# Optimizing Higher-Order Pattern Unification

Brigitte Pientka and Frank Pfenning*

Department of Computer Science
Carnegie Mellon University
bp@cs.cmu.edu,fp@cs.cmu.edu

Submitted, January 2003

**Abstract.** We present an abstract view of existential variables in a dependently typed lambda-calculus based on modal type theory. This allows us to justify optimizations to pattern unification such as linearization, which eliminates many unnecessary occurs-checks. The presented modal framework explains a number of features of the current implementation of higher-order unification in Twelf and provides insight into several optimizations. Experimental results demonstrate significant performance improvement in many example applications of Twelf, including those in the area of proof-carrying code.

## 1 Introduction

Unification lies at the heart of automated reasoning systems, logic programming and rewrite systems. Thus its performance affects in a crucial way the global efficiency of each of these applications. This need for efficient unification algorithms has led to many investigations in the first-order setting. However, the efficient implementation of higher-order unification, especially for dependently typed $\lambda$-calculus, is still a central open problem limiting the potential impact of higher-order reasoning systems such as Twelf [15], Isabelle [12], or $\lambda$Prolog [9].

The most comprehensive study on efficient and robust implementation techniques for higher-order unification so far has been carried out by Nadathur and colleagues for the simply-typed $\lambda$-calculus in the programming language $\lambda$Prolog [7, 8]. The Teyjus compiler [10] embodies many of the insights found, in particular an adequate representation of lambda terms and mechanisms to delay and compose substitutions. Higher-order unification is implemented via Huet's algorithm [5] and special mechanisms are molded into the WAM instruction set to support branching and carrying unification problems. To only perform an occurs-check when necessary, the compiler distinguishes between the first occurrence and subsequent occurrences of a variable and compiles them into different WAM instructions. While for the first occurrence of a variable the occurs-check may be omitted, full unification is used for all subsequent variables. This approach seems to work well in the simply-typed setting, however it is not clear how to generalize it to dependent types.

* This work has been partially supported by NSF Grant CCR-9988281 "*Logical and Meta-Logical Frameworks*" and by a *Siebel Scholarship* awarded to the first author.

In this paper, we discuss the efficient implementation of higher-order pattern unification for the dependently typed lambda-calculus. Unlike Huet's general higher-order unification algorithm which involves branching and backtracking, higher-order pattern unification [6, 13] is deterministic and decidable. An important step toward the efficient implementation of higher-order pattern unification was the development based on explicit substitutions and de Bruijn indices [3] for the simply-typed lambda-calculus. This allows a clear distinction between bound and existential variables and reduces the problem to essentially first-order unification. Although the use of de Bruijn indices leads to a simple formal system, the readability may be obstructed and critical principles are obfuscated by the technical notation. In addition, some techniques like pre-cooking of terms and optimizations such as lowering and grafting remain ad hoc. This makes it more difficult to transfer these optimizations to other calculi.

We present an abstract view of existential variables in the dependently typed lambda-calculus based on modal type theory. Our calculus does not require de Bruijn indices, nor does it require closures $M[\sigma]$ as first-class terms. This leads to a simple clean framework which allows us to explain a number of features of the current implementation of higher-order unification in Twelf [15] and provides insight into several optimizations. In the paper, we will particularly focus on one optimization called linearization, which eliminates many unnecessary occurs-checks. We have implemented this optimization of higher-order unification as part of the Twelf system. Experimental results demonstrate significant performance improvement, including those in the area of proof-carrying code.

The paper is organized as follows: First we give some background on modal logic and modal type theory, and discuss its relation to the dependently typed lambda calculus (see Section 2). In Section 3 we discuss higher-order pattern unification. In particular, we focus on the optimization, called linearization, which eliminates unnecessary occurs-checks. Finally in Section 4 we discuss experimental results. Related work is discussed in Section 5.

## 2 A modal foundation for typed existential variables

### 2.1 Motivation

We start by presenting a foundation for dependently typed existential variables based on modal logic. Following the methodology of Pfenning and Davies [14], we can assign constructive explanations to modal operators. A key characteristic of this view, is to distinguish between propositions that are true and propositions that are valid. A proposition is valid if its truth does not depend on the truth of any other propositions. This leads to the basic hypothetical judgment

$$A_1\ \mathit{valid}, \ldots A_n\ \mathit{valid}; B_1\ \mathit{true}, \ldots, B_m\ \mathit{true} \vdash C\ \mathit{true}.$$

Under the multiple-world interpretation of modal logic, $C$ *valid* corresponds to $C$ *true* in *all* reachable worlds. This means $C$ *true* without any assumptions, except those that are assumed to be true in all worlds. We can generalize this idea to also capture truth relative to a set of specified assumptions by writing $C\ \mathit{valid}\ \Psi$, where $\Psi$ is the abbreviation for $C_1\,\mathit{true}, \ldots, C_n\,\mathit{true}$. In terms of the multiple world semantics, this means that $C$ is true in any world where $C_1$ through $C_n$ are all true and we say $C$ is valid relative to the assumptions in $\Psi$. Hypotheses about relative validity are more complex now, so our general judgment form is

$$A_1\ \mathit{valid}\ \Psi_1, \ldots, A_n\ \mathit{valid}\ \Psi_n; B_1\,\mathit{true}, \ldots, B_m\,\mathit{true} \vdash C\ \mathit{true}$$

While it is interesting to investigate this modal logic above in its own right, it does not come alive until we introduce proof terms. In this paper, we investigate the use of a modal proof term calculus as a foundation for existential variables. We will view existential variables $u$ as modal variables of type $A$ in a context $\Psi$ while bound variables are treated as ordinary variables. This allows us to distinguish between existential variables $u{::}(\Psi{\vdash}A)$ for relative validity assumptions $A\ \mathit{valid}\ \Psi$ declared in a modal context, and $x{:}A$ for ordinary truth assumptions $A\,\mathit{true}$ declared in an (ordinary) context. If we have an assumption $A\ \mathit{valid}\ \Psi$ we can only conclude $A\,\mathit{true}$ if we can verify all assumptions in $\Psi$.

