Theoretical Computer Science 1 (1975) 27–57. © North-Holland Publishing Company

A UNIFICATION ALGORITHM FOR TYPED $\bar{\lambda}$-CALCULUS

G. P. HUET

IRIA—Laboria, 78150 Rocquencourt, France

Communicated by R. Milner
Received December 1973
Revised August 1974

Abstract. A semi-decision algorithm is presented, to search for unification of formulas in typed ω-order λ-calculus, and its correctness is proved.

It is shown that the search space is significantly smaller than the one for finding the most general unifiers. In particular, our search is not redundant. This allows our algorithm to have good directionality and convergence properties.

1. Introduction

Given two well-formed formulas e_1 and e_2 of a logic $\mathcal{L}$, any substitution σ for the free variables of e_1 and e_2 such that $\sigma e_1 = \sigma e_2$ is called a *unifier* of e_1 and e_2. (If these free variables are supposed to be independent in e_1 and e_2 we assume that we have renamed them so that independent variables are distinct.) The computation of unifiers is a very basic process in automatic theorem proving or proof checking since it gives us the basic pattern-matching capability by which we can recognize whether a rule of inference is applicable to one or several formulas by some substitution for their free variables. It is well-known that in first-order logic, for any terms e_1 and e_2 which can be unified at all, there exists a most general unifier (MGU) of these terms, such that every unifier of e_1 and e_2 can be obtained from σ by composition with some other substitution. A simple algorithm computing this MGU (or reporting on the failure of unification) was independently given by Guard [7] under the name of matching and Robinson [18]. It was used by Robinson to formulate a complete refutational system for first-order logic whose unique rule of inference, called resolution, can be described as a "cut modulo unification". Unification can thus be seen as an elegant way of getting rid of the substitution rule. Similarly, Robinson and Wos [17] defined a rule of inference called paramodulation which uses unification to recognize possible equality substitutions.

The situation is more complex in higher-order languages, since most general unifiers do not exist any more. The problem was first investigated by Guard [7] and then by Gould [6], who unfortunately restricted himself to the search for common instances of terms. Gould noted that in some cases one needs to consider infinite sets of unifiers. A few incomplete algorithms were given by Darlington [3, 4] for second-order logic (f-matching) and Ernst [5] for ω-order logic. Then Pietrzykowski [5] gave a complete enumeration algorithm for second-order logic, which was later extended to ω-order logic in Jensen and Pietrzykowski [13]. In this last algorithm, unifiers are computed by composition of five elementary rules, called elimination, iteration, projection, imitation and identification.

In this paper, we shall present an algorithm which searches for the existence of unifiers in ω-order logic. Although this problem is undecidable as shown in Huet [11] and Lucchesi [14], so that if two terms are not unifiable then our algorithm may never stop, we show here that this problem is much easier than the enumeration of maximal general unifiers. In particular, we need only compose two elementary rules, similar to imitation and projection. The very prolific rules of iteration and identification are not used, and our search seems to have good convergence properties. Most importantly, we show that there is no redundancy in our search for unification. This contrasts with the fact that any search for most general unifiers has to be redundant, as shown in Huet [10].

This algorithm is used in a refutation-complete system generalizing resolution to ω-order logic described in Huet [9, 12] for which we only need know the possibility of unification, and not the actual computation of unifiers. It should be noted however that whenever unification is possible we actually obtain an explicit unifier with our algorithm.

We shall first describe briefly the language we use (a slight modification of Church's typed λ-calculus). Then we describe our algorithm, prove its correctness and discuss a few heuristic improvements.

2. An overview of typed λ-calculus

Notation: $[n] = \{1, 2, ..., n\}$.

2.1 *Types*

Our language is derived from Church's simple theory of types (Church [2]). Every well-formed expression (term) of the language possesses a unique *type* which indicates its position in a functional hierarchy.

We choose a finite set T_0 of elementary types, and the set T of types is defined as the smallest superset of T_0 closed by:

$$\alpha, \beta \in T \Rightarrow (\alpha \to \beta) \in T.$$

(T is therefore a context-free language over $T_0 \cup \{(,), \to\}$.)

If A is a set of elements of type α and B a set of elements of type β, then $(\alpha \to \beta)$ denotes the type of functions with domain A and range B. We designate types by the Greek letters α, β, γ.

2.2. *Terms*

Terms are composed of atoms, applications and abstractions.

2.2.1. *Atoms*

We have a denumerable set $\mathcal{V}_\alpha$ of variables of type α for every $\alpha \in T$, and an at most denumerable set $\mathcal{C}$ of constants of arbitrary given types. All sets $\mathcal{V}_\alpha$ and $\mathcal{C}$ are pairwise disjoint, and the set $\mathcal{A}$ of *atoms* is defined as:

$$\mathcal{A} = \mathcal{C} \cup \mathcal{V}, \text{ where } \mathcal{V} = \bigcup_{\alpha \in T} \mathcal{V}_\alpha .$$

We designate variables by lower case letters $x, y, \ldots, f, g, \ldots$, constants by capitals $A, B, \ldots, F, G, \ldots$ and atoms by the symbols @, @′, ...; the type of atom @ is denoted by τ(@).

2.2.2. *Applications*

If e_1 is a term of type $(\alpha \to \beta)$ and e_2 a term of type α, then the *application* $(e_1\, e_2)$ is a term of type β.

2.2.3. *Abstractions*

If e is a term of type β and $x \in \mathcal{V}_\alpha$, then the *abstraction* λxe is a term of type $(\alpha \to \beta)$.

The set of terms is therefore defined as the smallest superset of $\mathcal{A}$ closed by application and abstraction. We designate terms by $e, e', \ldots, E, E' \ldots$ perhaps with subscripts. The type of term e is denoted by $\tau(e)$.

We define the relation *subterm of* as the reflexive and transitive closure of:

$$\begin{cases} e_1 \text{ and } e_2 \text{ are subterms of } (e_1\, e_2) \\ e \text{ is a subterm of } \lambda xe. \end{cases}$$

2.3. *Conversion*

2.3.1. *Notations*

We use here the *context* notation, denoting by $\mathcal{C}[e]$ a term having a subterm e Then $\mathcal{C}[e']$ denotes the term obtained from it by replacing the distinguished occurrence of e by e' (if $\tau(e') = \tau(e)$).

Let $E = \mathcal{C}[\lambda xe]$. All the occurrences of x in λxe are said to be *bound in E.* Any non-bound occurrence of x in E is said to be *free in E.* We denote by $\mathcal{F}(E)$ the set of variables having a free occurrence in E. We denote by $\mathcal{S}_e^x(E)$, where $\tau(e) = \tau(x)$, the term obtained by substituting e for every free occurrence of x in E.

Finally, we define a relation $\mathcal{R}$ by:

$$\mathcal{R}(x, y, E) \Leftrightarrow \forall e\, (E = \mathcal{C}[\lambda ye] \Rightarrow \text{every occurrence of } x \text{ in } e \text{ is bound in } E),$$

where $\tau(x) = \tau(y)$.

The intuitive meaning of $\mathcal{R}(x, y, E)$ is that free x's can be replaced by terms containing y's in E without confusion of bound variables. For instance, $\mathcal{R}(x, y, \lambda yy)$, $\mathcal{R}(x, y, \lambda xy)$, $\mathcal{R}(x, y, \lambda x\lambda yx)$ and $\mathcal{R}(x, y, \lambda y\lambda xx)$, but $\neg\,\mathcal{R}(x, y, \lambda yx)$. Note that, when $\tau(x) = \tau(y)$, $\mathcal{R}(x, y, E)$ iff all the introduced occurrences of y in $\mathcal{S}_y^x(E)$ are free.

2.3.2. *Rules of λ-conversion*

We use two rules of λ-conversion: α-conversion which renames the bound variable in an abstraction, and β-reduction which is the evaluation rule of our λ-calculus, i.e. it replaces the formal argument by the actual one in an application. We shall use α-conversion only when needed to permit an application of the β rule.

α-conversion. Let $E = \mathcal{E}[\lambda xe]$. For any $y \notin \mathcal{F}(e)$ such that $\tau(y) = \tau(x)$ and $\mathcal{R}(x, y, e)$, we say that $\mathcal{E}[\lambda y\, \mathcal{S}_y^x(e)]$ follows from E by α-conversion.

β-reduction. Let $e = \mathcal{E}[(\lambda xe'E)]$. If $\forall y \in \mathcal{F}(E)$: $\mathcal{R}(x, y, e')$, then we say that $\mathcal{E}[\mathcal{S}_E^x(e')]$ follows from e by β-reduction.

λ-conversion is the transitive and reflexive closure of α-conversion and β-reduction. Note that it is a type-preserving transformation.

We shall not use for the moment a rule of η-conversion. However our results apply as well (and actually the unification algorithm is simplified) in the λ-β-η calculus. This will be explained in 4.5 below.

2.3.3. *Normal form*

A term is said to be in *normal form* iff it is not of the form $\mathcal{E}[(\lambda xe_1\, e_2)]$. Guard [1] and Sanchis have proved that for every term e there exists a term e' in normal form derivable from e by λ-conversion. By the Church–Rosser property this term e' is unique modulo α-conversion, and we call it *the normal form of e*.

Abbreviations. We shall take the usual n-ary notation for functions by using the following abbreviations:

$e_1(e_2, e_3, \dots, e_n)$ for $(\dots((e_1\, e_2)\, e_3) \dots e_n)$

$\lambda x_1\, x_2 \dots x_n \cdot e$ for $\lambda x_1\, \lambda x_2 \dots \lambda x_n\, e$ provided $x_1, x_2, \dots, x_n$ are distinct.

Any term in normal form e can thus be abbreviated (possibly via some α-conversions) into an expression of the form:

$$\lambda x_1\, x_2 \dots x_n \cdot @\, (e_1, e_2, \dots, e_p)$$

where:

$@ \in \mathcal{A}$ is called the *head* of e

$n \geqslant 0$; if $n = 0$ we suppress $\lambda\cdot$

$p \geqslant 0$; if $p = 0$ we suppress ( )

$\{x_1, \dots, x_n\}$ is a set of n distinct variables, called the *binder* of e

$\forall i \in [p]$ e_i is an expression of the same form, called an *argument* of e
$\forall i \in [n]$, $\forall j \in [p]$ x_i does not occur bound in e_j.

Note that our notation is unambiguous for terms in normal form. Thus $\lambda u \cdot u(v)$ abbreviates $\lambda u\,(uv)$ and not (λuuv), which would be β-reduced to v. When we use this notation for non-normal form terms, we shall use extra square brackets.

We shall regard as identical terms which differ only by renaming of their bound variables. Therefore from now on "$=$" between terms will denote the equivalence generated by α-conversion. For instance:

$$\lambda u \cdot A\,(\lambda v \cdot u, \lambda v \cdot w) = \lambda v \cdot A\,(\lambda w \cdot v, \lambda t \cdot w)$$
$$\text{but: } \lambda u \cdot A\,(v) \neq \lambda v \cdot A\,(v)$$
$$\text{and: } \quad f \neq \lambda x \cdot f(x).$$

Definition 2.1. We call *heading* of a term e in abbreviated normal form as above the term $\lambda x_1\, x_2 \ldots x_n \cdot @$. If $@ \in \mathcal{C} \cup \{x_1, .., x_n\}$, we say that e is *rigid*; otherwise, we say that e is *flexible*.

