Focused Inductive Theorem Proving

David Baelde, Dale Miller, Zachary Snow

Focused Inductive Theorem Proving

David Baelde[1], Dale Miller[2], and Zachary Snow[1]

[1] Digital Technology Center and Dept of CS, University of Minnesota
[2] INRIA & LIX, École Polytechnique

Abstract. *Focused proof systems* provide means for reducing and structuring the non-determinism involved in searching for sequent calculus proofs. We present a focused proof system for a first-order logic with inductive and co-inductive definitions in which the introduction rules are partitioned into an *asynchronous* phase and a *synchronous* phase. These focused proofs allow us to naturally see proof search as being organized around interleaving intervals of computation and more general deduction. For example, entire Prolog-like computations can be captured using a single synchronous phase and many model-checking queries can be captured using an asynchronous phase followed by a synchronous phase. Leveraging these ideas, we have developed an interactive proof assistant, called Tac, for this logic. We describe its high-level design and illustrate how it is capable of automatically proving many theorems using induction and coinduction. Since the automatic proof procedure is structured using focused proofs, its behavior is often rather easy to anticipate and modify. We illustrate the strength of Tac with several examples of proved theorems, some achieved entirely automatically and others achieved with user guidance.

1 Introduction

The sequent calculus of Gentzen is a well-studied proof framework used to describe provability for a number of logics. This framework also seems to be a natural setting for organizing the search for proofs in a theorem prover. For example, a sequent of the form $\Sigma; \Gamma \vdash B$ denotes the obligation of showing that the formula B follows from the assumptions in the (multi)set Γ for every instantiation of the variables in Σ. An attempt to prove a formula, say B_0, then gives rise to attempts to apply inference rules repeatedly to the root sequent $\cdot; \cdot \vdash B_0$ leaving, at some point, open premises $\Sigma_1; \Gamma_1 \vdash B_1, \ \ldots, \ \Sigma_n; \Gamma_n \vdash B_n$. This set of sequents represents one way to decompose the original proof obligation into $n \geq 0$ subgoals. The frontier of the open proof tree can represent the abstract state of an idealized theorem prover.

The sequent calculus is, unfortunately, far too non-deterministic to directly organize a theorem prover. For example, consider the case when there are even just two hypotheses on which to work. The sequent calculus does not specify on which to work first, and so one might first work on one, then the other, alternating back and forth. This creates an explosive number of alternatives

to explore, many of which are often redundant. Similarly, the structural rules of weakening and contraction can be applied, in principle, to every formula anytime.

Anyone who has attempted to build a theorem prover based on the sequent calculus (or related systems such as tableaux) has undoubtedly observed that there are different ways to give some structure to many of these choices. For example, some inference rules are invertible and, as a result, choices in the order of their application do not affect provability. Additionally, sometimes when a formula is introduced it no longer needs to be maintained: thus the contraction rule need not be considered for that formula. Finally, sometimes selecting one formula for introduction can be seen as causing a cascade of other introduction rules. In recent years, a series of *proof theory* papers have appeared that present various *focused proof systems* for classical and intuitionistic logic. These new proof systems formalize exactly these kinds of observations and turn them into elegant and deep normal form theorems.

Focused proof systems require classifying connectives into two *polarities*, called synchronous and asynchronous. From a proof-search point of view, asynchronous connectives can be introduced early and in any order, since these connectives generally have invertible inference rules. In contrast, once a synchronous formula has been chosen for introduction, then all synchronous subformulas must also be selected immediately for introduction: the synchronous phase ends when the proof is finished or only asynchronous subformulas are reached. This discipline gives rise to the notion of *synthetic* connectives aggregating logical connectives of the same polarity, and focused proof systems can be seen as introducing such synthetic connectives, building their introduction rules from collections of individual, small, introduction rules. The identification of synthetic connectives allows us to view proof search in sequent calculus as revolving around *big step* inference rules and not the usual *small step* introduction rules of Gentzen. Furthermore, since the proof theory behind focused proof systems for classical and intuitionistic logic contains some ambiguity (for example, conjunctions and atoms can be considered as being part of an asynchronous or a synchronous phase), the theorem prover designer has some flexibility in what she wants to have as a synthetic connective. The formal results about focused proof systems provide a solid foundation for these engineered, big-step inference rules: they remain sound and complete with respect to the original small-step proof system.

From the perspective of designing a theorem prover, the above concepts are invaluable. As we shall see, focusing allows mixing computation and deduction in natural and transparent ways. For example, it is entirely possible to describe, say, the concatenation of two lists (in the relational style of Prolog) and then embed the entire computation of such a relationship within one synthetic connective. This is in striking contrast with the treatment in most resolution-style theorem provers where such a computation is emulated by a possibly large number of small-step resolution rules. Going one step further, a model-checking problem (*e.g.*, all members of one finite set are members of another set) can be naturally modeled as just two synthetic connectives: the first asynchronous (enumerating all members by case analysis) and the second synchronous (showing that they

belong to the other set). Thus, the high-level viewpoint of proof brought by the notion of *synthetic connective* can make for more effective proof-search.

In this paper, we present the design and applications of an automatic theorem prover, called Tac. Section 2 describes μLJ, the logic underlying Tac: it is an intuitionistic logic containing least and greatest fixed points. In Section 3 we introduce a focused proof system for μLJ. In Section 4, we describe the high-level design of Tac, in particular its automatic proof-search strategy involving the use of (co)induction. This design has been governed by the following three principles. First, while Tac is an interactive prover based on tacticals, it is hoped that a single, automatic tactic can be used to fill in the gaps between a theorem and a list of (human supplied) lemmas. Second, the automatic tactic is organized around the search for focused proofs via the use of synthetic connectives. Third, the only influence we allow on the automatic tactic's behavior involves those aspects of focused proof systems that proof theory has not fixed. Section 5 summarizes the behavior of Tac and compares it to some other theorem proving systems.

