Faster Proof Checking in the Edinburgh Logical Framework

Aaron Stump, David L. Dill

Computer Systems Laboratory
Stanford University, Stanford, CA 94305, USA
E-mail: {stump,dill}@cs.stanford.edu
Web: http://verify.stanford.edu

Abstract. This paper describes optimizations for checking proofs represented in the Edinburgh Logical Framework (LF). The optimizations allow large proofs to be checked efficiently which cannot feasibly be checked using the standard algorithm for LF. The crucial optimization is a form of result caching. To formalize this optimization, a path calculus for LF is developed and shown equivalent to a standard calculus.

1 Introduction

The ability for automated reasoning systems to produce easily verifiable proofs has been widely recognized as valuable (e.g., [23, 4]). Recently, applications like proof-carrying code and proof-carrying authentication have created a new need for proofs that can be efficiently verified by a simple proof checker [15, 2]. The Edinburgh Logical Framework (LF) [10] is a widely used meta-language for representing proof systems for these applications. The representation is such that proof checking is reduced to LF type checking.

The CVC ("a Cooperating Validity Checker") [21] system has the capability to produce proofs in a variant of LF for valid formulas in a quantifier-free fragment of first-order logic with background theories. During validity checking, CVC computes values like a normal form n for a first-order term t. The computation produces a proof p that $t = n$. The normal form and the proof are both cached with t, and they can then both be reused later without repeating the computation. Internally, CVC represents proofs as a directed acyclic graph (DAG) with maximal sharing of common subexpressions. Reusing the cached normal form n for t results in a proof where the subproof p has more than one incoming edge in the DAG.

The problem then arises, how are proofs represented as DAGs to be checked? Since the DAG form for an expression can be exponentially smaller than the tree form, it would be disastrous to unfold the DAG to a tree by duplicating shared subexpressions wherever they occur. Proofs produced for large formulas by CVC can be megabytes long as a DAG, and at least gigabytes long as trees. Clearly, the proof checker must itself use some form of result caching if it is to check a proof represented as a DAG. The result that should be cached is the theorem (if any) that a subproof proves.

A naive approach to caching would have the proof checker cache the theorem which a subproof proves and reuse that result whenever that subproof is encountered subsequently. This approach is unsound, because syntactically identical subproofs may prove different theorems, depending on where they appear in a proof. This is because proofs introduce (named) local assumptions. For example, a proof ($\mathit{modus\text{-}ponens}\ \mathit{assumption}_1\ \mathit{assumption}_2$) might prove B in a part of the proof where $\mathit{assumption}_1$ is an assumption that $A \to B$ holds and $\mathit{assumption}_2$ is an assumption that A holds; and it might prove C in a part of the proof where $\mathit{assumption}_1$ is an assumption that $A \to C$ holds. Proofs produced by CVC have this kind of dependency of subproofs on local assumptions. So some form of caching is required that respects the scope of local assumptions.

One approach that is sound but turns out to be inefficient is to maintain a table H mapping subproofs to the theorems they prove, and simply clear H during proof checking whenever the set of active assumptions changes. This approach is sound because multiple occurrences of the same subproof do indeed prove the same theorem under the same set of assumptions. It is inefficient because a subproof may depend on only a subset of the set of active assumptions, and so clearing all results may cause the theorem proved by that subproof to be computed again unnecessarily. A prototype proof checker using this approach was unable to check the proofs produced by CVC in a reasonable amount of time and space.

A natural alternative to clearing all cached results whenever an assumption $\mathbf{a}$ changes is to clear cached results only for those subproofs that depend on $\mathbf{a}$. Also, there is no need to clear cached results eagerly, as soon as $\mathbf{a}$ changes. Results may be cleared lazily; whenever an attempt is made to reuse the theorem cached for some subproof, a check can be performed to see if the assumptions that the subproof depends on have changed. This is better than eager clearing of cached results, because not all subproofs will occur multiple times, and so not all cached results will need to be cleared. This lazy approach to caching has been implemented in a proof checker called flea, which is able to check example proofs produced by CVC in a modest amount of time and space that could not be checked at all previously.

The first contribution of this paper is to formalize a generalization of this approach to context-dependent caching in the setting of LF, and prove it correct with respect to a standard algorithm for LF type-checking (Sections 2, 3, and 4). To improve the performance of type checking further, two other optimizations are developed: variable-sensitive safe substitution (Section 5) and incremental classifier computation for left-nested applications (Section 6). Experimental results using flea are given for proofs produced by CVC (Section 7). The paper assumes some familiarity with type theory.

2 The Edinburgh Logical Framework

In this section, we briefly describe a standard calculus for LF. This is done in technical detail to enable the precise statement of subsequent theorems. Let Sym be an infinite set of symbols. A *binding* of a symbol $a \in \mathit{Sym}$ is an expression of the form $a : \tau$. If a list L of bindings contains a binding $x : \tau$, we say that L binds x. The addition of a binding $x : \tau$ to the end of a list of bindings Γ is denoted $\Gamma,\ x : \tau$.

Definition 1 (*lookup*). *If Γ is a list of bindings of symbols , then $lookup(v, \Gamma)$ is defined inductively as follows. If Γ is of the form $\Gamma', x : \tau$, then $lookup(v, \Gamma)$ equals τ if $v \equiv x$, and $lookup(v, \Gamma')$ otherwise. If Γ is the empty list, then $lookup(v, \Gamma)$ is undefined.*

Figure 1 presents a standard calculus for LF (cf. [17], Chapter 5 of [3]). The primary derivable objects are sequents of the form $\Gamma \vdash X : Y$. It is required in all rules that $\vdash \Gamma\ Ctx$ is derivable. Symbols may be bound more than once by Γ in sequents; the *lookup* function is used to find the most recently added binding in Γ for a symbol. Bound variables may not be tacitly renamed. The notation $A[x := N]$ is used for the result of safely substituting expression N for symbol x in expression A. A formal definition of safe substitution is delayed till Section 5 below. The rule (app) requires the domain type A of the operator M and the type A' of the argument N to be equivalent ($A \cong A'$). This is different from other presentations, which usually require those types to be syntactically identical, not just equivalent; they then use a separate rule of conversion to reclassify N with type A, if indeed $A \cong A'$. The test used for equivalence is the term-directed, context-independent one of [7], whose description is omitted here. It should be possible to extend the results to a context-dependent test like that of [11].