Finally, we denote by $\pi(e)$ the number of applications in e; i.e.

$$\pi\,(\lambda x_1 \ldots x_n \cdot @\,(e_1, \ldots, e_p)) = p + \sum_{i=1}^{p} \pi\,(e_i).$$

Similarly to the abbreviation for terms, we shall abbreviate type

$$(\alpha_1 \to (\alpha_2 \to \ldots (\alpha_n \to \beta)\ldots)) \text{ into } (\alpha_1, \alpha_2, \ldots, \alpha_n \to \beta),$$

where $\forall i \in [n]$ $\alpha_i \in T$ and $\beta \in T_0$.

2.4. *Substitutions*

A *substitution pair* is a couple $\langle x, e \rangle$ where $x \in \mathcal{V}$, $e \neq x$ and $\tau(x) = \tau(e)$. It is said to *pertain to* x. We suppose e reduced to normal form.

A *substitution* is a finite set of substitution pairs pertaining to distinct variables:

$$\sigma = \{\langle x_i, e_i \rangle \mid i \in [n]\} \qquad \forall i, j \in [n]\; x_i = x_j \Rightarrow i = j.$$

Let $\mathcal{T}$ denote the set of terms in normal form. Defining σx as

$$\begin{cases} e & \text{if } \langle x, e \rangle \in \sigma \\ x & \text{otherwise} \end{cases}$$

we can interpret a substitution as a type-preserving mapping from $\mathcal{V}$ to $\mathcal{T}$, equal to the identity almost everywhere. We extend it to a mapping from $\mathcal{T}$ to $\mathcal{T}$ as follows: the application of σ to E, written as σE, is defined as the normal form of the term: $[\lambda x_1 \ldots x_n \cdot E]\,(e_1, e_2, \ldots, e_n)$.

It is easy to show that this definition is independent of the order in which we take the pairs $\langle x_i, e_i \rangle$ in σ. For instance, if

$$\sigma = \{\langle x, F(y) \rangle, \langle y, G(x) \rangle\}, \text{ and } E = A\,(x, y), \text{ then}$$
$$\sigma E = [\lambda xy \cdot A\,(x, y)]\,(F(y), G(x))$$

$$= [\lambda uv \cdot A(u, v)](F(y), G(x))$$
$$= A(F(y), G(x)).$$

For complete proofs of properties of substitutions, see Huet [10].

We shall use Greek letters σ, ρ, η, ξ to designate substitutions and substitution pairs (we shall often confuse the pair $\langle x, e\rangle$ with the unit substitution $\{\langle x, e\rangle\}$).

If V is any set of variables, we denote by $\sigma\restriction V$ the restriction:

$$\{\langle x, e\rangle \mid \langle x, e\rangle \in \sigma \ \&\ x \in V\},$$

and we define an equivalence $\underset{V}{=}$ between substitutions by:

$$\sigma \underset{V}{=} \sigma' \Leftrightarrow \sigma\restriction V = \sigma'\restriction V.$$

We have the following easy property:

$$\forall \sigma, e \quad \sigma e = [\sigma\restriction \mathcal{F}(e)]\, e.$$

From which we get:

Lemma 2.2. *A rigid term keeps its heading unchanged under application of any substitution.*

The *composition* of two substitutions σ and ρ is defined as the substitution:

$$\rho\sigma = \{\langle x, \rho[\sigma x]\rangle \mid x \in \mathcal{V} \ \&\ \rho[\sigma x] \neq x\}.$$

This is precisely the composition of σ and ρ considered as mappings in $\mathcal{T} \to \mathcal{T}$ and it is therefore associative. We can thus dispense with parentheses, and write $\rho\sigma e$ and $\rho\sigma\eta$.

We say that σ is *less general than* ρ on V, and write $\sigma \underset{V}{\leqslant} \rho$, if and only if there exists η such that

$$\sigma \underset{V}{=} \eta\rho.$$

Finally, two substitutions ρ and ρ' are said to be *independent on V*, and we write $\rho \underset{V}{|} \rho'$, iff $\nexists \eta, \eta' : \eta\rho \underset{V}{=} \eta'\rho'$.

3. The unification algorithm

3.1. *Outline*

Definition 3.1. Let e_0 and e_0' be two terms of the same type. We call *unifier* of e_0 and e_0' any substitution σ such that $\sigma e_0 = \sigma e_0'$.

We are interested in the existence of unifiers for e_0 and e_0'. For this purpose we are going to construct a tree, called a *matching tree*, for e_0 and e_0'.

In first-order logic this problem is decidable, and the search for a most general unifier can also be represented as the growing of a tree of pairs of terms. This is basically an AND tree, and at every step the set of terminal leaves, (called the disagreement set), contains all the pairs of subterms that are yet to be matched. There are no OR nodes, because of the existence of at most one most general match at every leaf. This is not the case in typed λ-calculus, since for instance $f(A)$ and A can be unified by the two independent unifiers:

$$\sigma_1 = \{\langle f, \lambda u\cdot A\rangle\}, \qquad \sigma_2 = \{\langle f, \lambda u\cdot u\rangle\}.$$

This is why our matching trees will rather be OR trees, whose nodes are labelled with the current disagreement set. More formally:

We call *disagreement set* any finite set of pairs of terms of the same type:

$$\{\langle e_i^1, e_i^2\rangle \mid i \in [n]\} \quad \text{with } \forall i \in [n]: \tau(e_i^1) = \tau(e_i^2).$$

Such a set is said to be *reduced* if $n > 0$ and:

$$\forall i \in [n]: e_i^1 \text{ is flexible,} \quad \text{and} \quad \exists i \in [n]: e_i^2 \text{ is rigid.}$$

Our matching trees are trees whose nodes are either terminal nodes, labelled S (for success) or F (for failure) or non-terminal nodes, labelled with reduced disagreement sets, and linked to a finite number of successors by arcs labelled with substitution pairs.

Example 3.2. Let $e_0 = f(x, A)$, $e_0' = B$ with:

$$\tau(x) = \tau(A) = \tau(B) = \gamma, \qquad \tau(f) = (\gamma, \gamma \to \gamma).$$

Fig. 1 shows a matching tree for e_0 and e_0'.

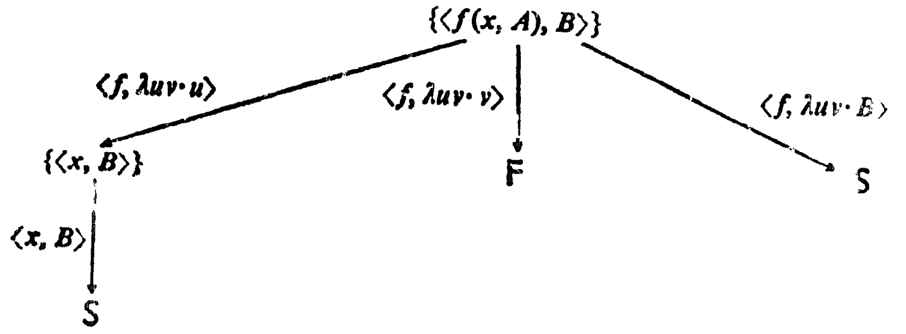

Fig. 1

From now on we shall identify a node with its associated disagreement set.

Definition 3.3. A substitution σ is said to unify a non-terminal node N iff it simultaneously unifies every pair in N. The set of unifiers of node N is denoted by $\mathcal{U}(N)$. $\mathcal{U}(\{\langle e_0, e_0'\rangle\})$ is abbreviated in $\mathcal{U}(e_0, e_0')$. Finally, we define:

$$\mathcal{F}(N) = \bigcup \{\mathcal{F}(e_1), \mathcal{F}(e_2) \mid \langle e_1, e_2\rangle \in N\}.$$

Our unification algorithm consists in growing progressively a matching tree, searching for a terminal success node.

3.2. *Construction of a matching tree*

The construction of a matching tree $\mathcal{M}$ for two terms e_0 and e'_0 of the same type is based on two procedures described below:
SIMPL, which takes as argument a disagreement set and returns as result a node, either terminal or not.
MATCH, which takes as arguments a flexible term e_1, a rigid term e_2 of the same type and a finite set of variables V; it returns a finite set Σ of substitution pairs.

The initial node in $\mathcal{M}$ is SIMPL $(\{\langle e_0, e'_0\rangle\})$.

Let N be the current node in $\mathcal{M}$. If N is S or F it has no successor. Otherwise, let us choose a pair $\langle e_1, e_2\rangle$ in N such that e_2 is rigid and let $\Sigma =$ MATCH $(e_1, e_2, \mathcal{F}(N))$. If $\Sigma = \emptyset$, replace N by F. Otherwise, for every σ in Σ, grow an arc, labelled with σ, from N to a new node:

$$N' = \text{SIMPL}\,(\{\langle \sigma e_1, \sigma e_2\rangle \mid \langle e_1, e_2\rangle \in N\}).$$

This construction is consistent with the properties of SIMPL and MATCH given above. It is non-deterministic because of the arbitrary choice of the pair given to MATCH. As we shall see, we do not need impose any way of making this choice, and furthermore all matching trees constructed as above are complete. This permits us to use any heuristic considerations in picking up a pair in the current node.

A unification algorithm can thus be described as any algorithm searching for a success node S in a matching tree, using level saturation for instance. Its completeness could be proved using our completeness theorem below and the fact that every matching tree is finitely generated. Again, heuristic considerations can be used to direct this search.

Note that our construction also applies to the simultaneous unification of any finite number of pairs of terms, by taking the (SIMPLified) set of these pairs as the initial node of the tree. It therefore applies also to the unification of a finite number of terms $e_1, e_2, \ldots, e_n$. For instance, unify the set of pairs $\{\langle e_1, e_2\rangle, \ldots, \langle e_1, e_n\rangle\}$.

3.3. *The procedure SIMPL*

3.3.1. *Algorithmic description of SIMPL(N)*

Step 1: If there does not exist in N a pair $\langle e_1, e_2\rangle$ such that e_1 and e_2 are both rigid then perform step 2, otherwise let us write:

$$N = \{\langle e_1, e_2\rangle\} \cup N', \text{ with:}$$
$$e_1 = \lambda u_1\, u_2 \ldots u_{n_1}\cdot @_1(e^1_1, \ldots, e^1_{p_1}) \quad n_1 \geqslant 0 \quad p_1 \geqslant 0,$$
$$e_2 = \lambda v_1\, v_2 \ldots v_{n_2}\cdot @_2(e^2_1, \ldots, e^2_{p_2}) \quad n_2 \geqslant 0 \quad p_2 \geqslant 0.$$

We now check that heading (e_1) = heading (e_2):

if $n_1 \neq n_2$ then return "F" else let $n = n_1 = n_2$,
if $@_1 \neq [\lambda v_1\, v_2 \ldots v_n \cdot @_2]\,(u_1, u_2, \ldots, u_n)$ then return "F",

otherwise let $p = p_1 = p_2$ [it has to be true since $\tau(e_1) = \tau(e_2)$]; then do:

$N \leftarrow \{\langle \tilde{e}_i^1, \tilde{e}_i^2 \rangle \mid i \in [p]\} \cup N'$, where:
$\tilde{e}_i^1 = \lambda u_1\, u_2 \ldots u_n \cdot e_i^1$,*,
$\tilde{e}_i^2 = \lambda v_1\, v_2 \ldots v_n \cdot e_i^2$; repeat step 1.

Step 2: Replace in N every pair $\langle e_1, e_2 \rangle$ where e_1 is rigid and e_2 is flexible, by the pair $\langle e_2, e_1 \rangle$; perform step 3.
Step 3: If there exists in N a pair $\langle e_1, e_2 \rangle$ such that e_2 is rigid then return N, else return "S"; end of SIMPL.

