**Noname manuscript No.**
(will be inserted by the editor)

# Hammer for Coq

## Automation for Dependent Type Theory

**Łukasz Czajka · Cezary Kaliszyk**

**Abstract** Hammers provide most powerful general purpose automation for proof assistants based on HOL and set theory today. Despite the gaining popularity of the more advanced versions of type theory, such as those based on the Calculus of Inductive Constructions, the construction of hammers for such foundations has been hindered so far by the lack of translation and reconstruction components.

In this paper, we present an architecture of a full hammer for dependent type theory together with its implementation for the Coq proof assistant. A key component of the hammer is a proposed translation from the Calculus of Inductive Constructions, with certain extensions introduced by Coq, to untyped first-order logic. The translation is "sufficiently" sound and complete to be of practical use for automated theorem provers. We also introduce a proof reconstruction mechanism based on an `eauto`-type algorithm combined with limited rewriting, congruence closure and some forward reasoning. The algorithm is able to re-prove in the Coq logic most of the theorems established by the ATPs.

Together with machine-learning based selection of relevant premises this constitutes a full hammer system. The performance of the whole procedure is evaluated in a bootstrapping scenario emulating the development of the Coq standard library. For each theorem in the library only the previous theorems and proofs can be used. We show that 40.8% of the theorems can be proved in a push-button mode in about 40 seconds of real time on a 8-CPU system.

### 1 Introduction

*Interactive Theorem Proving* (ITP) systems [HUW14] become more important in certifying mathematical proofs and properties of software and hardware. A large part of the process of proof formalisation consists of providing justifications for smaller goals. Many of such goals would be considered trivial by mathematicians. Still, modern ITPs require users to spend an important part of the formalisation effort on such easy goals. The main points that constitute this effort are usually library search, minor transformations on the already proved theorems (such as reordering

E-mail: {lukasz.czajka,cezary.kaliszyk}@uibk.ac.at
University of Innsbruck, Austria

assumptions or reasoning modulo associativity-commutativity), as well as combining a small number of simple known lemmas.

ITP automation techniques are able to reduce this effort significantly. Automation techniques are most developed for systems that are based on somewhat simple logics, such as those based on first-order logic, higher-order logic, or the untyped foundations of ACL2. The strongest general purpose proof assistant automation technique is today provided by tools called "hammers" [BKPU16] which combine learning from previous proofs with translation of the problems to the logics of automated systems and reconstruction of the successfully found proofs. For many higher-order logic developments a third of the proofs can be proved by a hammer in push-button mode [BGK+16,KU14].

Even if the more advanced versions of type theory, as implemented by systems such as Agda [BDN09], Coq [Ber08], Lean [dMKA+15], and Matita [ARS14], are gaining popularity, there have been no hammers for such systems. This is because building such a tool requires a usable encoding, and a strong enough proof reconstruction.

A typical use of a hammer is to prove relatively simple goals using available lemmas. The problem is to find appropriate lemmas in a large collection of all accessible lemmas and combine them to prove the goal. An example of a goal solvable by our hammer, but not solvable by any standard Coq tactics, is the following.

```
forall (A : Type) (l1 l2 : list A) (x y1 y2 y3 : A),
  In x l1 \/ In x l2 \/ x = y1 \/ In x (y2 :: y3 :: nil) ->
  In x (y1 :: (l1 ++ (y2 :: (l2 ++ (y3 :: nil)))))
```

The statement asserts that if `x` occurs in one of the lists `l1`, `l2`, or it is equal to `y1`, or it occurs in the list `y2 :: y3 :: nil` consisting of the elements `y2` and `y3`, then it occurs in the list

```
y1 :: (l1 ++ (y2 :: (l2 ++ (y3 :: nil))))
```

where `++` denotes list concatenation and `::` denotes the list cons operator. Eprover almost instantly finds a proof of this goal using six lemmas from the module `Lists.List` in the Coq standard library:

```
Lemma in_nil : forall (A : Type) (a : A), ~(In a nil).
Lemma in_inv : forall (A : Type) (a b : A) (l : list A),
               In b (a :: l) -> a = b \/ In b l.
Lemma in_cons : forall (A : Type) (a b : A) (l : list A),
                In b l -> In b (a :: l).
Lemma in_or_app : forall (A : Type) (l m : list A) (a : A),
                  In a l \/ In a m -> In a (l ++ m).
Lemma app_comm_cons : forall (A : Type) (x y : list A) (a : A),
                      a :: (x ++ y) = (a :: x) ++ y.
Lemma in_eq : forall (A : Type) (a : A) (l : list A), In a (a :: l).
```

The found ATP proof may be automatically reconstructed inside Coq.

The advantage of a hammer is that it is a general system not depending on any domain-specific knowledge. The hammer plugin may use all currently accessible lemmas, including those proven earlier in a given formalization, not only the lemmas from the standard library or other predefined libraries.

**Contributions.** In this paper we present a comprehensive hammer for the Calculus of Inductive Constructions together with an implementation for the Coq proof assistant. In particular:

- We introduce an encoding of the Calculus of Inductive Constructions, including the additional logical constructions introduced by the Coq system, in untyped first-order logic with equality.
- We implement the translation and evaluate it experimentally on the standard library of the Coq proof assistant showing that the encoding is sufficient for a hammer system for Coq: the success rates are comparable to those demonstrated by hammer systems for Isabelle/HOL and Mizar, while the dependencies used in the ATP proofs are most often sufficient to prove the original theorems.
- We present a proof reconstruction mechanism based on an `eauto`-type procedure combined with some forward reasoning, congruence closure and heuristic rewriting. Using this proof search procedure we are able to re-prove 44.5% of the problems in the Coq standard library, using the dependencies extracted from the ATP output.
- The three components are integrated in a plugin that offers a Coq automation tactic `hammer`. We show case studies how the tactic can help simplify certain existing Coq proofs and prove some lemmas not provable by standard tactics available in Coq.

Preliminary versions of the translation and reconstruction components for a hammer for Coq have been presented by us at HaTT 2016 [CK16]. Here, we improve both, as well as introduce the other required components creating a first whole hammer for a system based on the Calculus of Inductive Constructions.

The rest of this paper is structured as follows. In Section 2 we discuss existing hammers for other foundations, as well as existing automation techniques for variants of type theory including the Calculus of Constructions. In Section 3 we introduce $\mathrm{CIC}_0$, an approximation of the Calculus of Inductive Constructions which will serve as the intermediate representation for our translation. Section 4 discusses the adaptation of premise selection to $\mathrm{CIC}_0$. The two main contribution follow: the translation to untyped first-order logic (Section 5) and a mechanism for reconstructing in Coq the proofs found by the untyped first-order ATPs 6. The construction of the whole hammer and its evaluation is given in Section 7. Finally in Section 8 a number of case studies of the whole hammer is presented.

## 2 Related Work

A recent overview [BKPU16] discusses the three most developed hammer systems, large-theory premise selection, and the history of bridges between ITP and ATP systems. Here we briefly survey the architectures of the three existing hammers and their success rates on the various considered corpora, as well as discuss other related automation techniques for systems based on the Calculus of (Inductive) Constructions.

### 2.1 Existing Hammers

Hammers are proof assistant tools that employ external automated theorem provers (ATPs) in order to automatically find proofs of user given conjectures. Most developed hammers exist for proof assistants based on higher-order logic (Sledgehammer [PB10] for Isabelle/HOL [WPN08], HOLyHammer [KU14] for HOL Light [Har09]

and HOL4 [SN08]) or dependently typed set theory (MizAR [KU15c] for Mizar [Wie07, BBG+15]). Less complete tools have been evaluated for ACL2 [JKU14]. There are three main components of such hammer systems: premise selection, proof translation, and reconstruction.

Premise Selection is a module that given a user goal and a large fact library, predicts a smaller set of facts likely useful to prove that goal. It uses the statements and the proofs of the facts for this purpose. Heuristics that use recursive similarity include SInE [HV11] and the Meng-Paulson relevance filter [MP09], while the machine-learning based algorithms include sparse naive Bayes [Urb04] and $k$-nearest neighbours (k-NN) [KU13b]. More powerful machine learning algorithms perform significantly better on small benchmarks [ACI+16], but are today too slow to be of practical use in ITPs [FK15, KvLT+12].

Translation (encoding) of the user given conjecture together with the selected lemmas to the logics and input formats of automated theorem provers (ATPs) is the focus of the second module. The target is usually first-order logic (FOL) in the TPTP format [Sut10], as the majority of the most efficient ATPs today support this foundation and format. Translations have been developed separately for the different logics of the ITPs. An overview of the HOL translation used in Sledgehammer is given in [Bla12]. An overview of the dependently-typed set theory of MizAR is given in [US10]. The automated systems are in turn used to either find an ATP proof or just further narrow down the subset of lemmas to precisely those that are necessary in the proof (unsatisfiable core).

Finally, information obtained by the successful ATP runs can be used to re-prove the facts in the richer logic of the proof assistants. This is typically done in one of the following three ways. First, by a translation of the found ATP proof to the corresponding ITP proof script [PS07, BBF+15], where in some cases the script may be even simplified to a single automated tactic parametrised by the used premises. Second, by replaying the inference inside the proof assistant [BW10, KU13a, PS07]. Third, by implementing verified ATPs [AFG+11], usually with the help of code reflection.

The general-purpose automation provided by the most advanced hammers is able to solve 40–50% of the top-level goals in various developments [BKPU16], as well as more than 70% of the user-visible subgoals [BGK+16].

### 2.2 Related Automation Techniques

The encodings of the logics of proof assistants based on the Calculus of Constructions and its extensions in first-order logic have so far covered only very limited fragments of the source logic [ACN05, TS98, BHdN02]. Why3 [FP13] provides a translation from its own logic [Fil13] (which is a subset of the Coq logic, including features like rank-1 polymorphism, algebraic data types, recursive functions and inductive predicates) to the format of various first-order provers (in fact Why3 has been initially used as a translation back-end for HOLyHammer).

Certain other components of a hammer have already been explored for Coq. For premise selection, we have evaluated the quality of machine learning advice [KMU14] using custom implementations of Naive Bayes relevance filter, k-Nearest Neighbours, and syntactic similarity based on the Meng-Paulson algorithm [MP09]. *Coq Learning Tools* [Lau16] provides a user interface extension that suggests to the user lemmas

that are most likely useful in the current proof using the above algorithms as well as LDA. The suggestions of tactics which are likely to work for a given goal has been attempted in ML4PG [KHG13], where the Coq Proof General [Asp00] user interface has been linked with the machine learning framework Weka [HFH+09]. SEPIA [GWR15] tries to infer automata based on existing proofs that are able to propose likely tactic sequences.

The already available HOL automation has been able to reconstruct the majority of the automatically found proofs using either internal proof search [Hur03] or source-level reconstruction. The internal proof search mechanisms provided in Coq, such as the `firstorder` tactic [Cor03], have been insufficient for this purpose so far: we will show this and discuss the proof search procedures of `firstorder` and `tauto` in section 6. The `jp` tactic which integrates the intuitionistic first-order automated theorem prover JProver [SLKN01] into Coq does not achieve sufficient reconstruction rates either [CK16]. Matita's ordered paramodulation [AT07] is able to reconstruct many goals with up to two or three premises, and the congruence-closure based internal automation techniques in Lean [dMS16] are also promising.

The SMTCoq [AFG+11] project has developed an approach to use external SAT and SMT solvers and verify their proof witnesses. Small checkers are implemented using reflection for parts of the SAT and SMT proof reconstruction, such as one for CNF computation and one for congruence closure. The procedure is able to handle Coq goals in the subset of the logic that corresponds to the logics of the input systems.

## 3 Type Theory Preliminaries

In this section we present our approximation $\mathrm{CIC}_0$ of the Calculus of Inductive Constructions, i.e., of the logic of Coq. The system $\mathrm{CIC}_0$ will be used as an intermediate step in the translation, as well as the level at which premise selection is performed. Note that $\mathrm{CIC}_0$ is interesting as an intermediate step in the translation, but is not a sound type theory by itself (this will be discussed in Section 5.6). We assume the reader to be familiar with the Calculus of Constructions [CH88] and to have a working understanding of the type system of Coq [BC04, Coq16]. This section is intended to fix notation and to precisely define the syntax of the formalism we translate to first-order logic. The system $\mathrm{CIC}_0$ is intended as a precise description of the syntax of our intermediate representation. It is a substantial fragment of the logic of Coq as presented in [Coq16, Chapter 4], as well as of other systems based on the Calculus of Constructions. The features of Coq not represented in the formalism of $\mathrm{CIC}_0$ are: modules and functors, coinductive types, primitive record projections, and universe constraints on Type.

The formalism of $\mathrm{CIC}_0$ could be used as an export target for other proof assistants based on the Calculus of Inductive Constructions, e.g. for Matita or Lean. However, in $\mathrm{CIC}_0$, like in Coq, Matita and Lean, there is an explicit distinction between the universe of propositions Prop and the universe of sets Set or types Type. The efficiency of our translation depends on this distinction: propositions are translated directly to first-order formulas, while sets or types are represented by first-order terms. For proof assistants based on dependent type theories which do not make this distinction, e.g. Agda [BDN09] and Idris [Bra13], one would need a method to heuristically infer which types are to be regarded as propositions, in addition to possibly some adjustments to the formalism of $\mathrm{CIC}_0$.

The language of $\mathrm{CIC}_0$ consists of terms and three forms of declarations. First, we present the possible forms of terms of $\mathrm{CIC}_0$ together with a brief intuitive explanation of their meaning. The terms of $\mathrm{CIC}_0$ are essentially simplified terms of Coq. Below by $t$, $s$, $u$, $\tau$, $\sigma$, $\rho$, $\kappa$, $\alpha$, $\beta$, etc., we denote terms of $\mathrm{CIC}_0$, by $c$, $c'$, $f$, $F$, etc., we denote constants of $\mathrm{CIC}_0$, and by $x$, $y$, $z$, etc., we denote variables. We use $\vec{t}$ for a sequence of terms $t_1 \dots t_n$ of an unspecified length $n$, and analogously for a sequence of variables $\vec{x}$. For instance, $s\vec{y}$ stands for $sy_1 \dots y_n$, where $n$ is not important or implicit in the context. Analogously, we use $\lambda\vec{x} : \vec{\tau}.t$ for $\lambda x_1 : \tau_1.\lambda x_2 : \tau_2.\dots.\lambda x_n : \tau_n.t$, with $n$ implicit or unspecified.

A *term* of $\mathrm{CIC}_0$ has one of the following forms.

- $c$. A constant.
- $x$. A variable.
- $ts$. An application.
- $\lambda x : t.s$. A lambda-abstraction.
- $\Pi x : t.s$. A dependent product. If $x$ does not occur free in $s$ then we abbreviate $\Pi x : t.s$ by $t \to s$.
- $\mathtt{case}(t, c, n, \lambda\vec{a} : \vec{\alpha}.\lambda x : c\vec{p}\vec{a}.\tau, \lambda\vec{x_1} : \vec{\tau_1}.s_1, \dots, \lambda\vec{x_k} : \vec{\tau_k}.s_k)$. A case expression. Here $t$ is the term matched on, $c$ is a constant such that

$$I_n(c : \gamma := c_1 : \gamma_1, \dots, c_k : \gamma_k)$$

  is an inductive declaration in the global environment (see the definition of inductive declarations below for an explanation), the type of $t$ has the form $c\vec{p}\vec{u}$, the integer $n$ denotes the number of parameters (which is the length of $\vec{p}$), the type $\tau[\vec{u}/\vec{a}, t/x]$ is the return type, i.e., the type of the whole case expression, $\vec{a} \cap \mathrm{FV}(\vec{p}) = \emptyset$, and $s_i[\vec{v}/\vec{x_i}]$ is the value of the case expression if the value of $t$ is $c_i\vec{p}\vec{v}$.
- $\mathtt{fix}(f_i, f_1 : t_1 := s_1, \dots, f_n : t_n := s_n)$. A mutually recursive fixpoint definition. The value of this is the function $f_i$ (where $1 \le i \le n$) defined by $s_i$. The variables $f_1, \dots, f_n$ may occur in $s_1, \dots, s_n$. All functions are required to be terminating.
- $\mathtt{let}(x : t := s, u)$. A let-expression locally binding $x$ of type $t$ to $s$ in $u$.
- $\mathtt{cast}(t, \tau)$. A type cast: $t$ is forced to have type $\tau$.

We assume that the following special constants are among the constants of $\mathrm{CIC}_0$: Prop, Set, Type, $\top$, $\bot$, $\forall$, $\exists$, $\wedge$, $\vee$, $\leftrightarrow$, $\neg$, $=$. We usually write $\forall x : t.s$ and $\exists x : t.s$ instead of $\forall t(\lambda x : t.s)$ and $\exists t(\lambda x : t.s)$, respectively. For $\wedge$, $\vee$ and $\leftrightarrow$ we typically use infix notation. We usually write $t = s$ instead of $= \tau st$, omitting the type $\tau$. The purpose of having the *logical primitives* $\top, \bot, \forall, \exists, \wedge, \vee, \leftrightarrow, \neg, =$ in $\mathrm{CIC}_0$ is to be able to directly represent the Coq definitions of logical connectives. These primitives are used during the translation. We directly export the Coq definitions and inductive types which represent the logical connectives (the ones declared in the `Init.Logic` module), as well as equality, to the logical primitives of $\mathrm{CIC}_0$. In particular, `Init.Logic.all` is exported to $\forall$.