I. Classifications:

$$\text{(ax)}\ \frac{}{\Gamma \vdash v : A}\ A = lookup(v,\ \Gamma) \qquad \text{(type)}\ \frac{}{\Gamma \vdash type : kind}$$

$$\text{(lam)}\ \frac{\Gamma,\ x : A \vdash M : B \qquad \Gamma \vdash \Pi\, x : A.\, B : \alpha}{\Gamma \vdash \lambda\, x : A.\, M : \Pi\, x : A.\, B}\ \alpha \in \{type, kind\}$$

$$\text{(app)}\ \frac{\Gamma \vdash M : \Pi\, x : A.\, B \qquad \Gamma \vdash N : A'}{\Gamma \vdash (M\ N) : B[x := N]}\ A \cong A'$$

$$\text{(pi)}\ \frac{\Gamma \vdash A : type \qquad \Gamma,\ x : A \vdash B : \alpha}{\Gamma \vdash \Pi\, x : A.\, B : \alpha}\ \alpha \in \{type, kind\}$$

II. Contexts:

$$\text{(ctxemp)}\ \frac{}{\vdash \cdot\ Ctx} \qquad \text{(ctxadd)}\ \frac{\Gamma \vdash A : \alpha}{\vdash \Gamma,\ c : A\ Ctx}\ \alpha \in \{type, kind\}$$

Fig. 1. A standard presentation of LF

3 An annotation calculus for LF

This section presents a new calculus for LF, which is used to present the context-dependent caching optimization in Section 4 below. It is clear that some technical machinery beyond the standard calculus of Section 2 is required to describe caching. The technical device used here is that of typing annotations. The new calculus for LF derives an entire typing annotation of an expression e and all its subexpressions, as opposed to the type of a single expression only. Caching may then be formalized in a rule that says when typing information may be copied from one occurrence of a subterm of e to annotate another occurrence. An alternative technical device for formalizing caching, but one which we do not pursue here, might be some kind of explicit definition or abbreviation.

3.1 Preliminaries

The standard set-theoretic notion of partial function is assumed, together with related set-theoretic operations such as union, which will be performed on functions (cf. [9]). For a unary partial function e, we use the notation $e(x)\downarrow$ to mean e is defined at x and $e(x)\uparrow$ to mean it is not. The domain of definition of e is denoted $\mathit{Def}(e)$. Expressions are formalized as unary partial functions from the set of positions (Definition 2) to a set of symbols (as in, e.g., [6]). This approach allows expressions and annotations of expressions to be treated uniformly. We assume a countably infinite set Sym of symbols (disjoint from $\{\mathit{type}, \mathit{kind}, \lambda, \Pi, @\}$.)

Definition 2 (***Pos*** **: the set of positions).** *Let Nat be the set of natural numbers. The inductive set Pos of positions is defined by the following constructors:*

$$\begin{array}{ll} \epsilon : \mathit{Pos} & \\ . : \mathit{Pos} \rightarrow \mathit{Nat} \rightarrow \mathit{Pos} & \text{(applications of "." written infix)} \end{array}$$

If $n \in \mathit{Nat}$, n will often be used ambiguously to denote $\epsilon.n$.

We now define operations $\gg$ and $\ll$. The first right-shifts a position by prepending another position as a prefix, and the second left-shifts a position to remove a prefix. The notation is chosen to recall standard right and left bit-shifting operations. In meta-theoretic expressions, $\ll$ and $\gg$ will bind more loosely than the "." constructor.

Definition 3 ($\ll$ and $\gg$: shifting positions). *If $\pi, \psi \in \mathit{Pos}$ and $n \in \mathit{Nat}$, then the operation $\gg$ of prepending ψ to π is defined inductively by: $\epsilon \gg \psi = \psi$ and $\pi.n \gg \psi = (\pi \gg \psi).n$. Now suppose $\pi = (\pi' \gg \psi)$. Then $\pi \ll \psi$ is defined to be π'. Otherwise, it is undefined.*

Definition 4 (***Exp*** **: the set of LF expressions).** *The set Exp of LF expressions is the set of all partial functions e from Pos to $\mathit{Sym} \cup \{\mathit{type}, \mathit{kind}, \lambda, \Pi, @\}$ satisfying the following requirements:*

- *e is undefined on all but a finite non-empty subset of Pos .*

- *Def*(e) *is prefix-closed: if* $e(\pi.n)\downarrow$*, then* $e(\pi)\downarrow$.
- *If* $e(\pi) \in \mathit{Sym} \cup \{\mathit{type}, \mathit{kind}\}$*, then* $\{i \mid e(\pi.i)\downarrow\} = \emptyset$.
- *If* $e(\pi) = @$*, then* $\{i \mid e(\pi.i)\downarrow\} = \{0,1\}$.
- *If* $e(\pi) \in \{\lambda, \Pi\}$*, then* $e(\pi.0) \in \mathit{Sym}$ *and* $\{i \mid e(\pi.i)\downarrow\} = \{0,1,2\}$.

We will use the customary notation for LF expressions. The inductive definition of this notation is omitted, as is the justification for the principle of structural induction on LF expressions, which we will also use.