3.3.2. *Examples of applications of SIMPL(N)*

(1) $N = \{\langle A(\lambda u \cdot B(x, u), C), A(\lambda v \cdot B(y, v), f(C))\rangle\}$. The different values assumed by N are, successively:

step 1 $N = \{\langle \lambda u \cdot B(x, u), \lambda v \cdot B(y, v)\rangle, \langle C, f(C)\rangle\}$,
step 1 $N = \{\langle \lambda u \cdot x, \lambda v \cdot y\rangle, \langle \lambda u \cdot u, \lambda v \cdot v\rangle, \langle C, f(C)\rangle\}$,
step 1 $N = \{\langle \lambda u \cdot x, \lambda v \cdot y\rangle, \langle C, f(C)\rangle\}$,
step 2 $N = \{\langle \lambda u \cdot x, \lambda v \cdot y\rangle, \langle f(C), C\rangle\}$ which is returned in step 3.

(2) $N = \{\langle A(\lambda u \cdot B(x, u)), A(\lambda v \cdot B(y, v))\rangle\}$. After three applications of step 1 as above, we get $N = \{\langle \lambda u \cdot x, \lambda v \cdot y\rangle\}$, and after step 3 we return "S".

(3) $N = \{\langle \lambda uv \cdot A(u, \lambda w \cdot v), \lambda vw \cdot A(v, \lambda u \cdot v)\rangle\}$.
step 1 $N = \{\langle \lambda uv \cdot u, \lambda vw \cdot v\rangle, \langle \lambda uvw \cdot v, \lambda vwu \cdot v\rangle\}$.
step 1 $N = \{\langle \lambda uvw \cdot v, \lambda vwu \cdot v\rangle\}$.
step 1 we return "F".

3.3.3. *Correctness of SIMPL*

First we need to define a measure of complexity of nodes as follows. Let $N = \{\langle e_i^1, e_i^2 \rangle \mid i \in [n]\}$ be any disagreement set. We define

$$\Delta(N) = n + \sum_{i=1}^{n} \pi(e_i^1).$$

Let N be any node input to SIMPL. We shall prove that SIMPL will stop after a finite number of steps and will return as result a node N' such that:

Lemma 3.4. $N' = \text{"F"} \Rightarrow \mathcal{U}(N) = \emptyset$.

* If $e_i^1 = \lambda w_1 \ldots w_m \cdot @(\ldots)$ then $\tilde{e}_i^1 = \lambda u_1 \ldots u_n\, w_1 \ldots w_m \cdot @(\ldots)$ and by inspecting our definition on can check that every $\tilde{e}_i^1$ is going to be in reduced normal form assuming e_1 was: conflicts of variables cannot arise

Lemma 3.5. $N' =$ "S" $\Rightarrow \mathcal{U}(N) \neq \emptyset$.

Lemma 3.6. *N' not terminal* $\Rightarrow \mathcal{U}(N') = \mathcal{U}(N)$.

Proof of Lemmas 3.4. to 3.6. Let N_k be the set obtained after k successful applications of step 1, $k \geqslant 0$. We assume that $\mathcal{U}(N_k) = \mathcal{U}(N)$, which is trivially true for $k = 0$. If there exists a rigid-rigid pair in N_k, let $\langle e_1, e_2 \rangle$ be the pair selected. If e_1 and e_2 have distinct headings the algorithm stops with answer F, but then we know by Lemma 2.2 that $\mathcal{U}(N_k) = \emptyset$, and therefore $\mathcal{U}(N) = \emptyset$ which proves Lemma 3.4. Otherwise, let σ be any substitution, and let (using the notations of the algorithm):

$$\sigma_1 = \sigma\upharpoonright(\mathcal{V} - \{u_i | i \in [n]\})$$
$$\sigma_2 = \sigma\upharpoonright(\mathcal{V} - \{v_i | i \in [n]\}).$$

We can assume that $\forall x \in \mathcal{F}(e_1) \forall i \in [n]\ u_i \notin \mathcal{F}(\sigma_1 x)$. (Otherwise we rename u_i). Similarly, we assume that $\forall x \in \mathcal{F}(e_2) \forall i \in [n]\ v_i \notin \mathcal{F}(\sigma_2 x)$. Then:

$$\sigma e_1 = \lambda u_1 \ldots u_n \cdot @_1(\sigma_1 e_1^1, \ldots, \sigma_1 e_p^1)$$
$$\sigma e_2 = \lambda v_1 \ldots v_n \cdot @_2(\sigma_2 e_1^2, \ldots, \sigma_2 e_p^2).$$

Now

$$\sigma e_1 = \sigma e_2 \Leftrightarrow \forall i \in [p] : \lambda u_1 \ldots u_n \cdot [\sigma_1 e_i^1] = \lambda v_1 \ldots v_n \cdot [\sigma_2 e_i^2],$$

i.e. going in the reverse direction iff: $\sigma \tilde{e}_i^1 = \sigma \tilde{e}_i^2$. Since N_{k+1} differs only from N_k by the replacement of $\langle e_1, e_2 \rangle$ by the pairs $\langle \tilde{e}_i^1, \tilde{e}_i^2 \rangle$, this proves that $\mathcal{U}(N_{k+1}) = \mathcal{U}(N_k) = \mathcal{U}(N)$. Furthermore, we check that $\Delta(N_{k+1}) = \Delta(N_k) - 1$, which proves that we shall stop after a finite number of applications of step 1. If F has not been returned, we shall enter step 2 with a set N_k such that $\mathcal{U}(N_k) = \mathcal{U}(N)$. Since unification is symmetric this property is preserved by step 2, and this proves Lemma 3.6. To conclude Lemma 3.5 it is left to show that a node containing only flexible-flexible pairs is always unifiable.

Notation. For every elementary type $\alpha \in T_0$, let us choose a variable $h_\alpha \in \mathcal{V}_\alpha$. For every type $\alpha \in T$, we construct a term $\hat{E}_\alpha$ of type α by:

$$-\text{if } \alpha \in T_0 \quad \hat{E}_\alpha = h_\alpha$$

otherwise, writing $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n \to \beta)$ with $\beta \in T_0$, we take $\hat{E}_\alpha = \lambda w_1 \ldots w_n \cdot h_\beta$ where the w_i's are distinct variables different from h_β such that $\tau(w_i) = \alpha_i$.

Now we define $\zeta = \{\langle x, \hat{E}_{\tau(x)} \rangle | x \in \mathcal{V}\}$. ζ is a generalized (since infinite) substitution.

Let N be any disagreement set containing only flexible-flexible pairs. We prove that the substitution $\zeta_N = \zeta \upharpoonright \mathcal{F}(N)$ unifies N.

Proof. Let us consider any pair $\langle e_1, e_2 \rangle$ in N. Let us write:

$$e_1 = \lambda u_1 \dots u_{m_1} \cdot f(E_1^1, E_2^1, \dots, E_r^1)$$
$$e_2 = \lambda v_1 \dots v_{m_2} \cdot g\,(E_2^2, E_2^2, \dots, E_s^1).$$

We get:

$$\zeta_N\, e_1 = \lambda u_1 \dots u_{m_1}\, w_{r+1} \dots w_{n_1} \cdot h_{\beta_1} \quad \text{(assuming } u_i \neq h_{\beta_1}\ \forall i \in [m_1])$$
$$\zeta_N\, e_2 = \lambda v_1 \dots v_{m_2}\, w'_{s+1} \dots w'_{n_2} \cdot h_{\beta_2} \quad \text{(assuming } v_i \neq h_{\beta_2}\ \forall i \in [m_2])$$

and since $\tau(e_1) = \tau(e_2)$, we must have

$$m_1 + n_1 - r = m_2 + n_2 - s$$
$$\tau(u_1) = \tau(v_1), \dots, \tau(w_{n_1}) = \tau(w'_{n_2})$$
$$\beta_1 = \tau(h_{\beta_1}) = \tau(h_{\beta_2}) = \beta_2$$

and therefore $\zeta_N\, e_1 = \zeta_N\, e_2$. □

This concludes the proof of correctness of SIMPL. Note that we are able to give explicitly a unifier for any success node; however, this node will represent in general a whole set of unifiers. We shall come back to this remark in 4.3 below.

3.4. *The procedure MATCH*

We shall present MATCH in a loose algorithmic fashion, with many comments to explain and justify every step.

3.4.1. *Algorithmic description of MATCH(e_1, e_2, V)*

e_1 is a flexible term and e_2 a rigid term of the same type. V is a finite set of variables The motivation behind V is purely technical. V will contain all the free variables in the node in which the pair $\langle e_1, e_2 \rangle$ is selected, so as to avoid conflicts when choosing new variables. Let us write:

$$e_1 = \lambda u_1 \dots u_{n_1} \cdot f(e_1^1, e_2^1, \dots, e_{p_1}^1) \qquad n_1 \geqslant 0, \qquad p_1 \geqslant 0,$$
$$e_2 = \lambda v_1 \dots v_{n_2} \cdot @\,(e_1^2, e_2^2, \dots, e_{p_2}^2) \qquad n_2 \geqslant 0, \qquad p_2 \geqslant 0,$$

with $\tau(f) = (\alpha_1, \alpha_2, \dots, \alpha_{p_1}, \alpha_{p_1+1}, \dots, \alpha_{q_1} \to \beta)$, $q_1 \geqslant p_1$.

Since e_2 is rigid, its heading cannot be changed by substitution, whereas the heading of e_1 may be adjusted to that of e_2 under certain conditions. First, to adjust its binder, we need have $n_1 \leqslant n_2$, since the binder of a term can only increase in length by substitution. Therefore, if $n_1 > n_2$ we return $\Sigma = \emptyset$; else let $n = n_2 - n_1 \geqslant 0$.

Let us introduce p_1 variables $w_1, \dots, w_{p_1}$ such that $\forall i \in [p_1]\ \tau(w_i) = \alpha_i$. In order to avoid conflicts, we impose $w_i \notin \{v_{n_1+1}, \dots, v_{n_2}\}$.

To adjust the heading of e_1, we shall consider substitutions for f of a term $\lambda z_1 \dots z_r \cdot @'(\dots)$. The head $@'$ will be either $@$, and the most general such substitution will be given by the rule of *imitation* below, or z_k, $1 \leqslant k \leqslant r$, which corresponds to the rule of *projection* below.

As we shall see it is sufficient to consider these two cases. The result Σ will be the union of the results obtained using the two rules. In each case the possible values of r are found by type considerations.

3.4.1.1. *Imitation rule.* We want to "imitate" e_2 by e_1, substituting for f a term with head @. If @ is some v_i with $i \in [n_1]$, it will not be possible to introduce directly the corresponding v_i by substitution for f, since it is protected by the binder of e_1. It is only possible to introduce it indirectly if it appears in one of the arguments of e_1, and then the rule of projection will cover this case. So we limit the application of the rule of imitation to the case where $@ \in \mathcal{C}$ or $@ = v_i$ with $n_1 < i \leqslant n_2$.

(i) $n > 0$. This determines completely the heading of the term we must substitute for f. The rest of the term is filled in the most general way, by introducing new variables. Precisely, we return as unique solution in this case:

$$\sigma = \langle f, \lambda w_1 \dots w_{p_1} v_{n_1+1} \dots v_{n_2} \cdot @ (E_1, E_2, \dots, E_{p_2})\rangle,$$

where $E_i = h_i(w_1, \dots, w_{p_1}, v_{n_1+1}, \dots, v_{n_2})$ $i \in [p_2]$, the h_i's being distinct variables of the appropriate type not in V. (Note: no conflict may arise, even if some v_k $(n_1 < k \leqslant n_2)$ appears free in e_1; also there is no risk of conflict of h_i with some w_j or v_k because they have different types.)

