Functional Unification of Higher-Order Patterns*

Tobias Nipkow†
TU München‡

Abstract

The complete development of a unification algorithm for so-called *higher-order patterns*, a subclass of λ-terms, is presented. The starting point is a formulation of unification by transformation, the result a directly executable functional program. In a final development step the result is adapted to λ-terms in de Bruijn's notation. The algorithms work for both simply typed and untyped terms.

1 Introduction

Unification of typed λ-terms (aka *higher-order unification*) is the basic inference mechanism for many theorem provers and programming languages based on typed λ-calculus [1, 15, 11, 16]. The implementations of most, if not all of these systems, rely on the unification algorithm invented by Gérard Huet [6]. Although the problem itself is undecidable and infinite sets of most general unifiers may exist, Huet's algorithm performs surprisingly well in practice. This observation lead Dale Miller to the discovery of a subclass of λ-terms, so-called **patterns**, which behave almost like first-order terms w.r.t. unification: unification is decidable and unifiable terms have most general unifiers, which are easy to compute. In practice these patterns turn up very frequently, which explains why Huet's algorithm is well-behaved most of the time. Miller [8] formulated a unification algorithm for patterns which can ultimately be seen as a specialized and improved version of Huet's algorithm: imitation and projection coincide and flex-flex pairs can be solved. His results were taken up by Pfenning [17], who extended them to the Calculus of Constructions, and by Nipkow [12, 13], who reformulated and used them in the context of rewrite systems over simply typed λ-terms. Miller also showed that full higher-order unification can be regained by embedding pattern-unification in a search procedure [9], thus separating the treatment of variable binding from the search aspect.

The unification algorithm presented by Miller applies to quantified λ-terms (both untyped and simply typed) and is described informally (although rigorously); the algorithms by Pfenning and Nipkow are formulated at a fairly high level. The purpose of this paper is to present the development of a simple yet efficient implementation of unification of both untyped and simply typed patterns. The algorithm is written in a functional style and follows the structure of unification for first-order terms.

In the future Miller's algorithm is bound to acquire a similar importance in the field of automatic term manipulations (including theorem proving, logic programming, term rewriting and programming languages) that first-order unification has had for a long time. In fact, ELF has already abandoned full higher-order unification in favour of patterns [7], some implementations of λProlog treat patterns specially (by solving flex-flex patterns, rather than suspending them [10]), and Isabelle[1] [15] is about to follow suit. Hence it is vital that a simple and yet practical implementation of this algorithm is available.

After fixing the notation and the basic definitions (Section 2), a high-level description of unification using transformation rules (Section 3) is gradually refined to a directly executable functional algorithm (Sections 4–9). Initially we restrict our attention to simply typed terms, but at some point in the development the algorithm starts to work for both typed and untyped terms. An adaption of this algorithm to λ-terms in de Bruijn's notation is presented in Section 10.

2 Notation

2.1 λ-Calculus

We use standard λ-calculus notation subject to the following conventions: s and t denote terms, F, G

*Appeared in Proceedings of the 8th IEEE Symposium *Logic in Computer Science*, 1993.

†Research supported by ESPRIT BRA 6453, *TYPES*, and ESPRIT WG 6028, *CCL*.

‡Address: Institut für Informatik, Technische Universität München, Postfach 20 24 20, W-8000 München 2, Germany. Email: Tobias.Nipkow@Informatik.TU-Muenchen.De

[1] Larry Paulson's implementation of Huet's algorithm already contains some of the ideas of pattern unification.

and H free variables, x, y and z bound variables, a and b atoms (i.e. bound variables or constants), and c constants. Binary application of λ-terms is written $s.t$ in order to avoid confusion with application in the programming language used for expressing the algorithms. We follow the convention of keeping bound and free variables disjoint. Hence the language of λ-terms is defined by the following grammar:

$$t \;=\; F \mid x \mid c \mid \lambda x.t \mid (t_1.t_2)$$

Although it is extremely convenient to distinguish bound and free variables on a syntactic level, it introduces so-called **loose bound variables** as in $\lambda x.y$, where y is a bound variable without a corresponding enclosing binder. Such loose bound variables should essentially be treated like constants, except that renaming of bound variables has to take care not to capture such loose variables. The free variables in a term are denoted by $\mathcal{FV}(.)$.

Lists of variables and terms are written as $\overline{x_n}$ and $\overline{s_n}$, where n is the length of the list. In particular $\lambda\overline{x_n}.s$ is short for $\lambda x_1 \ldots \lambda x_n.s$ and $a(\overline{s_n})$ is short for the iterated application $((\cdots(a.s_1)\cdots).s_n)$. Lists can be turned into sets by enclosing them in braces. Conversely, given a set $\{\overline{x_m}\}$, the x_i are all assumed to be distinct and the list $\overline{x_m}$ denotes an arbitrary enumeration of the elements in $\{\overline{x_m}\}$.

Definition 2.1 A term t in β-normal form is a (**higher-order**) **pattern** if every free occurrence of a variable F is in a subterm $F(\overline{u_n})$ of t such that $\overline{u_n}$ is η-equivalent to a list of distinct bound variables.

Examples of higher-order patterns are $\lambda x.c(x)$, F, $\lambda x.F(\lambda z.x(z))$, and $\lambda x, y.F(y, x)$, examples of non-patterns are $F(c)$, $\lambda x.F(x, x)$ and $\lambda x.F(F(x))$. In the sequel all terms are either implicitly assumed to be patterns or are patterns by construction.

Substitutions, i.e. finite mappings from variables to terms, are denoted by θ; $t\theta$ is the application of θ to t (including renaming of bound variables in t), $t\theta{\downarrow_\beta}$ the β-normal form of $t\theta$. Individual substitutions are written as $\{v_1 \mapsto t_1, \ldots, v_n \mapsto t_n\}$ where the v_i can be free or bound. Composition of substitutions is $\circ$: $t(\theta' \circ \theta) = (t\theta)\theta'$. The restriction of θ to a set of variables V is written $\theta|_V$.