We now extend $\gg$ and $\ll$ to expressions. The intention is that if e is an expression with $e(\pi)\downarrow$, then $(e \ll \pi)$ is the subexpression at position π in e (usually written $e|_\pi$).

Definition 5 ($\gg$ and $\ll$ extended). *The extensions of* $\gg$ *and* $\ll$ *which shift a function whose domain is* Pos *by a position are also denoted* $\gg$ *and* $\ll$*. For input* $\pi' \in \mathit{Pos}$, $(f \gg \pi)$ *and* $(f \ll \pi)$ *are defined by*

$$(f \gg \pi)(\pi') = f(\pi' \ll \pi)$$
$$(f \ll \pi)(\pi') = f(\pi' \gg \pi)$$

Equivalently, viewing f *as a set of pairs,* $(\_ \gg \pi)$ *and* $(\_ \ll \pi)$ *are applied to* f *by applying them to the first component of each pair. We also extend* $\gg$ *and* $\ll$ *to shift sets of positions by a position. The definition for* $S \subseteq \mathit{Pos}$ *is*

$$(S \gg \pi) = \{\psi \gg \pi | \psi \in S\}$$
$$(S \ll \pi) = \{\psi \ll \pi | \psi \in S\}$$

Example 1. Consider the following expression e:

$$\{\epsilon \mapsto @,\ 0 \mapsto @,\ 0.0 \mapsto f,\ 0.1 \mapsto a,\ 1 \mapsto b\}$$

In customary notation, this is denoted $((f\ a)\ b)$, and $(e \ll 0)$ is $(f\ a)$, since, e.g., $(e \ll 0)(0) = e(0.0) = f$.

Definition 6 (annotations). *Suppose* $e \in \mathit{Exp}$ *. An annotation of* e *is a partial function from* *Def*(e) *to* Exp *.*

An annotation of $e \in \mathit{Exp}$ labels some subexpressions of e with other expressions.

3.2 A calculus of annotations

In this section, a new calculus for LF is presented. Derivable objects are essentially annotations of LF expressions. For a given $e \in \mathit{Exp}$, the idea is to derive an annotation a of e such that for all $\pi \in \mathit{Def}(e)$, $a(\pi) = \tau$ iff $\Gamma \vdash e(\pi) : \tau$ is derivable in the standard calculus, where Γ is a context determined by the bindings occurring at prefixes of π in e. By deriving such annotations, we make all computed classifiers available at every point in the derivation. This sets the stage for reusing computed classifiers in Section 4. Similar technical machinery is used in [8]. There is also some resemblance to the marked calculus for LF of [22].

Notation 1 (annotations) *Suppose a is an annotation of $e \in Exp$ with $Def(a) = \{\pi_1, \ldots, \pi_n\}$. Then a may be denoted $\{\pi_1 : a(\pi_1), \ldots, \pi_n : a(\pi_n)\}$.*

Figure 2 presents the calculus. The primary derivable objects are sequents $\Sigma \,|\, e \;\vdash\; a$, where a is an annotation and $e \in Exp$. It is required in all rules that $\vdash \Sigma \; Sig$ is derivable. The list Σ is called a signature instead of a context because it never changes during a derivation (after it has been proved well-formed using (sigemp) and (sigadd)). The (ax) rule specifies an expression's initial annotation, which is defined in the last part of Figure 2 by structural induction. The classificational rules (lam), (app), and (pi) just show how an annotation is extended to a new position. For the premises of those rules, the notation $\Sigma \,|\, e \vdash a \supseteq X$ means that $\Sigma \,|\, e \vdash a$ is derivable, where $a \supseteq X$.

A few further remarks on the calculus are needed. Meta-theoretic expressions like $(e \ll \pi.1)$ in the conclusion of (lam) denote the expression resulting from performing the shift. Second, the standard calculus is used computationally by applying the rules bottom-up to analyze a goal classification into subgoals. In contrast, the annotation calculus is used by applying the rules top-down to saturate the initial annotation specified by (ax). Finally, it is not obvious that the annotation calculus of Figure 2 is well-defined, since it is not obvious that the derived objects are always functional and not sometimes merely relational. Functionality up to equivalence will be a consequence of the equivalence of the annotation calculus with the standard presentation of LF, since LF enjoys unicity of classifiers up to equivalence (theorem 2.4 of [10]).

Notation 2 (derivable sequents) *In addition to denoting sequents (theoretical objects), expressions like $\Sigma \,|\, e \vdash a$ will be used to denote the meta-theoretic proposition that the corresponding sequent is derivable in the annotation calculus.*

3.3 Equivalence with the standard calculus

This section states the equivalence of the annotation calculus and the standard calculus of Section 2. This result may be proved from three lemmas about the annotation calculus, which are stated first. Omitted proofs may be found in [20].

Lemma 1 (monotonicity). *Suppose $\Sigma \,|\, e \vdash a$ is derivable starting from a sequent $\Sigma \,|\, e \vdash a'$ (used as an assumption to which the inference rules are then applied). Then $\Sigma \,|\, e \vdash a \cup X$ is derivable starting from sequent $\Sigma \,|\, e \vdash a' \cup X$. Furthermore, $\Sigma \,|\, e \vdash a$ implies $a \supseteq I_\Sigma(e)$.*

Proof: Both claims follow by induction on the assumed derivation: every rule only extends annotations (starting from the initial annotation $I_\Sigma(e)$), and no rule is prevented from being applied by definedness of an annotation at a position. $\square$