In $\mathrm{CIC}_0$ the universe constraints on Type present in the Coq logic are lost. This is not dangerous in practice, because the ATPs are not strong enough to exploit the resulting inconsistency. Proofs of paradoxes present in Coq's standard library are explicitly filtered-out by our plugin.

A *declaration* of $\mathrm{CIC}_0$ has one of the following forms.

- A *definition* $c = t : \tau$. This is a definition of a constant $c$ stating that $c$ is (definitionally) equal to $t$ and it has type $\tau$.

- A *typing declaration* $c : \tau$. This is a declaration of a constant $c$ stating that it has type $\tau$.
- An *inductive declaration* $I_k(c : \tau := c_1 : \tau_1, \ldots, c_n : \tau_n)$ of $c$ of type $\tau$ with $k$ parameters and $n$ constructors $c_1, \ldots, c_n$ having types $\tau_1, \ldots, \tau_n$ respectively. We require $\tau \Downarrow \Pi\vec{y} : \vec{\sigma}.\Pi\vec{y'} : \vec{\sigma'}.s$ with $s \in \{\mathrm{Prop}, \mathrm{Set}, \mathrm{Type}\}$ and $\tau_i \Downarrow \Pi\vec{y} : \vec{\sigma}.\vec{x_i} : \vec{\alpha_i}.c\vec{y}\vec{u_i}$ for $i = 1, \ldots, n$, where the length of $\vec{y}$ is $k$ and $a \Downarrow b$ means that $a$ evaluates to $b$. Usually, we omit the subscript $k$ when irrelevant or clear from the context.
  For instance, a polymorphic type of lists defined as an inductive type in Type with a single parameter of type Type may be represented by

$$\begin{aligned} I_1(&\texttt{List} : \mathrm{Type} \to \mathrm{Type} := \\ &\quad \texttt{nil} : (\Pi A : \mathrm{Type}.\texttt{List}\, A), \\ &\quad \texttt{cons} : (\Pi A : \mathrm{Type}.A \to \texttt{List}\, A \to \texttt{List}\, A)). \end{aligned}$$

  Mutually inductive types may also be represented, because we do not require the names of inductive declarations to occur in any specific order. For instance, the inductive predicates `even` and `odd` may be represented by two inductive declarations

$$\begin{aligned} I_0(&\texttt{even} : \texttt{nat} \to \mathrm{Prop} := \\ &\quad \texttt{even\_0} : \texttt{even}\, 0, \\ &\quad \texttt{even\_S} : \Pi n : \texttt{nat}.\texttt{odd}\, n \to \texttt{even}\, (Sn)). \\ I_0(&\texttt{odd} : \texttt{nat} \to \mathrm{Prop} := \\ &\quad \texttt{odd\_S} : \Pi n : \texttt{nat}.\texttt{even}\, n \to \texttt{odd}\, (Sn)). \end{aligned}$$

An *environment* of $\mathrm{CIC}_0$ is a set of declarations. We assume an implicit global environment $E$. The environment $E$ is assumed to contain appropriate typing declarations for the logical primitives. A $\mathrm{CIC}_0$ *context* is a list of declarations of the form $x : t$ with $t$ a term of $\mathrm{CIC}_0$ and $x$ the declared $\mathrm{CIC}_0$ variable. We assume the variables declared in a context are pairwise disjoint. We denote environments by $E$, $E'$, etc., and contexts by $\Gamma$, $\Gamma'$, etc. We write $\Gamma, x : \tau$ to denote the context $\Gamma$ with $x : \tau$ appended. We denote the empty context by $\langle\rangle$. A type judgement of $\mathrm{CIC}_0$ has the form $\Gamma \vdash t : \tau$ where $\Gamma$ is a context and $t, \tau$ are terms. If $\Gamma \vdash t : \tau$ and $\Gamma \vdash \tau : \sigma$ then we write $\Gamma \vdash t : \tau : \sigma$. A *$\Gamma$-proposition* is a term $t$ such that $\Gamma \vdash t : \mathrm{Prop}$. A *$\Gamma$-proof* is a term $t$ such that $\Gamma \vdash t : \tau : \mathrm{Prop}$ for some term $\tau$.

The set $\mathrm{FV}(t)$ of free variables of a term $t$ is defined in the usual way. To save on notation we sometimes treat $\mathrm{FV}(t)$ as a list. For a context $\Gamma$ which includes declarations of all free variables of $t$, the free variable context $\mathrm{FC}(\Gamma; t)$ of $t$ is defined inductively:

- $\mathrm{FC}(\langle\rangle; t) = \langle\rangle$,
- $\mathrm{FC}(\Gamma, x : \tau; t) = \mathrm{FC}(\Gamma; \lambda x : \tau.t), x : \tau$ if $x \in \mathrm{FV}(t)$,
- $\mathrm{FC}(\Gamma, x : \tau; t) = \mathrm{FC}(\Gamma; t)$ if $x \notin \mathrm{FV}(t)$.

If $\Gamma$ includes declarations of all variables from a set of variables $V$, then we define $\mathrm{FF}_\Gamma(V)$ to be the set of those $y \in V$ which are not $\Gamma$-proofs. Again, to save on notation we sometimes treat $\mathrm{FF}_\Gamma(V)$ as a list.

Our translation encodes $\mathrm{CIC}_0$ in untyped first-order logic with equality (FOL). We also implemented a straightforward information-forgetting export of Coq declarations into the syntax of $\mathrm{CIC}_0$. We describe the translation and the export in the next section.

In the translation of $\mathrm{CIC}_0$ we need to perform (approximate) type checking to determine which terms are propositions (have type Prop), i.e. we need to check whether a given term $t$ in a given context $\Gamma$ has type Prop. For this purpose we implemented a specialised efficient procedure to do so. In fact, this procedure is slightly incomplete. The point here is to approximately identify which types are intended to represent propositions. In proof assistants or proof developments where types other than those of sort Prop are intended to represent propositions the procedure needs to be changed.

All $\mathrm{CIC}_0$ terms we are interested in correspond to typable (and thus strongly normalizing) Coq terms, i.e., Coq terms are exported in a simple information-forgetting way to appropriate $\mathrm{CIC}_0$ terms. We will assume that for any exported term there exists a type in logic of Coq, it is unique, and it is preserved under context extension. This assumption is not completely theoretically justified, but is useful in practice.

## 4 Premise selection

The first component of a hammer preselects a subset of the accessible facts most likely to be useful in proving the user given goal. In this section we present the premise selection algorithm proposed for a hammer for dependently typed theory. We reuse the two most successful filters used in HOLyHammer [KU14] and Sledgehammer [BGK+16] adapted to the $\mathrm{CIC}_0$ representation of proof assistant knowledge. We first discuss the features and labels useful for that representation and further describe the $k$-NN and naive Bayes classifiers, which we used in our implementation.

### 4.1 Features and Labels

A simple possible characterization of statements in a proof assistant library is to use the sets of symbols that appear in these statements. It is possible to extend this set in many ways [KUV15], including various kinds of structure of the statements, types, and normalizing variables (all variables will be replaced by a single symbol `X`). In the case of $\mathrm{CIC}_0$, the constants are already both term constants and type constructors. We omit the basic logical constants, as they will not be useful for automated theorem provers which assume first-order logic. We further augment the set of features by inspecting the parse tree: constants and constant-variable pairs that share an edge in the parse tree give rise to a feature of the statement. We will denote such features of a theorem $T$ by $F(T)$.

For each feature $f$ we additionally compute a feature weight $w(f)$ that estimates the importance of the feature. Based on the HOLyHammer experiments with feature weights [KU15b], we use TF-IDF [Jon72] to compute feature weights. This ensures that rare features are more important than common ones.

Like in usual premise selection, the dependencies of theorems will constitute the labels for the learning algorithms. The dependencies for a theorem or definition $T$, which we will denote $D(T)$, are the constants occuring in the type of $T$ or in the proof term (or the unfolding) of $T$. Note that these dependencies may not be complete, because in principle an ATP proof of $T$ may need some additional information that in Coq is incorporated into type-checking but not used to build proof terms,

e.g. definitions of constants, facts which are necessary to establish types of certain terms.

For example, consider the theorem $T$ = `Between.between_le` from the Coq standard library with the statement:

```
forall k l, between k l -> k <= l.
```

In the section where this theorem is declared there is the following variable declaration:

```
Variable P : nat -> Prop.
```

The features and dependencies of $T$ are:

$$F(T) = \{ \texttt{"Between.Between.between", "Between.Between.between-X",} \\ \texttt{"Coq.Init.Datatypes.nat", "Coq.Init.Peano.le",} \\ \texttt{"Coq.Init.Peano.le-X"}\}$$

$$D(T) = \{ \texttt{"Between.Between.between", "Between.Between.between\_ind",} \\ \texttt{"Coq.Init.Datatypes.nat", "Coq.Init.Peano.le",} \\ \texttt{"Coq.Init.Peano.le\_S", "Coq.Init.Peano.le\_n", "P"}\}$$

The `-X` features correspond to constants applied to variables. Similarly, in more complex examples constant-constant applications (such as the successor of zero) give rise to such compound features.

### 4.2 $k$-Nearest Neighbors

The $k$ nearest neighbors classifier ($k$-NN) finds a given number $k$ of accessible facts which are most similar to the current goal. The distance for two statements $a, b$ is defined by the function (higher values means more similar, $\tau_1$ is a constant which gives more similar statements an additional advantage):

$$s(a,b) = \sum\nolimits_{f \in F(a) \cap F(b)} w(f)^{\tau_1}$$

The dependencies of the selected facts will be used to estimate the relevance of all accessible facts. Given the set of the $k$ nearest neighbors $N$ together with their nearness values, the relevance of a visible fact $a$ for the goal $g$ is

$$\left(\tau_2 \sum\nolimits_{b \in N | a \in D(b)} \frac{s(b,g)}{|D(b)|}\right) + \begin{cases} s(a,g) & \text{if } a \in N \\ 0 & \text{otherwise} \end{cases}$$

where $\tau_2$ is a constant which gives more importance to the dependencies. We have used the values $\tau_1 = 6$ and $\tau_2 = 2.7$ in our implementation, which were found experimentally in our previous work [KU13b].

There are two modifications of the standard $k$-NN algorithm. First, when deciding on the labels to predict based on the neighbors, we not only include the labels associated with the neighbors based on the training examples (this corresponds to past proofs) but also the neighbors themselves. This is because a theorem is in principle provable from itself in zero steps, and this information is not included in

the training data. Furthermore, theorems that have been proved, but have not been used yet, would not be accessible to the algorithm without this modification.

Second, we do not use a fixed number $k$, instead we fix the number of facts with non-zero relevance that need to be predicted. We start with $k = 1$ and if not enough facts have been selected, we increase $k$ iteratively. This allows creating ATP problems of proportionate complexity.

### 4.3 Sparse Naive Bayes

The sparse naive Bayes classifier estimates the relevance of a fact $a$ for a goal $g$ by the probability

$$P(a \text{ is used in the proof of } g)$$

Since the goal is only characterized by its features, the probability can be further estimated by:

$$P(a \text{ is used in a proof of } s \mid s \text{ has features } F(g))$$

where $s$ is an arbitrary proved theorem, abstracting from the goal $g$.

For efficiency reasons the computation of the relevance of $a$ is restricted to the features of $a$ and the features that were ever present when $a$ was used as a dependency. More formally, the *extended features* $\overline{F}(a)$ of $a$ are:

$$\overline{F}(a) = F(a) \cup \bigcup_{a \in \mathrm{D}(b)} F(b)$$

The probability can be thus estimated by the statements $s$ which have the features $F(g)$ but do not have the features $\overline{F}(a) - F(g)$:

$$P\big(a \text{ is used in a proof of } s \mid F(a) \subseteq F(g) \wedge F(a) \text{ misses } \overline{F}(a) - F(g)\big)$$

Assuming that the features are independent[1] the Bayes's rule can be applied to transform the probability to the following product of probabilities:

$$\begin{aligned}
&P(a \text{ is used in the proof of } s) \\
&\cdot \prod\nolimits_{f \in F(g) \cap \overline{F}(a)} P\big(s \text{ has feature } f \mid a \text{ is used in the proof of } s\big) \\
&\cdot \prod\nolimits_{f \in F(g) - \overline{F}(a)} P\big(s \text{ has feature } f \mid a \text{ is not used in the proof of } s\big) \\
&\cdot \prod\nolimits_{f \in \overline{F}(a) - F(g)} P\big(s \text{ does not have feature } f \mid a \text{ is used in the proof of } s\big)
\end{aligned}$$

The expressions can be finally estimated:

$$P(a \text{ is used in a proof of } s) = \frac{t(a)}{K}$$

$$P\big(s \text{ has feature } f \mid a \text{ is used in the proof of } s\big) = \frac{s(a,f)}{t(a)}$$

[1] there are many dependencies among the features, however considering such dependenceis makes premise selection very slow and gives little improvement both when it comes to machine learning metrics and in practical hammer use [AHK+14].

$$P\big(s \text{ does not have feature } f \mid a \text{ is used in the proof of } s\big) = 1 - \frac{s(a,f)}{t(a)}$$

using two auxiliary functions that can be computed from the dependencies:

- $s(a,f)$ is the number of times $a$ has been a dependency of a fact characterized by the feature $f$;
- $t(a)$ is the number of times $a$ has been a dependency;

as well as the number $K$ of all theorems proved so far.

In our actual implementation we further introduce minor modifications to avoid any of the probabilities become zero and we estimate the logarithms of probabilities to avoid multiplying small numbers which might cause numerical instability. The classifier can finally estimate the relevance of all visible facts and return the requested number of them that are most likely to lead to a successful proof of the conjecture.

## 5 Translation

In this section we describe a translation of Coq goals through $\mathrm{CIC}_0$ to untyped first-order logic with equality. The translation presented here is a significantly improved version of our translation presented at HaTT [CK16]. It has been made more complete, many optimisations have been introduced, and several mistakes have been eliminated.

The translation is neither sound nor complete. In particular, it assumes proof irrelevance (in the sense of erasing proof terms), it omits universe constraints on Type, and some information is lost in the export to $\mathrm{CIC}_0$. However, it is sound and complete "enough" to be practically usable by a hammer (just like the hammers for other systems, it works very well for essentially first-order logic goals and becomes much less effective with other features of the logics [BKPU16]). The limitations of the translation and further issues of the current approach are explained in more detail in Sections 5.6 and 9. Some similar issues were handled in the context of code extraction in [Let04].

The translation proceeds in three phases. First, we export Coq goals to $\mathrm{CIC}_0$. Next we translate $\mathrm{CIC}_0$ to first-order logic with equality. In the first-order language we assume a unary predicate $P$, a binary predicate $T$ and a binary function symbol @. Usually, we write $ts$ instead of $@(t,s)$. Intuitively, an atom of the form $P(t)$ asserts the provability of $t$, and $T(t,\tau)$ asserts that $t$ has type $\tau$. In the third phase we perform some optimisations on the generated FOL problem, e.g. replacing some terms of the form $P(cts)$ with $c(t,s)$.

A FOL *axiom* is a pair of a FOL formula and a constant (label). We translate $\mathrm{CIC}_0$ to a set of FOL axioms. The labels are used to indicate which axioms are translations of which lemmas. When we do not mention the label of an axiom, then the label is not important.

### 5.1 Export of Coq data

The Coq declarations are exported in a straightforward way, translating Coq terms to corresponding terms of $\mathrm{CIC}_0$, possibly forgetting some information like e.g. universe constraints on Type. We implemented a Coq kernel plugin which exports the Coq kernel data structures. We briefly comment on several aspects of the export.

- Definitions are exported as $\mathrm{CIC}_0$ definitions.
- Axioms are exported as $\mathrm{CIC}_0$ typing declarations.
- Free variables (e.g. current hypotheses or variables from a currently open section) are exported as $\mathrm{CIC}_0$ constants with appropriate typing declarations.
- Inductive types are exported as $\mathrm{CIC}_0$ inductive declarations. Induction principles and recursor definitions are exported as separate $\mathrm{CIC}_0$ definitions.
- Coinductive types are treated in the same way as inductive types, except that no induction principles or recursor definitions are exported for them.
- Mutual inductive types are exported separately for each constituent inductive type. See Section 3.
- The Coq construct `cofix` is exported to `fix` in $\mathrm{CIC}_0$ with a special flag that affects the evaluation algorithm. We omitted this flag from the description of $\mathrm{CIC}_0$ for the sake of simplicity.
- Modules and functors are not exported. Objects inside a module are exported with the name of the module prefixed to the name of the object.
- Universe constraints on Type are not exported. Proofs of paradoxes present in the standard library, e.g., Hurken's paradox, are explicitly filtered out and not exported.
- The following objects from the `Init.Logic` module are represented directly by the corresponding logical primitives of $\mathrm{CIC}_0$: `True`, `False`, `all`, `ex`, `and`, `or`, `iff`, `eq`. No other objects from the `Init.Logic` module are exported.
- Records are translated to inductive types already by Coq. Primitive record projections are not supported by our plugin.
- Existential metavariables are not exported. Currently it is not possible to use the hammer plugin when the proof state contains some uninstantiated existential metavariables.

