Fast cut-elimination using proof terms:
an empirical study

Gabriel Ebner

TU Wien, Austria*

gebner@gebner.org

Urban and Bierman introduced a calculus of proof terms for the sequent calculus LK with a strongly normalizing reduction relation in [33]. We extend this calculus to simply-typed higher-order logic with inferences for induction and equality, albeit without strong normalization. We implement this calculus in GAPT, our library for proof transformations. Evaluating the normalization on both artificial and real-world benchmarks, we show that this algorithm is typically several orders of magnitude faster than the existing Gentzen-like cut-reduction, and an order of magnitude faster than any other cut-elimination procedure implemented in GAPT.

1   Introduction

Cut-elimination is perhaps the most fundamental operation in proof theory, first introduced by Gentzen in [15]. Its importance is underlined by the immense variety of its applications, which are too numerous to list here. Though one application in particular motivates our interest in cut-elimination: cut-free proofs directly contain Herbrand disjunctions.

Herbrand's theorem [19, 7] captures the insight that the validity of a quantified formula is characterized by the existence of a tautological finite set of quantifier-free instances. In its simplest case, the validity of a purely existential formula is characterized by the existence of a tautological disjunction of instances, a Herbrand disjunction. Expansion proofs generalize this result to higher-order logic in the form of simple type theory [29].

A computational implementation of Herbrand's theorem as provided by cut-elimination lies at the foundation of many applications in computational proof theory: grammar-based compression of Herbrand disjunctions is used for the automatic introduction of non-analytic quantified lemmas [24, 22, 21, 11], and to automatically generate proofs with non-analytic induction [9, 10]. By comparing the Herbrand disjunctions of proofs, we obtain an important notion of equality that identifies proofs which use the same quantifier instances [3]. Proof deskolemization is naturally implemented as a linear-time algorithm on expansion proofs [4]. Herbrand disjunctions directly contain witnesses for the existential quantifiers and hence capture a certain computational interpretation of classical proofs. Furthermore, Luckhardt used Herbrand disjunctions to give a polynomial bound on the number of solutions in Roth's theorem [28].

Our GAPT system for proof transformations contains implementations of these Herbrand-based algorithms [13], as well as various proofs formalized in the sequent calculus LK and several cut-elimination procedures. However in practice we have proofs where none of these procedures are successful, due to multiple reasons: the performance may be insufficient, higher-order cuts cannot be treated, induction cannot be unfolded, or special-purpose inferences such as proof links are not supported.

The normalization procedure described in this paper has all of these features: it is fast, supports higher-order cuts, can unfold induction inferences, and does not fail in the presence of special-purpose

---

DRAFT   March 16, 2018

inference rules. This procedure is based on a term calculus for LK described by Urban and Bierman [33]. It is self-evident that proof normalization can be implemented more efficiently using the Curry-Howard correspondence to compute with proof terms instead of trees of sequents, as this significantly reduces the bureaucracy required during reduction. We also considered other calculi such as the $\lambda\mu$- [30] or the λ^{Sym}-calculus [6]. In the end we decided on the present calculus because of its close similarity to LK, as it allows us to straightforwardly integrate special-purpose inferences.

In Section 2 we present the syntax and typing rules for the calculus as implemented in GAPT. We then briefly describe the implementation of the normalization procedure in Section 3. Its performance is then empirically evaluated on both artificial and real-world proofs in Section 4. Finally, potential future work is discussed in Section 5.

2 Calculus

The proof system is modeled closely after the calculus described in the paper by Urban and Bierman [33]. Since the paper does not give a name to the introduced calculus, we call our variant LK_t as an abbreviation for "LK with terms". Proofs in LK_t operate on hypotheses (called names and co-names in [33]), which name formula occurrences in the current sequent. We found it useful to have a single type that combines both the names and co-names of [33] since it reduces code duplication. Each formula in a sequent is labelled by a hypothesis:

$$\frac{h_2\colon \varphi(t), h_1\colon \forall x\,\varphi(x), \Gamma \vdash \Delta}{h_1\colon \forall x\,\varphi(x), \Gamma \vdash \Delta}$$

Expressions in the object language are lambda expressions with simple types: an expression is either a variable, a constant, a lambda abstraction, or a function application. Connectives and quantifiers such as $\wedge^{o\to o\to o}$ and $\forall_\alpha^{(\alpha\to o)\to o}$ are represented as constants of the type indicated in the superscript. Formulas are expressions of type o, which is the type of Booleans. We identify $\alpha\beta$-equal expressions. A substitution $\sigma = [x_1\backslash s_1,\ldots,x_n\backslash s_n]$ is a type-preserving map from variables to expressions. Given an expression t, we write $t\sigma$ for the (capture-avoiding) application of the substitution σ to t.

The proof terms are almost untyped: in contrast to [33], we include the cut formula in the proof term for the cut inference to perform type-checking without higher-order unification. A typing judgement then tells us what sequent a proof proves. Figure 1 shows the syntax for the proof terms. Hypothesis arguments that are not bound are called main formulas: for example h_1 is a main formula of $NegL(h_1, h_2\colon \pi)$.

We use named variables as a binding strategy for the hypotheses in consistency with the implementation of the lambda expressions (as opposed to de Bruijn indices or a locally nameless representation). Hypotheses are stored as machine integers. A negative hypothesis refers to a formula in the antecedent, and a positive hypothesis refers to a formula in the succedent of the sequent. The notation *Hyp*: *Term* means that *Hyp* is bound in *Term*, c.f. the notation of abstract binding trees in [18]. This encoding of LK is also very similar to the encoding commonly used in logical frameworks (LF), see [31] for a description of such an approach.

Notably, there are no terms for weakening and contraction. These are implicit: we can use the same hypothesis zero or multiple times. The proof terms only contain new information that is not contained in the end-sequent; only cut formulas, weak quantifier instance terms, and eigenvariables are stored. We do not repeat the formulas or atoms of the end-sequent.

Let us now define the typing judgment. A local context is a finite map from hypotheses to formulas. We write $h_1\colon \varphi \vdash h_2\colon \psi$ as a suggestive notation for the map $\{h_1 \mapsto \varphi, h_2 \mapsto \psi\}$ where h_1 is negative and h_2 is positive. Outer occurrences overwrite inner ones, that is $\vdash h\colon \varphi, h\colon \psi$ means $\{h \mapsto \psi\}$.

$$Hyp ::= -\mathbb{N}^+ \mid +\mathbb{N}^+$$
$$Term ::= \text{Ax}(Hyp, Hyp) \mid \text{TopR}(Hyp)$$
$$\mid \text{Cut}(Formula, Hyp\colon Term, Hyp\colon Term)$$
$$\mid \text{NegL}(Hyp, Hyp\colon Term) \mid \text{NegR}(Hyp, Hyp\colon Term)$$
$$\mid \text{AndL}(Hyp, Hyp\colon Hyp\colon Term) \mid \text{AndR}(Hyp, Hyp\colon Term, Hyp\colon Term)$$
$$\mid \text{AllL}(Hyp, Expr, Hyp\colon Term) \mid \text{AllR}(Hyp, Var\colon Hyp\colon Term)$$
$$\mid \text{Rfl}(Hyp) \mid \text{Eql}(Hyp, Hyp, Bool, Expr, Hyp\colon Term)$$
$$\mid \text{Ind}(Hyp, Expr, Expr, Hyp\colon Term, Var\colon Hyp\colon Hyp\colon Term)$$

Figure 1: Syntax of LK_t

Given an (expression) substitution σ, we can apply it to a proof term π in the natural way to obtain $\pi\sigma$. The judgment $\pi ::_\sigma S$ means that $\pi\sigma$ is a valid proof in the local context S, that is, π proves that the sequent corresponding to S is valid. We may omit σ if it is the identity substitution, in this special case $\pi :: \Gamma \vdash \Delta$ corresponds to the notation $\Gamma \triangleright \pi \triangleright \Delta$ used in [33].

