Correctness of Multiplicative Proof Nets is Linear*

Stefano Guerrini
Dipartimento di Scienze dell'Informazione - Università di Roma I, 'La Sapienza'
Via Salaria, 113 - I-00198, Roma, Italy - Email: guerrini@dsi.uniroma1.it

Abstract

We reformulate Danos contractibility criterion in terms of a sort of unification. As for term unification, a direct implementation of the unification criterion leads to a quasi-linear algorithm. Linearity is obtained after observing that the disjoint-set union-find at the core of the unification criterion is a special case of union-find with a real linear time solution.

1 Introduction

A multiplicative proof net is a graph representation of a multiplicative linear logic derivation [5].[1] The proof net N corresponding to the derivation Π is a (directed) hypergraph with a link (*i.e.*, a hyperedge) for each rule and a vertex for each formula occurrence s.t. every link of N connects the active formulas of the corresponding rule of Π. For instance, let Π be a cut-free derivation using atomic axioms only and ending with the sequent $\vdash A$; the corresponding proof net N is the syntax tree of the formula A, plus a set of connections between pairs of occurrences of dual atoms—each node of the syntax tree is a $\parr$ or $\otimes$-link; each pairing connection between a pair of leaves of the syntax tree is an axiom link.

The derivation Π establishes an ordering, say a *sequentialization*, among the links of N. In general, the sequentialization of a proof net is not unique. For instance, let us assume that Π end with $\vdash A \parr B, C \parr D$ and that the last two rules of Π introduce the principal connectives of $A \parr B$ and $C \parr D$; the proof net N corresponding to Π does not depend on the actual order of that pair of rules.

A proof structure is a hypergraph built in accord to the syntax of proof nets, but without following any sequentialization order. A *correctness criterion* is a test that, given a proof structure N, answers *yes* when N is a proof net, *no* when there is no sequentialization of N. The *Danos-Regnier switching condition* is the most known correctness criterion: a *switch* of N is the graph obtained by disconnecting one of the premises of each $\parr$-link; N is a proof net iff every switch of N is a tree [3].

If n is the number of $\parr$-links in N, the Danos-Regnier criterion checks 2^n graphs. Moreover, there is good evidence that correctness cannot be inferred by the inspection of a fixed subset of the switches of N. For instance, for any n, we can construct a proof structure s.t. only one (or two, or three, etc.) of its switches is not a tree.

The situation is different for the multiplicative proof nets of non-commutative (and cyclic) linear logic. In that case, the Danos-Regnier criterion does not suffice for correctness; a proof net of commutative linear logic with only one conclusion is a non-commutative proof net iff it is a planar graph. Because of this additional requisite, verifying the correctness of a non-commutative proof structure requires the inspection of two switches only [9]. Unfortunately, planarity does not play any role in the commutative case; therefore, in order to obtain a linear algorithm, we must resort to something else.

Danos *contractibility* is the first step towards an efficient correctness criterion. In his thesis [2], Danos gave a set of shrinking rules for proof structures, characterizing proof nets as the only proof structures that contract to a point. As finding the next link to shrink is at most linear in the size of the graph, and as each shrinking rule removes a link, Danos contractibility is quadratic. The idea of Danos has been improved and extended by showing that it can be presented as a *parsing* algorithm [7, 6]. Here, we make a further step, showing that the implementation of parsing as a sort of *unification* leads to a linear time correctness criterion.

*Partially supported by the italian MURST project 'Tecniche Formali per la Specifica, l'Analisi, la Verifica, la Sintesi e la Trasformazione di Sistemi Software'. This work has been started while at IML, Marseille supported by a EU Marie Curie fellowship and completed while at the Department of Computer Science of Queen Mary and Westfield College, London supported by an EPSRC grant.

[1] We shall consider multiplicative linear logic without constants only. In the case with constants, proof net correctness is NP-hard [7].

Like term unification, the unification criterion can be formulated as a disjoint-set union-find problem. Thus, the use of any union-find α-algorithm leads to a quasi-linear criterion (that is, linear up to a factor α that is a sort of inverse of the Ackermann function). Although that kind of algorithms are morally linear (there is no feasible experiment showing their non-linearity) and behave very well in practice (their constants are very small), a more detailed analysis of the union-find required by the unification criterion shows that this is a special case with a real linear time solution. Therefore, getting rid of the α factor, we conclude that checking the correctness of a multiplicative proof net is linear.

We stress that all the efficient algorithms derived from contractibility and parsing verify correctness by constructing a sequentialization. Therefore, we have the particularly surprising result that forgetting the sequentialization of a proof net does not implies the loss of any information, neither in terms of the computational resources required to recover a sequentialization. The reconstruction of a sequentialization can be done in linear time, that is, in the minimal time required for reading the whole proof net.

2 Proof nets

A *link* is a pair

$$u_1, u_2, \ldots, u_h \rhd v_1, v_2, \ldots, v_k$$

in which the *premises* $\alpha = u_1, u_2, \ldots, u_h$ and the *conclusions* $\beta = v_1, v_2, \ldots, v_k$ are two disjoint sets of vertices s.t. $\alpha \cup \beta \neq \emptyset$.

Each premise or conclusion of a link has a distinct name (*e.g.*, in a link with two premises, the names might be left and right; in a link with n conclusions, we might distinguish the 1st, the 2nd, ..., the nth premise). However, since we shall not consider cut-elimination, names will not play any role.

A link without premises (source link) or without conclusions (target link) is a root link and is denoted by $\overset{0}{\rhd}$. In a proof structure there are two types of root links: the *axiom* links, which are source links with two conclusions; the *dummy* links, which are target links with one premise only (see Figure 1). All the internal links (*i.e.*, the links that are not a root of the proof structure) have two premises and one conclusion; nevertheless, they split in two types: the *unary links*, denoted by $\overset{1}{\rhd}$, and the *binary links*, denoted by $\overset{2}{\rhd}$ (see Figure 1).

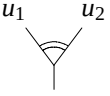
axiom	*dummy*	*unary*	*binary*
$\overset{0}{\rhd} u_1, u_2$	$u \overset{0}{\rhd}$	$u_1, u_2 \overset{1}{\rhd} u_0$	$u_1, u_2 \overset{2}{\rhd} u_0$
			

Figure 1. Links.

Definition 1 (proof structure). A *proof structure* (of links) G over the vertices $V(G)$ is a set of links $G = \alpha'_1 \rhd \alpha''_1; \alpha'_2 \rhd \alpha''_2; \ldots; \alpha'_k \rhd \alpha''_k$ s.t.:

1. Every vertex in $V(G)$ is a conclusion of one link (only one).
2. Every vertex in $V(G)$ either is a *conclusion* of G (*i.e.*, it is not a premise of any link of G) or is a premise of one link (only one).
3. The premise of every dummy link is the conclusion of a binary link.
4. The set α of the conclusions of G (written $G \vdash \alpha$) is not empty.