The limitations of the translation, including these stemming from the incompleteness of the export as well as of the current architecture will be discussed in Sections 5.6 and 9.

### 5.2 Translating terms

The terms of $\mathrm{CIC}_0$ are translated using three mutually recursively defined functions $\mathcal{F}$, $\mathcal{G}$ and $\mathcal{C}$. The function $\mathcal{F}$ encodes propositions as FOL formulas and is used for terms of $\mathrm{CIC}_0$ having type Prop, i.e., for propositions of $\mathrm{CIC}_0$. The function $\mathcal{G}$ encodes types as guards and is used for terms of $\mathrm{CIC}_0$ which have type Type but not Prop. The function $\mathcal{C}$ encodes $\mathrm{CIC}_0$ terms as FOL terms. During the translation we add some fresh constants together with axioms (in FOL) specifying their meaning. Hence, strictly speaking, the codomain of each of the functions $\mathcal{F}$, $\mathcal{G}$ and $\mathcal{C}$ is the Cartesian product of the set of FOL formulas (or terms) – the desired encoding – and the powerset of the set of FOL formulas – the set of axioms added during the translation. However, it is more readable to describe the functions assuming a global mutable collection of FOL axioms.

Our translation assumes proof irrelevance. We use a fresh constant `prf` to represent an arbitrary proof object (of any inhabited proposition). For the sake of efficiency, $\mathrm{CIC}_0$ propositions are translated directly to FOL formulas using the $\mathcal{F}$ function. The $\mathrm{CIC}_0$ types which are not propositions are translated to guards which essentially

specify what it means for an object to have the given type. The formula $\mathcal{G}(t, \alpha)$ intuitively means "$t$ has type $\alpha$". For instance, for a (closed) type $\tau = \Pi x : \alpha.\beta$ we have

$$\mathcal{G}(f, \tau) = \forall x.\mathcal{G}(x, \alpha) \to \mathcal{G}(fx, \beta)$$

So $\mathcal{G}(f, \tau)$ says that an object $f$ has type $\tau = \Pi x : \alpha.\beta$ if for any object $x$ of type $\alpha$, the application $fx$ has type $\beta$ (in which $x$ may occur free).

Below we give definitions of the functions $\mathcal{F}$, $\mathcal{G}$ and $\mathcal{C}$. These functions are in fact parameterised by a $\mathrm{CIC}_0$ context $\Gamma$, which we write as a subscript. In the description of the functions we implicitly assume that variable names are chosen appropriately so that no unexpected variable capture occurs. Also we assume an implicit global environment $E$. This environment is used for type checking. The typing declarations for $\mathrm{CIC}_0$ logical primitives, as described in the previous section, are assumed to be present in $E$. During the translation also some new declarations are added to the environment. We assume all $\mathrm{CIC}_0$ constants are also FOL constants, and analogously for variables. We use the notation $t_1 \approx_\Gamma t_2$ for $t_1 \leftrightarrow t_2$ if $\Gamma \vdash t_1 : \mathrm{Prop}$, or for $t_1 = t_2$ if $\Gamma \nvdash t_1 : \mathrm{Prop}$.

The function $\mathcal{F}$ encoding propositions as FOL formulas:

- If $\Gamma \vdash t : \mathrm{Prop}$ then $\mathcal{F}_\Gamma(\Pi x : t.s) = \mathcal{F}_\Gamma(t) \to \mathcal{F}_{\Gamma,x:t}(s)$.
- If $\Gamma \nvdash t : \mathrm{Prop}$ then $\mathcal{F}_\Gamma(\Pi x : t.s) = \forall x.\mathcal{G}_\Gamma(x, t) \to \mathcal{F}_{\Gamma,x:t}(s)$.
- $\mathcal{F}_\Gamma(\forall x : t.s) = \forall x.\mathcal{G}_\Gamma(x, t) \to \mathcal{F}_{\Gamma,x:t}(s)$.
- $\mathcal{F}_\Gamma(\exists x : t.s) = \exists x.\mathcal{G}_\Gamma(x, t) \wedge \mathcal{F}_{\Gamma,x:t}(s)$.
- $\mathcal{F}_\Gamma(t \circ s) = \mathcal{F}_\Gamma(t) \circ \mathcal{F}_\Gamma(s)$ where $\circ \in \{\wedge, \vee, \leftrightarrow\}$.
- $\mathcal{F}_\Gamma(\neg t) = \neg\mathcal{F}_\Gamma(t)$.
- $\mathcal{F}_\Gamma(t = s) = (\mathcal{C}_\Gamma(t) = \mathcal{C}_\Gamma(s))$.
- Otherwise, if none of the above apply, $\mathcal{F}_\Gamma(t) = P(\mathcal{C}_\Gamma(t))$.

The function $\mathcal{G}$ encoding types as guards:

- If $w = \Pi x : t.s$ and $\Gamma \vdash t : \mathrm{Prop}$ then

$$\mathcal{G}_\Gamma(u, w) = \mathcal{F}_\Gamma(t) \to \mathcal{G}_{\Gamma,x:t}(u, s).$$

- If $w = \Pi x : t.s$ and $\Gamma \nvdash t : \mathrm{Prop}$ then $\mathcal{G}_\Gamma(u, w) = \forall x.\mathcal{G}_\Gamma(x, t) \to \mathcal{G}_{\Gamma,x:t}(ux, s)$.
- If $w$ is not a product then $\mathcal{G}_\Gamma(u, w) = T(u, \mathcal{C}_\Gamma(w))$.

The function $\mathcal{C}$ encoding terms as FOL terms:

- $\mathcal{C}_\Gamma(c) = c$ for a constant $c$,
- $\mathcal{C}_\Gamma(x) = x$ for a variable $x$ if $x$ is not a $\Gamma$-proof,
- $\mathcal{C}_\Gamma(x) = \mathtt{prf}$ for a variable $x$ if $x$ is a $\Gamma$-proof,
- $\mathcal{C}_\Gamma(ts)$ is equal to:
  - $\mathtt{prf}$ if $\mathcal{C}_\Gamma(t) = \mathtt{prf}$,
  - $\mathcal{C}_\Gamma(t)$ if $\mathcal{C}_\Gamma(t) \neq \mathtt{prf}$ but $\mathcal{C}_\Gamma(s) = \mathtt{prf}$,
  - $\mathcal{C}_\Gamma(t)\mathcal{C}_\Gamma(s)$ otherwise.
- $\mathcal{C}_\Gamma(\Pi x : t.s) = R\vec{y}$ for a fresh constant $F$ where $\vec{y} = \mathrm{FF}_\Gamma(\mathrm{FC}(\Gamma; \Pi x : t.s))$ and
  - if $\Gamma \vdash (\Pi x : t.s) : \mathrm{Prop}$ then $\forall\vec{y}.P(F\vec{y}) \leftrightarrow \mathcal{F}_\Gamma(\Pi x : t.s)$ is a new axiom,
  - if $\Gamma \nvdash (\Pi x : t.s) : \mathrm{Prop}$ then $\forall\vec{y}z.T(z, F\vec{y}) \leftrightarrow \mathcal{G}_\Gamma(z, \Pi x : t.s)$ is a new axiom.
- $\mathcal{C}_\Gamma(\lambda\vec{x} : \vec{\tau}.t) = F\vec{y_0}$ for a fresh constant $F$ where
  - $t$ does not start with a lambda-abstraction any more,
  - $\Gamma, \vec{x} : \vec{\tau} \vdash t : \alpha$,
  - $\vec{y} : \vec{\rho} = \mathrm{FC}(\Gamma; \lambda\vec{x} : \vec{\tau}.t)$,

- $\vec{y_0} = \mathrm{FF}_\Gamma(\vec{y})$ and $\vec{x_0} = \mathrm{FF}_{\Gamma,\vec{x}:\vec{\tau}}(\vec{x})$,
- the typing declaration $F : \Pi\vec{y} : \vec{\rho}.\Pi\vec{x} : \vec{\tau}.\alpha$ is added to the global environment $E$ (before the recursive call to $\mathcal{F}_\Gamma$ below),
- the following is a new axiom:

$$\forall \vec{y_0}\vec{x_0}.\mathcal{F}_{\Gamma,\vec{x}:\vec{\tau}}(F\vec{y}\vec{x} \approx_{\Gamma,\vec{x}:\vec{\tau}} t).$$

  Note that the call to $\mathcal{F}$ will remove those variable arguments to $F$ which are $\Gamma, \vec{x} : \vec{\tau}$-proofs. Hence, ultimately $F$ will occur as $F\vec{y_0}\vec{x_0}$ in the above axiom.

- If $t$ is a $\Gamma$-proof then

$$\mathcal{C}_\Gamma(\mathtt{case}(t, c, n, \lambda\vec{a} : \vec{\alpha}.\lambda x : c\vec{p}\vec{a}.\tau, \lambda\vec{x_1} : \vec{\tau_1}.s_1, \ldots, \lambda\vec{x_k} : \vec{\tau_k}.s_k)) = C$$

  for a fresh constant $C$.
- If $t$ is not a $\Gamma$-proof then

$$\mathcal{C}_\Gamma(\mathtt{case}(t, c, n, \lambda\vec{a} : \vec{\alpha}.\lambda x : c\vec{p}\vec{a}.\tau, \lambda\vec{x_1} : \vec{\tau_1}.s_1, \ldots, \lambda\vec{x_k} : \vec{\tau_k}.s_k)) = F\vec{y_0}$$

  for a fresh constant $F$ where
  - $I(c : \gamma := c_1 : \gamma_1, \ldots, c_k : \gamma_k) \in E$,
  - $\vec{y} : \vec{\rho} = \mathrm{FC}(\Gamma; \mathtt{case}(t, c, n, \lambda\vec{a} : \vec{\alpha}.\lambda x : c\vec{p}\vec{a}.\tau, \lambda\vec{x_1} : \vec{\tau_1}.s_1, \ldots, \lambda\vec{x_k} : \vec{\tau_k}.s_k))$,
  - $\vec{y_0} = \mathrm{FF}_\Gamma(\vec{y})$,
  - $\vec{y_1} : \vec{\rho_1} = \mathrm{FC}(\Gamma; t)$,
  - $\Gamma \vdash t : c\vec{p}\vec{u}$ for some terms $\vec{u}$,
  - the declaration $F : \Pi\vec{y} : \vec{\rho}.\tau[\vec{u}/\vec{a}, t/x]$ is added to the global environment $E$,
  - the following is a new axiom:

$$\begin{aligned}\forall\vec{y_0}.\mathrm{guards}_{\vec{y_1}:\vec{\rho_1}}(\mathcal{F}_\Gamma\ ((\ &\exists\vec{x_1} : \vec{\tau_1}.t = c_1\vec{p}\vec{x_1} \wedge F\vec{y} \approx_{\Gamma,\vec{x_1}:\vec{\tau_1}} s_1)\\ &\vee \ldots\\ &\vee\ (\exists\vec{x_k} : \vec{\tau_k}.t = c_k\vec{p}\vec{x_k} \wedge F\vec{y} \approx_{\Gamma,\vec{x_k}:\vec{\tau_k}} s_k)))\end{aligned}$$

    where for a FOL formula $\varphi$ and a context $\Gamma$ we define $\mathrm{guards}_\Gamma(\varphi)$ inductively as follows:
    - $\mathrm{guards}_{\langle\rangle}(\varphi) = \varphi$,
    - $\mathrm{guards}_{\Gamma,x:\tau}(\varphi) = \mathrm{guards}_\Gamma(\mathcal{F}_\Gamma(\tau) \to \varphi)$ if $\Gamma \vdash \tau : \mathrm{Prop}$,
    - $\mathrm{guards}_{\Gamma,x:\tau}(\varphi) = \mathrm{guards}_\Gamma(\mathcal{G}_\Gamma(x, \tau) \to \varphi)$ if $\Gamma \nvdash \tau : \mathrm{Prop}$.

- $\mathcal{C}_\Gamma(\mathtt{fix}(f_j, f_1 : \tau_1 := t_1, \ldots, f_n : \tau_n := t_n)) = F_j\vec{y_0}$ where
  - $\vec{y} : \vec{\alpha} = \mathrm{FC}(\Gamma; \mathtt{fix}(f_j, f_1 : \tau_1 := t_1, \ldots, f_n : \tau_n := t_n))$,
  - $\vec{y_0} = \mathrm{FF}_\Gamma(\vec{y})$,
  - $F_1, \ldots, F_n$ are fresh constants,
  - for $i = 1, \ldots, n$ the typing declarations $F_i : \Pi\vec{y} : \vec{\alpha}.\tau_i$ are added to the global environment $E$,
  - for $i = 1, \ldots, n$ the following are new axioms:

$$\forall\vec{y_0}.\mathcal{F}_\Gamma(F_i\vec{y} \approx_\Gamma t_i[F_1\vec{y}/f_1, \ldots, F_n\vec{y}/f_n]).$$

- $\mathcal{C}_\Gamma(\mathtt{let}(x : \tau := t, s)) = \mathcal{C}_\Gamma(s[F\vec{y_0}/x])$ for a fresh constant $F$ where
  - $\vec{y} : \vec{\alpha} = \mathrm{FC}(\Gamma; t\tau)$,
  - $\vec{y_0} = \mathrm{FF}_\Gamma(\vec{y})$,
  - $\sigma = \Pi\vec{y} : \vec{\alpha}.\tau$,
  - the definition $F = (\lambda\vec{y} : \vec{\alpha}.t) : \sigma$ is added to the global environment $E$ (before the recursive call to $\mathcal{C}_\Gamma$ above),

  - if $\nvdash \sigma : \text{Prop}$ then $\forall \vec{y_0}.F\vec{y_0} = \mathcal{C}_\Gamma(t)$ is a new axiom.
- $\mathcal{C}_\Gamma(\texttt{cast}(\texttt{prf}, \tau)) = \texttt{prf}$.
- If $t \neq \texttt{prf}$ then $\mathcal{C}_\Gamma(\texttt{cast}(t, \tau)) = F\vec{y_0}$ for a fresh constant $F$ where
  - $\vec{y} : \vec{\alpha} = \text{FC}(\Gamma; t\tau)$,
  - $\vec{y_0} = \text{FF}_\Gamma(\vec{y})$,
  - $\sigma = \Pi\vec{y} : \vec{\alpha}.\tau$,
  - the definition $F = (\lambda\vec{y} : \vec{\alpha}.t) : \sigma$ is added to the global environment $E$,
  - if $\nvdash \sigma : \text{Prop}$ then $\forall \vec{y_0}.F\vec{y_0} = \mathcal{C}_\Gamma(t)$ is a new axiom.

*Example 1* A $\text{CIC}_0$ proposition

$$t = \Pi x : N.\Pi f : \alpha \to N \to N.\Pi q : \alpha.fqx = x$$

in the context

$$\Gamma = N : \text{Type}, \alpha : Prop$$

is translated to

$$\mathcal{F}_\Gamma(t) = \forall x.T(x, N) \to \forall f.(P(\alpha) \to \forall y.T(y, N) \to T(fy, N)) \to P(\alpha) \to fx = x.$$

In practice, checking the conditions $\Gamma \vdash t : \text{Prop}$ is performed by our specialised approximate proposition-checking algorithm. Checking whether a term $t$ is a $\Gamma$-proof occurs in two cases.

1. $t$ is the term matched on in a `case`-expression $\texttt{case}(t, c, \ldots)$. Then there is an inductive declaration $I_n(c : \gamma := \ldots)$ in the global environment. We check if the normal form of $\gamma$ has target Prop.
2. $t = x$ is a variable. Then we check if the type assigned to $x$ by the context $\Gamma$ is a proposition.

We write $\varphi(\sigma)$ to denote that a FOL formula $\varphi$ has $\sigma$ as a subformula. Then $\varphi(\sigma')$ denotes the formula $\varphi$ with $\sigma$ replaced by $\sigma'$. We use an analogous notation when $\sigma$ is a FOL term instead of a formula.

Note that each new axiom defining a constant $F$ intended to replace ("lift-out") a $\lambda$-abstraction, a case expression or a fixpoint definition has the form

$$\forall\vec{x}.\varphi(F\vec{x} = t)$$

or

$$\forall\vec{x}.\varphi(P(F\vec{x}) \leftrightarrow \psi).$$

We will call each such axiom the *lifting axiom for* $F$. For lambda abstractions, this is equivalent to lambda-lifing, which is a common technique used by hammers for HOL and Mizar. In $\text{CIC}_0$ however other kinds of terms do bind variables (for example `case` and `fix`) and lifting axioms need to be created for such terms as well.