Lemma 2 ($\gg$-shifting sequents). *Let $\Xi \in \{\lambda, \Pi\}$ be arbitrary.*

i. $\Sigma \,|\, M_i \vdash a$ implies $\Sigma \,|\, (M_0\; M_1) \vdash (a \gg i) \cup I_\Sigma(M_0\; M_1)$, for $i \in \{0, 1\}$

ii. $\Sigma \,|\, \sigma \vdash a$ implies $\Sigma \,|\, \Xi\, x : \sigma.\, M \vdash (a \gg 1) \cup I_\Sigma(\Xi\, x : \sigma.\, M)$

iii. $\Sigma, x : \sigma \,|\, M \vdash a$ implies $\Sigma \,|\, \Xi\, x : \sigma.\, M \vdash (a \gg 2) \cup I_\Sigma(\Xi\, x : \sigma.\, M)$

I. Classifications:

$$\text{(ax)}\ \frac{}{\Sigma \,|\, e \vdash I_\Sigma(e)}$$

$$\text{(lam)}\ \frac{\Sigma \,|\, e \vdash a \supseteq \{\pi.2 : \tau\} \quad \Sigma \,|\, \Pi\, x : \sigma.\, \tau \vdash a' \supseteq \{\epsilon : \alpha\}}{\Sigma \,|\, e \vdash a \cup \{\pi : \Pi\, x : \sigma.\, \tau\}} \ \begin{array}{l} e(\pi) = \lambda \text{ and } \alpha \in \{\mathit{type}, \mathit{kind}\}, \\ x = (e \ll \pi.0) \text{ and } \sigma = (e \ll \pi.1) \end{array}$$

$$\text{(app)}\ \frac{\Sigma \,|\, e \vdash a \supseteq \{\pi.0 : \Pi\, x : \sigma.\, \tau, \quad \pi.1 : \sigma'\}}{\Sigma \,|\, e \vdash a \cup \{\pi : \tau[x := (e \ll \pi.1)]\}} \ e(\pi) = @ \text{ and } \sigma \cong \sigma'$$

$$\text{(pi)}\ \frac{\Sigma \,|\, e \vdash a \supseteq \{\pi.1 : \mathit{type}, \quad \pi.2 : \alpha\}}{\Sigma \,|\, e \vdash a \cup \{\pi : \alpha\}} \ e(\pi) = \Pi \text{ and } \alpha \in \{\mathit{type}, \mathit{kind}\}$$

II. Signatures:

$$\text{(sigemp)}\ \frac{}{\vdash \cdot\ \mathit{Sig}} \qquad \text{(sigadd)}\ \frac{\Sigma \,|\, e \vdash a}{\vdash \Sigma,\, c : e\ \mathit{Sig}} \ a(\epsilon) \in \{\mathit{type}, \mathit{kind}\}$$

III. Initial annotation:

$$\begin{array}{lll} & I_\Sigma(\mathit{type}) & = \{\epsilon : \mathit{kind}\} \\ & I_\Sigma(\mathit{kind}) & = \emptyset \\ & I_\Sigma(M_0\ M_1) & = (I_\Sigma(M_0) \gg 0) \cup (I_\Sigma(M_1) \gg 1) \\ \text{for } \Xi \in \{\lambda, \Pi\}, & I_\Sigma(\Xi\, x : \sigma.\, M) & = (I_\Sigma(\sigma) \gg 1) \ \cup \ (I_{\Sigma,\, x:\sigma}(M) \gg 2) \\ \text{for } v \in \mathit{Sym}, & I_\Sigma(v) & = \begin{cases} \{\epsilon : \mathit{lookup}(v, \Sigma)\} & \text{if } \mathit{lookup}(v, \Sigma)\downarrow, \\ \emptyset & \text{otherwise} \end{cases} \end{array}$$

Fig. 2. Annotation calculus for LF

Lemma 3 ($\ll$-shifting sequents). *Let $\Xi \in \{\lambda, \Pi\}$ be arbitrary.*

$$\begin{array}{llll} i. & \Sigma \,|\, (M_0\ M_1) \vdash a & \text{implies} & \Sigma \,|\, M_i \vdash (a \ll i), \ \text{for } i \in \{0, 1\} \\ ii. & \Sigma \,|\, \Xi\, x : \sigma.\, M \vdash a & \text{implies} & \Sigma \,|\, \sigma \vdash (a \ll 1) \\ iii. & \Sigma \,|\, \Xi\, x : \sigma.\, M \vdash a & \text{implies} & \Sigma,\, x : \sigma \,|\, M \vdash (a \ll 2) \end{array}$$

Theorem 1 (correctness of annotation calculus). *Let $e, \tau \in \mathit{Exp}$ be arbitrary, and let Δ be an arbitrary list of bindings. Then the following are equivalent:*

1. *$\vdash \Delta\ \mathit{Ctx}$ and $\Delta \vdash e : \tau$ are both derivable in the standard calculus for LF.*
2. *There is an annotation a of e with $a(\epsilon) = \tau$ such that $\vdash \Delta\ \mathit{Sig}$ and $\Delta \,|\, e \vdash a$ are both derivable in the annotation calculus.*

4 Context-dependent classifier caching

This section presents the formalization of the approach to caching and reusing computed classifiers described in Section 1, and proves it sound. Some preliminary definitions are needed.

Definition 7 ($FS(e)$: free symbols of e). *The function FS computing the set of symbols occurring freely in $e \in Exp$ is defined by:*

$$\begin{aligned}
FS(type) &= FS(kind) = \emptyset \\
FS(M\ N) &= FS(M) \cup FS(N) \\
\text{for } \Xi \in \{\lambda, \Pi\}, FS(\Xi\, x : \sigma.\, M) &= FS(\sigma) \cup (FS(M) \setminus \{x\}) \\
\text{for } v \in Sym, FS(v) &= \{v\}
\end{aligned}$$

Definition 8 ($Ctx_e(\pi)$: context at π in e). *Let $e \in Exp$, $\pi \in Def(e)$, and $n \in Nat$. $Ctx_e(\pi)$ is the list of bindings at proper prefixes of π in e:*

$$Ctx_e(\epsilon) = \cdot$$

$$Ctx_e(\pi.n) = \begin{cases} Ctx_e(\pi), (e \ll \pi.0) : (e \ll \pi.1) & \text{if } e(\pi) \in \{\lambda, \Pi\} \text{ and } n = 2 \\ Ctx_e(\pi) & \text{otherwise} \end{cases}$$

