Certifying Proof-By-Linking

Kaustuv Chaudhuri, Pablo Donato, Luigi Massacci, Benjamin Werner

Certifying Proof-By-Linking

Kaustuv Chaudhuri[1,2], Pablo Donato[1], Luigi Masacci[3], and Benjamin Werner[2,3]

[1]Inria Saclay, Palaiseau, France
[2]Laboratoire d'Informatique (LIX), UMR 7161, Palaiseau, France
[3]Ecole polytechnique, Palaiseau, France

September 22, 2022

Abstract

Proof-by-linking (PBL) is an interactive theorem proving technique that uses *direct manipulation* UI actions such as drag-and-drop to assert *links* between subformulas of the goal. These links are used by the system to rewrite the goal in a way that reflects the intention of the user performing the manipulation. Because PBL avoids the use of proof *languages*, the same technique can potentially be used as a front-end to any existing formal reasoning system, making PBL a universal interaction method. In this paper we substantiate this claim by showing how to extract formal proofs from PBL derivations in a variety of existing proof languages, using a variety of techniques including *reflection*, *combinators*, and *meta-programming*. Crucially, all our extraction procedures use the native notions of propositions/formulas in the target proof systems, which makes the PBL technique compatible with the existing proof building machinery in these systems. We also provide implementations of the extraction procedures in an extension of the Profint framework, which runs on the client side in any modern web-browser.

1 Introduction

Existing interactive formal reasoning tools and languages are (nearly always) mutually incompatible. This is unsatisfactory for many reasons including fragmentation in the formal methods community and duplication of effort instead of reuse. There have been many attempts in recent years to address this issue in the *back-end*: various pairs of systems are made to exchange formal results by means of translations between their input and output languages, or in some cases by translations to and from more universal languages (e.g., [16, 12]).

The mutual incompatibility problem can also potentially be approached from the *front-end*. Many kinds of logical manipulations such as discharging assumptions, instantiating quantifiers, and composing lemmas, are conceptually universal; yet, a user wishing to carry out such manipulations in a particular proof system must express the wish *in terms of* the formalism of the proof system. A *universal user interface* would instead allow to represent such manipulations directly using physical, reversible, and incremental actions with immediate feedback;[1] a sequence of user actions should then be representable in terms of any formal reasoning language.

One recent approach to this kind of user interface is the *proof-by-linking* (PBL) method [9, 10, 13], which generalizes earlier work on *proof-by-pointing* [14, 5]. In this method, the user is presented the conjecture—the formula to prove—as a manipulable object, where subformulas can be moved around with common user interaction techniques such as drag-and-drop or multi-touch. The user uses such interaction devices to indicate connections or *links* between subformulas of the conjecture; the endpoints of these links are then brought next to each other—in a logically sound manner—and then made to *interact* in predictable ways. For instance, if the user drags a subformula A that occurs in a negative subformula position (i.e., the subformula occurrence behaves like an assumption) to another occurrence of A in a positive subformula position (i.e., it behaves like a conclusion), the goal formula is rewritten so the the two occurrences of A are brought together as the subformula $A \Rightarrow A$, and subsequently simplified to $\top$. The rewrite rules that are used to perform these transformations are *valid*, which is to say that if the rewritten conjecture is a theorem then so is the original formula. The PBL method and its underlying proof calculus is explained in more detail in section 2.

Prior to this work, there have been three main prototype implementations of the PBL technique: *Profound* [9] for first-order classical linear logic, *Profound-Intuitionistic* [10] (aka Profint, the starting point for this paper) for first-order intuitionistic logic, and *Actema* [13] also for first-order intuitionistic logic. None of these implementations produced any independently verifiable proof objects, leaving the main promise of the PBL technique—*universality*—unrealized. (Even without the goal of universality, having certificates is essential because the PBL implementations will never be part of the TCB of any formal system.) In this paper we address

[1]Paraphrasing: `https://www.nngroup.com/articles/direct-manipulation/`.

this deficiency by describing how to extract proof objects from PBL proofs. To be concrete, the reasoning logic we pick in this paper is first-order intuitionistic logic over simply typed λ-terms with $\alpha\beta\eta$-equivalence as its equational theory. This a non-trivial fragment of nearly every mainstream formal reasoning system.

Our aim is in fact more complicated than simply certifying the PBL method itself for this particular logic. We do not propose to formalize yet another proof system for yet another object logic. Instead, we wish to explore the universality of the PBL method by describing *techniques* for extracting formal proof certificates that are adapted to target proof systems. Some desiderata for such certificates include the following: (1) The certificates must use the native types and formulas of the target proof language without any additional axiomatic extensions such as extensionality. In other words, the encoding must be *shallow* and *conservative*. This is essential for compositionality. (2) The certificates must be checkable without relying on sophisticated automated proof search procedures in the target systems. Predictability in user interfaces is essential[2] and often at odds with an over-reliance on automation. (3) The certificates must be compact. At least, they should avoid repeating information that is easily (re)computed. (4) The certificates must be decipherable by human readers, so that any missing functionality in an existing PBL implementation can be manually incorporated.

Our approach relies on translating a PBL derivation in terms of its underlying trace in a variant of the *calculus of structures* proof system [11]. This is a *deep inference* formalism where inference rules are allowed to operate not just at the level of the topmost exposed connectives, but also to connectives that occur deeper in the subformula tree. The rules of the calculus are given in a *contextual* form: a *formula context* is a formula where a single subformula has been replaced with a *hole*. The hole may be *filled* with any other formula to create a *replacement*.[3] Crucially, the free variables of the filler formula are interpreted as *captured* by the binders that are in scope at the hole. Representing formulas and formula contexts as data, that allows for this kind of capture, while retaining a shallow embedding is the main representational challenge; as a benefit, we can then directly prove the soundness of the calculus of structures and then rely on this soundness to accept a simple inductive representation of the CoS trace as a proof object. We illustrate this *reflective* style of formalization using the Calculus of Inductive Constructions with cumulative universes and universe-polymorphic definitions, as supported by Coq; this is covered in section 3.

An alternative strategy avoids encoding formula contexts as data, but instead uses compositions of *transporting combinators* to transfer rewrite rules, viewed as implications, to arbitrary positions inside a formula. For example, if $a \Rightarrow b$ represents a rewrite rule that, reading from conclusion to premise, would transform b to a, then the rewrite that homomorphically transforms $c \wedge b$ to $c \wedge a$ would be represented by the implication $(c \wedge a) \Rightarrow (c \wedge b)$; hence the combinator that transports a rule to the right of a $\wedge$ is represented by the formula $(a \Rightarrow b) \Rightarrow ((c \wedge a) \Rightarrow (c \wedge b))$. Such combinators can easily handle quantifiers; for instance, the combinator that transports a rewrite from $q\,x$ to $p\,x$ into a formula context where x is existentially bound is represented by the formula $(\forall x.\, p\,x \Rightarrow q\,x) \Rightarrow (\exists x.\, p\,x) \Rightarrow (\exists x.\, q\,x)$. The contextual inference rules of CoS formalism as can then be expressed in the form of *local* rewrite rules that are then expanded into their contextual forms by a composition of the transporting combinators. We detail this approach in section 4 for the Lean 3 system; the implementation builds certificates in this style for Coq and Isabelle/HOL as well.

A third formalization strategy that avoids the complexity of the first strategy and the excessive detail of the second strategy is to define a domain-specific language for writing proofs of ordinary formulas in the CoS "style". This DSL is then interpreted by a *meta-program* with respect to the structure of the conjecture formula, simultaneously computing both the formal proof (in the internal representation of the target proof system) and the sequence of steps in the CoS proof, again expressed using the formulas of the target system. In section 5 we give an implementation of such a DSL using the Lean 4 system, which has sophisticated meta-programming support. We reuse not only the expressions (terms and formulas) of Lean 4, but also reuse the same techniques such as the *locally nameless representation* [8], macro system, and quotation features [21] that Lean 4 uses to build its own tactics language.

2 Overview of Proof-by-Linking

As mentioned, we will use first-order intuitionistic logic over simply typed λ-terms as our reasoning logic. Types, terms, and formulas have the following grammar:

$$\begin{aligned}
\alpha, \beta &::= b \mid \alpha \to \beta && \text{(Types)} \\
s, t &::= \mathtt{k} \mid x \mid \lambda x{:}\alpha.\, t \mid s\,t && \text{(Terms)} \\
A, B, \ldots &::= \mathtt{a}\,\vec{t} \mid A \wedge B \mid \top \mid A \vee B \mid \bot \mid A \Rightarrow B \\
&\quad \mid \forall x{:}\alpha.\, A \mid \exists x{:}\alpha.\, A \mid s \doteq t && \text{(Formulas)}
\end{aligned}$$

[2] The *principle of least astonishment*.

[3] We avoid the term *substitution* in this paper since that is often understood in the sense of capture-avoiding substitution. In this paper we intend for free variables to be captured!

where b denotes basic types, $\mathtt{k}$ denotes term constants, and $\mathtt{a}$ denotes predicates that are declared in an ambient *signature*. Atomic formulas are of the form $\mathtt{a}\,t_1\,\ldots\,t_n$ where $\mathtt{a}$ is a predicate of type $\alpha_1 \to \cdots \to \alpha_n \to \mathtt{o}$ in the signature and $t_i : \alpha_i$ (for $i \in 1..n$); we abbreviate this as $\mathtt{a}\,\vec{t}$ where $\vec{t} = [t_1, \ldots, t_n]$. The details of well-typedness for terms and formulas with respect to a given signature are standard and omitted here.

2.1 The Profint Tool

The *proof-by-linking* (PBL) method will be recapitulated here in terms of its implementation as an extension of the Profint tool [10]. The extension is provided in the accompanying materials as a HTML+JavaScript application that can be served with a local web-server; an example Github-hosted instance of this application is also provided.[4] The main page (*launcher*) of this tool has an editable section where the user can input their desired signature and goal formula using the concrete syntax that is described in more detail in appendix B; there are also a number of premade conjectures for the user to try.

Linking Let us consider one such conjecture, the formula $(\mathtt{a}\Rightarrow\mathtt{b}\Rightarrow\mathtt{c})\Rightarrow(\mathtt{a}\Rightarrow\mathtt{b})\Rightarrow(\mathtt{a}\Rightarrow\mathtt{c})$, which is an instance of one of Hilbert's axioms of minimal logic. Clicking on this conjecture from the launcher opens the main Profint proving interface which presents the goal formula as an interactive object. Hovering the mouse over the formula will highlight the smallest subformula that contains the position indicated with the mouse. This subformula can be *grabbed* by pressing the mouse, then *dragged* to a different subformula and then *dropped* there to indicate a link between the subformulas. (This is the usual *drag-and-drop* operation in GUIs.) For example, the leftmost occurrence of $\mathtt{b}$ (the *source*) can be dragged to the other occurrence of $\mathtt{b}$ (the *destination*), which we indicate schematically below with an arrow:

$$(\mathtt{a}\Rightarrow\underset{\smile}{\mathtt{b}}\Rightarrow\mathtt{c})\Rightarrow(\mathtt{a}\Rightarrow\underset{\frown}{\mathtt{b}})\Rightarrow(\mathtt{a}\Rightarrow\mathtt{c}).$$

Profint interprets this drag-and-drop action as an intention of the user to *bring the source to the destination*, so it attempts to rewrite the conjecture in such a way as to leave the linked subformulas as neighbors of an implication, $\mathtt{b}\Rightarrow\mathtt{b}$. In this case, this is achieved by composing the parent implications of the two occurrences of $\mathtt{b}$, which yields:

$$(\mathtt{a}\Rightarrow\mathtt{a}\Rightarrow(\underset{\smile}{\mathtt{b}}\Rightarrow\underset{\frown}{\mathtt{b}})\Rightarrow\mathtt{c})\Rightarrow(\mathtt{a}\Rightarrow\mathtt{c}).$$

At this point, because the linked implication $\mathtt{b}\Rightarrow\mathtt{b}$ is trivially true, it is replaced by $\top$ and the link is *resolved*, leaving:

$$(\mathtt{a}\Rightarrow\mathtt{a}\Rightarrow\top\Rightarrow\mathtt{c})\Rightarrow(\mathtt{a}\Rightarrow\mathtt{c}),$$

which Profint *simplifies* to: $(\mathtt{a}\Rightarrow\mathtt{a}\Rightarrow\mathtt{c})\Rightarrow(\mathtt{a}\Rightarrow\mathtt{c})$.

