Eliminating redundancy in higher-order unification: a lightweight approach

Brigitte Pientka

School of Computer Science
McGill University
Montreal, Canada

Abstract. In this paper, we discuss a lightweight approach to eliminate the overhead due to implicit type arguments during higher-order unification of dependently-typed terms. First, we show that some implicit type information is uniquely determined, and can therefore be safely skipped during higher-order unification. Second, we discuss its impact in practice during type reconstruction and during proof search within the logical framework Twelf. Our experimental results show that implicit type arguments are numerous and large in size, but their impact on run-time is between 10% and 20%. On the other hand optimizations such as eliminating the occurs check are shown to be crucial to achieve significant performance improvements.

1 Introduction

In recent years, logical frameworks which support formalizing language specifications together with their meta-theory have been pervasively used in small and large-scale applications, from certifying code [1] to advocating a general infrastructure for formalizing language meta-theory and semantics [2]. In particular, the logical framework LF [6], based on the dependently typed lambda-calculus, and light-weight variants of it like LF_i [11] have played a major role in these applications. While the acceptance of logical framework technology has grown and they have matured, one of the most criticized points is concerned with the run-time performance. To address this problem, we concentrate in this paper on one of the most common operations in type reconstruction and proof search: higher-order unification. In prior work, we have proposed to optimize higher-order pattern unification by eliminating unnecessary occurs checks during proof search [16]. This optimization leads to significant performance improvements in many example applications. In this work, we consider a different optimization where we skip some redundant implicit type arguments during unification. Unlike our prior optimization which is restricted to proof search, skipping some redundant type arguments during unification is a general optimization and hence impacts not only the proof search performance, but also any other algorithm relying on unification such as type-reconstruction, coverage checking, termination checking etc. Our approach is light-weight in the sense that we do not translate or change our internal representation of terms and types. This has the advantage

that it can be seamlessly incorporated into the current implementation of the Twelf system [14] and it can be easily compared to other existing optimizations. Adapting this lightweight approach is not just a matter of practical engineering convenience, but a change to a different internal representation of terms impacts the foundation of LF itself and it remains unclear whether other algorithms such as mode, termination, and coverage checking would still remain correct.

Our work is motivated by Necula and Lee's [11] observation that the amount of implicit type arguments can be substantial when representing terms with a deep dependent type structure. Their main concern [11] however was in compactly representing terms in a fragment of LF called LF_i by allowing some implicit type arguments to be omitted. Reed [17] has proposed an extension of their idea to full LF. In contrast, we are interested in investigating the run-time overhead due to implicit type arguments during higher-order unification and its impact on type reconstruction and proof search in logical frameworks. In an early empirical study, Michaylov and Pfenning [7] have conjectured that the impact of redundant types during run-time may be significant. This paper investigates this question in theory and practice. The contributions of this paper are twofold: 1) We identify arguments which can be safely omitted during higher-order unification based on a static analysis of the declared type of constants. This information is then taken into account during run-time by looking up the relevant information for each constant and exploiting it when unifying its arguments. We justify this optimization theoretically using a contextual modal type theory. 2) We have implemented this optimization as part of the Twelf system [14], and discuss its impact in practice. Our experimental results show that although the size of redundant arguments is large and there is a substantial number of them, their impact on run-time performance is surprisingly limited (roughly 20% improvement). Our experimental results also demonstrate that optimizations such as eliminating the occurs checks are more important than previously thought. These results provide interesting insights into efficient implementations of dependently typed systems in general, and can provide guidance for future implementations.

The paper is organized as follows: First, we give an example to illustrate the idea, and present some background on dependent type theory and type checking algorithm. Our presentation follows ideas discussed in [10], where we have meta-variables are first-class. Next, we present a formal algorithm which identifies redundant type arguments and show the correctness of optimized higher-order pattern unification where redundant type arguments are skipped. Finally, we present experimental results and discuss related work.

2 Example: Translating natural deduction proofs

To illustrate the problem of redundant arguments in dependently typed systems, let us consider the following example, where we translate natural deduction proofs into Hilbert-style proofs. We only consider a subset containing rules for implication and universal quantification and provide an implementation within

the logical framework Twelf. Assuming `o` and `i` represent the type family for propositions and individuals respectively, we define implication and universal quantification as `impi:o->o->o.` and `all:(i->o)->o.` using higher-order abstract syntax. The judgment for natural deduction is then implemented via a type family `nd` which is indexed by propositions. Following the judgements-as-types principle, we define constants `impi` and `impe` whose type corresponds to introduction and elimination rules for implication and universal quantifiers.

```
nd: o -> type.
impi:(nd A -> nd B) -> nd (A imp B).  alli:({a:i}nd A a) -> nd (all [x] A x).
impe:nd (A imp B) -> nd A -> nd B.    alle:nd (all [x] A x) -> nd (A T).
```

The lambda-abstraction $\lambda x.M$ is denoted by `[x] M` and the dependent function type $\Pi x{:}A_1.A_2$ is represented as $\{\mathtt{x}{:}\mathtt{A}_1\}\mathtt{A}_2$. We typically use capital letters denote meta-variables (or schematic variables), while small letters denote ordinary bound variables and meta-variables are assumed to be implicitly quantified at the outside. For example, the type of `impi` is in fact `{A:o}{B:o}(nd A -> nd B) -> nd (A imp B)`. Following similar ideas, we define constants `k`, `s`, `mp` for the Hilbert-style formulation.

```
hil: o -> type.
k : hil (A imp B imp A).
s : hil ((A imp B imp C) imp (A imp B) imp A imp C).
mp: hil (A imp B) -> hil A -> hil B.
f1: hil ((all [x:i] A x) imp (A T)).
f2: hil ((all [x:i] (B imp A x)) imp (B imp all [x:i] A x)).
```