$$\frac{\Delta, A\ \mathit{valid}\ \Psi, \Delta'; \Gamma \vdash \Psi}{\Delta, A\ \mathit{valid}\ \Psi, \Delta'; \Gamma \vdash A\,\mathit{true}}\ (*)$$

In other words, if we know $A\,\mathit{true}$ in $\Psi$, and all elements in $\Psi$ can be verified from the assumptions in $\Gamma$, then we can conclude $A\,\mathit{true}$ in $\Gamma$. As we will see in the next section, this transition from one context $\Psi$ to another context $\Gamma$, can be achieved via a substitutions from $\Psi$ to $\Gamma$.

## 2.2 Dependently typed lambda calculus based on modal logic

In this section, we introduce a dependently typed lambda calculus. Existential variables $u$ are treated as modal variables and $x$ denotes ordinary variables. $c$ and $a$ are constants, which are declared in a signature. This is a conservative extension of the LF [4] so we suppress some routine details such as signatures.

$$\begin{array}{rrcl}
\text{Kinds} & K & ::= & \mathsf{type} \mid \Pi x{:}A.\,K \\
\text{Families} & A, B, C & ::= & a \mid A\,M \mid \Pi x{:}A_1.\,A_2 \\
\text{Objects} & M, N & ::= & c \mid x \mid u[\sigma] \mid \lambda x{:}A.\,M \mid M_1\,M_2 \\
\text{Substitutions} & \sigma, \tau & ::= & \cdot \mid \sigma, M/x \\
\text{Contexts} & \Gamma, \Psi & ::= & \cdot \mid \Gamma, x{:}A \\
\text{Modal Contexts} & \Delta & ::= & \cdot \mid \Delta, u{::}(\Psi{\vdash}A)
\end{array}$$

Note that the substitution $\sigma$ is part of the syntax of existential variables. This eliminates the need of pre-cooking [3] which raises existential variables to the correct context.

The principal judgments are listed below. As usual, we omit similar judgments on types and kinds and all judgments concerning definitional equality.

| | |
|---|---|
| $\Delta; \Gamma \vdash M : A$ | Object $M$ has type $A$ |
| $\Delta; \Gamma \vdash \sigma : \Psi$ | Substitution $\sigma$ matches context $\Psi$ |
| $\vdash \Delta \ \mathsf{mctx}$ | $\Delta$ is a valid modal context |
| $\Delta \vdash \Psi \ \mathsf{ctx}$ | $\Psi$ is a valid context |

Note substitutions $\sigma$ are defined only on ordinary variables $x$ and not modal variables $u$. We write $\mathsf{id}_\Gamma$ for the identity substitution $(x_1/x_1, \ldots, x_n/x_n)$ for a context $\Gamma = (\cdot, x_1{:}A_1, \ldots, x_n{:}A_n)$. We will use $\pi$ for a substitution which may permute the variables, i.e $\pi = (x_{\Phi(1)}/x_1, \ldots, x_{\Phi(n)}/x_n)$ where $\Phi$ is a total permutation defined on the elements from a context $\Gamma = (\cdot, x_1{:}A_1, \ldots, x_n{:}A_n)$. We only consider well-typed substitutions, so $\pi$ must respect possible dependencies in its domain. We also streamline the calculus slightly by always substituting simultaneously for all ordinary variables. This is not essential, but saves some tedium in relating simultaneous and iterated substitution. Moreover, it is also closer to the actual implementation where we use de Bruijn indices and postpone explicit substitutions. The typing rules are given in Figure 1.

$$\frac{}{\Delta; \Gamma, x{:}A, \Gamma' \vdash x : A} \qquad \frac{\Delta, u{::}(\Psi {\vdash} A), \Delta'; \Gamma \vdash \sigma : \Psi}{\Delta, u{::}(\Psi {\vdash} A), \Delta'; \Gamma \vdash u[\sigma] : [\sigma]A}\ (*)$$

$$\frac{\Delta; \Gamma, x{:}A_1 \vdash M : A_2}{\Delta; \Gamma \vdash \lambda x.\, M : \Pi x{:}A_1.\, A_2} \qquad \frac{\Delta; \Gamma \vdash M_1 : \Pi x{:}A_2.\, A_1 \quad \Delta; \Gamma \vdash M_2 : A_2}{\Delta; \Gamma \vdash M_1\, M_2 : [\mathsf{id}_\Gamma, M_2/x]A_1}$$

$$\frac{}{\Delta; \Gamma \vdash (\cdot) : (\cdot)} \qquad \frac{\Delta; \Gamma \vdash \sigma : \Psi \quad \Delta; \Gamma \vdash M : [\sigma]A}{\Delta; \Gamma \vdash (\sigma, M/x) : (\Psi, x{:}A)}$$

$$\frac{}{\vdash (\cdot)\ \mathsf{mctx}} \qquad \frac{\vdash \Delta\ \mathsf{mctx} \quad \Delta \vdash \Psi\ \mathsf{ctx} \quad \Delta; \Psi \vdash A : \mathsf{type}}{\vdash (\Delta, u{::}(\Psi {\vdash} A))\ \mathsf{mctx}}$$

$$\frac{}{\Delta \vdash (\cdot)\ \mathsf{ctx}} \qquad \frac{\Delta \vdash \Psi\ \mathsf{ctx} \quad \Delta; \Psi \vdash A : \mathsf{type}}{\Delta \vdash (\Psi, x{:}A)\ \mathsf{ctx}}$$

**Fig. 1.** Typing rules for objects, substitutions, and context

Note that the rule for modal variables is the rule (*) presented in the previous section, annotated with proof terms and slightly generalized, because of the dependent type theory we are working in. This rule also justifies our implementation choice of using existential variables only in the form $u[\sigma]$.

