Open Local Scope signature_scope.
```

Another essential notion is the signature for objects with arrow types. We define a single compatibility arrow as a parametric extensionality relation on arrow types for two given relations on the input and output type.

```
Definition respectful {A B : Type} (R : relation A) (R' : relation B) : relation (A → B) :=
  λ f g, ∀ x y, R x y → R' (f x) (g y).
```

Naturally, a function f respects respectful $R\ R'$ if for any two objects related by R the outputs of f applied to those are related by R'. The respectful definition gives a relational version of respect, which can be applied to two different functions, which will eventually be instantiated by the same object in a proper statement. As this is a shallow embedding, we won't be able to match on applications of respectful in Coq but we will be able to do so *via* tactics in $\mathcal{L}_{\text{tac}}$.

We obtain the other combinators simply using a set of notations. The respect arrows associate to the right following the arrow type. The notation ++> for covariance is the naked respectful definition, $R \longrightarrow R'$ is an abbreviation for R^{-1} ++> R'. The equivariant arrow $\Longrightarrow$ is currently an alias for the covariant arrow.

```
Notation " R ++> R' " := (respectful R R') (right associativity, at level 55) : signature_scope.
```

[2] The standard library of Coq 8.2 uses **Morphism** instead of **Proper**

`Notation` " R ⟶ R' " := (respectful R^{-1} R') (*right associativity*, `at` *level* 55) : *signature_scope*.

We can start declaring **Proper** instances using these notations for usual operators like logical negation not : Prop → Prop.

`Program Instance` contraposed_morphism : **Proper** (impl ⟶ impl) not.
`Program Instance` not_iff_morphism : **Proper** (iff ++> iff) not.

Unfolding the definitions of respectful, inverse and **Proper**, we find that the goals boil down to usual compatibility lemmas, e.g. for the first: $\forall\ x\ y$, impl $y\ x \to$ impl $(\neg\ x)\ (\neg\ y)$

It is also possible to declare parametric instances, which act like Horn clauses in logic programming. Here we assert that every transitive relation is itself a morphism:

`Program Instance` trans_morphism `(`**Transitive** $A\ R$) : **Proper** $(R \longrightarrow R$ ++> impl$)\ R$.

The signature indicates that for every transitive relation R we have $R\ x'\ x \to R\ y\ y' \to R\ x\ y \to R\ x'\ y'$. Using this morphism instance we will be able to rewrite with any transitive relation. Before going into a description of the set of **Proper** instances used to rewrite with the standard operators, we will present the algorithm that generates these **Proper** constraints.

3.3 Constraint Generation

The ML algorithm is in charge of finding the subterm to be replaced and generating a proof skeleton and a set of constraints. Once these are solved (if possible), we will simply substitute the proofs inside the skeleton to get a complete proof that the rewrite is valid.

We will present the algorithm as a set of inference rules from which we will derive a syntax-directed variant in a standard way. The algorithm is parameterized by a rewriting lemma ρ of type $\forall \overrightarrow{\phi}, R\ \overrightarrow{\alpha}\ t\ u$, i.e. a type whose ultimate codomain is an applied binary relation (note that any variable in $R\ \overrightarrow{\alpha}\ t\ u$ can be bound in $\overrightarrow{\phi}$ here). The typing context Γ, represents the set of local hypotheses and the global context, it grows when we go under abstractions. The set of constraints builds up incrementally in each rule, so there are both input and output sets denoted with ψ which contain constraints of the form $?_x : \tau$ declaring hypothetical objects of a given potentially incomplete type τ.

The rewriting judgment $\Gamma \mid \psi \vdash \tau \rightsquigarrow^R_p \tau' \dashv \psi'$ defined in figure 1 means that in environment Γ, ψ, τ is rewritten to τ' with respect to relation R with p a proof of type $R\ \tau\ \tau'$ in context Γ, ψ'.

Initially, given a goal $\Gamma \vdash \tau$ and a rewrite lemma ρ we will want to find a judgment of the form

$$\Gamma \mid \emptyset \vdash \tau \rightsquigarrow^{\mathsf{impl}^{-1}}_{\_} \_ \dashv \_$$

Given a proof of such a rewrite from τ to some τ', that is a proof of τ impl^{-1} τ' we can apply it to the goal to progress to $\Gamma \vdash \tau'$. Dually, we use impl as the top relation when trying to rewrite in a hypothesis and specialize it with the resulting proof. *N.B.: We supposed that relations, hypotheses and goals were always in* `Prop`*, but the construction works just as well in* `Type`*, with computational relations.*

Rules The inference rules suppose as given a function **type**(Γ,ψ,t) which returns the type of a given term in a context. All our terms are well-typed so these calls can never fail. The unification function for a given lemma ρ $\mathbf{unify}_\rho(\Gamma, \psi, \tau)$ takes as input typing and constraint environments and a type and tries to unify the left-hand-side of the lemma's applied relation with the type. It

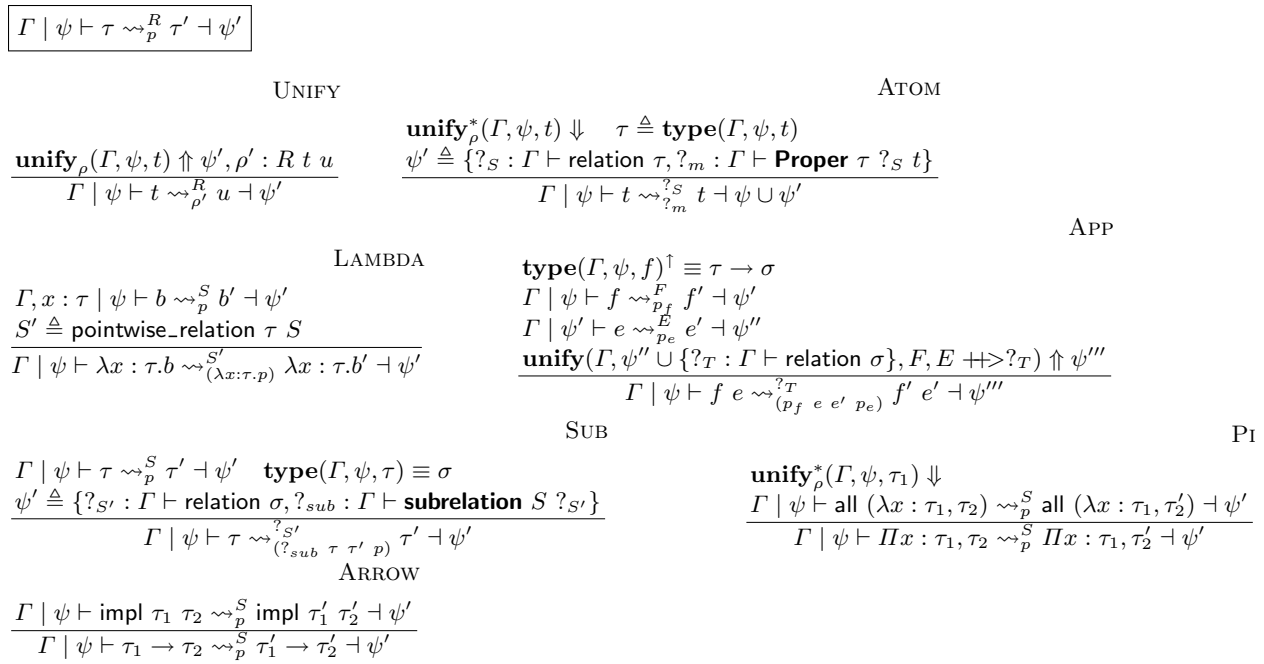


Fig. 1. Rewriting Constraint Generation - declarative version

may fail ($\Downarrow$) or succeed ($\Uparrow$), generating new constraints for the uninstantiated lemma arguments and an instantiated lemma ρ' whose type must be of the form $R\ t\ u$ for some R, u. The variant $\mathbf{unify}^*_\rho(\Gamma, \psi, \tau)$ tries unification on all subterms and succeeds if at least one unification does. The function $\mathbf{unify}(\Gamma, \psi, t, u)$ does a standard unification of t and u.

Let us now describe each rule:

- **Unify** The unification rule fires when the toplevel term unifies with the lemma. It directly uses the generated proof for the rewrite from t to some u with respect to some R.
- **Atom** This rule applies only when no other rule can apply and no rewrite can happen in the term. It asserts that the term must remain unchanged for some arbitrary relation $?_S$ during the rewrite, which is witnessed by a **Proper** proof. Typically, these constraints are either generated for unmodified arguments of a function and the **Proper** proof is solved by a reflexivity proof for the appropriate relation or they are generated for a function itself and the constraints get instantiated by user-provided proofs.
- **Lambda** To rewrite under an abstraction we simply rewrite the body inside the enriched context. The resulting proof can be extended pointwise to a closed proof in the original context by simply enclosing it in a λ.