Next, we define the translation of natural deduction proofs into Hilbert-style proofs using the type family `hildn`. We refer the reader not familiar with representing derivations in the logical framework LF to [13].

```
hilnd  :hil A -> nd A -> type.
hnd_k :hilnd k (impi [u] impi [v] u).
hnd_s :hilnd s (impi [u] impi [v] impi [w] impe (impe u w) (impe v w)).
hnd_mp:hilnd H2 D2 -> hilnd H1 D1 -> hilnd (mp H1 H2) (impe D1 D2).
```

The code only reflects the explicit arguments describing the natural deduction and Hilbert-style derivations respectively. To illustrate, consider the second clause where we translate the Hilbert axiom `k` to a natural deduction derivation. The correctness of this translation hinges on the underlying dependent type structure and the translation `hilnd` takes in fact three arguments: the actual proposition `((A imp (B imp C)) imp ((A imp B) imp (A imp C)))` being considered, a constant `k` representing the Hilbert-style proof for `hil ((A imp (B imp C)) imp ((A imp B) imp (A imp C)))` and the natural deduction proof for `nd ((A imp (B imp C)) imp ( (A imp B) imp (A imp C)))`. Similarly, when we build the natural deduction derivation for `nd ((A imp (B imp C)) imp ( (A imp B) imp (A imp C)))`, we record how we instantiate

the implication introduction and elimination rules. This leads to the following explicit representation of the clause `hnd_s`, where we marked implicit type arguments `X` with brackets $\lfloor$`X`$\rfloor$.

```
hilnd ⌊((A imp (B imp C)) imp ((A imp B) imp (A imp C)))⌋ (s ⌊A⌋ ⌊B⌋ ⌊C⌋)
(impi ⌊(A imp (B imp C))⌋ ⌊((A imp B) imp (A imp C))⌋
 ([u] impi ⌊(A imp B)⌋ ⌊(A imp C)⌋
  ([v] impi ⌊A⌋ ⌊C⌋
    ([w] impe ⌊B⌋ ⌊C⌋ (impe ⌊A⌋ ⌊(B imp C)⌋ u w) (impe ⌊A⌋ ⌊B⌋ v w))))).
```

As one can see from this example, the overhead of implicit type argument can be substantial. When we execute this translation via a proof search interpretation, we must unify a goal with a given clause head. This will involve unifying not only its explicit arguments, but also all its implicit type arguments. In this paper, we investigate how much type computation can be eliminated during unification, and what its effect and impact is on run-time performance. We exploit and make precise a simple idea that some of the implicit type arguments in a term M are uniquely determined by the type of the object M. Since we only unify objects of the same type, it is safe for unification to skip over the arguments in M which are uniquely determined by its type.

3 Contextual modal type theory

In this section we present the foundation for logical frameworks. We will follow our development in [10] where we present a contextual modal dependent type theory with first-class meta-variables. The presentation exploits a recent presentation technique for logical frameworks due to Watkins et al. [18] in which only canonical forms are well-typed. The key idea underlying is to introduce hereditary substitutions which always yields terms in canonical form after the substitution has been applied. In the object calculus, we distinguish between *atomic objects* R and *normal objects* M. Meta-variables together with a mediating substitution σ are in this presentation first-class and denoted by $u[\sigma]$. They are declared in the modal context Δ and carry their own context of bound variables Ψ and type A. Note that the substitution σ is part of the syntax of meta-variables. This eliminates the need of pre-cooking [4] which raises existential variables to the correct context. This is a conservative extension of LF [6] so we suppress some routine details such as signatures. For a full extension of this fragment as a contextual dependent type theory we refer the reader to [10].

Normal Kinds	$K ::= \mathsf{type} \mid \Pi x{:}A.K$	Contexts	$\Gamma, \Psi ::= \cdot \mid \Gamma, x{:}A$
Atomic Types	$P, Q ::= a \cdot S$	Modal Contexts	$\Delta ::= \cdot \mid \Delta, u{::}A[\Psi]$
Normal Types	$A, B ::= P \mid \Pi x{:}A.B$	Substitutions	$\sigma ::= \cdot \mid \sigma, M/x$
Atomic Objects	$R ::= x \cdot S \mid c \cdot S \mid u[\sigma] \cdot S$	Modal subst.	$\theta ::= \cdot \mid \theta, M/u$
Normal Objects	$M, N ::= \lambda x.\, M \mid R$		
Spines	$S ::= \mathsf{nil} \mid M; S$		

Typing at the level of objects is divided into three judgments:

$\Delta; \Gamma \vdash M \Leftarrow A$ Check object M against canonical A
$\Delta; \Gamma \vdash R \Rightarrow A$ Synthesize canonical A for atomic object R
$\Delta; \Gamma \vdash S : A \Rightarrow P$ Synthesize a canonical P by checking a spine S against type A
$\Delta; \Gamma \vdash \sigma \Leftarrow \Psi$ Check substitution σ against context Ψ

The central idea is based on two observations: First, we can characterize canonical forms via bi-directional type-checking. Second, we can formalize normalization as a primitive recursive functional exploiting the structure of types and objects. The key idea is to replace ordinary substitution operation with one which will always yield a canonical term, i.e. in places where the ordinary substitution operation would create a redex, we must make sure to normalize during substitutions. In particular, when applying the substitution $[M/x]$ to a term $x \cdot S$, we must apply the substitution $[M/x]$ to the spine S, but we also must reduce the redex $(M \cdot [M/x]S)$ which would be created, since it is not meaningful in the defined setting. Since when applying $[M/x]$ to the spine S, we again may encounter situations which require us to contract a redex, the substitution $[M/x]$ must be hereditary. We therefore call this operation *hereditary substitution*. Before we discuss this substitution operation further, let us first present the bi-directional type checking rules.