2.2 Algorithms

All algorithms are written in a mixture of simplified Standard ML and mathematics. In particular we rely on the following basic combinators:

$$\begin{array}{lcl} x :: [y_1, \ldots, y_n] & = & [x, y_1, \ldots, y_n] \\ [x_1, \ldots, x_m]@[y_1, \ldots, y_n] & = & [x_1, \ldots, x_m, y_1, \ldots, y_n] \\ map\ f\ [x_1, \ldots, x_m] & = & [f\ x_1, \ldots, f\ x_m] \end{array}$$

$$\begin{array}{lcl} foldl\ f\ a\ (x :: xs) & = & foldl\ f\ (f\ a\ x)\ xs \\ foldl\ f\ a\ [] & = & a \end{array}$$

$$\begin{array}{lcl} zip\ (x :: xs)\ (y :: ys) & = & (x, y) :: zip\ xs\ ys \\ zip\ []\ [] & = & [] \end{array}$$

$$\begin{array}{lcl} assoc\ x\ ((y, z) :: ps) & = & \texttt{if}\ x = y\ \texttt{then}\ [z] \\ & & \texttt{else}\ assoc\ x\ ps \\ assoc\ x\ [] & = & [] \end{array}$$

Above, *assoc* is the lookup function for association lists. Returning either an empty or a singleton list avoids the introduction of a new data type with two constructors. Two-letter variable names ending in "s" stand for lists.

Lists can also be defined by *comprehension* [2]:

$$[e(i) \mid 1 \leq i \leq n \wedge P(i)]$$

is the list $[e(i_1), \ldots, e(i_k)]$ where $\{i_1, \ldots, i_k\} = \{i \mid 1 \leq i \leq n \wedge P(i)\}$.

3 Unification by transformation

The starting point of the development is an extremely transparent but very high-level description of unification using five simple rewrite rules shown in Fig. 1, adapted and simplified from [12].

We assume that

α : α-equivalent terms are identified, and

η : terms are simply typed and in η-expanded form, except for the arguments of free variables, which are in η-normal form, i.e. bound variables.

Both assumptions are later relaxed.

The transformation rules are of the form $e \to \langle E', \theta' \rangle$ and should be read as follows: the unification problem e is reduced to the list of unification problems E' under the new substitution θ'. Note that in rule (1) the binder λx can be removed because free and bound variables are from disjoint sets and can always be distinguished. In rule (3) the terms $H_i(\overline{x_m})$ may need to be η-expanded in case s_i is of function type. The free variable H is assumed to be a *new* variable in each instance.

$$
\begin{array}{rcll}
\lambda x.s =^? \lambda x.t & \to & \langle [s =^? t], \{\}\rangle & (1)\\
a(\overline{s_n}) =^? a(\overline{t_n}) & \to & \langle [s_1 =^? t_1, \ldots, s_n =^? t_n], \{\}\rangle & (2)\\
F(\overline{x_m}) =^? a(\overline{s_n}) & \to & \langle [H_1(\overline{x_m}) =^? s_1, \ldots, H_n(\overline{x_m}) =^? s_n], \{F \mapsto \lambda\overline{x_m}.a(\overline{H_n(\overline{x_m})})\}\rangle & (3)\\
 & & \text{where } F \notin \mathcal{FV}(\overline{s_n}) \text{ and } a \text{ is a constant or } a \in \{\overline{x_m}\} & \\
F(\overline{x_n}) =^? F(\overline{y_n}) & \to & \langle [], \{F \mapsto \lambda\overline{x_n}.H(\overline{z_p})\}\rangle & (4)\\
 & & \text{where } \{\overline{z_p}\} = \{x_i \mid x_i = y_i\} & \\
F(\overline{x_m}) =^? G(\overline{y_n}) & \to & \langle [], \{F \mapsto \lambda\overline{x_m}.H(\overline{z_p}), G \mapsto \lambda\overline{y_n}.H(\overline{z_p})\}\rangle & (5)\\
 & & \text{where } F \neq G \text{ and } \{\overline{z_p}\} = \{\overline{x_m}\} \cap \{\overline{y_n}\} &
\end{array}
$$

Figure 1: Unification by transformation

The actual transformation relation is defined by

$$\langle e :: E, \theta\rangle \longrightarrow \langle E'@(E\theta'{\downarrow_\beta}), \theta' \circ \theta\rangle \quad \text{if } e \to \langle E', \theta'\rangle.$$

Using lists rather than sets is intentional. Treating the unification problems as sets can result in divergence: try $\{F =^? c(G), G =^? c(F)\}$.

Theorem 3.1 *A list of pattern unification problems E has a solution iff $\langle E, \{\}\rangle \longrightarrow^* \langle [], \theta\rangle$, in which case $\theta|_{\mathcal{FV}(E)}$ is a most general unifier of E.*

Proof sketch. Roughly speaking, correctness and completeness follow because $e \to \langle E', \theta'\rangle$ implies that $e\theta'$ and E' have the same set of unifiers and because the five rules cover all solvable cases. To prove termination, first convert all higher-order patterns to first-order terms by dropping all λs and all arguments of free variables. The resulting set of transformation rules is a variation on first-order unification as, for example, studied by Dershowitz and Jouannaud [5] and can be shown to terminate employing the same methods used there. □

The correctness of all further versions of pattern unification can be established by program transformation and data refinement techniques. The main correctness arguments in each step are sketched but details of the proofs have been left out.

Example 3.2 The following transformations show the unification of $\lambda x, y.F(x)$ and $\lambda x, y.c(G(y, x))$:

$$
\begin{array}{ll}
\langle [\lambda x, y.F(x) =^? \lambda x, y.c(G(y, x))], \{\}\rangle & \longrightarrow\\
\langle [\lambda y.F(x) =^? \lambda y.c(G(y, x))], \{\}\rangle & \longrightarrow\\
\langle [F(x) =^? c(G(y, x))], \{\}\rangle & \longrightarrow\\
\langle [H(x) =^? G(y, x)], \{F \mapsto \lambda x.c(H(x))\}\rangle & \longrightarrow\\
\langle [], \{F \mapsto \lambda x.c(H'(x)), H \mapsto \lambda x.H'(x),\\
\quad G \mapsto \lambda y, x.H'(x)\}\rangle &
\end{array}
$$

4 Going functional

The first development step is a serialization of the above inference rules leading to the functional program displayed in Fig. 2. The free variable H is assumed to be a *new* variable in each instance. The condition $m \neq n$ prepares the ground for untyped terms; for simply typed terms, $m = n$ is implied by $F = G$ and by $a = b$.