### 5.3 Translating declarations

Declarations of $\mathrm{CIC}_0$ are encoded as FOL axioms. As before, a global $\mathrm{CIC}_0$ environment $E$ is assumed. During the translation of a declaration the functions $\mathcal{F}$, $\mathcal{G}$ and $\mathcal{C}$ from the previous subsection are used. These functions may themselves add some FOL axioms, which are then also included in the result of the translation of the declaration. We proceed to describe the translation for each of the three forms of $\mathrm{CIC}_0$ declarations. Whenever we write $\mathcal{F}$, $\mathcal{G}$, $\mathcal{C}$ without subscript, the empty context $\langle\rangle$ is assumed as the subscript.

A definition $c = t : \tau$ is translated as follows.

- If $\vdash \tau : \mathrm{Prop}$ then add $\mathcal{F}(\tau)$ as a new axiom with label $c$.
- If $\nvdash \tau : \mathrm{Prop}$ then
  - add $\mathcal{G}(c, \tau)$ as a new axiom,
  - if $\tau = \mathrm{Prop}$ then add $c \leftrightarrow \mathcal{F}(t)$ as a new axiom with label $c$,
  - if $\tau = \mathrm{Set}$ or $\tau = \mathrm{Type}$ then add $\forall f.cf \leftrightarrow \mathcal{G}(f, t)$ as a new axiom with label $c$,
  - if $\tau \notin \{\mathrm{Prop}, \mathrm{Set}, \mathrm{Type}\}$ then add $c = \mathcal{C}(t)$ as a new axiom with label $c$.

A typing declaration $c : \tau$ is translated as follows.

- If $\vdash \tau : \mathrm{Prop}$ then add $\mathcal{F}(\tau)$ as a new axiom with label $c$.
- If $\nvdash \tau : \mathrm{Prop}$ then add $\mathcal{G}(c, \tau)$ as a new axiom with label $c$.

An inductive declaration $I(c : \tau := c_1 : \tau_1, \ldots, c_n : \tau_n)$ is translated as follows, where $\tau \Downarrow \Pi\vec{p} : \vec{\beta}.\Pi\vec{y} : \vec{\gamma}.s$ and $s \in \{\mathrm{Prop}, \mathrm{Set}, \mathrm{Type}\}$ and $\vec{\beta}$ are the types of the parameters of the inductive type and $\tau_i \Downarrow \Pi\vec{p} : \vec{\beta}.\Pi\vec{x_i} : \vec{\alpha_i}.c\vec{p}\vec{t_i}$ and the length of $\vec{y}$ and each $\vec{t_i}$ is $m$.

- Translate the typing declaration $c : \tau$.
- Translate each typing declaration $c_i : \tau_i$ for $i = 1, \ldots, n$.
- If $s \neq \mathrm{Prop}$ then for each $i = 1, \ldots, n$ add the following injectivity axiom:

$$\mathcal{F}(\forall\vec{x_i} : \vec{\alpha_i}.\forall\vec{x_i}' : \vec{\alpha_i}'.c_i\vec{x_i} = c_i\vec{x_i}' \to x_{i,1} = x'_{i,1} \wedge \ldots \wedge x_{i,k_i} = x'_{i,k_i})$$

  where $\vec{\alpha_i}' = \vec{\alpha_i}[\vec{x_i}'/\vec{x_i}]$.
- If $s \neq \mathrm{Prop}$ then for each $i, j = 1, \ldots, n$ with $i \neq j$ add the following discrimination axiom:

$$\mathcal{F}(\forall\vec{x_i} : \vec{\alpha_i}.\forall\vec{x_j} : \vec{\alpha_j}.c_i\vec{x_i} \neq c_j\vec{x_j}).$$

- If $s \neq \mathrm{Prop}$ then add the following inversion axiom:

$$\begin{aligned}\mathcal{F}(\forall\vec{p} : \vec{\beta}.\forall\vec{y} : \vec{\gamma}.\forall z : c\vec{p}\vec{y}\ .\ & (\exists\vec{x_1} : \vec{\alpha_1}.z = c_1\vec{p}\vec{x_1} \wedge y_1 = t_{1,1} \wedge \ldots \wedge y_m = t_{1,m}) \\ & \vee \ldots \\ & \vee\ (\exists\vec{x_n} : \vec{\alpha_n}.z = c_n\vec{p}\vec{x_n} \wedge y_1 = t_{n,1} \wedge \ldots \wedge y_m = t_{n,m})).\end{aligned}$$

- If $s = \mathrm{Prop}$ then add the following inversion axiom:

$$\begin{aligned}\mathcal{F}(\forall\vec{p} : \vec{\beta}.\forall\vec{y} : \vec{\gamma}.c\vec{p}\vec{y} \to ((\ & \exists\vec{x_1} : \vec{\alpha_1}.y_1 = t_{1,1} \wedge \ldots \wedge y_m = t_{1,m}) \\ & \vee \ldots \\ & \vee\ (\exists\vec{x_n} : \vec{\alpha_n}.y_1 = t_{n,1} \wedge \ldots \wedge y_m = t_{n,m}))).\end{aligned}$$

### 5.4 Translating problems

A $\mathrm{CIC}_0$ problem consists of a set of assumptions which are $\mathrm{CIC}_0$ declarations, and a conjecture which is a $\mathrm{CIC}_0$ proposition. A $\mathrm{CIC}_0$ problem is translated to a FOL problem by translating the assumptions to FOL axioms in the way described in the previous subsection, and translating the conjecture $t$ to a FOL conjecture $\mathcal{F}(t)$. New declarations added to the environment during the translation are *not* translated. For every $\mathrm{CIC}_0$ problem the following FOL axioms are added to the result of the translation:

- $T(\mathrm{Prop}, \mathrm{Type})$, $T(\mathrm{Set}, \mathrm{Type})$, $T(\mathrm{Type}, \mathrm{Type})$,
- $\forall y.T(y, \mathrm{Set}) \to T(y, \mathrm{Type})$.

### 5.5 Optimisations

We perform the following optimisations on the generated FOL problems, in the given order. Below, by an occurrence of a term $t$ (in the FOL problem) we mean an occurrence of $t$ in the set of FOL formulas comprising the given FOL problem.

- We recursively simplify the lifting axioms for the constants encoding $\lambda$-abstractions, case expressions and fixpoint definitions. For any lifting axiom $A$ for a constant $F$, if $A$ has the form
$$\forall \vec{x}.\varphi(F\vec{x} = G\vec{x})$$
such that $G$ has a lifting axiom $B$
$$\forall \vec{x}\forall \vec{y}.\psi(G\vec{x}\vec{y} = t)$$
and either $\varphi(\Box) = \Box$ or $\vec{y}$ is empty, then we replace the axiom $A$ by
$$\forall \vec{x}.\varphi(\forall \vec{y}.\psi(F\vec{x}\vec{y} = t))$$
and we remove the axiom $B$ and replace all occurrences of $G$ by $F$. When in the lifting axioms $A$ and $B$ we have logical equivalence $\leftrightarrow$ instead of equality $=$, then we adjust the replacement of $A$ appropriately, using $\leftrightarrow$ instead of $=$. We repeat applying this optimisation as long as possible.
- For a constant $c$, we replace any occurrence of $T(s, ct_1 \dots t_n)$ by $c_T(t_1, \dots, t_n, s)$ where $c_T$ is a new function symbol of arity $n + 1$. We then also add a new axiom:
$$\forall x_1 \dots x_n y.c_T(x_1, \dots, x_n, y) \leftrightarrow T(y, cx_1 \dots x_n).$$
Note that after performing this replacement the predicate $T$ may still occur in the FOL problem, e.g., a term $T(s, xt_1 \dots t_n)$ may occur. This optimisation is useful, because it simplifies the FOL terms and replaces the $T$ predicate with a specialised predicate for a constant. This makes it easier for the ATPs to handle the problem.
- For each occurrence of a constant $c$ with $n > 0$ arguments, i.e., each occurrence $ct_1 \dots t_n$ where $n > 0$ is maximal (there are no further arguments), we replace this occurrence with $c^n(t_1, \dots, t_n)$ where $c^n$ is a new $n$-ary function symbol. We then also add a new axiom:

  - $\forall x_1 \ldots x_n.P(c^n(x_1,\ldots,x_n)) \leftrightarrow P(cx_1 \ldots x_n)$ if (after replacement of all such occurrences) all terms of the form $c^n(t_1,\ldots,t_n)$ occur only as arguments of the predicate $P$, i.e., occur only as in $P(c^n(t_1,\ldots,t_n))$.
  - $\forall x_1 \ldots x_n.c^n(x_1,\ldots,x_n) = cx_1 \ldots x_n$ otherwise.

  This optimisation is similar to the optimisation originally described by Meng and Paulson in [MP08, Section 2.7].
- For any constant $c$ and $n > 0$, if all terms of the form $c^n(t_1,\ldots,t_n)$ occur only as arguments of $P$, then replace each occurrence of a term of the form $P(c^n(t_1,\ldots,t_n))$ by $c^n(t_1,\ldots,t_n)$.

### 5.6 Properties of the translation

In this section we briefly comment on the theoretical aspects of the translation. Further limitations of the whole approach will be mentioned in Section 9. The translation is neither sound nor complete. The lack of soundness is caused e.g. by the fact that we forget universe constraints on Type, the assumption of proof irrelevance, and the combination of omitting type guards for lifted-out lambda-abstractions with translating Coq equality to FOL equality. However, our experimental evaluation indicates that the translation is both sound and complete "enough" to be practically usable. Also, a "core" version of our translation is sound. A soundness proof and a more detailed discussion of the theoretical properties of a core version of our translation may be found in [Cza16].

Note that e.g. in the axiom added for lifted-out lambda-abstractions

$$\forall \vec{y_0}\vec{x_0}.\mathcal{F}_{\Gamma,\vec{x}:\vec{\tau}}(F\vec{y}\vec{x} \approx_{\Gamma,\vec{x}:\vec{\tau}} t)$$

we do not generate type guards for the free ($\vec{y_0}$) or bound ($\vec{x_0}$) variables of the lambda-expression. In practice, omitting these guards slightly improves the success rate of the ATPs without significantly affecting the reconstruction success rate. We conjecture that, ignoring other unsound features of the translation, omitting these guards is sound provided that the inductive Coq equality type `eq` is *not* translated to FOL equality. Note also that it is not sound (and our translation does not do it) to omit guards for the free variables of the term matched on in the `case` construct, even if Coq equality is not translated to FOL equality. For example, assume $I_0(c : \mathrm{Set} := c_0 : c)$ is in the global environment. With the guards omitted, for the `case`-expression $\texttt{case}(x, c, 0, c, c_0)$ we would add an axiom

$$\forall x.x = c_0 \wedge Fx = c_0$$

with $F$ a fresh first-order constant. This obviously leads to an inconsistency by substituting for $x$ two distinct constants $c_1, c_2$ such that $c_1 \neq c_2$ is provable.

In our translation we map Coq equality to FOL equality which is not sound in combination with omitting the guards for free variables. In particular, if a $\mathrm{CIC}_0$ problem contains a functional extensionality axiom then the generated FOL problem may be inconsistent, and in contrast to the inconsistencies that may result from omitting certain universe constraints, this inconsistency may be "easy enough" for the ATPs to derive. Our plugin has an option to turn on guard generation for free variables. See also [Cza16, Section 6].

## 6 Proof reconstruction

In this section we will discuss a number of existing Coq internal automation mechanisms that could be useful for proof reconstruction and finally introduce our combined proof reconstruction tactic.

The tactic `firstorder` is based on an extension of the contraction-free sequent calculus LJT of Dyckhoff [Dyc92] to first-order intuitionistic logic with inductive definitions [Cor03]. A decision procedure for intuitionistic propositional logic based on the system LJT is implemented in the tactic `tauto`. The tactic `firstorder` does not take into account many features of Coq outside of first-order logic. In particular, it does not fully axiomatise equality.

In general, the tactics based on extensions of LJT do mostly forward reasoning, i.e., they predominantly manipulate the hypotheses in the context to finally obtain the goal. Our approach is based more on an `auto`-type proof search which does mostly backward Prolog-style reasoning – modifying the goal by applying hypotheses from the context. The core of our search procedure may be seen as an extension of the Ben-Yelles algorithm [BY79,Hin97] to first-order intuitionistic logic with all connectives [Urz16, ZS16]. It is closely related to searching for $\eta$-long normal forms [BD05,Dow93]. Our implementation extends this core idea with various heuristics. We augment the proof search procedure with the use of existential metavariables like in `eauto`, a looping check, some limited forward reasoning, the use of the `congruence` tactic, and heuristic rewriting using equational hypotheses.

It is important to note that while the external ATPs we employ are classical and the translation assumes proof irrelevance, the proof reconstruction phase does not assume any additional axioms. We re-prove the theorems in the intuitionistic logic of Coq, effectively using the output of the ATPs merely as hints for our hand-crafted proof search procedure. Therefore, if the ATP proof is inherently classical then proof reconstruction will fail. Currently, the only information from ATP runs we use is a list of lemmas needed by the ATP to prove the theorem (these are added to the context) and a list of constant definitions used in the ATP proof (we try unfolding these constants and no others).

Another thing to note is that we do not use the information contained in the Coq standard library during reconstruction. This would not make sense for our evaluation of the reconstruction mechanism, since we try to re-prove the theorems from the Coq standard library. In particular, we do not use any preexisting hint databases available in Coq, not even the core database (for the evaluation we use the `auto` and `eauto` tactics with the `nocore` option, but in the final version of the reconstruction tactics we also use `auto` without this option). Also, we do not use any domain-specific decision procedures available as Coq tactics, e.g., `field`, `ring` or `omega`. Including such techniques in HOLyHammer did allow fast solving of many simple arithmetic problems [KU15a].

We now describe a simplification of our proof search procedure. We will treat the current proof state as a collection of judgements of the form $\Gamma \vdash G$ and describe the rules as manipulating a single such judgement. In a judgement $\Gamma \vdash G$ the term $G$ is the *goal* and $\Gamma$ is the *context* which is a list of *hypothesis* declarations of the form $H : A$. We use an informal notation for Coq terms similar to how they are displayed by Coq. For instance, by $\forall x : A, B$ we denote a dependent product. We write $\forall x, B$ when the type of $x$ is not essential. Note that in $\forall x, B$ the variable $x$ may be a proposition, so $\forall x, B$ may actually represent a logical implication $A \to B$

if $A$ is the omitted type of $x$ which itself has type Prop and $x$ does not occur in $B$. To avoid confusion with $=$ used to denote the equality inductive predicate in Coq, we use $\equiv$ as a metalevel symbol to denote identity of Coq terms. We use the notation $\Gamma; H : A$ to denote $\Gamma$ with $H : A$ inserted at some fixed position. By $\Gamma, H : A$ we denote the context $\Gamma$ with $H : A$ appended. We omit the hypothesis name $H$ when irrelevant. By $C[t]$ we denote an occurrence of a term $t$ in a term context $C$.

The proof search procedure applies the rules from Figure 1. An application of a rule of the form

$$\frac{\Gamma_1 \vdash G_1 \quad \ldots \quad \Gamma_n \vdash G_n}{\Gamma \vdash G}$$

replaces a judgement $\Gamma \vdash G$ in the current proof state by the judgements $\Gamma_1 \vdash G_1$, $\ldots$, $\Gamma_n \vdash G_n$. The notation $\texttt{tac}[\Gamma \vdash G]$ (resp. $\texttt{tac}(A)[\Gamma \vdash G]$) in a rule premise means applying the Coq tactic `tac` (with argument $A$) to the judgement $\Gamma \vdash G$ and making the judgements (subgoals) generated by the tactic be the premises of the rule. In a rule of the form e.g.

$$\frac{\Gamma; A' \vdash G}{\Gamma; A \vdash G}$$

the position in $\Gamma$ at which $A$ is inserted is implicitly assumed to be the same as the position at which $A'$ is inserted.

In Figure 1 the variables $?e_i$, $?e$ denote fresh existential metavariables of appropriate types. These metavariables need to be instantiated later by Coq's unification algorithm. In the rules (orsplit) and (exsimpl) the types of $x_1, \ldots, x_n$ are assumed not to be propositions. In the rule (exinst) the types of $x_1, \ldots, x_k$ are not propositions and either $k = n$ or the type of $x_{k+1}$ is a proposition. In the rule (orinst) the $x_{i_1}, \ldots, x_{i_m}$ are all those among $x_1, \ldots, x_n$ for which $T_{i_1}, \ldots, T_{i_m}$ are not propositions; and the index $k$ ranges over all $k \in \{1, \ldots, n\} \setminus \{i_1, \ldots, i_m\}$ (so that each $T_k$ is a proposition) – all judgements for any such $k$ are premises of the rule, not just a single one. Moreover, in these rules for any term $T$ by $T'$ we denote $T[?e_{i_1}/x_{i_1}, \ldots, ?e_{i_m}/x_{i_m}]$, and $T_{j_1}, \ldots, T_{j_{m:k}}$ are those among $T_1, \ldots, T_k$ which are propositions. In the (apply) and (invert) rules $P$ is an atomic proposition, i.e., a proposition which is not a dependent product, an existential, a disjunction or a conjunction. In the (destruct) rule $T$ is not a proposition.