Check normal object M against type

$$\frac{\Delta; \Gamma, x{:}A \vdash M \Leftarrow B}{\Delta; \Gamma \vdash \lambda x.\, M \Leftarrow \Pi x{:}A.B} \qquad \frac{\Delta; \Gamma \vdash R \Rightarrow P \quad P = P'}{\Delta; \Gamma \vdash R \Leftarrow P}$$

Synthesize atomic type P for atomic object R

$$\frac{\Delta; \Gamma \vdash S : A \Rightarrow P \quad \Sigma(c) = A}{\Delta; \Gamma \vdash c \cdot S \Rightarrow P} \qquad \frac{\Delta; \Gamma \vdash S : A \Rightarrow P \quad \Gamma(x) = A}{\Delta; \Gamma \vdash x \cdot S \Rightarrow P}$$

$$\frac{(\Delta_1, u{::}A[\Psi], \Delta_2); \Gamma \vdash \sigma \Leftarrow \Psi \quad (\Delta_1, u{::}A[\Psi], \Delta_2); \Gamma \vdash S : [\sigma]^a_\Psi A \Rightarrow P}{(\Delta_1, u{::}A[\Psi], \Delta_2); \Gamma \vdash u[\sigma] \cdot S \Rightarrow P}$$

Synthesize atomic type P from spine S and type A

$$\frac{}{\Delta; \Gamma \vdash \mathsf{nil} : P \Rightarrow P} \qquad \frac{\Delta; \Gamma \vdash M \Leftarrow A \quad \Delta; \Gamma \vdash S : [M/x]^a_A(B) \Rightarrow P}{\Delta; \Gamma \vdash (M; S) : \Pi x{:}A.B \Rightarrow P}$$

Next we will describe the hereditary substitution operation. As we mentioned above, we must carefully design it such that it also ensures that the result of applying a substitution σ to a canonical object M yields again a canonical object. We define hereditary substitutions as a primitive recursive functional where we pass in the type of the variable we substitute for. This will be crucial in determining termination of the overall substitution operation. If we hereditarily substitute $[\lambda y.M/x](x \cdot S)$, then if everything is well-typed, $x : A_1 \to A_2$ for some A_1 and A_2 and we will write $[\lambda y.M/x]_{A_1 \to A_2}(x \cdot S)$ indexing the substitution with the

type for x. These will all be total operations since any side condition can be satisfied by α-conversion. It is worth pointing out that it suffices for the type annotation A of the substitution $[M/x]_A$ to be "approximately" correct[1]

The definition for $[R/x]^n_P(M)$ is straightforward, and we omit it here, since no redices will be produced. Instead we concentrate on the ordinary substitution $[M/x]^n_A(N)$ where potentially redices are created. We define substitution as a primitive recursive functional $[M/x]^n_A(N)$, $[M/x]^r_A(R)$, $[M/x]^l_A(S)$, and$[M/x]^s_A(\sigma)$.

$$
\begin{array}{lll}
[M/x]^n_A(\lambda y.\, N) & = \lambda y.\, N' & \text{where } N' = [M/x]^n_A N,\ y \notin \mathsf{FV}(M) \text{ and } y \neq x \\
[M/x]^n_A(R) & = R' & \text{where } R' = [M/a]^r_A R \\
 & & \\
[M/x]^r_A(c \cdot S) & = c \cdot S' & \text{where } S' = [M/x]^l_A S \\
[M/x]^r_A(x \cdot S) & = R & \text{where } S' = [M/x]^l_A S \text{ and } R = \mathsf{reduce}(M : A, S') \\
[M/x]^r_A(y \cdot S) & = y \cdot S' & \text{if } y \neq x \text{ and } S' = [M/x]^l_A S \\
[M/x]^n_A(u[\sigma] \cdot S) & = u[\sigma'] \cdot S' & \text{where } \sigma' = [M/x]^s_A \sigma \text{ and } S' = [M/x]^l_A \\
 & & \\
[M/x]^l_A(\mathsf{nil}) & = \mathsf{nil} & \\
[M/x]^l_A(N; S) & = N'; S' & \text{where } N' = [M/x]^n_A N \text{ and } S' = [M/x]^l_A S \\
 & & \\
[M/x]^s_A(\cdot) & = \cdot & \\
[M/x]^s_A(\sigma, N/y) & = (\sigma', N'/y) & \text{where } \sigma' = [M/x]^s_A \sigma \text{ and } N' = [M/x]^n_A N
\end{array}
$$

The interesting case is when we substitute a term M for a variable x in the term $x \cdot S$. As we outlined above, we need to possibly reduce the resulting redex to maintain canonical forms. Hence we define the $\mathsf{reduce}(M : A, S) = R$ next.

$$
\begin{array}{lll}
\mathsf{reduce}(\lambda y.M : \Pi x{:}A_1.A_2, (N; S)) = M'' & \text{where } [N/y]^n_{A_1} M = M' \\
 & \quad \text{and } \mathsf{reduce}(M' : A_2, S) = M'' \\
\mathsf{reduce}(R : P, \mathsf{nil}) & = R & \\
\mathsf{reduce}(M : A, S) & \text{fails} & \text{otherwise}
\end{array}
$$

Substitution may fail to be defined only if substitutions into the subterms are undefined. The side conditions $y \notin \mathsf{FV}(M)$ and $y \neq x$ do not cause failure, because they can always be satisfied by appropriately renaming y. However, substitution may be undefined if we try for example to substitute an atomic term R for x in the term $x \cdot S$ where the spine S is non-empty. Similarly, the reduce operation is undefined. The substitution operation is well-founded since recursive appeals to the substitution operation take place on smaller terms with equal type A, or the substitution operates on smaller types (see the case for $\mathsf{reduce}(\lambda y.M : A_1 \to A_2, (N; S))$).