A set of links G' is a *structure of links* when $G' \subseteq G$, for some proof structure G (note that G' might not be a proof structure). A *premise* of G' is a vertex that is not the conclusion of any link in G'. Let α_1 be the set of the premises of G' and α_0 be the set of its conclusions; we shall write $G' : \alpha_1 \vdash \alpha_0$. When G' is a proof structure, we shall say that G' is a *proof substructure* of G.

Definition 1 does not postulate anything about the intended meaning of G. The concrete examples corresponding to that abstraction are the proof structures of the multiplicative fragment without constants of linear logic. Therefore, G is a *concrete proof structure* when $V(G) = F \cup C$ is formed of a set F of occurrences of multiplicative linear logic formulas and a set C of occurrences of the reserved symbol cut s.t. every link of G matches one of the following patterns:

$$\overset{0}{\rhd} A, A^\perp \qquad A, B \overset{1}{\rhd} A \parr B \qquad \mathsf{cut} \overset{0}{\rhd}$$

$$A, A^\perp \overset{2}{\rhd} \mathsf{cut} \qquad A, B \overset{2}{\rhd} A \otimes B$$

with $A, B \in F$.

Remark 2. Concrete proof structures differ from the usual proof structures because of the representation of cuts. In our structures, tensors and cuts collapse into the same type of link (with the minor technicality of a dummy below every cut). Indeed, for checking correctness, there is no difference between a tensor and a cut (compare the tensor and cut rules in Figure 2).

2.1 Danos-Regnier correctness criterion

We shall represent graphs by means of their incidence relation. Namely, an (undirected) graph is a pair $(V, \frown)$, where $\frown$ is a symmetric and antireflexive binary relation over the set of vertices V and $u \frown v$ reads 'there is an edge between u and v'. Moreover, we shall write $u \smile v$, when there is no edge between u and v.

Definition 3 (switch). A *switch* S of the structure of links G, say $S \in \mathsf{Sw}(G)$, is a graph $(V(G), \frown)$ s.t.:

- $\overset{0}{\triangleright} u_1, u_2 \in G$ implies $u_1 \frown u_2$, *i.e.*, each axiom link is replaced by an edge between its conclusions;
- $u_1, u_2 \overset{1}{\triangleright} u_0$ implies $(u_1 \frown u_0) \wedge (u_2 \smile u_0)$ or $(u_2 \frown u_0) \wedge (u_1 \smile u_0)$, *i.e.*, each unary link is replaced by an edge (only one) between the conclusion and one of the premises of the link;
- $u_1, u_2 \overset{2}{\triangleright} u_0$ implies $(u_1 \frown u_0) \wedge (u_2 \frown u_0)$, *i.e.*, each binary link is replaced by a pair of edges between its conclusion and its premises.

In a switch, a dummy link does not correspond to any edge.

Definition 4 (proof net). A structure of links G is DR-*correct* when, for every $S \in \mathsf{Sw}(G)$, S is connected and acyclic (*i.e.*, S is a tree). The proof structures that are DR-correct are named *proof nets*.

The vertices of a proof net can be ordered according to the vertical layout in Figure 1. Given a proof structure G, let us say that u_1 is immediately below u_0 when there is a link s.t. $\alpha_0, u_0 \triangleright \alpha_1, u_1$; the least preorder $\preccurlyeq$ induced by '$u_1 \preccurlyeq u_0$ if u_1 is immediately below u_0' is a partial order (by the way, $u_1 \preccurlyeq u_0$ can be equivalently red u_1 below u_0 or u_0 above u_1).

2.2 Sequentialization

The class of the sequentializable proof structures Seq is the smallest set of proof structures inductively defined by the rules in Figure 2. In each of that rules, u_0 is a fresh vertex; moreover, the rules tensor and cut have the proviso $V(G_1) \cap V(G_2) = \emptyset$. Indeed, these side conditions are implicit in the fact that Seq contains well-formed proof structures only.

The rules in Figure 2 are the graph theoretic abstraction of the derivation rules of multiplicative linear logic. In particular, any sequentialization of a concrete proof structure G (*i.e.*, any derivation of $G \in \mathsf{Seq}$) corresponds to a linear logic derivation.

Theorem 5 (sequentialization). *A proof structure G is a proof net iff $G \in \mathsf{Seq}$.*

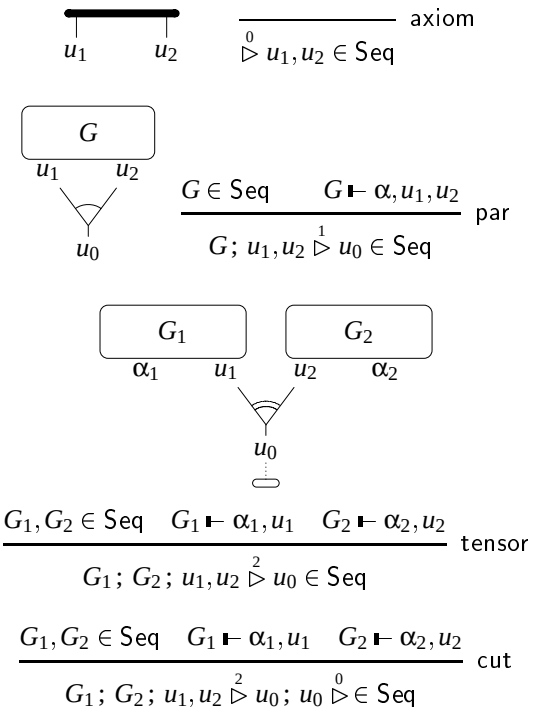


Figure 2. Sequentialization.

2.3 The size of a proof structure

Let us define the *size* of a proof structure G as the number of registers $size(G)$ required for the memorization of G on some random access machine (RAM). Although the definition of $size(G)$ depend on the representation of G, in any non-redundant coding, $size(G)$ is linear in the number of vertices of G. Therefore, we shall postulate $size(G) = \Theta(|V(G)|)$. Moreover, since the number of links in G is linear in the number of vertices of G (*i.e.*, $|G| = \Theta(|V(G)|)$), $size(G) = \Theta(|G|)$ also. In the following, we shall analyze the worst case asymptotic complexity of DR-correctness in terms of $size(G)$.