The reason for parameterizing the typing judgement by a substitution is twofold: due to our use of named variables, we may need to rename bound eigenvariables (in AllR and Ind) when traversing a term. However, we do not want to apply a substitution to the proof term to ensure that the eigenvariable is fresh. This would be both costly and also introduces an unnecessary dependency on the local context in operations that would otherwise not require any typing information.

The proof terms and corresponding typing rules are chosen in such a way that they correspond as much as possible to the already implemented sequent calculus LK, see [12, Appendix B.1] for detailed description of that calculus. The implementation also contains further inferences for special applications, such as proof links for schematic proofs [8], definition rules [2], and Skolem inferences to represent Skolemized proofs in higher-order logic [23]. The implemented inference rule for induction is also more general than the one shown here: it supports structural recursions over other types than the natural numbers.

Equational reasoning is implemented using the Rfl inference for reflexivity, and an Eql inference to rewrite in arbitrary contexts and on both sides of the sequent. The third argument indicates whether we rewrite from left-to-right or right-to-left. Syntactically, we support equations between terms of arbitrary type, however cut-elimination can fail with equations between functions or Booleans as quantified cuts can remain.

In our version of higher-order logic, the connectives $\vee, \rightarrow, \perp$, and $\exists$ are also primitive. By a heavy abuse of notation, we simply reuse the proof terms for $\wedge, \top$, and $\forall$. This representation causes no confusion, since the intended connective is always clear from the polarities of the hypotheses, and many operations are defined identically for the different connectives. The corresponding typing rules are derived in the natural way, we only show the case for AndL and an implication on the right side as an example:

$$\frac{\pi ::_\sigma h_2\colon \varphi, \Gamma \vdash \Delta, h_1\colon \varphi \rightarrow \psi, h_3\colon \psi}{\text{AndL}(h_1, h_2\colon h_3\colon \pi) ::_\sigma \Gamma \vdash \Delta, h_1\colon \varphi \rightarrow \psi}$$

$$\frac{}{\mathrm{Ax}(h_1,h_2) \; ::_\sigma \; h_1\colon \varphi, \Gamma \vdash \Delta, h_2\colon \varphi} \qquad\qquad \frac{}{\mathrm{TopR}(h_1) \; ::_\sigma \; \Gamma \vdash \Delta, h_1\colon \top}$$

$$\frac{\pi_1 \; ::_\sigma \; \Gamma \vdash \Delta, h_1\colon \varphi\sigma \qquad \pi_2 \; ::_\sigma \; h_2\colon \varphi\sigma, \Gamma \vdash \Delta}{\mathrm{Cut}(\varphi, h_1\colon \pi_1, h_2\colon \pi_2) \; ::_\sigma \; \Gamma \vdash \Delta}$$

$$\frac{\pi \; ::_\sigma \; h_1\colon \neg\varphi, \Gamma \vdash \Delta, h_2\colon \varphi}{\mathrm{NegL}(h_1, h_2\colon \pi) \; ::_\sigma \; h_1\colon \neg\varphi, \Gamma \vdash \Delta} \qquad \frac{\pi \; ::_\sigma \; h_2\colon \varphi, \Gamma \vdash \Delta, h_1\colon \neg\varphi}{\mathrm{NegR}(h_1, h_2\colon \pi) \; ::_\sigma \; \Gamma \vdash \Delta, h_1\colon \neg\varphi}$$

$$\frac{\pi \; ::_\sigma \; h_3\colon \psi, h_2\colon \varphi, h_1\colon \varphi \wedge \psi, \Gamma \vdash \Delta}{\mathrm{AndL}(h_1, h_2\colon h_3\colon \pi) \; ::_\sigma \; h_1\colon \varphi \wedge \psi, \Gamma \vdash \Delta}$$

$$\frac{\pi_1 \; ::_\sigma \; \Gamma \vdash \Delta, h_1\colon \varphi \wedge \psi, h_2\colon \varphi \qquad \pi_2 \; ::_\sigma \; \Gamma \vdash \Delta, h_1\colon \varphi \wedge \psi, h_3\colon \psi}{\mathrm{AndR}(h_1, h_2\colon \pi_1, h_3\colon \pi_2) \; ::_\sigma \; \Gamma \vdash \Delta, h_1\colon \varphi \wedge \psi}$$

$$\frac{\pi \; ::_\sigma \; h_2\colon \varphi(t\sigma), h_1\colon \forall x\, \varphi(x), \Gamma \vdash \Delta}{\mathrm{AllL}(h_1, t, h_2\colon \pi) \; ::_\sigma \; h_1\colon \forall x\, \varphi(x), \Gamma \vdash \Delta}$$

$$\frac{\pi \; ::_{[x\backslash y]\sigma} \; \Gamma \vdash \Delta, h_1\colon \forall x\, \varphi(x), h_2\colon \varphi(y)}{\mathrm{AllR}(h_1, x, h_2\colon \pi) \; ::_\sigma \; \Gamma \vdash \Delta, h_1\colon \forall x\, \varphi(x)} \; (y \text{ fresh})$$

$$\frac{}{\mathrm{Rfl}(h) \; ::_\sigma \; \Gamma \vdash \Delta, h\colon t = t} \qquad \frac{\pi \; ::_\sigma \; h_1\colon t = s, \Gamma \vdash \Delta, h_2\colon \varphi\sigma(t), h_3\colon \varphi\sigma(s)}{\mathrm{Eql}(h_1, h_2, \mathtt{true}, \varphi, h_3\colon \pi) \; ::_\sigma \; h_1\colon t = s, \Gamma \vdash \Delta, h_2\colon \varphi\sigma(t)}$$

$$\frac{\pi_1 \; ::_\sigma \; \Gamma \vdash \Delta, h_1\colon \varphi(t)\sigma, h_2\colon \varphi\sigma(0) \qquad \pi_2 \; ::_{[x\backslash y]\sigma} \; \Gamma, h_3\colon \varphi\sigma(y) \vdash \Delta, h_1\colon \varphi(t)\sigma, h_4\colon \varphi\sigma(s(y))}{\mathrm{Ind}(h_1, \varphi, t, h_2\colon \pi_1, x\colon h_3\colon h_4\colon \pi_2) \; ::_\sigma \; \Gamma \vdash \Delta, h_1\colon \varphi(t)\sigma} \; (y \text{ fresh})$$

Figure 2: Typing rules for LK$_t$

3   Cut-normalization

Normalization is performed in a big-step evaluation approach using 3 mutually recursive functions $\mathcal{N}$, $\mathcal{E}$, and $\mathcal{S}$[1]. All of these functions return fully normalized proof terms. We do not create temporary Cut terms, all produced Cut terms are irreducible (for example because they are "stuck" on Eql or Ind). Figure 3 shows the definition of the functions $\mathcal{N}, \mathcal{E},$ and $\mathcal{S}$. Note that since contraction is implicit, the cut rule behaves more like Gentzen's mix rule [15].