(ii) $n = 0$. Considering the remark above, we must have here $@ \in \mathcal{C}$. The heading of the term substituted for f is going to be some $\lambda w_1 \dots w_k \cdot @$, with $0 \leqslant k \leqslant p_1$. However, any such k will not do, because we have a type condition on the remaining arguments of e_1. More precisely, we must be able to complete the term with a number of arguments $= p_2 - (p_1 - k) \geqslant 0$ which gives the first condition:

(1) $\quad \max(0, p_1 - p_2) \leqslant k \leqslant p_1.$

Also, the unchanged argument of e_1 must be type-compatible with the ones of e_2:

(2) $\quad \alpha_j = \tau(e^2_{p_2 - p_1 + j}) \quad \forall j \, (k < j \leqslant p_1).$

Conversely, these conditions are sufficient for the substitution below to be legitimate. Therefore, for every k satisfying (1) and (2), we include in Σ

$$\sigma = \langle f, \lambda w_1 \dots w_k \cdot @ (E_1, E_2, \dots, E_{p_2 - p_1 + k})\rangle,$$

where $E_i = h_i(w_1, \dots, w_k)$, $i \in [p_2 - p_1 + k]$, the h_i's being distinct variables of the appropriate type not in V. In this subcase, we have therefore at most $\min(p_1, p_2) + 1$ solutions. Note that conditions (1) and (2) are always satisfied with $k = p_1$, which guarantees at least one solution. But we cannot restrict ourselves to this case unless we admit the η-reduction rule:

$$\mathcal{E}[\lambda u \cdot e(u)] = \mathcal{E}[e] \quad \text{if } u \notin \mathcal{F}(e).$$

For instance, consider $e_1 = f(B(A))$ and $e_2 = A(B(f))$, with $\tau(f) = \tau(A) = (\alpha \to \alpha)$, $\tau(B) = ((\alpha \to \alpha) \to \alpha)$. The unique unifier of e_1 and e_2 is $\{\langle f, A\rangle\}$, corresponding to $p_1 = p_2 = 1$, $k = 0$.

3.4.1.2. *Projection rule.* We want to project f on one of its arguments. The possible headings for the term we may substitute for f are:

(i) $\lambda w_1 \dots w_k \cdot w_i$ $\quad k \in [p_1], \quad i \in [k]$,
(ii) $\lambda w_1 \dots w_{p_1} v_{n_1+1} \dots v_{n_1+k} \cdot w_i$ $\quad k \in [n], \quad i \in [p_1]$,
(iii) $\lambda w_1 \dots w_{p_1} v_{n_1+1} \dots v_{n_1+k} \cdot v_{n_1+i}$ $\quad k \in [n], \quad i \in [k]$.

However in case (iii), after substitution of the term for f, we would have to unify

$$\sigma e_1 = \lambda u_1 \dots u_{n_1} v'_{n_1+1} \dots v'_{n_1+k} \cdot v'_{n_1+i}(\dots)$$

with

$$\sigma e_2 = \lambda v_1 \dots v_{n_1} v_{n_1+1} \dots v_{n_2} \cdot @ (\dots),$$

and this will be rejected by SIMPL unless

$$k = n \text{ and } @ = v_{n_1+i},$$

which case we have already considered in 3.4.1.1(i), and so we can limit ourselves to cases (i) and (ii). We shall include in Σ the union of their solutions.

(i) Heading $\lambda w_1 \dots w_k \cdot w_i$

1) $\quad k \in [p_1], i \in [k], \quad$ (i.e. $1 \leqslant i \leqslant k \leqslant p_1$).

Here we have a supplementary condition on the type of w_i, so that the term we construct be of the same type as f. Precisely, there must exist $m \geqslant 0$ such that:

(2) $\quad \alpha_i = (\gamma_1, \gamma_2, \dots, \gamma_m, \alpha_{k+1}, \dots, \alpha_{q_1} \to \beta)$

for some $\gamma_1, \dots, \gamma_m \in T$. (This condition is satisfied by $m = 0$ if $k = q_1$ and $\alpha_i = \beta$.)

For each i and k satisfying (1) and (2), we take as solution in Σ:

$$\sigma = \langle f, \lambda w_1 \dots w_k \cdot w_i(E_1, E_2, \dots, E_m)\rangle$$

where $E_j = h_j(w_1, \dots, w_k), j \in [m]$, the h_j's being distinct variables of the appropriate type not in V.

Note that, given i and k, m is completely determined from (2). This gives us at most $p_1(p_1+1)/2$ solutions.

(ii) Heading $\lambda w_1 \dots w_{p_1} v_{n_1+1} \dots v_{n_1+k} \cdot w_i$. This case is very similar to the previous one, changes are just notational.

(1) $\quad k \in [n], i \in [p_1]$

and similarly:

(2) $\quad \alpha_i = (\gamma_1, \gamma_2, \dots, \gamma_m, \alpha_{p_1+k+1}, \dots, \alpha_{q_1} \to \beta)$.

(As above, this condition is satisfied by $m = 0$ if $p_1 + k = q_1$ and $\alpha_i = \beta$.)

The solutions in this case are all the:

$$\sigma = \langle f, \lambda w_1 \dots w_{p_1} v_{n_1+1} \dots v_{n_1+k} \cdot w_i(E_1, E_2, \dots, E_m)\rangle$$

where $E_j = h_j(w_1, \dots, w_{p_1}, v_{n_1+1}, \dots, v_{n_1+k})$, $j \in [m]$, the h_j's being distinct variables of the appropriate type not in V. Here we have at most $n \times p_1$ solutions.

This completes our analysis of the projection rule. Note that we have at most $p_1(p_1+2n+1)/2$ solutions. In practice of course we shall get fewer solutions because of the conditions (2). End of MATCH.

3.4.2. *Example of application of MATCH*

Let $e_1 = f(x, B)$, $e_2 = A(B)$, with

$$\tau(f) \to ((\gamma, \gamma \to \gamma), \gamma \to \gamma),\ \tau(x) = (\gamma, \gamma \to \gamma),\ \tau(B) = \gamma,\ \tau(A) = (\gamma \to \gamma).$$

With the notations of the algorithm, we have:

$$\alpha_1 = (\gamma, \gamma \to \gamma),\quad \alpha_2 = \gamma,\quad \beta = \gamma,\quad n_1 = n_2 = 0,\quad p_1 = q_1 = 2,\quad p_2 = 1.$$

We call MATCH $(e_1, e_2, \{x, f\})$.

3.4.2.1. *Imitation.* $n = n_2 - n_1 = 0$ and so we are in case 3.4.1.1(ii). $p_1 - p_2 = 1 \leqslant k \leqslant p_1 = 2$, so we have two choices:

•$k = 1$ for which condition (2): $\alpha_2 = \tau(B) = \gamma$ holds, and we take $\sigma_1 = \langle f, \lambda w_1 \cdot A\rangle$; note that x will be eliminated here.

•$k = 2$ for which condition (2) is vacuous, and we take

$$\sigma_2 = \langle f, \lambda w_1\, w_2 \cdot A\,(h_1(w_1, w_2))\rangle$$

with $\tau(h_1) = ((\gamma, \gamma \to \gamma), \gamma \to \gamma)$.

3.4.2.2. *Projection.* Since $n = 0$ we consider only case 3.4.1.2(i), $1 \leqslant k \leqslant 2$. With $k = 1$, condition (1) forces $i = 1$, and condition (2) is satisfied for $m = 1$, which gives the solution:

$$\sigma_3 = \langle f, \lambda w_1 \cdot w_1(h_1(w_1))\rangle, \text{ with } \tau(h_1) = ((\gamma, \gamma \to \gamma) \to \gamma).$$

The h_1 here is of course distinct from the one in σ_2, since they have a different type. But the only constraint we impose on the "new variables" h_i is that they should not appear free in the node of the matching tree from which we call MATCH, and so we could use the same "new" variables for distinct successors of a node. With $k = 2$, we have two choices for i:

•$i = 1$; then $m = 2$ and we take:

$$\sigma_4 = \langle f, \lambda w_1\, w_2 \cdot w_1(h_1(w_1, w_2), h_2(w_1, w_2))\rangle,$$

with $\tau(h_1) = \tau(h_2) = \tau(f)$.

•$i = 2$; then $m = 0$ and we take:

$$\sigma_5 = \langle f, \lambda w_1\, w_2 \cdot w_2\rangle.$$

The reader will check that σ_1 to σ_5 are independent substitutions. If the node from which we called MATCH was $N = \{\langle e_1, e_2\rangle\}$, then its corresponding successors would be:

$N_1 = S$
$N_2 = \{\langle h_1(x, B), B\rangle\}$
$N_3 = \{\langle x\,(h_1(x), B), A\,(B)\rangle\}$

$N_4 = \{\langle x\,(h_1(x, B), h_2(x, B)), A\,(B)\rangle\}$
$N_5 = \mathrm{F}.$

3.4.3. *Correctness of MATCH*
Let us first introduce a complexity measure for substitutions. Let $\xi = \{\langle u_i, e_i\rangle \mid i \in [r]\}$ be any substitution. We define:

$$\theta(\xi) = r + \sum_{i=1}^{r} \pi(e_i).$$

Lemma 3.7. *For any terms e_1 and e_2 of the same type, where e_1 is flexible and e_2 rigid, and for any finite set V of variables, if there exists $\rho \in \mathcal{U}(e_1, e_2)$, then MATCH (e_1, e_2, V) returns a non-empty set Σ in which there is a unique substitution pair σ such that $\rho \underset{V}{\leqslant} \sigma$. Moreover we can find an η such that $\rho \underset{V}{=} \eta\sigma$ with $\theta(\eta) < \theta(\rho)$.*

Proof. Let f be the head of e_1, @ that of e_2. If ρ unifies e_1 and e_2, there must exist in ρ a pair pertaining to f. Let us write:

$$\rho f = \lambda z_1 \dots z_k \cdot @'(E'_1, \dots, E'_l)$$

and we define $\bar{\rho} = \rho \restriction (V - \{f\})$.

First case. $@'$ is @, or some z_i ($i \in [k]$). We saw that we considered in MATCH all possible cases of such headings, which means that there exists in Σ = MATCH (e_1, e_2, V) some:

$$\sigma = \langle f, \lambda w_1\, w_2 \dots w_k \cdot @''(E_1, \dots, E_l)\rangle,$$

where:

$$\tau(w_i) = \tau(z_i) \quad \forall i \in [k]$$
$$@'' = [\lambda z_1 \dots z_k \cdot @']\,(w_1, \dots, w_k)$$
$$E_j = h_j(w_1, \dots, w_k) \quad h_j \notin V \quad \forall j \in [l].$$

Moreover, such a σ is unique, since no two terms constructed in MATCH have the same heading (note the remark concerning case (iii) in 3.4.1.2). Now let

$$\eta = \{\langle h_j, \lambda z_1 \dots z_k \cdot E'_j\rangle \mid j \in [l]\} \cup \bar{\rho};$$

η is a legitimate substitution, and we check that: $\rho \underset{V}{=} \eta\sigma$.
Finally:

$$\theta(\eta) = \theta(\bar{\rho}) + \sum_{j=1}^{l} \pi(\lambda z_1 \dots z_k \cdot E'_j) + l = \theta(\rho \restriction V) - 1 < \theta(\rho).$$

Second case. $@'$ is neither @ nor some z_i. Then, we get:

$$\rho e_1 = \begin{cases} \lambda u_1 \dots u_{n_1} \cdot @''(\dots) & \text{if } k \leqslant p_1, \\ \lambda u_1 \dots u_{n_1}\, z'_{p_1+1} \dots z'_k \cdot @''(\dots) & \text{if } k > p_1, \end{cases}$$

where $@''$ is distinct from @, u_j and z'_i ($j \in [n_1], p_1 < i \leqslant k$) and therefore cannot be equal to $\rho e_2 = \lambda v_1 \dots v_{n_2} \cdot @\,(\dots)$. This case will never arise, which shows that the two rules of imitation and projection are sufficient for our purpose. This concludes the proof of Lemma 3.7. □

In the case of interest to us, $V = \mathcal{F}(N)$, where $\langle e_1, e_2\rangle \in N$ and therefore we can conclude that $\eta\sigma$ unifies e_1 and e_2.