The tactic `yapply` in rule (apply) works like `eapply` except that instead of simply unifying the goal with the target of the hypothesis, it tries unification modulo some simple equational reasoning. The idea of the `yapply` tactic is broadly similar to the smart matching of Matita [AT10], but our implementation is more heuristic and not based on superposition.

The tactic `yrewrite` in rule (rewrite) uses Coq's tactic `erewrite` to try to rewrite the hypothesis in the goal. If it fails to rewrite it directed from left to right, then it tries the other direction.

The rules in Figure 1 are divided into groups. The rules in each group are either applied with backtracking (marked by (b) in the figure), i.e., if applying one of the rules in the group to a judgement $\Gamma \vdash G$ does not ultimately succeed in finishing the proof then another of the rules in the group is tried on $\Gamma \vdash G$; or they are applied eagerly without backtracking (marked by (e) in the figure). There are also restrictions on when the rules in a given group may be applied. The rules in the group "Leaf tactics" must close a proof tree branch, i.e., they are applied only when they generate zero premises. The rules in the group "Final splitting" are applied only before

the "leaf tactics". The rules in the groups "Splitting", "Hypothesis simplification" and "Introduction" are applied whenever possible. The rules in the group "Proof search" constitute the main part of the proof search procedure. They are applied only when none of the rules in the groups "Splitting", "Hypothesis simplification" and "Introduction" can be applied. The rules in the group "Initial proof search" may only be applied after an application of (intro) followed by some applications of the rules in the "Splitting" and "Hypothesis simplification" groups. They are applied only if none of the rules in the groups "Splitting", "Hypothesis simplification" and "Introduction" can be applied.

The above description is only a readable approximation of what is actually implemented. Some further heuristics are used and more complex restrictions are put on what rules may be applied when. In particular, some loop checking (checking whether a judgement repeats) is implemented, the number of times a hypothesis may be used for rewriting is limited, and we also use heuristic rewriting in hypotheses and heuristic instantiation of universal hypotheses. Some heuristics we use are inspired by the `crush` tactic of Adam Chlipala [Chl13].

As mentioned before, our proof search procedure could be seen as an extension of a search for $\eta$-long normal forms for first-order intuitionistic logic using a Ben-Yelles-type algorithm [Urz16, ZS16]. As such it would be complete for the fragment of type theory "corresponding to" first-order logic, barring two simplifications we introduced to make it more practical. For the sake of efficiency, we do not backtrack on instantiations of existential metavariables solved by unification, and the rules (exinst) and (orinst) are not general enough. These cause incompleteness even for the first-order fragment, but this incompleteness does not seem to matter much in practice. The usual reasons why proof reconstruction fails is that either the proof is inherently classical, too deep, or uses too much rewriting which cannot be easily handled by our rewriting heuristics. It is left for future work to integrate rewriting into our proof search procedure in a more principled way.

The proof reconstruction phase in the `hammer` tactic uses a number of tactics derived from the procedure described above, with different depth limits, a bit different heuristics and rule application restrictions; plus a few other tactics, including Coq's `intuition`, `simpl`, `subst`, and heuristic constant unfolding. Various reconstruction tactics are tried in order with a time limit for each, until one of them succeeds (or none succeed – then the proof cannot be reconstructed).

It is important to note that no time limits are supposed to be present in the final proof scripts. The CoqHammer plugin shows which of the tactics succeeded, and the user is supposed to copy this tactic, replacing the `hammer` tactic invocation. The final reconstruction tactic does not rely on any time limits or make any calls to external ATPs. Its results are therefore completely reproducible on different machines, in contrast to the main `hammer` tactic itself.

## 7 Integrated Hammer and Evaluation

In this section we present the technique used to select the combination of strategies included in the integrated hammer and present an evaluation of the components as well as the final offered strategy.

The evaluation in this section will perform a push-button re-proving of Coq problems without using their proofs. In order for the evaluation of the system to be

$$\frac{\texttt{eauto}[\Gamma \vdash G]}{\Gamma \vdash G} \quad \frac{\texttt{congruence}[\Gamma \vdash G]}{\Gamma \vdash G} \quad \frac{\texttt{constructor}[\Gamma \vdash G]}{\Gamma \vdash G} \quad \frac{\texttt{easy}[\Gamma \vdash G]}{\Gamma \vdash G}$$

Leaf tactics (b)

$$\frac{\Gamma; A[?e_1/x_1, \ldots, ?e_n/x_n] \vdash G \quad \Gamma; B[?e_1/x_1, \ldots, ?e_n/x_n] \vdash G}{\Gamma; \forall x_1 \ldots x_n, A \vee B \vdash G} \text{ (orsplit)}$$

$$\frac{\Gamma; A[?e_1/x_1, \ldots, ?e_n/x_n] \vdash G \quad y \text{ fresh}}{\Gamma; \forall x_1 \ldots x_n, \exists y, A \vdash G} \text{ (exsimpl)}$$

Final splitting (e)

$$\frac{\texttt{destruct}(t)[\Gamma \vdash C[\texttt{match}\, t\, \texttt{with}\, b]]}{\Gamma \vdash C[\texttt{match}\, t\, \texttt{with}\, b]} \quad \frac{\texttt{destruct}(t)[\Gamma; C[\texttt{match}\, t\, \texttt{with}\, b] \vdash G]}{\Gamma; C[\texttt{match}\, t\, \texttt{with}\, b] \vdash G}$$

$$\frac{\Gamma \vdash G_1 \quad \Gamma \vdash G_2}{\Gamma \vdash G_1 \wedge G_2}$$

Splitting (e)

$$\frac{\Gamma; A; B \vdash G}{\Gamma; A; A \to B \vdash G} \quad \frac{\Gamma; \forall x_1 \ldots x_n, A; \forall x_1 \ldots x_n, B \vdash G}{\Gamma; \forall x_1 \ldots x_n, A \wedge B \vdash G}$$

$$\frac{\Gamma; \forall x_1 \ldots x_n, A \to B \to C \vdash G}{\Gamma; \forall x_1 \ldots x_n, A \wedge B \to C \vdash G} \quad \frac{\Gamma; \forall x_1 \ldots x_n, A \to C; \forall x_1 \ldots x_n, B \to C \vdash G}{\Gamma; \forall x_1 \ldots x_n, A \vee B \to C \vdash G}$$

$$\frac{\Gamma; A \vdash G \quad x \text{ fresh}}{\Gamma; \exists x, A \vdash G} \quad \frac{\Gamma; A \vdash G}{\Gamma; A; A \vdash G}$$

Hypothesis simplification (e)

$$\frac{\Gamma; x : A \vdash B}{\Gamma \vdash \forall x : A, B} \text{ (intro)}$$

Introduction (e)

$$\frac{}{\Gamma; \text{False} \vdash G} \quad \frac{}{\Gamma; G \vdash G} \quad \frac{\Gamma \vdash G[?e/x]}{\Gamma \vdash \exists x, G} \quad \frac{\Gamma \vdash G_1}{\Gamma \vdash G_1 \vee G_2} \quad \frac{\Gamma \vdash G_2}{\Gamma \vdash G_1 \vee G_2}$$

$$\frac{\texttt{yapply}(H)[\Gamma; H : A \vdash P]}{\Gamma; H : A \vdash P} \text{ (apply)} \quad \frac{\texttt{yrewrite}(H)[\Gamma; H : \forall x_1 \ldots x_n, A = B \vdash G]}{\Gamma; H : \forall x_1 \ldots x_n, A = B \vdash G} \text{ (rewrite)}$$

$$\frac{\Gamma; \forall x_{k+1} \ldots x_n, \exists x, A[?e_1/x_1, \ldots, ?e_k/x_k] \vdash G}{\Gamma; \forall x_1 \ldots x_k \ldots x_n, \exists x, A \vdash G} \text{ (exinst)}$$

$$\frac{\Gamma; T'_{j_1}, \ldots, T'_{j_{m:n}}, A'_1 \vdash G \quad \Gamma; T'_{j_1}, \ldots, T'_{j_{m:n}}, A'_2 \vdash G \quad \Gamma; T'_{j_1}, \ldots, T'_{j_{m:k-1}} \vdash T'_k}{\Gamma; \forall (x_1 : T_1) \ldots (x_n : T_n), A_1 \vee A_2 \vdash G} \text{ (orinst)}$$

Proof search (b)

$$\frac{\Gamma; \forall x_1 \ldots x_n, \text{False} \vdash \text{False}}{\Gamma; \forall x_1 \ldots x_n, \text{False} \vdash G}$$

$$\frac{\texttt{inversion}(H)[\Gamma; H : P \vdash G]}{\Gamma; H : P \vdash G} \text{ (invert)} \quad \frac{\texttt{destruct}(x)[\Gamma; x : T \vdash G]}{\Gamma; x : T \vdash G} \text{ (destruct)}$$

Initial proof search (b)

**Fig. 1** Simplified proof search rules

fair, we need ensure that no information from a proof is used in its re-proving, as well as that the actual strategy that is used by the whole system has been developed without the knowledge of the proofs being evaluated.

The system will be evaluated on the problems generated from all theorems in the Coq standard library of Coq version 8.5 (a version of the plugin works with Coq 8.6 and 8.7 as well). The problems were generated from the source code of the library, counting as theorems all definitions (introduced with any of `Lemma`, `Theorem`, `Corollary`, `Fact`, `Instance`, etc.) that were followed by the `Proof` keyword. The source code of the library was then modified to insert a hook to our hammer plugin after each `Proof` keyword. The plugin tries to re-prove the theorem using the Coq theorems accessible at the point when the statement of the theorem is introduced, using the three phases of premise selection, ATP invocation and proof reconstruction as described above.

This simulates how a hammer would be used in the development of the Coq standard library. In particular, when trying to re-prove a given theorem we use only the objects accessible in the Coq kernel at the moment the theorem statement is encountered by Coq. Of course, neither the re-proved theorem itself nor any theorems or definitions that depend on it are used. The number of problems obtained by automatically analysing the Coq standard library source code in the way described above is 9276. This differs significantly from the number of problems reported in [CK16]. There the theorems in the Coq standard library were extracted from objects of type Prop in the Coq kernel. Because of how the Coq module system works, there may be many Coq kernel objects corresponding to one definition in a source file (this is the case e.g. when using the `Include` command).

Furthermore, the problems are divided in a training set consisting of about 10% of the problems in the standard library and a validation set containing the remaining 90% of the problems. The training set is used to find a set of complementary strategies. Just like for the hammers for higher-order logic based systems and for Mizar a single best combination of the premise-selection algorithm, number of selected premises, and ATP run for a longer time is much weaker than running a few such combinations even for a shorter time. Contrary to existing hammer constructions [KU14,KU15c], we decided to include the reconstruction mechanism among the considered strategy parameters since generally reconstruction rates are lower and it could happen that proofs originating from a particular prover and number of premises would be too hard to reconstruct.

In our evaluation we used the following ATPs: E Prover version 1.9 [Sch13], Vampire version 4.0 [KV13] and Z3 version 4.0 [dMB08]. The evaluation was performed on a 48-core server with 2.2GHz AMD Opteron CPUs and 320GB RAM. Each problem was always assigned one CPU core. The two considered premise selection algorithms were asked for an ordering of premises, and all powers of two between 16 and 1024 were considered. Finally we considered both `firstorder` and `hrecon` reconstruction. Having evaluated all combinations of premise selection algorithms we ordered them in a greedy sequence: each following strategy is the one that adds most to the current selection of strategies. The first 14 strategies in the greedy sequence are presented in Table 1. The column "Solved" indicates the number of problems that were successfully solved by the given ATP with the given premise selection method and a given number of premises, *and* they could be reconstructed by the proof reconstruction procedure described in Section 6. The ATPs were run with a time limit of 30s. The maximum time limit for a single reconstruction tactic was 10s, depending

| Prover | Selection | Premises | Reconstruction | Solved (%) | Solved |
|---|---|---|---|---|---|
| Vampire | k-NN | 1024 | hrecon | 30.778 | 285 |
| Z3 | k-NN | 128 | hrecon | 37.473 | 347 |
| E-Prover | k-NN | 1024 | hrecon | 39.741 | 368 |
| Vampire | k-NN | 64 | hrecon | 40.929 | 379 |
| Z3 | n. Bayes | 32 | hrecon | 41.469 | 384 |
| Z3 | n. Bayes | 512 | hrecon | 42.009 | 389 |
| Z3 | n. Bayes | 128 | hrecon | 42.549 | 394 |
| E-Prover | n. Bayes | 256 | hrecon | 43.089 | 399 |
| Z3 | n. Bayes | 16 | hrecon | 43.521 | 403 |
| E-Prover | n. Bayes | 1024 | hrecon | 43.952 | 407 |
| Vampire | n. Bayes | 256 | hrecon | 44.276 | 410 |
| Z3 | k-NN | 64 | hrecon | 44.492 | 412 |
| Vampire | k-NN | 512 | hrecon | 44.708 | 414 |
| E-Prover | k-NN | 512 | firstorder | 44.924 | 416 |
| total | | | | 46.112 | 427 |

**Table 1** Success rates of the strategies on the training set in the greedy sequence order.

| Prover | Solved (%) | Solved |
|---|---|---|
| Vampire | 24.749 | 2292 |
| Z3 | 23.961 | 2219 |
| E-Prover | 23.162 | 2145 |
| Total | 26.747 | 2477 |

**Table 2** Prover results on the dependencies

on the tactic, as described in Section 6. No time limit was placed on the premise selection phase, however for goals with largest number of available premises the time does not exceed 0.5s for either of the considered algorithms. The first strategy that includes `firstorder` appears only on twelfth position in the greedy sequence and is therefore not used as part of the hammer. We show cumulative success rates to display the progress in the greedy sequence.

The results of the hammer strategies including the premise selection are very good in comparison with the results on the dependencies. Evaluating the translation with `hrecon` reconstruction is presented in Table 2. The results are significantly worse, mainly for two reasons. First, some dependencies are missing due to our way of recording them which does not take into account the delta-conversion. Secondly, the dependencies in proof terms often were added by automated tactics and are difficult to use for the ATPs. It is sometimes easier for the ATPs to actually prove the theorem from other lemmas in the library than from the original dependencies.

Given the common hardware configuration of computers today, we consider as the integrated system a combination of eight complementary strategies. The final results of the hammer including reconstruction on the validation set are presented in Table 3.

## 8 Case Studies

The intended use of a hammer is to prove relatively simple goals using available lemmas. The main problem a hammer system tries to solve is that of finding appropriate

| Prover | Selection | Premises | Reconstruction | Solved (%) | Solved |
|---|---|---|---|---|---|
| Vampire | k-NN | 1024 | hrecon | 28.816 | 2673 |
| E-Prover | k-NN | 1024 | hrecon | 25.593 | 2374 |
| Vampire | k-NN | 64 | hrecon | 25.367 | 2353 |
| Z3 | n. Bayes | 128 | hrecon | 24.299 | 2254 |
| Z3 | k-NN | 128 | hrecon | 24.127 | 2238 |
| Z3 | n. Bayes | 512 | hrecon | 23.243 | 2156 |
| Z3 | n. Bayes | 32 | hrecon | 19.028 | 1765 |
| E-Prover | n. Bayes | 256 | hrecon | 17.497 | 1623 |
| total | | | | 40.815 | 3786 |

**Table 3** The success rate of of the combination of strategies on the validation set

lemmas in a large collection and combining them to prove the goal. The advantage of a hammer over specialised domain-specific tactics is that it is a general system not depending on any domain knowledge. The hammer plugin may use all currently accessible lemmas, which includes lemmas proven earlier in a given formalization, not only the lemmas from the standard library or other predefined libraries.

It sometimes happens that the ATPs find proofs with fewer dependencies than the proofs in the standard library. One example is the Coq lemma `isometric_rotation`:

```
Lemma isometric_rotation : forall x1 y1 x2 y2 theta : R,
                           dist_euc x1 y1 x2 y2 =
                           dist_euc (xr x1 y1 theta) (yr x1 y1 theta)
                              (xr x2 y2 theta) (yr x2 y2 theta).
```

Its current proof in the Coq standard library uses 6 auxiliary facts and is performed using the following 7 line script:

```
unfold dist_euc; intros; apply Rsqr_inj;
   [ apply sqrt_positivity; apply Rplus_le_le_0_compat
      | apply sqrt_positivity; apply Rplus_le_le_0_compat
      | repeat rewrite Rsqr_sqrt;
         [ apply isometric_rotation_0
            | apply Rplus_le_le_0_compat
            | apply Rplus_le_le_0_compat ] ]; apply Rle_0_sqr
```

Multiple ATPs found a shorter proof which uses only two of the dependencies: the definition of euclidean distance and the lemma `isometric_rotation_0`. This suggests that the proof using the injectivity of square root is a detour, and indeed it is possible to write a much simpler valid Coq proof of the lemma using just the two facts used by the ATPs:

```
unfold dist_euc; intros;
   rewrite (isometric_rotation_0 _ _ _ _ theta); reflexivity.
```

The proof may also be reconstructed from the found dependencies inside Coq. This is also the case for all other examples presented in this section.