Copying vs. Moving By default, Profint attempts to *move* the source to the destination; thus, the link $\underset{\smile}{\mathtt{a}}\Rightarrow\underset{\frown}{\mathtt{a}}\wedge\mathtt{a}$ is reduced to $(\underset{\smile}{\mathtt{a}}\Rightarrow\underset{\frown}{\mathtt{a}})\wedge\mathtt{a}$. However, in a proof of this formula the assumption $\mathtt{a}$ has to be used twice, once each at each conjunct. To achieve this in Profint, the dragged formula $\mathtt{a}$ must be dropped while holding the '`ctrl`' key, which turns the *drag operation* from a *move* to a *copy*, indicated in the web browser UI by means of a small '`+`' symbol over the mouse cursor, and which we depict in this paper with a tail on the arrow: $\underset{\smile}{\mathtt{a}}\Rightarrow\underset{\frown}{\mathtt{a}}\wedge\mathtt{a}$.

To resolve such *copy* links, Profint first attempts to make a copy of the largest negatively occurring subformula of the link that contains the source of the link before turning the link into a *move* link; in other words, the link resolution goes as follows: $\underset{\smile}{\mathtt{a}}\Rightarrow\underset{\frown}{\mathtt{a}}\wedge\mathtt{a}$ to $\mathtt{a}\Rightarrow(\underset{\smile}{\mathtt{a}}\Rightarrow\underset{\frown}{\mathtt{a}}\wedge\mathtt{a})$ to $\mathtt{a}\Rightarrow(\underset{\smile}{\mathtt{a}}\Rightarrow\underset{\frown}{\mathtt{a}})\wedge\mathtt{a}$.

Quantifiers and Predicates Consider now the goal $\mathtt{p}\,\mathtt{j}\Rightarrow\exists x.\,\mathtt{p}\,x$. As before, we can drag $\mathtt{p}\,\mathtt{j}$ to $\mathtt{p}\,x$:

$$\underset{\smile}{\mathtt{p}\,\mathtt{j}}\Rightarrow\exists x.\,\underset{\frown}{\mathtt{p}\,x}.$$

The effect, as before, would be to bring the source and destination together, in this case by extruding the scope of $\exists x$:

$$\exists x.\,(\underset{\smile}{\mathtt{p}\,\mathtt{j}}\Rightarrow\underset{\frown}{\mathtt{p}\,x}).$$

Here, since the source and the destination of the link are not precisely the same, Profint cannot (immediately) resolve the link to $\top$. However, this implication holds whenever the arguments to $\mathtt{p}$ are equal, so Profint reduces

[4] `https://cpp2023p33.github.io`

the problem to an equation over these term arguments of the predicate; in this cases, the result is: $\exists x.\, \mathtt{j} \doteq x$. (If the predicate symbols are different in the source and destination, no further resolution is possible and the implication is left as is.)

Of course, not all forms of first-order reasoning can be reduced to extruding the scopes of quantifiers and simplifying. For completeness, it is also essential to *instantiate* the *synchronous/weak* quantifiers, in particular to make progress on any residual equations. For instance, in the above rewritten conjecture $\exists x.\, \mathtt{j} \doteq x$, the proof cannot progress without instantiating x to $\mathtt{j}$. This is done in the Profint interface by highlighting the quantifier to instantiate and then pressing '`w`': the witness term can then be entered directly in place of the bound variable. In this case we would enter $\mathtt{j}$ for the instance, and the resulting formula $\mathtt{j} \doteq \mathtt{j}$ is simplified by Profint to $\top$.

Equational Reasoning Profint maintains all terms in canonical (i.e., β-normal) form. Whenever Profint encounters a positively occurring equation of the form $\mathtt{k}\,\vec{s} \doteq \mathtt{k}\,\vec{t}$ where $\vec{s} = [s_1, \ldots, s_n]$ and $\vec{t} = [t_1, \ldots, t_n]$, it simplifies it to $(s_1 \doteq t_1) \wedge \cdots \wedge (s_n \doteq t_n)$ if $n \geq 1$, and $\top$ otherwise. This is obviously sound; however, it may not be complete in reasoning logics where $\doteq$ is interpreted as definitional equality instead of as $\alpha\beta\eta$-equivalence.

When equations occur in negative context, Profint treats them as rewrite rules: a left-to-right rewrite is attempted on a link with an $\doteq$ assumption, unless the l.h.s. of the equation is not a subterm of the destination, in which case a right-to-left rewrite is attempted. To illustrate, consider:

$$(\forall x.\, \mathtt{p}\, x \Rightarrow \underbrace{x \doteq \mathtt{k}}) \Rightarrow \mathtt{p}\, \mathtt{j} \Rightarrow \mathtt{p}\, \mathtt{k},$$

which Profint reduces to:

$$\mathtt{p}\, \mathtt{j} \Rightarrow (\exists x.\, p\, x \wedge (\underbrace{x \doteq \mathtt{k}} \Rightarrow \mathtt{p}\, \mathtt{k})).$$

Since x is not a subterm of the destination, the equation is used as a rewrite in the right-to-left direction to leave the residue: $\mathtt{p}\, \mathtt{j} \Rightarrow \exists x.\, \mathtt{p}\, x \wedge \mathtt{p}\, x$.

2.2 Core Calculus (of Structures)

As already mentioned, we approach the problem of building proof objects from PBL derivations in two steps: first, we transform a sequence of PBL user actions into a derivation in a core calculus that is a variant of the intuitionistic *calculus of structures* (CoS), closely related to the calculus that was presented in [10]. The CoS derivation has a completely deterministic semantics, unlike PBL itself which requires many non-deterministic choices that must be carefully orchestrated to preserve provability. (This orchestration is covered in [10] and is beyond the scope of the present paper.) Indeed, in this core calculus the links do not even have a directionality. The purpose of the link directions is to resolve certain ambiguities that arise in the PBL approach. In this section, we will draw the link arcs without arrowheads, and give no semantic meaning to the shading for the source/destination subformulas.

Let us now formally define *formula contexts*. A *positive(ly signed) formula context*, written $\mathcal{C}\{\,\}$, has the same syntax as an ordinary formula except there is a single occurrence of the hole, $\{\,\}$, in the place of a positively signed subformula, i.e., a subformula that occurs as a left-descendant of an even number of $\Rightarrow$s. It is defined mutually inductively with the notion of a *negative(ly signed) formula context*, written $\mathcal{A}\{\,\}$, with the following grammar:

$$\begin{aligned} \mathcal{C}\{\,\} &::= \{\,\} \mid A * \mathcal{C}\{\,\} \mid \mathcal{C}\{\,\} * A \mid \mathsf{Q}x{:}\alpha.\, \mathcal{C}\{\,\} \mid \mathcal{A}\{\,\} \Rightarrow B \mid A \Rightarrow \mathcal{C}\{\,\} \\ \mathcal{A}\{\,\} &::= B * \mathcal{A}\{\,\} \mid \mathcal{A}\{\,\} * B \mid \mathsf{Q}x{:}\alpha.\, \mathcal{A}\{\,\} \mid \mathcal{C}\{\,\} \Rightarrow A \mid B \Rightarrow \mathcal{A}\{\,\} \end{aligned} \tag{1}$$

where $* \in \{\wedge, \vee\}$ and $\mathsf{Q} \in \{\forall, \exists\}$. The *replacement* of the hole in $\mathcal{C}\{\,\}$ (resp. $\mathcal{A}\{\,\}$) with a formula F is a new formula written $\mathcal{C}\{F\}$ (resp. $\mathcal{A}\{F\}$). In this replacement $\mathcal{C}\{F\}$, any free variables of F must be bound variables that are in the scope for the hole in $\mathcal{C}\{\,\}$, and these variables are intended to be *captured* by the corresponding binding occurrences. For example, if $\mathcal{C}\{\,\}$ is $\mathtt{a} \wedge ((\forall x.\, \{\,\}) \Rightarrow \mathtt{c})$, then $\mathcal{C}\{\mathtt{p}\, x \Rightarrow \bot\}$ is the formula: $\mathtt{a} \wedge ((\forall x.\, \mathtt{p}\, x \Rightarrow \bot) \Rightarrow \mathtt{c})$. Note that $\mathcal{C}\{F\}$ and $\mathcal{A}\{F\}$ are both formulas for any formula F and contexts $\mathcal{C}\{\,\}$ and $\mathcal{A}\{\,\}$; hence, it is perfectly acceptable to build nested replacements such as $\mathcal{A}\{\mathcal{C}\{F\} \Rightarrow \mathcal{A}'\{F\}\}$. For the replacement $\mathcal{C}\{F\}$ (resp. $\mathcal{A}\{F\}$), we say that F is *in the context* $\mathcal{C}\{\,\}$ (resp. $\mathcal{A}\{\,\}$).

The core CoS system will be given in the form of single premise inference rules where both premise and conclusion are formulas. These rules will generally be specified in a *contextual form*, which is to say that the premises and conclusions of each rule will be written as replacements in the same context. There are three main kinds of rules: (1) *Positive linking rules* are rules that operate on a positively occurring $\Rightarrow$-formula whose operands contain the source and the destination of the indicated link. (2) *Negative linking rules* are rules that operate on a negatively occurring $\wedge$-formula whose operands contain the source and the destination of the indicated link. (3) *Simplification rules* are rules that do not involve a link. They may operate on formulas in both positive and negative contexts. The full list of rules is given in appendix A; here, we describe some of the important rules.

Ordinary Logical (a.k.a. "Switch") Rules These are positive linking rules where either the source or the destination of the indicated link occurs as a strict subformula of the left or the right of the $\Rightarrow$-formula. A characteristic example is the rule `goal_ts_and_l`:

$$\frac{\mathcal{C}_0\{(\mathcal{C}_l\{A\} \Rightarrow \mathcal{C}_r\{B\}) \wedge F\}}{\mathcal{C}_0\{\mathcal{C}_l\{A\} \Rightarrow (\mathcal{C}_r\{B\} \wedge F)\}}$$

Observe that the nesting of positive signed contexts guarantees that the source and destination of the indicated link are oppositely signed subformulas of the overall formula. Furthermore, the link is drawn without arrowheads, so it is applicable even if the source is B and the destination A.