Remark 6 (asymptotic notation). We recall the definition of the following classes of functions (where $g(n)$ and $f(n)$ are positive functions): $O(f(n)) = \{g(n) \mid \exists c, k > 0 : n > k \implies c\,f(n) \geq g(n)\}$ (asymptotic upper-bound); $\Omega(f(n)) = \{g(n) \mid \exists c, k \geq 0 : n > k \implies c\,f(n) \leq g(n)\}$ (asymptotic lower-bound); $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$. In the following, we shall also use $O(f(n))$, $\Theta(f(n))$ and $\Omega(f(n))$ to denote a generic element of the corresponding classes; because of this, for instance, $g(n) = O(f(n))$ will be an abuse of notation for $g(n) \in O(f(n))$.

3 Quadratic time criteria

D(anos)R(egnier)-correctness is the simplest and most appealing characterization of the proof structures that can be inductively constructed according to the rules of multiplicative linear logic. However, a direct application of Danos-Regnier criterion requires $\Theta(size(G)\,2^{size(G)})$ time: a proof structure G with $n = \Theta(|G|)$ unary links has 2^n switches and checking that a switch is a tree is $\Theta(|G|)$. Instead, the criteria that we present in this section are quadratic. Moreover, we could easily give a parsing strategy checking correctness in $O(n \log n)$ time but, in the next section, we shall do much better.

3.1 Contractibility

Checking the correctness of the proof structure G_1 can be reduced to checking the correctness of some substructure of G_1. Namely, let $G_0 : \alpha_1 \vdash \alpha_0$ be a DR-correct substructure of G_1. In any switch of G_1 there is a path between every pair of vertices $u, v \in \alpha_1 \cup \alpha_0$. Therefore, let G_2 be the structure obtained from G_1 by replacing the link $\alpha_1 \triangleright \alpha_0$ for G_0; any switch of G_1 is topologically equivalent to a corresponding switch of G_2, provided that the new link $\alpha_1 \triangleright \alpha_0$ be interpreted as an acyclic graph connecting each pair $u, v \in \alpha_1 \cup \alpha_0$.

More formally. A *star link* is a link $\alpha_1 \overset{*}{\triangleright} \alpha_0$ with any number of premises and conclusions. A *contracting structure of links* (proof structure) is a structure of links (proof structure) that may contain star links also.

A switch $S = (|G|, \frown)$ of the contracting structure of links G is a graph verifying the constraints in Definition 3 plus:

$\alpha_1 \overset{*}{\triangleright} \alpha_0 \in G$ implies that there is a permutation $u_0, u_1, \ldots u_k$ of α_1, α_0 s.t. $u_i \frown u_{i+1}$, for $i = 0, 1, \ldots, k-1$.

The definition of DR-correctness is the one already given: every switch of the contracting structure of links G is connected and acyclic.

Lemma 7. *Let $G : \alpha_1 \vdash \alpha_0$ be a DR-correct contracting structure of links. A contracting proof structure $G_1 = G_0$; G is DR-correct iff the contracting proof structure $G_2 = G_0$; $\alpha_1 \overset{*}{\triangleright} \alpha_0$ is DR-correct.*

The previous lemma suggests the rewriting rules in Figure 3 and the following *contractibility criterion*: a structure of links $G : \alpha_1 \vdash \alpha_0$ is correct when $\rightarrow_c$ reduces G to a structure N formed of a star link only, *i.e.*, $N = \alpha_1 \overset{*}{\triangleright} \alpha_0$ is a normal form (indeed, the only one) of G w.r.t. $\rightarrow_c$ (note that the rules in Figure 3 preserve the conclusions and the premises of the initial structure). By the way, this criterion yields to another characterization of proof nets [2].

$(\overset{0}{\triangleright})$ $G;\ \alpha \overset{0}{\triangleright} \beta \ \rightarrow_c\ G;\ \alpha \overset{*}{\triangleright} \beta$

$(\overset{1}{\triangleright})$ $G;\ \alpha \overset{*}{\triangleright} \beta, u_1, u_2;\ u_1, u_2 \overset{1}{\triangleright} u_0 \rightarrow_c\ G;\ \alpha \overset{*}{\triangleright} u_0, \beta$ when $u_0 \notin \alpha$

$(\overset{2}{\triangleright})$ $G;\ u_1, u_2 \overset{2}{\triangleright} u_0 \ \rightarrow_c\ G;\ u_1, u_2 \overset{*}{\triangleright} u_0$

$(\overset{*}{\triangleright})$ $G;\ \alpha_1 \overset{*}{\triangleright} \beta_1, u;\ u, \alpha_2 \overset{*}{\triangleright} \beta_2 \rightarrow_c\ G;\ \alpha_1, \alpha_2 \overset{*}{\triangleright} \beta_1, \beta_2$ when $\alpha_1 \cap \beta_2 = \alpha_2 \cap \beta_1 = \emptyset$

Figure 3. Contraction rules.

Proposition 8. *Let us say that the proof structure $G \vdash \alpha$ is c-correct when $G \rightarrow_c^* \overset{*}{\triangleright} \alpha$. Then, G is c-correct iff G is DR-correct.*

3.2 Parsing

The proof of Proposition 8 (see [2, 7, 6]) suggests a particular strategy for the application of the contractibility criterion. In fact, in any c-correct contracting structure of links G, there is at least a contraction $G \rightarrow_c G'$ such that G' is obtained by applying one of the rules in Figure 3 at a source link. In other words, the contraction rules can be applied following an uppermost strategy, see Figure 4. The relevant issue is that the rules in Figure 4 define a *parsing algorithm*, as any reduction $G \rightarrow_c^* \overset{*}{\triangleright} \alpha$ yields a sequentialization of $G \vdash \alpha$.

$(\overset{0}{\triangleright})$ $G;\ \overset{0}{\triangleright} u_1, u_2 \ \rightarrow_p\ G;\ \overset{*}{\triangleright} u_1, u_2$

$(\overset{0}{\triangleright})$ $G;\ \overset{*}{\triangleright} \alpha, u;\ u \overset{0}{\triangleright} \rightarrow_p\ G;\ \overset{*}{\triangleright} \alpha$

$(\overset{1}{\triangleright})$ $G;\ \overset{*}{\triangleright} \alpha, u_1, u_2;\ u_1, u_2 \overset{1}{\triangleright} u_0 \rightarrow_p\ G;\ \overset{*}{\triangleright} \alpha, u_0$ (if $u_0 \notin \alpha$)

$(\overset{2}{\triangleright})$ $G;\ \overset{*}{\triangleright} \alpha_1, u_1;\ \overset{*}{\triangleright} \alpha_2, u_2;\ u_1, u_2 \overset{2}{\triangleright} u_0 \rightarrow_p\ G;\ \overset{*}{\triangleright} \alpha_1, \alpha_2, u_0$ (if $\alpha_1 \cap \alpha_2 = \emptyset$)

Figure 4. Parsing rules.