Our convention is that substitutions as defined operations on expressions are written in prefix notation $[\sigma]P$ for an object, family, kind, or substitution $P$. These operations are capture-avoiding as usual. Moreover, we always assume that all free variables in $P$ are declared in $\sigma$. Substitutions that are part of the syntax are written in postfix notation, $u[\sigma]$. Note that such explicit substitutions occur only for variables $u$ labeling relative validity assumptions.

Substitutions are defined in a standard manner. We omit the details at the level of types and kinds for the sake of brevity.

$$\begin{aligned}
[\sigma]c &= c \\
[\sigma_1, M/x, \sigma_2]x &= M \\
[\sigma](u[\tau]) &= u[[\sigma]\tau] \\
[\sigma](N_1\, N_2) &= ([\sigma]N_1)\,([\sigma]N_2) \\
[\sigma](\lambda y{:}A.\, N) &= \lambda y{:}[\sigma]A.\, [\sigma, y/y]N \text{ provided } y \text{ not declared or free in } \sigma \\
\\
[\sigma](\cdot) &= (\cdot) \\
[\sigma](\tau, N/y) &= ([\sigma]\tau, [\sigma]N/y) \qquad \text{provided } y \text{ not declared or free in } \sigma
\end{aligned}$$

The side conditions can always be verified by (tacitly) renaming bound variables. We do not need an operation of applying a substitution $\sigma$ to a context. The last principle makes it clear that $[\sigma]\tau$ corresponds to composition of substitutions, which is sometimes written as $\tau \circ \sigma$.

The following substitution principles for substitutions $\sigma$ hold. They are suggested by the modal interpretation and proved by simple structural inductions. We elide corresponding principles for families and kinds.

**Theorem 1.**

1. *If $\Delta; \Gamma \vdash \sigma : \Psi$ and $\Delta; \Psi \vdash N : C$ then $\Delta; \Gamma \vdash [\sigma]N : [\sigma]C$.*
2. *If $\Delta; \Gamma \vdash \sigma : \Psi$ and $\Delta; \Psi \vdash \tau : \Psi'$ then $\Delta; \Gamma \vdash [\sigma]\tau : \Psi'$.*
3. *$[\sigma]([\tau]M) = [[\sigma]\tau]M$ and $[\sigma]([\tau]\tau') = [[\sigma]\tau]\tau'$*

A new and interesting operation arises from the substitution principles for relative validity. The new operation of substitution is compositional, but two interesting situations arise: when a variable $u$ is encountered, and when we substitute into a $\lambda$-abstraction. For sake of brevity, we only give the substitution on objects.

$$\begin{aligned}
[\![M/u]\!]c &= c \\
[\![M/u]\!]x &= x \\
[\![M/u]\!](u[\sigma]) &= [[\![M/u]\!]\sigma]M \\
[\![M/u]\!](v[\sigma]) &= v[[\![M/u]\!]\sigma] \quad \text{for } u \neq v \\
[\![M/u]\!](N_1\, N_2) &= ([\![M/u]\!]N_1)\,([\![M/u]\!]N_2) \\
[\![M/u]\!](\lambda y{:}A.\, N) &= \lambda y{:}[\![M/u]\!]A.\, [\![M/u]\!]N
\end{aligned}$$

We remark that the rule for substitution into a $\lambda$-abstraction does not require a side condition. This is because the object $M$ is defined in a different context, which is accounted for by the explicit substitution stored at occurrences of $u$. This ultimately justifies implementing substitution for existential variables by mutation.

Finally, consider the case of substituting into a closure, which is the critical case of this definition.

$$[\![M/u]\!](u[\sigma]) = [[\![M/u]\!]\sigma]M$$

This is clearly well-founded, because $\sigma$ is a subexpression (so $[\![M/u]\!]\sigma$ will terminate) and application of an ordinary substitution has been defined previously without reference to the new form of substitution.

Using the given definitions, we can then show that the new substitution operation for relative validity satisfies the substitution principles. Again, this is motivated by the logical interpretation and follows by simple inductions after straightforward generalization to encompass all syntactic categories.

**Theorem 2.**

1. *If* $\Delta;\Psi \vdash M : A$ *and* $\Delta, u{::}(\Psi{\vdash}A), \Delta'; \Gamma \vdash N : C$ *then* $\Delta, [\![M/u]\!]\Delta'; [\![M/u]\!]\Gamma \vdash [\![M/u]\!]N : [\![M/u]\!]C$
2. *If* $\Delta;\Psi \vdash M : A$ *and* $\Delta, u{::}(\Psi{\vdash}A), \Delta'; \Gamma \vdash \tau : \Psi'$ *then* $\Delta, [\![M/u]\!]\Delta'; [\![M/u]\!]\Gamma \vdash [\![M/u]\!]\tau : [\![M/u]\!]\Psi'$
3. $[\![M/u]\!]([\sigma]P) = [[\![M/u]\!]\sigma]([\![M/u]\!]P)$
4. $[\![M/u]\!]([\![N/v]\!]P) = [\![[\![M/u]\!]N/v]\!]([\![M/u]\!]P)$ *if* $u \neq v$ *and* $v$ *not free in* $M$

### 2.3 Normal Forms

There are two notions of normal forms that are useful in the implementation. The first corresponds to a $\beta$-normal form. We simultaneously define normal ($U$) and neutral ($R$) objects and normal substitutions $\eta$.