- The function $\mathcal{N}$ takes a proof term π as input and returns a normal form $\mathcal{N}(\pi)$.

- If π_1 and π_2 are already in normal form, then $\mathcal{E}(\varphi, h_1\colon \pi_1, h_2\colon \pi_2)$ computes a normal form of $\mathrm{Cut}(\varphi, h_1\colon \pi_1, h_2\colon \pi_2)$.

- Let π_1 and π_2 be again in normal form, then $\mathcal{S}(\pi_1, \varphi, h_1 := h_2\colon \pi_2)$ performs a proof substitution, which corresponds to the rank-reduction step of cut-elimination in LK. The function $\mathcal{S}$ takes one side of the cut and directly moves it to all inferences in the other side where the cut formula occurs as the main formula. This operation is symmetric in the side of the cut, and only needs to be

---
[1]In the implementation these are called `normalize`, `evalCut`, and `ProofSubst`, resp.

implemented once. Hence we have two cases for the type preservation of $\mathcal{S}$:

$$\frac{\pi_1 ::_\sigma \Gamma \vdash \Delta, h_1 \colon \varphi\sigma \qquad \pi_2 ::_\sigma h_2 \colon \varphi\sigma, \Gamma \vdash \Delta}{\mathcal{S}(\pi_1, \varphi, h_1 := h_2 \colon \pi_2) ::_\sigma \Gamma \vdash \Delta} \qquad \frac{\pi_1 ::_\sigma h_1 \colon \varphi\sigma, \Gamma \vdash \Delta \qquad \pi_2 ::_\sigma \Gamma \vdash \Delta, h_2 \colon \varphi\sigma}{\mathcal{S}(\pi_1, \varphi, h_1 := h_2 \colon \pi_2) ::_\sigma \Gamma \vdash \Delta}$$

We perform a few noteworthy optimizations:

- Every term stores the set of its free hypotheses and free (expression) variables. These are fields in the Scala classes implementing the proof terms. We can hence efficiently (in logarithmic time) check whether a given hypothesis or variable is free in a proof term.

- Due to this extra data, we can effectively skip many calls of the normalization procedure. We do not need to substitute or evaluate cuts if the hypothesis for the cut formula is not free in the subterm, in this case we can immediately return the subterm.

- When producing the resulting proof terms, we check whether we can skip any inferences. For example, instead of NegL($h_1, h_2 \colon \pi$) we can directly return π if h_2 is not free in π. In Fig. 3 we denote these "skipping" constructors with the $\cdot^?$ superscript. This optimization is extremely important from a practical point of view, since it effectively prevents a common blow-up in proof size.

The cut-normalization in [33] is presented as a single-step reduction relation. The strong normalization of that relation depends on the fact that all cuts can be eliminated in their calculus. In LK_t however, cuts can be irreducible—for example because they are stuck on an induction or on Eql. This has the unfortunate consequence that the natural single-step reduction relation for LK_t is not strongly normalizing. Since multiple cuts can be stuck on the same inference we have the traditional counterexample of two commuting cuts, where for example $\pi_3 = \mathrm{Eql}(h_2, h_4, \texttt{true}, \dots)$:

$$\mathrm{Cut}(\varphi, h_1 \colon \pi_1, h_2 \colon \mathrm{Cut}(\psi, h_3 \colon \pi_2, h_4 \colon \pi_3)) \mapsto \mathrm{Cut}(\psi, h_3 \colon \pi_2, h_4 \colon \mathrm{Cut}(\varphi, h_1 \colon \pi_1, h_4 \colon \pi_3)) \mapsto \dots$$

3.1 Induction unfolding

Elimination of induction inferences is handled in a similar way to Gentzen's proof of the consistency of Peano Arithmetic [16]. Induction inferences whose terms are constructor applications are unfolded:

$$\mathrm{Ind}(h_1, \varphi, 0, h_2 \colon \pi_1, x \colon h_3 \colon h_4 \colon \pi_2) \mapsto \pi_1[h_2 \backslash h_1]$$

$$\mathrm{Ind}(h_1, \varphi, s(t), h_2 \colon \pi_1, x \colon h_3 \colon h_4 \colon \pi_2) \mapsto \mathrm{Cut}(\varphi(t), h_1 \colon \mathrm{Ind}(h_1, \varphi, t, h_2 \colon \pi_1, x \colon h_3 \colon h_4 \colon \pi_2), h_3 \colon \pi_2[x \backslash t][h_4 \backslash h_1])$$

The full induction-elimination procedure then alternates between cut-normalization and full induction unfolding until we can no longer unfold any induction inferences. We also rewrite the term t in the induction inference using a list of universally quantified equations representing the primitive recursive definitions to bring the term into constructor form. The generated proof $\pi_s :: \Gamma, h_5 \colon \varphi(t') \vdash h_4 \colon \varphi(t)$ is then added via a cut, where t' is the simplified term which is now in constructor form:

$$\mathrm{Ind}(h_1, \varphi, t, \dots) \mapsto \mathrm{Cut}(\varphi(t'), h_4 \colon \mathrm{Ind}(h_1, \varphi, t', \dots), h_5 \colon \pi_s)$$

3.2 Equational reduction

As noted in Section 3, cuts on equational inferences are stuck. Consider for example the following term, which cannot be reduced further:

$$\mathrm{Cut}(\forall x\, P(x, 0), h_1 \colon \mathrm{Eql}(h_1, h_2, \texttt{true}, \lambda y\, \forall x\, P(x, y), h_3 \colon \pi_1), h_4 \colon \pi_2)$$

$$\mathcal{N}(\mathrm{Cut}(\varphi,h_1\colon\varphi,h_2\colon\psi)) = \mathcal{E}(\varphi,h_1\colon\mathcal{N}(\varphi),h_2\colon\mathcal{N}(\psi))$$
$$\mathcal{N}(\mathrm{Ax}(h_1,h_2)) = \mathrm{Ax}(h_1,h_2)$$
$$\mathcal{N}(\mathrm{NegL}(h_1,h_2\colon\pi)) = \mathrm{NegL}^?(h_1,h_2\colon\mathcal{N}(\pi))$$