2 The logic μLJ

The logic μLJ [2] is the extension of first-order intuitionistic logic[3] with inductive and coinductive definitions given using least and greatest fixed points. The proof system for μLJ contains familiar rules for inductive and coinductive inference based on the selection of invariants, which notably provides the intuitionistic version of Peano's arithmetic. Its study is inspired by that of μMALL [4].

We consider the following simply typed language of formulas:

$$\begin{aligned} P ::= {} & P \wedge P \mid P \vee P \mid P \supset P \mid \bot \mid \top \\ & \mid \exists_\gamma x.\ P \mid \forall_\gamma x.\ P \mid s \stackrel{\gamma}{=} t \mid \mu_{\gamma_1 \ldots \gamma_n}(\lambda p \lambda \boldsymbol{x}.\ P)\boldsymbol{t} \mid \nu_{\gamma_1 \ldots \gamma_n}(\lambda p \lambda \boldsymbol{x}.\ P)\boldsymbol{t}. \end{aligned}$$

The syntactic variable γ represents a term type, *e.g.,* natural numbers or lists. The quantifiers have type $(\gamma \to o) \to o$ and the equality has type $\gamma \to \gamma \to o$. The least fixed point connective μ and the greatest fixed point connective ν have type $(\tau \to \tau) \to \tau$ where τ is $\gamma_1 \to \cdots \to \gamma_n \to o$ for some arity $n \geq 0$. We shall almost always elide the references to γ, assuming that they can be determined from the context when it is important to know their value. Note that we do not consider atoms, *i.e.,* predicate constants: although μLJ accomodates them without any problem, atoms are often unnecessary since fixed points play their role in practice, and thus we leave them out for simplicity.

Formulas with top-level connective μ or ν are called fixed point expressions. Fixed points can be arbitrarily nested and interleaved — that is, we can have mutually recursive definitions. The first argument of a fixed point connective is a predicate operator expression, called its *body*, and shall be denoted by B. In order for the logic to enjoy consistency and other useful properties, all fixed

[3] While intuitionistic logic is the natural choice for the specifications we consider, note that our proof-theoretical approach also applies well to linear or classical settings.

point bodies are required to be *monotonic*, *i.e.,* there should be no negative occurrence of the bound predicate variable p in $\lambda p \lambda \boldsymbol{x}.\ B p \boldsymbol{x}$.

Example 1. Assuming a term type n and two constants $0 : n$ and $s : n \to n$, the natural number predicate *nat* of type $n \to o$ can be defined as the inductive expression μB_{nat}, where B_{nat} is defined as $\lambda N \lambda x.\ x = 0 \vee \exists y.\ x = s\ y \wedge N\ y$.

The inference rules of μLJ deal with usual first-order intuitionistic sequents, of the form $\Sigma; \Gamma \vdash P$ where Σ is a set of universal (eigen)variables $x_1, \ldots, x_n$ and Γ is a set of formulas $P_1, \ldots, P_m$. The logical reading of such a sequent is $\forall x_1 \ldots \forall x_n.\ (P_1 \wedge \ldots \wedge P_m \supset P)$.

The inference rules of μLJ are the usual ones for the propositional connectives and first-order quantifiers. The left and right-introduction rules for equality date back to [6, 12]:

$$\frac{\{(\Sigma; \Gamma \vdash Q)\theta : \theta \in csu(t \doteq t')\}}{\Sigma; \Gamma, t = t' \vdash Q}\ {=}L \qquad \frac{}{\Sigma; \Gamma \vdash t = t}\ {=}R$$

In the left equality rule (=L), *csu* stands for *complete set of unifiers* [7]. This set can be restricted to have at most one element when terms are first-order but might be infinite if terms are interpreted modulo some algebraic theory or if they are simply typed λ-terms. The application of a substitution to the signature of a sequent consists in removing instantiated variables and adding newly introduced ones; the application to the rest of the sequent simply propagates it to the terms of every formula. Note that this treatment of equality is stronger than Leibniz equality, as it notably expresses the injectivity of term constructors. More generally, it provides our proof system with an approach to *negation-as-failure*: if the equality $t = t'$ is a *failure* (that is, $csu(t, t')$ is empty) then the equality left rule yields a *successful* proof (that is, the rule has no premises).

The least fixed point μB is characterized as the least of the prefixed points.

$$\frac{\Sigma; \Gamma, S\boldsymbol{t} \vdash P \qquad \boldsymbol{x}; BS\boldsymbol{x} \vdash S\boldsymbol{x}}{\Sigma; \Gamma, \mu B\boldsymbol{t} \vdash P}\ \textit{induction} \qquad \frac{\Sigma; \Gamma \vdash B(\mu B)\boldsymbol{t}}{\Sigma; \Gamma \vdash \mu B\boldsymbol{t}}\ \mu\textit{-unfolding}$$

The right *unfolding* rule expresses $B(\mu B)\boldsymbol{t} \supset \mu B\boldsymbol{t}$, and the left *induction* rule expresses that μB entails any prefixed point S, also called an *invariant*. Notice that the universal variables $\boldsymbol{x}$ in the induction rule are new.