Most of the rules in this category have this flavor, where the link is permuted into the context of some other connective. Two rules break this pattern; one is `goal_or_ts_l`:

$$\frac{\mathcal{C}_0\{(\mathcal{C}_l\{A\} \Rightarrow \mathcal{C}_r\{B\}) \wedge (F \Rightarrow \mathcal{C}_r\{B\})\}}{\mathcal{C}_0\{(\mathcal{C}_l\{A\} \vee F) \Rightarrow \mathcal{C}_r\{B\}\}}$$

Here, the link persists into one of the two conjuncts in the premise, while in the other conjunct the right hand side is duplicated but lacks a link. The other atypical rule is `goal_ts_imp_l`:

$$\frac{\mathcal{C}_0\{(\mathcal{C}_l\{A\} \wedge \mathcal{C}_r\{B\}) \Rightarrow F\}}{\mathcal{C}_0\{\mathcal{C}_l\{A\} \Rightarrow (\mathcal{C}_r\{B\} \Rightarrow F)\}}$$

In this case, the link in the premise is in a negative context, so subsequent rules above will be negative.

Negative Linking Rules These rules deal with neighboring conjuncts in a negative context, and nearly every rule amounts to a permutation of the link into a deeper negative context. A prototypical example is `asms_or_l_l`:

$$\frac{\mathcal{A}_0\{(\mathcal{C}_l\{A\} \wedge \mathcal{C}_r\{B\}) \vee F\}}{\mathcal{A}_0\{\mathcal{C}_l\{A\} \wedge (\mathcal{C}_r\{B\} \vee F)\}}$$

Observe that the premise in the above rule itself has the link in a negative context, forcing only negative linking rules to be applied above. The only way to exit this *phase* of negative rules is with the rule `asms_imp_l_l` (and its symmetric variant), which brings the link back to a positive context in its premise.

$$\frac{\mathcal{A}_0\{(\mathcal{C}_l\{A\} \Rightarrow \mathcal{C}_r\{B\}) \Rightarrow F\}}{\mathcal{A}_0\{\mathcal{C}_l\{A\} \wedge (\mathcal{C}_r\{B\} \Rightarrow F)\}}$$

Link Resolution These rules apply when the source and destination of the link are next to each other in a positive context; in every case, the link disappears in the premise. The simplest case is when the source and destination are both atoms of the same predicate, which is captured by the `init` rule:

$$\frac{\mathcal{C}_0\{(s_1 \doteq t_1) \wedge \cdots \wedge (s_n \doteq t_n)\}}{\mathcal{C}_0\{\mathtt{p}\, s_1 \cdots s_n \Rightarrow \mathtt{p}\, t_1 \cdots t_n\}}$$

When the left operand of the $\Rightarrow$ is an $\doteq$-atom, then the rule is interpreted as a term rewrite captured by the `rew` rule, which has two symmetric forms:

$$\frac{\mathcal{C}_0\{F\langle t\rangle\}}{\mathcal{C}_0\{s \doteq t \Rightarrow F\langle s\rangle\}} \qquad \frac{\mathcal{C}_0\{F\langle s\rangle\}}{\mathcal{C}_0\{s \doteq t \Rightarrow F\langle t\rangle\}}$$

Here, $F\langle s\rangle$ stands for a formula F that contains several (≥ 1) occurrences of the subterm s, all of which are replaced with t in $F\langle t\rangle$.

Finally, if neither of the above rules apply, then the link is simply removed, restoring the original formula. Note that the links we have been drawing are just notational devices to highlight the source of the rules from the initial user drag-and-drop interaction. In the core calculus, the links are not part of the syntax. Therefore, while we can write this kind of link removal as an inference rule, it is not present as such in the CoS trace, so we indicate it with a dashed line.

$$\frac{\mathcal{C}_0\{A \Rightarrow B\}}{\mathcal{C}_0\{A \Rightarrow B\}}$$

Contraction In the CoS derivation, the contraction rule `contr` is represented simply as follows:

$$\frac{\mathcal{C}_0\{A \Rightarrow A \Rightarrow B\}}{\mathcal{C}_0\{A \Rightarrow B\}}$$

In the Profint interface, this kind of contraction can be triggered with a '`ctrl+click`' on the implication whose antecedent is to be duplicated. However, it is more likely that the contraction happens implicitly because the user indicates a drag-and-drop action in *copy* mode. Thus, the above `contr` rule will more likely be seen in the following guise:

$$\frac{\mathcal{C}_0\{\mathcal{C}_l\{A\} \Rightarrow \mathcal{C}_l\{\underbrace{A}\} \Rightarrow \mathcal{C}_r\{\underbrace{B}\}\}}{\mathcal{C}_0\{\mathcal{C}_l\{\underbrace{A}\} \Rightarrow \mathcal{C}_r\{\underbrace{B}\}\}}$$

Other structural rules such as *weakening* or *exchange* are not needed for completeness of the PBL method. (The Profint tool nevertheless supports weakening by '`alt+click`'.)

Instantiation These are a pair of rules that are used to instantiate synchronous/weak quantifiers. The instantiation term t in each case must be well-formed in the typing context formed by the variables in scope at the where the quantified formula occurs.

$$\frac{\mathcal{C}_0\{[t/x]A\}}{\mathcal{C}_0\{\exists x.\, A\}} \qquad \frac{\mathcal{A}_0\{[t/x]A\}}{\mathcal{A}_0\{\forall x.\, A\}}$$

Simplifications Profint uses these rules to remove occurrences of $\top$ and $\bot$ eagerly. All of these rules are based on logical equivalences and hence have no chance of affecting provability. The following are some examples of simplification rules:

$$\frac{\mathcal{C}_0\{A\}}{\mathcal{C}_0\{A \wedge \top\}} \qquad \frac{\mathcal{C}_0\{\top\}}{\mathcal{C}_0\{\bot \Rightarrow A\}} \qquad \frac{\mathcal{C}_0\{\top\}}{\mathcal{C}_0\{\forall x.\, \top\}}$$

As mentioned earlier, Profint also treats congruence as a simplification rule because of the weak $\alpha\beta\eta$-equality assumed for λ-terms, which interprets term constants as *constructors*, i.e., as injective.

$$\frac{\mathcal{C}_0\{(s_1 \doteq t_1) \wedge \cdots \wedge (s_n \doteq t_n)\}}{\mathcal{C}_0\{\mathtt{k}\, s_1 \cdots s_n \doteq \mathtt{k}\, t_1 \cdots t_n\}}$$

From the Profint user interface, the CoS trace corresponding to a PBL proof can be obtained in terms of the *PDF* export option (drop-down in the top right). This will generate a self-contained LaTeXfile that merely displays the completed CoS inference figure.

3 Reflective Certification (Coq)

The goal of this section will be to certify a CoS derivation, as defined in section 2.2 and appendix A, as a valid proof of a formula/proposition in a target proof system; in this section we will target Coq.[5] We would like to use PBL to generate a replacement for the proof place-holder in the following Coq theorem, where `A` and `C` are Coq formulas (of type `Prop`). The premise (`prem`) is the formula at the top of the CoS proof; if the formula `C` is a theorem, then the formula `A` will be `True`.

```
Goal (prem : A) : C.
Proof. (* proof place-holder *) Qed.
```

3.1 Overall Structure

To meet the desiderata laid out in the introduction, particularly compactness, we would like to represent formula contexts simply using the simple type `path` below, where each natural number denotes the operand to descend into for connectives and quantifiers, starting from the topmost connective of the subformula tree. We would also like to have a lightweight inductive definition of the CoS rule names, `rule_name`; in particular, the rule names are not allowed to store the goal formulas as that involves too much repetition. We thus have:

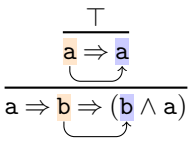
[5] Tested on versions 8.15.2 and 8.16.0

```
Inductive rule_name : Type :=
  | goal_ts_and_l | goal_ts_and_r | ··· | init.
Definition path := list nat.
Definition deriv := list (rule_name * path).
```

To say that a derivation `d : deriv` is a proof of the `P : Prop`, we first define a recursive fixed point `check : deriv -> Prop -> Prop` such that the predicate `check d P` is true only if `d` indeed is a valid CoS proof of `P`. In other words, it satisfies the following meta-theorem:

```
Theorem correctness (d : deriv) (P : Prop) :
  check d P -> P.
```

As an example, consider the goal `a -> b -> b /\ a` for which the user might have built the following PBL derivation:



Here is the proof extracted by Profint:

```
Context {a b : Prop}.
Goal (prem : True) : a -> b -> b /\ a.
Proof.
  pose (d := [
     (* list is backwards -- last to first *)
     (goal_ts_and_l,   [1]); (init, [1; 0]);
     (simp_and_true_r, [1]); (init, [])
  ] : deriv);
  apply (correctness d);
  solve_check. (* solves "check" goals *)
Qed.
```

3.2 Formula Contexts

The informal grammar of contexts, (1), requires quite a bit of care to formalize because of the requirement that replacements involve variable capture. Since Coq has no built in mechanism for dealing with expressions with free variables using nominal techniques (e.g., [3, 18, 22]), we need some mechanism for representing such objects with the features Coq does provide, λ-abstractions. Stated simply, a `Prop` with one free variable of type `A` would be represented with a λ-abstraction of type `A -> Prop`, etc.

We can then encode contexts as follows: a context that can accept a filler with free variables of types $\mathtt{A}_1$, $\mathtt{A}_2$, ..., $\mathtt{A}_n$ (in that nesting order) should have type: `ctx [`$\mathtt{A}_1$`; `$\mathtt{A}_2$`; ···; `$\mathtt{A}_n$`]`; in other words, `ctx : list Type -> Type`. We actually need to be slightly more general, since positive and negative contexts contexts are defined mutually inductively. This can be achieved by means of an extra parameter `side`, giving:

```
Inductive side : Type := Pos | Neg.
Definition flip side :=
  match side with Pos => Neg | _ => Pos end.
Polymorphic Inductive
  ctx : side -> list Type -> Type :=
| Hole : ctx Pos nil
| AndL side Ts : ctx side Ts -> Prop -> ctx side Ts
| AndR side Ts : Prop -> ctx side Ts -> ctx side Ts
··· (* OrL, OrR, ImpR similar *)
| ImpL side Ts :
    ctx (flip side) Ts -> Prop -> ctx side Ts
| ExD A side Ts :
    (A -> ctx side Ts) -> ctx side (A :: Ts)
··· (* AllD similar *).
```

The definition `ctx` is a *universe polymorphic* definition because the different occurrences of `ctx` in the constructors are defined at different universes depending on the universes of their argument `Type`s. Without universe

polymorphism, we would require an auxiliary definition using `sigma` types to thread a proof that the various `Type` arguments are compatible. Indeed, this is how the above definition would be written in Lean, which does not have cumulativity; we will have to explicitly insert the universe lifting and lowering functions to align the universes of all the type arguments. (Such auxiliary definitions would be required regardless of the system if the argument types had any internal dependencies; we do not currently allow such uses of dependent types in our reasoning logic.)