[1] In [10] we define approximate types as dependent types where dependencies have been erased. This is not necessary for the correctness of substitution, but it clarifies the role of the type annotation of substitutions.

Hereditary substitution operations terminate, independently of whether the terms involved are well-typed or not. The operation may fail, in particular if we have ill-typed terms, or yield a canonical term as a result.

Theorem 1 (Substitution on Terms).

1. *If* $\Delta; \Gamma \vdash M \Leftarrow A$ *and* $\Delta; \Gamma, x{:}A, \Gamma' \vdash N \Leftarrow B$ *and* $[M/x]^n_A N = N'$, $[M/x]^a_A B = B'$ *and* $[M/x]^g_A(\Gamma') = \Gamma''$ *then* $\Delta; \Gamma, \Gamma'' \vdash N' \Leftarrow B'$.
2. *If* $\Delta; \Gamma \vdash M \Leftarrow A$ *and* $\Delta; \Gamma, x{:}A, \Gamma' \vdash R \Rightarrow P$ *and* $R' = [M/x]^r_A R$, $[M/x]^a_A P = P'$ *and* $[M/x]^g_A(\Gamma') = \Gamma''$ *then* $\Delta; \Gamma, \Gamma' \vdash R' \Rightarrow P'$.

Substitutions for meta-variables u are a little more difficult. Recall that meta-variables u are always associated with a postponed substitution and $u[\sigma]$ forms a closure. As soon as we know which term u stands for we can apply σ to it.

Moreover, because of α-conversion, the ordinary variables occurring in the term M being substituted and the domain of σ may be different. As a result, substitution for a meta-variable must carry a context, written as $[\Psi.M/u]N$ and $[\Psi.M/u]\sigma$ where Ψ binds all free variables in M. This complication can be eliminated in an implementation of our calculus based on de Bruijn indexes.

Finally, just as with ordinary substitutions we must be careful to only construct canonical terms. In particular, when we substitute $\lambda x.\, M$ into the term $u[\sigma] \cdot S$ the resulting term $[\sigma]M \cdot S$ is not in canonical form. Hence similar to ordinary hereditary substitutions we will define a primitive recursive functional for contextual substitutions which is indexed by the type of the contextual variable and ensure that the result is always in normal form.

$$
\begin{array}{lll}
[\![\Psi.M/u]\!]^n_{A[\Psi]}(\lambda y.\, N) & = \lambda y.\, N' & \text{where } N' = [\![\Psi.M/u]\!]^n_{A[\Psi]} N \\
[\![\Psi.M/u]\!]^n_{A[\Psi]}(R) & = R' & \\
[\![\Psi.M/u]\!]^r_{A[\Psi]}(c \cdot S) & = c \cdot S' & \text{where } S' = [\![\Psi.M/u]\!]^r_{A[\Psi]} S \\
[\![\Psi.M/u]\!]^r_{A[\Psi]}(x \cdot S) & = x \cdot S' & \text{where } S' = [\![\Psi.M/u]\!]^r_{A[\Psi]} S \\
[\![\Psi.M/u]\!]^r_{A[\Psi]}(u[\tau] \cdot S) & = R & \text{where } \tau' = [\![\Psi.M/u]\!]^s_{A[\Psi]}(\tau),\ M' = [\tau'/\Psi]^n_\Psi(M) \\
 & & \quad S' = [\![\Psi.M/u]\!]^l_{A[\Psi]}(S) \text{ and } \mathsf{reduce}(M' : A, S') = R \\
[\![\Psi.M/u]\!]^r_{A[\Psi]}(v[\tau] \cdot S) & = v[\tau'] \cdot S' & \text{where } v \neq u,\ \tau' = [\![\Psi.M/u]\!]^s_{A[\Psi]}\tau \\
 & & \quad \text{and } S' = [\![\Psi.M/u]\!]^l_{A[\Psi]} S \\
[\![\Psi.M/u]\!]^l_{A[\Psi]}(\mathsf{nil}) & = \mathsf{nil} & \\
[\![\Psi.M/u]\!]^l_{A[\Psi]}(N; S) & = N'; S' & \text{where } N' = [\![\Psi.M/u]\!]^n_{A[\Psi]} N \\
 & & \quad \text{and } S' = [\![\Psi.M/u]\!]^l_{A[\Psi]} S \\
[\![\Psi.M/u]\!]^s_{A[\Psi]}(\cdot) & = \cdot & \\
[\![\Psi.M/u]\!]^s_{A[\Psi]}(\tau, N/y) & = \tau', N'/y & \text{where } \tau' = [\![\Psi.M/u]\!]^s_{A[\Psi]}\tau \\
 & & \quad \text{and } N' = [\![\Psi.M/u]\!]^n_{A[\Psi]} N
\end{array}
$$

Applying $[\![\Psi.M/u]\!]$ to the term $u[\tau] \cdot S$ will first apply $[\![\Psi.M/u]\!]$ to the closure $u[\tau]$. This will yield the simultaneous substitution $\tau' = [\![\Psi.M/u]\!]\tau$, but instead

of returning $M[\tau']$, it proceeds to eagerly apply τ' to M. Before τ' can be carried out, however, it's domain must be renamed to match the variables in Ψ, denoted by τ'/Ψ. In addition, the substitution $[\![\Psi.M/u]\!]$ must be applied to the spine S yielding S'. Since the result $M' \cdot S'$ may not be canonical, we again must call the reduce operation. Contextual substitutions are compositional, and contextual substitution properties hold. We only show the one for normal terms but the other can be stated similarly.