From the induction rule one can always derive a left unfolding rule for μ, using the invariant $B(\mu B)$:

$$\frac{\Sigma; \Gamma, B(\mu B)\boldsymbol{t} \vdash P}{\Sigma; \Gamma, \mu B\boldsymbol{t} \vdash P}$$

Thus, the least prefixed point is a fixed point, *i.e.,* $\mu B\boldsymbol{x}$ and $B(\mu B)\boldsymbol{x}$ are provably equivalent. The introduction rules for greatest fixed points are the dual rules:

$$\frac{\Sigma; \Gamma, B(\nu B)\boldsymbol{t} \vdash P}{\Sigma; \Gamma, \nu B\boldsymbol{t} \vdash P}\ \nu\textit{-unfolding} \qquad \frac{\Sigma; \Gamma \vdash S\boldsymbol{t} \qquad \boldsymbol{x}; S\boldsymbol{x} \vdash BS\boldsymbol{x}}{\Sigma; \Gamma \vdash \nu B\boldsymbol{t}}\ \textit{coinduction}$$

Finally, the initial rule can be restricted to fixed point expressions.

$$
\begin{aligned}
\mathit{eq} &\stackrel{def}{=} \mu(\lambda E \lambda x \lambda y.\ (x = 0 \wedge y = 0) \vee (\exists x' \exists y'.\ x = s\ x' \wedge y = s\ y' \wedge E\ x'\ y')) \\
\mathit{leq} &\stackrel{def}{=} \mu(\lambda L \lambda x \lambda y.\ x = y \vee (\exists y'.\ y = s\ y' \wedge L\ x\ y')) \\
\mathit{half} &\stackrel{def}{=} \mu(\lambda H \lambda x \lambda h.\ ((x = 0 \vee x = s\ 0) \wedge h = 0) \\
&\qquad \vee (\exists x' \exists h'.\ x = s^2\ x' \wedge h = s\ h' \wedge H\ x'\ h')) \\
\mathit{append} &\stackrel{def}{=} \mu(\lambda A \lambda x \lambda y \lambda z.\ (x = \mathit{nil} \wedge y = z) \\
&\qquad \vee (\exists e \exists x' \exists z'.\ x = e :: x' \wedge z = e :: z' \wedge A\ x'\ y\ z')) \\
\mathit{reverse} &\stackrel{def}{=} \mu(\lambda R \lambda l \lambda r.\ (l = \mathit{nil} \wedge r = \mathit{nil}) \\
&\qquad \vee (\exists h \exists l' \exists r'.\ l = h :: l' \wedge R\ l'\ r' \wedge \mathit{append}\ r'\ (h :: \mathit{nil})\ r)) \\
\mathit{sim} &\stackrel{def}{=} \nu(\lambda S \lambda p \lambda q.\ \forall l \forall p'.\ \mathit{step}\ p\ a\ p' \supset \exists q'.\ \mathit{step}\ q\ a\ q' \wedge S\ p'\ q')
\end{aligned}
$$

Fig. 1. Examples of fixed point expressions

Example 2. In the particular case of *nat*, the induction rule with invariant S yields the usual induction principle:

$$
\frac{\Sigma; \Gamma, S\ t \vdash P \qquad \dfrac{\vdash S(0) \quad y; S(y) \vdash S(s\ y)}{x; (B_{nat} S)x \vdash Sx}\ \vee L, \exists L, \wedge L, {=}L}{\Sigma; \Gamma, \mathit{nat}\ t \vdash P}
$$

The logic μLJ results from a line of work on definitions [6, 12] and induction and coinduction [9, 11]. The presentation using μ and ν makes for a more direct proof theoretical study, and notably naturally brings the possibility to treat mutual (co)inductive definitions in an expressive way. Figure 1 contains several example fixed point definitions. These examples use *nil* as the empty list constructor and :: as the non-empty list constructor.

For brevity, we shall omit the signature Σ from the sequents in the next sections; its treatment should be clear from the above presentation.

3 Focused proofs for μLJ

The proof system for μLJ described in the previous section is *unfocused* since there is no particular structure imposed on how one occurrence of an inference rule relates to another. In contrast, a focused proof system classifies inference rules into synchronous and asynchronous ones and then groups those of similar classification into one "synthetic introduction rule". The first focused proof system for a full logic was given by Andreoli for linear logic [1]. Eventually, focused proof systems have been developed for other logics where it was revealed that, unlike in linear logic, proof theory concerns do not fix all polarization choices (in particular, for atoms [1], conjunctions [8], and fixed points [2]). As a result, these non-fixed items could be placed into either the asynchronous or the synchronous phases: such choices do not affect provability but can have a striking

effect on the size and shape of proofs. In linear logic, asynchronous connectives are exactly those with invertible right-introduction rules; this does not hold for richer logics, such as μLJ. Indeed, fixed points can always be treated in an invertible way (provability is never lost by unfolding) but completeness cannot be obtained with a strategy that eagerly unfolds all fixed points. For example, proving $nat\ x \vdash nat\ x$ cannot succeed by repeatedly unfolding the hypothesis; at some point, one has to stop unfolding and, instead, use the initial rule.

Definition 1 (Polarities for μLJ). *The connectives $\wedge$, $\vee$, $\exists$, $=$, and μ are* synchronous *while the connectives $\forall$, $\supset$ and ν are* asynchronous*. A synchronous (resp. asynchronous) formula is one whose top-level connective is synchronous (resp. asynchronous). If* every *connective of a formula is synchronous (resp. asynchronous), it is called* fully *synchronous (resp. asynchronous). Finally, a fixed point formula can be annotated as* frozen*, which is denoted by $(\mu B\boldsymbol{t})^*$ and $(\nu B\boldsymbol{t})^*$, in which case it is neither synchronous nor asynchronous.*

Figures 2, 3 and 4 present μLJF, a focused proof system for μLJ. There are two kinds of sequents: the *unfocused sequent* is written $\Gamma \vdash P$ (as before) and the *focused sequent* is written with a bracketed formula (the focus) as either $\Gamma \vdash [P]$ or $\Gamma, [P] \vdash Q$. In each of these sequents, Γ is a multiset of formulas. There is an unsurprising symmetry between left and right hand-sides of sequents: a synchronous connective is treated as asynchronous on the left and *vice-versa*. The *asynchronous phase* contains sequents of the form $\Gamma \vdash P$ and introduces asynchronous connectives on the right and synchronous ones on the left. The *synchronous phase* contains sequents containing one distinguished (bracketed) formula that is *under focus*. When the focus is on the right ($\Gamma \vdash [P]$) only the toplevel synchronous connectives of P can be introduced. When the focus is on the left ($\Gamma, [P] \vdash Q$) only toplevel asynchronous connectives of P can be introduced. The alternation between the two phases is allowed only when no other rule applies: the asynchronous phase ends when no synchronous formula remains on the left, and the conclusion is synchronous; the synchronous phase ends when the focus is on the left on a synchronous formula, or on the right on an asynchronous one. Finally, the structural rule of contraction is used (implicitly) only in the rule establishing a left focus formula — and thus, only for asynchronous formulas.