$$\vdots \quad \text{(other cases recurse analogously)}$$

$$(\text{if } h_1 \text{ is not free in } \pi_1\colon) \quad \mathcal{E}(\varphi,h_1\colon\pi_1,h_2\colon\pi_2) = \pi_1$$
$$(\text{if } h_2 \text{ is not free in } \pi_2\colon) \quad \mathcal{E}(\varphi,h_1\colon\pi_1,h_2\colon\pi_2) = \pi_2$$
$$\mathcal{E}(\varphi,h_1\colon\mathrm{Ax}(h_2,h_1),h_3\colon\pi) = \pi[h_3\backslash h_2]$$
$$\mathcal{E}(\varphi,h_1\colon\pi,h_2\colon\mathrm{Ax}(h_2,h_3)) = \pi[h_1\backslash h_3]$$
$$\mathcal{E}(\neg\varphi,h_1\colon\mathrm{NegR}(h_1,h_2\colon\pi_1),h_3\colon\mathrm{NegL}(h_3,h_4\colon\pi_2)) = \mathcal{E}(\varphi,h_4\colon\pi_2',h_2\colon\pi_1')$$
$$\mathcal{E}(\varphi\wedge\psi,h_1\colon\mathrm{AndR}(h_1,h_2\colon\pi_1,h_3\colon\pi_2),h_4\colon\mathrm{AndL}(h_4,h_5\colon h_6\colon\pi_3)) = \mathcal{E}(\psi,h_3\colon\pi_2',h_6\colon\mathcal{S}(\pi_3',\varphi,h_5:=h_2\colon\pi_1'))$$
$$\mathcal{E}(\varphi\circledast\psi,h_1\colon\mathrm{AndL}(h_1,h_2\colon h_3\colon\pi_1),h_4\colon\mathrm{AndR}(h_4,h_5\colon\pi_2,h_6\colon\pi_3)) = \mathcal{E}(\psi,h_2\colon\mathcal{S}(\pi_1',\varphi,h_3:=h_5\colon\pi_3'),h_5\colon\pi_2')$$
$$\mathcal{E}(\forall x\,\varphi(x),h_1\colon\mathrm{AllR}(h_1,y,h_2\colon\pi_1),h_3\colon\mathrm{AllL}(h_3,t,h_4\colon\pi_2)) = \mathcal{E}(\varphi(t),h_2\colon\pi_1[y\backslash t]',h_3\colon\pi_2')$$
$$\mathcal{E}(\exists x\,\varphi(x),h_1\colon\mathrm{AllL}(h_1,t,h_2\colon\pi_1),h_3\colon\mathrm{AllR}(h_3,y,h_4\colon\pi_2)) = \mathcal{E}(\varphi(t),h_2\colon\pi_1',h_4\colon\pi_2[y\backslash t]')$$
$$(\text{if } h_1 \text{ is not a main formula of } \pi_1\colon) \quad \mathcal{E}(\varphi,h_1\colon\pi_1,h_2\colon\pi_2) = \mathcal{S}(\pi_1,\varphi,h_1:=h_2\colon\pi_2)$$
$$(\text{if } h_2 \text{ is not a main formula of } \pi_2\colon) \quad \mathcal{E}(\varphi,h_1\colon\pi_1,h_2\colon\pi_2) = \mathcal{S}(\pi_2,\varphi,h_2:=h_1\colon\pi_1)$$
$$(\text{otherwise}\colon) \quad \mathcal{E}(\varphi,h_1\colon\pi_1,h_2\colon\pi_2) = \mathrm{Cut}^?(\varphi,h_1\colon\pi_1,h_2\colon\pi_2)$$

$$\mathcal{S}(\mathrm{Ax}(h_1,h_2),\varphi,h_1:=h_3\colon\pi) = \pi[h_3\backslash h_2]$$
$$\mathcal{S}(\mathrm{Ax}(h_1,h_2),\varphi,h_2:=h_3\colon\pi) = \pi[h_3\backslash h_1]$$
$$\mathcal{S}(\mathrm{NegL}(h_1,h_2\colon\pi_1),\varphi,h_3:=h_4\colon\pi_2) = \mathrm{NegL}^?(h_1,h_2\colon\mathcal{S}(\pi_1,\varphi,h_3:=h_4\colon\pi_2)) \quad (\text{if } h_1 \neq h_3)$$

$$\vdots \quad \text{(other cases recurse analogously)}$$

$$\mathcal{S}(\pi_1,\varphi,h_1:=h_2\colon\pi_2) = \mathcal{E}(\varphi,h_1\colon\pi_1,h_2\colon\pi_2) \quad (\text{otherwise, if } h_1 < 0)$$
$$\mathcal{S}(\pi_1,\varphi,h_1:=h_2\colon\pi_2) = \mathcal{E}(\varphi,h_2\colon\pi_2,h_1\colon\pi_1) \quad (\text{otherwise, if } h_1 > 0)$$

$$\mathrm{NegL}^?(h_1,h_2\colon\pi) = \pi \quad (\text{if } h_2 \text{ not free in } \pi)$$
$$\mathrm{NegL}^?(h_1,h_2\colon\pi) = \mathrm{NegL}(h_1,h_2\colon\pi) \quad (\text{otherwise})$$

$$\vdots \quad \text{(other cases analogously)}$$

Figure 3: Evaluator for LK$_t$. For reasons of space, we use the abbreviation π_2' to abbreviate the proof substitution which substitutes the opposite part of the cut for the main formula: for example, π_2' abbreviates $\mathcal{S}(\pi_2,\varphi,h_2:=h_1\colon\pi_1)$ in the case of $\mathcal{E}(\varphi,h_1\colon\pi_1,h_2\colon\mathrm{NegL}(h_2,h_3\colon\pi_2))$.

This is clearly a problem since we cannot obtain Herbrand disjunctions from proofs with such quantified cuts. On the other hand, cuts on atoms would pose no problem since we can still obtain Herbrand disjunctions by examining the weak quantifier inferences. We hence reduce quantified equational inferences to atomic equational inferences—then only atomic cuts can be stuck. The translation replaces every Eql inferences on a non-atomic formula by a cut with a proof that performs the rewriting step using only atomic Eql inferences. Note that this translation can fail if we have equations between predicates.

4 Empirical evaluation

4.1 Artificial examples

The calculus and normalization procedure presented in this paper has been implemented in the open source GAPT system[2] for proof transformations [13], version 2.10. We now compare the performance of several cut-normalization procedures implemented in GAPT on benchmarks used in [27].

- *LK:* Gentzen-style reductive cut-elimination in LK. The proofs in LK are tree-like data structure where every node has a (formula) sequent. The output is again a proof in LK, atomic cuts can appear directly below equational inferences.

- *CERES (LK):* Cut-elimination by resolution [5] reduces the problem of cut-elimination in LK to finding a resolution refutation of a first-order clause set. The output is a proof in LK with at most atomic cuts.

- *CERES (expansion):* a variant of CERES that takes proofs with cuts in LK, and directly produces expansion proofs [27]. This uses the same first-order clause sets as *CERES (LK)*.

- *semantic:* by "semantic cut-elimination", we refer to the procedure that throws away the input proofs, and generates a cut-free proof from scratch. GAPT contains interfaces to several resolution provers, including the built-in Escargot prover. Here we used Escargot to obtain a cut-free expansion proof of the end-sequent of the input proof.

- *expansion proof:* the expansion proofs implemented in GAPT support cuts—such cuts corresponds to cuts in LK and are simply expansions of the formula $\forall X\,(X \to X)$. First-order cuts in expansion proofs can eliminated using a procedure described in [25], which operates just on the quantifier instances of the proof, and is similar to the proofs of the epsilon theorems [26]. Both the input and output formats are expansion proofs, the resulting expansion proof is cut-free.

- *LKt:* the normalization procedure shown in Section 3.

- *LKt (until atomic):* same as *LKt*, but we do not reduce atomic cuts. The resulting proof may still contain cuts on atoms, but this is sufficient for the extraction of Herbrand disjunctions. We can directly extract Herbrand disjunctions from proofs as long as all cut formulas are propositional.