| | |
|---|---|
| Normal Objects | $U ::= \lambda x{:}A.\, U \mid R$ |
| Neutral Objects | $R ::= c \mid x \mid u[\eta] \mid R\, U$ |
| Normal Substitutions | $\eta ::= \cdot \mid \eta, U/x$ |

We obtain the canonical objects (long $\beta\eta$-normal forms) by requiring normal objects of the form $R$ to have base type (that is, not to have function type). In the implementation we use a stronger normal form where existential variables (represented here by modal variables) must also be of atomic type. This is accomplished by a technique called *lowering*. Lowering replaces a variable $u{::}(\Psi{\vdash}\Pi x{:}A_1.\, A_2)$ by a new variable $u'{:}(\Psi, x{:}A_1{\vdash}A_2)$. This process is repeated until all existential variables have a type of the form $\Psi \vdash b\, N_1 \ldots N_k$. This operation has been proved correct for the simply-typed case by Dowek et al. [3], but remains somewhat mysterious. Here, it is justified by the modal substitution principle.

**Lemma 1.**

1. *(Lowering) If* $\Delta, u{::}(\Psi{\vdash}\Pi x{:}A_1.\, A_2), \Delta'; \Gamma \vdash M : A$ *then* $\Delta, u'{::}(\Psi, x{:}A_1{\vdash}A_2), \Delta'^{-}; \Gamma^{-} \vdash M^{-} : A^{-}$ *where* $(P)^{-} = [\![(\lambda x{:}A_1.u'[\mathsf{id}_\Psi, x/x])/u]\!]P$.

2. *(Raising) If* $\Delta, u'{::}(\Psi, x{:}A_1 \vdash A_2), \Delta'; \Gamma \vdash M : A$ *then* $\Delta, u{::}(\Psi \vdash \Pi x{:}A_1.\, A_2), \Delta'^{+}; \Gamma^{+} \vdash M^{+} : A^{+}$ *where* $(P)^{+} = [\![(u[\mathsf{id}_\Psi]\, x)/u']\!]P$.
3. $()^{+}$ *and* $()^{-}$ *are inverse substitutions (modulo* $\beta\eta$*-conversion).*

*Proof.* Direct, by weakening and the modal substitution principle. For part (1) we observe that $\Delta, u'{::}(\Psi, x{:}A_1 \vdash A_2); \Psi \vdash \lambda x{:}A_1.u'[\mathsf{id}_\Psi, x/x] : \Pi x{:}A_1.\, A_2$. For part (2) we use instead that $\Delta, u{::}(\Psi \vdash \Pi x{:}A_1.\, A_2); \Psi, x{:}A_1 \vdash u[\mathsf{id}_\Psi]\, x : A_2$. Part (3) is direct by calculation.

Since we can lower all modal variables, we can change the syntax of normal forms so that terms $u[\eta]$ are also normal objects of base type, rather than neutral objects. This is, in fact, what we chose in the implementation.

## 2.4 Existential Variables

As mentioned several times above, in the implementation the modal variables in $\Delta$ are used to represent existential variables (also known as meta-variables), while the variables in $\Gamma$ are universal variables (also known as parameters).

Existential variables are created in an ambient context $\Psi$ and then lowered. We do not explicitly maintain a context $\Delta$ of these existential variables, but it is important that a proper order for them exists. Existential variables are created with a mutable reference, which is updated with an assignment when we need to carry out a substitution $[\![M/u]\!]$.

In certain operations, and particularly after type reconstruction, we need to abstract over the existential variables in a term. Since the LF type theory provides no means to quantify over $u{::}(\Psi \vdash A)$ we raise such variables until they have the form $u'{::}(\cdot \vdash A')$. It turns out that in the context of type reconstruction we can now quantify over them as ordinary variables $x'{:}A'$. However, this is not satisfactory as this requires first raising the type of existential variables for abstraction, and later again lowering the type of existential variables during unification to undo the effect of raising. To efficiently treat existential variables, we would like to directly quantify over modal variables $u$.

The judgmental reconstruction in terms of modal logic suggests two ways to incorporate modal variables. One way is via a new quantifier $\Pi^{\square} u{::}(\Psi \vdash A_1).\, A_2$, the other is via a general modal operator $\square_\Psi$. Proof-theoretically, the former is slightly simpler, so we will pursue this here. The new operator then has the form $\Pi^{\square} u{::}(\Psi \vdash A_1).\, A_2$ and is defined by the following rules.

$$\frac{\Delta; \Psi \vdash A : \mathsf{type} \quad \Delta, u{::}(\Psi \vdash A); \Gamma \vdash B : \mathsf{type}}{\Delta; \Gamma \vdash \Pi^{\square} u{::}(\Psi \vdash A).\, B : \mathsf{type}}$$

$$\frac{\Delta, u{::}(\Psi \vdash A); \Gamma \vdash M : B}{\Delta; \Gamma \vdash \lambda^{\square} u.\, M : \Pi^{\square} u{::}(\Psi \vdash A).\, B} \qquad \frac{\Delta; \Gamma \vdash N : \Pi^{\square} u{::}(\Psi \vdash A).\, B \quad \Delta; \Psi \vdash M : A}{\Delta; \Gamma \vdash N \mathbin{\square} M : [\![M/u]\!]B}$$

The main complication of this extension is that variables $u$ can now be bound and substitution must be capture avoiding. In the present implementation, this is handled by de Bruijn indices.

## 3 Toward efficient higher-order pattern unification

### 3.1 Preliminaries

In the following, we will consider the pattern fragment of the modal lambda-calculus. Higher-order patterns are terms where existential variables must be applied to distinct bound variables. This fragment was first identified by Miller [6] for the simply-typed lambda-calculus, and later extended by Pfenning [13] to the dependently typed and polymorphic case. We enforce that all terms are in normal form, and the type of existential variables has been lowered and is atomic. We call a normal term $U$ an *atomic pattern*, if all the subterms of the form $u[\sigma]$ are such that $\sigma = y_1/x_1, \ldots y_k/x_k$ 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.