Theorem 2 (Contextual Substitution on Terms).
If $\Delta; \Psi \vdash M : A$ *and* $(\Delta, u{::}A[\Psi], \Delta'); \Gamma \vdash N : B$ *and*
$[\![\Psi.M/u]\!]^n_{A[\Psi]} N = N'$, $[\![\Psi.M/u]\!]^a_{A[\Psi]} B = B'$, *and* $[\![\Psi.M/u]\!]^g_{A[\Psi]} \Gamma = \Gamma'$
then $(\Delta, \Delta'); \Gamma' \vdash N' : B'$.

Remark 1. Although our theory allows for meta-variables $u[\sigma] \cdot S$, it is often convenient to require that meta-variables are lowered and their spine S is therefore empty. This optimization is based on the observation that meta-variables $u{::}(\Pi x{:}A_1.A_2)[\Psi]$ must always be instantiated with λ-abstractions, because λ-abstractions are the only canonical objects of function type. We can therefore anticipate part of the structure of the instantiation of u and create a new variable $u'{::}A_2[\Psi, x{:}A_1]$. Note that u' has a simpler type, although a longer context. In this way we can always lower existential variables until they have atomic type, $v{::}P[\Psi]$. As a consequence, the only occurrences of meta-variables are as $u[\sigma] \cdot \mathsf{nil}$ and we often abbreviate this term simply by writing $u[\sigma]$. Finally it is worth pointing out that any instantiation of u must be an atomic object R, and applying a substitution $[\![\Psi.R/u]\!]$ to a term M will always directly yield a canonical object.

Remark 2. Often it is convenient to refer to the pattern fragment [8, 12]. We call a normal term M an *atomic pattern*, if all the subterms of the form $u[\sigma] \cdot \mathsf{nil}$ are such that $u{::}Q[\Psi]$ and $\sigma = y_1/x_1, \ldots y_k/x_k$ where $y_1, \ldots, y_k$ are distinct bound variables. This is already implicitly assumed for $x_1, \ldots, x_k$ because all variables defined by a substitution must be distinct. Such a substitution is called a *pattern substitution*. In addition, the type of any occurrence of $u[\sigma]$ is atomic and we will write Q for atomic types. Finally, we can show that pattern substitutions and contextual substitutions commute [15].

To illustrate the use of meta-variables and ordinary variables, let us briefly reconsider the previous example of translating natural deduction proofs to proofs in Hilbert-style. Recall that bound variable dependencies are crucial when defining in `f2:hil ((all [x:i] (B imp A x)) imp (B imp all [x:i] A x))`, since we are only allowed to move the universal quantifier inside an implication if the formula `B` does not depend on the bound variable `x`. Our contextual modal type theory, will give us an elegant way of distinguishing between meta-variables and ordinary variables, and describing possible dependencies between them. We can represent these clauses as follows: `A`, `B`, and `T` are meta-variables and will be represented by contextual variables `u`, `v`, `t`.

```
f1: hil ((all λx.u[x/x']) imp (u[t[·]/x'])).
f2: hil ((all λx.(v[·] imp u[x/x'])) imp (v[·] imp (all λx.u[x/x']))).
```

Note that meta-variables `u` and `v` are associated with a substitution which precisely characterizes their dependencies. Since v cannot depend on the bound variable `x`, it is associated with the empty substitution. The instantiation for meta-variable `u` on the other hand may refer to the bound variable `x`, which is characterized by the associated substitution `[x/x']`. In this example the type of the constant `f1` is not a higher-order pattern since it contains a subterm `u[t[·]/x']` where the substitution associated with the modal variable `u` is not a pattern substitution. On the other hand the type of the constants `f2` is a higher-order pattern, since both meta-variable occurrences are patterns.

4 Synthesizing spine arguments from types

As we saw in the previous section, typing proceeds in a bi-directional way where the type of atomic objects is synthesized. The central idea behind bi-directional type-checking is to distinguish between objects whose type is uniquely determined and hence can be synthesized and objects whose type we already know and we may not be able to uniquely determine from the surrounding information and hence need to be checked against a given type. In this section, we will take a slightly different view. In particular, we ask what information in the object is uniquely determined, if we know its type. For example in the rule for the object $c \cdot S$ we always synthesize the type P, by first looking up the type A of the constant c and then inferring P from the spine S and the type A. Switching perspectives we ask what information in the spine S can be synthesized if we know the type A and its final target type P. In other words, we will think of the object $c \cdot S$ as a normal object and we can check it against a given type P. Intuitively, we can always recover argument M_i occurring in a spine S if $[M_i/x_i]P$ is injective in the argument M_i. Therefore some information already present in P is duplicated in S. Hence we will target the rule for checking a term $c \cdot S$ and the rules for type checking spines. In particular, we will introduce a new judgment which says that we can synthesize a substitution θ by checking a canonical P against the type A and a spine S:

$$\Delta; \Theta; \Gamma \vdash S : A \Leftarrow P/\theta$$

The context Θ characterizes the holes in the type A which can be uniquely inferred from target type P. θ is a contextual substitution with domain Θ s.t. $[\![\theta]\!]A$. Holes are described by meta-variables and we will ensure that only A can refer to these meta-variables characterized by Θ. Hence we will replace the rule for synthesizing an atomic type P for $c \cdot S$ with the following rule which checks $c \cdot S$ against an atomic type P.

$$\frac{\Delta; \cdot; \Gamma \vdash S : A \Leftarrow P/\cdot \quad \Sigma(c) = A}{\Delta; \Gamma \vdash c \cdot S \Leftarrow P}$$

Next, we show the rules for synthesizing a substitution θ by checking the spine S and the type A against the atomic type P. Assume $S = M_1; \ldots; M_n; \mathsf{nil}$, $A = \Pi x_1{:}A_1 \ldots \Pi x_n{:}A_n.P'$, and atomic type P. Then for every x_i where x_i occurs strict in P', we can retrieve M_i from P. θ will exactly keep track of those M_i which we can synthesize from P. When we encounter $\Pi x_i{:}A_i.B$ where x_i is strict in the target type of B, we will introduce a fresh meta-variable u which will later be instantiated by higher-order pattern matching[2]. Note that the criteria of x_i being strict in P is crucial because only for those x_i can we ensure that higher-order pattern matching will find a unique instantiation.