- *LKt (until quant.-free):* same as *LKt*, but we do not reduce quantifier-free cuts.

The graphs in Fig. 4 show the runtime for each of these procedures on several artificial example proofs. The runtime is measured in seconds of wall clock time; we used a logarithmic scale for the time since the performance of the procedures differs by several orders of magnitude. In one case, *LKt (until quant.-free)* is 1000000 times faster than *LK*.

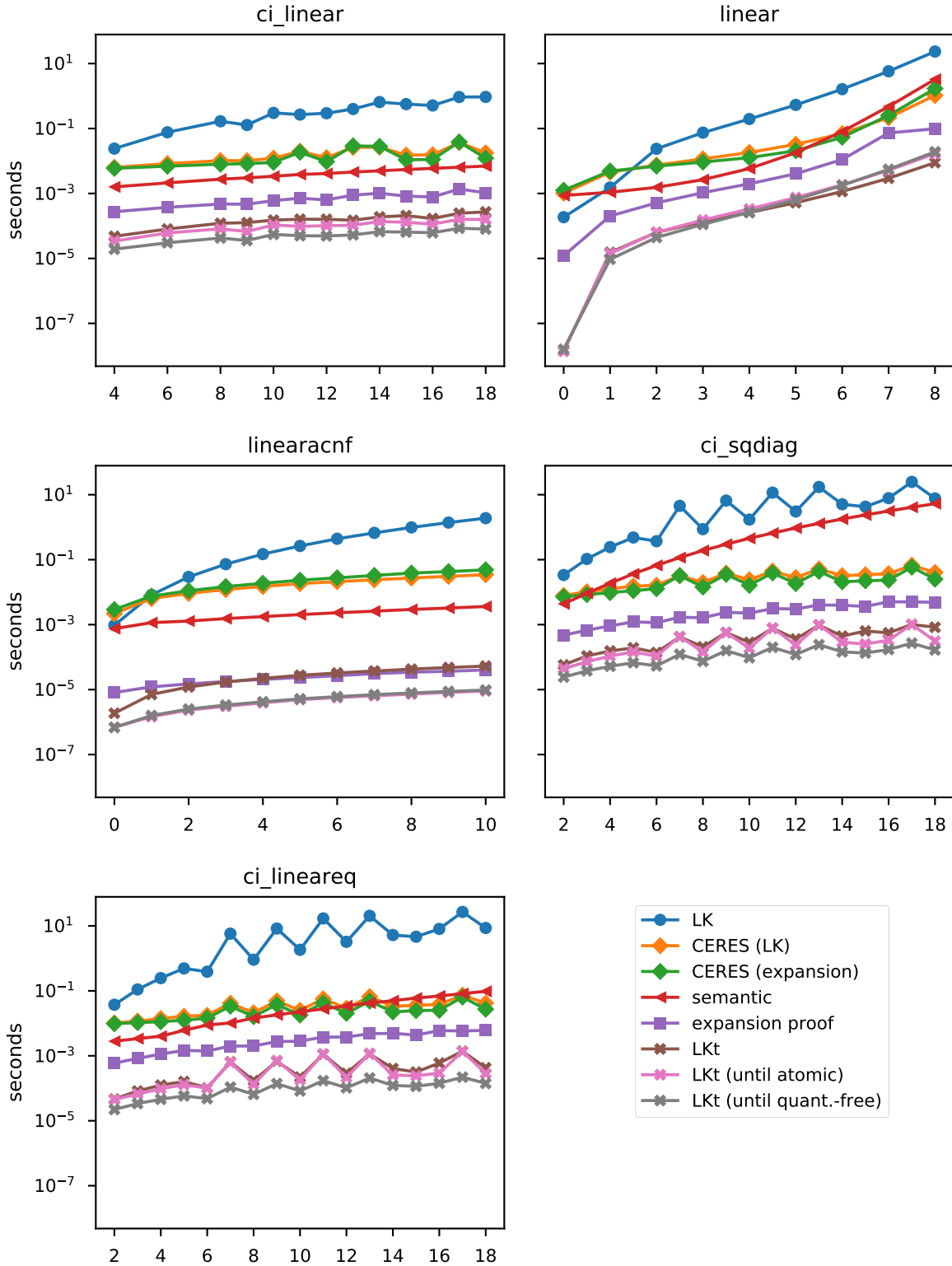[2]available at `https://logic.at/gapt`

Figure 4: Runtime of cut-elimination procedures implemented in GAPT on several artificial examples.

Linear example after cut-introduction (`ci_linear`)   The name "linear example" refers to the sequence of (proofs of) the sequent $P(0), \forall x\,(P(x) \to P(s(x))) \vdash P(s^n(0))$ for $n \geq 0$. In GAPT, the function `LinearExampleProof(_)` produces the natural cut-free proofs of these sequents. We then use an automated method that introduces universally quantified cuts [11] to obtain a proof with cut. These proofs with universally quantified cuts are produced with `CutIntroduction(LinearExampleProof(n)).get`.

In this example, all of the $\mathrm{LK_t}$ normalization procedures are faster than the CERES variants by a factor of about 100x. Even semantic cut-elimination is faster. $\mathrm{LK_t}$ normalization is also faster than expansion proof cut-elimination by a factor of about 10x. We also see that not eliminating atomic cuts is a bit faster than full cut-elimination, and not eliminating quantifier-free cuts is even faster.

Linear example proof with manual cuts (`linear`)   Cut-introduction often produces unnecessarily complicated lemmas, resulting in irregularity when used in proof sequences. It is also limited to small proofs. To produce a more regular sequence and obtain larger proofs, we manually formalized natural proofs of the linear example for $n = 2^m$ using $m - 1$ cuts with the cut formulas $\forall x\,(P(x) \to P(s^{2^k}(x)))$ for $1 \leq k < m$. These proofs can be obtained with `LinearCutExampleProof(m)`.

The results are similar to the proofs obtained with cut-introduction, although we observe new phenomena at both ends of the sequence: for $n = 0$, the proofs consist of a single axiom. Here, the $\mathrm{LK_t}$-based procedures produce a cut-free proof in about 15 nanoseconds. On the other end, at $n \geq 6$, we finally see CERES becoming slightly faster than semantic cut-elimination.

Linear example proof with atomic cuts (`linearacnf`)   To complete the discussion of the linear example, we also consider a proof sequence in atomic cut-normal form (ACNF). In these proofs, the quantifier and propositional inferences are on the top of the proof, and the bottom part consists only of atomic cuts—very much like a ground resolution refutation. Interestingly, atomic cut-elimination is surprisingly cheap in this example: the $\mathrm{LK_t}$-based normalization only takes 10 microseconds. On the other hand, the CERES-based methods require as much time as they do for the proofs with universally quantified cuts: they refute a clause set whose size is linear in n.

Square diagonal proof after cut-introduction (`ci_sqdiag`)   These proofs are generated in GAPT using `CutIntroduction(SquareDiagonalExampleProof(n))`. $\mathrm{LK_t}$ normalization until quantifier-free cuts is an order of magnitude faster than expansion proof cut-elimination, and two orders of magnitude faster than CERES.