Higher-order pattern unification can be done in two phases (see [13, 3] for a more detailed account). During the first phase, we decompose the terms until one of the two terms we unify is an existential variable $u[\sigma]$. This decomposition phase is straightforward and resembles first-order unification closely. During the second phase, we need to find an actual instantiation for the existential variable $u$. There are two main cases to distinguish: (1) when we unify two existential variables, $u[\sigma] \doteq v[\sigma']$, and when we unify an existential variable with another kind of term, $u[\sigma] \doteq M$. The latter case is transformed into $u \doteq [\sigma]^{-1} M$ assuming $u$ does not occur in $M$ and all variables $v[\tau]$ are *pruned* so that the free variables in $\tau$ all occur in the image of $\sigma$ (see [6, 3] for details). Note that we view $[\sigma]^{-1} M$ as a new meta-level operation such as substitution, because it may be defined even if $\sigma$ is not invertible in full generality.

The main efficiency problem in pattern unification lies in treating this last case. In particular, we must traverse the term $M$. First of all, we must perform the occurs-check to prevent cyclic terms. Second, we may need to prune the substitutions associated with existential variables occurring in $M$. Third, we need to ensure that all bound variables occurring in $M$ do occur in the range of $\sigma$, otherwise $[\sigma]^{-1} M$ does not exist. To illustrate the problem, we give the definition for inverting substitutions:

$$\begin{array}{rll}
[\sigma]^{-1}\, c = c & & \\
[\sigma]^{-1}\, x = y & & \text{if } x/y \in \sigma, \text{ undefined otherwise} \\
[\sigma]^{-1}\, (v[\tau]) = v[[\sigma]^{-1}\, \tau] & & \\
[\sigma]^{-1}\, (R\, U) = ([\sigma]^{-1}\, R)\, ([\sigma]^{-1}\, U) & & \\
[\sigma]^{-1}\, (\lambda x{:}A.\, U) = \lambda x{:}([\sigma]^{-1}\, A).\, [\sigma, x/x]^{-1}\, U & & \text{if } x \text{ not declared or free in } [\sigma]^{-1} \\
[\sigma]^{-1}\, (\cdot) = (\cdot) \qquad [\sigma]^{-1}\, (\tau, U/x) = ([\sigma]^{-1}\, \tau, [\sigma]^{-1}\, U/x) & &
\end{array}$$

In the next section, we will show how linearization can be used to enforce the two criteria which eliminates the need to traverse $M$. First, we will enforce that all existential variables occur only once, thereby eliminating the occurs-check. Second, we will require that the substitution $\sigma$ associated with existential variables is always a permutation $\pi$. This ensures that the substitutions are always invertible and eliminates the need for pruning.

### 3.2 Linearization

One critical optimization in unification is to perform the occurs-check only when necessary. While unification with the occurs-check is at best linear in the sum of the sizes of the terms being unified, unification without the occurs-check is linear in the smallest term being unified. In fact the occurs-check can be omitted if the terms are linear, i.e., every existential variable occurs only once.

Let us consider the following clause from a program which evaluates expressions from a small functional language Mini-ML. It says that functions evaluate to themselves. Using the introduced modal term language, this can be expressed as follows:

$\mathsf{exp} : \mathsf{type}.$
$\mathsf{lam} : (\mathsf{exp} \longrightarrow \mathsf{exp}) \longrightarrow \mathsf{exp}.$

$\mathsf{eval} : \mathsf{exp} \longrightarrow \mathsf{exp} \longrightarrow \mathsf{type}.$
$\mathsf{ev\_lam} : \Pi^{\Box} e{::}(y{:}\mathsf{exp} \vdash \mathsf{exp}).\mathsf{eval}\ (\mathsf{lam}\ (\lambda x{:}\mathsf{exp}.e[x/y]))\ (\mathsf{lam}\ (\lambda x{:}\mathsf{exp}.e[x/y])).$

The existential variable $e$ in the clause ev_lam is quantified by $\Pi^{\Box} e{::}(y{:}\mathsf{exp} \vdash \mathsf{exp})$. To enforce that every existential variable occurs only once, the clause head of ev_lam can be translated into the linear type

$$\begin{array}{l}\Pi^{\Box} e'{::}(z{:}\mathsf{exp} \vdash \mathsf{exp}).\Pi^{\Box} e{::}(y{:}\mathsf{exp} \vdash \mathsf{exp}).\\ \mathsf{eval}\ (\mathsf{lam}\ (\lambda x{:}\mathsf{exp}.e[x/y]))\ (\mathsf{lam}\ (\lambda x{:}\mathsf{exp}.e'[x/z]))\end{array}$$

together with the following variable definition

$$\forall x{:}\mathsf{exp}.e'[x/z] \overset{D}{=} e[x/y]$$

where $e'$ is a new existential variable. Then a constant time assignment algorithm can be used for assigning a linear clause head to a goal, and the variable definitions are solved by conventional unification. As a result, the occurs-check is only performed if necessary.

In the dependently typed lambda-calculus, there are several difficulties in performing this optimization. First of all, all existential variables carry their context $\Psi$ and type $A$. If we introduce a new existential variable, then the question arises what type should be assigned to it. As type inference is undecidable in the dependently typed case, this may be expensive. In general, we may even obtain a term which is not necessarily well-typed.

Let us modify the previous example, and annotate the expressions with their type thus enforcing that any evaluation of a Mini-ML expression will be well-typed.