Also for some theorems the ATPs found proofs which use premises not present in the dependencies extracted from the proof of the theorems in the standard library. An example is the lemma `le_double` from `Reals.ArithProp`:

```
forall m n:nat, 2 * m <= 2 * n -> m <= n.
```

The proof of this lemma in the standard library uses 6 auxiliary lemmas and is performed by the following proof script (two lemmas not visible in the script were added by the tactic `prove_sup0`):

```
intros; apply INR_le.
assert (H1 := le_INR _ _ H).
do 2 rewrite mult_INR in H1.
apply Rmult_le_reg_l with (INR 2).
replace (INR 2) with 2; [ prove_sup0 | reflexivity ].
assumption.
```

ATPs found a proof of `le_double` using only 3 lemmas: `Arith.PeanoNat.Nat.le_0_l`, `Arith.Mult.mult_S_le_reg_l` and `Init.Peano.le_n`. None of these lemmas appear among the original dependencies.

Another example of hammer usage is a proof of the following fact:

```
forall m n k : nat, m * n + k = k + n * m.
```

This cannot be proven using the `omega` tactic because of the presence of multiplication. The tactic invocations `eauto with arith` or `firstorder with arith` do not work either. The hammer tool finds a proof using two lemmas from `Arith.PeanoNat.Nat`: `add_comm` and `mul_comm`.

A similar example is the goal

```
forall n : nat, 3 * 3 ^ n = 3 ^ (n + 1).
```

This goal cannot be solved using standard Coq tactics, including the tactic `omega`. Z3 with 128 preselected premises found a proof using the following lemmas from `Arith.PeanoNat.Nat`: `add_succ_r`, `le_0_l`, `pow_succ_r`, `add_0_r`. The proof may be reconstructed using `hexhaustive 0` or `hyelles 5` tactic invocations.

The next example of a goal solvable by the hammer involves operations on lists.

```
forall {A} (x : A) l1 l2 (P : A -> Prop),
  In x (l1 ++ l2) -> (forall y, In y l1 -> P y) ->
  (forall y, In y l2 -> P y) ->
  P x.
```

This goal cannot be solved (in reasonable time) using either `eauto with datatypes` or `firstorder with datatypes`. The hammer solves this goal using just one lemma: `Lists.List.in_app_iff`.

A similar example is

```
forall {A} (y1 y2 y3 : A) l l' z, In z l \/ In z l' ->
                                  In z (y1 :: y2 :: l ++ y3 :: l').
```

This goal cannot be solved using standard Coq tactics. Eprover with 512 preselected premises found a proof using two lemmas from `Lists.List`: `in_cons` and `in_or_app`.

The hammer is currently not capable of reasoning by induction, except in some very simple cases. Here is an example of a goal where induction is needed.

```
forall (A : Type) (P : A -> Prop) (a : A) (l l' : list A),
   List.Forall P l /\ List.Forall P l' /\ P a ->
   List.Forall P (l ++ a :: l').
```

This goal can be solved neither by standard Coq tactics nor by the hammer. However, it suffices to issue the ltac command `induction l` and the hammer can solve the resulting two subgoals, none of which could be solved by standard Coq tactics. The subgoal for induction base is:

```
A : Type
P : A -> Prop
a : A
============================
forall l' : list A, Forall P nil /\ Forall P l' /\ P a ->
                    Forall P (nil ++ a :: l')
```

The hammer solves this goal using the lemma `Forall_cons` from `Lists.List` and the definition of `++` (`Datatypes.app`). The subgoal for the induction step is:

```
A : Type
P : A -> Prop
a, a0 : A
l : list A
IHl : forall l' : list A, Forall P l /\ Forall P l' /\ P a ->
                          Forall P (l ++ a :: l')
============================
forall l' : list A, Forall P (a0 :: l) /\ Forall P l' /\ P a ->
                    Forall P ((a0 :: l) ++ a :: l')
```

The hammer solves this goal using the lemma `Forall_cons`, the inductive hypothesis (`IHl`) and the definition of `++`. Note that to reconstruct the ATP proof for this goal it is crucial that our reconstruction tactics can do inversion on inductive predicates in the context.

## 9 Limitations

In this section we briefly discuss the limitations of the current implementation of the CoqHammer tool. We also compare the hammer with the automation tactics already available in Coq.

The intended use of a hammer is to prove relatively simple goals using accessible lemmas. Currently, the hammer works best with lemmas from the Coq standard library. Testing with other libraries has been as yet very limited and the hammer tool may need some adjustments to achieve comparable success rates.

The hammer works best when the goal and the needed lemmas are "close to" first-order logic, as some more sophisticated features of the Coq logic are not translated adequately. In particular, when dependent types are heavily used in a development then the effectiveness of the hammer tool is limited. Specifically, case analysis over inhabitants of small propositional inductive types is not translated properly, and the fact that in Coq all inhabitants of Prop are also inhabitants of Type is not accounted for.

A small propositional inductive type is an inductive type in Prop having just one constructor and whose arguments are all non-informative (e.g. propositional). In Coq it is possible to perform case analysis over an inhabitant of a small propositional inductive type. This is frequently done when dealing with data structures where

dependent types are heavily exploited to capture the data structure invariants. Currently, all such pattern matches are translated to a fresh constant about which nothing is assumed. Therefore, the ATPs will fail to find a proof, except for trivial tautologies.

In Coq all propositions (inhabitants of Prop) are also types (inhabitants of Type). Therefore, type formers expecting types as arguments may sometimes be fed with propositions. For instance, one can use the pair type former as if it was a conjunction. Our translation heavily relies on the possibility of detecting whether a subterm is a proposition or not, in order to translate it to a FOL formula or a FOL term. The currently followed approach to proposition detection is relatively simplistic. For example, the pair type former should be translated to four different definitions, one taking in input two propositions, etc. Currently, only one definition is generated (the one with both arguments being of type Type).

In the context of code extraction the above two problems and some similar issues were handled in Pierre Letouzey's PhD thesis [Let04]. In [Let04] Coq terms are translated into an intermediate language where propositions are either removed from the terms or turned into unit types when used as types. It may be worthwhile to investigate if our translation could be factorized reusing the intermediate representation from [Let04]. If successful, this would be a better approach.

We leave it for future work to increase effectiveness of the hammer on a broader fragment of dependent type theory. In this regard our hammer is similar to hammers for proof assistants based on classical higher-order logic, which are less successful when the goal or the lemmas make heavy use of higher-order features.

The success of the `hammer` tactic is not guaranteed to be reproducible, because it relies on external ATPs and uses time limits during proof reconstruction. Indeed, small changes in the statement of the goal or a change of hardware may change the behaviour of the hammer. However, once a proof has been found and successfully reconstructed the user should replace the `hammer` tactic with an appropriate reconstruction tactic shown by the hammer in the response window. This reconstruction tactic does not depend on any time limits or external ATPs, so its success is independent of the current machine.

In comparison to the hammer, domain-specific decision procedures, e.g., the `omega` tactic, are generally faster and more consistently reliable for the goals they can solve. On the other hand, the proof terms generated by the `hammer` tactic are typically smaller and contain fewer dependencies which are more human-readable.

An advantage of Coq proof-search tactics like `auto`, `eauto` or `firstorder` is that they can be configured by the user by means of hint databases. However, they are in general much weaker than the hammer. The idea of a hammer is to be a strong general-purpose tactic not requiring much configuration by the user.

## 10 Conclusions and Future Work

We have developed a first whole hammer system for intuitionistic type theory. This involved proposing an approximation of the Calculus of Inductive Constructions, adapting premise selection to this foundation, developing a translation mechanism to untyped-first order logic, and proposing reconstruction mechanisms for the proofs found by the ATPs. We have implemented the hammer as a plugin for the Coq proof assistant and evaluated it on all the proofs in its standard library. The source code

of the plugin for Coq versions 8.5, 8.6 and 8.7, as well as all the experiments are available at:

http://cl-informatik.uibk.ac.at/cek/coqhammer/

The hammer is able to re-prove completely automatically 40.8% of the standard library proofs on a 8-CPU system in about 40 seconds. This success rate is already comparable to that offered by the first generations of hammer systems for HOL and Mizar and can already offer a huge saving of human work.

To our knowledge this is the first translation which is usable by hammers. Strictly speaking, our translation is neither sound nor complete. However, our experiments suggest that the encoding is "sound enough" to be usable and that it is particularly good for goals close to first-order logic. Moreover, a "core" version of the translation is in fact sound [Cza16].

There are many ways how the proposed work can be extended. First, the reconstruction mechanism currently is able to re-prove only 85.2% (4215 out of 4841) of the proofs founds by the ATPs, which is lower than that in other systems. The premise selection algorithms are not as precise as those involving machine learning algorithms tailored for particular logics. In particular, for similar size parts of the libraries almost the same premise selection algorithms used in HOLyHammer [KU14] or Isabelle/MaSh on parts of the Isabelle/HOL library [BGK+16], require on average 200–300 best premises to cover the dependencies, whereas in the Coq standard library on average 499-530 best premises are required.

The core of the hammer – the translation to FOL – could be improved to make use of more knowledge available in the prover in order to offer a higher success rate. It could also be modified to make it more effective on developments heavily using dependent types, and to more properly handle the advanced features of the Coq logic, possibly basing on some of the ideas in [Let04]. Finally, the dependencies extracted from the Coq proof terms do miss information used implicitly by the kernel, and are therefore not as precise as those offered in HOL-based systems.

In our work we have focused on the Coq standard library. Evaluations on a proof assistant standard library were common in many hammer comparisons, however this is rarely the level at which users are actually working, and looking at more advanced Coq libraries could give interesting insights for all components of a hammer. Since we focused on the standard library during development, it is likely that the effectiveness of the hammer is lower on libraries not similar to the standard library.

In particular, the Mathematical Components Library based on SSReflect [GM10] would be a particularly interesting example, as it heavily relies on unification hints to guide Coq automation. It has been used for example in the proofs of the four color theorem [Gon07] and the odd order theorem [GAA+13]. On a few manually evaluated examples, the success rate is currently quite low. It remains to be seen, whether a hammer can provide useful automation also for such developments, and how the currently provided translation could be optimized, to account for the more common use of dependent types. Lastly, we would like to extend the work to other systems based on variants of CIC and other interesting foundations, including Matita, Agda, and Idris.

**Acknowledgments** We thank the organisers of the First Coq Coding Sprint, especially Yves Bertot, for the help with implementing Coq export plugins. We wish to thank Thibault Gauthier for the first version of the Coq exported data, as as well as Claudio Sacerdoti-Coen for improvements to the exported data and

fruitful discussions on Coq proof reconstruction. This work has been supported by the Austrian Science Fund (FWF) grant P26201 and European Research Council (ERC) grant no. 714034 *SMART*.

## References

ACI+16. A. A. Alemi, F. Chollet, G. Irving, C. Szegedy, and J. Urban. DeepMath – Deep sequence models for premise selection. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NIPS 2016)*, pages 2235–2243, 2016.

ACN05. A. Abel, T. Coquand, and U. Norell. Connecting a logical framework to a first-order logic prover. In B. Gramlich, editor, *Frontiers of Combining Systems (FroCoS 2005)*, volume 3717 of *LNCS*, pages 285–301. Springer, 2005.

AFG+11. M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In J. Jouannaud and Z. Shao, editors, *Certified Programs and Proofs (CPP 2011)*, volume 7086 of *LNCS*, pages 135–150. Springer, 2011.

AHK+14. J. Alama, T. Heskes, D. Kühlwein, E. Tsivtsivadze, and J. Urban. Premise selection for mathematics by corpus analysis and kernel methods. *J. Autom. Reasoning*, 52(2):191–213, 2014.

ARS14. A. Asperti, W. Ricciotti, and C. Sacerdoti Coen. Matita tutorial. *J. Formalized Reasoning*, 7(2):91–199, 2014.

Asp00. D. Aspinall. Proof General: A generic tool for proof development. In S. Graf and M. I. Schwartzbach, editors, *Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, TACAS 2000*, volume 1785 of *LNCS*, pages 38–42. Springer, 2000.

AT07. A. Asperti and E. Tassi. Higher order proof reconstruction from paramodulation-based refutations: The unit equality case. In M. Kauers, M. Kerber, R. Miner, and W. Windsteiger, editors, *Mathematical Knowledge Management (MKM 2007)*, volume 4573 of *LNCS*, pages 146–160. Springer, 2007.

AT10. A. Asperti and E. Tassi. Smart matching. In *Intelligent Computer Mathematics, 10th International Conference, AISC 2010, 17th Symposium, Calculemus 2010, and 9th International Conference, MKM 2010, Paris, France, July 5-10, 2010. Proceedings*, pages 263–277, 2010.

BBF+15. J. C. Blanchette, S. Böhme, M. Fleury, S. J. Smolka, and A. Steckermeier. Semi-intelligible Isar proofs from machine-generated proofs. *J. Autom. Reasoning*, 2015.

BBG+15. G. Bancerek, C. Byliński, A. Grabowski, A. Korniłowicz, R. Matuszewski, A. Naumowicz, K. Pąk, and J. Urban. Mizar: State-of-the-art and beyond. In *Intelligent Computer Mathematics - International Conference, CICM 2015, Washington, DC, USA, July 13-17, 2015, Proceedings*, pages 261–279, 2015.

BC04. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

BD05. S. Broda and L. Damas. On long normal inhabitants of a type. *J. Log. Comput.*, 15(3):353–390, 2005.

BDN09. A. Bove, P. Dybjer, and U. Norell. A brief overview of Agda - A functional language with dependent types. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *LNCS*, pages 73–78. Springer, 2009.

Ber08. Y. Bertot. A short presentation of Coq. In O. A. Mohamed, C. A. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2008)*, volume 5170 of *LNCS*, pages 12–16. Springer, 2008.

BGK+16. J. C. Blanchette, D. Greenaway, C. Kaliszyk, D. Kühlwein, and J. Urban. A learning-based fact selector for Isabelle/HOL. *J. Autom. Reasoning*, 57(3):219–244, 2016.

BHdN02. M. Bezem, D. Hendriks, and H. de Nivelle. Automated proof construction in type theory using resolution. *J. Autom. Reasoning*, 29(3-4):253–275, 2002.

BKPU16. J. C. Blanchette, C. Kaliszyk, L. C. Paulson, and J. Urban. Hammering towards QED. *J. Formalized Reasoning*, 9(1):101–148, 2016.

Bla12. J. C. Blanchette. *Automatic Proofs and Refutations for Higher-Order Logic*. PhD thesis, Technische Universität München, 2012. http://www21.in.tum.de/~blanchet/phdthesis.pdf.

Bra13. E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013.

BW10. S. Böhme and T. Weber. Fast LCF-style proof reconstruction for Z3. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving (ITP 2010)*, volume 6172 of *LNCS*, pages 179–194. Springer, 2010.

BY79. C. Ben-Yelles. *Type-assignment in the lambda-calculus: syntax and semantics*. PhD thesis, Mathematics Department, University of Wales, Swansea, UK, 1979.

CH88. T. Coquand and G. P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

Chl13. A. Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.

CK16. Ł. Czajka and C. Kaliszyk. Goal translation for a hammer for Coq (extended abstract). In J. C. Blanchette and C. Kaliszyk, editors, *First International Workshop on Hammers for Type Theories (HaTT 2016)*, volume 210 of *EPTCS*, pages 13–20, 2016.

Coq16. Coq development team. *The Coq proof assistant reference manual*, 2016. Version 8.6.

Cor03. P. Corbineau. First-order reasoning in the calculus of inductive constructions. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs (TYPES 2003)*, volume 3085 of *LNCS*, pages 162–177. Springer, 2003.

Cza16. Ł. Czajka. A shallow embedding of pure type systems into first-order logic. Submitted. Available at http://www.mimuw.edu.pl/~lukaszcz/emb.pdf, 2016.

dMB08. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

dMKA+15. L. M. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The Lean theorem prover. In A. P. Felty and A. Middeldorp, editors, *International Conference on Automated Deduction (CADE 2015)*, volume 9195 of *LNCS*, pages 378–388. Springer, 2015.

dMS16. L. de Moura and D. Selsam. Congruence closure in intensional type theory. In N. Olivetti and A. Tiwari, editors, *International Joint Conference on Automated Reasoning, IJCAR 2016*, volume 9706 of *LNCS*. Springer, 2016.

Dow93. G. Dowek. A complete proof synthesis method for the cube of type systems. *J. Log. Comput.*, 3(3):287–315, 1993.

Dyc92. R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *J. Symb. Log.*, 57(3):795–807, 1992.

Fil13. J.-C. Filliâtre. One logic to use them all. In M. P. Bonacina, editor, *International Conference on Automated Deduction (CADE 2013)*, volume 7898 of *LNCS*, pages 1–20. Springer, 2013.

FK15. M. Färber and C. Kaliszyk. Random forests for premise selection. In C. Lutz and S. Ranise, editors, *Frontiers of Combining Systems (FroCoS 2015)*, volume 9322 of *LNCS*, pages 325–340, 2015.

FP13. J.-C. Filliâtre and A. Paskevich. Why3 - where programs meet provers. In M. Felleisen and P. Gardner, editors, *European Symposium on Programming (ESOP 2013)*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.