- **App** We rewrite under an application by rewriting successively in the function and the argument. Here we assert that the rewrite relation for the function must unify with $E \mathrel{+\!\!+\!\!>} ?_T$ for some new relation $?_T$ to ensure that the constraint on the argument corresponds to the expected relation for the function argument. The resulting proof is a combination of the respectful proof for the function and the proof found for the argument which builds related results in $?_T$.
- **Sub** We add a subsumption rule to the system which allows to assign multiple relations to a single rewrite. The **subrelation** type class models a lattice of relations ordered by inclusion. It

allows for example to see that **subrelation** (pointwise_relation τ S) (eq_τ ++> S) and use a rewrite under an abstraction as a premise of the App rule.
- **Pi** This rule is an administrative step to rewrite inside the codomain of a dependent product, knowing that we can't rewrite in its domain. It translates the product into an application of the combinator all whose **Proper** instances will be presented later.
- **Arrow** The rewrite can happen in the domain only in non-dependent products. In this case we use the impl combinator instead.

Algorithm We must now derive an algorithm from this declarative specification of the system. To do so, we must eliminate the subsumption rule which is not syntax directed. The side conditions of the other rules are sufficient to ensure determinism otherwise. As the **subrelation** class is user-driven, we will only make assumptions on the associated set of instances. First, the relation must be transitive to be able to compress a stack of Sub applications into a single one. It must also be closed under pointwise_relation to go through the Lambda rule. The last problem is with App as it forces the relation on the function to be of a particular shape: we must simply change the rule to integrate Sub in the first premise:

$$\frac{\begin{array}{l}\Gamma \mid \psi \vdash f \rightsquigarrow^{F}_{p_f} f' \dashv \psi' \quad \mathbf{type}(\Gamma, \psi, f)^{\uparrow} \equiv \tau \to \sigma \\ \Gamma \mid \psi' \vdash e \rightsquigarrow^{E}_{p_e} e' \dashv \psi'' \\ \psi''' \triangleq \{?_T : \Gamma \vdash \mathsf{relation}\ \sigma, ?_{sub} : \Gamma \vdash \textbf{subrelation}\ F\ (E \mathrel{++>} ?_T)\}\end{array}}{\Gamma \mid \psi \vdash f\ e \rightsquigarrow^{?_T}_{?_{sub}\ f\ f'\ p_f\ e\ e'\ p_e} f'\ e' \dashv \psi'' \cup \psi'''} \quad \textsc{AppSub}$$

Note that we explicitly take the head-normal form of the function's type to be able to generate the constraints and we assume that this arrow is not dependent.

With these changes, we can directly extract an algorithm $\mathbf{rew}(\Gamma, \rho, \tau)$ directed by the type τ, which always succeeds and returns a tuple (ψ, R, τ', p) with the output constraints, a relation R, a new term τ' and a proof $p : R\ \tau\ \tau'$. In case no rewrite happens, we will just have an application of Atom. Obviously, we can decorate the actual algorithm to count the number of successful unifications and fail if nothing was rewritten. We can use this to stop at the first rewrite too.

We now have the skeleton of a proof with holes and just need to solve the constraints to complete the proof. We assume here that we can mark constraints in the constraint set to indicate if they come from the unification of the lemma or as part of the algorithm itself. We will solve only the latter and leave the former for the user to prove.

3.4 Resolution

The proof-search problems generated by the **rew** algorithm are sets of constraints of the form **Proper** A $(R_1 \mathrel{++>} \cdots \mathrel{++>} R_n)$ m or **subrelation** A R_1 R_2, where A, m are closed terms but the R_i's are open. The existential variables appearing in them may of course be shared across multiple constraints, notably because of the App rule.

The R_i's may actually be arbitrary relations, and we want to be able to support some particular signature constructions automatically, notably the inverse combinator. We also need to actually implement a satisfying **subrelation** relation and support other features like higher-order morphisms and partial applications. To handle all these, we will write logic clauses that allow to prove **Proper** and **subrelation** as class instances. We will also extend the proof-search algorithm using a few $\mathcal{L}_{\text{tac}}$ tactics to handle more complex resolution steps. All of this is defined in a standard Coq script that we present now.

As seen before, we can declare generic morphisms for standard polymorphic combinators that preserve compatibility, e.g. for flip:

```
Program Instance flip_proper '(mor : Proper (A → B → C) (RA ++> RB ++> RC) f) :
  Proper (RB ++> RA ++> RC) (flip f).
```

For higher-order morphisms, we must show how applications to pointwise equivalent functional arguments are related. For example, to show that existential quantification lifts logical equivalence we have to prove:

```
Instance ex_iff_mor : Proper (pointwise_relation A iff ++> iff) (@ex A).
```

The statement of this proof unfolds to: $(\forall\ x,\ P\ x \leftrightarrow Q\ x) \rightarrow (\exists\ x,\ P\ x) \leftrightarrow (\exists\ x,\ Q\ x)$. Using this instance we can now rewrite under existentials with the equivalence relation, e.g:

```
Goal Π A P Q, (∀ x : A, P x ↔ Q x) → (∃ x, ¬ P x) → (∃ x, ¬ Q x).
Proof. intros A P Q H HnP. setoid_rewrite ← H. assumption. Qed.
```

We won't detail here the various proper declarations for standard operators and classes like **PER**, **Equivalence**, etc... they can be found in the standard library modules. We will just detail the specific ones that allow to support some interesting features.

Partial applications During constraint generation we build signatures for applications starting at the first rewritten argument as we generate invariance constraints for the largest invariant subterms. This allows us to support parametric morphisms very easily as we are generally not interested in rewriting in the first few type parameters. However, this interacts with non-parametric morphisms as well. Suppose we have $P \rightarrow Q$ and rewrite with $H : Q \leftrightarrow Q'$. The generated constraint will be of the form **Proper** (iff ++> inverse impl) (impl P). However, we generally declare our morphisms for complete applications, e.g.: **Proper** (iff ++> iff ++> inverse impl) impl. Hence we need a way to derive the former from the latter. It suffices to declare the following instance whose application will generate a metavariable for the unknown relation on the argument.