Linear equality example proof after cut-introduction (`ci_lineareq`)   These proofs are generated using `CutIntroduction(LinearEqExampleProof(n))`. Note that we replaced the equality predicate by a binary E relation to prevent accidental introduction of equational inferences. Again, $\mathrm{LK_t}$ normalization until propositional cuts is 10x faster than expansion proof cut elimination, which is 10x faster than CERES.

The astute reader will have noticed the spikes in the runtime of the reductive cut-elimination procedures at $n \in \{7, 9, 11, 13, 17\}$. These spikes are due to convoluted cut formulas produced by cut-introduction. For example at $n = 17$, the cut formula is $\forall x\,((E(f(x),a) \to E(f^3(x),a)) \wedge (E(x,a) \to E(f(x),a)))$ and we use it to prove $E(a,a) \to E(f^{17}(a),a)$—this proof is almost as complicated on the propositional level as the cut-free proof, even though it has a lower quantifier complexity.
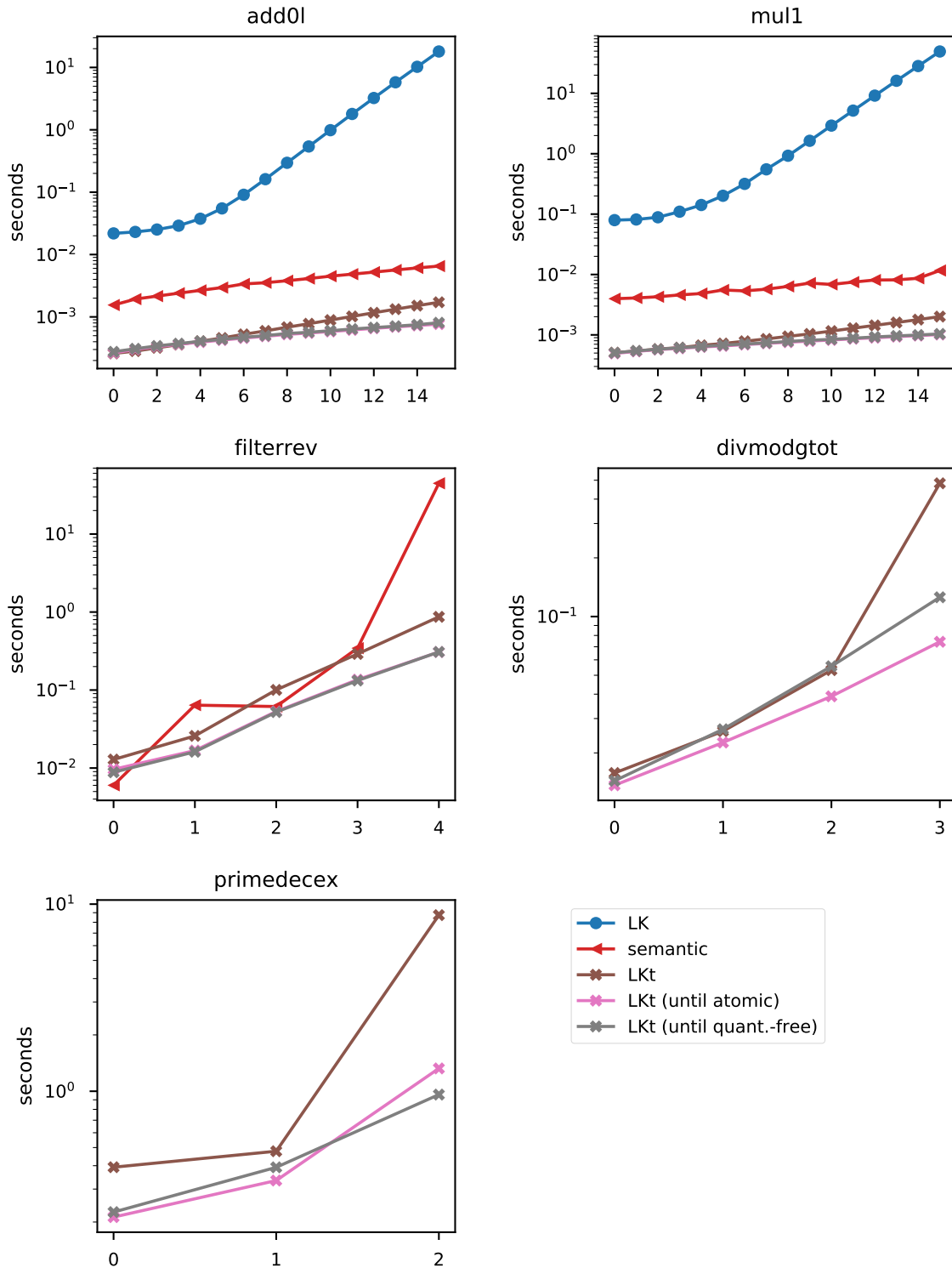
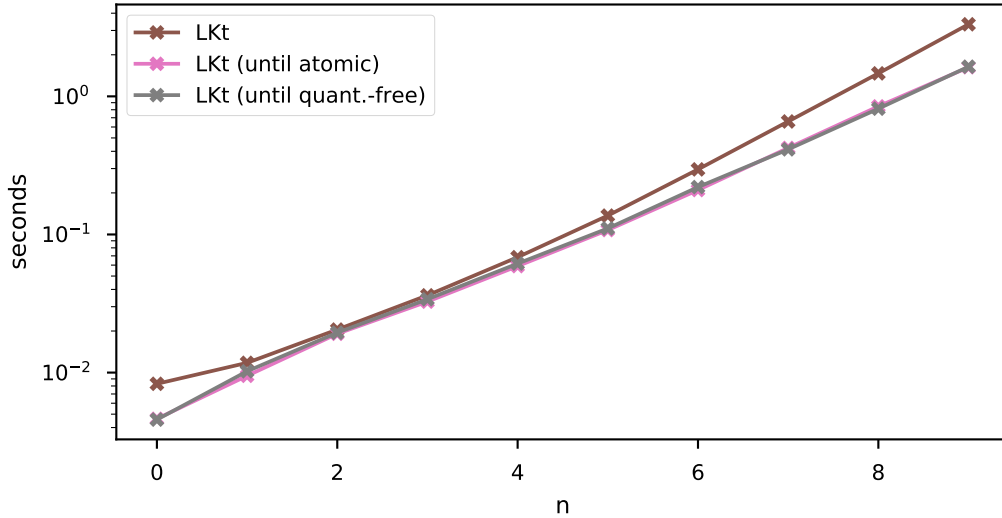Figure 5: Runtime of induction-elimination procedures on lemmas formalized in GAPT's library.

Figure 6: Normalization runtime on a formalization of Furstenberg's proof of the infinitude of primes.