$$\frac{\Delta; \Theta; \Gamma \vdash P \doteq P'/\theta}{\Delta; \Theta; \Gamma \vdash \mathsf{nil} : P' \Leftarrow P/\theta}$$

$$\frac{\Delta; \Theta; \Gamma \vdash \ \mathsf{strict}\ (x, B) \quad \Delta; \Theta, u{::}A[\Gamma]; \Gamma \vdash S : [u[\mathsf{id}_\Gamma]/x]^a_A(B) \Leftarrow P/(\theta, \Gamma.M/u)}{\Delta; \Theta; \Gamma \vdash (M; S) : \Pi x{:}A.B \Leftarrow P/\theta}$$

$$\frac{\Delta; \Theta; \Gamma \not\vdash (\ \mathsf{strict}\ (x, B) \quad \Delta; \Gamma \vdash M \Leftarrow [\![\theta]\!]^a_\Theta(A) \quad \Delta; \Theta; \Gamma \vdash S : [M/x]^a_A(B) \Leftarrow P/\theta}{\Delta; \Theta; \Gamma \vdash (M; S) : \Pi x{:}A.B \Leftarrow P/\theta}$$

If u occurs strict in the target type of $[u[\mathsf{id}_\Gamma]/x]^a_A(B)$ and u occurs as a higher-order pattern, then we can always reconstruct M, the information which is present in the spine $(M; S)$. Otherwise, we will continue to type check the spine S and infer an instantiation for all the meta-variables occurring in Θ. Next, we can show that the new type-checking algorithm which skips over some elements is correct. The crucial lemma needed is the following:

Lemma 1.

1. *If* $\Delta; \Theta; \Gamma \vdash S : B \Leftarrow P/\theta$ *then* $\Delta; \Gamma \vdash S : [\![\theta]\!]^a_\Theta B \Rightarrow P$
2. *If* $\Delta; \Gamma \vdash S : [\![\theta]\!]^a_\Theta B \Rightarrow P$ *then* $\Delta; \Theta; \Gamma \vdash S : B \Leftarrow P/\theta$.

The type A of a constant c determines therefore which arguments in the spine of the term $c \cdot S$ can be omitted. Therefore, we generate a simple binary recipe from the type A which is associated with the constant c. Let $A = \Pi x_1{:}A_1 \ldots x_n{:}A_n.P$. If x_i occurs strict as a higher-order pattern in P, then we record 0 at the i-th position in the recipe b. If x_i does not occur strict as a higher-order pattern in P, then we record 1 at the i-th position in the recipe b. Consider the constant `f1:hil ((all λx.u[x/x']) imp (u[t[·]/x']))`. The type has one occurrence of the meta-variable u which is not a pattern, namely `u[t[·]/x']`. As a consequence, every time we encounter a term of type P with head `f1`, we must also consider the instantiations for `u` and `t`, because their instantiation cannot be uniquely determined from the type P. The recipe associated with `f1` is therefore `11`. On the other hand the type of constant `f2` only

[2] *Technically we should say* $u[\mathsf{id}] \cdot \mathsf{nil}$ *instead of just* $u[\mathsf{id}]$

contains occurrences of the meta-variable `u` which are higher-order patterns. If we encounter a term of type P with head `f2`, we can uniquely recover the instantiations for `u` and `v`. The recipe associated with `f2` will be `00`.

We will then modify higher-order unification as follows: when we are unifying two terms $c \cdot S$ and $c \cdot S'$, we will first lookup the recipe b associated with c, and then unify the spines S and S' taking into account the recipe b. If the i-th position in the recipe lists 0 then the i-th position in the spine S and S' will be skipped. If the i-th position in the recipe lists 1 then the i-th position in the spine S and S' must be unified. Crucial to the correctness of this optimization is the fact that the synthesized modal substitution θ is uniquely determined by the target type P and the type of the spine A.

Lemma 2. *If* $\Delta; \Theta; \Gamma \vdash S : A \Leftarrow P/\theta$ *and* $\Delta; \Theta; \Gamma \vdash S' : A \Leftarrow P/\theta'$ *then* $\theta = \theta'$.

Therefore, we already know that the spine S and S' agree on some of its arguments, and those arguments must not to be unified..

5 Experimental evaluation

The optimization of skipping some redundant type arguments during higher-order unification is a general optimization which can affect any algorithm relying on it. In this section, we discuss the impact of unifying redundant type arguments during proof search and report on our experience in type reconstruction. All experiments are done on a machine with the following specifications: 3.40GHz Intel Pentium Processor, 1024 MB RAM. We are using SML of New Jersey 110.55 under the Linux distribution Gentoo 16.14. Times are measured in seconds. For the timing analysis, we have done five runs, and we report on the average over these runs as well as the standard deviation observed.

In this discussion, we will present our measures on run-time and run-time improvement. In addition we also present quantitative evaluation by reporting on how many redundant type arguments were omitted, the average and maximum size of the omitted arguments.

5.1 Proof search

Unification is a central operation during proof search and its performance directly impacts the overall run-time performance. In previous work [16], we investigated optimizing higher-order pattern unification by linearizing terms and delaying the occurs check together with other expensive checks concerning bound variable occurrences. This optimization is called linear head compilation, since the head of a logic programming clause is translated into a linear term and constraints during compilation. This optimization can only be exploited during proof search since it relies on the fact that the meta-variables in the head of a clause and the meta-variables in the query are distinct. Here, we will first compare the impact of eliminating the need to unify redundant type arguments when no optimization of unification is done. Next, we will compare it to linear

head compilation, and finally we will report results of combining linear head compilation with eliminating the need to unify redundant type arguments.