Each fixed point has two rules per phase: one of these rules treats the fixed point as a structured formula; the other treats it as an atom. The synchronous rules are unfolding and the initial rule and the asynchronous rules are (co)induction and *freezing*. A strong constraint of the asynchronous phase is that it requires that any least fixed point hypothesis (and greatest fixed point conclusion) is either immediately used for (co)induction (which includes unfolding) or frozen, in which case it can never again be unfolded or used for induction: it can only be used in an initial rule later in the proof. Also note that when one focuses on a fully synchronous least fixed point, such as *nat* and all predicates of Figure 1 except *sim*, focus can never be released. Hence, the proof has to be completed in that phase, eventually reaching units, equality, or the initial rule if an appropriate frozen side-formula is available.

$$\frac{\Gamma, P, P' \vdash Q}{\Gamma, P \wedge P' \vdash Q} \quad \frac{\Gamma, P \vdash Q \quad \Gamma, P' \vdash Q}{\Gamma, P \vee P' \vdash Q} \quad \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \supset Q} \quad \frac{}{\Gamma, \bot \vdash P} \quad \frac{\Gamma \vdash P}{\Gamma, \top \vdash P}$$

$$\frac{\Gamma \vdash Px}{\Gamma \vdash \forall x.\ Px} \quad \frac{\Gamma, Px \vdash Q}{\Gamma, \exists x.\ Px \vdash Q} \quad \frac{\{(\Gamma \vdash P)\theta : \theta \in csu(t \doteq t')\}}{\Gamma, t = t' \vdash P}$$

$$\frac{\Gamma, S\boldsymbol{t} \vdash P \quad BS\boldsymbol{x} \vdash S\boldsymbol{x}}{\Gamma, \mu B\boldsymbol{t} \vdash P} \quad \frac{\Gamma, (\mu B\boldsymbol{t})^* \vdash P}{\Gamma, \mu B\boldsymbol{t} \vdash P} \quad \frac{\Gamma \vdash S\boldsymbol{t} \quad S\boldsymbol{x} \vdash BS\boldsymbol{x}}{\Gamma \vdash \nu B\boldsymbol{t}} \quad \frac{\Gamma \vdash (\nu B\boldsymbol{t})^*}{\Gamma \vdash \nu B\boldsymbol{t}}$$

Fig. 2. μLJF: asynchronous rules

$$\frac{\Gamma \vdash [P] \quad \Gamma \vdash [P']}{\Gamma \vdash [P \wedge P']} \quad \frac{\Gamma \vdash [P_i]}{\Gamma \vdash [P_0 \vee P_1]} \quad \frac{\Gamma, [P'] \vdash Q \quad \Gamma \vdash [P]}{\Gamma, [P \supset P'] \vdash Q} \quad \frac{}{\Gamma \vdash [\top]}$$

$$\frac{\Gamma, [Pt] \vdash Q}{\Gamma, [\forall x.\ Px] \vdash Q} \quad \frac{\Gamma \vdash [Pt]}{\Gamma \vdash [\exists x.\ Px]} \quad \frac{}{\Gamma \vdash [t = t]}$$

$$\frac{\Gamma, [B(\nu B)\boldsymbol{t}] \vdash P}{\Gamma, [\nu B\boldsymbol{t}] \vdash P} \quad \frac{}{\Gamma, [\nu B\boldsymbol{t}] \vdash (\nu B\boldsymbol{t})^*} \quad \frac{\Gamma \vdash [B(\mu B)\boldsymbol{t}]}{\Gamma \vdash [\mu B\boldsymbol{t}]} \quad \frac{}{\Gamma, (\mu B\boldsymbol{t})^* \vdash [\mu B\boldsymbol{t}]}$$

Fig. 3. μLJF: synchronous rules

$$\frac{\Gamma, Q, [Q] \vdash P}{\Gamma, Q \vdash P} \quad \frac{\Gamma \vdash [P]}{\Gamma \vdash P} \quad \frac{\Gamma, P \vdash Q}{\Gamma, [P] \vdash Q} \quad \frac{\Gamma \vdash Q}{\Gamma \vdash [Q]}$$

Fig. 4. μLJF: structural rules (P synchronous, Q asynchronous)

The standard, unfocused proof system for μLJ can be recovered from μLJF by removing all focusing annotations.

Theorem 1 (Completeness [2]). *The sequent $\Gamma \vdash P$ is provable in μLJ if and only if it is provable in μLJF.*

Although not visible in the statement of completeness, the asynchronous rules can be applied in any order — permuting them actually leaves the proof essentially unchanged.

As usual, the completeness of the focused proof system justifies a reading of logic based on synthetic connectives and synthetic introduction rules. A synthetic introduction rule for a synthetic synchronous connective is a big-step rule that has a focused sequent as its conclusion and which extends upwards until there are only unfocused sequents present. Dually, a synthetic introduction rule for a synthetic asynchronous connective is a big-step rule that has an unfocused sequent as its conclusion and which extends upwards until no asynchronous rule can be applied. Note that this is especially powerful with fixed points, since we can now have synthetic introduction rules built from unbounded numbers of micro-rules. An interesting particular case of this is that of fully synchronous formulas, such as *nat* and more generally any Prolog-style computation, which constitute a synthetic unit, with an infinity of synthetic introduction rules.