The correctness of this algorithm can be shown by relating it back to the transformation rules. The functions *rigidrigid*, *flexflex*1 and *flexflex*2 correspond to rules (2), (4) and (5). Function *proj*, and hence *flexrigid*, corresponds to a sequence of transformations by rule (3), interleaved with rules (4) and (5). The precise relationship is:

Lemma 4.1 *Let $t = s\theta{\downarrow_\beta}$. If proj $\overline{x_m}$ θ $s = \delta \circ \theta$ and $F \notin \mathcal{FV}(t)$ then $\langle (F(\overline{x_m}) =^? t) :: E, \theta'\rangle \longrightarrow^* \langle E\delta', \delta' \circ \theta'\rangle$ such that $\delta'|_{\mathcal{FV}(t)} = \delta \circ \{F \mapsto \lambda\overline{x_m}.t\}$.*

Proof by structural induction on s. □

Intuitively, *proj* $\overline{x_m}$ θ s tries to eliminate all loose bound variables from t which are not in $\{\overline{x_m}\}$. It does so by replacing free variables by projections onto that subset of their arguments contained in $\{\overline{x_m}\}$.

Example 4.2 The unification problem from Example 3.2 is first stripped down to $F(x) =^? c(G(y, x))$. Now *flexrigid* calls *proj* $\{x\}$ θ $(c(G(y, x)))$, where $\theta = \{F \mapsto \lambda x.c(G(y, x))\}$ (note the loose bound y!), which in turn composes $\{G \mapsto \lambda y, x.H(x)\}$ with θ, thus eliminating y from $G(y, x)$.

Note that the above algorithm, and all further ones, may fail for two reasons:

1. Exception `fail` is raised explicitly. This means that the input is not unifiable.

$$
\begin{array}{lcl}
unif\ \theta\ (s,t) & = & \texttt{case}\ (s\theta{\downarrow_\beta}, t\theta{\downarrow_\beta})\ \texttt{of} \\
& & \quad (\lambda x.s, \lambda x.t) \Rightarrow unif\ \theta\ (s,t) \\
& & \quad (s,t) \Rightarrow cases\ \theta\ (s,t)
\end{array}
$$

$$
\begin{array}{lcl}
cases\ \theta\ (F(\overline{x_m}), G(\overline{y_n})) & = & flexflex(F, \overline{x_m}, G, \overline{y_n}, \theta) \\
cases\ \theta\ (F(\overline{x_m}), t) & = & flexrigid(F, \overline{x_m}, t, \theta) \\
cases\ \theta\ (t, F(\overline{x_m})) & = & flexrigid(F, \overline{x_m}, t, \theta) \\
cases\ \theta\ (a(\overline{s_m}), b(\overline{t_n})) & = & rigidrigid(a, \overline{s_m}, b, \overline{t_n}, \theta)
\end{array}
$$

$$
flexflex(F, \overline{x_m}, G, \overline{y_n}, \theta) \quad = \quad \texttt{if}\ F = G\ \texttt{then}\ flexflex1(F, \overline{x_m}, \overline{y_n}, \theta)\ \texttt{else}\ flexflex2(F, \overline{x_m}, G, \overline{y_n}, \theta)
$$

$$
flexflex1(F, \overline{x_m}, \overline{y_n}, \theta) \quad = \quad \texttt{if}\ m \neq n\ \texttt{then fail else}\ \{F \mapsto \lambda\overline{x_m}.H([x_i \mid 1 \leq i \leq m \wedge x_i = y_i])\} \circ \theta
$$

$$
flexflex2(F, \overline{x_m}, G, \overline{y_n}, \theta) \quad = \quad \texttt{let}\ \{\overline{z_k}\} = \{\overline{x_m}\} \cap \{\overline{y_n}\}\ \texttt{in}\ \{F \mapsto \lambda\overline{x_m}.H(\overline{z_k}), G \mapsto \lambda\overline{y_n}.H(\overline{z_k})\} \circ \theta
$$

$$
flexrigid(F, \overline{x_m}, t, \theta) \quad = \quad \texttt{if}\ F \in \mathcal{FV}(t)\ \texttt{then fail else}\ proj\ \{\overline{x_m}\}\ (\{F \mapsto \lambda\overline{x_m}.t\} \circ \theta)\ t
$$

$$
rigidrigid(a, \overline{s_m}, b, \overline{t_n}, \theta) \quad = \quad \texttt{if}\ a \neq b\ \texttt{orelse}\ m \neq n\ \texttt{then fail else}\ foldl\ unif\ \theta\ (zip\ \overline{s_m}\ \overline{t_n})
$$

$$
\begin{array}{lcl}
proj\ V\ \theta\ s & = & \texttt{case}\ s\theta{\downarrow_\beta}\ \texttt{of} \\
& & \quad \lambda x.t \Rightarrow proj\ (V \cup \{x\})\ \theta\ t \\
& & \quad c(\overline{s_m}) \Rightarrow foldl\ (proj\ V)\ \theta\ \overline{s_m} \\
& & \quad x(\overline{s_m}) \Rightarrow \texttt{if}\ x \in V\ \texttt{then}\ foldl\ (proj\ V)\ \theta\ \overline{s_m}\ \texttt{else fail} \\
& & \quad F(\overline{y_m}) \Rightarrow \texttt{let}\ \{\overline{z_n}\} = \{\overline{y_m}\} \cap V\ \texttt{in}\ \{F \mapsto \lambda\overline{y_m}.H(\overline{z_n})\} \circ \theta
\end{array}
$$

Figure 2: Functional unification

2. A function is called with an argument which is not covered by any of the clauses defining the function. This means that the algorithm was applied to arguments outside its domain, namely non-patterns.