Example 2. Let e be $\lambda\, x : \sigma.\, \lambda\, y : \sigma.\, \lambda\, x : \tau.\, x$. Then

$$Ctx_e(2) = x : \sigma \qquad Ctx_e(2.2.2) = x : \sigma,\ y : \sigma,\ x : \tau$$

Definition 9 ($Dep_e(\pi)$: dependencies at π in e). *Let $e \in Exp$ and $\pi \in Def(e)$. $Dep_e(\pi) = \{x : lookup(x, Ctx_e(\pi)) \ \mid\ x \in FS(e \ll \pi)\}$.*

Intuitively, the intention is that $Dep_e(\pi)$ be the set of all those bindings in $Ctx_e(\pi)$ that the classification of $(e \ll \pi)$ depends on.

Example 3. Let e be as in Example 2. Then $Dep_e(2.2.2) = \{x : \tau\}$.

Now consider the extension of the annotation calculus of Figure 2 by the following new rule:

$$(\text{copy})\,\frac{\Sigma \,|\, e \,\vdash\, a}{\Sigma \,|\, e \,\vdash\, a \cup ((a \ll \pi) \gg \pi')} \ \ (e \ll \pi) = (e \ll \pi') \text{ and } Dep_e(\pi) = Dep_e(\pi')$$

Suppose subexpression $(e \ll \pi)$ of e occurs at position $\pi' \neq \pi$ (as well as at position π). Then informally, the (copy) rule says that if the annotation for the subexpression at the first position has been computed, then it can be simply copied in a single step to annotate that subexpression at the second position (π'), as long as the dependencies of the subexpression are the same at the two positions.

Example 4 (derivation using (copy)). Let Σ be a signature

$$\tau : type,\ f : \tau \to (\tau \to \tau),\ g : \tau \to \tau,\ \ldots.$$

Let e be $\lambda x : \tau.\, (f\ (g\ (g\ x)))\ (g\ (g\ x))$. The following derivation uses (copy) to avoid recomputing the annotation for the second occurrence of $(g\ (g\ x))$. Note that the position of the first occurrence of that expression is 2.0.1 and the position of the second occurrence is 2.1. We abbreviate the derivation by showing just how annotations are extended. The rules used are, from top to bottom, (ax), (app), (app), and (copy). In the instance of (copy), $\pi = 2.0.1$ and $\pi' = 2.1$. Note that (copy) extends the annotation at several positions, because (copy) copies the entire annotation of one subexpression to another. So a single application of (copy) annotates at the second occurrence of $(g\ (g\ x))$ all those subterms which are annotated at the first occurrence.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma \,|\, e \vdash \{\ldots,\ 2.0.1.0 : \tau \to \tau,\ 2.0.1.1.0 : \tau \to \tau,\ 2.0.1.1.1 : \tau\}}}{2.0.1.1 : \tau}}{2.0.1 : \tau}}{2.1.1 : \tau,\ 2.1 : \tau}$$

Theorem 2 (conservativity of (copy)). *A sequent is derivable in the calculus with (copy) iff it is derivable in the original calculus without (copy).*

Proof: The "if" direction of the proof is obvious. For the "only if" direction, the proof is by induction on the assumed derivation with (copy). All cases follow easily using the induction hypothesis, except the case for (copy). In that case, by the induction hypothesis, we have a derivation without (copy) of $\Sigma \,|\, e \vdash a$. We must show that $\Sigma \,|\, e \vdash a \cup ((a \ll \pi) \gg \pi')$ is derivable without (copy), where $(e \ll \pi) = (e \ll \pi')$ and $Dep_e(\pi) = Dep_e(\pi')$.

Let $C = Ctx_e(\pi)$ and $C' = Ctx_e(\pi')$. Observe first that

$$\Sigma \,|\, e \vdash a \ \text{ implies } \ \Sigma, C \,|\, (e \ll \pi) \vdash (a \ll \pi).$$

This follows readily by induction on π using Lemma 3 (shifting). Then we can show that $\Sigma, C' \,|\, (e \ll \pi') \vdash (a \ll \pi)$ is derivable. This is proved by showing that $I_{\Sigma,C}(e \ll \pi) = I_{\Sigma,C'}(e \ll \pi')$, which is done readily by induction on the structure of $(e \ll \pi)$: all cases go through easily using the induction hypothesis, except for the case when $(e \ll \pi)$ is a symbol, where the fact that $Dep_e(\pi) = Dep_e(\pi')$ is used. Finally, it can then be proved by induction on π' using Lemma 2 (shifting) that

$$\Sigma, C' \,|\, (e \ll \pi') \vdash (a \ll \pi) \ \text{ implies } \ \Sigma \,|\, e \vdash ((a \ll \pi) \gg \pi') \cup I_\Sigma(e)$$