$\mathsf{tp} : \mathsf{type}.$
$\mathsf{arrow} : \mathsf{tp} \longrightarrow \mathsf{tp} \longrightarrow \mathsf{tp}.$
$\mathsf{exp} : \mathsf{tp} \longrightarrow \mathsf{type}.$
$\mathsf{lam} : \Pi^{\Box} t_1{::}(\cdot \vdash \mathsf{tp}).\Pi^{\Box} t_2{::}(\cdot \vdash \mathsf{tp}).(\mathsf{exp}\ t_1\ \longrightarrow \mathsf{exp}\ t_2) \longrightarrow \mathsf{exp}\ (t_1 \Rightarrow t_2).$
$\mathsf{eval} : \Pi^{\Box} t{::}(\cdot \vdash \mathsf{tp}).\mathsf{exp}\ t\ \longrightarrow \mathsf{exp}\ t\ \longrightarrow \mathsf{type}.$
$\mathsf{ev\_lam} : \Pi^{\Box} t_1{::}(\cdot \vdash \mathsf{tp}).\Pi^{\Box} t_2{::}(\cdot \vdash \mathsf{tp}).\Pi^{\Box} e{::}(y{:}\mathsf{exp}\ t_1 \vdash \mathsf{exp}\ t_2).$
$\quad\quad \mathsf{eval}\ (\mathsf{arrow}\ t_1\ t_2)(\mathsf{lam}\ t_1\ t_2\ (\lambda x{:}\mathsf{exp}\ t_1.e[x/y]))\ (\mathsf{lam}\ t_1\ t_2\ (\lambda x{:}\mathsf{exp}\ t_1.e[x/y])).$

During linearization, the clause head of $\mathsf{ev\_lam}$ will now be translated into

$$\begin{array}{l}\Pi^{\Box} t_1{::}(\cdot{\vdash}\mathsf{tp}).\Pi^{\Box} t_2{::}(\cdot{\vdash}\mathsf{tp}).\Pi^{\Box} e{::}(y{:}\mathsf{exp}\ t_1{\vdash}\mathsf{exp}\ t_2).\\ \Pi^{\Box} t_3{::}(\cdot{\vdash}\mathsf{tp}).\Pi^{\Box} t_4{::}(\cdot{\vdash}\mathsf{tp}).\Pi^{\Box} t_5{::}(\cdot{\vdash}\mathsf{tp}).\Pi^{\Box} t_6{::}(\cdot{\vdash}\mathsf{tp}).\Pi^{\Box} e'{::}(z{:}\mathsf{exp}\ t_1{\vdash}\mathsf{exp}\ t_2).\\ \mathsf{eval}\ (\mathsf{arrow}\ t_1\ t_2)(\mathsf{lam}\ t_3\ t_4\ (\lambda x{:}\mathsf{exp}\ t_1.e[x/y]))\ (\mathsf{lam}\ t_5\ t_6\ (\lambda x{:}\mathsf{exp}\ t_1.e'[x/z])).\end{array}$$

and the following variable definitions

$$t_1 \overset{D}{=} t_3 \wedge\ t_1 \overset{D}{=} t_5 \wedge\ t_2 \overset{D}{=} t_4 \wedge\ t_2 \overset{D}{=} t_6 \wedge\ \forall x{:}\mathsf{exp}\ t_1.e'[x/z] \overset{D}{=} e[x/y]$$

Due to the linearization, the linear clause head is clearly not well-typed. However, it is well-typed modulo variable definitions. Therefore, it will be well-typed after all existential variables have been instantiated during assignment and the variable definitions have been solved. It would be interesting to accord first-class type-theoretic status to the variable definitions, but we leave this to future work, since the implementation treats them only in a very special manner explained in Section 3.3. Note that some of these variable definitions are in fact redundant, which is another orthogonal optimization (see [11] for an analysis on a fragment of LF).

It is worth pointing out that this situation does not arise in the simply typed case. These considerations lead to the following description of variable definitions:

$$\text{Variable Definitions}\quad D ::= \mathsf{true} \mid u[\mathsf{id}_{\Psi}] \overset{D}{=} U \mid D_1 \wedge\ D_2 \mid \forall x{:}A.D$$

The idea of factoring out duplicate existential variables can be generalized to replacing arbitrary subterms by new existential variables and creating variable definitions. In particular, the process of linearization also replaces any existential variables $v[\sigma]$ where $\sigma$ is *not* a permutation by a new variable $u[\mathsf{id}_{\Psi}]$ and a variable definition $u[\mathsf{id}_{\Psi}] \overset{D}{=} v[\sigma]$.

The linearization itself is quite straightforward and we will omit the details here. In the actual implementation, we do not generate types $A$ and contexts $\Psi$ for the new, linearly occurring existential variables, but ensure that all such variables are in instantiated and disappear by the time the variable definitions have been solved.

### 3.3 Assignment for higher-order patterns

In this section, we give a refinement of the general higher-order pattern unification which exploits the presented ideas. The algorithm proceeds in three phases. First we will unify a linear atomic higher-order pattern $L$ with an object $U$. The following judgments capture the assignment between a linear atomic higher-order pattern $L$ and a normal object $U$. We write $\theta$ for simultaneous substitutions $[\![U_1/u_1, \ldots U_n/u_n]\!]$ for existential variables which have straightforward definition and properties.