Actually, the minimal V would be $\mathcal{F}(N) - \{f\}$, but in practical implementations it is simpler to have MATCH generate totally new variables for the h_i's (using the GENSYM operator of LISP, for instance).

3.5. *Examples of matching trees*

3.5.1.

$$e_0 = f(f(x)),\ e_0' = A(A(B)) \quad \text{with } \tau(x) = \tau(B) = \gamma,$$
$$\tau(f) = \tau(A) = (\gamma \to \gamma).$$

Here we get a unique matching tree, and it is finite. See Fig. 2.

$$\begin{tikzcd}
 & & & \{\langle f(f(x)), A(A(B))\rangle\} \arrow[dlll, "\langle f, A\rangle"'] \arrow[d, "\langle f, \lambda u\cdot A(h(u))\rangle"] \arrow[drrr, "\langle f, \lambda u\cdot u\rangle"] & & & \\
\{\langle x, B\rangle\} \arrow[d, "\langle x, B\rangle"'] & & & \{\langle h(A(h(x))), A(B)\rangle\} \arrow[dl, "\langle h, \lambda u\cdot A(h'(u))\rangle"'] \arrow[d, "\langle h, A\rangle"'] \arrow[dr, "\langle h, \lambda u\cdot u\rangle"] & & & \{\langle x, A(A(B))\rangle\} \arrow[d, "\langle x, A(y)\rangle"] \\
\mathsf{S} & & \{\langle h'(A(A(h'(x)))), B\rangle\} \arrow[dl, "\langle h', \lambda u\cdot B\rangle"'] \arrow[dr, "\langle h', \lambda u\cdot u\rangle"] & \mathsf{F} & \{\langle x, B\rangle\} \arrow[d, "\langle x, B\rangle"'] & & \{\langle y, A(B)\rangle\} \arrow[d, "\langle y, A(z)\rangle"] \\
 & \mathsf{S} & & \mathsf{F} & \mathsf{S} & & \{\langle z, B\rangle\} \arrow[d, "\langle z, B\rangle"] \\
 & & & & & & {}
\end{tikzcd}$$

Fig. 2

This is the same as Example 1 in Pietrzykowski [15]. However, note that we get more solutions because here we do not have the η-reduction rule. We shall come back to this in 4.5.

3.5.2.

$$e_0 = A(y, x(B)), \quad e_0' = A(x(\lambda u\, y), B(y)),$$

with

$$\tau(y) = \tau(u) = \gamma, \quad \tau(B) = (\gamma \to \gamma), \quad \tau(x) = ((\gamma \to \gamma) \to \gamma),$$
$$\tau(A) = (\gamma, \gamma \to \gamma).$$

Here we get a unique infinite matching tree, with a unique success node at level 1; the pair input to MATCH is underlined in its node. See Fig. 3.

$$\begin{tikzcd}[column sep=small]
 & \{\langle y, x(\lambda u\cdot y)\rangle, \underline{\langle x(B), B(y)\rangle}\} \arrow[dl, "{\langle x, \lambda w\cdot B(h(w))\rangle}"'] \arrow[dr, "{\langle x, \lambda w\cdot w(h(w))\rangle}"] & \\
\{\underline{\langle y, B(h(\lambda u\cdot y))\rangle}, \langle h(B), y\rangle\} \arrow[d, "{\langle y, B(z)\rangle}"] & & S \\
\{\langle z, h(\lambda u\cdot B(z))\rangle, \underline{\langle h(B), B(z)\rangle}\} \arrow[d, "{\langle h, \lambda w\cdot B(h'(w))\rangle}"'] \arrow[dr, "{\langle h, \lambda w\cdot w(h'(w))\rangle}"] & & \\
\{\underline{\langle z, B(h'(\lambda u\cdot B(z)))\rangle}, \langle h'(B), z\rangle\} \arrow[d] & \{\underline{\langle z, B(z)\rangle}, \langle h'(B), z\rangle\} \arrow[d] & \\
\text{these branches repeat} & \text{indefinitely similar nodes} &
\end{tikzcd}$$

Fig. 3

Note that the success node is obtained directly from SIMPL applied to the se $\{\langle y, y\rangle, \langle B(h(B)), B(y)\rangle\}$. Note also that, in the nodes with two pairs, only one of these pairs is flexible-rigid, but which pair it is keeps alternating.

3.5.3.

$$e_0 = x, \quad e_0' = \lambda u\cdot u(\lambda v\cdot w, x(e), x(f)) \quad \text{where } e = \lambda u'v'w'\cdot A(u'(v'), w'),$$

with:

$$\tau(u) = \tau(f) = ((\gamma \to \gamma), \gamma, \gamma \to \gamma), \quad \tau(x) = (((\gamma \to \gamma), \gamma, \gamma \to \gamma) \to \gamma),$$
$$\tau(u') = (\gamma \to \gamma), \tau(v) = \tau(w) = \tau(v') = \gamma, \quad \tau(A) = (\gamma, \gamma \to \gamma).$$

Here we get a unique finite tree. See Fig. 4. Note that the first substitution step is an imitation, not a projection.

$$\begin{tikzcd}[column sep=tiny]
 & \{\langle x, \lambda u\cdot u(\lambda v\cdot w, x(e), x(f))\rangle\} \arrow[d, "{\langle x, \lambda u\cdot u(h_1(u), h_2(u), h_3(u))\rangle}"'] & \\
 & \{\langle \lambda u\cdot h_1(u), \lambda uv\cdot w\rangle, \underline{\langle \lambda u\cdot h_2(u), \lambda u\cdot A(h_1(e, h_2(e)), h_3(e))\rangle}, \langle \lambda u\cdot h_3(u), \lambda u\cdot f(h_1(f), h_2(f), h_3(f))\rangle\} \arrow[dl, "{\langle h_2, \lambda u\cdot A(h_4(u), h_5(u))\rangle}"'] \arrow[dr, "{\langle h_2, \lambda u\cdot u(h_6(u), h_7(u), h_8(u))\rangle}"] & \\
S & & F
\end{tikzcd}$$

Fig. 4

We leave to the reader to check that the following substitution, obtained by the method in 3.3.3 composed with the substitutions on the path to the success node, is indeed a unifier of e_0 and e_0':

$$\sigma = \{\langle x, \lambda u \cdot u\,(\lambda v \cdot z, A\,(z, z), z)\rangle, \langle f, \lambda u' v' w' \cdot z\rangle, \langle w, z\rangle\}.$$

This example exhibits a good convergence. It demonstrates that the new variables introduced by MATCH (the h_i's) do not complicate the unification search, since they are never considered in "don't care" situations.

4. Correctness of the unification process

We shall now prove that our unification process is correct. First we prove a soundness theorem, stating that if any matching tree for e_0 and e_0' possesses a success node then e_0 and e_0' are unifiable. Then we prove a completeness theorem, stating that if e_0 and e_0' are unifiable, then every matching tree for e_0 and e_0' prossesses a success node at a finite level.

4.1. *Soundness*

Let us suppose that we have constructed a matching tree for e_0 and e_0' which possesses a success node on some branch:

$$N_0 \overset{\sigma_1}{\rightarrow} N_1 \overset{\sigma_2}{\rightarrow} \dots \overset{\sigma_p}{\rightarrow} N_p = \mathrm{S} \qquad p \geqslant 0.$$

Let ξ_p be defined as a substitution which unifies N_p, as constructed in the proof of Lemma 3.5 above: $\xi_p = \zeta_{N_p}$.* Now, we define:

$$\xi_i = \xi_{i+1}\,\sigma_{i+1} \quad p > i \geqslant 0,$$

and prove by (descending) induction on i that ξ_i unifies N_i. The base step has been shown in Lemma 3.5. For the induction step, assume that ξ_{i+1} unifies N_{i+1}. Let $\bar{N}_{i+1} = \{\langle \sigma_{i+1}\,e_1, \sigma_{i+1}\,e_2\rangle \mid \langle e_1, e_2\rangle \in N_i\}$. By induction hypothesis and Lemma 3.6, since $N_{i+1} = \mathrm{SIMPL}\,(\bar{N}_{i+1})$, ξ_{i+1} unifies $\bar{N}_{i+1}$; that is, for every $\langle e_1, e_2\rangle \in N_i$:

$$\xi_{i+1}\,\sigma_{i+1}\,e_1 = \xi_{i+1}\,\sigma_{i+1}\,e_2,$$

i.e. ξ_i unifies N_i, which concludes the induction step.

As a particular case, we get that $\xi_0 = \xi_p\,\sigma_p\,\sigma_{p-1} \dots \sigma_1$ unifies $N_0 = \mathrm{SIMPL}\,(\{\langle e_0, e_0'\rangle\})$ and therefore by Lemma 3.6 again, ξ_0 unifies e_0 and e_0'. Whence, defining $\sigma_{N_p} = \sigma_p\,\sigma_{p-1} \dots \sigma_1$, we get:

* Strictly speaking, ζ_{N_p} should be written ζ_N, where N is the disagreement set from which N_p derives: $N_p = \mathrm{SIMPL}\ (N)$.

Theorem 4.1. *Let $\mathcal{M}$ be a matching tree for e_0 and e_0'. For any success node N in $\mathcal{M}$, $\zeta_N \sigma_N$ unifies e_0 and e_0'.*

4.2. *Completeness*

Let us assume that terms e_0 and e_0' are unifiable by substitution ρ. Let us consider an arbitrary matching tree for e_0 and e_0'. We shall construct a branch

$$N_0 \xrightarrow{\sigma_1} N_1 \xrightarrow{\sigma_2} \dots \xrightarrow{\sigma_i} N_i$$

of this tree, together with a sequence of substitutions $\xi_0, \xi_1, \dots, \xi_i$ such that

$$\forall i \geqslant 0 \quad \xi_i \in \mathcal{U}(N_i) \text{ or } N_i = \text{S},$$

and of complexities strictly decreasing. We prove this by (ascending) induction on i. The base step is obvious, taking $\xi_0 = \rho$, since if $N_0 = \text{SIMPL}(\{\langle e_0, e_0'\rangle\}) \neq \text{S}$ then by Lemma 3.6 $\rho \in \mathcal{U}(N_0)$.

For the induction step, we assume that $\xi_i \in \mathcal{U}(N_i)$. Now either $N_i = \text{S}$, in which case we stop the construction, or in N_i we are considering a pair $\langle e_1, e_2\rangle$ such that e_2 is rigid. N_i cannot be a failure node by Lemma 3.4, since by hypothesis $\xi_i \in \mathcal{U}(N_i)$. Using Lemma 3.7, we know that MATCH $(e_1, e_2, \mathcal{F}(N_i))$ returns some σ such that there exists η with

$$\xi_i \underset{\mathcal{F}(N_i)}{=} \eta\sigma,$$

and therefore there exists a branch $N_i \xrightarrow{\sigma} N'$ in the tree.

Choose: $\quad \xi_{i+1} = \eta, \quad \sigma_{i+1} = \sigma, \quad N_{i+1} = N'$.

By construction: $\quad N_{i+1} = \text{SIMPL}(\overline{N}_{i+1})$,

where $\quad \overline{N}_{i+1} = \{\langle \sigma_{i+1} e_1, \sigma_{i+1} e_2\rangle \mid \langle e_1, e_2\rangle \in N_i\}$.