The result now follows using Lemma 1 (monotonicity). □

4.1 Implementation

The (copy) rule forms the theoretical basis for context-dependent caching. One way to implement (copy) would be to maintain a hash table H mapping a pair of the form $\langle (e \ll \pi), Dep_e(\pi)\rangle$ to the classifier computed for $(e \ll \pi)$ at a position with dependencies $Dep_e(\pi)$. This would result in the maximum reusing of computed classifiers allowed by (copy). The table H, however, could become very large, with no guarantee that cached results would be reused frequently. The flea proof checker implements a more memory-efficient approximation to this scheme; the approximation seems to work well in practice, although there are cases where it achieves less reuse than the full caching scheme. We approximate $Dep_e(\pi)$ by the position of e's deepest relevant binding. More precisely, we approximate $Dep_e(\pi)$ by the longest prefix ψ of π in e such that the binding $(e \ll \psi.0) : (e \ll \psi.1)$ is in $Dep_e(\pi)$. This position ψ is called the *context id* of the occurrence of $(e \ll \pi)$ at π. Now H is taken to be a hash table from subexpressions $(e \ll \pi)$ to pairs $\langle \tau, id \rangle$, where τ is a classifier and id is the context identifier of the occurrence of $(e \ll \pi)$ for which τ was computed. Before trying to compute the classifier for a subexpression s, H is consulted to see whether or not there is a cached classifier τ for s which has the same context identifier as this occurrence of s. If so, τ is used for this occurrence. If not, the classifier τ' for s at this occurrence is computed, and H is modified to map s to $\langle \tau', id \rangle$, where id is the context id for s at this occurrence. See Section 7 for empirical results.

4.2 Proof compression

We briefly sketch an approach based on context-dependent caching to proof compression. An expression e is compressed as follows. With each subexpression $(e \ll \pi)$ at position π, we cache in a context-dependent way an abbreviation

$$a := \lambda\, x_1 : \tau_1. \ldots \lambda\, x_n : \tau_n. (e \ll \pi)$$

where $Dep_e(\pi) = \{x_1 : \tau_1, \ \ldots, \ x_n : \tau_n\}$ and a is some new symbol. We then transform the subexpression $(e \ll \pi)$ by replacing it with $(\ldots(a\ x_1)\ldots\ x_n)$. If the type checker permits, the abbreviation can be inserted at the position in the transformed expression corresponding to the context id of this subexpression at this position π. More work is required to pull the abbreviation all the way to the front of the transformed expression. This is because the types $\tau_1, \ldots, \tau_n$ may contain free variables besides the ones $x_1, \ldots, x_n$ accounted for in the body of the abbreviation. To pull the abbreviation upward in the expression across a λ- or Π-binding of a variable v, if the body of the abbreviation as it currently stands contains v free, then that body will have to be augmented by a λ-abstraction of v. Applications of the new constant a will then have to take in v as another argument.

5 Variable-sensitive safe substitution

Profiling runs of flea checking proofs produced by CVC typically show that a large fraction of the time (around 30% on the dlx-pc example of Section 7) is spent performing safe substitutions. In this section, we describe some improvements to safe substitution. We define a substitution to be a function $\sigma : Sym \rightarrow Exp$ with a finite domain of definition $\{v_1, \ldots, v_n\}$. It may be denoted $\{v_1 := \sigma(v_1), \ \ldots, \ v_n := \sigma(v_n)\}$, possibly without the curly brackets. Let $\Xi \in \{\lambda, \Pi\}$ be arbitrary. Then the safe application $e[\sigma]$ of such a substitution σ to $e \in Exp$ is defined by induction on the structure of e. In the second case for λ-abstractions, $\hat{x}$ is a symbol not occurring in M or the range of σ.

$$\alpha[\sigma] = \alpha, \ \text{for } \alpha \in \{type, kind\}$$

$$(M\ N)[\sigma] = (M[\sigma]\ N[\sigma])$$

$$(\Xi\, x : A.\, M)[\sigma] = \begin{cases} (\Xi\, x : A[\sigma].\, M[\sigma']) & \text{if } \sigma = (\sigma' \cup \{x := t\}) \text{ for some } t \\ \Xi\, \hat{x} : A[\sigma].\, M[\sigma \cup \{x := \hat{x}\}] & \text{if } x \in \mathit{FS}(\sigma(v)) \text{ for some } v, \\ \Xi\, x : A[\sigma].\, M[\sigma] & \text{otherwise} \end{cases}$$

$$\text{for } v \in \mathit{Sym}, v[\sigma] = \begin{cases} \sigma(v) & \text{if } \sigma(v)\downarrow \\ v & \text{otherwise} \end{cases}$$

Safe substitution requires sets of free symbols of expressions to be computed. If the symbols bound by signature Σ are disjoint from the symbols bound by all Π- and λ-abstractions, then instead of $\mathit{FS}(e)$ it can easily be shown that we can use the set of free variables of e (relative to Σ), defined by

$$FV_\Sigma(e) = \{v \in \mathit{FS}(e) \mid \Sigma \text{ does not bind } v\}.$$

Furthermore, $FV_\Sigma(e)$ can be cached with e, so it need not be recomputed. Also, it is to be expected that many expressions will have the same set of free variables. A set S of the sets of free variables is maintained, and memory is allocated for a new set of free variables only if a set storing exactly those variables is not already in S. Finally, if $FV_\Sigma(e) \cap \mathit{Def}(\sigma) = \emptyset$, then an easy inductive proof shows that $e[\sigma] = e$. So there is no need to compute $e[\sigma]$ if this intersection is empty.

5.1 de Bruijn indices

The optimizations of safe substitution are described for the version of LF that uses named variables; this is what is implemented in flea. An alternative is to use de Bruijn indices for variables (see, e.g., [13]). The optimization of returning e for $e[\sigma]$ without actually applying the substitution in the case where $FV_\Sigma(e) \cap \mathit{Def}(\sigma) = \emptyset$ is still relevant with de Bruijn indices. The definition of $\mathit{FS}(\Xi\, x : \sigma.\, M)$ for $\Xi \in \{\lambda, \Pi\}$ is changed to be the following:

$$\mathit{FS}(\Xi\, x : \sigma.\, M) = \mathit{FS}(\sigma) \cup \{v - 1 | v \in \mathit{FS}(M)\}$$

The optimization is performed by checking the intersection of $\mathit{FS}(e)$ and $\mathit{Def}(\sigma)$ for emptiness. It would then make sense to share sets of free variables, as proposed in the case of named variables. We might expect many fewer different sets of de Bruijn indices, and so the optimization of using a set of sets of free variables could be even more effective. Furthermore, sets of de Bruijn indices can be compactly represented as bitvectors.