$\Delta; \Gamma \vdash L \doteq U/(\theta, E)$ assignment for normal objects
$\Delta; \Gamma \Vdash L' \doteq R'/(\theta, E)$ assignment for neutral objects

where $R$ is a linear neutral object. $E$ denotes residual equations which may be generated during assignment. The assignment algorithm itself is given below.

$$\frac{\Delta; \Gamma, x{:}A \vdash L \doteq U/(\theta, E)}{\Delta; \Gamma \vdash \lambda x{:}A.L \doteq \lambda x{:}A.U/(\theta, \forall x{:}A.E)}\ lam \qquad \frac{}{\Delta; \Gamma \vdash u[\pi] \doteq U/([\pi]^{-1}\, U/u, \mathsf{true})}\ existsL$$

$$\frac{\Delta; \Gamma \Vdash L' \doteq R/(\theta, E)}{\Delta; \Gamma \vdash L' \doteq R/(\theta, E)}\ coerce \qquad \frac{}{\Delta; \Gamma \vdash L \doteq u[\sigma]/(\cdot, L\ \doteq\ u[\sigma])}\ existsR$$

$$\frac{}{\Delta; \Gamma \Vdash x \doteq x/(\cdot; \mathsf{true})}\ const \qquad \frac{}{\Delta; \Gamma \Vdash c \doteq c/(\cdot; \mathsf{true})}\ var$$

$$\frac{\Delta; \Gamma \Vdash L' \doteq R/(\theta_1; E_1) \quad \Delta; \Gamma \vdash L \doteq U/(\theta_2, E_2)}{\Delta; \Gamma \Vdash L'\ L \doteq R\ U/(\theta_1 \cup \theta_2; E_1 \wedge\ E_2)}\ app$$

Note that we do not need to worry about capture in the rule *lam*, since existential variables and bound variables are defined in different contexts. In the rule *app*, we are allowed to union the two substitutions $\theta_1$ and $\theta_2$, as the linearity requirement ensures that the domains of both substitutions are disjoint. Note that the case for unifying an existential variable $u[\pi]$ with another term $U$ is now simpler and more efficient than in the general higher-order pattern case. In particular, it does not require a traversal of $U$ (see rule *existsL*). Since the inverse of the substitution $\pi$ can be computed directly and will be total, we know $[\pi]^{-1}\, U$ exists and can simply generate a substitution $[\pi]^{-1}\, U/u$. Finally, we may need to postpone solving some unification problems and generate some residual equations if the non-linear term is an existential variable (see *existsR*).

The result of the assignment algorithm is a substitution $\theta_1$ for the existential variables in $L$ and potentially some residual equations $E$. In the second phase, we apply $\theta_1$ to the variable definitions $D$ which were generated during linearization of $U'$ and solve $[\![\theta_1]\!]D$ using conventional pattern unification. We only need to pay attention to the case where we unify an existential variable $u[\mathsf{id}_\Psi]$ with another variable $u'[\sigma]$. In this case, we simply generate a substitution $[\![u'[\sigma]/u]\!]$, as the inverse of $\mathsf{id}_\Psi$ will be the identity substitution again. This ensures that all existential variables introduced during linearization, will be instantiated after assignment succeeds.

As a final result of solving the variable definitions $D$, we obtain an additional substitution $\theta_2$. In the third phase, we solve the remaining residual equations $E$, which were generated during phase 1 under $\theta_1 \circ \theta_2$.

## 4 Experiments

In this section, we discuss some experimental results with different programs written in Twelf. All experiments are done on a machine with the following specifications: 1.60GHz Intel Pentium Processor, 256 KB cache. We are using SML of New Jersey 110.0.3 under Linux Red Hat 7.1. Times are measured in seconds. In the tables below, the column "opt" refers to the optimized version with linearization and assignment, while the column "stand" refers to the standard implementation using general higher-order pattern unification.

## 4.1 Higher-order logic programming

In this section, we present two experiments with higher-order logic programming. The first one uses an implementation of a meta-interpreter for ordered linear logic by Polakow and Pfenning [16]. In the second experiment we evaluate our unification algorithm using an implementation of foundational proof-carrying code developed at Princeton University [1].

Meta-interpreter for ordered linear logic

| example | opt | stand | speed-up | variable def | | calls to assign | |
|---|---|---|---|---|---|---|---|
| | | | | trivial | non-tr. fail | success | fail |
| sqnt (bf) | 0.84 | 2.09 | 149% | 44% | 46% | 23% | 77% |
| sqnt (dfs) | 0.93 | 2.35 | 152% | 44% | 47% | 22% | 78% |
| sqnt (perm) | 4.44 | 7.11 | 60% | 44% | 52% | 20% | 80% |
| sqnt (rev) | 1.21 | 1.70 | 40% | 45% | 48% | 21% | 79% |
| sqnt (mergesort) | 2.26 | 3.39 | 50% | 46% | 53% | 20% | 80% |

As the results for the meta-interpreter demonstrate, the performance improvement ranges between 40% and 152%. Roughly, 45% of the time there were no variable definitions at all. From the non-trivial equations roughly 45% were not unifiable. This means overall, in approx. 20% - 30% of the cases the assignment algorithm succeeded and the failure of unification was delayed. It is worth noting that 77% to 80% of the calls to assignment presented in Section 3.3 fail immediately.

Foundational proof-carrying code

| example | opt | stand | speed-up | variable def | | calls to assign | |
|---|---|---|---|---|---|---|---|
| | | | | trivial | non-tr. fail | success | fail |
| inc | 5.8 | 9.19 | 58% | 64% | 46% | 18% | 82% |
| switch | 36.00 | 49.69 | 38% | 64% | 48% | 19% | 81% |
| mul2 | 5.51 | 9.520 | 72% | 64% | 46% | 18% | 82% |
| div2 | 121.96 | 153.610 | 26% | 63% | 48% | 20% | 80% |
| divx | 333.69 | 1133.150 | 239% | 63% | 50% | 21% | 79% |
| listsum | 1073.33 | $\infty$ | $\infty$ | 65% | 45% | 18% | 82% |
| polyc | 2417.85 | $\infty$ | $\infty$ | 65% | 41% | 17% | 83% |
| pack | 197.07 | 1075.610 | 445% | 66% | 45% | 19% | 82% |

In the table above we show the performance on programs from the proof-carrying code benchmark. Performance is improved by up to 445% and some examples are not executable without linearization and assignment. The results clearly demonstrate that an efficient unification algorithm is critical in large-scale examples.