Example 3. Consider the synthetic introduction rule that ends with the right-focused sequent $\Gamma \vdash [(leq\ m\ n \wedge B_1) \vee (leq\ n\ m \wedge B_2)]$, where m and n are natural numbers (terms over s and 0) and leq is the purely synchronous fixed point in Figure 1 denoting the less-than-or-equal-to relation. If both B_1 and B_2 are asynchronous formulas, then there are exactly two possible synthetic rules: one with premise $\Gamma \vdash B_1$ when $m \leq n$ and one with premise $\Gamma \vdash B_2$ when $n \leq m$ (thus, if $m = n$, both premises are possible). In this sense, a synthetic synchronous connective can contain an entire Prolog-style computation.

Being complete, the focused proof system for μLJ obviously does not render theorem proving decidable. But the focused structure of proofs can be very useful when building a theorem prover, as we shall see in the main contribution of this paper: the design of an automatic tactic that is organized around synthetic connectives.

4 Tac

There are several ways to exploit focusing for proof-search. For instance, the inverse method — performing top-down proof-search — yields impressive results when combined with focusing [5, 10]. Proof search that must generate (co)invariants is hard to do in such a top-down style, particularly when contexts are used to generate the (co)invariants. Thus, we use bottom-up proof search. As we outline next, that choice is also compatible with the use of tactics and tacticals in a proof assistant.

Tac is built around a small kernel implementing elementary operations and (small-step) inferences rules. Each inference rule gives rise to a primitive tactic. Complex tactics can then be formed using tacticals such as `then`, `repeat`, etc. The successful application of a series of tactics to prove a theorem triggers the production of a proof, which can be inspected. There is no specific support for any particular datatype.

Focusing is built into the foundations of Tac: formulas are annotated with polarity information and that information is used to guide the application of logical rules. Such annotations are useful not only for the automation of inductive proof search but also for human interaction. For example, the tactic `async` simplifies a goal by repeatedly applying asynchronous rules, the common tactic `apply` actually consists of focusing on a lemma and performing a synchronous phase, and the tactic `freeze` is used to prevent induction and thereby guide automated theorem proving. Finally, we also use focusing to display proofs more concisely by showing synthetic inference rules instead of the "micro" rules.

In the following we describe our automated tactic, called `prove`. This tactic performs focused proof-search as described in Section 3, with a special treatment of computation and of the crucial deduction step of (co)induction.

4.1 Progress

We generally think of proof search as being a process composed primarily of deduction, but of course large portions of a particular proof may be given over

to computation. Any implementation of proof search should recognize and exploit this distinction, not only for efficiency (computations should not involve (co)inductions or certain other deductive techniques) but also for robustness and predictability. Focusing allows us to circumscribe computations to single phases, even if that computation is non-deterministic. A useful fragment to identify is that of deterministic computation. For example, given the goal $\forall x.\ \mathit{mult}\ 10\ 10\ x \supset P\ x$ or the goal $\exists x.\ \mathit{mult}\ 10\ 10\ x \wedge P\ x$, the prover should compute the value of x immediately in a single step. The notion of *progressing unfolding* presented below allows us to do so and, in fact, to treat these examples in exactly the same way.

Definition 2 (Patterns). *A* pattern $\mathcal{C}$ *of type* $\gamma_1, \ldots, \gamma_n \to \gamma'_1, \ldots, \gamma'_m$ *is a vector of* m elementary patterns p_i*, which are themselves closed terms of type* $\gamma_1, \ldots, \gamma_n \to \gamma'_i$*. The* input arity *of the pattern is* n*, and* m *is its* output arity*. Both can be zero. When* $\boldsymbol{t}$ *is a vector of terms* $\langle t_1 : \gamma_1, \ldots, t_n : \gamma_n \rangle$*, the expression* $\mathcal{C}\boldsymbol{t}$ *denotes the vector* $\langle p_1\boldsymbol{t}, \ldots, p_n\boldsymbol{t} \rangle$*. For two vectors of terms of equal length* n*, the expression* $\boldsymbol{t} = \boldsymbol{t'}$ *denotes the formula* $t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n$*.*

Definition 3 (Matcher). *Let* $\boldsymbol{\mathcal{C}}$ *be a vector* $\langle \mathcal{C}_1, \ldots, \mathcal{C}_n \rangle$ *of patterns, all of the same output arity* m*. The matcher* $M_{\boldsymbol{\mathcal{C}}}$ *is defined as the term:*

$$M_{\boldsymbol{\mathcal{C}}} \stackrel{def}{=} \lambda\phi_1 \ldots \lambda\phi_n \lambda x_1 \ldots \lambda x_m . (\exists \boldsymbol{y}_1.\ \boldsymbol{x} = \mathcal{C}_1\boldsymbol{y} \wedge \phi_1\boldsymbol{y}) \vee \ldots \vee (\exists \boldsymbol{y}_n.\ \boldsymbol{x} = \mathcal{C}_n\boldsymbol{y} \wedge \phi_n\boldsymbol{y})$$

Example 4. We can define *nat* from the matcher on the patterns $\mathcal{C}_1 := \langle 0 \rangle$ and $\mathcal{C}_2 := \langle \lambda p.\ s\ p \rangle$ by $\mathit{nat} := \mu(\lambda N \lambda x.\ M_{\boldsymbol{\mathcal{C}}} \top N x)$. The fixed point *half* is built on the patterns $\langle 0, 0 \rangle$, $\langle s\ 0, 0 \rangle$, and $\langle \lambda p.\ s^2 p, \lambda p.\ s\ p \rangle$. Finally, the binary fixed point *eq* is built on the patterns $\langle 0, 0 \rangle$ and $\langle \lambda p.\ s\ p, \lambda p.\ s\ p \rangle$.

In fact, matchers correspond to synchronous synthetic connectives, leaving out the least fixed points. In most fixed point definitions, the structure of matchers is used literally, but even when it is not strictly followed, it can be recovered by re-arranging the outermost layer of synchronous connectives, namely $\wedge$, $\vee$ and $\exists$. Hence, any fixed point body can be assumed of the form $(\lambda p \lambda \boldsymbol{x}.\ M_{\boldsymbol{\mathcal{C}}}(\boldsymbol{\phi} p)\boldsymbol{x})$ for some patterns $\boldsymbol{\mathcal{C}}$ and predicate operator expressions $\boldsymbol{\phi}$.