5.2 Explicit substitutions

A further improvement, which was also not implemented in flea, would be to use explicit substitutions and cache the result of computing $e[\sigma]$. This could be done by maintaining a hash table mapping expressions $e[\sigma]$ to the results of carrying out the substitutions. Using such a caching scheme could help improve performance, although it might be necessary to use a cache of fixed maximum size and implement some cache replacement policy to keep from using too much memory.

6 Classifier computation for left-nested applications

A left-nested application is an application like $(((c\ a_1)\ a_2)\ a_3)$. Suppose the classifier for a classifiable left-nested application $(\ldots(c\ a_1)\ldots\ a_n)$ is to be computed. Assuming that classifiers $\tau_1', \ldots, \tau_n'$ for $a_1, \ldots, a_n$ and $\Pi\, x_1 : \tau_1.\ \ldots \Pi\, x_n : \tau_n.\, \tau$ (call this *optype*) for c have already been computed, the derivation in the annotation calculus (with positions written in bold for readability) would be

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdots}{\mathbf{0^{n-1}.0} : \Pi\, x_1 : \tau_1.\ \ldots \Pi\, x_n : \tau_n.\, \tau, \quad \mathbf{0^{n-1}.1} : \tau_1'}}{\mathbf{0^{n-2}.0} : \Pi\, x_2 : \tau_2[x_1 := a_1].\ \ldots \Pi\, x_n : \tau_n[x_1 := a_1].\, \tau[x_1 := a_1], \quad \mathbf{0^{n-2}.1} : \tau_2}}{\vdots}}{\mathbf{0} : \Pi\, x_n : \tau_n[x_1 := a_1]\ldots[x_{n-1} := a_{n-1}].\, \tau[x_1 := a_1]\ldots[x_{n-1} := a_{n-1}], \quad \mathbf{1} : \tau_n}}{\epsilon : \tau[x_1 := a_1]\ldots[x_n := a_n]}$$

where the following definition has been used:

Definition 10 (iterated "."). *For $n, m \in Nat$, n^m is defined inductively by $n^0 = \epsilon$ and $n^{m+1} = (n^m).n$*

Several things may be observed about how substitutions are carried out in this derivation. First, $\Omega(n^2)$ time is spent doing substitutions, since in the i'th step of the derivation, a substitution is performed on $(optype \ll 2^i)[x_1 := a_1]\ldots[x_{i-1} := a_{i-1}]$. This last expression is at least as big as $(optype \ll 2^i)$, which is of size at least $n - i$. There are n steps in the derivation, so the total time is at least quadratic. Second, consider what happens in, for example, step 3 of such a derivation. If τ_3 has an occurrence of x_1 in it, then computing $\tau_3[x_1 := a_1][x_2 := a_2]$ will require computing $a_1[x_2 := a_2]$, even though a_1 cannot possibly contain x_2 free. Variable-sensitive substitution would keep the second observation from leading to inefficiency, but not the first; the following solution prevents both problems. The annotation calculus's (app) rule is replaced by the rule (app') below. For notational simplicity, we give just the instance of the rule that extends the annotation at position ϵ, instead of at an arbitrary position. For $i \in \{1, \ldots, n\}$, let a_i abbreviate $(e \ll 0^{n-i}.1)$, which is the i'th innermost argument in the nested application.

$$\text{(app')}\ \dfrac{\mathbf{0^{n-1}.0} : \Pi\, x_1 : \tau_1.\ \ldots \Pi\, x_n : \tau_n.\, \tau, \quad \mathbf{0^{n-1}.1} : \tau_1',\ \ldots,\ \mathbf{1} : \tau_n'}{\epsilon : \tau[x_1 := a_1, \ldots, x_n := a_n]}\ \phi$$

The side condition ϕ on the rule is the conjunction of the following:

- $e(0^i) = @$, for all $i \in \{0, \ldots, n-1\}$
- $\tau_1 \cong \tau_1'$
- for $i \in \{2, \ldots, n-1\}$, $\tau_i[x_1 := a_1, \ldots, x_{i-1} := a_{i-1}] \cong \tau_i'$

In the case where for some $1 \leq i < j \leq n$, $x_i \equiv x_j$, we stipulate that the binding of x_i occurring further to the left in the substitution is dropped from the substitution. If the types $\tau_1, \ldots, \tau_n$ are each of constant size, then this rule (whose proof of correctness we omit to save space) requires only $O(n)$ time to be spent performing substitutions. This is because the side condition applies a substitution (of size $O(i)$) n times to something of constant size.

7 Empirical results

This section gives some empirical results on proofs produced by CVC for formulas generated from verification problems. The proofs use some extensions to LF for dealing with lists more directly. In order to compare with two other systems implementing LF type-checking, a large piece of one example was hand-translated to remove the extensions, and common subexpressions were pulled out using abbreviations (see Section 4.2); dlx-pc.pure is the result. This example could then be run through Twelf [18] and, with a minor translation, LEGO [14].[1]

Figure 3 gives the results. Entries in the table show time and peak memory usage on a 850MHz Pentium III with 256M of main memory. "all optimizations" is the flea proof checker with all the optimizations described above. "-distinct vars" is "all" except without the optimization allowed by keeping bound variables distinct from constants declared in the signature (Section 5). "-intersection check" is "all" without the check on the intersection of domain of definition of substitution and set of free variables of expression (Section 5). "-fv sets" is "all" except that memory for a set of free variables is allocated without consulting a set of sets of free variables (Section 5). "-left nest" is "all" except without the modification to allow efficient computation of the classifier of left-nested applications (Section 6). Example size is the size in ASCII text with maximal sharing of common subexpressions. No results are given in the table for checking without context-dependent caching, because as mentioned earlier, checking is not feasible without this optimization. As one example of this, a small subproof of dlx-pc of size 90K takes 1.3s to check with context-dependent caching and over 1 minute to check without. This is because as a tree, the subproof is 3.5M long.