Given a list of types $[T_1; \ldots; T_n]$ and a target type `U`, we recursively define the notation $[T_1; \ldots; T_n] \triangleright U$ to stand for the type $T_1 \to \cdots \to T_n \to U$. Formulas that can fill the hole in `ctx _ Ts` thus must have type `Ts` $\triangleright$ `Prop`. This allows us to define the replacement operation as follows:

```
Fixpoint ctx_place side Ts (Cx : ctx side Ts) :=
match
  Cx in (ctx _ Us) return (Us ▷ Prop) -> Prop
with
| Hole => fun A => A
| AndL _ _ Cx B => fun A => Cx{{A}} /\ B
| AndR _ _ B Cx => fun A => B /\ Cx{{A}}
··· (* OrL, OrR, ImpL, ImpR similar *)
| ExD T _ _ Cx =>
    fun A => exists (x : T), (Cx x){{A x}}
··· (* AllD similar *)
end where "Cx {{ A }}" := (ctx_place _ _ Cx A).
```

As an example, consider the negative context $\forall x{:}\mathtt{i}.\,\{\,\} \Rightarrow \mathtt{p}\,x$ and the formula $\mathtt{p}\,(\mathtt{f}\,x)$ containing one free variable x; replacing this in the context should produce the formula $\forall x{:}\mathtt{i}.\,\mathtt{p}\,(\mathtt{f}\,x) \Rightarrow \mathtt{p}\,x$. In Coq:

```
Context (i : Type) (p : i -> Prop) (f : i -> i).
Definition Ax :=
  AllD i _ _ (fun x => ImpL Neg _ Hole (p x)).
Check Ax. (* ctx Neg [i] *)
Eval compute in Ax{{fun x => p (f x)}}.
  (* (forall x : i, p (f x) -> p x) : Prop *)
```

3.3 Paths

The `ctx` definition allows for a fairly obvious definition of `ctx_place`, which can easily shown to be correct. In fact, we can even prove that `ctx Pos Ts` is covariant, and `ctx Neg Ts` contravariant, for a suitable notion of entailment (see below). However, the actual terms of type `ctx _ _` are too unintuitive to use in practice for CoS proofs because of their *inside-outness*; this is immediately apparent in the `Ax` example above. These context terms are also far from compact since it duplicates parts of the goal formula.

We will now see how to *compute* these context terms on demand using a much lighter indexing scheme in formulas: lists of natural numbers—`path`s—where each number denotes the operand to descend along, starting from the top of the subformula tree. Specifically, given a `path` and a formula `A : Prop` as inputs, we will compute a a list of types `Ts`, a context `Cx : ctx _ Ts` and a filler formula of type `F : Ts` $\triangleright$ `Prop` such that `Cx{{F}} <-> A`.

Observe that this computation cannot be total because the `ctx` type covers a fraction of possible `Prop`s. One way to encode a partial computation in a total recursive function space is to use the `Option` type, but even with such a device it is not a simple matter to encode the computation as a recursive fixpoint, mainly because the `Prop` type is itself not inductively defined and therefore has no induction principle. We therefore instead encode the computation *relationally*, as the inductively defined relation `resolve`:

```
(* recall: path := list nat *)
Inductive resolve : Prop -> path -> forall side Ts,
    ctx side Ts -> (Ts ▷ Prop) -> Prop :=
| Here A :
    resolve A nil Pos nil Hole A
| And0 A B path side Ts Cx F :
    resolve A path side Ts Cx F ->
      resolve (A /\ B) (0 :: path)
        side Ts (AndL side Ts Cx B) F
... (* And1, Or0, Or1, Imp0, Imp1 similar *)
| Ex0 T A path side Ts Cx F :
    (forall x, resolve (A x) path
                  side Ts (Cx x) (F x)) ->
      resolve (exists x, A x) (0 :: path)
        side (T :: Ts) (ExD T side Ts Cx) F
... (* All0 similar *).
```

We then prove its coherence property:

```
Lemma resolve_place A path side Ts Cx F :
  resolve A path side Ts Cx F -> Cx{{F}} <-> A.
```

We prove this meta-theorem easily in Coq by making use of the `setoid_rewrite` facility that allows for rewriting up to logical equivalence in terms of type `Prop`. This equivalence can be turned into definitional equality with the use of propositional extensionality; this will make certain computations to come slightly more efficient, but obviously at the cost of assuming extensionality.

3.4 The `check` Fixpoint

We have most of the machinery in place to start defining the `check` predicate. We will write it as a recursive fixpoint on the derivations; at each step in the derivation, the fixpoint will assert that the conclusion of the CoS rule has the right structure, and that the rest of the derivation can (recursively) check the premise of the rule. Unfolding of the `check` definition fully will therefore generate a nested $\exists\wedge$ formula.

Here is the general shape of the `check` definition, where we have highlighted one particular rule, `goal_ts_and_r`.

```
Fixpoint check (d : deriv) (goal : Prop) : Prop :=
match d with
| nil => goal
| (rule_name, path) :: d =>
    match rule_name with
    ...
    | goal_ts_and_r =>
        exists Ts Cx A B C,
        resolve goal path Pos Ts Cx ?1
        /\ check d Cx{{ ?2 }}
    ...
    end
end.
```

The above definition has two missing terms, denoted with `?1` and `?2`. By the type of `resolve`, we know that `?1` must be of type `Ts ▷ Prop`. By the fact that this is the case for the CoS rule named `goal_ts_and_r`, we know that the filler formula in the context in this rule must have the shape $A \Rightarrow (C \wedge B)$. However, this latter formula would have the Coq type `Prop`, which is not compatible with `Ts ▷ Prop`.

To make it fit, we need to find the equivalent of the $\Rightarrow$ and $\wedge$ connectives that work on the type `Ts ▷ Prop`. Such connectives are definable as recursive fixpoints that follow the structure of the type list `Ts`. Here, for instance, is how we would lift the `/\` connective on `Prop` to this type.

```
Fixpoint ho_and Ts :=
  (Ts ▷ Prop) -> (Ts ▷ Prop) -> (Ts ▷ Prop) :=
match Ts with
| nil => (fun p q => p /\ q)
| A :: Ts =>
    (fun (p q : A -> (Ts ▷ Prop)) (x : A) =>
       ho_and Ts (p x) (q x))
end.
```

We can similarly lift every first-order connective at type `Prop` to `Ts ▷ Prop`, and use this to finish the case of `check` above. The prefix `ho_` is used to signify that these are (potentially) "higher-order" connectives; however, as we are using the type `Ts ▷ Prop` to stand for a proposition with free variables of types `Ts`, the definition `ho_and` etc. are not truly higher-order.

Finally, here is how we complete the definition of `check`:

```
Fixpoint check ... := ...
  | goal_ts_and_r =>
      exists Ts Cx (A B C : Ts ▷ Prop),
      resolve goal path Pos Ts Cx
        (ho_imp Ts A (ho_and Ts C B))
      /\ check d Cx{{ ho_and Ts C (ho_imp Ts A B) }}
...
```

3.5 Instantiation

The only rules of CoS that cannot be indicated with a combination of subformula paths and rule names are the two instantiation rules which need to supply the instantiation terms as well. We can store such terms easily in `rule_name` inductive type:

```
Inductive rule_name : Type := ...
| inst_r {T : Type} T
| inst_l {T : Type} T
...
```

Like the filler formulas in contexts, these instantiation terms must also allow for their free variables to be captured by the variables in scope at the hole. Following the outline of the previous sections, we would therefore expect that the type of the instantiation terms would have a type like `Ts ▷ U` for some type U. While we could make this change in the types of the `inst_r`/`inst_l` constructors, it is better to leave them in the more general form and force the type into the right form using heterogeneous equality in the `check` fixpoint. The case for `inst_r` in `check` is then:

```
Fixpoint check ... := ...
| @inst_r U u =>
    exists T Ts Cx A (t : Ts ▷ T),
    resolve goal path Pos Ts Cx (ho_ex T Ts A)
    /\ u ~= t (* ensures U = Ts ▷ T *)
    /\ check d Cx{{ ho_apply Ts T A t }}
...
```

where the following are defined like `ho_and` above:

```
Fixpoint ho_ex T Ts :
   (T -> Ts ▷ Prop) -> (Ts ▷ Prop) := ....
Fixpoint ho_apply Ts T R :
   (T -> (Ts ▷ R)) -> (Ts ▷ T) -> (Ts ▷ R) := ....
```

3.6 Correctness and `solve_check`

To assemble the framework described in section 3.1, we need two final ingredients. The first is the `correctness` theorem, which is a straightforward but long proof by induction on the structure of the CoS derivation. Many of the cases have a similar structure and are therefore amenable to carefully tuned tacticals using the `match goal` feature.

Finally, we need a way to automatically discharge `check` goals that are generated by backchaining on the `correctness` theorem. Since `check` is defined by recursion on the derivation, it can be fully expanded at once, leaving a nested $\exists\wedge$ formula. Fortunately, the first conjunct of every $\wedge$ in this goal is a `resolve` formula, which can be computed by repeating the `econstructor` tactic, since there is always at most one case that can be matched (without needing to guess any variable instances). Moreover, once this `resolve` goal has been fully discharged, the `side`, `Ts`, `Cx` and `F` formulas (as in the `resolve_place` theorem) will be fully instantiated, which can then be used to compute the replacements in the other conjuncts. This deceptively simple strategy, which is just a trifle more complicated than `repeat econstructor`, is implemented as the Ltac tactic `solve_check`.

3.7 Discussion: `init` and `congr`

The benefit of the `correctness` theorem is a formal verification that any CoS derivation that passes `check` will be sound. This strong property can be established thanks to the fact that `rule_name` is inductively defined. Unfortunately, this also means that the set of "rules" that can be used as CoS rules is fixed once and for all. A consequence is that it is not possible to add rule *schemas* that are not uniform instances of existing rules. Good examples of such rules are the `init` and `congr` rules of Cos (see sec. 2.2 and appendix A) that each have premises that are built from the *type* of the predicate or term constant respectively; in particular, they produce n-ary conjuncts in the premises. Since predicate and term constants are open ended—one can always add new ones—there is a potentially infinite numbers of types that must be handled by such `init` and `congr` constructors of the `rule_name` type.

The way this restriction manifests in practice is that Profint is not able to export proofs in this style for a CoS derivation such as the one on the left below:

$$\dfrac{\dfrac{\dfrac{\dfrac{\top}{\mathtt{j} \doteq \mathtt{j}}}{\exists x.\, \mathtt{j} \doteq x}}{\exists x.\, \mathtt{p}\,\mathtt{j} \Rightarrow \mathtt{p}\,x}\,\textsc{init}}{\mathtt{p}\,\mathtt{j} \Rightarrow \exists x.\, \mathtt{p}\,x} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\top}{\mathtt{p}\,\mathtt{j} \Rightarrow \mathtt{p}\,\mathtt{j}}\,\textsc{init*}}{\exists x.\, \mathtt{p}\,\mathtt{j} \Rightarrow \mathtt{p}\,x}}{\exists x.\, \mathtt{p}\,\mathtt{j} \Rightarrow \mathtt{p}\,x}}{\mathtt{p}\,\mathtt{j} \Rightarrow \exists x.\, \mathtt{p}\,x}$$

Instead, Profint will require the derivation to be in the form on the right above, where the `init*` rule stands for $\frac{\mathcal{C}\{\top\}}{\mathcal{C}\{A \Rightarrow A\}}$.

4 Compositional Certificates (Lean3)

In this section we present an alternative mechanism for exporting CoS derivations to target systems that can be used in the absence of features like universe polymorphism (and cumulativity). While this alternative will lack a theorem like `correctness` (sec. 3.6), it will nevertheless be able to certify every concrete CoS derivation. We present this alternative encoding in Lean 3,[6] although in the accompanying supplementary material we also provide certificates for Coq and Isabelle/HOL in this style.