GAA+13. G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. L. Roux, A. Mahboubi, R. O'Connor, S. O. Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A machine-checked proof of the odd order theorem. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving (ITP 2013)*, volume 7998 of *LNCS*, pages 163–179. Springer, 2013.

GM10. G. Gonthier and A. Mahboubi. An introduction to small scale reflection in Coq. *J. Formalized Reasoning*, 3(2):95–152, 2010.

Gon07. G. Gonthier. The four colour theorem: Engineering of a formal proof. In D. Kapur, editor, *ASCM*, volume 5081 of *LNCS*, page 333. Springer, 2007.

GWR15. T. Gransden, N. Walkinshaw, and R. Raman. SEPIA: search for proofs using inferred automata. In A. P. Felty and A. Middeldorp, editors, *International Conference on Automated Deduction (CADE 2015)*, volume 9195 of *LNCS*, pages 246–255. Springer, 2015.

Har09. J. Harrison. HOL Light: An overview. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *LNCS*, pages 60–66. Springer, 2009.

HFH+09. M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The Weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, 2009.

Hin97. J. R. Hindley. *Basic Simple Type Theory*, volume 42 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1997.

Hur03. J. Hurd. First-order proof tactics in higher-order logic theorem provers. In M. Archer, B. D. Vito, and C. Muñoz, editors, *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003)*, number NASA/CP-2003-212448 in NASA Technical Reports, pages 56–68, 2003.

HUW14. J. Harrison, J. Urban, and F. Wiedijk. History of interactive theorem proving. In J. Siekmann, editor, *Handbook of the History of Logic vol. 9 (Computational Logic)*, pages 135–214. Elsevier, 2014.

HV11. K. Hoder and A. Voronkov. Sine qua non for large theory reasoning. In N. Bjørner and V. Sofronie-Stokkermans, editors, *23rd International Conference on Automated Deduction (CADE 2011)*, volume 6803 of *LNCS*, pages 299–314. Springer, 2011.

JKU14. S. Joosten, C. Kaliszyk, and J. Urban. Initial experiments with TPTP-style automated theorem provers on ACL2 problems. In F. Verbeek and J. Schmaltz, editors, *ACL2 Theorem Prover and its Applications (ACL2 2014)*, volume 152 of *EPTCS*, pages 77–85, 2014.

Jon72. K. S. Jones. A statistical interpretation of term specificity and its application in retrieval. *J. Documentation*, 28:11–21, 1972.

KHG13. E. Komendantskaya, J. Heras, and G. Grov. Machine learning in Proof General: Interfacing interfaces. In C. Kaliszyk and C. Lüth, editors, *User Interfaces for Theorem (UITP 2012)*, volume 118 of *EPTCS*, pages 15–41, 2013.

KMU14. C. Kaliszyk, L. Mamane, and J. Urban. Machine learning of Coq proof guidance: First experiments. In T. Kutsia and A. Voronkov, editors, *Symbolic Computation in Software Science (SCSS 2014)*, volume 30 of *EPiC*, pages 27–34. EasyChair, 2014.

KU13a. C. Kaliszyk and J. Urban. PRocH: Proof reconstruction for HOL Light. In M. P. Bonacina, editor, *International Conference on Automated Deduction (CADE 2013)*, volume 7898 of *LNCS*, pages 267–274. Springer, 2013.

KU13b. C. Kaliszyk and J. Urban. Stronger automation for Flyspeck by feature weighting and strategy evolution. In J. C. Blanchette and J. Urban, editors, *Proof Exchange for Theorem Proving (PxTP 2013)*, volume 14 of *EPiC*, pages 87–95. EasyChair, 2013.

KU14. C. Kaliszyk and J. Urban. Learning-assisted automated reasoning with Flyspeck. *J. Autom. Reasoning*, 53(2):173–213, 2014.

KU15a. C. Kaliszyk and J. Urban. HOL(y)Hammer: Online ATP service for HOL Light. *Mathematics in Computer Science*, 9(1):5–22, 2015.

KU15b. C. Kaliszyk and J. Urban. Learning-assisted theorem proving with millions of lemmas. *J. Symbolic Computation*, 69:109–128, 2015.

KU15c. C. Kaliszyk and J. Urban. MizAR 40 for Mizar 40. *J. Autom. Reasoning*, 55(3):245–256, 2015.

KUV15. C. Kaliszyk, J. Urban, and J. Vyskočil. Efficient semantic features for automated reasoning over large theories. In Q. Yang and M. Wooldridge, editors, *International Joint Conference on Artificial Intelligence (IJCAI 2015)*, pages 3084–3090. AAAI Press, 2015.

KV13. L. Kovács and A. Voronkov. First-order theorem proving and Vampire. In N. Sharygina and H. Veith, editors, *Computer-Aided Verification (CAV 2013)*, volume 8044 of *LNCS*, pages 1–35. Springer, 2013.

KvLT+12. D. Kühlwein, T. van Laarhoven, E. Tsivtsivadze, J. Urban, and T. Heskes. Overview and evaluation of premise selection techniques for large theory mathematics. In B. Gramlich, D. Miller, and U. Sattler, editors, *International Joint Conference on Automated Reasoning (IJCAR 2012)*, volume 7364 of *LNCS*, pages 378–392. Springer, 2012.

Lau16. J. Laurent. Suggesting relevant lemmas by learning from succesful proofs. Technical report, École normale supérieure, 2016. Internship Report.

Let04. P. Letouzey. *Programmation fonctionnelle certifiée : L'extraction de programmes dans l'assistant Coq. (Certified functional programming : Program extraction within Coq proof assistant)*. PhD thesis, University of Paris-Sud, Orsay, France, 2004.

MP08. J. Meng and L. C. Paulson. Translating higher-order clauses to first-order clauses. *Journal of Automated Reasoning*, 40(1):35–60, 2008.

MP09. J. Meng and L. C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *J. Applied Logic*, 7(1):41–57, 2009.

PB10. L. C. Paulson and J. Blanchette. Three years of experience with Sledgehammer, a practical link between automated and interactive theorem provers. In *8th IWIL*, 2010.

PS07. L. C. Paulson and K. W. Susanto. Source-level proof reconstruction for interactive theorem proving. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2007)*, volume 4732 of *LNCS*, pages 232–245. Springer, 2007.

Sch13. S. Schulz. System description: E 1.8. In K. L. McMillan, A. Middeldorp, and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence (LPAR 2013)*, volume 8312 of *LNCS*, pages 735–743. Springer, 2013.

SLKN01. S. Schmitt, L. Lorigo, C. Kreitz, and A. Nogin. Jprover : Integrating connection-based theorem proving into interactive proof assistants. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Automated Reasoning, First International Joint Conference, IJCAR 2001, Siena, Italy, June 18-23, 2001, Proceedings*, volume 2083 of *Lecture Notes in Computer Science*, pages 421–426. Springer, 2001.

SN08. K. Slind and M. Norrish. A brief overview of HOL4. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *TPHOLs 2008*, volume 5170 of *LNCS*, pages 28–32. Springer, 2008.

Sut10. G. Sutcliffe. The TPTP World - Infrastructure for automated reasoning. In E. Clarke and A. Voronkov, editors, *LPAR-16*, number 6355 in LNAI, pages 1–12. Springer, 2010.

TS98. T. Tammet and J. M. Smith. Optimized encodings of fragments of type theory in first-order logic. *J. Log. Comput.*, 8(6):713–744, 1998.

Urb04. J. Urban. MPTP - Motivation, Implementation, First Experiments. *J. Autom. Reasoning*, 33(3-4):319–339, 2004.

Urz16. P. Urzyczyn. Intuitionistic games: Determinacy, completeness, and normalization. *Studia Logica*, 104(5):957–1001, 2016.

US10. J. Urban and G. Sutcliffe. Automated reasoning and presentation support for formalizing mathematics in Mizar. In S. Autexier, J. Calmet, D. Delahaye, P. D. F. Ion, L. Rideau, R. Rioboo, and A. P. Sexton, editors, *Intelligent Computer Mathematics (CICM 2010)*, volume 6167 of *LNCS*, pages 132–146, 2010.

Wie07. F. Wiedijk. Mizar's soft type system. In *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, pages 383–399, 2007.

WPN08. M. Wenzel, L. C. Paulson, and T. Nipkow. The Isabelle framework. In O. A. Mohamed, C. A. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2008)*, volume 5170 of *LNCS*, pages 33–38. Springer, 2008.

ZS16. M. Zielenkiewicz and A. Schubert. Automata theory approach to predicate intuitionistic logic. In *Logic-Based Program Synthesis and Transformation - 26th International Symposium, LOPSTR 2016, Revised Selected Papers*, pages 345–360, 2016.

## Appendix A The Results of all Strategies on the Training Set

| Prover | Selection | Premises | Reconstruction | Solved% | Solved |
|---|---|---|---|---|---|
| Vampire | k-NN | 1024 | hrecon | 30.778 | 285 |
| Vampire | n. Bayes | 1024 | hrecon | 30.778 | 285 |
| Vampire | n. Bayes | 256 | hrecon | 30.454 | 282 |
| Vampire | k-NN | 512 | hrecon | 29.590 | 274 |
| Vampire | k-NN | 128 | hrecon | 29.482 | 273 |
| Vampire | n. Bayes | 128 | hrecon | 29.266 | 271 |
| Vampire | k-NN | 256 | hrecon | 29.158 | 270 |
| Vampire | n. Bayes | 512 | hrecon | 28.726 | 266 |
| E-Prover | k-NN | 1024 | hrecon | 27.538 | 255 |
| E-Prover | n. Bayes | 1024 | hrecon | 26.890 | 249 |
| Z3 | n. Bayes | 256 | hrecon | 26.890 | 249 |
| Vampire | n. Bayes | 64 | hrecon | 26.674 | 247 |
| Z3 | k-NN | 256 | hrecon | 26.566 | 246 |
| Z3 | k-NN | 128 | hrecon | 26.458 | 245 |
| Vampire | k-NN | 64 | hrecon | 26.134 | 242 |
| Z3 | n. Bayes | 128 | hrecon | 26.026 | 241 |
| Z3 | k-NN | 512 | hrecon | 25.918 | 240 |
| Z3 | n. Bayes | 512 | hrecon | 25.918 | 240 |
| Z3 | n. Bayes | 1024 | hrecon | 24.946 | 231 |
| E-Prover | n. Bayes | 64 | hrecon | 24.838 | 230 |
| Z3 | k-NN | 1024 | hrecon | 23.974 | 222 |
| E-Prover | n. Bayes | 128 | hrecon | 23.434 | 217 |
| Z3 | n. Bayes | 64 | hrecon | 23.434 | 217 |
| E-Prover | k-NN | 64 | hrecon | 22.786 | 211 |
| Z3 | k-NN | 64 | hrecon | 22.570 | 209 |
| E-Prover | k-NN | 128 | hrecon | 22.030 | 204 |
| Vampire | k-NN | 32 | hrecon | 21.382 | 198 |
| E-Prover | k-NN | 512 | hrecon | 20.950 | 194 |
| E-Prover | k-NN | 32 | hrecon | 20.842 | 193 |
| Vampire | n. Bayes | 32 | hrecon | 20.626 | 191 |
| E-Prover | n. Bayes | 256 | hrecon | 20.518 | 190 |
| E-Prover | n. Bayes | 32 | hrecon | 20.194 | 187 |
| Z3 | n. Bayes | 32 | hrecon | 19.546 | 181 |
| E-Prover | k-NN | 256 | hrecon | 19.438 | 180 |
| Z3 | k-NN | 32 | hrecon | 19.438 | 180 |
| Z3 | n. Bayes | 16 | hrecon | 17.063 | 158 |
| E-Prover | n. Bayes | 512 | hrecon | 16.739 | 155 |
| Vampire | n. Bayes | 16 | hrecon | 16.739 | 155 |
| E-Prover | n. Bayes | 16 | hrecon | 16.091 | 149 |
| Vampire | k-NN | 1024 | firstorder | 15.551 | 144 |
| Vampire | k-NN | 16 | hrecon | 15.335 | 142 |
| Vampire | k-NN | 512 | firstorder | 15.227 | 141 |
| Vampire | n. Bayes | 1024 | firstorder | 15.119 | 140 |
| Z3 | k-NN | 16 | hrecon | 15.011 | 139 |

| | | | | | |
|---|---|---|---|---|---|
| Vampire | k-NN | 256 | firstorder | 14.579 | 135 |
| E-Prover | k-NN | 16 | hrecon | 14.579 | 135 |
| Vampire | n. Bayes | 512 | firstorder | 14.363 | 133 |
| Vampire | k-NN | 128 | firstorder | 14.147 | 131 |
| Vampire | n. Bayes | 256 | firstorder | 14.147 | 131 |
| Z3 | k-NN | 256 | firstorder | 13.931 | 129 |
| Z3 | n. Bayes | 512 | firstorder | 13.823 | 128 |
| Z3 | k-NN | 1024 | firstorder | 13.715 | 127 |
| Vampire | n. Bayes | 128 | firstorder | 13.607 | 126 |
| Z3 | n. Bayes | 1024 | firstorder | 13.607 | 126 |
| Vampire | k-NN | 64 | firstorder | 13.499 | 125 |
| Z3 | k-NN | 512 | firstorder | 13.499 | 125 |
| E-Prover | k-NN | 1024 | firstorder | 13.283 | 123 |
| Vampire | n. Bayes | 64 | firstorder | 13.175 | 122 |
| Z3 | k-NN | 128 | firstorder | 13.175 | 122 |
| Z3 | n. Bayes | 256 | firstorder | 13.175 | 122 |
| E-Prover | n. Bayes | 1024 | firstorder | 12.743 | 118 |
| Z3 | n. Bayes | 128 | firstorder | 12.635 | 117 |
| Z3 | n. Bayes | 64 | firstorder | 12.095 | 112 |
| Z3 | k-NN | 64 | firstorder | 11.771 | 109 |
| Vampire | n. Bayes | 32 | firstorder | 11.447 | 106 |
| Vampire | k-NN | 32 | firstorder | 11.339 | 105 |
| E-Prover | n. Bayes | 128 | firstorder | 11.015 | 102 |
| E-Prover | k-NN | 128 | firstorder | 10.907 | 101 |
| E-Prover | n. Bayes | 64 | firstorder | 10.907 | 101 |
| E-Prover | n. Bayes | 32 | firstorder | 10.691 | 99 |
| Z3 | k-NN | 32 | firstorder | 10.691 | 99 |
| E-Prover | k-NN | 32 | firstorder | 10.583 | 98 |
| Z3 | n. Bayes | 32 | firstorder | 10.475 | 97 |
| E-Prover | k-NN | 64 | firstorder | 10.259 | 95 |
| Z3 | n. Bayes | 16 | firstorder | 10.259 | 95 |
| E-Prover | k-NN | 256 | firstorder | 9.935 | 92 |
| E-Prover | k-NN | 512 | firstorder | 9.395 | 87 |
| E-Prover | n. Bayes | 16 | firstorder | 9.395 | 87 |
| Vampire | n. Bayes | 16 | firstorder | 9.395 | 87 |
| E-Prover | n. Bayes | 256 | firstorder | 9.179 | 85 |
| Z3 | k-NN | 16 | firstorder | 8.855 | 82 |
| Vampire | k-NN | 16 | firstorder | 8.531 | 79 |
| E-Prover | n. Bayes | 512 | firstorder | 8.099 | 75 |
| E-Prover | k-NN | 16 | firstorder | 7.883 | 73 |
| total | | | | 46.112 | 427 |

## Appendix B The proof reconstruction tactics

The core of our proof search tactic `yelles` is presented in Figure 3 in an ltac-like pseudocode. Figure 4 presents most of the reconstruction tactics that the hammer plugin tries in sequence with a fixed time limit for each. Figure 2 presents the tactic `ysplit` used for goal splitting and the tactic `isolve` used at the leaves of the proof search tree.

We proceed to describe the tactics in more detail. The notations and terminology are as in Section 6.

The tactic `ysplit` destructs arguments to a match. Also, if in the current proof state $\Gamma \vdash G$ the goal is a conjunction $G \equiv A \wedge B$, then `ysplit` splits the state into $\Gamma \vdash A$ and $\Gamma, H : A \vdash B$ with $H$ fresh. The hypothesis $H$ is then simplified using the tactic `simp_hyp` (not shown in any of the figures). The hypothesis simplification consists of, among other things, applying some of the left rules of Dyckhoff's system LJT [Dyc92] to $H$, i.e., in essence applying some forward reasoning involving $H$. The simplifications performed by the tactic `simp_hyp` invoked on a hypothesis $H : A$ are as follows.