In a real implementation it may be advantageous to test for non-patterns during unification, possibly branching to a full higher-order unification algorithm. To make sure that our algorithms reject all non-patterns, one has to impose the restriction that $\overline{x_m}$ only matches a list of *distinct* variables. Unsurprisingly, the solution computed in the repeated variable case is not a most general unifier: try $\lambda x.F(x,x) =^? \lambda x.x$.

5 On-the-fly α-conversion

Let us now drop assumption α and take α-conversion (almost) seriously.

$$
\begin{array}{lcl}
unif\ \theta\ (s,t) = \texttt{case}\ (s\theta{\downarrow_\beta}, t\theta{\downarrow_\beta})\ \texttt{of} \\
\quad (\lambda x.s, \lambda y.t) \Rightarrow unif\ \theta\ (s, t\{y \mapsto x\}) \\
\quad (s,t) \Rightarrow cases\ \theta\ (s,t)
\end{array}
$$

We have simply converted $\lambda y.t$ to $\lambda x.t\{y \mapsto x\}$. However, in case t contains a loose bound x introduced by removing some corresponding λx, this x is now captured. Hence we have to assume that in a term $\lambda x.s$, s does not contain a subterm $\lambda x.t$, i.e. nested abstractions bind distinct variables. If necessary, this assumption can be enforced by preprocessing. In addition we have to ensure that renaming, for example during β-reduction, cannot introduce bound variables that have already been used. The obvious solution is to generate a completely new bound variable in each renaming step. With the introduction of de Bruijn's indices in Section 10 all such complications disappear.

6 Implementing substitutions

Now we implement substitutions as association lists of variables and terms. Consequently we replace substitutions of the form $\{F \mapsto t\} \circ \theta$ by $(F,t) :: \theta$. The expression $s\theta{\downarrow_\beta}$ now becomes $devar\ \theta\ s$, where $devar$ is a "lazy" form of substitution application: only if s is of the form $F(\ldots)$ is F replaced by $F\theta$; otherwise s

$$
\begin{array}{lcl}
cases\ \theta\ (s,t) & = & \texttt{case}\ (strip\ s, strip\ t)\ \texttt{of} \\
& & \quad ((F,ys),(G,zs)) \Rightarrow flexflex(F,ys,G,zs,\theta) \\
& & \quad ((F,ys),\_) \Rightarrow flexrigid(F,ys,t,\theta) \\
& & \quad (\_,(F,ys)) \Rightarrow flexrigid(F,ys,s,\theta) \\
& & \quad ((a,ss),(b,ts)) \Rightarrow rigidrigid(a,ss,b,ts,\theta) \\
 & & \\
proj\ V\ \theta\ s & = & \texttt{case}\ strip(devar\ \theta\ s)\ \texttt{of} \\
& & \quad (\lambda x.t,[]) \Rightarrow proj\ (V \cup \{x\})\ \theta\ t \\
& & \quad (c,ss) \Rightarrow foldl\ (proj\ V)\ \theta\ ss \\
& & \quad (x,ss) \Rightarrow \texttt{if}\ x \in V\ \texttt{then}\ foldl\ (proj\ V)\ \theta\ ss\ \texttt{else fail} \\
& & \quad (F,\overline{y_m}) \Rightarrow \texttt{let}\ \{\overline{z_n}\} = \{\overline{y_m}\} \cap V\ \texttt{in}\ (F, \lambda\overline{y_m}.H(\overline{z_n})) :: \theta
\end{array}
$$

Figure 3: Pattern-matching

is returned unchanged:

$$
\begin{array}{lcl}
devar\ \theta\ (F(\overline{x_n})) & = & \texttt{case}\ assoc\ F\ \theta\ \texttt{of} \\
& & \quad [\lambda\overline{y_n}.t] \Rightarrow devar\ \theta\ (t\{\overline{y_n \mapsto x_n}\}) \\
& & \quad [] \Rightarrow F(\overline{x_n}) \\
devar\ \theta\ s & = & s
\end{array}
$$

This is justified because the `case`-expressions in *unif* and *proj* examine only the outermost layer of a term. However, the test $F \in \mathcal{FV}(t)$ in *flexrigid* may examine all of t and is therefore replaced by $occ\ F\ \theta\ t$ where

$$
\begin{array}{lcl}
occ\ F\ \theta\ G & = & (F = G)\ \texttt{orelse} \\
& & \texttt{case}\ assoc\ G\ \theta\ \texttt{of} \\
& & \quad [s] \Rightarrow occ\ F\ \theta\ s \\
& & \quad [] \Rightarrow false \\
occ\ F\ \theta\ (s.t) & = & occ\ F\ \theta\ s\ \texttt{orelse}\ occ\ F\ \theta\ t \\
occ\ F\ \theta\ (\lambda x.s) & = & occ\ F\ \theta\ s \\
occ\ F\ \theta\ \_ & = & false
\end{array}
$$

7 On-the-fly η-expansion

The following modifications remove the need to work with η-expanded simply typed terms. This is not just relevant for applications to untyped terms. It also paves the way for terms containing type variables which may get instantiated during unification (as in λProlog and Isabelle): such instantiations may require further η-expansions. It is not difficult to see that the η-expansions below are correct for typed terms as well. Hence the resulting algorithm, and all further refinements, apply to both typed and untyped terms alike.