4.1 Transporting Combinators

Unlike the reflective encoding that directly represented formula contexts, here we are going to take the approach of "transporting" inference rules *to* the hole in a formula context. Suppose that `a → b` is an inference rule with premise `a` and conclusion `b`; we can then build an inference rule `a ∧ c → b ∧ c` simply by exploiting functoriality of $- \wedge$ `c`. The remaining propositional connectives are also similar, with the sole exception being the left of implications which are contra-functorial.

```
variables {a b c : Prop}
theorem go_left_and :
   (a → b) → (a ∧ c → b ∧ c) := ···
/- go_right_and, go_left_or, go_right_or,
   go_right_imp similar -/
theorem go_left_imp :
   (b → a) → ((a → c) → (b → c)) := ···
```

As before in the reflective encoding, the main difficulty is with the quantifiers. Here, we make use of the following trick: when descending into a connective, we assume that the inference rule we are transporting is abstracted over the quantified variable:

[6] Tested on version 3.46.0

```
universe u
variables {T : Type u} {p q : T → Prop}
theorem go_down_ex :
   (∀ x, p x → q x) → (∃ x, p x) → (∃ x, q x) := ···
/- go_down_all similar -/
```

In this encoding, the type of the variable `x` in the abstracted inference rule `(∀ x, p x → q x)` must precisely match the type of the quantified variable in the formula the rule is being transported into. It is the job of the exporter from CoS to build an inference rule of the correct type. (An alternative would have been to make the transported inference rule polymorphic over the type of the variable, but in our experience this causes Lean's type inference to get stuck in any non-trivial example.)

4.2 Rule Combinators

Each of the rules of CoS (appendix A) corresponds to a theorem in Lean For CoS rules in a positive context, the theorem is an implication from the premise to the conclusion of the CoS rule specialized to an empty context; likewise, CoS rules in a negative context become implications from the conclusion to the premise. The following are a few characteristic examples.

```
/- positive context rules -/
theorem init : true → (a → a) := ···
theorem goal_ts_and_l :
  ((a → b) ∧ c) → (a → (b ∧ c)) := ···
theorem goal_ts_ex :
  (∃ x, a → p x) → (a → ∃ x, p x) := ···
```

```
/- negative context rules -/
theorem asms_or_l_l :
  (a ∧ (b ∨ c)) → ((a ∧ b) ∨ c) := ···
theorem asms_imp_l_l :
  (a ∧ (b → c)) → ((a → b) → c) := ···
```

```
/- simplification rules -/
theorem simp_goal_and_true : a → (true ∧ a) := ···
theorem simp_goal_false_imp : true → (false → a) := ···
```

As an illustration, here is a proof of `a → b → b ∧ a` using the transport and rule combinators, where the `show` lines are not necessary but used to display the premise computed by the `refine` tactic immediately above.

```
example (prem : true) : a → b → b ∧ a := begin
  refine (go_right_imp goal_ts_and_l) _,
  show a → (b → b) ∧ a,
  refine (go_right_imp (go_left_and init)) _,
  show a → true ∧ a,
  refine (go_right_imp simp_and_true_r) _,
  show a → a,
  refine init _,
  show true,
  exact prem
end
```

4.3 Equations and Congruence

The rule combinator `init` above can only handle conclusions of the form $\mathcal{C}\{A \Rightarrow A\}$, which has the same issue with generality as discussed in sec. 3.7. As before, the issue here is that the transport combinators require the rule combinators to have compatible types, but the type of the premise of the `init` rule can have a variable number of conjuncts depending on the arity of the predicate symbol. Fortunately, since our collection of rule combinators are not fixed since they are not constructors of an inductive type, it is possible to deal with this problem by generating specialized forms of `init` for each predicate in the signature, and likewise specialized `congr` rules for each term constant.

However, we can present an alternative that uses the property of Lean's `refine` tactic that allows it to leave holes anywhere in the term (using `_`), which are then promoted to new subgoals. We can use this facility to

simply *state* the required specialized type of `init` and `congr`. Then, because the generated subgoal for this hole has a predictable form, we can write a small tactical (called `profint_discharge` – details omitted here) to prove it automatically. Thus, here is how the link $\texttt{p j} \Rightarrow \exists x.\, \texttt{p}\, x$ is handled:

```
example (prem : ∃ x, j = x) : p j → ∃ x, p x :=
begin
  refine goal_ts_ex _,
  show ∃ x, p j → p x,
  refine (go_down_ex
           (fun x, (_ : j = x → p j → p x))) _,
    { profint_discharge },
  exact prem
end
```

4.4 Instantiation

A useful property of the transport combinators `go_down_ex` and `go_down_all` is that their argument abstracts over the variable being descended; thus, the rule being transported by these combinators is brought into the scope of all the quantifiers encountered on the way. Instantiation is therefore considerably simpler in this case than in sec. 3, since we can just use the variables directly. The `inst_r` rule is obvious:

```
theorem inst_r t : p t → ∃ x, p x := ···
```

Here is an illustration:

```
example (prem : ∀ (x : T), x = x)
      : ∀ (x : T), ∃ y, x = y :=
begin
  refine (go_down_all (fun u,
            (inst_r u : _ → ∃ y, u = y))) _,
  show ∀ (x : T), x = x,
  exact prem
end
```

Observe that when descending the ∀ the abstracted variable is (re)named as `u` instead of `x`. There is no requirement to use the same name as in the goal formula, nor indeed is there any way to force it to be the same name. Also observe that the application of `inst_r u` has to be given an explicit target type because Lean 3 is not able to automatically synthesize the argument `p` in the definition of `inst_r`. (A similar explicit type will be needed for `inst_l` in the argument position.) This problem does not exist in Coq that appears to have more finely tuned heuristics for term reconstruction.

4.5 A Note on Isabelle/HOL

We have also used this compositional style to generate proof objects in Isabelle/HOL, written as declarative proofs in the Isar language. The biggest difference in the Isabelle/HOL output vs. the Lean 3 and Coq versions is the treatment of transport combinators like `go_down_ex`. Unlike in Lean and Coq, where lemma application is the same as function application, in Isabelle/HOL one needs to explicitly use the `impE` rule of natural deduction, which can only be applied to named facts. This forces all inner traversals to be lifted to facts (using the `have` Isar command). The Isar proofs therefore seem more like a puzzle assembled from lots of *little* proofs rather than one big linear proof in the case of Lean 3 or Coq.

5 Meta-Programming (Lean4)

The compositional proofs built in the previous section have low requirements from a target proof system, which makes them potentially easy to adapt. However, the compositional proofs do not meet the compactness requirement. Firstly, the type inference engine requires typing hints when traversing quantifiers and for instantiations, which repeats parts of the goal formula (or some intermediate formula) in the proof object. Secondly, the `init` and `congr` rules are also implemented by repeating the conclusion and premise formulas in the type of the omitted subterm. Most importantly, these transporting tactics are very cumbersome to use by humans because they require a perfect matching between the transport lemma and the form of the goal.

```
example (prem : True) :
    (∃ (w : i), ∀ (z : i), r z w) →
    (∀ (x : i), ∃ (y : i), r x y) := by
  within        use goal_ts_all
  within d      use goal_ex_ts
  within d 2    use goal_ts_ex
  within d 3    use goal_all_ts
  within d 4    use init
  within d, i w use inst_r w
  within d 3, r use congr
  within d 3    use simp_asms_and_true
  within i x, d use inst_r x
  within d 2    use congr
  within d      use simp_goal_all_true
  within        use simp_goal_all_true
  exact prem
```

Figure 1: CoS proof written in a Lean 4 DSL

To improve matters requires us to design a domain specific language (DSL) that more closely matches our intuition of the CoS rule steps, accompanied by some meta-programs to interpret the DSL in the target proof system. Many mainstream proof systems support meta-programming, either in the form of tactics programming languages such as Ltac 2 [17] or Mtac [23] (in Coq), or by providing a plugin API to be used by the implementation language of the proof system, or in some cases by providing an LCF-style API to build arbitrary theorems that that can be used by traditional programming languages. Here we give an implementation in Lean 4,[7] which has comprehensive meta-programming support – nearly all of the Lean 4 system is implemented in its own language.

We implement a tactic form called `within ... use` that uses a slight extension of the `path` type from sec. 3, together with the rule combinators from sec. 4. The syntax of `within` is inspired by the existing `conv` tactic in Lean, although there isn't a perfect parallel. (There are also some such tactics in HOL4, particularly `irule_at` and `drule_at`, but their use is far more limited than the general transfer combinators we use in this work.) An example of the use of the `within ... use` tactic is shown in figure 1; the various parts will be explained in the subsections below.

5.1 Paths and Transport

In the reflective implementation in sec. 3, paths were implemented as lists of natural numbers. In Lean 4, since its own syntax is exposed to the Lean 4 language in the form of the `Syntax` type, we can design a particular data structure to represent paths.

```
inductive Direction : Type where
  | l | r | d | i (x : Ident)
def Path := Array Direction
```

Here, `Ident` is a particular kind of syntax. The constructors `l` and `r` are used to descend into the left and right of binary connectives, `d` to descend into a quantifier without naming the variable, and `i x` to descend into a quantifier by giving the bound variable the name `x`. We interpret `i x` as a binding occurrence of `x` whose scope is the rule combinator term that follows that `use` keyword. If we use `d` instead of `i x` to descend through a quantifier, then the quantified variable becomes unavailable to form terms in the payload of the `within ... use` tactic. For ease of use we define a parser from Lean 4 syntax to `Path`s, where the constructors `l`, `r`, and `d` can take an optional argument to denote repetitions; i.e., `r 3` stands for `r, r, r` etc.

To transport a rule along a path in a goal, we proceed recursively on the path from left to right. For every direction in the path we bring the goal formula to WHNF and compare the direction to its topmost connective. Both (weak head) normalization and pattern matching on Lean kernel expressions (using `Lean.Expr`) can be done directly inside Lean using a combination of the `partial` keyword and `match` expressions. The rule formula being transported is kept as mere syntax (using the `Term` type, which is a variant of `Syntax`), whereas the goal being analyzed is a Lean kernel expression. Whenever an `i` direction is successfully processed, the corresponding identifier is added to the local context present inside the `TacticM` monad.

Here is a part of the `transport` function that shows the case of transporting a rule to the left of a conjunction:

[7] Lean 4 is under active development and its API is unstable; the implementation we provide with this draft is known to work in the `lean4-nightly:2022-09-17` version.

```
1 partial def transport (rule : Term) (goal : Expr)
2              (path : Path) (pos : Nat)
3          : TacticM Term := do
4   if pos ≥ path.size then return rule else
5   (← whnf goal).withApp fun h args =>
6     if h.isConstOf ``And then
7       match path[pos]! with
8       | Direction.l => do
9           let rule ← transport rule args[0]!
10                       path (pos + 1)
11          `(go_left_and $rule)
12      ···
```

In line 5 we put the `goal` in WHNF and consider its head constant; if it is conjunction (`` ``And ``), we analyze the current direction. In Line 9 we know that the direction is `l`, so we first transport the rule into the first operand of the conjunction (`args[0]!`); then, in line 11, we wrap that transported rule in `go_left_and` to ensure proper descent from the original `goal`. Line 11 is written using Lean4's monadic quotation syntax that produces a syntactic (pre-type-checking) application; this syntax is explained in [21]. The rest of the `transport` function merely covers all remaining combinations of connectives and directions for which there is a transport combinator (sec. 4.1).