```
Instance partial '(Proper (A → B) (R ++> R') m) '(Proper A R x) : Proper R' (m x) | 4.
```

We give a low priority to this instance so that it won't be used except if no other **Proper** is declared on $m\ x$.

Subrelations We have seen that logical equivalence is smaller than implication or its inverse. This means that any morphism that ends with iff can be viewed as a morphism producing impl-related arguments or its dual. We can actually make this even clearer by proving the **subrelation** instance for respect arrows. Classicaly, the arrow is contravariant for the subrelation relation on the left and covariant on the right:

```
Instance respectful_subrelation '(subrelation A R2 R1, subrelation B S1 S2) :
  subrelation (R1 ++> S1) (R2 ++> S2).
```

We mentionned previously that for the constraint generation algorithm to be sound and complete with respect to the declarative presentation, **subrelation** had to be transitive. We will not add anything like a generic recursive transitive subrelation instance as that would render proof-search useless: it would always loop if we tried to search for an invalid **subrelation** constraint. Instead transitivity should be proved for the specific set of instances that are declared at some point. We

can assure that the set of instances declared in the library is transitive: no two rules could form the premises of a non-trivial use of transitivity.

We also mentionned that pointwise_relation had to be congruent for **subrelation**. Indeed it is a covariant morphism for it:

```
Instance pointwise_sub : Proper (subrelation ++> subrelation) (@pointwise_relation A B).
Instance subrelation_pointwise A `(sub : subrelation B R R') :
  subrelation (pointwise_relation A R) (pointwise_relation A R').
```

These instances allow us to bootstrap the system in a natural way: we can now rewrite inside signatures and under pointwise_relation, by showing that respectful is a morphism for **subrelation** as well. We can prove compatibility with **subrelation** and also relation_equivalence (defined as double inclusion of the relations and denoted by $=_{\mathcal{R}}$) of many of the combinators we have seen like inverse, complement and even **Proper** itself:

```
Instance morphism_proper A : Proper (relation_equivalence ++> @eq _ ++> iff) (@Proper A).
```

It follows that if we can find a **Proper** instance for m using signature R_2 and a subrelation R_1 of R_2 then m is also a proper element of it: this is exactly what is internalized by the Sub rule. However, we will not directly integrate the rule as it should only be applied once at the top of a search.

```
Lemma subrelation_proper `(Proper A R1 m, subrelation A R1 R2) : Proper R2 m.
```

Indeed, this lemma is way too general to introduce it to the **Proper** instance search: it could be applied endlessly. Instead, we construct a tactic that restricts its use to the top of the goal when some flag **apply_subrelation** is set.

```
CoInductive apply_subrelation : Prop := do_subrelation.
Hint Extern 5 (@Proper _ _ _) ⇒ match goal with [ H : apply_subrelation ⊢ _ ] ⇒
  clear H ; class_apply @subrelation_proper end : typeclass_instances.
```

We add this tactic to the instance database to apply it when the goal is a **Proper**. Thanks to this control, we can do all the logic programming we want inside Coq using $\mathcal{L}_{\text{tac}}$ and let the user customize the proof search in the same way.

Dual Morphisms Finally, we can construct a tactic to handle the signatures involving inverse in the same way. First, we observe that a term m is a **Proper** element for a relation R^{-1} if and only if it is for R.

```
Program Instance proper_inverse `(Proper A R m) : Proper R^-1 m.
```

The goal is to make it possible for the user to declare a morphism for R only and automatically infer that it is also a morphism for R^{-1} or any relation equivalent to it with respect to the equational theory generated by:

```
Lemma inverse_invol A (R : relation A) : R^-1-1 =_R R.
Lemma inverse_arrow A (R : relation A) B (R' : relation B) : (R ++> R')^-1 =_R R^-1 ++> R'^-1.
```

Of course R^{-1} might not be in any kind of normal form: we want to push the inverse relation as far as possible inside the signature. Our strategy is to expand every part of a signature to applications of inverse and add **subrelation** instances to relate all the signatures in the produced equivalence class. We introduce a new class to normalize signatures, resolution will be based on the first one (m).

`Class` **Normalizes** $\{A : \texttt{Type}\}$ $(m\ m' : \mathsf{relation}\ A) : \texttt{Prop} := \mathsf{normalizes} : m =_{\mathcal{R}} m'^{-1}$.

Our strategy works by adding inverse everywhere in the signatures, going through arrows.

`Lemma` norm1 $A\ R$: @**Normalizes** $A\ R$ (inverse R).
`Lemma` norm2 '(**Normalizes** $A\ R_0\ R_1$, **Normalizes** $B\ U_0\ U_1$) : **Normalizes** $(R_0 \text{ ++> } U_0)\ (R_1 \text{ ++> } U_1)$.

We implement the strategy by a tactic that figures out if we have an arrow or an atomic type at the head and applies the appropriate lemma. Once we have resolved the inverse signature we can use **subrelation** to prove that the signature is related to the one declared by the user.

4 Analysis

4.1 Quantitative analysis

The constraint generation algorithm is clearly linear in the size of the rewritten term, so it has got a minor influence on the performance of the whole tactic. On the other hand, the proof search strategy is a (bounded) depth-first search using the instance database whose performance tends to be close to linear in the size of the constraints, when no backtracking is needed. We must always take care that the instances don't loop, that is why we control precisely the application of some lemmas.

The tactic has much better performance than the one by [5] on deep goals, as depth-first search often allows to prove goals directly without much backtracking. In practice the tactic gives immediate responses even on large goals. It should be noted that this tactic unlike the former only returns the first solution of the constraints. We leave the generalization of the search procedure for future work. Regarding the proof term size it is of the order of the rewritten term plus the proof terms for the constraints which are again generally linear in the size of their type.

4.2 Implementation & experiments

The tactic presented here is already available as part of Coq 8.2 where it replaces the previous one [9]. The implementation has been tested on the standard library of Coq as well as all the user contributions of Coq (`http://coq.inria.fr/contribs-eng.html`) which contains large projects using setoids such as CoRN and CoLoR. It is not clear whether the performance gains on these later examples come from the new `setoid_rewrite` implementation or some other improvement over previous versions of Coq but the standard library's times on setoid-intensive files have dropped significantly (-30%). Also, some other developments that could not be handled previously clearly benefit from the improved performance, e.g. the one done by Benton and Tabareau [1] which provided the impetus to reimplement the whole tactic.

To speed up proof search of instances, we use an enhanced discrimination net that can handle existentials contrary to the one already used in the `eauto` tactic of Coq. We also added a dependency analysis between subgoals to perform so-called green cuts in the search tree when two subgoals become independent (i.e. do not share existential variables).

4.3 Refinements

The tactic extends the previous one by supporting the `at` option which allows to select which occurences of the lemma should be rewritten, in a left-to-right traversal of the term. It should be

noted that the semantic of the tactic is different from the standard `rewrite`'s in that it tries to unify the lemma with each subterm independently and in its local context instead of doing a single unification and rewritting all subterms that match the resulting instantiated lemma. Typically our semantics allows to rewrite with a general lemma and select deep occurences in the goal without having to mention the term, e.g. consider:

`Goal` $\forall\ x\ y\ z : \mathbf{nat}, (x + y) + z = y + (x + z)$.

If we want to rewrite with the commutativity lemma for addition, we get four different possible instantiations that can be selected with `at`. This new semantic allows finer-grain control over occurences but it is also essential to be able to rewrite under binders, where unification can capture variables introduced inside subterms. Let's consider the following goal:

`Goal` $\forall\ H : (\forall\ x, x \times 1 = x), \exists\ x, x \times 1 \neq 0$.

To rewrite under the existential quantifier, we must apply H to x itself, hence do unification in the local context.

5 Conclusion

We have presented a new tactic for generalized rewriting in Coq, based on a constraint generation algorithm generating type class constraints to be solved by a generic but customizable instance search. The tactic extends previous ones in a number of directions, providing support for arbitrary polymorphic relations and morphisms, subrelations, automatic dualization of signatures and rewriting under binders. The new architecture allows for far greater extensibility *via* $\mathcal{L}_{\text{tac}}$ and for finer grain control on performance through its modular implementation. Finally, the choice of a shallow embedding and use of type classes allows easy integration inside user developments.

Acknowledgments I thank anonymous referees and Gregory Malecha for their remarks. I thank Nicolas Tabareau for helping in the design of the dualization instances, glueing to the previous interface and feedback and Arnaud Spiwack for initial discussions that led to the definitions of **Proper** and respectful.

References

1. Benton, N., Tabareau, N.: Compiling Functional Types to Relational Specifications for Low Level Imperative Code. In: TLDI. (2009)
2. Robinson, P.J., Staples, J.: Formalizing a Hierarchical Structure of Practical Mathematical Reasoning. Journal of Logic and Computation **3** (1993) 47–61
3. Paulson, L.C.: A Higher-Order Implementation of Rewriting. Science of Computer Programming **3** (1983) 119–149 (or 119–150??)
4. Basin, D.A.: Generalized Rewriting in Type Theory. Elektronische Informationsverarbeitung und Kybernetik **30** (1994) 249–259
5. Sacerdoti Coen, C.: A Semi-reflexive Tactic for (Sub-)Equational Reasoning. In Filliâtre, J.C., Paulin-Mohring, C., Werner, B., eds.: TYPES. Volume 3839 of Lecture Notes in Computer Science., Springer (2004) 98–114
6. Sozeau, M., Oury, N.: First-Class Type Classes. In Otmane Ait Mohamed, C.M., Tahar, S., eds.: Theorem Proving in Higher Order Logics, 21th International Conference. Volume 5170 of Lecture Notes in Computer Science., Springer (2008) 278–293
7. Pottier, F.: A versatile constraint-based type inference system. Nordic Journal of Computing **7** (2000) 312–347
8. Sozeau, M.: Un environnement pour la programmation avec types dépendants. PhD thesis, Université Paris 11, Orsay, France (2008)
9. Sozeau, M.: User defined equalities and relations. In: Coq 8.2 Reference Manual. INRIA TypiCal (2008)

Descente Infinie Proofs in Coq

Răzvan Voicu Mengran Li
Department of Computer Science
National University of Singapore
{razvan,limengra}@comp.nus.edu.sg

Abstract

Descente infinie is a type of inductive reasoning that allows an instance of a goal that occurs in a goal-directed (backward reasoning) proof process to assume the role of a hypothesis later in the proof, without the explicit application of an induction principle or rule. This works only if the instance used as hypothesis is smaller than the original, with respect to a well-founded order. Unlike explicit induction, which is the inductive technique of choice in most mainstream proof assistants, including Coq, proofs by *descente infinie* do not require guessing (or even creating) an adequate induction scheme at the beginning of the proof process and are thus more intuitive, and more amenable to automation.

In this paper, we argue that proofs by *descente infinie* are straightforward to implement in Coq, and thus easier to produce than their explicit induction counterparts. In addition, we present our ongoing attempt at building an automated inductive tactic, and discuss some preliminary experiments.

1 Introduction

In automated theorem proving, the term *induction* denotes a set of proof techniques that allow the use of *not yet certified* information as a means of validating a potentially infinite set of formulas. In fact, according to [5, 6], the research community is divided into the two schools of *explicit* and *implicit* induction, of which the former represents the established mainstream, which excels in the most powerful theorem provers today, including Coq [3]. Although there is no generally accepted characterization of the two paradigms, the *descente infinie* induction principle typically falls in the implicit induction category, and is surveyed in [14, 15].

Explicit induction realizes the familiar idea of inductive theorem proving using induction axioms. In accordance with this view, one reason to call this paradigm *"explicit"* is that in the underlying inference systems every cyclic argument must be made explicit in a single inference step by applying a so-called *induction principle*. Apart from generating *base cases*, this inference step joins induction hypotheses and conclusions in *induction step* formulas. Moreover, it explicitly guarantees the termination of the cyclic argument by a sub-proof of the well-foundedness of the induction ordering resulting from the step formulas.

In contrast, *implicit induction* represents a type of inductive reasoning whereby no induction principle is explicitly applied. The survey [15] differentiates between *proof by consistency* [8], based on the Knuth-Bendix completion procedure, and *descente infinie* (also called *lazy induction* in [12]), which is an induction principle discovered by the ancient Greeks, and reinvented by Fermat in the 17^{th} century. Intuitively, the *descente infinie* principle follows the reasoning patterns of the working mathematician in realizing an inductive argument. The experienced mathematician usually starts with

a conjecture, which he simplifies by case analysis, and applications of axioms and rules available in the theory at hand. When he realizes that the current goal has become similar to an instance of the conjecture, he applies the instantiated conjecture just like a lemma, but keeps in mind that he has actually applied an induction hypothesis. Finally, he searches for some well-founded ordering in which all instances of the conjecture he has applied as an induction hypothesis are smaller than the original conjecture itself. The advantage of the *descente infinie* principle, as compared to explicit induction, is that the mathematician is not required to commit to an induction principle or scheme, at a point in the proof process when the success of such a decision is uncertain.

To illustrate the difference between explicit induction and *descente infinie*, let us consider the simple (and probably ubiquitous) example of proving that $\forall n : nat, n + 0 = n$. In an explicit induction proof, we first choose an induction scheme, of the many induction schemes available for natural numbers. In this simple case, it is obvious that the one we need to pick is $\forall P : nat \rightarrow Prop, P0 \rightarrow (\forall n, Pn \rightarrow P(Sn)) \rightarrow \forall n, Pn$. However, in more complicated proofs, the correct choice of an induction scheme may not be equally obvious at this stage in the proof process. Now, by instantiating P with the conjecture at hand, we derive the following two proof obligations: $0 + 0 = 0$, and $\forall n, n + 0 = n \rightarrow Sn + 0 = Sn$. Both these proof obligations are now easily discharged using the definition of addition for natural numbers and the injectivity of the successor constructor.

In contrast, a proof by *descente infinie* usually starts with performing case analysis on the conjecture, without giving any consideration to any inductive reasoning that may be required later. Since n may be either 0, or the successor of some other number n_1, case analysis yields the following two proof obligations: $0 + 0 = 0$, which is immediately discharged, and $Sn_1 + 0 = Sn_1$, which can be further simplified, using the definition of addition, into $n_1 + 0 = n_1$. The current proof obligation is now recognized as an instance of the original conjecture and, since n_1 is a subterm of n, and thus smaller with respect to a well-founded order, it can be concluded at this point, by the principle of *descente infinie*, that the original conjecture holds. We note that in this entire proof process we did not have to make an early commitment to an induction principle whose success was uncertain. This important property contributes to simpler inductive proofs, and makes a descent-infinie-based tactic be a valuable addition to an interactive proof assistant.

At this point, the experienced proof assistant user would probably argue, quite legitimately, that the validity of the *descente infinie* principle needs to be established first. In Coq, however, this result comes almost for free. Let us remember that the proof process consists of transitioning through a sequence (or rather a tree) of subgoals, each transition being the result of applying a tactic. Moreover, each subgoal has a corresponding partial proof term which becomes further instantiated with each transition. Whenever we encounter a repeating subgoal, by inspecting its proof term, we can identify the sub-term that was "filled in" by the sequence of tactics that were applied between the two repetitions. As we will show later in the paper, this subterm can be transformed into a *recursive function definition* that would reconcile the two repeating goals, making the first occurrence of the repeating goal act as an induction hypothesis for the second occurrence. Importantly, the well-foundedness check is carried out automatically by Coq's typechecker. Thus, in Coq, *descente infinie* becomes a *method of manipulating proof terms into recursive definitions that reconcile repeating goals*, rather than a set of axioms or inference rules.

We are aware that induction automation is an ample task, which has been investigated extensively with mixed results (for a detailed overview, cf. [14]). However, for the time being, our objective is rather modest: automatically solve simple inductive goals faster than they would be solved by hand. In that respect, our paper makes the following contributions:

- A methodology of inductive reasoning in Coq, that does not require early commitment to an

induction principle, and a discussion on the challenges of implementing such a methodology.

- A naïve automation algorithm for simple inductive proofs.

As related work, we mention [1, 2, 13] which discuss the use of *descente-infinie*-based methods for automated theorem proving, and [9, 11], which apply a similar principle to co-inductive reasoning. The soundness of the *descente infinie* principle is presented in [4].

The rest of the paper is organized as follows. Section 2 discusses by means of example two ways of implementing *descente infinie* reasoning in Coq, and argues that this type of reasoning may lead to shorter, more goal-oriented, and more expressive proofs in certain cases. Section 3 discusses various aspects of the use of tactics in connection with *descente infinie* reasoning. Sections 4 and 5 present a simple automation algorithm, and the corresponding experimental results. Section 6 concludes.

2 Illustrative Examples

In this section, we argue by means of examples, that proofs by *descente infinie* are in fact more intuitive and straightforward to produce as compared to their explicit induction counterparts.

2.1 A Proof Repair Scenario

First, let us consider a proof repair scenario. Consider the following attempt to produce a proof of the `le_Sn_le` theorem that is available in the `Arith` library:

```
Coq < Lemma le_Sn_le: forall n m, S n <= m -> n <= m. (* original goal *)
1 subgoal

  ============================
   forall n m : nat, S n <= m -> n <= m

le_Sn_le < intros n m H. (* first instance of repeating goal *)
1 subgoal

  n : nat
  m : nat
  H : S n <= m
  ============================
   n <= m

le_Sn_le < destruct H ; constructor ; [constructor|].
                          (* second instance of repeating goal *)
1 subgoal

  n : nat
  m : nat
  H : S n <= m
  ============================
   n <= m
```

In this attempt, we aim to model the proof search process of the working mathematician[1], which would typically start with performing case analysis on the conjecture to be proven, followed by simplifications that would either discharge the goal at hand, or would produce a goal similar to some

[1] We assume that the working mathematician has just formulated the $\leq$ relation and is in the process of exploring its properties; he has not yet discovered its transitivity.

goal encountered before in the proof process. This is indeed the case in the proof attempt given above. After the case analysis and simplification performed by the sequence of tactics `destruct H ; constructor ; [constructor|]`, the resulting goal is identical to its predecessor. At this point, we notice that hypothesis `H` in the last subgoal is in fact a *subterm* of hypothesis `H` of the second subgoal, and thus *smaller* w.r.t. an adequately defined well-founded order. Hence, the repeating goal signals the potential of applying an induction principle and completing the proof. This, however, cannot be accomplished directly in Coq.

A less direct approach would be to *repair* the proof. Let us first examine the current proof term.

```
le_Sn_le < Show Proof.
LOC:
Subgoals
1 -> forall n m : nat, S n <= m -> n <= m
Proof: fun (n m : nat) (H : S n <= m) =>
       match H in (_ <= n0) return (n <= n0) with
       | le_n => le_S n n (le_n n)
       | le_S m0 H0 => le_S n m0 (?1 n m0 H0)
       end
```

The fact that the proof had a repeating goal actually means that the type of the entire `match...end` expression must be the same as the type of the expression `(?1 n m0 H0)`, embedded in the second arm of the `match`. This leads immediately to the idea of changing the proof term from a function definition to a definition of a *recursive function*, whose name can then be used as the value of the `?1` meta-variable. Thus, we can produce a new proof term by cutting and pasting the current proof term, and then performing the following small changes: (a) transform the function heading so as to reflect a recursive function definition, and (b) replace the `?1` meta-variable with the recursive function name. The new proof term would solve the original goal completely, as shown below.