Definition 4 (Progressing unfolding). *Let* $\boldsymbol{\mathcal{C}}$ *be a vector* $\langle \mathcal{C}_1, \ldots, \mathcal{C}_n \rangle$ *of patterns of the same output arity. A least fixed point instance* $\mu(\lambda p \lambda \boldsymbol{x}.\ M_{\boldsymbol{\mathcal{C}}}(\boldsymbol{\phi} p)\boldsymbol{x})\boldsymbol{t}$ *has a progressing unfolding if* $\exists \boldsymbol{x}.\ \boldsymbol{t} = \mathcal{C}_j \boldsymbol{x}$ *holds for at most one* $j \in \{0, \ldots, n\}$*.*

Example 5. The formulas *nat* 0 and *nat* $(s\ t)$ have progressing unfoldings, for any term t, *e.g.,* $s^n 0$, x, *nil*. For any h, the formulas *half* 0 h, *half* $(s\ 0)\ h$ and *half* $(s^2 t)\ h$ for any t have progressing unfoldings. This is not true of *half* $(s\ x)\ h$, because the term $s\ x$ satisfies two patterns in the definition of *half*. For *eq*, the progressing unfoldings are on instances of *eq* 0 0, *eq* $(s\ x)\ y$ and *eq* $x\ (s\ y)$.

This definition is critically tied to focusing: we only inspect one synchronous synthetic connective, namely a synchronous fixed point and the outermost synchronous layer of its body. After an unfolding on the left hand-side, all absurd

branches of that structure are discarded during the current asynchronous phase, and at most one remains. Symmetrically, after an unfolding on the right hand-side, at most one branch remains by the end of the synchronous phase.

Note that the definition of progressing unfolding described above does not attempt to embody a notion of termination. There is no restriction placed on the structure under the pattern, hence no guarantee that the result of the unfolding is simpler than the initial fixed point. It is in fact undecidable to identify non-termination, and not even desired, since we also want to partially explore infinite computations, as we shall see in Example 6.

4.2 Discovering (co)invariants

The construction of (co)invariants in theorem proving is an extremely difficult problem, one that has been addressed by a large number of researchers. We take a simple approach with the `prove` tactic, trying only one invariant per possible induction site. That invariant is obtained directly from the context:

$$\frac{\Sigma;\ \Gamma, S\boldsymbol{t} \vdash G \quad \boldsymbol{x}\ ;\ BS\boldsymbol{x} \vdash S\boldsymbol{x}}{\Sigma;\ \Gamma, \mu B\boldsymbol{t} \vdash G} \quad \text{with } S := \lambda\boldsymbol{x}.\ \forall\Sigma.\ \boldsymbol{x} = \boldsymbol{t} \supset (\bigwedge \Gamma) \supset G.$$

With this invariant, the first premise is trivially provable, as it is essentially an instance of the identity. The second premise is where proof-search continues. This approach to induction is similar to that used in Coq, where the induction tactic always uses the current goal as the invariant. When that is not sufficient, one has to generalize the goal manually — for example by introducing a lemma.

We proceed dually for coinduction:

$$\frac{\Sigma;\ \Gamma \vdash S\boldsymbol{t} \quad \boldsymbol{x}\ ;\ S\boldsymbol{x} \vdash BS\boldsymbol{x}}{\Sigma;\ \Gamma \vdash \nu B\boldsymbol{t}} \quad \text{with } S := \lambda\boldsymbol{x}.\ \exists\Sigma.\ \boldsymbol{x} = \boldsymbol{t} \wedge (\bigwedge \Gamma).$$

Such trivial (co)inductions suffice for many examples. For instance, when proving $\forall n.\ \mathit{even}\ n \supset \mathit{nat}\ n$, the generated formula $\lambda x.\ \mathit{nat}\ x$ is actually an invariant of *even*. Similarly, when proving $\forall p.\ \mathit{sim}\ p\ p$, the context does provide a coinvariant: $\lambda p_1 \lambda p_2.\ \exists p.\ p_1 = p \wedge p = p_2$, that is, $\lambda p_1 \lambda p_2.\ p_1 = p_2$.

Example 6. Consider the following theorem of μLJ: $\forall x.\ \mathit{eq}\ (s\ x)\ (s\ (s\ x)) \supset \bot$. In the asynchronous phase, $\forall$ and $\supset$ are introduced, after which the fixed point has to be treated. It can be either frozen, inducted on, or unfolded. Obviously, freezing cannot lead to a proof. Induction fails as the context does not yield a valid invariant ($\lambda m \lambda n.\ \forall x.\ m = s\ x \supset n = s\ (s\ x) \supset \bot$). The last possibility is to perform the (progressing) unfolding of the fixed point, in which case we obtain the subgoal $\mathit{eq}\ x\ (s\ x) \vdash \bot$. At this point one can quickly obtain a proof, as the context does yield an invariant: $\lambda m \lambda n.\ n = s\ m \supset \bot$. Notice that performing another progressing unfolding of *eq* would loop, producing the same subgoal.

4.3 Organization of the `prove` tactic

In order to turn focused proof search into a practical automated tactic, several compromises must be made and these compromises will result, ultimately, in the loss of completeness. We detail below the main compromises for `prove`, namely the design of search in the asynchronous phase and how termination is ensured.

Except for freezing, all asynchronous rules are invertible – in the case of induction, this is obtained by instantiating it into a left unfolding. However, from a bottom-up proof-search point of view, this does not mean that these rules can be applied eagerly without backtracking. Namely, an asynchronous fixed point instance can be treated in a number of different ways: it may be frozen, (co)inducted on with an arbitrary (co)invariant (although in automated proof search we only use the (co)invariant generated from the context) or unfolded, perhaps in a progressing way. Our approach is to postpone these choices as much as possible, first applying all non-backtracking asynchronous rules. Moreover, we treat progressing unfoldings as non-backtracking rules, reflecting their deterministic nature. Therefore the asynchronous phase proceeds as follows: (1) apply the non-backtracking asynchronous rules, (2) try for each remaining asynchronous fixed point to either freeze, (co)induct or unfold[4], backtracking on those possibilities and coming back to Step (1) after each attempt.