The `within` ... `use` tactic can now straightforwardly be assembled as an elaborator:

```
elab "within " path:term,* " use " rule:term
  : tactic => do
  let path ← parsePath path
  let goalVar ← getMainGoal
  let goalForm ← Lean.MVarId.getType goalVar
  let rule ← transport rule goalForm path 0
  evalTactic (← `(tactic| refine' $rule _))
```

In the final line, the transported rule is sent directly to Lean4's own `refine'` tactic that generates a subgoal for every hole in the proof term; one hole is explicitly placed by us, while other holes may occur in the user-provided rule terms. Type-checking of the transported rule term is performed by the `refine'` tactic.

5.2 Generating Rules on Demand

Because the `rule` argument to `transport` is a piece of syntax, we can choose to do other things besides use it unmodified in the output from `transport`. The `init` and `congr` rules, as we saw in the previous section, have non-uniform types; in fact, these types can only be known once it is known which subformula they are being applied to. Fortunately, `transport` has this information in the `goal` argument. Therefore, we can modify line 4 of the `transport` listing as follows:

```
if pos ≥ path.size then processRule rule goal else ···
```

where we can now define:

```
def processRule (rule : Term) (goal : Expr)
    : TacticM Term :=
  match rule with
  | `(init) => doInit goal
  | `(congr) => doCongr goal
  /- and similar special cases -/
  | _ => return rule
```

These functions `doInit` and `doCongr` are defined in the same way; here for instance is `doCongr`:

```
1 def doCongr (goal : Expr) : TacticM Expr := do
2   (← whnf goal).withApp fun h args => do
3   if h.isConstOf ``Eq then do
4     (← whnf args[1]!).withApp fun f ss => do
5     (← whnf args[2]!).withApp fun g ts => do
6     if (← isDefEq f g) then doCongrEqs ss ts
7     else throwError s!"{f} ≠ {g}"
8   else throwError s!"{goal} is not an equation"
```

The `doCongr` function begins by checking that the given `goal` is an equation (lines 2–3), then checks that the heads of each of the operands of the equation are compatible (line 6). The computation `doCongrEqs ss ts` in line 7 generates a proof term whose type is of the form `s1 = t1` $\wedge \cdots \wedge$ `sn = tn` $\rightarrow$ `s = t` where `ss = #[s1, ..., sn]` and `ts = #[t1, ..., tn]`. The details of this function are omitted here.

6 Related and Future Work

6.1 Proof-By-Pointing

There have been several attempts to build user interfaces for formal reasoning systems that incorporate some aspects of direct manipulation. Perhaps the most well known is the *proof-by-pointing* effort that was initiated in the 1990's [5] and eventually became part of the *CtCoq* [4] and *Pcoq* [2] projects to improve the UI of Coq. Some ideas from this line of work have made it to other systems; for instance, the ProofGeneral mode in Emacs has support for proof-by-pointing for any backend prover that implements "*subterm markup*." While there is no reason in principle why one couldn't build a common interface to many provers in the style of Profint for proof-by-pointing, this appears not to have been attempted.

6.2 Non-Textual User Interfaces

A number of implementations exist for systems for graphically exploring formal proof objects or constructing such proofs in particular systems using some form of pointer-based interactivity. Some examples include *Building Blocks* [15], the *Incredible Proof Machine* (IPM) [6], *Click & coLLecT* [7], and *Sequoia* [20]. In all of these systems, the user is constrained to use a palette of inference steps according to the rules of some proof system such as natural deduction or the sequent calculus. These systems do not therefore have the same freedom of manipulation like in PBL, or even the relative freedom of clicking anywhere at any time in proof-by-pointing. Furthermore, we are not aware of any attempts for these systems to build proof objects for many different target formal reasoning systems.

6.3 Universality and Interchange

Universal proof languages such as *Dedukti* [12] are designed to simplify the use of shallow embeddings from a variety of other source languages. It would be fairly straightfowrard to write a Dedukti backend for Profint, since we can just reuse the compositional style in the highly explicit form that is currently used for Isabelle/HOL. An interesting alternative would be to target a knowledge representation framework such as MMT [19] which allows for the possibility of building general tools that can be used across a wide variety of logical domains. Finally, PBL and similar interaction techniques show a lot of promise for deductive formalisms that work on graph structures rather than formulas [1].

We chose to use the CoS formalism to translate PBL proofs because it is a deterministic core language with relatively easy to formalize rules. An alternative would have been to try to directly translate PBL proofs without going through CoS. Such a PBL "proof" would amount to a textual representation of a trace of a user's UI actions (the *UI-trace*), with commands such as `(link` π_1 π_2`)` or `(contract` π_1`)` where π_1 and π_2 are subformula paths. Meta-programming systems such as Lean 4 can possibly consume such UI traces and nevertheless use them as certificates. (Profint already contains an *elaborator* from this UI-trace language to CoS, which it uses for its own regression testing.) It is also worth trying to exchange such UI-trace objects between different implementations of PBL or other gestural interfaces.

7 Conclusion

We have described three styles of formalization of deep inference proofs produced in the *proof-by-linking* (PBL) technique, namely a *reflective*, a *combinatorial*, and a *meta-programming based* style, and used it to extract

formal proofs from the Profint implementation of the PBL method in a variety of target formal reasoning systems including: Coq, Isabelle/HOL, Lean 3, and Lean 4.[8]

As mentioned in [10], in order for Profint to be a serious candidate for an ITP interface, it needs to be extended to support (co-)induction and some kind of specialized automation for unification, numerical reasoning, etc. While the PBL style, in isolation, may be adapted for many such uses, the challenge will always be to retain its universality.

References

[1] M. Acclavio, R. Horne, and L. Straßburger. Logic beyond formulas: A proof system on graphs. In H. Hermanns, L. Zhang, N. Kobayashi, and D. Miller, editors, *35th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 38–52. ACM, 2020.

[2] A. Amerkad, Y. Bertot, L. Pottier, and L. Rideau. Mathematics and Proof Presentation in Pcoq. Technical Report RR-4313, INRIA, Nov. 2001.

[3] D. Baelde, K. Chaudhuri, A. Gacek, D. Miller, G. Nadathur, A. Tiu, and Y. Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2):1–89, 2014.

[4] Y. Bertot. The CtCoq system: Design and architecture. *Formal Aspects of Computing*, 11(3):225–243, Sept. 1999.

[5] Y. Bertot, G. Kahn, and L. Théry. Proof by pointing. In M. Hagiya and J. C. Mitchell, editors, *International Conference on the Theoretical Aspects of Computer Software TACS*, volume 789 of *Lecture Notes in Computer Science*, pages 141–160. Springer, 1994.

[6] J. Breitner. Visual theorem proving with the incredible proof machine. In J. C. Blanchette and S. Merz, editors, *Interactive Theorem Proving*, pages 123–139, Cham, 2016. Springer International Publishing.

[7] E. Callies and O. Laurent. Click and coLLecT An Interactive Linear Logic Prover. In *5th International Workshop on Trends in Linear Logic and Applications (TLLA 2021)*, Rome (virtual), Italy, June 2021.

[8] A. Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, pages 1–46, May 2011.

[9] K. Chaudhuri. Subformula linking as an interaction method. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Proceedings of the 4th Conference on Interactive Theorem Proving (ITP)*, volume 7998 of *LNCS*, pages 386–401. Springer, July 2013.

[10] K. Chaudhuri. Subformula linking for intuitionistic logic with application to type theory. In A. Platzer and G. Sutcliffe, editors, *28th International Conference on Automated Deduction*, volume 12699 of *Lecture Notes in Computer Science*, pages 200–216. Springer, 2021.

[11] K. Chaudhuri, N. Guenot, and L. Straßburger. The Focused Calculus of Structures. In *Computer Science Logic: 20th Annual Conference of the EACSL*, Leibniz International Proceedings in Informatics (LIPIcs), pages 159–173. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Sept. 2011.

[12] The Dedukti system. `https://deducteam.github.io/`, 2013.

[13] P. Donato, P. Strub, and B. Werner. A drag-and-drop proof tactic. In A. Popescu and S. Zdancewic, editors, *11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 197–209. ACM, 2022.

[14] G. Kahn, L. Thery, and Y. Bertot. Real theorem provers deserve real user-interfaces. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*. ACM Press, 1992. Software Engineering Notes, Vol 17, No 5.

[15] S. Lerner, S. R. Foster, and W. G. Griswold. Polymorphic blocks: Formalism-inspired UI for structured connectors. In *33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 3063–3072, New York, NY, USA, 2015. Association for Computing Machinery.

[16] D. Miller. ProofCert: Broad spectrum proof certificates. An ERC Advanced Grant funded for the five years 2012-2016, Feb. 2011.

[8] We are also implementing proof export to HOL4, HOL-Light, and PVS in the near future.

[17] P.-M. Pédrot. Ltac2: tactical warfare. In *The Fifth International Workshop on Coq for Programming Languages, CoqPL*, pages 13–19, 2019.

[18] A. M. Pitts. Nominal logic, A first order theory of names and binding. *Information and Computation*, 186(2):165–193, 2003.

[19] F. Rabe. The future of logic: Foundation-independence. *Logica Universalis*, 10(1):1–20, 2016.

[20] G. Reis, Z. Naeem, and M. Hashim. Sequoia: A playground for logicians - (system description). In N. Peltier and V. Sofronie-Stokkermans, editors, *10th International Joint Conference on Automated Reasoning (IJCAR)*, volume 12167 of *Lecture Notes in Computer Science*, pages 480–488. Springer, 2020.

[21] S. Ullrich and L. de Moura. Beyond notations: Hygienic macro expansion for theorem proving languages. *Log. Methods Comput. Sci.*, 18(2), 2022.

[22] C. Urban. Nominal reasoning techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008.

[23] B. Ziliani, D. Dreyer, N. R. Krishnaswami, A. Nanevski, and V. Vafeiadis. Mtac: a monad for typed tactic programming in coq. *ACM SIGPLAN Notices*, 48(9):87–100, 2013.

A Calculus of Structures

Here we present the rules of the Calculus of Structures (CoS) for the reasoning logic of this paper, intuitionistic first-order logic over simply typed λ-terms with λ-equivalence as its equational theory.