	dlx-pc.pure	dlx-pc	satyaki5	pp-invariant
all optimizations	4.4s, 31M	24.7s, 49M	21.6 s, 36M	68s, 68M
-distinct vars	8.9s, 42M	46.7s, 65M	47.6s, 39M	97s, 95M
-intersection check	4.0s, 31M	>1000s, >256M	720s, 167M	127s, 72M
-fv sets	4.1s, 32M	22.3s, 60M	19.9s, 42M	84s, 90M
-left nest	6.4s, 60M	21.3s, 72M	24.5s, 48M	53.5s, 94M
Twelf	26.8s, 84M	[examples not in pure LF format]		
LEGO	930s, 51M	[examples not in pure LF format]		
example size	2.4M	2.2M	0.9M	4.6M

Fig. 3. Empirical results comparing optimizations

Without "intersection check", checking time varies rather widely, possibly because the intersection check ameliorates some of the inefficiency of performing substitution over a DAG without caching. The results of substituting cannot

[1] The example in pure LF is available on the web in Twelf and LEGO formats at http://verify.stanford.edu/~stump/dlx-pc.pure/. Also, the flea proof checker ships with CVC, which is freely available at http://verify.stanford.edu/CVC/.

be easily cached, because the substitution being applied changes; as explained in Section 5 above, using explicit substitutions would help with this problem. The asymptotically better classifier computation for left-nested applications is slower on half the benchmarks, possibly because the average depth of immediate nesting of Π-abstractions (less than 10 in these examples) is not great enough for the linear algorithm to overcome its constant overhead.

8 Conclusion

Several optimizations for LF type-checking have been presented. To formalize context-dependent caching, a path calculus for LF has been developed and proven equivalent to a standard calculus. Optimizations related to safe substitution and classifier computation for left-nested applications have also been presented. Further empirical work is needed to see which optimizations are most important for particular classes of represented proofs. The results of the paper should generalize to Pure Type Systems (PTSs), although it is not obvious how to extend caching to PTSs that are not singly sorted, since unicity of classifiers can fail (see Lemma 5.2.21 of [3]). For similar reasons, it is not clear to what extent context-dependent caching can be used in type-checking implicit LF [16] or systems of the Rho Cube [5].

9 Acknowledgements

This work was supported under ARPA/Air Force contract F33615-00-C-1693 and NSF contract CCR-9806889. We thank Nikolaj Bjørner and Iliano Cervesato for valuable discussion and criticism of this paper, and the anonymous reviewers for their helpful suggestions.

References

1. S. Abramsky, D. Gabbay, and T. Maibaum, editors. *Handbook of Logic in Computer Science*. Oxford University Press, 1992.
2. A. Appel and E. Felten. Proof-carrying authentication. In *6th ACM Conference on Computer and Communication Security*, 1999.
3. H. Barendregt. *Lambda Calculi with Types*, pages 117–309. Volume 2 of Abramsky et al. [1], 1992.
4. S. Berghofer and T. Nipkow. Proof terms for simply typed higher order logic . In *Theorem Proving in Higher Order Logics, 13th International Conference*, volume 1869 of *LNCS*, 2000.
5. H. Cirstea, C. Kirchner, and L. Liquori. The Rho Cube. In F. Honsell, editor, *Foundations of Software Science and Computation Structures (FOSSACS)*, 2001.
6. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available at http://www.grappa.univ-lille3.fr/tata, 1997.
7. T. Coquand. *An algorithm for testing conversion in Type Theory*, pages 255–79. In Huet and Plotkin [12], 1991.
8. A. Degtyarev and A. Voronkov. *The Inverse Method*, chapter IV. In Robinson and Voronkov [19], 2001.

9. W. Farmer and J. Guttman. A Set Theory with Support for Partial Functions. *Logica Studia*, 66(1):59–78, 2000.
10. R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.
11. R. Harper and F. Pfenning. On Equivalence and Canonical Forms in the LF Type Theory. Technical Report CMU-CS-00-148, Carnegie Mellon University, July 2000.
12. G. Huet and G. Plotkin, editors. *Logical Frameworks*. Cambridge University Press, 1991.
13. F. Kamareddine. Reviewing the classical and the de Bruijn notation for λ-calculus and pure type systems. *Logic and Computation*, 11(3):363–394.
14. Z. Luo and R. Pollack. LEGO Proof Development System: User's Manual. Technical Report ECS-LFCS-92-211, Edinburgh LFCS, 1992.
15. G. Necula. Proof-Carrying Code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, January 1997.
16. G. Necula and P. Lee. Efficient representation and validation of proofs. In *13th Annual IEEE Symposium on Logic in Computer Science*, pages 93–104, 1998.
17. F. Pfenning. *Logical Frameworks*, chapter XXI. In Robinson and Voronkov [19], 2001.
18. F. Pfenning and Carsten Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *16th International Conference on Automated Deduction*, 1999.
19. A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier and MIT Press, 2001.
20. A. Stump. *Checking Validities and Proofs with CVC and flea*. PhD thesis, Stanford University, 2002. In preparation: check http://verify.stanford.edu/~stump/ for a draft.
21. A. Stump, C. Barrett, and D. Dill. CVC: a Cooperating Validity Checker. In *14th International Conference on Computer-Aided Verification*, 2002.
22. R. Virga. *Higher-Order Rewriting with Dependent Types*. PhD thesis, Carnegie Mellon University, October 1999.
23. W. Wong. Validation of HOL Proofs by Proof Checking. *Formal Methods in System Design*, 14(2):193–212, 1999.