Labeling of table: The first column NO describes the runtime in seconds when no optimization is done to unification. The second column TE describes the runtime when we skip redundant type arguments. LH describes the time using linear head compilation, and TELH gives the time when we combine linear head compilation with skipping redundant type arguments. The column #op refers to the number of skipped arguments during unification, and the column Av(size) refers to the average size of the omitted arguments.

Compiler translations for MiniML We consider some examples from compiler verification (see [5]). When given an evaluation of some programs using a big-step semantics, we translate this evaluation to a sequence of transitions on an abstract machine and vice versa. The implicit type arguments denote the actual program being evaluated, and hence depending on the size of the program, this may be large. The standard deviation on the reported examples was less than 1%.

The first set of examples use a continuation based machine, and the example programs being translated are simple programs involving multiplication, addition, square, and minus.

CPM – Proof search

	NO	TE	LH	TELH	no-te	no-lh	lh-telh	#op	Av(size)
addMin1	134.61	133.32	12.26	12.21	1%	91%	0%	829	31.02
square3a	489.70	478.02	89.02	81.63	2%	82%	8%	1182	33.40
square4b	779.55	766.23	153.62	121.73	2%	80%	21%	488	28.20
squmin3a	435.52	423.91	69.92	62.10	3%	84%	11%	1128	33.74
squmin4a	743.83	622.08	140.03	130.21	16%	81%	7%	1496	30.75

Unifying redundant type arguments has limited impact on the overall performance compared to no optimization. The last example shows an improvement by 16%. This can be substantial if we consider the absolute runtime. However, in many cases, the improvement is almost negligible given the standard deviation of 1%. The results clearly demonstrate that linear head optimization is crucial to achieve good performance. Redundant type elimination combined with linear head compilation, can give an additional improvement between 0% and 21%. In our examples many redundant type arguments were skipped (up to 1496), the average size of the skipped argument was around 30 constructors, and the maximum size of argument skipped was 185. Given this set-up, we expected a much stronger impact on run-time performance. We believe that the reason for the limited impact is that at the time when we need to unify redundant type arguments they are syntactically equal. This means it is very cheap to unify two arguments which are already syntactically equal.

The second set of examples use a CLS machine, a variant of the SECD abstract machine. Examples are similar to the CPM machine involving programs

with addition, multiplication, square and minus. We are mainly interested translating evaluations of terms to reductions of their de Bruijn representation. Since de Bruijn representations can be very large in size, our examples exhibit very large redundant arguments. On average the size of omitted arguments was up to 549.38, and the maximum size of omitted argument was 75218. In the examples considered there were also a substantial number of omitted terms (up to 6219). If the time limit of 3h had been exceeded and no solution was found, this is indicated by – in the table below

CLS – Proof search

	NO	TE	LH	TELH	no-te	no-lh	lh-telh	#op	Av(size)	max(size)
cls01	4044.18	3821.47	1.94	1.72	5.51%	99.95%	11.48%	1875	121.90	240
cls02	–	–	30.29	23.67	–	–	21.87%	1852	214.30	596
cls03	4450.01	4417.17	1.35	1.20	0.74%	99.97%	11.19%	1741	121.90	240
cls04	–	–	2.87	2.64	–	–	8.04%	2378	114.03	358
cls05	–	–	482.61	413.28	–	–	14.36%	6219	549.38	75218

Again there is almost no impact of skipping redundant arguments when we compare it to no optimization at all. Linear head optimization is however crucial to execute some examples. If we combine skipping over redundant type arguments with linear head optimization, we can see an additional improvement between 8% and 22%.

Translating classical proofs into cut-free proofs The next few examples exploit a translation of proofs in classical logic into cut-free sequent calculi proofs. Relative standard deviation was up to 2.7%.

Cut-elimination – Proof search producing cut-free proofs

	NO	TE	LH	TELH	no-te	no-lh	lh-telh	#op	Av(size)
ndcf01	13.26	13.16	6.39	6.22	1%	52%	3%	97687	1.305
ndcf02	30.24	29.65	17.16	15.45	2%	43%	10%	264387	7.15

Due to space constraints, we only show here two significant examples. Although there may be many redundant type arguments (up to 264,1433) their size may not be very large, and their impact on runtime behavior is limited.

Summary of results To summarize the results, redundant type arguments occur often as we can see by the large number of arguments skipped and they are substantial in size. Their impact on runtime in the current Twelf implementation ranges up to 20% especially if we combine it with linear head compilation. This seems counter-intuitive. To our surprise optimizations such as linear head compilation are in fact crucial to overcome some performance barriers and several examples were not executable without this optimization. Although we had noticed the importance of linear head compilation in simply typed examples, we were surprised that this optimization was also crucial for examples with a deeply nested dependent type structure.

5.2 Type reconstruction

Higher-order unification does not only play an important role during proof search, but is used in many algorithms, such as type reconstruction, termination, and totality checking. Skipping redundant type arguments during unification therefore impacts these algorithms.

We have experimented with a wide variety of type reconstruction examples, from Kolmogoroff proof translation, Hilbert-style proof translations, typed assembly language [3]. Similar to proof search, we observe that although there is quite a large number of redundant implicit type arguments (< 1488) in some of these test-suites), the impact on the performance during type reconstruction is up to 13% on our examples. In the CPM compilation examples for example, we observe between 2% and 13% runtime improvement, The maximum size of omitted argument was 159, and the average size was between 13.37 and 32.33.