A.1 Syntax

We have the following syntactic categories.

$$\begin{aligned} b &:: \langle \text{Basic Type} \rangle \\ \alpha, \beta &::= b \mid \alpha \to \beta && \text{(Type)} \end{aligned}$$

$$\begin{aligned} \mathtt{k} &:: \langle \text{Term Constant} \rangle \\ x &:: \langle \text{Term Variable} \rangle \\ s, t &::= \mathtt{k} \mid x \mid \lambda x{:}\alpha.\, t \mid s\, t && \text{(Term)} \end{aligned}$$

$$\begin{aligned} \mathtt{a} &:: \langle \text{Predicate} \rangle \\ A, B, \ldots &::= \mathtt{a}\,\vec{t} \mid A \wedge B \mid \top \mid A \vee B \mid \bot \mid A \Rightarrow B \\ &\quad \mid \forall x{:}\alpha.\, A \mid \exists x{:}\alpha.\, A \mid s \doteq t && \text{(Formula)} \end{aligned}$$

$$\begin{aligned} \Sigma &::= \cdot \mid \Sigma, b{:}\mathsf{type} \mid \Sigma, \mathtt{k}{:}\alpha \\ &\quad \mid \Sigma, \mathtt{a}{:}\alpha_1 \to \cdots \to \alpha_n \to \mathtt{o} && \text{(Signature)} \end{aligned}$$

$$\begin{aligned} \mathcal{C}\{\,\} &::= \{\,\} \mid A * \mathcal{C}\{\,\} \mid \mathcal{C}\{\,\} * A \mid \mathtt{Q}x{:}\alpha.\, \mathcal{C}\{\,\} \\ &\quad \mid \mathcal{A}\{\,\} \Rightarrow B \mid A \Rightarrow \mathcal{C}\{\,\} && \text{(Pos. Context)} \\ \mathcal{A}\{\,\} &::= B * \mathcal{A}\{\,\} \mid \mathcal{A}\{\,\} * B \mid \mathtt{Q}x{:}\alpha.\, \mathcal{A}\{\,\} \\ &\quad \mid \mathcal{C}\{\,\} \Rightarrow A \mid B \Rightarrow \mathcal{A}\{\,\} && \text{(Neg. Context)} \\ * &\in \{\wedge, \vee\} \qquad \mathtt{Q} \in \{\forall, \exists\} \\ \mathcal{U}\{\,\} &::= \mathcal{C}\{\,\} \mid \mathcal{A}\{\,\} && \text{(Any Context)} \end{aligned}$$

A.2 Typing

Signatures

- A type is well-formed in a signature if every basic type in the type is declared in the signature.
- A term constant is well-typed in a signature if its type is well-formed in the signature
- A predicate constant is well-formed if all its argument types are well formed in the signature and the target type is $\mathtt{o}$.

- A signature is well-formed if every element of it is well-formed in it. We will assume an ambient well-formed signature Σ in the rest of this section.

Terms The type system for terms will be defined as contextual judgements of the form $\mathcal{U}\{t : \alpha\}$, with the following inference rules:

$$\frac{(\mathtt{k}{:}\alpha) \in \Sigma}{\mathcal{U}\{\mathtt{k} : \alpha\}} \qquad \frac{(\mathsf{Q} \in \{\forall, \exists\})}{\mathcal{U}_0\{\mathsf{Q}x{:}\alpha.\, \mathcal{U}_{\backslash x}\{x : \alpha\}\}}$$

$$\frac{\mathcal{U}\{\forall x{:}\alpha.\, (t : \alpha \to \beta)\}}{\mathcal{U}\{(\lambda x{:}\alpha.\, t) : \alpha \to \beta\}} \qquad \frac{\mathcal{U}\{s : \alpha \to \beta\} \quad \mathcal{U}\{t : \alpha\}}{\mathcal{U}\{s\, t : \beta\}}$$

The notation $\mathcal{U}_{\backslash x}\{\,\}$ stands for a context $\mathcal{U}\{\,\}$ where the variable x is not bound in any quantifier whose scope includes the hole.

Formulas Formula typing will be given in terms of a judgement of the form $\mathcal{U}\{A : \mathtt{o}\}$ with the following rules.

$$\frac{(\mathtt{a}{:}\alpha_1 \to \cdots \to \alpha_n \to \mathtt{o}) \in \Sigma \qquad \mathcal{U}\{t_1 : \alpha_1\} \quad \cdots \quad \mathcal{U}\{t_n : \alpha_n\}}{\mathcal{U}\{\mathtt{a}\, t_1 \cdots t_n : \mathtt{o}\}}$$

$$\frac{(* \in \{\wedge, \vee, \Rightarrow\}) \quad \mathcal{U}\{A : \mathtt{o}\} \quad \mathcal{U}\{B : \mathtt{o}\}}{\mathcal{U}\{A * B : \mathtt{o}\}} \qquad \frac{(\dagger \in \{\top, \bot\})}{\mathcal{U}\{\dagger : \mathtt{o}\}}$$

$$\frac{(\mathsf{Q} \in \{\forall, \exists\}) \quad \mathcal{U}\{\mathsf{Q}x{:}\alpha.\, (A : \mathtt{o})\}}{\mathcal{U}\{(\mathsf{Q}x{:}\alpha.\, A) : \mathtt{o}\}} \qquad \frac{\mathcal{U}\{s : \alpha\} \quad \mathcal{U}\{t : \alpha\}}{\mathcal{U}\{(s \doteq t) : \mathtt{o}\}}$$

A.3 Inference Rules

The CoS inference system consists of rules with exactly one formula premise and conclusion. Some rules have side conditions that may make use of one of the above typing judgements. All the rules will be written together with their rule name in monospaced font – these correspond exactly to the names used in the exported proofs.

Positive Linking Rules These rules involve replacements with an implication in a positive context in the conclusion. The rules are as follows.

Initial.

$$\frac{\mathcal{C}\{s_1 \doteq t_1 \wedge \cdots \wedge s_n \doteq t_n\}}{\mathcal{C}\{\mathtt{a}\, s_1 \cdots s_n \Rightarrow \mathtt{a}\, t_1 \cdots t_n\}}\ \mathtt{init}$$

with the interpretation that if $n = 0$ then the filler formula in the premise is $\top$.

Rewrites.

$$\frac{\mathcal{C}\{A\langle t\rangle\}}{\mathcal{C}\{s \doteq t \Rightarrow A\langle s\rangle\}}\ \mathtt{rew\_ltr} \qquad \frac{\mathcal{C}\{A\langle s\rangle\}}{\mathcal{C}\{s \doteq t \Rightarrow A\langle t\rangle\}}\ \mathtt{rew\_rtl}$$

Conjunction.

$$\frac{\mathcal{C}\{(A \Rightarrow B) \wedge C\}}{\mathcal{C}\{A \Rightarrow (B \wedge C)\}}\ \mathtt{goal\_ts\_and\_l} \qquad \frac{\mathcal{C}\{B \wedge (A \Rightarrow C)\}}{\mathcal{C}\{A \Rightarrow (B \wedge C)\}}\ \mathtt{goal\_ts\_and\_r}$$

$$\frac{\mathcal{C}\{A \Rightarrow C\}}{\mathcal{C}\{(A \wedge B) \Rightarrow C\}}\ \mathtt{goal\_and\_ts\_l} \qquad \frac{\mathcal{C}\{B \Rightarrow C\}}{\mathcal{C}\{(A \wedge B) \Rightarrow C\}}\ \mathtt{goal\_and\_ts\_r}$$

Disjunction.

$$\frac{\mathcal{C}\{A \Rightarrow B\}}{\mathcal{C}\{A \Rightarrow (B \vee C)\}}\ \mathtt{goal\_ts\_or\_l} \qquad \frac{\mathcal{C}\{A \Rightarrow C\}}{\mathcal{C}\{A \Rightarrow (B \vee C)\}}\ \mathtt{goal\_ts\_or\_r}$$

$$\frac{\mathcal{C}\{(A \Rightarrow C) \wedge (B \Rightarrow C)\}}{\mathcal{C}\{(A \vee B) \Rightarrow C\}}\ \mathtt{goal\_or\_ts}$$

Implication

$$\frac{\mathcal{C}\{(A \wedge B) \Rightarrow C\}}{\mathcal{C}\{A \Rightarrow (B \Rightarrow C)\}}\ \texttt{goal\_ts\_imp\_l} \qquad \frac{\mathcal{C}\{B \Rightarrow (A \Rightarrow C)\}}{\mathcal{C}\{A \Rightarrow (B \Rightarrow C)\}}\ \texttt{goal\_ts\_imp\_r}$$

$$\frac{\mathcal{C}\{A \wedge (B \Rightarrow C)\}}{\mathcal{C}\{(A \Rightarrow B) \Rightarrow C\}}\ \texttt{goal\_imp\_ts}$$

Quantifiers

$$\frac{\mathcal{C}\{\forall x{:}\alpha.\,(A \Rightarrow B)\}}{\mathcal{C}\{A \Rightarrow \forall x{:}\alpha.\,B\}}\ \texttt{goal\_ts\_all} \qquad \frac{\mathcal{C}\{\exists x{:}\alpha.\,(A \Rightarrow B)\}}{\mathcal{C}\{A \Rightarrow \exists x{:}\alpha.\,B\}}\ \texttt{goal\_ts\_all}$$

$$\frac{\mathcal{C}\{\exists x{:}\alpha.\,(A \Rightarrow B)\}}{\mathcal{C}\{(\forall x{:}\alpha.\,A) \Rightarrow B\}}\ \texttt{goal\_all\_ts} \qquad \frac{\mathcal{C}\{\forall x{:}\alpha.\,(A \Rightarrow B)\}}{\mathcal{C}\{(\exists x{:}\alpha.\,A) \Rightarrow B\}}\ \texttt{goal\_ex\_ts}$$

Negative Linking Rules These rules involved replacements with a conjunction in a negative context in the conclusion. There are a large number of symmetric variants just to keep the subformulas on the "same side" of the conjunction from the user's perspective.

$$\frac{\mathcal{A}\{A \wedge B\}}{\mathcal{A}\{A \wedge (B \wedge C)\}}\ \texttt{asms\_and\_l\_l} \qquad \frac{\mathcal{A}\{A \wedge C\}}{\mathcal{A}\{A \wedge (B \wedge C)\}}\ \texttt{asms\_and\_l\_r}$$

$$\frac{\mathcal{A}\{A \wedge C\}}{\mathcal{A}\{(A \wedge B) \wedge C\}}\ \texttt{asms\_and\_r\_l} \qquad \frac{\mathcal{A}\{B \wedge C\}}{\mathcal{A}\{(A \wedge B) \wedge C\}}\ \texttt{asms\_and\_r\_r}$$

$$\frac{\mathcal{A}\{(A \wedge B) \vee C\}}{\mathcal{A}\{A \wedge (B \vee C)\}}\ \texttt{asms\_or\_l\_l} \qquad \frac{\mathcal{A}\{B \vee (A \wedge C)\}}{\mathcal{A}\{A \wedge (B \vee C)\}}\ \texttt{asms\_or\_l\_r}$$

$$\frac{\mathcal{A}\{(A \wedge C) \vee B\}}{\mathcal{A}\{(A \vee B) \wedge C\}}\ \texttt{asms\_or\_r\_l} \qquad \frac{\mathcal{A}\{A \vee (B \wedge C)\}}{\mathcal{A}\{(A \vee B) \wedge C\}}\ \texttt{asms\_or\_r\_r}$$

$$\frac{\mathcal{A}\{(A \Rightarrow B) \Rightarrow C\}}{\mathcal{A}\{A \wedge (B \Rightarrow C)\}}\ \texttt{asms\_imp\_l\_l} \qquad \frac{\mathcal{A}\{B \Rightarrow (A \wedge C)\}}{\mathcal{A}\{A \wedge (B \Rightarrow C)\}}\ \texttt{asms\_imp\_l\_r}$$

$$\frac{\mathcal{A}\{(C \Rightarrow A) \Rightarrow B\}}{\mathcal{A}\{(A \Rightarrow B) \wedge C\}}\ \texttt{asms\_imp\_r\_l} \qquad \frac{\mathcal{A}\{A \Rightarrow (B \wedge C)\}}{\mathcal{A}\{(A \Rightarrow B) \wedge C\}}\ \texttt{asms\_imp\_r\_r}$$

$$\frac{\mathcal{A}\{\forall x{:}\alpha.\,(A \wedge B)\}}{\mathcal{A}\{A \wedge (\forall x{:}\alpha.\,B)\}}\ \texttt{asms\_all\_l} \qquad \frac{\mathcal{A}\{\forall x{:}\alpha.\,(A \wedge B)\}}{\mathcal{A}\{(\forall x{:}\alpha.\,A) \wedge B\}}\ \texttt{asms\_ex\_r}$$

$$\frac{\mathcal{A}\{\exists x{:}\alpha.\,(A \wedge B)\}}{\mathcal{A}\{A \wedge (\exists x{:}\alpha.\,B)\}}\ \texttt{asms\_ex\_l} \qquad \frac{\mathcal{A}\{\exists x{:}\alpha.\,(A \wedge B)\}}{\mathcal{A}\{(\exists x{:}\alpha.\,A) \wedge B\}}\ \texttt{asms\_ex\_r}$$