Proposition 9. *Let us say that the proof structure $G \vdash \alpha$ is p-correct when $G \rightarrow_p^* \overset{*}{\triangleright} \alpha$. Then, G is p-correct iff G is DR-correct.*

The parsing rules simulate the inference rules in Figure 2. In fact, it is readily seen that $G \in \mathsf{Seq}$ iff G is p-correct. Therefore, any proof of Proposition 9 (see [7, 6]) proves the sequentialization theorem also.

$$\text{(start)} \quad \mu :: \pi \xrightarrow{G}_u \mu[u_1, u_2 \mapsto \mathit{next}(\mu)] :: (\pi;\ \mathit{next}(\mu)) \quad \text{when } \overset{0}{\triangleright}\, u_1, u_2 \in G \text{ and } \mu(u_1) = \mu(u_2) = \bot$$

$$\text{(forward)} \quad \mu :: \pi \xrightarrow{G}_u \mu[u_0 \mapsto i] :: \pi \quad \text{when } u_1, u_2 \overset{1}{\triangleright} u_0 \in G \text{ and } \pi(\mu(u_1)) = \pi(\mu(u_2)) = i$$

$$\text{(unify)} \quad \mu :: \pi \xrightarrow{G}_u \mu[u_0 \mapsto j] :: \pi[i = j] \quad \text{when } u_1, u_2 \overset{2}{\triangleright} u_0 \in G \text{ and } j = \pi(\mu(u_1)) < \pi(\mu(u_2)) = i$$

Figure 5. Unification.

4 Unification

It is a trivial programming exercise to give an algorithm equivalent to parsing that, instead of removing the vertices contracted along the reduction, marks them with a token. The rules of the algorithm are:

(start) Assign a *fresh* token to the conclusions of an unmarked axiom (axiom $\overset{0}{\triangleright}$ rule).

(forward) Assign the token t to the conclusion of a unary link whose premises yield t ($\overset{1}{\triangleright}$ rule).

(unify) When the premises of a binary link yield two distinct tokens s and t, *equate* s and t and assign s or t to the conclusion of the link ($\overset{2}{\triangleright}$ rule).

Each of the previous rules corresponds to a virtual application of a parsing rule (the one written in parentheses). In fact, let us assume that a token be a set of integer indexes; that a vertex yield the token t when it is marked by one of the indexes in t; that, in the start rule, 'fresh' mean 'the least index in the interval[2] $[0,k]$ that has not been used yet'; that, in the unify rule, 'equate' mean 'unite the sets of'. If the proof structure G has $k+1$ axioms, the application of a sequence of marking rules leads to the construction of a pair $\mu :: \pi$, where $\mu : V(G) \rightharpoonup [0,k]$ is a (partial) *marking function* and π is a partition of the range of μ, s.t.: (*i*) every structure of links $G[t]$ formed of the set of links whose premises and conclusions yield the token t is a correct proof substructure of G; (*ii*) if $G[t] \vdash \alpha[t]$, then there exists a parsing reduction $G[t] \to_p^* \overset{*}{\triangleright}\, \alpha[t]$; (*iii*) there is $G \to_p^* G'$, s.t. G' is obtained from G by replacing each proof substructure $G[t] \vdash \alpha[t]$ with $\overset{*}{\triangleright}\, \alpha[t]$.

Moreover, any parsing reduction induces a corresponding unification reduction. Namely, given $G \to_p^* G'$, there is some marking of G s.t. each star link in G' is the representation of some $G[t]$.

[2]For $h \geq 0$, $[0,h]$ denotes the closed interval of $\mathbb{N}$ from 0 to h; $[0,h[$ denotes instead the corresponding right open interval, *i.e.*, $[0,h[= [0,h] \setminus \{h\}$.

Figure 5 defines the rewriting system (parametric in G) that, when the reduction starts with the empty marking $() :: ()$, derives the valid markings of G. Thus, let us write $\downarrow^G (\mu :: \pi)$, when $() :: () \xrightarrow{G}_u{}^* \mu :: \pi$.

In the rules of Figure 5, a marking function is represented by means of a list of pairs $\mu[i] \mapsto i$, where $\mu[i] = \{u \in V(G) \mid \mu(u) = i\}$, *i.e.*, $\mu = (\mu[0] \mapsto 0;\ \ldots;\ \mu[h-1] \mapsto h-1)$; while a partition is represented by means of a list of disjoint sets π_i, *i.e.*, $\pi = (\pi_1;\ \ldots;\pi_l)$. Moreover, we use the following notations: (*i*) if μ has range $[0,h[$, then $\mathit{next}(\mu) = h$; (*ii*) $\mu(u) = \bot$, if $\mu(u)$ is undefined; (*iii*) $\mu[\alpha \mapsto i]$ is the marking μ' s.t. $\mu'(u) = i$, when $u \in \alpha$, and $\mu'(u) = \mu(u)$, otherwise; (*iii*) $\pi(i)$ is the canonical representative of the set containing i, say its least element; (*iv*) if $\pi(i) \neq \pi(j)$, then $\pi[i = j]$ is the partition obtained from π by merging the set containing i and the set containing j.

Figure 6 gives a pictorial account of the unification rules. In that picture, π and π' are the partitions before and after the rewriting; the relevant values of the marking functions are drawn in the place of the corresponding vertices.

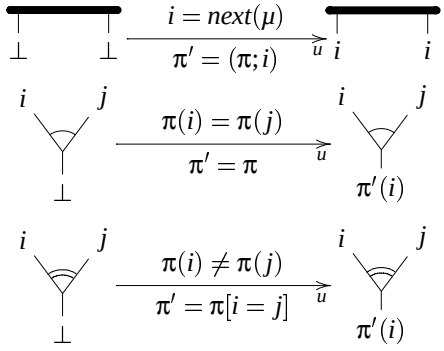


Figure 6. Unification: a pictorial account.

Remark 10. Let ρ be a unification reduction s.t. $\downarrow^G (\mu :: \pi)$, and t be some token in π. The *thread* corresponding to t is the subset of the rewriting rules in ρ

that assign an index $i \in t$ to some vertex of G. Each thread of ρ spans the proof substructure $G[t]$. Then, interpreting each thread as an independent unification process running in parallel with the other threads, we see that each start rule creates a new thread, and that the unify and forward rules are thread synchronization directives: forward asks for the synchronization of the threads of its premises; unify signals that the threads of its premises have synchronized and unites them into a unique thread.

When G has $k+1$ axioms and $\downarrow_G (\mu :: \pi)$, the range of μ is an interval $[0,h[$ with $h \leq k+1$, and π is a partition of $[0,h[$ with a class for each thread of the reduction. Then, G is a proof net iff there is a marking pair $\mu :: \pi$ of G s.t. μ marks all the vertices of G and π equates all the tokens in the range of μ; that is, G is a proof net iff unification ends up with a unique thread that spans all G.