For every $\langle e_1, e_2\rangle$ in N_i: $\xi_i e_1 = \xi_i e_2$ by hypothesis, and therefore

$$\xi_{i+1} \sigma_{i+1} e_1 = \xi_{i+1} \sigma_{i+1} e_2,$$

since $\mathcal{F}(N_i)$ contains all the free variables of e_1 and e_2. This shows that ξ_{i+1} unifies $\overline{N}_{i+1}$ and thus either $N_{i+1} = \text{S}$, or ξ_{i+1} unifies N_{i+1} by Lemma 3.6. Finally, we know by Lemma 3.7 that $\theta(\xi_{i+1}) < \theta(\xi_i)$, which concludes the induction step. Thus our construction will generate a success node S at a level at most equal to $\theta(\rho)$. This completes the proof of our completeness theorem:

Theorem 4.2. *If $\mathcal{U}(e_0, e_0') \neq \emptyset$, then every matching tree for e_0 and e_0' possesses a success node at a finite level.*

Of course if $\mathcal{U}(e_0, e_0') = \emptyset$ then we may build an infinite tree. This may happen even for first-order pairs such as $\langle x, A(x)\rangle$, which we could easily recognize as failure nodes. We shall discuss in Section 5.3 a few heuristics for such special cases.

4.3. *Complete unification*

Theorem 4.1 leads us to wonder whether our algorithm could be used to find all unifiers of e_0 and e'_0, by enumerating all success nodes in a matching tree.

Definition 4.3. *A complete set of unifiers* (CSU) *of e_0 and e'_0 on V* is any set Σ of substitutions such that:

1) $\Sigma \subset \mathcal{U}(e_0, e'_0)$;

2) $\forall \rho \in \mathcal{U}(e_0, e'_0)$, $\exists \sigma \in \Sigma$: $\rho \underset{V}{\leqslant} \sigma$, where V is any finite set of variables containing $\mathcal{F}(e_0) \cup \mathcal{F}(e'_0)$.

For instance, if one wants to use Σ to compute resolvents [18, 15] of literals e_0 and $\neg e'_0$ in normalized clauses C and C', then we ought to take $V = \mathcal{F}(C) \cup \mathcal{F}(C')$. From now on we consider V given, and containing $\mathcal{F}(e_0) \cup \mathcal{F}(e'_0)$.

Now let us say a *complete matching tree for e_0 and e'_0 on V* is a matching tree $\mathcal{M}$ constructed as above except that the third argument given to MATCH at node N is now V_N, defined recursively as:

$$V_N = \begin{cases} V & \text{if } N = N_0 \\ V_{N'} \cup \mathcal{F}(e) & \text{if } N' \xrightarrow{\langle x, e\rangle} N \text{ in } \mathcal{M}, \end{cases}$$

i.e. V_N contains V and all new variables introduced on the branch leading to N. This rules out such branches as:

$$\{\langle f(x, y), A(B)\rangle\} \xrightarrow{\langle f, \lambda uv \cdot u\rangle} \{\langle x, A(B)\rangle\} \xrightarrow{\langle x, A(y)\rangle} \{\langle y, B\rangle\} \xrightarrow{\langle y, B\rangle} \mathrm{S}$$
$$\{\langle f(\lambda u \cdot y), A(B)\rangle\} \xrightarrow{\langle f, \lambda x \cdot x(h(x))\rangle} \{\langle y, A(B)\rangle\} \xrightarrow{\langle y, A(h)\rangle} \{\langle h, B\rangle\} \xrightarrow{\langle h, B\rangle} \mathrm{S}.$$

For instance, in the second example, where types are:

$$\tau(u) = \alpha, \quad \tau(y) = \beta, \quad \tau(f) = ((\alpha \to \beta) \to \beta),$$
$$\tau(B) = \tau(h) = ((\alpha \to \beta) \to \alpha), \quad \tau(A) = (((\alpha \to \beta) \to \alpha) \to \beta),$$

the variable h is used in two independent occasions, and the final unifier obtained,

$$\sigma_N \upharpoonright \{f, y\} = \{\langle f, \lambda x \cdot x(B(x))\rangle, \langle y, A(B)\rangle\},$$

is not as general as

$$\{\langle f, \lambda x \cdot x(h(x))\rangle, \langle y, A(B)\rangle\},$$

which we obtain if we construct a complete matching tree on $\{f, y\}$.

Note that the trivial implementation of matching trees, forgetting the third argument to MATCH altogether and generating totally new variables, always constructs a complete matching tree. We are now able to prove a stronger completeness statement.

Theorem 4.4. *Let $\mathcal{M}$ by any complete matching tree for e_0 and e'_0 on V. For any $\rho \in \mathcal{U}(e_0, e'_0)$ there exists in $\mathcal{M}$ at a finite level a success node N such that $\rho \underset{V}{\leqslant} \sigma_N$.*

Proof. We just adapt the proof of Theorem 4.2, adding $\rho \underset{V}{=} \xi_t \sigma_{N_t}$ to the induction hypothesis.

For the base step we have $\xi_0 = \rho$, $\sigma_{N_0} = \emptyset$; trivially satisfied.

For the induction step we have now

$$\xi_t \underset{V_{N_t}}{=} \xi_{t+1} \sigma_{t+1}.$$

We now remark that

$$\forall \langle x, e \rangle \in \sigma_{N_t}, \mathcal{F}(e) \subset V_{N_t},$$

since this property is true of $\sigma_1, \sigma_2, \ldots, \sigma_t$. This implies that

$$\xi_t \sigma_{N_t} \underset{V_{N_t}}{=} \xi_{t+1} \sigma_{t+1} \sigma_{N_t} = \xi_{t+1} \sigma_{N_{t+1}},$$

and therefore that

$$\rho \underset{V}{=} \xi_{t+1} \sigma_{N_{t+1}}$$

since $V \subset V_{N_t}$ which concludes the induction. Therefore, for the success node N_p found by this process, we get

$$\rho \underset{V}{=} \xi_p \sigma_{N_p}. \quad \square$$

Now Theorem 4.1 and Theorem 4.4 combined together indicate to what extent we can compute a CSU with our algorithm; by enumerating all success nodes in a complete matching tree we get a complete set of "initial segments" of unifiers, and each such segment can be extended to a unifier by ζ.

Still, this does not give us a way of computing a complete set of remainders, i.e. a CSU for the set of flexible-flexible pairs forming the success node before simplification. Actually, these pairs represent the most difficult cases to treat for general unification, and to enumerate this CSU we would need supplementary rules similar to Pietrzykowski's elimination, iteration and identification. This is rather satisfactory, since it shows that our algorithm is taking great advantage of the fact that we are looking only for the possibility of unification.

4.4. *Elimination of redundancies*

It has been proved [10] that it is not always possible to impose on CSU's a third condition of independence:

3) $\quad \forall \sigma_1 \sigma_2 \in \Sigma \quad \sigma_1 \neq \sigma_2 \Rightarrow \neg\, \sigma_1 \underset{V}{\leqslant} \sigma_2$

This implies that any search enumerating a CSU must be redundant, since if Σ is a CSU containing σ_1 and σ_2 such that $\sigma_1 \underset{V}{\leqslant} \sigma_2$ then $\Sigma - \{\sigma_1\}$ is a CSU too. By contrast, we shall show that the search for the existence of a unifier need not be redundant.

$$\begin{tikzcd}
 & N_1 = \mathrm{F} & & & N_5 = \mathrm{S} \\
N_0 = \{\langle f(B), A(C(B))\rangle\} \arrow[ur, "\langle f, A\rangle"] \arrow[drr, "\langle f, \lambda u\cdot A(h(u))\rangle"'] & & & N_4 = \{\langle h'(B), B\rangle\} \arrow[ur, "\langle h', \lambda u\cdot u\rangle"] \arrow[dr, "\langle h', \lambda u\cdot B\rangle"'] & \\
 & & N_2 = \{\langle h(B), C(B)\rangle\} \arrow[ur, "\langle h, \lambda u\cdot C(h'(u))\rangle"] \arrow[dr, "\langle h, \lambda u\cdot u\rangle"] \arrow[dd, "\langle h, C\rangle"'] & & N_6 = \mathrm{S} \\
 & & & N_7 = \mathrm{F} & \\
 & & N_3 = \mathrm{S} & &
\end{tikzcd}$$

Fig. 5

First, we need to restrict further our MATCH algorithm. A simple example wlil show why. Consider the matching tree of Fig. 5, with:

$$\tau(B) = \tau(u) = \gamma, \quad \tau(A) = \tau(f) = (\gamma \to \alpha),$$
$$\tau(C) = \tau(h) = \tau(h') = (\gamma \to \gamma), \quad V = \{f\}.$$

From the viewpoint of complete unification, N_1 is not redundant, since $\sigma_{N_1} \underset{V}{\neq} \sigma_{N_7}$ whereas N_3 is, since $\sigma_{N_3} \underset{V}{=} \sigma_{N_5}$. From the strict point of view of existence of unification, even N_1 is redundant, because $\langle f, \lambda u\cdot A(h(u))\rangle$ is more general than $\langle f, A\rangle$ *in the context* $f(B)$. We shall now describe the modifications in the construction of matching trees that eliminate these respective redundancies. First, we need some new notations.

Definition 4.5. A *maximal* subterm of a term e in normal form is a subterm of e which is not in the left part of an application; i.e. in $\mathcal{E}\,[(e_1\, e_2)]$ e_2 is maximal and e_1 is not. Maximal subterms are the subterms of the abbreviated notation and so are easy to recognize. For instance in $f(A, B)$, only A, B and $f(A, B)$ are maximal subterms; f and $f(A)$ are non-maximal subterms.

Let e be any term in normal form, $f \in \mathcal{F}(e)$. We call *degree* of f in e, and write $\delta_e(f)$, the smallest i such that there exists a maximal subterm in e with head f and i arguments. For a node N, we define $\delta_N(f)$ as

$$\min\,\{\delta_e(f) \mid \langle e, e'\rangle \in N \text{ or } \langle e', e\rangle \in N\}.$$

Now we define a *reduced matching tree* for e_0 and e'_0 as a matching tree modified as follows:

(i) MATCH is given a fourth argument: in node N, we call MATCH $(e_1, e_2, V, \delta_N(x))$, where x is the head of e_1.

(ii) MATCH only constructs terms with a binder of size at least $\delta_N(x)$.

Theorem 4.1 is still true, of course. Although Lemma 3.7 may fail, Theorem 4.2 still holds. The proof is tedious and technical, and we shall omit it.

For example, in the matching tree in Fig. 5, nodes N_1 and N_3 would not be generated. Generally, reduced matching trees will have a very low degree of branching. This allows a more efficient search, although the lowest level of success nodes may increase a little. For instance in Fig. 5, we now have to go' down to level 3 to find a success node.

To get an equivalent of Theorem 4.4, we can define in the same way a *reduced complete matching tree* for e_0 and e_0' on V. Defining V_N as in 4.3, at node N we call MATCH (e_1, e_2, V_N, δ) where, denoting by x the head of e_1:

$$\delta = \begin{cases} 0 & \text{if } x \in V \quad [\text{i.e. } x \in \mathcal{F}(e_0) \cup \mathcal{F}(e_0')] \\ \delta_e(x) & \text{if } x \text{ has been introduced by MATCH in term } e \text{, on the branch} \\ & \text{leading to } N. \end{cases}$$

For instance, we get a reduced complete matching tree for $f(B)$ and $A(C(B))$ on $\{f\}$ by suppressing N_3 in Fig. 5. Note that now reduced complete matching trees may be larger than reduced matching trees.