However, we still retain the assumption that all arguments to free variables must be bound variables. This merely simplifies the notation and can be relaxed, for example, by a preprocessing phase that η-normalizes the arguments of free variables.

$$
\begin{array}{lcl}
unif\ \theta\ (s,t) & = & \texttt{case}\ (devar\ \theta\ s, devar\ \theta\ t)\ \texttt{of} \\
& & \quad (\lambda x.s, \lambda y.t) \Rightarrow unif\ \theta\ (s, t\{y \mapsto x\}) \\
& & \quad (\lambda x.s, t) \Rightarrow unif\ \theta\ (s, t.x) \\
& & \quad (s, \lambda x.t) \Rightarrow unif\ \theta\ (s.x, t) \\
& & \quad (s,t) \Rightarrow cases\ \theta\ (s,t)
\end{array}
$$

$$
\begin{array}{lcl}
devar\ \theta\ s & = & \texttt{case}\ strip\ s\ \texttt{of} \\
& & \quad (F,ys) \Rightarrow \texttt{case}\ assoc\ F\ \theta\ \texttt{of} \\
& & \quad\quad [t] \Rightarrow devar\ \theta\ (red(t,ys)) \\
& & \quad\quad [] \Rightarrow s \\
& & \quad \_ \Rightarrow s
\end{array}
$$

$$
\begin{array}{lcl}
red(\lambda x.s, y :: ys) & = & red(s\{x \mapsto y\}, ys) \\
red(s, y :: ys) & = & red(s.y, ys) \\
red(s, []) & = & s
\end{array}
$$

$$
\begin{array}{lcl}
strip\ t & = & \texttt{let}\ strip\ (f.t, ts) = strip\ (f, t :: ts) \\
& & \quad\quad strip\ (t, ts) = (t, ts) \\
& & \texttt{in}\ strip\ (t, [])
\end{array}
$$

Calling $red(s, ys)$ applies s to the list of variables ys, performing as many β-reductions as possible. Since terms are no longer η-expanded, the number of λs at the front of s and the length of ys may differ.

8 Pattern-matching

As a final step towards an executable algorithm we remove all occurrences of (programming language) patterns of the form $a(\overline{s_m})$ in favour of the destructor *strip*. This has already occurred in the last version of *devar*. The necessary changes to *cases* and *proj* are shown in Fig. 3.

The point is that we have chosen the traditional binary application representation of λ-terms $(s.t)$ but

$$
\begin{array}{lcl}
flexflex1(F, \overline{x_m}, \overline{y_m}, \theta) & = & \texttt{if } \overline{x_m} = \overline{y_m} \texttt{ then } \theta \texttt{ else } (F, \lambda\overline{x_m}.H([x_i \mid 1 \le i \le m \wedge x_i = y_i])) :: \theta \\
\\
flexflex2(F, \overline{x_m}, G, \overline{y_n}, \theta) & = & \texttt{if } \{\overline{x_m}\} \subseteq \{\overline{y_n}\} \texttt{ then } (G, \lambda\overline{y_n}.F(\overline{x_m})) :: \theta \texttt{ else} \\
 & & \texttt{if } \{\overline{x_m}\} \supseteq \{\overline{y_n}\} \texttt{ then } (F, \lambda\overline{x_m}.G(\overline{y_n})) :: \theta \texttt{ else} \\
 & & \texttt{let } \{\overline{z_k}\} = \{\overline{x_m}\} \cap \{\overline{y_n}\} \texttt{ in } (F, \lambda\overline{x_m}.H(\overline{z_k})) :: (G, \lambda\overline{y_n}.H(\overline{z_k})) :: \theta
\end{array}
$$

Figure 4: Minimizing new variables

that we often want to view the terms in their functor-arguments form $(a(\overline{s_m}))$. The functions *red* and *strip* realize the isomorphisms between both representations: $red(strip\ t) = t$, if t is in β-normal form, and $strip(red(a, ss)) = (a, ss)$.

Making binary application primitive and deriving the functor-arguments form from it is not necessarily optimal. Measurements by Michaylov and Pfenning [7] indicate that in an environment where unification is the bottleneck it may be advantageous to make the functor-arguments form primitive. In our algorithm this would certainly pay off since translations back to binary application form would not be required.

9 Minimizing new variables

Careful handling of flex-flex cases can often avoid the introduction of new variables, as shown in Fig. 4. The last line of *proj* can be improved similarly:

$$
\begin{array}{lcl}
(F, \overline{x_m}) & \Rightarrow & \texttt{if } \{\overline{x_m}\} \subseteq V \texttt{ then } \theta \\
 & & \texttt{else let } \{\overline{y_n}\} = \{\overline{x_m}\} \cap V \\
 & & \quad \texttt{in } (F, \lambda\overline{x_m}.H(\overline{y_n})) :: \theta
\end{array}
$$

10 De Bruijn's notation

Recall that λ-terms in de Bruijn's notation [4] are generated by the following grammar

$$e \;=\; F \mid i \mid c \mid (e_0.e_1) \mid \lambda e$$

where i ranges over natural numbers. A natural number i represents a variable bound by the $(i+1)^{\text{st}}$ λ above i. For example $\lambda x.\lambda y.(x.y)$ is represented by $\lambda\lambda(1.0)$. The abbreviation λ_k denotes k nested abstractions.

The main advantage of de Bruijn's notation is the fact that α-equivalent terms are really identical, which has made this representation popular with implementors. Unfortunately this is also a bit of a problem because the previous unification algorithm made essential use of the ability to rename bound variables. For example the solution to the flex-flex pair $F(x, y) =^? G(y, z)$ is the substitution $\{F \mapsto \lambda x, y.H(y), G \mapsto \lambda y, z.H(y)\}$, where the bodies are the same $(H(y))$ and the binders differ. In de Bruijn's notation it is just the other way around: $\{F \mapsto \lambda_2 H(0), G \mapsto \lambda_2 H(1)\}$. The problem becomes worse in the flex-rigid case: $F(\overline{x_m}) =^? t$ has the solution $\theta \circ \{F \mapsto \lambda\overline{x_m}.t\}$, where θ contains the necessary projections. The point is that t can be copied from the problem to the solution. With de Bruijn's indices we cannot force new binders on t but we have to adjust the loose bound variables inside t.

The following algorithm is a translation of the version in Section 8, i.e. without minimizing new variables, into de Bruijn's notation. The main unification functions are displayed in Fig. 5, auxiliary ones in Fig. 6. Functions which are syntactically identical to their alphabetic counterpart, e.g. $flexflex$ and *occ*, have been omitted. Although the structure of the two algorithms is very close, computations with de Bruijn indices are notoriously subtle and deserve special attention.