Definition 11 (u-correctness). Let G be a proof structure with $k+1$ axiom links. A marking function μ is a *unifier* of G when μ is a total function onto $[0,k]$ and $\downarrow^G (\mu :: 0,\ldots,k)$. The proof structure G is u-*correct* when it has a unifier.

Proposition 12. *A proof structure is* u-*correct iff it is* DR-*correct.*

5 Linear time unification

The worst case for the unification algorithm is when the proof structure G is correct—in that case, every unification reduction of G requires $|G|$ steps. Nevertheless, this does not yet implies linearity. In fact, for a correct evaluation of the computational cost of unification, we must take into account the cost of choosing the rule that applies and the cost of the operations on the partition π—the marking function μ is not a problem, it is an index field in the records used to represent links.

5.1 Disjoint-set union-find

The data structure implementing the partition π must be optimized for the so-called *disjoint-set union-find* operations [1, Chapter 22]:

FINDSET(i): It computes the least element in the token of i (*i.e.*, it computes $\pi(i)$).

UNION(i,j): It merges the tokens of i and j and leaves the other tokens unchanged (*i.e.*, it computes $\pi[i=j]$, provided that $\pi(i) \neq \pi(j)$).

MAKESET(i): It adds the token $\{i\}$ to π (*i.e.*, it computes $(\pi; i)$, provided that $\pi(i) = \bot$).

Efficient data structures for (disjoint-set) union-find have been widely studied and used (*e.g.*, in term unification [8]). The main property of the corresponding algorithms, also know as α-algorithms, is that the overall cost of m FINDSET, UNION and MAKESET operations is $\mathsf{UF}(m,n) = O(m\,\alpha(m,n))$, where n is the number of MAKESET and α is a very slowly increasing function—in terms of growth slope, α is the inverse of the Ackermann function.

Remark 13. From a theoretical point of view, an α-algorithm is not linear, for $\alpha(h,k)$ is not a constant. Nevertheless, in any conceivable application, $\alpha(h,k) < 4$. In fact, $\alpha(h,k) = \min\{i \geq 1 : \mathsf{Ack}(i, \lfloor h/k \rfloor) > \lg k\}$, where Ack (the Ackermann function) is defined by: $\mathsf{Ack}(1,j) = 2^j$; $\mathsf{Ack}(i,1) = \mathsf{Ack}(i-1,2)$; $\mathsf{Ack}(i,j) = \mathsf{Ack}(i-1,\mathsf{Ack}(i,j-1))$; in particular, $\mathsf{Ack}(2,n)$ is a tower of exponential of length n. Then, for every $h \geq k$, $\mathsf{Ack}(4,\lfloor h/k \rfloor) \geq \mathsf{Ack}(4,1) = \mathsf{Ack}(2,16)$, that is far greater than the estimated number of atoms in the observable universe (roughly 10^{80}).

5.2 Ready and waiting links

During unification, a link is *armed* when both its premises yield a token, and is *idle* otherwise. Each forward or unify rule *fires* an armed link and, because no rule can erase or change the token assigned to a vertex, a link cannot be fired twice. Not every armed link can be fired. For instance, an armed binary link whose premises yield the same token is a *deadlock*, for in no case unification will be able to fire it. At the same time, neither an armed unary link whose premises yield distinct tokens can be fired, but, since a following unify might unite the tokens of its premise, that unary link is in a *waiting* state. An armed link that is not waiting nor is a deadlock is *ready* to be fired. A vertex is in the same state of the link above it, *e.g.*, a vertex is ready when it is the conclusion of a ready link.

The unification algorithm consists of a main loop that picks a ready vertex (link) and fires one of the rules in Figure 5. During this process, unification enquires and updates the set of the vertices that are ready and the set of the vertices that are waiting, say R and W respectively. In particular, after removing and marking a ready vertex v from R, one of the following two cases applies: (*i*) if the marking of v activates a unary link with conclusion w, then insert w in R or W, according to its state; (*ii*) if the marking of

v activates a binary link with conclusion w and w is not a deadlock, then apply a unify rule, put w in R and move the waiting vertices become ready from W to R. The latter is the critical operation: if W has no structure, finding the waiting vertices that have become ready requires scanning all W. As a consequence, a flat implementation of W causes a linear cost of unify, and an overall quadratic cost.

5.3 Sequential unification

The rules in Figure 5 defines a parallel unification algorithm: there is a thread for each token and the threads run in parallel. Unfortunately, this parallel point of view does not give any help implementing the data structures R and W, as vertices are inserted and moved from R and W in no particular order. Instead, an efficient implementation of R and W requires finding a good unification strategy; in particular, it requires controlling thread creation.

During unification, the start rules number axioms (the index of an axiom is that of its conclusion) according to the order in which they are visited. Therefore, the token of i is older than the token of j when $\pi(i) < \pi(j)$, and similarly for the corresponding threads. Now, let us say that a ready link belongs to a thread if the corresponding rule belongs to that thread (*e.g.*, a ready unary link belongs to the thread of its premises token). We want to control the order in which the links are fired by giving priority to the youngest thread of the reduction—the youngest thread (token) of the reduction is its *active thread* (token) and a ready link is active when it belongs to the active thread.

After an initialization step that fires an axiom (any one), we may apply the following *sequential strategy*: (*i*) repeat firing an active link, if any, as long as you do not find a vertex v that is the premise of a binary link whose other premise w is not marked; (*ii*) if step (*i*) ends finding the vertex v, fire an axiom above w (*i.e.*, start a new thread) and return to step (*i*).

The sequential strategy ensures that threads are united according to their age. In fact, let the thread of i create a new thread assigning the index j to some axiom. By construction, $j > x$ for every x in the token of i. Moreover, there is a binary link whose commitment is to unite the thread of i and the thread of j. Then, let us assume that j create a new thread assigning the index $j+1$ to some axiom; there is a binary link whose commitment is to unite the threads of j and $j+1$. After some steps, let the thread of $j+1$ unite with the thread of i. If j and $j+1$ are not in the same thread, there are two binary links committed to unite the same pair of threads—the thread of j and the thread of i and $j+1$. Therefore, unification will eventually find a deadlock.

