Unification and anti-unification

a report in continuation of the course
”Logik programmering”

Erik Jacobsen
740500

June 6, 1991

Abstract

Unification as defined by Robinson [Rob65] is one of two dual concepts, the other is called anti-unification. These concepts are presented in terms of category theory and concepts from algebra.

The unification algorithm from [Rob65] is presented and I argue that it has a worst-case exponential runtime. Modifying this algorithm, I remove one source of exponential runtime. Paterson & Wegman showed in [PW78] that a linear unification algorithm exists. I present this algorithm and compare actual runtimes for all algorithms.

I then present an algorithm for anti-unification by Plotkin [Plo69] and Reynolds [Rey69]. This algorithm has $O(n^2)$ runtime. Two new linear algorithms, based on the same principle as the linear Paterson & Wegman unification algorithm, are presented. Finally a parallel anti-unification algorithm from [KMPP88] is presented.

Contents

Chapter 1

Introduction

This report is written in continuation of the course ”Logik Programmering E86” given by Gudmund Frandsen. The background for the report is that I want to

- see how efficient the linear unification algorithm by Paterson & Wegners is.
- use the things I learned in another course on Category Theory.

Chapter 2 is an attempt to present unification, and its dual anti-unification, in terms of Category Theory and other concepts from higher algebra. The main results are that unification and anti-unification are coproduct and product in the category **Term**, and that terms form a semi-lattice with anti-unification as the meet-operation. A lattice with unification as the join-operation can also be constructed.

Chapter 3 presents unification algorithms. I investigate Robinsons original algorithm, with and without occurcheck, and I modify it to make the occurcheck more efficient – this removes the potential exponential runtime for the occurcheck. The modified algorithm is still worst-case exponential, so finally the linear unification algorithm by Paterson & Wegman is presented.

I have implemented all the algorithms, and I conclude the chapter by comparing actual runtimes on selected terms.

Chapter 4 presents anti-unification algorithms. Plotkin [Plo69] and Reynolds [Rey69] were the first to present (almost identical) algorithms for anti-unification, and this is presented. Two new linear algorithms, inspired by the Paterson & Wegman algorithm, are then presented.

The process of unification cannot profitably be executed on parallel processors, but anti-unification can, and I present a parallel algorithm from [KMPP88].

None of the anti-unification algorithms have been implemented.

The intended reader of this report should have knowledge of

- Logic programming, e.g. corresponding to the course "Logik programmering" (course-notes: [Fra86]).
- Elementary category theory, e.g. corresponding to chapters 1 and 2 of Arbib & Manes: "Arrows, Structures, and Functors" [AM75].
- Lattices, posets etc., e.g. corresponding to MacLane & Birkoff: "Algebra", chapter XIV, sections 1 and 2 [MB79].

A good survey article about unification is [Kni89].

I want to thank my supervisor Gudmund Frandsen for advice and suggestions, and for his patience.

Erik Jacobsen, June 1991

Chapter 2

Theory

This chapter summarizes the contents of several articles:

- Definitions and theorems about substitutions come from [Rob65] and [LMM86].
- The lattice structure of terms comes from [Rey69].
- The category **Term** is from [Plo69].

2.1 Terms

Definition 2.1.1 *Variable.* A variable is represented by an identifier starting with an uppercase letter – it can be subscripted. □

Example 2.1.2 The following are variables:

$$A,\ B,\ C,\ A_1,\ B_2,\ C_3$$

$\square$

Definition 2.1.3 *Function symbol.* A function symbol is represented by an identifier starting with a lowercase letter – it can be subscripted. Function symbols taking n arguments are called *n-ary.* 0-ary function symbols are *individual constants.* $\square$

Example 2.1.4 The following are function symbols.

$$f,\ g,\ h,\ f_1,\ g_2,\ h_3$$

$\square$

Definition 2.1.5 *Term.* A term is either

- a variable, or
- an n-ary function symbol with n terms as arguments.