The usual problem of top-down proof-search is that it may encounter infinite branches during search — in μLJ, such branches are caused by contractions, fixed point unfoldings, and (co)induction. A common way to address this issue is to bound the depth of search. This technique can be too rough, but we tame its downsides by working at the level of synthetic connectives. Moreover, we exploit the distinction between computation and deduction. First, we make use of a *deductive* bound, attached to individual sequents, which limits the number of non-progressing fixed point unfoldings, (co)inductions, and contractions in one branch. Second, we introduce a *computational* bound to limit the number of progressing unfoldings performed on a given fixed point. Computational bounds must be attached to individual formulas: since progressing unfoldings are performed eagerly, we must prevent one chain of such unfoldings from starving another. The deductive bound controls the number of critical choices made in a proof, and therefore the complexity of proof-search; increasing the bound even slightly can lead to significantly longer attempts. However, since computations are factored out of the deductive bound, a low bound typically suffices; as a default we perform iterative deepening up to a depth of 3. In contrast, the computational bound can be set much higher without affecting the cost of proof-search. This design has proved critical to the success of our tactic.

5 Comparison and experimental results

In order to show the strengths of our approach, we provide several examples of its successes in Figure 5 and use them to compare Tac with two es-

[4] It might be surprising that we attain the best results by attempting (co)induction before unfolding; this is because these non-progressing unfoldings are rarely useful.

tablished inductive theorem provers. The first group of examples come from the IWC suite (`http://www.cs.nott.ac.uk/~lad/research/challenges`) and the second one from Twelf's examples. The last group consists of interesting original examples leveraging Tac's specificities: *sim* illustrates that Tac deals with coinductive definitions and coinduction just as naturally as with inductive definitions; the other examples show interesting developments[5] that exploit Tac's support for generic quantification [3] and the role that the automatic tactic `prove` can play within interactive proof development. We do not provide timings since our implementation was not optimized for speed, but stress that all automated examples pass in less than a few seconds. The implementation, including all of the examples mentioned in this paper, is available at `http://slimmer.gforge.inria.fr/tac/`.

Name	Description	Success
IWC 02	$append(l, l')$ of even length iff $append(l', l)$ is too	Automatic
IWC 03	$x \in l$ implies $x \in append(l, l')$	Automatic
IWC 04	$l = rotate(length(l), l)$	Guided, Lemma
IWC 06	equiv. of mutual and straight definitions of even	Automatic
IWC 07	natural numbers are even or odd	Automatic
IWC 12	verifying abstractions in model checking	Guided, Lemmas
IWC 16	whisky problem	Automatic, Lemma
plus	commutativity and associativity of *plus*	Automatic
arith	totality of many Horn programs, *e.g., half*, *ack*	Automatic
prop-calc	Hilbert's abstraction theorem	Automatic
reverse	involutivity of list reversal	Automatic
binarytree	antisymmetry of the subtree ordering on binary trees	Automatic
sim	reflexivity and transitivity of *sim*	Automatic
PCF	subject reduction and determinacy of typing for PCF	Manual, Lemmas
POPL-1A	transitivity of subtyping for $F^{\leq}$	Manual, Lemmas

Fig. 5. Examples of Tac proofs. "Lemma" indicates that a lemma had to be manually specified. "Automatic" indicates that `prove` derived the theorem and all lemmas. "Guided" denotes a small amount of user guidance while "Manual" denotes a mostly interactive development.

5.1 Comparison with rewriting based approaches

Rewriting based approaches to inductive theorem proving are common: for example, ACL2 and the many provers that make use of rippling. These specialized foundations, together with refined heuristics, make for powerful tools, but also have some drawbacks. Notably, they do not provide as solid proof witnesses as

[5] We also have a partial solution of POPLMark problem 2A, and we do not see any obstacle to its short-term completion.

sequent calculus, and in general cannot be related to general-purpose proof assistants such as Coq or Isabelle. Leaving aside problems that involve higher-order functions, and thus could not even be stated in our framework, the IWC challenges highlight the main differences between our approach and rewriting-based techniques.

Several challenges were infeasible in Tac because they relied too heavily on equational reasoning. When stated in a relational way in our tool, the complexity of such statements increases a lot: for example, $(x+y)+z = t$ becomes $\exists i.\ x+y = i \wedge i + z = t$. This weakness is not so important in application areas such as operational semantics, where commutativity and associativity are rarely relevant and equational reasoning is little used.

On the other hand, the challenges that Tac passed show that our relatively straightforward but general approach replicates reasoning schemes that have to be implemented as special techniques in rewriting and termination-based approaches, such as case analysis, generalization, simultaneous and mutual induction schemes. Our single induction principle and limited invariant generation already embeds an interesting amount of generalization, and the nesting and interleaving of inductions gives rise to complex induction schemes (corresponding, for example, to multiset and lexicographic orderings). This is visible in some of our custom examples such as the totality of *ack* or the involutivity of list reversal.

In some cases, it is unreasonable to expect an automatic proof: the user will need to explicitly introduce generalizations or lemmas. The same goes for all provers and Tac is no exception. In our test suite, once lemmas were stated, Tac was able to prove them automatically and deduce the final theorem. In IWC challenges 4 and 12, however, some non-trivial human guidance was required in this process.

5.2 Comparison with Twelf

Twelf [13] allows writing specifications in LF, a dependently typed intuitionistic framework, and can perform proof-search in LF as well as meta-reasoning about LF specifications. Meta-reasoning is done in two ways.