4.2   Mathematical proofs

GAPT contains a small library of formalized proofs for testing. These are mostly basic properties of natural numbers and lists. The biggest formalized result is the fundamental theorem of arithmetic, showing the existence and uniqueness of prime decompositions for natural numbers. We evaluated the performance of LK_t as well as other procedures (see Section 4.1 for a description) on several of these proofs. Figure 5 shows the runtime of the induction- and cut-elimination. We tested instances of proofs of the following statements:

$$
\begin{array}{rl}
\texttt{add0l} & \forall x\,(0+x=x) \\
\texttt{mul1} & \forall x\,(x*1=x) \\
\texttt{filterrev} & \forall p\,\forall l\,(\mathrm{filter}(p,\mathrm{rev}(l))=\mathrm{rev}(\mathrm{filter}(p,l))) \\
\texttt{divmodgtot} & \forall a\,\forall b\,(b\neq 0 \rightarrow \exists d\,\exists r\,(r<b \wedge d*b+r=a)) \\
\texttt{primedecex} & \forall n\,(n\neq 0 \rightarrow \exists d\,\mathrm{primedec}(d,n))
\end{array}
$$

The proofs contain the primitive recursive definitions in the antecedent. For example, `add0l` is a proof with induction of the following sequent:

$$
\forall x\,(x+0=x),\ \forall x\,\forall y\,(x+s(y)=s(x+y)) \vdash \forall x\,(0+x=x)
$$

As before, induction-elimination in LK is several orders of magnitude slower than in LK_t. Semantic cut-elimination is surprisingly fast, it is as fast as LK_t for small instances of `filterrev`.

4.3   Furstenberg proof

Furstenberg's well-known proof of the infinitude of primes [14] equips the integers with a topology generated by arithmetic progressions, and uses this machinery to show that there are infinitely many primes. For every natural number $n \in \mathbb{N}$ we hence get a second-order proof π_n showing that there are more

than n prime numbers. Cut-elimination of π_n then extracts the computational content of Furstenberg's argument: we get a new prime number as a witness.

CERES was used to perform this extraction manually [3]. The key step in cut-elimination using CERES consists of the refutation of a so-called characteristic clause set. In the case of Furstenberg's proof automated theorem provers could only refute this first-order clause set for $n = 0$, that is, to show that there is more than one prime number. The authors hence manually constructed a sequence of refutations, taking Euclid's proof of the infinitude of primes as a guideline.

Using LK$_t$, GAPT can now perform the cut-elimination and extract the witness term automatically. Figure 6 shows the performance of the LK$_t$ normalization on instances of Furstenberg's proof. The concrete formalization closely resembles the one described in [3], however there have been minor changes to account for subtle differences in the LK calculus currently implemented in GAPT. Now that we could cut-eliminate this particular formalization for the first time, we were excited to find an interesting feature. We expected that the cut-elimination of Furstenberg's proof would compute the same witness as Euclid's proof: a prime divisor of $p_0 \cdots p_n + 1$. However as of version 2.10, we get the following witness instead, which contains 2 as an additional factor:

$$\texttt{primediv\_of}(1 + 2 * p(0) * \cdots * p(n))$$

This constant factor seems to depend on the concrete way in which we formalize the lemma that nonempty open sets are infinite (this lemma is called φ_2 in [3]). With a slightly different quantifier instance there, we can also get 3 instead of 2.

5 Future work

As our focus here lies in the practical applications of cut-elimination, termination of the LK$_t$ normalization procedure is only of secondary concern. For an actual implementation, there is little difference between an algorithm that does not terminate or one that terminates after a thousand years—as long as it quickly terminates on the instances we apply it to. For some classes of proofs, it is straightforward to see that normalization indeed always terminates. Due to the direct correspondence with the traditional presentation of LK, we can reuse termination arguments. Whenever we observe non-termination in the LK$_t$ normalization, we get a corresponding non-terminating reduction sequence in LK with an uppermost-first strategy. We believe that induction unfolding can be shown to terminate via a similar argument as used in [32] for the proof of the consistency of Peano Arithmetic. It remains open whether normalization terminates for proofs with higher-order (or even just second-order) quantifier inferences.

The current handling of equational and induction inferences as described in Sections 3.1 and 3.2 is unsatisfactory as they are not integrated in the main normalization function but require a separate pass over the proof. Furthermore, the normalized proof may contain Eql inferences on atoms. We are not aware of any terminating procedure using local rewrite rules that eliminates unary equational inferences such as the ones used in LK$_t$.

Renaming hypotheses and applying expression substitutions incurs a significant cost in the benchmarks. An obvious solution is to introduce an explicit substitution inference to implement these operations without the need to traverse the proof term. In fact, one of the motivations behind the substitution parameter σ in the typing judgment $::_\sigma$ was the support for explicit substitution inferences.

As a cheap optimization, we could grade-reduce blocks of quantifier inferences in a single substitution. This should speed up the common case of eliminating lemmas with many universal quantifiers. Another possible optimization is the use of caching: all of the functions $\mathcal{N}, \mathcal{E}$, and $\mathcal{S}$ are pure, making it easy to

cache their results. It is not clear whether normalization problems in LK_t repeat often enough to warrant a cache.

We used named variables as a binding strategy since this is traditionally used in GAPT. As expected, this choice has resulted in a number of overbinding-related bugs, which were difficult to debug. However, with named variables we can often avoid renaming when traversing and substituting proofs—where other approaches such as de Bruijn indices or locally nameless would always require renaming, or instantiation and abstraction, resp. Since every term in LK_t contains (multiple) binders, it seems prudent to avoid renaming in the common case. It may be possible to implement an efficient binding strategy using de Bruijn indices or a locally nameless representation by adding explicit renaming inferences.

Proof assistants such as Lean, Coq, or Minlog also provide functions to normalize proofs. It would be interesting to compare their performance to the approaches implemented in GAPT.

6   Conclusion

The normalization procedure described in this paper is fast, supports higher-order cuts, can unfold induction inferences, and can normalize cuts in the presence of all inference rules supported by GAPT. As shown in Section 4.3, we can now practically cut-eliminate proofs which were out of reach before.

However our ultimate interest lies in the quantifier structure of (cut-free) proofs as captured by Herbrand disjunctions or (in general) expansion proofs. From this point of view, we are not restricted to cut-elimination in LK or inessential variations like LK_t. Another option that is radically different from what we have considered so far is to use functional interpretation to compute expansion proofs as described by Gerhardy and Kohlenbach in [17].

For proofs with only universally quantified first-order cuts, a certain type of tree grammar describes the quantifier inferences [20], and the language generated by such a grammar then directly corresponds to a Herbrand sequent. We plan to develop and implement extensions of this grammar-based approach to general first-order cuts for an efficient extraction of Herbrand disjunctions, see [1] for grammars describing general prenex cuts.

References

[1] Bahareh Afshari, Stefan Hetzl & Graham Leigh (2018): *Herbrand's Theorem as Higher Order Recursion.* doi:10.14760/OWP-2018-01.

[2] Matthias Baaz, Stefan Hetzl, Alexander Leitsch, Clemens Richter & Hendrik Spohr (2006): *Proof Transformation by CERES.* In Jonathan M. Borwein & William M. Farmer, editors: *5th International Conference on Mathematical Knowledge Management, MKM, Lecture Notes in Computer Science* 4108, Springer, pp. 82–93, doi:10.1007/11812289_8.

[3] Matthias Baaz, Stefan Hetzl, Alexander Leitsch, Clemens Richter & Hendrik Spohr (2008): *CERES: An analysis of Fürstenberg's proof of the infinity of primes.* Theoretical Computer Science 403(2-3), pp. 160–175, doi:10.1016/j.tcs.2008.02.043. Available at https://doi.org/10.1016/j.tcs.2008.02.043.