The two flex-flex cases involve some index arithmetic but are straightforward translations. As motivated above, the main difference to the version with alphabetic bound variables is found in $flexrigid$ where *mapbnd* adjusts the loose bound variables inside the rigid term t. In general, calling *mapbnd* f t applies f to all loose bound variables in t, taking the depth, i.e. the number of enclosing λs, into account. The function *idx* $\overline{i_n}$ performs two tasks: it adjusts those loose bound variables which appear in $\overline{i_n}$ and replaces the ones not in $\overline{i_n}$ by $-\infty$. On a second pass through t, *proj* eliminates all occurrences of $-\infty$ with appropriate projections. Since the index arithmetic involved is quite subtle, an example may help.

Example 10.1 Consider again the unification problem from Example 3.2. In de Bruijn's notation it becomes $\lambda\lambda F(1) =^? \lambda\lambda c(G(0, 1))$. Applying *mapbnd* $(idx\ [1])$ to $c(G(0, 1))$ yields $u = c(G(-\infty, 0))$. Calling $proj\ \theta\ u$, where $\theta = \{F \mapsto \lambda u\}$, eliminates $-\infty$ by composition of $\{G \mapsto \lambda\lambda H(0)\}$ with θ.

Note that $-\infty$ was chosen to indicate "dead" variables because *mapbnd* may add an offset to bound variables. But since $-\infty + d = -\infty$ for any natural number d, this adjustment leaves dead variables dead.

$$
\begin{array}{lcl}
unif\ \theta\ (s,t) & = & \texttt{case}\ (devar\ \theta\ s, devar\ \theta\ t)\ \texttt{of} \\
& & \quad (\lambda s, \lambda t) \Rightarrow unif\ \theta\ (s,t) \\
& & \quad (\lambda s, t) \Rightarrow unif\ \theta\ (s, (incr\ t).0) \\
& & \quad (s, \lambda t) \Rightarrow unif\ \theta\ ((incr\ s).0, t) \\
& & \quad (s,t) \Rightarrow cases\ \theta\ (s,t)
\end{array}
$$

$$
\begin{array}{lcl}
cases\ \theta\ (s,t) & = & \texttt{case}\ (strip\ s, strip\ t)\ \texttt{of} \\
& & \quad ((F, \overline{i_m}), (G, \overline{j_n})) \Rightarrow flexflex(F, \overline{i_m}, G, \overline{j_n}, \theta) \\
& & \quad ((F, \overline{i_m}), \_) \Rightarrow flexrigid(F, \overline{i_m}, t, \theta) \\
& & \quad (\_, (F, \overline{i_m})) \Rightarrow flexrigid(F, \overline{i_m}, s, \theta) \\
& & \quad ((a, \overline{s_m}), (b, \overline{t_n})) \Rightarrow rigidrigid(a, \overline{s_m}, b, \overline{t_n}, \theta)
\end{array}
$$

$$
flexflex1(F, \overline{i_m}, \overline{j_n}, \theta) \quad = \quad \texttt{if}\ m \neq n\ \texttt{then fail else}\ (F, \lambda_m H([m-k \mid 1 \leq k \leq m \wedge i_k = j_k])) :: \theta
$$

$$
flexflex2(F, \overline{i_m}, G, \overline{j_n}, \theta) \quad = \quad \texttt{let}\ \{\overline{k_l}\} = \{\overline{i_m}\} \cap \{\overline{j_n}\}\ \texttt{in}\ (F, lam(\overline{i_m}, H, \overline{k_l})) :: (G, lam(\overline{j_n}, H, \overline{k_l})) :: \theta
$$

$$
lam(\overline{i_m}, G, \overline{j_n}) \quad = \quad \lambda_m G(map\ (idx\ \overline{i_m})\ \overline{j_n})
$$

$$
flexrigid(F, \overline{i_n}, t, \theta) \quad = \quad \texttt{if}\ occ\ F\ \theta\ t\ \texttt{then fail else let}\ u = mapbnd\ (idx\ \overline{i_n})\ t\ \texttt{in}\ proj\ ((F, \lambda_n u) :: \theta)\ u
$$

$$
\begin{array}{lcl}
proj\ \theta\ s & = & \texttt{case}\ strip(devar\ \theta\ s)\ \texttt{of} \\
& & \quad (\lambda t, []) \Rightarrow proj\ \theta\ t \\
& & \quad (c, ss) \Rightarrow foldl\ proj\ \theta\ ss \\
& & \quad (i, ss) \Rightarrow \texttt{if}\ i < 0\ \texttt{then fail else}\ foldl\ proj\ \theta\ ss \\
& & \quad (F, \overline{i_n}) \Rightarrow (F, \lambda_n H([n-j \mid 1 \leq j \leq n \wedge i_j \geq 0])) :: \theta
\end{array}
$$

Figure 5: Unification in de Bruijn's notation

Nevertheless $-\infty$ smells of a hack. The main reason for doing it this way is one of presentation. A more efficient way of solving the flex-rigid case is to traverse t only once, adjusting its loose bound variables and at the same time computing the necessary projections. Unfortunately this leads to a more complex function because both a term and a substitution are computed. Nevertheless it is preferable, not just for efficiency reasons: the introduction of $-\infty$ destroys the link to the corresponding binder which may hold information such as the type or (for printing) the external name of the bound variable.

11 Efficiency

The algorithms presented above may take exponential time. This is not hard to see since their restriction to first-order terms is essentially Robinson's algorithm, where substitutions are represented as graphs rather than trees. This leads to linear space consumption but is still exponential in time [3]. However, the amazing success of Prolog, many of whose implementations are both unsound (no occurs-check) and exponential, has shown that in practice this is not a problem.