```
le_Sn_le < Restart.
1 subgoal

  ============================
   forall n m : nat, S n <= m -> n <= m

le_Sn_le < exact (
  fix le_Sn_le (n m : nat) (H : S n <= m) {struct H} :=
       match H in (_ <= n0) return (n <= n0) with
       | le_n => le_S n n (le_n n)
       | le_S m0 H0 => le_S n m0 (le_Sn_le n m0 H0)
       end).
Proof completed.
```

This proof repairing trick implements in effect a proof by *descente infinie*. It is the user's responsibility to identify the decreasing argument in the {`struct ...`} declaration. Nevertheless, Coq is capable of verifying automatically that the recursive application is well-founded, and as a result, it accepts the proof when the user types the command `Save.`

A more elegant approach can be produced if the user suspects that a proof by *descente infinie* might occur later. In such a case, the conclusion of the current goal may be "saved" as a hypothesis using a Coq command of the following form:

```
le_Sn_le < refine (fix le_Sn_le n m (H:S n <= m) {struct H} := _).
1 subgoal

  le_Sn_le : forall n m : nat, S n <= m -> n <= m
  n : nat
  m : nat
  H : S n <= m
  ============================
   n <= m
```

The original conclusion is now available as the hypothesis `le_Sn_le`[2]. Then, the proof proceeds as usual, with case analysis and simplification, and when the repeating goal is encountered, the "saved" hypothesis can be readily applied to solve the goal. Obviously, the proof can only be saved if the application of the saved hypothesis is well-founded. Coq 8.2 provides even more support for such proofs, in the form of the `fix` tactic, and the `Guarded` command, leading to the following implementation of a proof by *descente infinie*.

```
Coq < Lemma le_Sn_le: forall n m, S n <= m -> n <= m.
1 subgoal

  ============================
   forall n m : nat, S n <= m -> n <= m

le_Sn_le < fix 3; intros n m H;
           destruct H as [|n0 H]; constructor ;
           [constructor|apply le_Sn_le;exact H].
Proof completed.

le_Sn_le < Guarded.
The condition holds up to here
```

In the above proof, `fix 3` "saves" the current conclusion as a hypothesis, by creating a partial proof term in the form of a recursive definition whose third argument (later identified as `H` due to the `intros` tactic) is expected to decrease. The `Guarded` command verifies that the application of the saved hypothesis is well-founded, a condition that could only be checked by attempting to save the proof in previous versions of Coq.

At this point, there are two aspects we would like to emphasize. Firstly, while the use of the `fix` tactic makes the implementation of proofs by *descente infinie* more elegant, it still has the major drawback that it requires advance knowledge of the repeating goal, and its decreasing subterm. Nevertheless, as the next two examples illustrate, even under such circumstances, proofs by *descente infinie* would still be easier to produce, since they do not require the existence of an adequate *induction principle*. Secondly, the "proof-repair" trick used in our first proof attempt is easily automatable, and can be performed systematically by a procedure that could be used as a basis for an automated induction tactic.

2.2 Goal-Directedness

The example above, while simple enough to showcase the relationship between a proof attempt with a repeating goal, and its corresponding partial proof term, it is also simple enough to be solved by the explicit induction tactic available in Coq. In contrast, the next two examples showcase goals

[2]Still, the user must remember to apply this hypothesis *only* to a subterm of `H`.

whose *descente infinie*-based proofs are simpler, and arguably, more goal directed than their explicit induction-based counterparts.

First, consider the following inductive predicate that holds if its argument is an even number.

```
Inductive even : nat -> Prop :=
| e0 : even 0
| eS : forall n, even n -> even (S (S n)).
```

Consider now proving the goal:

$$\forall n : \texttt{nat},\ \texttt{even}\, n \vee \neg\, \texttt{even}\, n.$$

Again, we use `fix 1` to "save" the conclusion as an inductive hypothesis. Then, we perform case analysis and simplification.

```
even_or_not_even < fix 1; intros [|n1] ;
                   [ left; constructor |
                     destruct n1 as [|n2] ;
                       [ right ;
                         intro H ; inversion H | ] ].
1 subgoal

  even_or_not_even : forall n : nat, even n \/ ~ even n
  n2 : nat
  ============================
   even (S (S n2)) \/ ~ even (S (S n2))
```

At this point, it is time to use the induction hypothesis.

```
even_or_not_even < destruct (even_or_not_even n2).
2 subgoals

  even_or_not_even : forall n : nat, even n \/ ~ even n
  n : nat
  H : even n2
  ============================
   even (S (S n2)) \/ ~ even (S (S n2))

subgoal 2 is:
 even (S (S n2)) \/ ~ even (S (S n2))

even_or_not_even < Guarded.
The condition holds up to here
```

We note that the current variable `n2` is a subterm of variable `n` that appeared in the first subgoal, and thus the use of the induction hypothesis is well-founded. This condition is verified with the command `Guarded`. From this point, the proof can be completed by case analysis and simplification, using the sequence of tactics

```
left;constructor;exact H. right;intro H0;apply H;inversion H0;trivial.
```

Note that the proof is, to a great extent, *goal directed*, in the sense that at each step, the most obvious choice of a tactic is made.

Now, let us consider an explicit induction proof for the same goal. One way to achieve this is to create a new induction scheme, materialized in the following theorem:

$$\forall P : nat \to Prop,\ P\,0 \to P\,1 \to (\forall n,\ P\,n \to P\,(S\,(S\,n))) \to (\forall n,\ P\,n).$$

And another way to achieve the same objective is to prove the following auxiliary lemma first:

$$\forall n, (\neg \texttt{even}\,(S\,n) \to \texttt{even}\,n) \wedge (\neg \texttt{even}\,n \to \texttt{even}\,(S\,n)).$$

Both explicit induction solutions require advance knowledge of how the goal will turn out after performing all the simplifications and case analysis steps, in order to either select an adequate induction scheme, or invent an adequate lemma. This knowledge cannot be derived solely from the information available at the point when the knowledge is required. Thus, the user will have to try several proof paths to acquire this knowledge, and then undo certain proof steps to return to the proof state where the knowledge is applicable. In this sense, the explicit induction proof is not goal-directed, and in fact, a lot more cumbersome than its *descente infinie* counterpart.

For the next example, let us consider two more inductive definitions.

```
Inductive m3: nat -> Prop :=
| m30 : m3 0
| m3S : forall n, m3 n -> m3 (S (S (S n))).

Inductive m6: nat -> Prop :=
| m60 : m6 0
| m6S : forall n, m6 n -> m6 (S (S (S (S (S (S n)))))).
```

Yet again, let us consider a conjecture that is not provable by explicit induction in a goal-directed fashion.

$$\forall n : \texttt{nat}, \texttt{even}\,n \to \texttt{m3}\,n \to \texttt{m6}\,n.$$

Similar to the previous example, we proceed by case analysis and simplification, choosing at each step the most obvious tactic for the current goal. This is simulated by the application of the following compound tactic.