The set of variables occuring in a term t is called $var(t)$. $\square$

Example 2.1.6 The following are terms:

$$f,\ A,\ g(A, f)$$

$\square$

Example 2.1.7 Let $t = f(g(X, h), X, g(h, Z))$. Then $var(t) = \{X, Z\}$. The term t may be viewed as a tree, see figure 2.1. The nodes in the tree represent the function-symbols, and the leaves represent variables and individual constants. $\square$

For simplicity we will assume in the following that whenever two nodes have the same function-symbol, they also have the same arity.

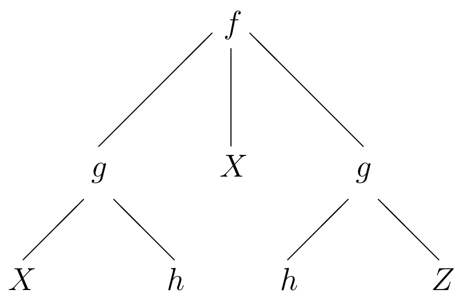

Figure 2.1: The term $f(g(X, h), X, g(h, Z))$ viewed as a tree.

2.2 Substitutions

Definition 2.2.1 *Substitution.* A (non-ordered) substitution is a finite set of pairs of variables and terms:

$$\theta = \{v_1/t_1, \ldots, v_n/t_n\},\ n < \infty \tag{2.1}$$

such that $i \neq j \Rightarrow v_i \neq v_j$. If $n = 1$ we call it a *single substitution.* □

Definition 2.2.2 *Domain and range.* Let $\theta = \{v_1/t_1, \ldots, v_n/t_n\}$ be a substitution. We define

$$\begin{aligned} \mathit{domain}(\theta) &= \{v_1, \ldots, v_n\} && (2.2)\\ \mathit{range}(\theta) &= \bigcup_1^n \mathit{var}(t_i) && (2.3) \end{aligned}$$

□

Definition 2.2.3 *Ordered substitution.* An ordered substition is a finite sequence of single substitutions:

$$\eta = \{v_1/t_1\}\{v_2/t_2\} \cdots \{v_n/t_n\},\ n < \infty \tag{2.4}$$

□

Definition 2.2.4 *Applying substitutions.* The non-ordered substitution θ applied to a term t, written $t\theta$, is the term t where each occurrence of variables in t is replaced with its substitution in θ:

$$t\theta = \begin{cases} f(t_1\theta, \ldots, t_k\theta) & \text{if } t = f(t_1, \ldots, t_k) \\ t' & \text{if } t = v \text{ and } v/t' \in \theta \\ v & \text{if } t = v \text{ and } v/t' \notin \theta \end{cases} \tag{2.5}$$

where f is a k-ary function-symbol and v represents a variable.

When applying a sequence of substitutions η to a term t, written $t \circ \eta$, we must take each substitution in order:

$$t \circ \eta = \begin{cases} t\{v_1/t_1\} \circ \{v_2/t_2\} \cdots \{v_n/t_n\} & \text{if } \eta = \{v_1/t_1\} \cdots \{v_n/t_n\},\ 0 < n \\ t & \text{if } \eta = \{\ \} \end{cases} \tag{2.6}$$

□

Definition 2.2.5 *Composition of substitutions.* The composition of two substitutions

$$\theta_1 = \{v_1/t_1, \ldots, v_n/t_n\} \tag{2.7}$$

$$\theta_2 = \{w_1/s_1, \ldots, w_k/s_k\} \tag{2.8}$$

is

$$\theta_1\theta_2 = \{v_1/t_1\theta_2, \ldots, v_n/t_n\theta_2, w_1/s_1, \ldots, w_k/s_k\} \tag{2.9}$$

If $\exists i, j :\ w_i = v_j$, then the pair w_i/u_i is omitted. □

Lemma 2.2.6 Definition 2.2.5 is welldefined, i.e. the composition of two substitutions is again a substitution.