Let $\downarrow^G(\mu :: \pi)$ be obtained according to the sequential strategy. If $[0,h[$ is the range of μ, there is a sequence $i_0 < \cdots < i_n < \cdots < i_{l+1}$, with $i_0 = 0$ and $i_{l+1} = h$, s.t. π splits $[0,h[$ in the subintervals $[i_0,i_1[$, $[i_1,i_2[$, $\ldots$ $[i_l,i_{l+1}[$. Now, let us represent π by means of the stack $\sigma = i_0 : \cdots : i_n : \cdots : i_l$ and write $\sigma(i) = i_n$ when $i_n \leq i < i_{n+1}$. The ready vertices can be arranged into a stack of sets $R = \rho_0 : \cdots : \rho_n : \cdots : \rho_l$ s.t. $v \in \rho_n$ iff v is the conclusion of a unary link belonging to the thread of i_n. Each unify merges the intervals $[i_{l-1},i_l[$ and $[i_l,h[$, and activates the vertices in ρ_{l-1}, *i.e.*, it pops i_l from σ and merges the two sets on the top of R.

Using the sequential strategy has a consequence on the structure of the waiting vertices too. Let us say that v_0 is *waiting for* i when it is the conclusion of a unary link $v_1, v_2 \overset{1}{\rhd} v_0$ s.t. $i = \pi(\mu(v_1))$ and $\mu(v_1) < \mu(v_2)$ (note that this means $\pi(\mu(v_1)) < \pi(\mu(v_2))$ also); we can assume that W is a function (or an array) s.t.: for $j = i_0, \ldots, i_{l-1}$ (we recall that $\sigma = i_0 : \cdots : i_l$), then $W(j)$ is the set of vertices that are waiting for j; otherwise, $W(j) = \bot$. Since unify unites the tokens of i_l and i_{l-1}, after a unify, we have that: (*i*) the vertices in $W(i_{l-1})$ become ready and active, *i.e.*, they must be added to the set on the top of R; (*ii*) for $n < l-1$, the vertices in $W(i_n)$ keep waiting for i_n.

The latter considerations lead to the *sequential unification* algorithm in Figure 7. The following notations are new: (*i*) $W' = W[i \leftarrow u]$, with the proviso $W(i) \neq \bot$, means that $W'(i) = W(i), u$, while $W'(j) = W(j)$, if $j \neq i$; (*ii*) $W' = W[i \mapsto \bot]$ and $W' = W[i \mapsto \emptyset]$ set the value of $W'(i)$ to $\bot$ and to $\emptyset$, respectively, leaving $W'(j) = W(j)$, for $j \neq i$ (we stress that $W(j) = \emptyset$ means that the jth stack has been initialized and is empty, while $W(j) = \bot$ means that the jth stack is undefined, therefore no operation can be done on it).

Figure 8 gives a pictorial account of sequential unification. We have add the case in which the vertex v popped from R is the premise of a deadlock; by the way, that corresponds to an incorrect proof structure.

Sequential unification rewrites four-tuples with the shape $\mu :: \sigma :: W :: R$. Again, $\Downarrow^G(\mu :: \sigma :: W :: R)$ iff $((() :: () :: () :: ())) \xrightarrow{G}{}^*_s (\mu :: \sigma :: W :: R)$, where $\xrightarrow{G}{}^*_s$ is the transitive and reflexive closure of $\xrightarrow{G}{}_s$.

For technical reasons, in the rules of sequential unification, we do not mark a vertex immediately after firing the link above it; instead, we put it into R. That simplifies the specification of the algorithm,

$$\text{(init)} \quad () :: () :: () :: () \xrightarrow{G}_s () :: (0) :: () :: (u_1, u_2) \quad \text{when } \overset{0}{\triangleright} u_1, u_2 \in G$$

$$\text{(concl)} \quad \mu :: (\sigma : i) :: W :: (R : \rho, u) \xrightarrow{G}_s \mu[u \mapsto i] :: (\sigma : i) :: W :: (R : \rho) \quad \text{when } G \vdash \alpha, u \vee u \overset{0}{\triangleright} \in G$$

$$\text{(nop)} \quad \mu :: (\sigma : i) :: W :: (R : \rho, u_1) \xrightarrow{G}_s \mu[u_1 \mapsto i] :: (\sigma : i) :: W :: (S : \rho) \quad \text{when } u_1, u_2 \overset{1}{\triangleright} u_0 \in G \text{ and } \mu(u_2) = \bot$$

$$\text{(wait)} \quad \mu :: (\sigma : i) :: W :: (R : \rho, u_1) \xrightarrow{G}_s \mu[u_1 \mapsto i] :: (\sigma : i) :: W[\sigma(\mu(u_2)) \hookleftarrow u_0] :: (R : \rho) \quad \text{when } u_1, u_2 \overset{1}{\triangleright} u_0 \in G \text{ and } \mu(u_2) < i$$

$$\text{(forward)} \quad \mu :: (\sigma : i) :: W :: (R : \rho, u_1) \xrightarrow{G}_s \mu[u_1 \mapsto i] :: (\sigma : i) :: W :: (R : \rho, u_0) \quad \text{when } u_1, u_2 \overset{1}{\triangleright} u_0 \in G \text{ and } \mu(u_0) = \bot \text{ and } \mu(u_2) \geq i$$

$$\text{(new)} \quad \mu :: (\sigma : i) :: W :: (R : \rho, u_1) \xrightarrow{G}_s \mu[u_1 \mapsto i] :: (\sigma : i : j) :: W[j \mapsto \emptyset] :: (R : \rho : v_1, v_2) \quad \text{when } u_1, u_2 \overset{2}{\triangleright} u_0 \in G \text{ and } \mu(u_2) = \bot \text{ and } u_2 \preccurlyeq v_1 \text{ and } \overset{0}{\triangleright} v_1, v_2 \in G \text{ and } j = \mathit{next}(\mu) \text{ and } \mu(v_1) = \mu(v_2) = \bot$$

$$\text{(unify)} \quad \mu :: (\sigma : j : i) :: W :: (R : \rho' : \rho, u_1) \xrightarrow{G}_s \mu[u_1 \mapsto i] :: (\sigma : j) :: W[j \mapsto \bot] :: (R : u_0, W(j), \rho', \rho) \quad \text{when } u_1, u_2 \overset{2}{\triangleright} u_0 \in G \text{ and } j \leq \mu(u_2) < i$$

Figure 7. Sequential unification.

preserving the property that each set in R contains the vertices that are ready to be marked with the corresponding index in σ. Indeed, apart for init, each rule in Figure 7 pops a vertex u from R, marks it with the top index of σ, verifies the state of the link below u and, according to that state, performs some operations on $\mu :: \sigma :: W :: R$. Namely, let u be the premise of a unary link with conclusion v; wait inserts v in the proper set of W, for the link below u is waiting; forward inserts v in the top set of R, for the link below u is ready; and so on. We remark new: it implements step (*ii*) of the sequential strategy.