```
multiples < fix 3 ; intros n H1 H2; destruct H2 as [|n H2];
            [  constructor
             | destruct H2 as [|n H2];
               [  inversion_clear H1 as [|? H]; inversion H
                | constructor; inversion_clear H1 as [|? H];
                  inversion_clear H as [|? H1];
                  inversion_clear H1 ] ].
1 subgoal

  multiples : forall n : nat, even n -> m3 n -> m6 n
  n : nat
  H2 : m3 n
  H : even n
  ============================
   m6 n
```

At this point, the induction hypothesis applies, and the tactic `apply multiples; trivial` solves the goal.

As in the previous example, an explicit induction proof is more difficult to achieve. A relatively simple solution is to use the induction principle implemented by the `lt_wf_ind` theorem available in the `Omega` library. However, the commitment to this induction principle must be made early on in the proof process, well before the success of this approach becomes obvious.

3 A *Descente Infinie* Tactic

The central step of our proof method is a proof term transformation procedure that introduces a recursive function capable of creating a well-founded induction hypothesis. We shall describe this step by taking a closer look at the `even_or_not_even` example that has already been discussed in the previous section. However, unlike in the previous attempt to prove this goal, we *no longer save the current conjecture into an induction hypothesis* at the beginning of the proof. Instead, the induction hypothesis will be derived *lazily*, when needed, from the current partial proof term.

Consider the interaction with the Coq system given in Figure 1, which mimics the state of our automated proof process after applying a series of case analysis and inversion steps (the algorithm applies these steps non-deterministically, and at this point it is only important to understand that this is a *possible* state in the proof process). To make the text easier to follow, we have augmented the code in the figure with Coq-style comments, enclosed in `(*···*)`.

Let us analyze the current state of the proof process. The current proof term contains the meta-variable `?1` (on the third line from the bottom) that corresponds to the current sub-goal. We note that this variable occurs inside two nested `match` constructs. The crucial property of a `match` construct is that it isolates a *subterm* of a given term, and thus creates an opportunity for a well-founded application of an induction hypothesis. However, an induction hypothesis must exist first. This can be achieved by a proof transformation whose result is given in Figure 2. The `match` construct is wrapped inside a recursive definition, called `circ` in the figure. The recursive argument of `circ` has the same name as the matched variable, and the well-founded application of the induction hypothesis can now be produced by applying `circ` to the subterm of the matched variable. This expression must now be offered as an argument to `?1`.

Our *descente infinie* (`DI`) tactic takes a numeric argument `k`, called the *depth*, and applies this proof term transformation procedure to the `k`*th* closest `match` construct. When the depth is `1`, the tactic applies the transformation to the innermost `match` and produces the proof term and subgoal given in Figure 2 (again, the interaction mimics the result).

Our tactic produces a transformed proof term. To illustrate its effect, we restart the proof process, and submit the transformed proof term via the `refine` tactic. For the sake of brevity, in the code above we have omitted the proof subterm corresponding to the case when `n1` is 0, since it is not changed by this transformation.

Let us take a closer look at the differences produced by this transformation in the new proof term. We notice that the innermost `match` is now wrapped inside a recursive definition named `circ`. The argument to this definition is `n1`, identical to the matched variable. This makes `n2` a subterm of `n1`, and thus the recursive call `(circ n2)` becomes a well-founded induction hypothesis. This expression is added, with the name `IH` as an argument to the meta-variable `_` (underscore) representing the current subgoal. To "close the loop", the entire recursive definition of `circ` is applied to the outer variable `n1`, which in effect restores the matched expression of the innermost `match` construct. In the resulting subgoal, `IH` is now available and ready to be applied.

While the `IH` tactic is the product of a sound procedure, it is in fact not the one suitable for solving the conjecture. To produce the needed induction hypothesis, we need to apply the transformation procedure to the outermost `match`, by invoking the `DI` tactic with a depth of `2`. This will result in the subgoal:

```
n2 : nat
IH : even n2 \/ ~ even n2
============================
 even (S (S n2)) \/ ~ even (S (S n2))
```

Figure 1 A Coq Interaction

```
Coq < Lemma even_or_not_even: forall n, even n \/ ~ even n.
1 subgoal

  ============================
   forall n : nat, even n \/ ~ even n

....... (* steps omitted *)

  n2 : nat
  ============================          (* current subgoal *)
   even (S (S n2)) \/ ~ even (S (S n2))

even_or_not_even < Show Proof. (* prints the current proof term *)
LOC:
Subgoals
1 -> forall n2 : nat, even (S (S n2)) \/ ~ even (S (S n2))
Proof: fun n : nat =>
       match (* further from the goal, DI tactic arg := 2 *)
         n as n0
         return (even n0 \/ ~ even n0)
       with
       | 0 => or_introl (~ even 0) e0
       | S n1 =>
           match (* closest to the goal, DI tactic arg := 1 *)
             n1 as n01
             return (even (S n01) \/ ~ even (S n01))
           with
           | 0 =>
               or_intror (even 1)
                 (fun H : even 1 =>
                  match
                    H in (even k)
                    return
                      match k with
                      | 0 => True
                      | 1 => False
                      | S (S _) => True
                      end
                  with
                  | e0 => I
                  | eS _ _ => I
                  end)
           | S n2 => (fun _ : nat => ?1 n2) n2
           end                    (* ^^ Metavariable for current subgoal *)
       end
```

which can be easily discharged by case analysis and simplification, as already seen in the previous section.

This systematic partial proof term transformation is implemented in OCaml, and currently statically linked in the Coq source tree.

Figure 2 Simulation of DI Tactic

```
even_or_not_even < Restart.
1 subgoal

  ============================
   forall n : nat, even n \/ ~ even n

even_or_not_even < refine (  (* result of proof transformation *)
  fun n : nat =>
       match n as n0 return (even n0 \/ ~ even n0) with
       | 0 => or_introl (~ even 0) e0
       | S n1 =>
         (fix circ n1 := (* recursive definition wrapper *)
           match n1 as n01 return (even (S n01) \/ ~ even (S n01)) with
           | 0 => (* omitted, identical to corresp. subproof in Fig 1 *)
           | S n2 => (fun n2 IH => _) n2 (circ n2)
           end) n1                    (* ^^^^^^^ induction hypothesis *)
       end). (* ^^ restore original argument of match *)
1 subgoal

  n2 : nat
  IH : even (S n2) \/ ~ even (S n2)
  ============================
   even (S (S n2)) \/ ~ even (S (S n2))