First, Twelf is equipped with a number of specialized procedures for establishing whether some LF relation is total, functional, etc. That feature is well developed and widely used. We can replicate many of the established meta-theorems, but those dedicated procedures outperform our generic tactic, which is not able to re-use properties of nested fixed points. It should be noted, however, that in order to check a totality assertion, a termination order should be explicitly given, while our system infers it.

Secondly, Twelf has a more generic meta-theorem prover [13] which searches for proofs in the $\mathcal{M}_2$ meta-logic, whose objects are LF terms. This logic only deals with Π_2 statements and Twelf implements a simply structured proof-search strategy for them. The main phases of this strategy can be formulated in terms of focusing, but their general organization differs. Notably, this strategy only

attempts an outermost induction, whose validity is based on a termination ordering provided by the user. In contrast, the (co)invariant generation of Tac can discover induction schemes without assistance from the user and can notably use nested inductions to discover proofs that are inaccessible to Twelf. A striking example is the involutivity of list reversal, which Tac proves without any additional lemmas thanks to nested inductions. On the other hand, Twelf can sometimes prove a theorem, *e.g.,* the totality of *half*, using a single induction when Tac would require nested inductions, since Twelf's meta-logic allows the use of an induction hypothesis on an arbitrary predecessor, while our scheme corresponds to restricting to the immediate predecessor. Finally, there is no notion of progress in Twelf's meta-theorem prover, which results in a critical need to tweak bounds.

The major strength of Twelf is not so much the architecture of its proof-search strategy, but the expressivity of the LF objects manipulated in the $\mathcal{M}_2$ logic. Twelf handles higher-order abstract syntax (HOAS) specifications easily, enabling its simple meta-theorem prover to prove important results such as type preservation and progress for programming languages. Tac supports higher-order terms and features minimal generic quantification [3] to expressively reason about such objects. Focused proof search is not affected by the introduction of these rich notions, and `prove` indeed handles them without any modification. But HOAS notably involves dealing with hypothetical contexts, for which there is no built-in support in Tac. When working with such contexts in Tac one must therefore implement them by hand, generally using lists, which tends to create artificial details. As a result, while Tac seems to be in general more powerful than Twelf's meta-theorem prover on examples not involving HOAS, it is unable to carry out automatically developments significantly dealing with hypothetical contexts. However, since Tac is an interactive proof assistant, the user can guide the proving of such theorems, still benefiting from `prove` to fill in many simple proof obligations.

6 Conclusion

We have developed a strong focusing proof system for the expressive logic μLJ, and we have shown how this important proof-theoretic result can be applied to the construction of an inductive theorem prover. The Tac prover is capable of proving automatically many non-trivial theorems, which is particularly impressive given the genericity and simplicity of its design.

There are a number of ways to enhance the current system. A general challenge with proof-search and our treatment of equality lies in the handling of logical variables in left hand-side equalities, which obviously hinders automated proof search in several examples; we are currently exploring ways to address this unusual aspect of unification. Another important challenge to address is integrating support for hypothetical contexts when reasoning about HOAS specifications. We also plan to explore the use of flexibilities in polarity assignment left in the design of focused systems for μLJ; in our experience, such choices can

have an important impact on proof-search. Finally, it is crucial that we develop efficient techniques for re-using previously proved lemmas — a common problem in theorem proving.

Acknowledgments. The authors wish to thank Alexandre Viel for his help in developing Tac and the reviewers of an earlier version of this paper for their comments. This work has been supported in part by INRIA through the "Equipes Associées" Slimmer and by the NSF grant CCF-0917140. Opinions, findings, and conclusions or recommendations expressed in this papers are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *J. of Logic and Computation*, 2(3):297–347, 1992.
2. D. Baelde. *A linear approach to the proof-theory of least and greatest fixed points.* PhD thesis, Ecole Polytechnique, Dec. 2008.
3. D. Baelde. On the expressivity of minimal generic quantification. In A. Abel and C. Urban, editors, *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2008)*, number 228 in Electronic Notes in Theoretical Computer Science, pages 3–19, 2008.
4. D. Baelde and D. Miller. Least and greatest fixed points in linear logic. In N. Dershowitz and A. Voronkov, editors, *International Conference on Logic for Programming and Automated Reasoning (LPAR)*, volume 4790 of *LNCS*, 2007.
5. K. Chaudhuri and F. Pfenning. Focusing the inverse method for linear logic. In C.-H. L. Ong, editor, *CSL 2005: Computer Science Logic*, volume 3634 of *LNCS*, pages 200–215. Springer, 2005.
6. J.-Y. Girard. A fixpoint theorem in linear logic. An email posting to the mailing list linear@cs.stanford.edu, Feb. 1992.
7. G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.
8. C. Liang and D. Miller. Focusing and polarization in intuitionistic logic. In J. Duparc and T. A. Henzinger, editors, *CSL 2007: Computer Science Logic*, volume 4646 of *LNCS*, pages 451–465. Springer, 2007.
9. R. McDowell and D. Miller. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science*, 232:91–119, 2000.
10. S. McLaughlin and F. Pfenning. Imogen: Focusing the polarized focused inverse method for intuitionistic propositional logic. In I. Cervesato, H. Veith, and A. Voronkov, editors, *15th International Conference on Logic, Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 5330 of *LNCS*, Nov. 2008.
11. A. Momigliano and A. Tiu. Induction and co-induction in sequent calculus. In M. Coppo, S. Berardi, and F. Damiani, editors, *Post-proceedings of TYPES 2003*, number 3085 in LNCS, pages 293–308, Jan. 2003.
12. P. Schroeder-Heister. Rules of definitional reflection. In M. Vardi, editor, *Eighth Annual Symposium on Logic in Computer Science*, pages 222–232. IEEE Computer Society Press, IEEE, June 1993.
13. C. Schürmann and F. Pfenning. Automated theorem proving in a simple meta-logic for LF. In C. Kirchner and H. Kirchner, editors, *15th Conference on Automated Deduction (CADE)*, volume 1421 of *Lecture Notes in Computer Science*, pages 286–300. Springer, 1998.