Proof: This is easy. Obviously $\theta_1\theta_2$ is a finite set, and per construction the variable-parts of the variable/term-pairs are disjoint, as required by definition 2.2.1. □

Example 2.2.7 Composition of substitutions is not commutative. A very simple example of this is:

$$\theta_1 = \{a/b\} \tag{2.10}$$
$$\theta_2 = \{a/c\} \tag{2.11}$$

Applying definition 2.2.5 we get

$$\theta_1\theta_2 = \{a/b\} \tag{2.12}$$
$$\theta_2\theta_1 = \{a/c\} \tag{2.13}$$

□

Lemma 2.2.8 Application of substitutions is distributive, i.e.:

$$t(\theta_1\theta_2) = (t\theta_1)\theta_2 \tag{2.14}$$

Proof: It is enough to show that $v(\theta_1\theta_2) = (v\theta_1)\theta_2$ for any variable v. This gives us 3 cases:

a) If $v/t_v \in \theta_1$, then $v(\theta_1\theta_2) = t_v\theta_2 = (v\theta_1)\theta_2$.

b) If $v/t_v \in \theta_2 \wedge v \notin \mathit{domain}(\theta_1)$, then $v(\theta_1\theta_2) = t_v = v\theta_2 = (v\theta_1)\theta_2$.

c) If $v \notin \mathit{domain}(\theta_1) \cup \mathit{domain}(\theta_1)$, then $v(\theta_1\theta_2) = v = (v\theta_1)\theta_2$.

□

Lemma 2.2.9 Composition of substitutions is associative, i.e.:

$$(\theta_1\theta_2)\theta_3 = \theta_1(\theta_2\theta_3) \tag{2.15}$$

Proof: A little healthy exercise in symbol-manipulation. Let

$$\theta_1 = \{v_1/t_1, \ldots, v_n/t_n\} \tag{2.16}$$
$$\theta_2 = \{w_1/s_1, \ldots, w_k/s_k\} \tag{2.17}$$
$$\theta_3 = \{u_1/r_1, \ldots, u_m/r_m\} \tag{2.18}$$

Then we have

$$\begin{aligned}
\theta_1\theta_2 &= \{v_1/t_1\theta_2, \ldots, v_n/t_n\theta_2, w_1/s_1, \ldots, w_k/s_k\} && (2.19)\\
(\theta_1\theta_2)\theta_3 &= \{v_1/t_1\theta_2\theta_3, \ldots, v_n/t_n\theta_2\theta_3, w_1/s_1\theta_3, \ldots, w_k/s_k\theta_3, \\
&\quad u_1/r_1, \ldots, u_m/r_m\} && (2.20)\\
\theta_2\theta_3 &= \{w_1/s_1\theta_3, \ldots, w_k/s_k\theta_3, u_1/r_1, \ldots, u_m/r_m\} && (2.21)\\
\theta_1(\theta_2\theta_3) &= \{v_1/t_1\theta_2\theta_3, \ldots, v_n/t_n\theta_2\theta_3, w_1/s_1\theta_3, \ldots, w_k/s_k\theta_3, \\
&\quad u_1/r_1, \ldots, u_m/r_m\} && (2.22)
\end{aligned}$$

where (2.20) and (2.22) show us what we want (we still have to remove the rightmost of any pair with the same variable-part in all the equations, but that will make no difference). □

Lemma 2.2.10 The set of substitutions with composition as operation form a *monoid.*

Proof: From lemma 2.2.6 we see that substitutions are closed under composition, and from lemma 2.2.9 we get that composition is an associative binary operation. The unit element is $\epsilon = \{\}$, since

$$\forall\theta : \epsilon\theta = \theta\epsilon = \theta \qquad (2.23)$$

□

Example 2.2.11 Obviously the set of substitutions with composition does not form a *group*, since substitutions generally does not have inverses. The only substitutions that do have inverses, are those described in example 2.5.5. □

Definition 2.2.12 *Idempotent substitution.* A substitution θ is idempotent if

$$\theta = \theta\theta \qquad (2.24)$$

□

Lemma 2.2.13 A substitution θ is idempotent if and only if $\mathit{domain}(\theta) \cap \mathit{range}(\theta) = \emptyset$.