6 Related Work

J. Reed [17] investigates a bi-directional type checking algorithm for the logical framework LF where some implicit type arguments can be omitted. The motivation for his work is to compactly represent proofs and check them. However the price is a complicated meta-theory, and a different dependently type lambda calculus where some terms are explicitly annotated with types. The motivation of his work lies in generalizing Necula and Lee's work on compact proof representations to the full power of dependent types. If we would want to adopt his approach as a foundation for optimizing unification, proof search and type-reconstruction, we would need to abandon our current representation of terms. This could be not only a bothersome and daunting engineering task, but it is also not clear whether other algorithms such as mode, termination, and coverage checking would still continue to work on this dependently typed variant of LF. In contrast, we propose a lightweight approach which does not impact our foundations itself, but can be employed locally to optimize unification.

Although the problem of index arguments is due to the dependent type structure of LF, a similar problem arises in λProlog [9] due to polymorphism. A similar criteria as the one described in this paper, has been exploited by Nadathur and Qi [9] in recent work on optimizing λProlog. They explore optimizations such as eliminating typing annotations at lambda-labels and some implicit type arguments due to polymorphism within the WAM for λProlog. However, their proposal does not provide a high-level justification for this optimization and no experimental comparison or discussion is given how much impact these optimizations have in practice.

7 Conclusion

We have presented a lightweight approach to eliminate overhead of redundant type information during unification. We have presented a clean theoretical foundation for it based on contextual modal types, and evaluated the impact of

redundant type arguments in practice. Although redundant arguments arise frequently and they are large in size, their impact on run-time is in the current Twelf implementation between 10% and 20%. This may seem surprising at first, since it is commonly believed that redundant type arguments can yield up to a factor of two improvement. Our experimental results however seem to indicate that unifying two arguments which are syntactically the same is very cheap. A few interesting lessons we have learned from this work is that we have underestimated the impact of linearization and delaying the occurs check during unification and proof search. Linearization allows for quick failure, and more importantly reduces the overhead of trailing during runtime, which can substantially improve the performance. As our results indicate, the size of omitted arguments is in fact substantial, and reducing the overall size of terms may yield better run-time performance, since most computation seems to be memory bound. This seems to suggest that generating compact representations of terms for proof search may be desirable and may yield a substantial run-time improvement. On the other hand, choosing a different more compact representation of terms as a basis of the implementation could be a bothersome and daunting engineering task. In addition, it is also not clear whether other algorithms such as mode, termination, and coverage checking would still continue to work on this dependently typed variant of LF. Nevertheless, it may be worthwhile to consider a compact term representation if a system is newly designed and is specialized towards proof search.

There are several possible improvements which can be made. First, we could take into account mode information to omit more type arguments during proof search. However, the experimental results seem to indicate that the additional performance improvement gained by omitting a few more implicit type arguments may not be worth the effort.

Acknowledgments: Thanks to Jason Reed for many discussions related to this topic.

References

1. Andrew Appel. Foundational proof-carrying code. In J. Halpern, editor, *Proceedings of the 16th Annual Symposium on Logic in Computer Science (LICS'01)*, pages 247–256. IEEE Computer Society Press, June 2001. Invited Talk.
2. B. Aydemir, A. Bohannon, M. Fairbairn, J. Foster, B. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The poplmark challenge, 2005.
3. Karl Crary and Susmit Sarkar. Foundational certified code in a metalogical framework. In *19th International Conference on Automated Deduction*, Miami, Florida, USA, 2003. Extended version published as CMU technical report CMU-CS-03-108.
4. Gilles Dowek, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 259–273, Bonn, Germany, September 1996. MIT Press.

5. John Hannan and Frank Pfenning. Compiler verification in LF. In Andre Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, June 1992.
6. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.
7. Spiro Michaylov and Frank Pfenning. An empirical study of the runtime behavior of higher-order logic programs. In D. Miller, editor, *Proceedings of the Workshop on the λProlog Programming Language*, pages 257–271, Philadelphia, Pennsylvania, July 1992. University of Pennsylvania. Available as Technical Report MS-CIS-92-86.
8. Dale Miller. Unification of simply typed lambda-terms as logic programming. In *Eighth International Logic Programming Conference*, pages 255–269, Paris, France, June 1991. MIT Press.
9. Gopalan Nadathur and Xiaochu Qi. Optimizing the runtime processing of types in polymorphic logic programming languages. In Geoff Sutcliffe and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005, Montego Bay, Jamaica, December 2-6, 2005, Proceedings*, volume 3835 of *Lecture Notes in Computer Science*, pages 110–124. Springer, 2005.
10. Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. A contextual modal type theory. 2005.
11. George C. Necula and Peter Lee. Efficient representation and validation of logical proofs. In Vaughan Pratt, editor, *Proceedings of the 13th Annual Symposium on Logic in Computer Science (LICS'98)*, pages 93–104, Indianapolis, Indiana, June 1998. IEEE Computer Society Press.
12. Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.
13. Frank Pfenning. Logical frameworks. In A. Robinson and A. Voronkov, editors, *Handbook of automated reasoning*, pages 1063–1147, Amsterdam, The Netherlands, The Netherlands, 2001. Elsevier Science Publishers B. V.
14. Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag Lecture Notes in Artificial Intelligence (LNAI) 1632.
15. Brigitte Pientka. *Tabled higher-order logic programming.* PhD thesis, Department of Computer Sciences, Carnegie Mellon University, December 2003. CMU-CS-03-185.
16. Brigitte Pientka and Frank Pfennning. Optimizing higher-order pattern unification. In F. Baader, editor, *19th International Conference on Automated Deduction, Miami, USA*, Lecture Notes in Artificial Intelligence (LNAI) 2741, pages 473–487. Springer-Verlag, July 2003.
17. Jason Reed. Redundancy Elimination for LF. In Carsten Schuermann, editor, *Fourth Workshop on Logical Frameworks and Meta-languages — LFM'04*, Cork, Ireland, 5 July 2004.
18. Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002. Forthcoming.