[4] Matthias Baaz, Stefan Hetzl & Daniel Weller (2012): *On the complexity of proof deskolemization.* Journal of Symbolic Logic 77(2), pp. 669–686, doi:10.2178/jsl/1333566645.

[5] Matthias Baaz & Alexander Leitsch (2000): *Cut-elimination and Redundancy-elimination by Resolution.* Journal of Symbolic Computation 29(2), pp. 149–177, doi:10.1006/jsco.1999.0359.

[6] Franco Barbanera & Stefano Berardi (1996): *A Symmetric Lambda Calculus for Classical Program Extraction.* Information and Computation 125(2), pp. 103–117, doi:10.1006/inco.1996.0025.

[7] Samuel R. Buss (1995): *On Herbrand's Theorem*. In: *Logic and Computational Complexity*, *Lecture Notes in Computer Science* 960, Springer, pp. 195–209, doi:10.1007/3-540-60178-3_85.

[8] David Cerna & Anela Lolic (2018): *System Description: GAPT for schematic proofs*. Available at `http://www.risc.jku.at/publications/download/risc_5591/schematicGapt.pdf`. Preprint.

[9] Sebastian Eberhard & Stefan Hetzl (2015): *Inductive theorem proving based on tree grammars*. *Annals of Pure and Applied Logic* 166(6), pp. 665–700, doi:10.1016/j.apal.2015.01.002.

[10] Gabriel Ebner & Stefan Hetzl (2018): *Tree grammars for induction on inductive data types modulo equational theories*. Available at `https://gebner.org/pdfs/2018-01-29_indmodth.pdf`. Preprint.

[11] Gabriel Ebner, Stefan Hetzl, Alexander Leitsch, Giselle Reis & Daniel Weller (2017): *On the Generation of Quantified Lemmas*. Available at `https://gebner.org/pdfs/2018-03-03_lemmagen.pdf`. Preprint.

[12] Gabriel Ebner, Stefan Hetzl, Bernhard Mallinger, Giselle Reis, Martin Riener, Marielle Louise Rietdijk, Matthias Schlaipfer, Christoph Spörk, Janos Tapolczai, Jannik Vierling, Daniel Weller, Simon Wolfsteiner & Sebastian Zivota (2018): *GAPT user manual, version 2.10*. Available at `https://logic.at/gapt/downloads/gapt-user-manual.pdf`.

[13] Gabriel Ebner, Stefan Hetzl, Giselle Reis, Martin Riener, Simon Wolfsteiner & Sebastian Zivota (2016): *System Description: GAPT 2.0*. In Nicola Olivetti & Ashish Tiwari, editors: *International Joint Conference on Automated Reasoning (IJCAR)*, *Lecture Notes in Computer Science* 9706, Springer, pp. 293–301, doi:10.1007/978-3-319-40229-1_20.

[14] Harry Furstenberg (1955): *On the infinitude of primes*. *The American Mathematical Monthly* 62(5), p. 353, doi:10.2307/2307043.

[15] Gerhard Gentzen (1935): *Untersuchungen über das logische Schließen I*. *Mathematische Zeitschrift* 39(1), pp. 176–210. Available at `http://eudml.org/doc/168546`.

[16] Gerhard Gentzen (1936): *Die Widerspruchsfreiheit der reinen Zahlentheorie*. *Mathematische Annalen* 112, pp. 493–565.

[17] Philipp Gerhardy & Ulrich Kohlenbach (2005): *Extracting Herbrand disjunctions by functional interpretation*. *Archive for Mathematical Logic* 44(5), pp. 633–644, doi:10.1007/s00153-005-0275-1.

[18] Robert Harper (2016): *Practical foundations for programming languages*. Cambridge University Press.

[19] Jacques Herbrand (1930): *Recherches sur la théorie de la démonstration*. Ph.D. thesis, Université de Paris.

[20] Stefan Hetzl (2012): *Applying Tree Languages in Proof Theory*. In Adrian-Horia Dediu & Carlos Martín-Vide, editors: *Language and Automata Theory and Applications*, *Lecture Notes in Computer Science* 7183, Springer, pp. 301–312, doi:10.1007/978-3-642-28332-1_26.

[21] Stefan Hetzl, Alexander Leitsch, Giselle Reis, Janos Tapolczai & Daniel Weller (2014): *Introducing Quantified Cuts in Logic with Equality*. In Stéphane Demri, Deepak Kapur & Christoph Weidenbach, editors: *7th International Joint Conference on Automated Reasoning, IJCAR*, *Lecture Notes in Computer Science* 8562, Springer, pp. 240–254, doi:10.1007/978-3-319-08587-6_17.

[22] Stefan Hetzl, Alexander Leitsch, Giselle Reis & Daniel Weller (2014): *Algorithmic introduction of quantified cuts*. *Theoretical Computer Science* 549, pp. 1–16, doi:10.1016/j.tcs.2014.05.018.

[23] Stefan Hetzl, Alexander Leitsch & Daniel Weller (2011): *CERES in higher-order logic*. *Annals of Pure and Applied Logic* 162(12), pp. 1001–1034, doi:10.1016/j.apal.2011.06.005.

[24] Stefan Hetzl, Alexander Leitsch & Daniel Weller (2012): *Towards Algorithmic Cut-Introduction*. In: *Logic for Programming, Artificial Intelligence and Reasoning (LPAR-18)*, *Lecture Notes in Computer Science* 7180, Springer, pp. 228–242, doi:10.1007/978-3-642-28717-6_19.

[25] Stefan Hetzl & Daniel Weller (2013): *Expansion Trees with Cut*. *CoRR* abs/1308.0428. Available at `https://arxiv.org/abs/1308.0428`.

[26] David Hilbert & Paul Bernays (1939): *Grundlagen der Mathematik II*. Springer.

[27] Alexander Leitsch & Anela Lolic (2018): *Extraction of Expansion Trees*. *Journal of Automated Reasoning*, pp. 1–38, doi:10.1007/s10817-018-9453-9.

[28] Horst Luckhardt (1989): *Herbrand-Analysen zweier Beweise des Satzes von Roth: Polynomiale Anzahlschranken*. Journal of Symbolic Logic 54(1), pp. 234–263, doi:10.2307/2275028.

[29] Dale A. Miller (1987): *A compact representation of proofs*. Studia Logica 46(4), pp. 347–370, doi:10.1007/BF00370646.

[30] Michel Parigot (1992): *λμ-Calculus: An Algorithmic Interpretation of Classical Natural Deduction*. In Andrei Voronkov, editor: *Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, Lecture Notes in Computer Science 624, Springer, pp. 190–201, doi:10.1007/BFb0013061.

[31] Frank Pfenning (1995): *Structural Cut Elimination*. In: *Logic in Computer Science*, IEEE Computer Society, pp. 156–166, doi:10.1109/LICS.1995.523253.

[32] Gaisi Takeuti (1987): *Proof theory*, second edition. *Studies in Logic and the Foundations of Mathematics* 81, North-Holland Publishing Co., Amsterdam.

[33] Christian Urban & Gavin M. Bierman (2001): *Strong Normalisation of Cut-Elimination in Classical Logic*. *Fundamenta informaticae* 45(1-2), pp. 123–155. Available at `http://content.iospress.com/articles/fundamenta-informaticae/fi45-1-2-07`.