Asymptotically efficient first-order unification algorithms, for example Paterson and Wegman's [14], rely on pointer structures for their efficiency. For first-order terms this works well, because the unification of a variable with a term can be implemented by updating a pointer. In the higher-order case this is less straightforward because the variable may be applied to some arguments. Yet Qian [18] has presented a linear (imperative) unification algorithm for patterns. Not surprisingly, the functional algorithm in this paper is far simpler. Whether it is also faster for small terms can only be assessed by comparing two implementations. However, no implementation of Qian's algorithm is available to the author.

12 Variations

There are two variations of the unification problem that are important in practice: matching, and unification of terms whose types may contain type variables (as in ML, LCF, HOL, Isabelle, ...).

Matching is significantly easier than unification. There are no flex-flex cases and the flex-rigid case $F(\overline{x_m}) =^? t$ can be solved in one step: either all loose

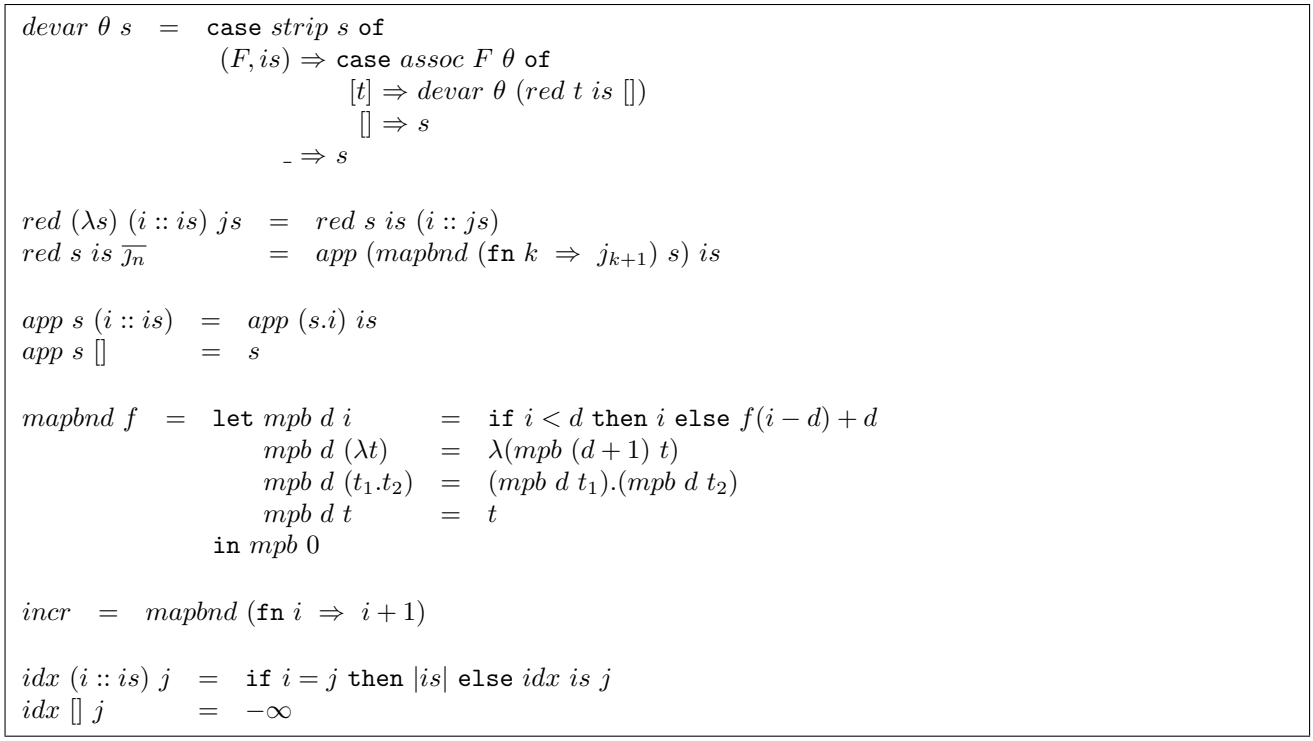

Figure 6: Auxiliary functions for de Bruijn's notation

bound variables in t are contained in $\{\overline{x_m}\}$, in which case the solution is $F \mapsto \lambda\overline{x_m}.t$, or there is no solution.

Type variables can be dealt with by maintaining the invariant that the two terms to be unified have the same type. The invariant can only be destroyed in the rigid-rigid case. If the heads of the rigid terms are bound variables, they must be identical and hence of the same type[2]. If they are constants, they can have different instances of the same polymorphic type. These instances must be unified, thus establishing the invariant for the equations between the arguments of the constants.

Acknowledgement Many thanks are due to Larry Paulson, Christian Prehofer, Konrad Slind and Oskar Slotosch for their careful reading and their comments.

References

[1] P. B. Andrews, S. Issar, D. Nesmith, and F. Pfenning. The TPS theorem proving system. In M. Stickel, editor, *Proc. 10th Int. Conf. Automated Deduction*, pages 641–642. LNCS 449, 1990.

[2] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall, 1988.

[3] J. Corbin and M. Bidoit. A rehabilitation of Robinson's unification algorithm. In R. Pavon, editor, *Proc. 1983 IFIP Congress*, pages 909–914. North-Holland, 1983.

[4] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[5] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Formal Models and Semantics, Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. Elsevier - The MIT Press, 1990.

[6] G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[7] S. Michaylov and F. Pfenning. An empirical study of the runtime behaviour of higher-order logic programs. In D. Miller, editor, *Proc. Workshop on λProlog*, pages 257–271, 1992.

[2] λ-bound variables are monomorphic.

```
fun foldl f (a,x::xs) = foldl f (f(a,x),xs)
  | foldl f (a,[]) = a;

fun foldr f (x::xs,a) = f(x,foldr f (xs,a))
  | foldr f ([],a) = a;

datatype 'a option = Some of 'a | None;

fun assoc x ((y,t)::ps) = if x=y then Some(t) else assoc x ps
  | assoc x [] = None;

infix mem;
fun (x mem (y::ys)) = x=y orelse x mem ys
  | (x mem []) = false;

infix /\;
fun ((x::xs) /\ ys) = if x mem ys then x::(xs /\ ys) else xs /\ ys
  | ([] /\ _) = [];

fun map f (x::xs) = f x :: map f xs
  | map f [] = [];
```

Figure 7: Basic library

[8] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In P. Schroeder-Heister, editor, *Extensions of Logic Programming*, pages 253–281. LNCS 475, 1991.