- If $A$ is a proposition then for each $H' : A \to B$ in the context replace $H'$ with $H'' : B$.
- If $A$ has the form $\forall x_1 \ldots x_n, A_1 \wedge A_2$ then replace $H$ with $H_1 : \forall x_1 \ldots x_n, A_1$ and $H_2 : \forall x_1 \ldots x_n, A_2$, and simplify $H_1$ and $H_2$ recursively.
- If $A$ has the form $\forall x_1 \ldots x_n, A_1 \wedge A_2 \to C$ then replace $H$ with
$$H' : \forall x_1 \ldots x_n, A_1 \to A_2 \to C$$
and simplify $H'$ recursively.
- If $A$ has the form $\forall x_1 \ldots x_n, A_1 \vee A_2 \to C$ then replace $H$ with
$$H_1 : \forall x_1 \ldots x_n, A_1 \to C$$
and
$$H_2 : \forall x_1 \ldots x_n, A_2 \to C,$$
and simplify $H_1$ and $H_2$ recursively.
- If $A$ has the form $\exists x : T, C$ the replace $H$ with $x : T$ and $H' : C$ (assuming $x$ is fresh), and simplify $H'$ recursively.
- If $A$ is a simple tautology like $X = X$ or $X \to X$ then remove $H$.
- If $H' : A$ occurs in the context with $H' \neq H$ then remove $H$.

The tactic `simp_hyps` used in the tactic `isolve` calls `simp_hyp` repeatedly on every hypothesis in the context.

The tactic `trysolve` tries to solve the current goal using a combination of the tactics `eauto` and `congruence`. The tactic `isolve` invokes `trysolve` after first trying to split the goal and simplify the hypotheses. In addition to the tactics `ysplit` and `simp_hyps` it uses the following tactics:

- `orsplit` splits the state $\Gamma, H : \forall x_1 \ldots x_n, A_1 \vee A_2 \vdash G$, where none of the types of $x_1, \ldots, x_n$ is a proposition, into the states
$$\Gamma, H_1 : A_1[?e_1/x_1, \ldots, ?e_n/x_n] \vdash G$$
and
$$\Gamma, H_2 : A_2[?e_1/x_1, \ldots, ?e_n/x_n] \vdash G$$
where $?e_1, \ldots, ?e_n$ are fresh existential metavariables of appropriate types,

```
Ltac ysplit :=
  match goal with
    | [ ⊢ ?A ∧ _ ] ⇒
      cut A; [ let H := fresh "H" in
               intro H; split; [ exact H | simp_hyp H ] | idtac ]
    | [ ⊢ context[match ?X with _ ⇒ _ end] ] ⇒ destruct X eqn:?
    | [ H : context[match ?X with _ ⇒ _ end] ⊢ _ ] ⇒ destruct X eqn:?
  end.

Ltac trysolve :=
  eauto 2 with nocore; try solve [ constructor ]; try subst;
  match goal with
    | [ ⊢ ?t = ?u ] ⇒ try solve [ hnf in *; congruence 8 ]
    | [ ⊢ ?t ≠ ?u ] ⇒ try solve [ hnf in *; congruence 8 ]
    | [ ⊢ False ] ⇒ try solve [ hnf in *; congruence 8 ]
    | _ ⇒ idtac
  end.

Ltac isolve :=
  let msplit splt simp :=
      simp tt;
      repeat (progress splt tt; simp tt).
  in
  let rec msolve splt simp :=
      msplit splt simp;
      lazymatch goal with
        | [ H : False ⊢ _ ] ⇒ exfalso; exact H
        | [ ⊢ _ ∨ _ ] ⇒
          trysolve;
            try solve [ left; msolve splt simp | right; msolve splt simp ]
        | [ ⊢ ∃ x, _ ] ⇒
          trysolve; try solve [ eexists; msolve splt simp ]
        | _ ⇒
          trysolve
      end
  in
  msolve
    ltac:(fun _ ⇒ first [ ysplit | orsplit ])
    ltac:(fun _ ⇒ intros; simp_hyps; repeat exsimpl).
```

**Fig. 2** Tactics `ysplit` and `isolve`.

– `exsimpl` replaces any hypothesis of the form $H : \forall x_1 \ldots x_n, \exists y, A$ where none of the types of $x_1, \ldots, x_n$ is a proposition with

$$H' : A[?e_1/x_1, \ldots, ?e_n/x_n]$$

where $?e_1, \ldots, ?e_n$ are fresh existential metavariables of appropriate types.

The tactic `yelles` shown in Figure 3 implements the core of our `eauto`-type proof search procedure. The argument *defs* is a list of definitions to try unfolding on, $n$ is the desired search depth, *rtrace* is a list of hypotheses with which rewriting was tried since the last context modification, and *gtrace* is a list of goals encountered since the last context modification. At the leaves of the search tree, i.e., when the depth $n$ is 0, the tactic `isolve` is used. Otherwise, the tactic `yelles` checks if the current goal occurs in the list *gtrace* and fails if it does. If the goal does not occur in *gtrace* then nondeterministically (i.e. with backtracking) one of the following is tried.

– If there is a hypothesis $H$ : False in the context then solve the current goal using the `exfalso` tactic.

- If the goal is one of the hypotheses then use the `assumption` tactic.
- If the goal has the form $A \to B$ and a hypothesis $H : A$ occurs in the context, then change the goal to $B$ and call the tactic `yelles` with the same arguments, failing the whole procedure (i.e. without trying any further actions listed here) if it fails.
- If the goal has the form $\forall x, A$ or $A \wedge B$, or the goal or one of the hypotheses contains a match on a term with no bound variables, then call the tactic `doyelles` (described below), failing the whole procedure if it fails.
- If the goal is an existential $\exists x, A$ then try instantiating it with a fresh existential metavariable and calling `yelles` recursively with the same depth.
- If there is a hypothesis $H : \forall x_1 \ldots x_n, A = B$ in the context which does not occur in the list *rtrace*, then try rewriting the goal with $H$ calling `yelles` recursively with depth $n - 1$. More precisely, the tactic `yrewrite` used to rewrite using $H$ is defined as:
  `Ltac` *yrewrite* $H :=$ (`erewrite` $H$ `by isolve`) || (`erewrite` $\leftarrow H$ `by isolve`).
- Try applying a hypothesis from the context using the `yapply` tactic, and then calling `yelles` recursively with depth $n - 1$. The tactic `yapply` (not shown in the figures) works like `eapply` except that instead of simply unifying the goal with the target of the hypothesis, it tries unification modulo some simple equational reasoning. The idea of the `yapply` tactic is broadly similar to the smart matching of Matita [AT10], but our implementation is more heuristic and not based on superposition.
- Try solving the current goal with the `isolve` tactic.
- If there is a hypothesis $H : \forall x_1 \ldots x_k \ldots x_n, \exists y, A$ such that the types of $x_1, \ldots, x_k$ are not propositions and either $k = n$ or the type of $x_{k+1}$ is a proposition, then replace $H$ with $H' : \forall x_{k+1} \ldots x_n, \exists y, A[?e_1/x_1, \ldots, ?e_k/x_k]$ where $?e_1, \ldots, ?e_k$ are fresh existential metavariables of appropriate types, and call `yelles` recursively with depth $n - 1$, resetting *rtrace* and *gtrace* to empty lists. The hypothesis $H'$ is introduced into the context using the tactic `yintro` described below.
- If the goal is a disjunction then try applying the left or the right disjunction introduction rule, and then calling `yelles` recursively with depth $n - 1$.
- If there is a hypothesis $H : \forall (x_1 : T_1) \ldots (x_n : T_n), A \vee B$ in the context, then use the tactic `orinst` on $H$ and call `yelles` recursively with depth $n - 1$, resetting *rtrace* and *gtrace*. The tactic `orinst` (not shown in the figures) removes $H$ from the context $\Gamma$, yielding $\Gamma'$, and then splits the proof state $\Gamma' \vdash G$ into the following where $T_{i_1}, \ldots, T_{i_m}$ are those among $T_1, \ldots, T_n$ which are not propositions, and $?e_{i_1}, \ldots, ?e_{i_m}$ are fresh existential metavariables of appropriate types, and for any term $C$ we use $C'$ to denote $C[?e_{i_1}/x_{i_1}, \ldots, ?e_{i_m}/x_{i_m}]$, and $T_{j_1}, \ldots, T_{j_{m_k}}$ are those among $T_1, \ldots, T_k$ which are propositions, and $H_{j_1}, \ldots, H_{j_{n_k}}$ are fresh hypothesis names.
  - $\Gamma', H_{j_1} : T'_{j_1}, \ldots, H_{j_{m_{k-1}}} : T'_{j_{m_{k-1}}} \vdash T'_k$ for each $k$ such that $T_k$ is a proposition.
  - $\Gamma', H_{j_1} : T'_{j_1}, \ldots, H_{j_{m_n}} : T'_{j_{m_n}}, A' \vdash G'$.
  - $\Gamma', H_{j_1} : T'_{j_1}, \ldots, H_{j_{m_n}} : T'_{j_{m_n}}, B' \vdash G'$.

The tactic `doyelles` in Figure 3 first repeatedly invokes the tactic `yintro` (described below), then repeatedly invokes `cbn` and `ysplit`, and finally either calls `isolve` if the depth $n$ is 0, or otherwise tries the first of the following that does not fail.

- Invoke the tactic `yelles` with depth $n$.

- Destruct a non-propositional hypothesis and invoke `doyelles` recursively with depth $n-1$.
- Do inversion on a propositional atomic hypothesis and invoke `doyelles` recursively with depth $n-1$.
- If the goal is an equation $A = B$ then try destructing $A$ and/or $B$, and invoke `yelles` with depth $n-1$.
- If there exists a hypothesis with target False, then use `exfalso` and invoke `yelles` with depth $n$.

The tactic `yintro` used in the tactics `yelles` and `doyelles` works as follows. If the goal has the form $A \to B$ then it changes the goal to $B$ and adds a new hypothesis $H : A$ to the context, unless $A$ is a proposition and it already occurs in the context. Next, the hypothesis is simplified using the `simp_hyp` tactic. Additionally, if $A$ is a proposition and the total number of hypotheses does not exceed 8 then the following actions are performed.

- For each $H' : A' \to B'$ in the context try unifying $A$ and $A'$ and then if successful replace $H'$ with $H'' : B$ where $B$ is $B'$ with some of the existential metavariables instantiated by the unification of $A$ and $A'$.
- If $A = B \to C$ with $B$ an atom and there is a hypothesis $H' : B'$ in the context such that $B$ and $B'$ unify, then unify them and replace $H$ with $H'' : C'$ where $C'$ is $C$ with some of the existential metavariables instantiated by the unification of $B$ and $B'$.
- If $A$ is an equality $A_1 = A_2$ then try rewriting $H$ in every other hypothesis *H1*, using the following ltac code:
  (`rewrite` $H$ `in` *H1* `by isolve`) || (`rewrite` $\leftarrow$ $H$ `in` *H1* `by isolve`).

The tactic `dsolve` (not shown in the figures) does one of the following.

- If the goal is a not proposition then try the tactic `auto` and the ltac code:
  `try solve` [ `repeat constructor` ].
- If the goal is a proposition then try the tactics `auto` and `easy`.

Finally, the tactic `yellesd` in Figure 3 implements our proof search procedure by first normalising the goal using the call-by-name strategy (tactic `cbn`), then invoking `doyelles`, unshelving the shelved goals (these are typically the types of existential metavariables that were not instantiated by unification) and solving them using `dsolve`.

Figure 4 presents most of the tactics that are tried during the proof reconstruction phase. The tactic invocation `hinit` *hyps lems defs* used in Figure 4 removes the hypotheses not present in *hyps*, adds *lems* to the context and tries unfolding the definitions in *defs*, depending on some heuristic (whether they unfold to a term with a "simple" form). The tactic `gunfolding` tries unfolding definitions, using slightly less stringent heuristic criteria than `hinit`. The tactic `generalizing` repeatedly removes a hypothesis $H : A$ with $A$ a proposition, changing the goal $G$ to $A \to G$. The rest of the code in Figure 4 depends on the tactics already described and should be self-explanatory.

During proof reconstruction the following tactic invocations are tried in the given order with a time limit for each shown in brackets. Proof reconstruction fails if none of the listed tactics manages to solve the goal within the given time limit. If one of the listed tactics fails, it typically fails quickly and the next tactic is tried. In principle, the tactics could be run in parallel.

```
Ltac yelles defs n rtrace gtrace :=
  lazymatch n with
    | O ⇒ solve [ isolve ]
    | S ?k ⇒
      let G := getgoal in
      notInList G gtrace;
      match goal with
        | [ H : False ⊢ _ ] ⇒ exfalso; exact H
        | [ H : G ⊢ _ ] ⇒ assumption
        | [ H : ?P ⊢ ?P → ?Q ] ⇒
          (let H1 := fresh "H" in intro H1; try clear H1;
           move H at bottom; yelles defs n rtrace gtrace) || fail 1
        | [ ⊢ ∀ x, _ ] ⇒ doyelles defs n || fail 1
        | [ ⊢ _ ∧ _ ] ⇒ doyelles defs n || fail 1
        | [ ⊢ context[match ?X with _ ⇒ _ end] ] ⇒ doyelles defs n || fail 1
        | [ H : context[match ?X with _ ⇒ _ end] ⊢ _ ] ⇒ doyelles defs n || fail 1
        | [ ⊢ ∃ x, _ ] ⇒
          eexists; yelles defs n rtrace (gtrace, G)
        | [ H : ∀ x1 ... xn, _ = _ ⊢ _ ] ⇒
          notInList H rtrace;
          yrewrite H; yelles defs k (rtrace, H) (gtrace, G)
        | [ H : _ ⊢ _ _ ] ⇒
          yapply H; yelles defs k rtrace (gtrace, G)
        | [ ⊢ _ ] ⇒
          solve [ isolve ]
        | [ H : ∀ x1 ... xn, ∃ e, _ ⊢ _ ] ⇒
          einst H; clear H; yintro; yelles defs k Empty Empty
        | [ ⊢ _ ∨ _ ] ⇒
          (left; yelles defs k rtrace (gtrace, G))
            ||
          (right; yelles defs k rtrace (gtrace, G))
        | [ H : ∀ x1 ... xn, _ ∨ _ ⊢ _ ] ⇒
          orinst H; yelles defs k Empty Empty
      end
  end
with doyelles defs n :=
  yintros; repeat (cbn; try ysplit);
  lazymatch n with
    | 0 ⇒ solve [ isolve ]
    | S ?k ⇒
      first [ yelles defs n Empty Empty |
              match goal with
                | [ x : ?T ⊢ _ ] ⇒
                  notProp T; destruct x; unfolding defs; doyelles defs k
                | [ H : ?T ⊢ _ ] ⇒
                  isPropAtom T;
                  inversion H; try subst;
                  unfolding defs; doyelles defs k
                | [ ⊢ ?A = ?B ] ⇒
                  progress (try destruct A eqn:?; try destruct B eqn:?);
                  unfolding defs;
                  yelles defs k Empty Empty
                | [ H : ∀ x1 ... xn, False ⊢ _ ] ⇒
                  exfalso; yelles defs n Empty Empty
              end]
  end.
Ltac yellesd defs n := cbn; unshelve doyelles defs n; dsolve.
```

**Fig. 3** Tactic `yelles`.

- `htrivial` (2s).
- `hobvious` (2s).
- `heasy` (3s).
- `hsimple` (3s).
- `hyelles` 4 (5s).
- `hyelles` 6 (7s).
- `hyelles` 8 (10s).
- `hyreconstr` (10s).
- `hexhaustive` 0 (3s).
- `hreconstr` 4 (5s).
- `hexhaustive` 2 (7s).
- `hreconstr` 6 (7s).
- `hreconstr` 8 (10s).
- `hexhaustive` 4 (10s).

It is important to note that the time limits are used only when invoking the `hammer` tactic. The specific reconstruction tactics, which the user is supposed to copy into the source from the response window upon success, do not use any time limits and are machine-independent.

The whole source of the tactics `hyreconstr` and `hexhaustive` is not included in Figure 4. The tactic `hyreconstr` essentially tries to invoke `yellesd` with various depth parameters, using some additional heuristics to unfold definitions and destruct some variables. The tactic invocation `hexhaustive` $n$ performs exhaustive proof search up to depth 2 with the tactics `eapply` and `erewrite` (this involves backtracking on existential metavariable instantiations to ensure that no possible instantiations are missed, which is not done with ordinary backtracking), using `yellesd` *defs* $n$ at the leaves.

```
Ltac htrivial hyps lems defs :=
  hinit hyps lems defs;
  simp_hyps;
  intuition (auto with nocore);
  try easy;
  try subst;
  try solve [ hnf in *; congruence 8 ];
  try solve [ constructor ].
Ltac hobvious hyps lems defs :=
  htrivial hyps lems defs;
  simp_hyps;
  try solve [ isolve ];
  try yellesd defs 1.
Ltac heasy hyps lems defs :=
  hobvious hyps lems defs;
  try solve [ unshelve (intuition isolve; eauto 10 with nocore); dsolve ];
  try congruence.
Ltac hsimple hyps lems defs :=
  hobvious hyps lems defs;
  gunfolding defs;
  simp_hyps;
  try yellesd defs 2.
Ltac hyelles n hyps lems defs :=
  hobvious hyps lems defs;
  try yellesd defs n.
Ltac hreconstr n hyps lems defs :=
  hsimple hyps lems defs;
  generalizing;
  repeat (yintros; repeat ysplit);
  try yellesd defs n.
```

**Fig. 4** Reconstruction tactics.