The argument of Theorem 4.4 can now be easily adapted to reduced complete matching trees. We shall not give the details here, but we merely justify the complicated definition above by remarking that it would not be sufficient to take $\delta = \delta_N(x)$ in case $x \notin V$. For instance, consider the following counter example:

$$e_0 = f(x, A), \quad e_0' = B(A),$$

with

$$\tau(A) = \gamma, \quad \tau(B) = (\gamma \to \gamma), \quad \tau(x) = ((\gamma \to \gamma), \gamma \to \gamma),$$
$$\tau(f) = (((\gamma \to \gamma), \gamma \to \gamma), \gamma \to \gamma).$$

Suppose we consider the unifier:

$$\sigma = \{\langle f, \lambda u \cdot u(B)\rangle, \langle x, \lambda v \cdot v\rangle\}$$

with

$$\tau(u) = \tau(x), \tau(v) = (\gamma \to \gamma).$$

The corresponding branch of the reduced complete matching tree is going to be:

$$\begin{tikzcd}
\{\langle f(x, A), B(A)\rangle\} \arrow[d, "\langle f, \lambda u \cdot u(h(u))\rangle"] \\
\{\langle x(h(x), A), B(A)\rangle\} \arrow[d, "\langle x, \lambda v \cdot v\rangle"] \\
\{\langle h(\lambda v \cdot v, A), B(A)\rangle\} = N \arrow[d, "\langle h, \lambda u \cdot B\rangle"] \\
S
\end{tikzcd}$$

If in N we take $\delta = \delta_N(h) = 2$ we shall not generate the pair $\langle h, \lambda u \cdot B\rangle$ and so we shall miss σ. But we choose rather to take $\delta = \delta_e(h) = 1$, since here e is $\lambda u \cdot u\,(h\,(u))$.

We shall now motivate our introduction of reduced complete matching trees by proving a strong independence result for these trees. Let us call two nodes independent if one is not the ancestor of the other.

Theorem 4.6. *Let $\mathcal{M}$ be any reduced complete matching tree for e_0 and e_0' on V. If N_1 and N_2 are two independent nodes in $\mathcal{M}$ then $\sigma_{N_1} \underset{F}{|} \sigma_{N_2}$, where $F = \mathcal{F}(e_0) \cup \mathcal{F}(e_0')$.*

Proof. Let $\bar{N}$ be the closest common ancestor to N_1 and N_2 in $\mathcal{M}$:

$$\begin{tikzcd} & & & \dots \arrow[r] & N_1 \\ N_0 \arrow[r] & \dots \arrow[r] & \bar{N} \arrow[ur] \arrow[dr] & & \\ & & & \dots \arrow[r] & N_2 \end{tikzcd}$$

Let us define, for any node N in the branch $N_0 \to \dots \to \bar{N}$, $\rho_1(N)$ as the composition of the substitution pairs from N to N_1, $\rho_2(N)$ as the composition of the substitution pairs from N to N_2, and $v\,(N)$ as the variable substituted for in node N. Now we define property P by:

$$\mathrm{P}\,(N) \Leftrightarrow \nexists \eta_1, \eta_2 : \eta_1\, \rho_1(N)\, v\,(N) = \eta_2\, \rho_2(N)\, v\,(N).$$

Let us call N^{-1} the ancestor to N in $\mathcal{M}$ which introduces $v\,(N)$ as a new variable, if such a node exists. In this case, we prove that: $\mathrm{P}\,(N) \Rightarrow \mathrm{P}\,(N^{-1})$. Let us write:

$$N^{-1} \xrightarrow{\sigma} \underbrace{\qquad \to}_{} N \to \dots \to \bar{N},$$

with

$$\sigma v\,(N^{-1}) = \lambda w_1 \dots w_n \cdot @\,(\dots, v\,(N)\,(w_1, \dots, w_n), \dots),$$

and assume $\neg \mathrm{P}\,(N^{-1})$, i.e. $\exists \eta_1, \eta_2 : \eta_1\, \rho_1(N^{-1})\, v\,(N^{-1}) = \eta_2\, \rho_2(N^{-1})\, v\,(N^{-1})$. Using $\rho_1(N^{-1}) = \rho_1(N)\, \xi\sigma$, $\rho_2(N^{-1}) = \rho_2(N)\, \xi\sigma$ and the fact that ξ does not pertain to $v\,(N)$, we get:

$$\eta_1\, \rho_1(N)\, [\lambda w_1 \dots w_n \cdot @\,(\dots, v\,(N)\,(w_1, \dots, w_n), \dots)] =$$
$$= \eta_2\, \rho_2(N)\, [\lambda w_1 \dots w_n \cdot @\,(\dots, v\,(N)\,(w_1, \dots, w_n), \dots)].$$

Since $v\,(N) \neq w_i\ \forall i \in [n]$ and $@ \in \mathcal{C} \cup \{w_1, \dots, w_n\}$, this implies:

$$\lambda w_1 \dots w_n \cdot [[\eta_1\, \rho_1(N)\, v\,(N)]\,(w_1, \dots, w_n)] =$$
$$= \lambda w_1 \dots w_n \cdot [[\eta_2\, \rho_2(N)\, v\,(N)]\,(w_1, \dots, w_n)].$$

Now, by definition $\delta_e(v\,(N)) = n$, and therefore $\sigma' v\,(N)$ has a binder of length at least n, for any substitution pair σ' given by MATCH in N. Therefore this property is also true of $\eta_1\, \rho_1(N)\, v\,(N)$ and $\eta_2\, \rho_2(N)\, v\,(N)$ by Lemma 2.2, using for σ' the first components of $\rho_1(N)$ and $\rho_2(N)$ (which are equal if $N \neq \bar{N}$). Using α-conversion, we easily get:

$$\eta_1 \rho_1(N)\, v(N) = \eta_2 \rho_2(N)\, v(N)$$
$$\text{i.e.} \neg\, \mathrm{P}(N).$$

By contraposition we get $\mathrm{P}(N) \Rightarrow \mathrm{P}(N^{-1})$ which proves the induction step. The basis of the induction is $\mathrm{P}(\bar{N})$, obtained easily because MATCH does not construct two rigid terms with the same heading.

Using the least number principle of induction, we get that there exists a node N between N_0 and $\bar{N}$ which has no N^{-1}, i.e. such that $v(N) \in F$, and such that $\mathrm{P}(N)$. Now, for any η, η';

$$\eta\sigma_{N_1}\, v(N) = \eta\rho_1(N)\, v(N) \neq \eta'\rho_2(N)\, v(N) = \eta'\sigma_{N_2}\, v(N),$$

and therefore $\sigma_{N_1} \underset{F}{|} \sigma_{N_2}$. □

We can sum up the previous results as follows:
Let e_0 and e'_0 be any two terms of the same type, V be any finite set of variables containing $F = \mathcal{F}(e_0) \cup \mathcal{F}(e'_0)$, and $\mathcal{U}$ be the set of unifiers of e_0 and e'_0. Let $\mathcal{M}$ be any reduced complete matching tree for e_0 and e'_0 on V, and $\Sigma(\mathcal{M}) = \{\sigma_N | N = \mathrm{S} \text{ in } \mathcal{M}\}$. Then:

$\forall \sigma_N \in \Sigma(\mathcal{M}) : \zeta_N\, \sigma_N \in \mathcal{U}$ (Theorem 4.1);

$\forall \rho \in \mathcal{U}\ \exists! \sigma \in \Sigma(\mathcal{M}) : \rho \underset{V}{\leqslant} \sigma$ (Theorems 4.4 and 4.6);

$\forall \underset{N \neq N'}{\sigma_N, \sigma_{N'}} \in \Sigma(\mathcal{M}) : \sigma_N \underset{F}{|} \sigma_{N'}$ (Theorem 4.6).

We can conclude that using our algorithm to enumerate success nodes is as close as possible to enumerating a CSU without redundancies. In other words, the redundancy implied by a search for complete unifiers can be pushed into the flexible-flexible cases which we never examine. This gives a strong efficiency argument in favor of theorem proving methods based on unifiability [12] as opposed to methods based on CSU's.

4.5. *The η-rule*

We have not so far assumed the η-rule of the λ-calculus:

$$\mathcal{E}\, [\lambda x\, (ex)] = \mathcal{E}\, [e] \quad \text{where } x \notin \mathcal{F}(e).$$

This rule is implied by the axiom of functional extensionality:

$$\forall f, g\, (\forall x\, f(x) = g\,(x) \Rightarrow f = g).$$

Since one does not always want to assume this axiom, we formulated our unification algorithm without the η-rule, following Andrews [1]. However if this rule is valid, our algorithm can be greatly simplified, because now many cases will become redundant. The necessary changes affect only the definition of normal form and the algorithm MATCH.

4.5.1. *η-normal form*

The η-normal form of a term is obtained from its normal form by replacing every subterm of the abbreviated form

$$\lambda u_1 \ldots u_n \cdot @\,(e_1, \ldots, e_p),$$

where

$$\tau\,(@) = (\alpha_1, \alpha_2, \ldots, \alpha_p, \ldots, \alpha_q \to \beta) \qquad q \geqslant p,$$

by the term:

$$\lambda u_1 \ldots u_n\, w_1 \ldots w_{q-p} \cdot @\,(e_1, \ldots, e_p, w_1, \ldots, w_{q-p})$$

where the w_i's are new distinct variables of the appropriate type.

4.5.2. *Changes in MATCH*

First we note that we have always $n_1 = n_2$, and that therefore we can suppress 3.4.1.1(i) and 3.4.1.2(ii). Also, we now have $p_1 = q_1$. In cases of 3.4.1.1(ii) and 3.4.1.2(i) we shall restrict ourselves to the only solution $k = p_1$. The justification is obvious: now the other possible solutions construct terms which are instances of the one kept. For instance, if

$$e_1 = f(A) \quad e_2 = B\,(A) \quad \text{with}\ \tau\,(A) = \gamma, \quad \tau\,(f) = \tau\,(B) = (\gamma \to \gamma),$$

then $B = \lambda u \cdot B\,(u)$ is an instance of $\lambda u \cdot B\,(h\,(u))$ by the substitution $\{\langle h, \lambda u \cdot u\rangle\}$. In the same way in the example in 3.4.2, we do not generate σ_1 and σ_3 which are less general than σ_2 and σ_4 respectively. Generally, we shall consider at most one case of imitation and p_1 cases of projection. This is to be compared with the possible $\min(p_1, p_2)+1$ cases of imitation and $p_1(p_1+2n+1)/2$ cases of projection we had before; the MATCH algorithm is much simpler with η-conversion than without, and the degree of the matching tree is sharply reduced.

However, note that we may now get solutions where we failed using only the α and β rules, for instance for the pair $\langle \lambda u \cdot f(u), A\rangle$, where $\tau\,(u) = \gamma$, $\tau\,(f) = \tau\,(A) = (\gamma \to \gamma)$. This simpler algorithm would therefore be neither sound nor complete for non-extensional theories in which the η-conversion is ruled out.

Assuming η-conversion valid, Theorems 4.1 and 4.2 are still true, Theorem 4.4 is true if we give to MATCH sets V_N as in 4.3. But now η-matching trees are non-redundant without any further modifications: we easily get Theorem 4.6, since every variable has its maximum degree in every term because of the η-normal form. For instance, in the example of Fig. 5, N_1 and N_3 would not be generated.