[9] D. Miller. Unification of simply typed lambda-terms as logic programming. In P. Furukawa, editor, *Proc. 1991 Joint Int. Conf. Logic Programming*, pages 253–281. MIT Press, 1991.

[10] D. Miller. Private communication, November 1992.

[11] G. Nadathur and D. Miller. An overview of λProlog. In R. A. Kowalski and K. A. Bowen, editors, *Proc. 5th Int. Logic Programming Conference*, pages 810–827. MIT Press, 1988.

[12] T. Nipkow. Higher-order critical pairs. In *Proc. 6th IEEE Symp. Logic in Computer Science*, pages 342–349, 1991.

[13] T. Nipkow. Orthogonal higher-order rewrite systems are confluent. In M. Bezem and J. F. Groote, editors, *Proc. Int. Conf. Typed Lambda Calculi and Applications*. LNCS 664, 1993.

[14] M. Paterson and M. Wegman. Linear unification. *J. Computer and System Sciences*, 16:158–167, 1978.

[15] L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–385. Academic Press, 1990.

[16] F. Pfenning. Logic programming in the LF Logical Framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 66–78. Cambridge University Press, 1991.

[17] F. Pfenning. Unification and anti-unification in the calculus of constructions. In *Proc. 6th IEEE Symp. Logic in Computer Science*, pages 74–85, 1991.

[18] Z. Qian. Linear unification of higher-order patterns. In J.-P. Jouannaud, editor, *Proc. 1993 Coll. Trees in Algebra and Programming*. LNCS, 1993. To appear.

A The complete ML code

Figures 7 through 10 present the translation of the algorithm from Section 8 into Standard ML. This implementation, together with one for de Bruijn's notation, is also available by email/ftp from the author.

```
(*** Generation of new names --- non functional!     ***)
(*** Assumption: names in input terms are not primed ***)

val Fname = ref("F");
val Bname = ref("x");

fun newname(r) = (r := !r ^ "'"; !r);
fun newB() = newname Bname;

infix 9 $;
infixr \;
datatype term = V of string            (* free var  *)
              | B of string            (* bound var *)
              | C of string            (* constant  *)
              | op $ of term * term    (* application *)
              | op \ of string * term; (* abstraction *)

fun newV() = V(newname Fname);

fun B1(B s) = s;

fun strip t = let fun strip(s$t,ts) = strip(s,t::ts)
                    | strip p = p
              in strip(t,[]) end;

fun abs(xs,t) = foldr op \ (map B1 xs,t);

fun hnf(xs,F,ss) = abs(xs, foldl op $ (F,ss));
```

Figure 8: Terms

```
fun occ F S (V G) = (F=G) orelse (case assoc G S of Some(s) => occ F S s | None => false)
  | occ F S (s$t) = occ F S s orelse occ F S t
  | occ F S (_\s) = occ F S s
  | occ F S (_)   = false;

fun subst y x = let fun sub(b as B z) = if z=x then B y else b
                      | sub(s1$s2) = (sub s1) $ (sub s2)
                      | sub(t as z\s) = if z=x then t else
                                        if z<>y then z \ sub s else
                                        let val z' = newB() in z' \ sub(subst z' z s) end
                      | sub(s) = s
                in sub end;

fun red (x\s) (y::ys) = red (subst (B1 y) x s) ys
  | red s     (y::ys) = red (s$y) ys
  | red s     []      = s;

fun devar S t = case strip t of
                  (V F,ys) => (case assoc F S of Some(t) => devar S (red t ys) | None => t)
                | _ => t;
```

Figure 9: Substitutions

```
exception Unif;

fun proj W (S,s) = case strip(devar S s) of
      (x\t,_)  => proj (x::W) (S,t)
    | (C _,ss) => foldl (proj W) (S,ss)
    | (B x,ss) => if x mem W then foldl (proj W) (S,ss) else raise Unif
    | (V F,ss) => (F, hnf(ss, newV(), ss /\ (map B W))) :: S;

fun eqs (x::xs) (y::ys) = if x=y then x::eqs xs ys else eqs xs ys
  | eqs [] [] = []
  | eqs _ _  = raise Unif;

fun flexflex1(F,ym,zn,S) = (F, hnf(ym, newV(), eqs ym zn)) :: S;

fun flexflex2(F,ym,G,zn,S) =
    let val xk  = ym /\ zn and H = newV()
    in (F, hnf(ym,H,xk)) :: (G, hnf(zn,H,xk)) :: S end;

fun flexflex(F,ym,G,zn,S) = if F=G then flexflex1(F,ym,zn,S) else flexflex2(F,ym,G,zn,S);

fun flexrigid(F,ym,t,S) = if occ F S t then raise Unif
                          else proj (map B1 ym) (((F,abs(ym,t))::S),t);

fun zip (x::xs) (y::ys) = (x,y) :: zip xs ys
  | zip [] [] = []
  | zip _ _ = raise Unif;

fun unif (S,(s,t)) = case (devar S s,devar S t) of
        (x\s,y\t) => unif (S,(s,if x=y then t else subst x y t))
      | (x\s,t)   => unif (S,(s,t$(B x)))
      | (s,x\t)   => unif (S,(s$(B x),t))
      | (s,t)     => cases S (s,t)

and cases S (s,t) = case (strip s,strip t) of
                      ((V F,ym),(V G,zn)) => flexflex(F,ym,G,zn,S)
                    | ((V F,ym),_)        => flexrigid(F,ym,t,S)
                    | (_,(V F,ym))        => flexrigid(F,ym,s,S)
                    | ((a,sm),(b,tn))     => rigidrigid(a,sm,b,tn,S)

and rigidrigid(a,ss,b,ts,S) = if a <> b then raise Unif else foldl unif (S,zip ss ts);
```

Figure 10: Unification