Proof: $\Rightarrow$: (contradiction) Assume $\mathit{domain}(\theta) \cap \mathit{range}(\theta) \neq \emptyset$. Let $Y \in \mathit{domain}(\theta) \cap \mathit{range}(\theta)$ and $Z/t \in \theta$, such that $Y \in \mathit{var}(t)$. Then $Z\theta = t$ and $(Z\theta)\theta = t\theta$. But $t \neq t\theta$ since $y \in \mathit{var}(t)$, and $\exists Y/t' \in \theta : Y \neq t'$, so θ is not idempotent.

$\Leftarrow$: Let $V \in \mathit{domain}(\theta)$. Since $\mathit{var}(V\theta) \subseteq \mathit{range}(\theta)$ we have that $\mathit{var}(V\theta) \cap \mathit{domain}(\theta) = \emptyset$. Thus $V\theta = (V\theta)\theta$, so $\theta = \theta\theta$. $\square$

Example 2.2.14 The composition of two idempotent substitutions is not necessarily idempotent. Consider

$$\theta_1 = \{X/f(Y)\} \tag{2.25}$$
$$\theta_2 = \{Y/X\} \tag{2.26}$$

Then

$$\theta_1\theta_2 = \{X/f(X)\} \tag{2.27}$$

$\square$

Robinsons unification algorithm – to be introduced later – will produce idempotent substitutions, see example 3.1.5.

2.3 Unification and anti-unification

Definition 2.3.1 *Unificator*. A substitution θ is called a unificator for a set of terms $\{t_1, \ldots, t_k\}$, if $|\{t_1\theta, \ldots, t_k\theta\}| = 1$. $t_I = t_i\theta$ is a *common instance* of $\{t_1, \ldots, t_k\}$. $\square$

Example 2.3.2 The set of terms

$$\{p(X, f(X), Y), p(g(Z), W, W)\} \tag{2.28}$$

has this unificator:

$$\theta = \{X/g(Z), W/f(g(Z)), Y/f(g(Z))\} \tag{2.29}$$

since

$$\{p(X, f(X), Y)\theta, p(g(Z), W, W)\theta\} = \{p(g(Z), f(g(Z)), f(g(Z)))\} \tag{2.30}$$

□

Definition 2.3.3 *Anti-unificator.* A set of substitutions $\{\theta_1, \dots, \theta_k\}$ is called an anti-unificator for a set of terms $\{t_1, \dots, t_k\}$, if $\exists t_G \forall i : t_G\theta_i = t_i$. The term t_G is called a *common generalization* of $\{t_1, \dots, t_k\}$. □

Example 2.3.4 The set of terms

$$\{p(f(a, g(Y)), X, g(Y)), p(h(a, g(X)), X, g(X))\} \tag{2.31}$$

has the anti-unificator $\{\theta_1, \theta_2\}$, where

$$\theta_1 = \{Z_1/f(a, g(Y)), Z_2/Y\} \tag{2.32}$$
$$\theta_2 = \{Z_1/h(a, g(X)), Z_2/X\} \tag{2.33}$$

and the common generalization

$$p(Z_1, X, g(Z_2)) \tag{2.34}$$

since

$$p(Z_1, X, g(Z_2))\theta_1 = p(f(a, g(Y)), X, g(Y)) \tag{2.35}$$
$$p(Z_1, X, g(Z_2))\theta_2 = p(h(a, g(X)), X, g(X)) \tag{2.36}$$

□

Example 2.3.5 We can view unification and anti-unification with the diagrams in figure 2.2. □

$$\begin{tikzcd}
t_1 \arrow[dr, "\theta"'] & t_2 \arrow[d, "\theta"] & \dots & t_k \arrow[dll, "\theta"] \\
 & t_I & &
\end{tikzcd}
\qquad
\begin{tikzcd}
t_1 & t_2 & \dots & t_k \\
 & t_G \arrow[ul, "\theta_1"] \arrow[u, "\theta_2"'] \arrow[urr, "\theta_k"'] & &
\end{tikzcd}$$

Figure 2.2: Unification and anti-unification.