even_or_not_even < Guarded.
The condition holds up to here
```

4 Towards an Automated Induction Tactic

A simple automation idea would be to implement the `DI` procedure as a standalone tactic, and then add it as an external hint to a hint database, so that it can be used in conjunction with `auto`. However, this is not possible, since (at least to our knowledge), the Coq system does not allow changing the partial proof term on the fly. Nevertheless, our current automated tactic, implemented in Ocaml, uses a similar approach. Non-deterministic proof steps are specified via a set of rules, and the tactic employs an exhaustive search process of limited depth, which backtracks over all the choices of case analysis, simplification, rewriting, generalization, and generation of induction hypothesis at all possible nesting depths. For each new subgoal, precedence is given to "cheaper" tactics, that is, tactics that have a higher chance of solving the goal at hand in fewer steps. One such situation is when a repeating subgoal is detected (i.e. a subgoal that is similar to one encountered before), in which case the proof term transformation procedure creates an induction hypothesis that can be immediately applied to solve the goal.

5 Experiments

In its current stage of development, our tactic implements the repeated goal detection and the proof transformation step described in Section 3. However, it only performs a restricted form of rewriting, and no generalization. Table 1 lists several of the interesting inductive lemmas that can be proven by the tactic, together with the search depth that is needed to discover the proof, and the time taken. The experiment was conducted on a 2.4GHz linux machine.

Figures 3 and 4 show two of the proof terms produced by our tactic. We note that in both cases,

Lemma	depth	time(seconds)
$\forall n\,m\,p, n \leq m \rightarrow m \leq p \rightarrow n \leq p$	4	3.192
$\forall n, 0 \leq n$	3	0.007
$\forall n\,m, n \leq m \rightarrow Sn \leq Sm$	3	0.2
$\forall n\,m, Sn \leq m \rightarrow n \leq m$	3	0.108
$\forall n\,m, Sn \leq Sm \rightarrow n \leq m$	4	0.384
$\forall n\,m, n < Sm \rightarrow n \leq m$	4	0.395
$\forall n\,m, n \leq m \rightarrow n < Sm$	3	0.12
$\forall n\,m, n < m \rightarrow Sn < Sm$	3	0.147
$\forall n\,m, Sn < Sm \rightarrow n < m$	4	0.418
$\forall n, 0 < Sn$	3	0.008
$\forall n\,m\,p, n < m \rightarrow m < p \rightarrow n < p$	4	2.688
$\forall n\,m\,p, n < m \rightarrow m \leq p \rightarrow n < p$	4	2.688
$\forall n\,m\,p, n \leq m \rightarrow m < p \rightarrow n < p$	4	2.688
$\forall n\,m, n \leq m \rightarrow n < m \vee n = m$	5	4.132
$\forall n, n = n + 0$	4	2.133
$\forall n\,m, S(n+m) = n + Sm$	4	2.212
$\forall l : listA, l = l \texttt{ ++ } nil$	4	2.542
$\forall l\,m\,n : listA, (l \texttt{ ++ } m) \texttt{ ++ } n = l \texttt{ ++ } m \texttt{ ++ } n$	4	3.391
$\forall l\,l' : listA, length\,(l \texttt{ ++ } l') = length\,l + length\,l'$	4	3.422

Table 1: Performance of automated induction tactic

the recursive function `circ` (for *circular*) is produced by transforming a partial proof term when a repeating goal is detected.

Figure 3 Proof term of $\forall$ *l1 l2* : *list A*, *samelength l1 l2* $\rightarrow$ *length l1* = *length l2*

```
Inductive samelength : (list A)->(list A)->Prop :=
| sl0 : sl (nil:list A) (nil:list A)
| sl1 : forall (a b:A) (l1 l2:list A), sl l1 l2 ->
         sl (a::l1) (b::l2).

fix circ (l1 l2 : list A) (H : sl l1 l2) {struct H} :
  length l1 = length l2 :=
  match H in (sl l l0) return (length l = length l0) with
  | sl0 => refl_equal (length nil)
  | sl1 _ _ l3 l4 H0 =>
      let H1 := circ l3 l4 H0 in
      trans_eq (f_equal (fun f : nat -> nat => f (length l3))
                (refl_equal S)) (f_equal S H1)
  end
```

6 Conclusion and Further Work

We have presented a methodology for realizing *descente infinie* proofs in Coq, and an implementation of this methodology in the form of an automated induction tactic. The tactic is capable of solving simple goals that would otherwise not be solvable by any of the existing automated tactic. In the current implementation, our use of generalization and rewriting techniques is rather rudimentary. In future work, we plan to explore the use of rippling [7] to guide rewriting, and lemma discovery methods [10].

Figure 4 Proof term of $\forall l\, l' : listA, length\,(l \texttt{ ++ } l') = length\, l + length\, l'$

```
fix circ (l l' : list A) {struct l} :
  length (l ++ l') = length l + length l' :=
  match
    l as l0 return (length (l0 ++ l') = length l0 + length l')
  with
  | nil => refl_equal (length nil + length l')
  | _ :: l0 =>
      let H := circ l0 l' in
      (fun H0 : length (l0 ++ l') = length l0 + length l' =>
       trans_eq
         (f_equal (fun f : nat -> nat => f (length (l0 ++ l')))
            (refl_equal S)) (f_equal S H0)) H
  end
```

References

[1] Alessandro Armando, Michaël Rusinowitch, and Sorin Stratulat. Incorporating decision procedures in implicit induction. *Journal of Symbolic Computation*, 34(4):241–258, October 2002.

[2] G. Barthe and S. Stratulat. Validation of the javacard platform with implicit induction techniques. In R. Nieuwenhuis, editor, *Rewriting Techniques and Applications, 14th International Conference, RTA-03*, LNCS 2706, pages 337–351, Valencia, Spain, June 9–11, 2003. Springer.

[3] Yves Bertot and P. (Pierre) Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Texts in theoretical computer science. Springer-Verlag, pub-SV:adr, 2004.

[4] James Brotherston and Alex Simpson. Complete sequent calculi for induction and infinite descent. In *LICS*, pages 51–62, 2007.

[5] A. Bundy. The automation of proof by mathematical induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 13, pages 845–911. Elsevier Science, 2001.

[6] Alan Bundy. A survey of automated deduction. In *Artificial Intelligence Today*, pages 153–174. 1999.

[7] Alan Bundy, David Basin, Dieter Hutter, and Andrew Ireland. *Rippling: Meta-Level Guidance for Mathematical Reasoning*, volume 56 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, June 2005.

[8] Hubert Comon. Inductionless induction. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 14, pages 913–962. Elsevier Science Publishers, 2001.

[9] Joxan Jaffar, Andrew E. Santosa, and Razvan Voicu. A coinduction rule for entailment of recursively defined properties. In Peter J. Stuckey, editor, *CP*, volume 5202 of *Lecture Notes in Computer Science*, pages 493–508. Springer, 2008.

[10] D. Kapur and M. Subramaniam. Lemma discovery in automating induction. *Lecture Notes in Computer Science*, 1104:538–??, 1996.

[11] Dorel Lucanu and Grigore Rosu. CIRC: A circular coinductive prover. In Till Mossakowski, Ugo Montanari, and Magne Haveraaen, editors, *CALCO*, volume 4624 of *Lecture Notes in Computer Science*, pages 372–378. Springer, 2007.

[12] Martin Protzen. Lazy generation of induction hypotheses. In Alan Bundy, editor, *12th International Conference on Automated Deduction*, LNAI 814, pages 42–56, Nancy, France, June 26–July 1, 1994. Springer-Verlag.

[13] Sorin Stratulat. Automatic 'descente infinie' induction reasoning. In Bernhard Beckert, editor, *TABLEAUX*, volume 3702 of *Lecture Notes in Computer Science*, pages 262–276. Springer, 2005.

[14] Claus-Peter Wirth. History and future of implicit and inductionless induction: Beware the old jade and the zombie! In Dieter Hutter and Werner Stephan, editors, *Mechanizing Mathematical Reasoning*, volume 2605 of *Lecture Notes in Computer Science*, pages 192–203. Springer, 2005.

[15] Claus-Peter Wirth. A self-contained and easily accessible discussion of the method of descente infinie and fermat's only explicitly known proof by descente infinie. *CoRR*, abs/0902.3623, 2009. informal publication.