Structural Rules The contraction rule (`contract`) is used to implement the drag-and-drop in copy mode.

$$\frac{\mathcal{C}\{A \Rightarrow A \Rightarrow B\}}{\mathcal{C}\{A \Rightarrow B\}}\ \texttt{contract}$$

Weakening (`weaken`) is also present in the system, though there is no need to appeal to weakening explicitly.

$$\frac{\mathcal{C}\{B\}}{\mathcal{C}\{A \Rightarrow B\}}\ \texttt{weaken}$$

The final structural rule is congruence (`congr`) that is used as part of the simplification process (see below); however, unlike the rest of the simplification rules, congruence is not an equivalence except in the weak equational theory (λ-equivalence) we have assumed for terms. In a different equational theory, congruence would be under the explicit control of the user, just like contraction and weakening.

$$\frac{\mathcal{C}\{s_1 \doteq t_1 \wedge \cdots \wedge s_n \doteq t_n\}}{\mathcal{C}\{\mathtt{k}\, s_1 \cdots s_n \doteq \mathtt{k}\, t_1 \cdots t_n\}}\ \texttt{congr}$$

Simplification Rules These are rules that are applied after (i.e., above) the phase of linking rules, and is largely used to remove occurrences of $\top$ and $\bot$. All these rules are based on logical equivalences and hence the rules themselves are invertible; as a consequence there is no chance for these rules to break provability, and there is no harm in applying them eagerly.

$$\frac{\mathcal{U}\{A\}}{\mathcal{U}\{A \wedge \top\}}\ \texttt{simp\_and\_top} \qquad \frac{\mathcal{U}\{A\}}{\mathcal{U}\{\top \wedge A\}}\ \texttt{simp\_top\_and}$$

$$\frac{\mathcal{U}\{\top\}}{\mathcal{U}\{A \vee \top\}}\ \texttt{simp\_or\_top} \qquad \frac{\mathcal{U}\{\top\}}{\mathcal{U}\{\top \vee A\}}\ \texttt{simp\_top\_or}$$

$$\frac{\mathcal{U}\{\top\}}{\mathcal{U}\{A \Rightarrow \top\}}\ \texttt{simp\_imp\_top} \qquad \frac{\mathcal{U}\{A\}}{\mathcal{U}\{\top \Rightarrow A\}}\ \texttt{simp\_top\_imp}$$

$$\frac{\mathcal{U}\{\top\}}{\mathcal{U}\{\forall x{:}\alpha.\, \top\}}\ \texttt{simp\_all\_top}$$

$$\frac{\mathcal{U}\{\bot\}}{\mathcal{U}\{A \wedge \bot\}}\ \texttt{simp\_and\_bot} \qquad \frac{\mathcal{U}\{\bot\}}{\mathcal{U}\{\bot \wedge A\}}\ \texttt{simp\_bot\_and}$$

$$\frac{\mathcal{U}\{A\}}{\mathcal{U}\{A \vee \bot\}}\ \texttt{simp\_or\_bot} \qquad \frac{\mathcal{U}\{A\}}{\mathcal{U}\{\bot \vee A\}}\ \texttt{simp\_bot\_or}$$

$$\frac{\mathcal{U}\{\top\}}{\mathcal{U}\{\bot \Rightarrow A\}}\ \texttt{simp\_bot\_imp} \qquad \frac{\mathcal{U}\{\bot\}}{\mathcal{U}\{\exists x{:}\alpha.\, \bot\}}\ \texttt{simp\_ex\_bot}$$

B Guide to **Profint**

B.1 Installing and Running

In the accompanying materials of this paper, we provide a zip archive called `cpp2023p33.github.io-main.zip`. This contains a dump of the contents of the following URL at the time of submission:

```
https://cpp2023p33.github.io
```

You can use any modern browser to browse the above URL or, alternatively, unpack the archive and run your own server. A lightweight HTTP server for development purposes is distributed as part of Python 3's standard library; it can be run with the following command:

```
python3 -m http.server
```

This should allow HTTP requests at the port indicated, such as `http://127.0.0.1:8000/`.

B.2 Launching a Goal

The main window that appears after browsing to the above URL is the *Profint Launcher* window. In the top pane there are two editable fields. The top field is for the signature elements (basic types, predicates, term constants), while the second field is for stating the conjecture formula that will use the elements from the signature as needed.

Signatures ⟨sig⟩ have the following concrete grammar.

$$\begin{aligned}
\langle \mathsf{sig} \rangle &::= (\langle \mathsf{ident} \rangle \texttt{ ":" } \langle \mathsf{sig\text{-}ty} \rangle \texttt{ "."})^* \\
\langle \mathsf{sig\text{-}ty} \rangle &::= \texttt{"\type"} \mid (\langle \mathsf{ty} \rangle \texttt{ "->"})^* \texttt{ "\o"} \mid \langle \mathsf{ty} \rangle \\
\langle \mathsf{ty} \rangle &::= \langle \mathsf{ident} \rangle \mid \langle \mathsf{ty} \rangle \texttt{ "->" } \langle \mathsf{ty} \rangle \mid \texttt{"("} \langle \mathsf{ty} \rangle \texttt{ ")"}
\end{aligned}$$

Terms ⟨term⟩ have the following grammar.

$$\begin{aligned}
\langle \mathsf{term} \rangle &::= \langle \mathsf{ident} \rangle \mid \langle \mathsf{term} \rangle \ \langle \mathsf{term} \rangle \mid \texttt{"["} \langle \mathsf{binder} \rangle \texttt{"]"} \langle \mathsf{term} \rangle \\
\langle \mathsf{binder} \rangle &::= \langle \mathsf{ident} \rangle \ (\texttt{","} \ \langle \mathsf{ident} \rangle)^* \ (\texttt{":"} \ \langle \mathsf{ty} \rangle)^?
\end{aligned}$$

Formulas ⟨form⟩ have the following concrete grammar.

$$\begin{aligned}
\langle \mathsf{form} \rangle &::= \langle \mathsf{ident} \rangle \ \langle \mathsf{term} \rangle^* \mid \texttt{"("} \langle \mathsf{form} \rangle \texttt{")"} \\
&\quad \mid \langle \mathsf{form} \rangle \langle \mathsf{binop} \rangle \langle \mathsf{form} \rangle \\
&\quad \mid \langle \mathsf{unit} \rangle \mid \texttt{"\eq"} \ \langle \mathsf{term} \rangle \ \langle \mathsf{term} \rangle \\
&\quad \mid \langle \mathsf{quant} \rangle \texttt{"["} \langle \mathsf{binder} \rangle \texttt{"]"} \langle \mathsf{form} \rangle \\
\langle \mathsf{binop} \rangle &::= \texttt{"\and"} \mid \texttt{"\&"} \mid \texttt{"\or"} \mid \texttt{"|"} \mid \texttt{"\to"} \mid \texttt{"=>"} \\
\langle \mathsf{unit} \rangle &::= \texttt{"\top"} \mid \texttt{"\#t"} \mid \texttt{"\bot"} \mid \texttt{"\#f"} \\
\langle \mathsf{quant} \rangle &::= \texttt{"\A"} \mid \texttt{"\E"}
\end{aligned}$$

B.3 User Actions

The following mouse actions are supported.

- `Click` on a subformula
  - If there is already a source, then clicked subformula is destination of a *move* link.
  - If there is no source, then clicked subformula is source of a link.
- `Ctrl+Click` on a subformula (contract)
  - If there is already a source, then clicked subformula is a destination of a *copy* link.
  - If there is no source, and the clicked subformula is of the form $A \Rightarrow B$, then the antecedent is contracted, i.e., the subformula becomes $A \Rightarrow A \Rightarrow B$.
- `Alt+Click` on a subformula (weaken)
  - If the clicked subformula is a positively occurring and of the form $A \Rightarrow B$, then replace it with B.

- `Drag`-start on a subformula
  - If there is already a source, then this is an error
  - If there is no source, then mark current subformula as the source and initiate a drag.
- `Drag`-finish on a subformula
  - If there is already a source and it is being dragged, then the formula the drag finishes on becomes the destination of a *move* link.
- `Ctrl+Drag`-finish on a subformula
  - If there is already a source and it is being dragged, then the formula the drag finishes on becomes the destination of a *copy* link.

The following keyboard actions are supported.

- `Ctrl+Z` or `Ctrl+Down` keypress
  - Go backwards one step in history if possible.
- `Ctrl+Y` or `Ctrl+Up` keypress
  - Go forwards one step in history if possible.
- `Escape` keypress
  - If there is already a source, then it stops being the source
- `W` keypress
  - If the underlined subformula is a positively occurring $\exists$ or a negatively occurring $\forall$, and there are no other instantiation boxes, then create a instantiation box for this quantifier. Any $\langle$term$\rangle$ may be entered into the box using constants in the signature and variables in scope. Hit `Enter` to accept the instantiation term.

B.4 Downloading Proofs

The "Copy" button or the "Download" button can be used to obtain a formal proof object in a target proof system. The proof system can be selected with the dropdown.

"Copy" will just copy the proof text to the primary clipboard. It is intended to be used when you have Profint loaded in a user session in the target system and the proof can just be pasted in.

"Download" will produce a `.zip` file of one of the following kinds, based on the selected proof system, using the user supplied name as a prefix (default `proof`). Here are the corresponding build instructions.

- Coq (tracing), default `proof-coq.zip`:
  - Unzip to get the directory `proof-coq`
  - `cd proof-coq`
  - `make` (requires Coq 8.15.2+)
- Coq (reflective), default `proof-coq_reflective.zip`:
  - Unzip to get the directory `proof-coq_reflective`
  - `cd proof-coq_reflective`
  - `make` (requires Coq 8.15.2+)
- Isabelle/HOL (tracing), default `proof-isahol.zip`:
  - Unzip to get the directory `proof-isahol`
  - `cd proof-isahol`
  - Open `Proof.thy` using Isabelle2021-1
- Lean 3 (tracing), default `proof-lean3.zip`:
  - Unzip to get the directory `proof-lean3`

  - `cd proof-lean3`
  - `leanpkg build` (requires `leanpkg`)
- Lean 4, default `proof-lean4.zip`:
  - Unzip to get the directory `proof-lean4`
  - `cd proof-lean4`
  - `lake build` (require `elan` and `lake`)
- PDF (display only), default `proof-pdf.zip`:
  - Unzip to get the directory `proof-pdf`
  - `cd proof-pdf`
  - `latexmk -pdf proof` (requires `latexmk` and a recent `texlive`)