5. Heuristic improvements

5.1. *Selection of the pair given to MATCH*

Let N be a node containing several flexible-rigid pairs. Our algorithm does not impose any condition on the selection of the pair we give to MATCH, when we want to compute the successors of N. As proved in Theorem 4.1, if e_0 and e_0' are unifiable at all, this selection is indifferent, in the sense that any matching tree will possess a success node at a finite level. However, if $\mathcal{U}(e_0, e_0') = \emptyset$ certain strategies pertaining to this selection could lead to a finite (failure) tree, while others could lead to an infinite tree. For instance, consider:

$$N = \{\langle f(A), F(f(A))\rangle, \langle \lambda uv\cdot g(v), \lambda uv\cdot u\rangle\}$$

where

$$\tau(A) = \tau(u) = \alpha, \quad \tau(v) = \beta, \quad \tau(f) = (\alpha \to \beta),$$
$$\tau(F) = (\beta \to \beta), \quad \tau(g) = (\beta \to \alpha).$$

If we input the second pair to MATCH, we get as result $\Sigma = \emptyset$ since no rule applies and therefore we can replace N directly by a failure node F. If on the contrary we input the first pair, we may get an infinite tree:

$$\begin{tikzcd}[row sep=large]
N \arrow[d, "{\langle f, \lambda u\cdot F(h(u))\rangle}"] \\
N' = \{\langle h(A), F(h(A))\rangle, \langle \lambda uv\cdot g(v), \lambda uv\cdot u\rangle\} \arrow[d, "{\langle h, \lambda u\cdot F(h'(u))\rangle}"] \\
\cdots
\end{tikzcd}$$

To avoid this kind of situation, it may be preferable to process the pairs in some kind of first-in first-out fashion.

5.2. *Recognition of cycles*

A useful heuristic to incorporate in the unification process would be to suppress branches that lead to infinite repeating cycles. That is, if a branch $N_0 \to N_1 \to \ldots \to N_p$ is such that for some $n < p$, N_n is identical to N_p up to a one to one type-preserving renaming of their free variables, then N_p can be safely replaced by F.

For instance, in the previous example in 5.1, we would recognize N' identical to N up to renaming of f by h and therefore stop at N' with failure.

Note that replacing N_p by a failure node F does not mean that N_p is not unifiable, but merely that if it possesses a success descendant, then so does N_n, but at a lower level on some other branch. The proof of completeness of this heuristic is straightforward if each node contains a unique flexible-rigid pair since then there is a unique way of constructing the subtrees from N_n and N_p, and therefore these subtrees are identical; the proof follows easily. In the general case the proof is more complex and we omit it here.

This heuristic will not avoid other kinds of sterile infinite branches. Consider for instance $e_0 = f(f(A))$, $e'_0 = F(f(f(A)))$ with types as above:

$$\begin{tikzcd}
\{\langle f(f(A)), F(f(f(A)))\rangle\} \arrow[d, "{\langle f, \lambda u\cdot F(h(u))\rangle}"] \\
\{\langle h(F(h(A))), F(h(F(h(A))))\rangle\} \arrow[d] \\
\cdots
\end{tikzcd}$$

Actually we cannot hope to be able to recognize all such cases, since unification is undecidable in languages of order three or more. If we restrict our language to order two, i.e. allow only variables of types $(\alpha_1, \alpha_2, \ldots, \alpha_n \to \beta)$ with $\forall i \in [n]\ \alpha_i \in T_0$, $\beta \in T_0$, then the result is not known. In the monadic case (i.e. $n \leqslant 1$ above) the problem reduces to finding a solution to a system of equations in a free monoid. (It is not equivalent, since the possibility of having constant functions $\lambda u\cdot v$ amounts to having a zero element in the monoid.) The monoid problem is still unresolved in the general case.

5.3. *The fixpoint problem*

We shall now look at the special case of unification of a variable x with a term e of the same type. This case is specially important since it occurs often in practice. First we note that this is the same problem as finding a syntactic fixpoint of an expression of our language.

We call *fixpoint* of a term $\lambda x\cdot e$, where $\tau(e) = \tau(x)$, any term e' of type $\tau(x)$ such that $[\lambda x\cdot e](e') = e'$. If e' is a fixpoint of $\lambda x\cdot e$, then $\{\langle x, e'\rangle\}$ is a unifier of x and e. Conversely, if we write a unifier σ of x and e as $\sigma = \{\langle x, e'\rangle\} \cup \rho$, then e' is a fixpoint of $\rho[\lambda x\cdot e]$.

For instance, let $e = \lambda u\cdot u(x(\lambda vw\cdot v), f(x))$ with $\tau(v) = \tau(w) = \alpha$, $\tau(u) = (\alpha, \alpha \to \alpha)$, $\tau(x) = ((\alpha, \alpha \to \alpha) \to \alpha)$ and $\tau(f) = (((\alpha, \alpha \to \alpha) \to \alpha) \to \alpha)$. A unifier of x and e is $\sigma = \{\langle x, \lambda u\cdot u(y, z)\rangle, \langle f, \lambda x\cdot z\rangle\}$, where $\tau(y) = \tau(z) = \alpha$. Then $e' = \lambda u\cdot u(y, z)$ is a fixpoint of $\lambda xu\cdot u(x(\lambda vw\cdot v), z)$.

Lemma 5.1. *If $x \notin \mathcal{F}(e)$, then $\sigma = \{\langle x, e\rangle\}$ is a most general unifier of x and e.*

Proof. Let $\rho \in \mathcal{U}(x, e)$. We prove that $\rho = \rho\sigma$.

Case 1. $\rho x = E \neq x$. There must exist in ρ a pair pertaining to x, i.e.:
$\rho = \{\langle x, E\rangle\} \cup \bar{\rho}$. $\rho x = E = \rho e$ by hypothesis, and therefore $\rho\sigma = \{\langle x, E\rangle\} \cup \bar{\rho} = \rho$.

Case 2. $\rho x = x = \rho e$; then again $\rho\sigma = \rho$. □

This lemma, which is a generalization to typed λ-calculus of a well-known property of first-order unification, could be used by SIMPL to suppress such pairs $\langle x, e\rangle$ by effecting the corresponding substitution in the node. However, its converse is not true any more; i.e., x may appear free in e and still be unifiable with e. For example, if $e = f(x)$, by substitution $\{\langle f, \lambda u\cdot x\rangle\}$ or $\{\langle f, \lambda u\cdot u\rangle\}$. This may happen even if there is no free variable such as f above to eliminate x. For instance, take:

$$e = \lambda u\cdot u\,(x\,(\lambda v\cdot v)) \quad \text{with } \tau(v) = \gamma, \tau(u) = (\gamma \to \gamma),$$
$$\tau(x) = ((\gamma \to \gamma) \to \gamma);$$

$\sigma = \{\langle x, \lambda u\cdot u\,(y)\rangle\}$ unifies e and x, with $\tau(y) = \gamma$. Example 3.5.3, where we had a combination of the two phenomena, is similar.

Let us call *rigid path for x in e* a sequence of terms:

$$e = e_1, e_2, \ldots, e_n, e_{n+1} \qquad n \geqslant 1$$

such that:

$\forall i \in [n]$ e_{i+1} is an argument of e_i

$\forall i \in [n]$ $\exists j \in [i]$ such that the head of e_i is a constant or a variable in the binder of some e_j

the head of e_{n+1} is an occurrence of x free in e.

If there exists no rigid path for x in e, then e and x are easily unifiable. However, the converse is false, as shown in 3.5.3 for instance. We can conclude partially in two special cases:

if e has an empty binder and there exists a rigid path for x in e, then $\mathcal{U}(e, x) = \emptyset$

if there exists a rigid path for x in e such that its last element e_{n+1} has no arguments, then $\mathcal{U}(e, x) = \emptyset$.

In particular, in the case of first-order terms, we get the well-known result that $\mathcal{U}(e, x) = \emptyset$ if x appears in e. Since this case occurs often in practice, it is important to incorporate it in SIMPL.

6. Conclusion

We have presented in this paper two algorithms which search for the existence of unifiers for sets of formulas in ω-order typed λ-calculus. The first one assumes the α and β rules of λ-conversion. The second one, which assumes in supplement the η rule, is significantly simpler and is more appealing for practical computer implementation purposes.

These algorithms exhibit very good convergence properties. This is due to the fact that we perform only substitutions which are absolutely mandatory, when we match two terms one of which is rigid. This condition gives us a good directionality, since the heading of the rigid term is used as a handle on the unification process. We never search in "don't care" situations since our algorithm does not examine the formulas in a left to right fashion, but rather tests pairs according to an easy-first-hard-last criterion. Finally situations which can be described as "don't care in all positions" are recognized directly as successes. This permits us to ignore the most complex cases of general unification (the flexible-flexible cases) and to get rid of the rules of elimination, iteration and identification. These last two rules are very prolific, since not directional.

This study gives strong efficiency arguments in favor of theorem-proving methods based on unifiability rather than complete sets of unifiers. This work is part of a case study in such a method, applied to higher-order logic based on typed λ-calculus [12].

The algorithm with α-β conversion has been implemented at IRIA for experiments on higher-order deduction.

Acknowledgments

The author thanks P. Andrews, G. Kahn, T. Pietrzykowski and G. Plotkin for their help in preparing this paper.

References

[1] P. B. Andrews, Resolution in type theory, J. Symb. Logic 36 (3) (1971) 414–432.
[2] A. Church, A formulation of the simple theory of types, J. Symb. Logic 5 (1)(1940) 56–68.
[3] J. L. Darlington, A partial mechanization of second-order logic, Machine Intelligence 6 (American Elsevier, New York, 1971) 91–100.
[4] J. L. Darlington, Automatic program synthesis in second-order logic, Proceedings of 3rd Intern. Joint Conf. on Artificial Intelligence (1973).
[5] G. W. Ernst, A matching procedure for type theory, Personal Communication (1971).
[6] W. E. Gould, A matching procedure for ω-order logic, Scientific Report No. 4, A F C R L (1966) 66–781, AD 646 560.
[7] J. R. Guard, Automated logic for semi-automated mathematics, Scientific Report No. 1. A F C R L (1964) 64–411, AD 602 710.
[8] J. R. Hindley, B. Lercher and J. P. Seldin, Introduction to Combinatory Logic, London Mathematical Society Lecture Note Series 7 (Cambridge University Press, 1972).
[9] G. P. Huet, Constrained resolution: a complete method for type theory, Jennings Computing Center Report 1117. Case Western Reserve University (1972).
[10] G. P. Huet, Unification en théorie des types, Séminaire IRIA Théorie des Automates, des languages et de la programmation, volume 1973.
[11] G. P. Huet, The undecidability of unification in third order logic, Information and Control 22 (3) (1973) 257–267.
[12] G. P. Huet, A mechanization of type theory, Proceedings of 3rd Intern. Joint Conf. on Artificial Intelligence (1973).
[13] D. Jensen and T. Pietrzykowski, Mechanizing ω-order type theory through unification, Report CS-73-16, Dept. of Applied Analysis and Computer Science, University of Waterloo (1973).
[14] C. L. Lucchesi, The undecidability of the unification problem for third order languages,

Report C S R R 2059, Dept. of Applied Analysis and Computer Science, University of Waterloo (1972).

[15] T. Pietrzykowski, A complete mechanization of second order logic, J. Assoc. Comp. Mach. 20 (2) (1971) 333–364.

[16] T. Pietrzykowski and D. Jensen, A complete mechanization of ω-order type theory, Assoc. Comp. Mach. Nat. Conf. 1972, Vol. 1, 82–92.

[17] G. A. Robinson and L. T. Wos, Paramodulation and theorem proving in first-order theories with equality, Machine Intelligence 4 (American Elsevier, New York, 1969) 135–150.

[18] J. A. Robinson, A machine-oriented logic based on the resolution principle, J. Assoc. Comp. Mach. 12 (1) (1965) 23–41.