Remark 14. We recall that R is a stack of sets, *i.e.*, $R = \alpha_0 : \alpha_1 : \ldots : \alpha_k$. Therefore, the vertex u popped from R is any vertex in α_k. We also remark that, when the top set of R is empty, no rule of sequential unification applies. In particular, the case in which $k > 0$ and $\alpha_k = \emptyset$ corresponds to a deadlock and cannot arise in a correct proof structure.

Neglecting the cost of the union-find operations, all the sequential unification rules but new can be executed in $O(1)$ time. That is not true for new, as it requires to go up along the net until we find an axiom. The following procedure implements that search.

NEXTAXIOM(v)
1: **if** $\mathit{tag}(v) = \mathsf{true}$ or $\mu(v) \neq \bot$ **then**
2: **return** error
3: **else if** $\overset{0}{\triangleright} v', v \in G$ **then**
4: $\mathit{tag}(v') \leftarrow \mathsf{true}$; $\mathit{tag}(v) \leftarrow \mathsf{true}$;
5: **return** v, v';
6: **else if** $v', v'' \overset{1,2}{\triangleright} v \in G$ **then**
7: $\mathit{tag}(v) \leftarrow \mathsf{true}$;
8: **return** NEXTAXIOM(v');

During the search, NEXTAXIOM sets a tag associated to each vertex. That tag, initially equal to false, is true when the vertex v has been already visited by some NEXTAXIOM or when v is the conclusion of the axiom fired by init—in practice, after initializing all the tags to false, we can assume that the starting axiom is found calling NEXTAXIOM(v), where v is any vertex. According to this, NEXTAXIOM(v) returns an error whenever v has been already visited, *i.e.*, $\mu(v) \neq \bot$ or $\mathit{tag}(v) = \mathsf{true}$ (if NEXTAXIOM is properly called by init and by new only, both that cases are not possible in a proof net). We stress that the use of tags ensures that NEXTAXIOM cannot loop (by the way, this might happen in an incorrect proof structure only) and that, during sequential unification, either NEXTAXIOM does not visit twice any vertex or it stops after finding an already marked vertex.

If $\Downarrow^G (\mu :: 0 : i_1 \cdots : i_l :: W :: R)$ and $\mathit{next}(\mu) = h$, then $\downarrow^G (\mu :: [0, i_1[; [i_1, i_2[; \ldots ; [i_l, h[)$. Moreover, if G is a proof net and μ is not a unifier, the top set of R is not empty. Hence, sequential unification of a proof net cannot stop before finding a unifier.

Proposition 15. *The proof structure G is* DR*-correct iff $\Downarrow^G (\mu :: 0 :: () :: ())$ and μ marks every vertex of G (moreover, μ is a unifier of G).*

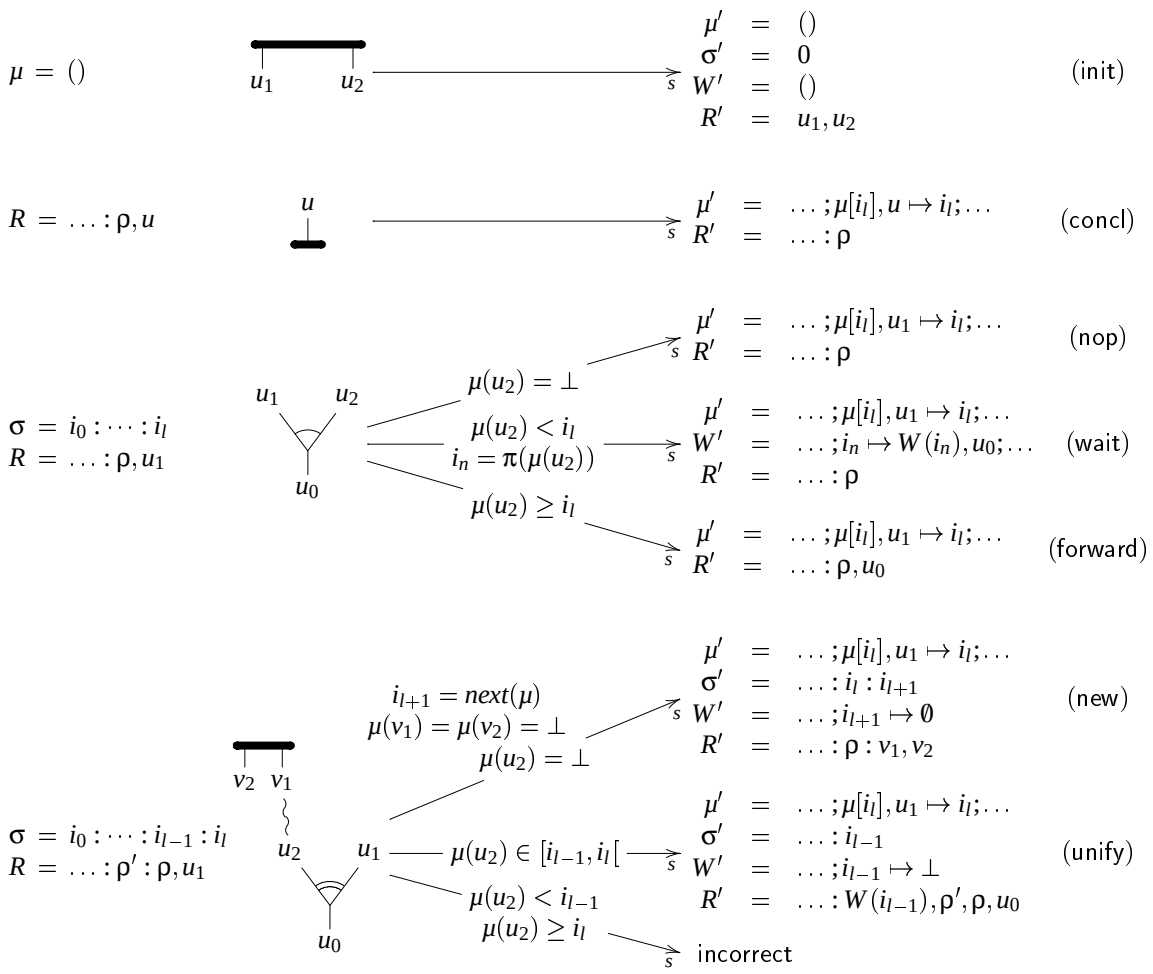


Figure 8. Sequential unification: a pictorial account.