Example 2.3.6 A unificator does not always exist. The terms

$$\{f, g\} \tag{2.37}$$

cannot have any substitution as unificator, since the terms are different, and contain no variables.

On the other hand an anti-unification always exists. We may e.g. always construct the trivial anti-unification for the terms $\{t_1, \ldots, t_k\}$ as:

$$\theta_i = \{Z/t_i\}, \;\; 1 \leq i \leq k \tag{2.38}$$

and will thus have this common generalization:

$$Z \tag{2.39}$$

□

Definition 2.3.7 *Most general unificator (MGU).* A unificator θ, for a set of terms, $\{t_1, \ldots, t_k\}$, is called a most general unificator, when given any other unificator, η, there exists a substitution ρ such that $\eta = \rho\theta$. We call $t_I = \theta t_i$ the *least common instance (LCI).* □

Example 2.3.8 The common instance in 2.30 is a MGU for the terms given. See example 3.1.4. □

Lemma 2.3.9 A MGU exists if and only if a unificator exists.

Proof: See corollar 3.1.3. □

Definition 2.3.10 *Most general anti-unificator (MGA).* An anti-unificator $\{\theta_1, \ldots, \theta_k\}$, for a set of terms, $\{t_1, \ldots, t_k\}$, is called a most general anti-unificator, when given any other anti-unificator, $\{\eta_1, \ldots, \eta_k\}$, there exists a substitution ρ such that $\eta_i = \theta_i \rho, \ \forall i$. We call t_G, such that $t_G \theta_i = t_i$, the *greatest common generalization (GCG).* □

Example 2.3.11 The common generalization in 2.34 is a GCG for the terms given, since it is a result of applying the algorithm in section 4.1. □

Example 2.3.12 Neither MGA and MGU are unique. In example 2.3.2 we could have chosen this unificator:

$$\theta' = \{X/g(V), W/f(g(V)), Y/f(g(V)), Z/V\} \tag{2.40}$$

giving a LCI of

$$p(g(V), f(g(V)), f(g(V))) \tag{2.41}$$

In example 2.3.4 we could choose

$$\theta'_1 = \{Z_1/f(a, g(Y)), Z_2/Y, Z3/X\} \tag{2.42}$$
$$\theta'_2 = \{Z_1/h(a, g(X)), Z_2/X, Z3/X\} \tag{2.43}$$

giving a GCG of

$$p(Z_1, Z_3, g(Z_2)) \tag{2.44}$$

It looks like LCI's and GCG's are unique up to renaming of variables, and indeed we shall prove this in lemma 2.5.3 and 2.5.4. □

2.4 The category Term

Definition 2.4.1 *The category* **Term**. This category is defined with objects being terms (definition 2.1.5), and morphisms between these objects being substitutions (definition 2.2.1), such that if θ is the morphism from t_1 to t_2, then θ must act as the identity on variables not in $var(t_1)$, i.e.:

$$X \notin var(t_1) \quad \Rightarrow \quad X\theta = X \tag{2.45}$$

We see that if there is a morphism between two objects, then it is uniquely determined due to the demand in (2.45). Thus from one object to another there is exactly zero or one morphism. □

We need to recall two definitions from category theory (see [AM75, pages 10, 13 and 29]):