### 4.2 Higher-order theorem proving

Besides a logic programming engine, the Twelf system also provides a theorem prover which is based on iterative deepening search. In this section, we consider two examples, theorem proving in an intuitionist sequent calculus and theorem proving in the classical natural deduction calculus.

Proof search in the intuitionist sequent calculus

| example | opt | stand | speed-up | variable def | | calls to assign | |
|---|---|---|---|---|---|---|---|
| | | | | trivial | non-tr. fail | success | fail |
| dist-1 | 53.00 | 57.11 | 8% | 100% | 0% | 52% | 48% |
| distImp | 0.40 | 0.44 | 10% | 100% | 0% | 53% | 47% |
| pierce | 1520.77 | 1563.35 | 3% | 100% | 0% | 52% | 48% |
| trans | 0.13 | 0.13 | 0% | 100% | 0% | 53% | 47% |

As the results demonstrate, the performance of the theorem prover is not greatly influenced by the optimized unification algorithm. The main reason is that we have many dynamic assumptions, which need to be unified with the current goal. However, we use the standard higher-order pattern unification algorithm for this operation and use the optimized algorithm only for selecting a clause. For dynamic assumptions we cannot maintain the linearity requirement and linearizing the dynamic assumptions at run-time seems too expensive.

Proof search in NK (and, impl, neg)

| example | opt | stand | speed-up | variable def | | calls to assign | |
|---|---|---|---|---|---|---|---|
| | | | | trivial | non-tr. fail | success | fail |
| andEff1-nk | 7.67 | 13.14 | 71% | 100% | 0% | 80% | 20% |
| andEff2-nk | 3.86 | 6.58 | 70% | 100% | 0% | 81% | 19% |
| assocAnd-nk | 2.24 | 3.74 | 67% | 100% | 0% | 81% | 19% |
| combS-nk | 3.85 | 6.64 | 72% | 100% | 0% | 81% | 19% |

The second example is theorem proving in the natural deduction calculus. In contrast to the previous experiments with the sequent calculus, there is a substantial performance improvement by approximately 70%. Although linear head compilation substantially improves performance, more optimizations, such as tabelling and indexing, are needed to solve more complex theorems.

## 5 Related Work

The language most closely related to our work, is $\lambda$Prolog. Two main different implementations of $\lambda$Prolog exist, Prolog/Mali and Teyjus. In Prolog/MALI implementation, the occurs-check is left out entirely[2]. While Teyjus [8, 7] also eliminates some unnecessary occurs-checks statically during compilation. However, in addition to the presence of dependencies in Twelf, there are several other differences between our implementation and Teyjus. 1) Teyjus compiles first and subsequent occurrences of existential variables into different instruction. Therefore, assignment and unification are freely mixed during the execution. This may lead to expensive failure in some cases, since unification is still called. In our approach, we perform a simple fast assignment check and delay unification entirely. As the experimental results demonstrate, only a small percentage of the cases fails after it already passed the assignment test and most cases benefit from a fast simple assignment check. 2) We always assume that the types of existential variables are lowered. This can be done at compile time and incurs no run-time overhead. In Huet's unification algorithm, projection and imitation rules are applied at run-time to construct the correct prefixes of $\lambda$-abstractions. 3) Our approach can easily incorporate definitions and constraint domains. This is important since unifying definitions and constraint expressions may potentially be expensive. In fact, we generalize and extend the idea of linearization in the implementation and factor out not only duplicate existential variables but also any difficult sub-expressions such as definitions and constraint expressions. Therefore, our approach seems more general than the one adopted in Teyjus.

## 6 Conclusion

We have presented modal foundation for existential variables which underlies our higher-order pattern unification implementation in Twelf. This leads to a simple framework in which many optimizations such as lowering, grafting and linearization can be justified. As experiments show, performance is improved substantially. This is especially important in large-scale applications such as proof-carrying code and allows us explore the full potential of logical frameworks in real-world applications.

In the future, we plan to investigate and implement further optimizations to reduce the performance gap between higher-order and first-order system. One optimization which is particularly important to sustain performance in large-scale examples is term indexing. However, indexing of higher-order terms is still an open problem. We believe the presented strategy is a suitable basis to adopt indexing techniques from the first-order setting for two reasons: 1) the presented strategy reduces the higher-order unification problem to a simple class of problems which is essentially first-order. 2) the experimental results show that many unification problems arising in practice fall into this class and can be solved by this strategy. This indicates that a term indexing algorithm based on this assignment strategy may also be effective in practice.

## 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. Pascal Brisset and Olivier Ridoux. Naive reverse can be linear. In Koichi Furukawa, editor, *International Conference on Logic Programming*, pages 857–870, Paris, France, June 1991. MIT Press.

3. 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.
4. 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.
5. Gérard Huet. A unification algorithm for typed $\lambda$-calculus. *Theoretical Computer Science*, 1:27–57, 1975.
6. 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.
7. Gopalan Nadathur. A treatment of higher-order features in logic programming. Technical Report draft, available upon request, Department of Computer Science and Engineering, University of Minnesota, January 2003.
8. Gopalan Nadathur, Bharat Jayaraman, and Debra Sue Wilson. Implementation considerations for higher-order features in logic programming. Technical Report CS-1993-16, Department of Computer Science, Duke University, June 1993.
9. Gopalan Nadathur and Dale Miller. An overview of $\lambda$Prolog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.
10. Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus—a compiler and abstract machine based implementation of lambda prolog. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 287–291, Trento, Italy, July 1999. Springer-Verlag LNCS.
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. Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.
13. 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.
14. Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.
15. 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.
16. Jeff Polakow and Frank Pfenning. Ordered linear logic programming. Technical Report CMU-CS-98-183, Department of Computer Science, Carnegie Mellon University, December 1998.