5.4 A special case of union-find

The α-algorithms for union-find are practically linear (see Remark 13). Therefore, since the constants in the upper-bound are small, a pseudo-linear implementation using an α-algorithm is frequently preferred to a linear solution whose upper-bound requires bigger constants (*e.g.*, this is the case of term unification). However, there is a special case—and the disjoint-set union-find used by sequential unification is an instance of it—where the amortized cost of union-find becomes linear without any particular increasing of the upper bound constants [4]. The linear algorithm for that special case uses an α-algorithm, but, exploiting the order in which sets are united, optimizes the size of the problem on which to apply the α-algorithm. Let us see how that technique applies in the case of sequential unification.

The natural data structure for the implementation of $\sigma = i_0 : \cdots : i_n : \cdots : i_l$ is an array of bits b s.t. $b[i] = 0$, for $i = i_0, \ldots, i_l$, and $b[i] = 1$, otherwise. In this way, setting $b[i_n]$ to 1 unites the intervals $[i_{n-1}, i_n[$ and $[i_n, i_{n+1}[$, while $\sigma(i)$ is the greatest $j \leq i$ s.t. $b[j] = 0$. In order to optimize the use of memory registers, we represent b by means of an array B of words of length w and define $b[i] = bit(i \mod w, B[i \div w])$, where $bit(n,x)$ is the nth bit of x (start counting bits from 0). Computing $\sigma(i)$ decomposes in the following steps: (*i*) check if the bit of $\sigma(i)$ is in the word $i \div w$, *e.g.*, by means of the function *firstz*(n,x) that returns the greatest $m \leq n$ s.t. $bit(m,x) = 0$, if any, and -1 otherwise; (*ii*) if *firstz*$(i \mod w, i \div w) = -1$, find the greatest $n < i \div w$ s.t. $B[n]$ contains a bit equal to 0; (*iii*) return *firstz*$(i \mod w, B[i \div w])$, if this is not equal to -1, or *firstz*$(w-1, B[n])$, otherwise.

Step (*ii*) is the critical operation. Before to analyze it, let us remark that *firstz*(n,x) is $O(1)$. For

instance, let us assume that the processor compute $bor(x,y)$ (the bitwise-or of the words x and y) in $O(1)$ time[3]; let $x = mask(i)$ be the word s.t. $bit(j,x) = 0$, for $j \leq i$, and $bit(j,x) = 1$, for $j > i$; let $lastz(x)$ be equal to the greatest j s.t. $bit(j,x) = 0$, if any, and equal to -1, otherwise. Then, $firstz(i,x) = lastz(bor(mask(i),x))$. Now, *mask* and *lastz* can be implemented by means of two tables of length $O(w)$ and $O(2^w)$, respectively. That is, at the cost of a $O(2^w)$ initialization, $firstz(n,x)$ is $O(1)$.

The efficient algorithm for $\sigma(i)$ uses a disjoint-set data structure for the implementation of step (ii). In fact, let $j_0 < \cdots < j_n < \cdots < j_k$ be the sequence of indexes s.t.: if $j = j_n$, then $B[j]$ contains at least a 0; otherwise, $B[j]$ does not contain any 0 (by hypothesis, $j_0 = 0$, for in any case $b[0] = 0$). If $j_{k+1} = L$ is the length of B, then B splits into the family of disjoint intervals $[j_0, j_1[$, $[j_1, j_2[$, $\ldots$, $[j_k, j_{k+1}[$, and step (ii) is a FINDSET of the corresponding disjoint-set problem. Hence, using an α-algorithm, the cost of n find/set-bit operations on b is $O(n\ \alpha(n+L,L))$, plus $O(2^w)$ for the initialization of $firstz(n,x)$. Then, if N is the length of b and $w = \Omega(\log\log N)$ (but even smaller values of w suffice [10]), $\alpha(n+L,L) = \alpha(n+O(N/\log\log N), O(N/\log\log N)) = O(1)$; that is, n union-find operations on b cost $\mathsf{UF}(n,N) = O(n) + O(2^w)$.

5.5 Sequential unification is linear

Any sequential unification of G requires at most $|V(G)|$ steps—no vertex can be inserted twice into R and, when G is a proof net, every vertex transits into R; moreover, as NEXTAXIOM does not visit twice the same vertex, the amortized cost of its calls is $O(size(G))$. The array b has a bit for each axiom, *i.e.*, $N = O(size(G))$. The number of find-bit operations is bound by the number of unary links (wait is the only rule that requires a find-bit); the number of set-bit operations is bound by the number of binary links. Therefore, sequential unification costs $O(size(G)) + \mathsf{UF}(2\ size(G), size(G))$. Moreover, when $w \leq \log size(G)$ and $w = \Omega(\log\log size(G))$,[4] the cost simplifies to $O(size(G)) + O(size(G)) + O(size(G))$.

Theorem 16. *The cost of any sequential unification of a proof structure G is $O(size(G))$.*

[3] If n is the size of the problem, one of the basic hypotheses of the RAM model is that its word length is $O(\log n)$. Under that hypothesis, the arithmetic and bitwise operations on words have cost $O(1)$.

[4] That assumptions on w are compatible with the basic hypothesis $w = O(size(G))$, see footnote 3.

6 Conclusions

The relevant point of sequential unification is that it is not a theoretical trick. All the steps leading to its linear algorithm correspond to natural optimizations that do not increase the constants in the upper-bound.

Apart for the result on the complexity of correctness, the unification criterion is interesting from a semantical point of view. The operations that unification performs at each ⅋/⊗-link correspond to synchronization directives: a ⅋-link asks for the synchronization of its premise; a ⊗-link (or cut) notifies that its premises have synchronized. One of the point under investigation is if the interpretation of that directives in some kind of process algebra may lead to an interesting semantics of proof nets in terms of concurrent processes.

References

[1] T. H. Cormen, C. H. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT electrical engineering and computer science series. The MIT Press, 1989.

[2] V. Danos. *Une Application de la Logique Linéaire à l'Étude des Processus de Normalisation (principalement du λ-calcul)*. PhD Thesis, Université Paris 7, June 1990.

[3] V. Danos and L. Regnier. The structure of multiplicatives. *Archive for Mathematical Logic*, 28:181–203, 1989.

[4] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985.

[5] J.-Y. Girard. Linear logic. *Theoretical Comput. Sci.*, 50(1):1–102, 1987.

[6] S. Guerrini and A. Masini. Parsing MELL Proof Net. *Theoretical Comput. Sci.*, 1999. To appear.

[7] Y. Lafont. From proof nets to interaction nets. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, pages 225–247. Cambridge University Press, 1995. Proceedings of the Workshop on Linear Logic, Ithaca, New York, June 1993.

[8] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(3):258–282, Apr. 1982.

[9] M. Nagayama and M. Okada. A new correctness criterion for the proof-nets of non-commutative multiplicative linear logic. Draft available at `http://www.twcu.ac.jp/~misao/papers.html`, Nov. 1998.

[10] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.