Definition 2.4.2 In a category a *product* of objects $O_1, \ldots, O_n$ is an object O equipped with n morphisms (called *projections*)

$$\pi_1 : \ O \to O_1, \ldots, \pi_n : \ O \to O_n \tag{2.46}$$

such that given any other object P with morpishms

$$p_1 : \ P \to O_1, \ldots, p_n : \ P \to O_n \tag{2.47}$$

there exists a unique morphism p, such that

$$p_i = \pi_i p, \ 1 \leq i \leq n \tag{2.48}$$

Similarly a *coproduct* of objects $O_1, \ldots, O_n$ is an object O equipped with n morphisms (called *injections*)

$$\iota_1 : \ O_1 \to O, \ldots, \iota_n : \ O_n \to O \tag{2.49}$$

such that given any other object P with morpishms

$$q_1 : \ O_1 \to P, \ldots, q_n : \ O_n \to P \tag{2.50}$$

there exists a unique morphism q, such that

$$q_i = q\iota_i, \ 1 \leq i \leq n \tag{2.51}$$

□

$$\begin{tikzcd}
t_1 \arrow[dr, "\theta"] \arrow[ddr, bend right=60, "\eta"'] & \ldots & t_k \arrow[dl, "\theta"'] \arrow[ddl, bend left=60, "\eta"] \\
& t_I \arrow[d, "\rho"] & \\
& t'_I &
\end{tikzcd}
\qquad
\begin{tikzcd}
t_1 & \ldots & t_k \\
& t_G \arrow[ul, "\theta_1"] \arrow[ur, "\theta_k"'] & \\
& t'_G \arrow[u, "\rho"'] \arrow[uul, bend left=60, "\eta_1"] \arrow[uur, bend right=60, "\eta_k"'] &
\end{tikzcd}$$

Figure 2.3: MGU and MGA.

Example 2.4.3 We can profit from looking at the diagrams in figure 2.3. This diagram shows us that the MGU is a *coproduct* of terms and the MGA is a *product* of terms in the category **Term** (we note that ρ is unique – see the argument following (2.45)), where the projections in the product are θ_i, and the injections in the coproduct are all equal to θ. Thus MGU and MGA are *dual* concepts. $\square$

Example 2.4.4 In [AM75, pages 10-14] we see that in the category **Set** product and coproduct of sets always exist – the product is the usual cartesian product and the coproduct is the disjoint union. In the category **Term** the coproduct of terms does not always exist, whereas the product of terms always exists (see example 2.3.6). $\square$

2.5 A poset

Example 2.5.1 The set of terms can be equipped with the *quasi-ordering* $\trianglelefteq$, defined as[1]

$$t_1 \trianglelefteq t_2 \quad \Leftrightarrow \quad \exists \theta : \; t_1\theta = t_2 \tag{2.52}$$

[1]See section 2.7 on page 21 for a discussion of this definition.

When $t_1 \trianglelefteq t_2$ we also say that t_1 *matches* t_2. $\trianglelefteq$ is a quasi-ordering since

$$t \trianglelefteq t \tag{2.53}$$

$$t_1 \trianglelefteq t_2 \wedge t_2 \trianglelefteq t_3 \Rightarrow t_1 \trianglelefteq t_3 \tag{2.54}$$

To prove reflexivity (2.53) we only need to observe that the empty substitution $\epsilon = \{\}$ has the effect that $t\epsilon = t$. For transitivity (2.54) we will use that if $t_1\theta_1 = t_2 \wedge t_2\theta_2 = t_3$, then $t_1\theta_1\theta_2 = t_3$. □

$\trianglelefteq$ is not a partial ordering since that would also require anti-symmetri, $t_1 \trianglelefteq t_2 \wedge t_2 \trianglelefteq t_1 \Rightarrow t_1 = t_2$, which is not the case. We can however define an equivalence relation $\sim$:

$$t_1 \sim t_2 \quad \Leftrightarrow \quad t_1 \trianglelefteq t_2 \wedge t_2 \trianglelefteq t_1 \tag{2.55}$$

$\sim$ is clearly reflexive and symmetric, and transitivity is obtained directly from the transitivity of $\trianglelefteq$.

Example 2.5.2 A few examples of equivalences and non-equivalences:

$$\begin{aligned} Z &\sim X &(2.56)\\ f(A,B) &\sim f(C,A) &(2.57)\\ f(A,B) &\not\sim f(Z,Z) &(2.58)\\ f(A) &\not\sim f(g) &(2.59) \end{aligned}$$

□

The ordering $\trianglelefteq$ can be extended in a natural way to the equivalence classes just defined:

$$\tilde{t}_1 \,\tilde{\trianglelefteq}\, \tilde{t}_2 \quad \Leftrightarrow \quad \exists t_1 \in \tilde{t}_1,\ \exists t_2 \in \tilde{t}_2 \colon t_1 \trianglelefteq t_2 \tag{2.60}$$

$\tilde{\trianglelefteq}$ is a partial ordering: it inherits reflexivity and transitivity from the quasi-ordering above, and anti-symmetry is obtained directly from (2.60).

So these equivalence classes forms a *poset*, a partially ordered set.

We can still talk about unification and anti-unification between these equivalence classes, except that now the θ-substitutions are no longer unique – they depend on the actual representative choosen. We have these lemmas:

Lemma 2.5.3 a) If t_G and t'_G are both GCG's for the same set of terms, then $t_G \sim t'_G$.
b) On the other hand, if t_G is a GCG, and $t \sim t_G$ then t is also a GCG for the same set of terms.

Proof: a) We have that

$$\exists \rho_1 \colon t_G = t'_G \rho_1 \quad \Rightarrow \quad t'_G \trianglelefteq t_G \tag{2.61}$$

$$\exists \rho_2 \colon t'_G = t_G \rho_2 \quad \Rightarrow \quad t_G \trianglelefteq t'_G \tag{2.62}$$

which means that $t_G \sim t'_G$.
b) Let t'_G be any common generalization. Since t_G is a GCG, then

$$\exists \rho \colon t_G = t'_G \rho \tag{2.63}$$

and since $t \sim t_G$ then

$$\exists \theta \colon t = t_G \theta \tag{2.64}$$

but then

$$t = t_G \theta = t'_G \rho \theta \tag{2.65}$$

which is what is required for t to be a GCG. □

Lemma 2.5.4 a) If t_I and t'_I are both LCI's for the same set of terms, then $t_I \sim t'_I$.
b) On the other hand, if t_I is an LCI, and $t \sim t_I$ then t is also an LCI for the same set of terms.

Proof: a) We have that

$$\exists \rho_1 \colon \rho_1 t_I = t'_I \quad \Rightarrow \quad t_I \trianglelefteq t'_I \tag{2.66}$$

$$\exists \rho_2 \colon \rho_2 t'_I = t_I \quad \Rightarrow \quad t'_I \trianglelefteq t_I \tag{2.67}$$

which means that $t_I \sim t'_I$.
b) Let t'_I be any common instance. Since t_I is an LCI, then

$$\exists \rho \colon \rho t_I = t'_I \tag{2.68}$$

and since $t \sim t_I$ then

$$\exists \theta \colon t = \theta t_I \tag{2.69}$$

but then

$$t = \theta t_I = \theta \rho t'_I \tag{2.70}$$

which is what is required for t to be an LCI. □

The obvious similarities between the proofs of lemma 2.5.3 and 2.5.4 suggest that we should have proved the latter just referring to dual concepts.

Example 2.5.5 If $t_1 \sim t_2$ then a substitution θ, such that $t_1\theta = t_2$, can only contain variable/variable substitution pairs, and

$$X_1/Y_1 \in \theta \wedge X_2/Y_2 \in \theta \colon X_1 \neq X_2 \Rightarrow Y_1 \neq Y_2 \tag{2.71}$$

This is another way of saying that θ is a permutation of the variables, and as such θ also has an inverse. So two terms in the same equivalence class have the same function-symbols at the same places in the terms, and

$$t_1 \sim t_2 \Rightarrow |var(t_1)| = |var(t_2)| \tag{2.72}$$

□

2.6 A semi-lattice.

Lemma 2.6.1 The partial ordering $\tilde{\trianglelefteq}$ on the equivalence classes is a *semi-lattice* (or *meet-lattice*), where the g.l.b. ($\sqcap$) is defined as a generalization of the GCG:

$$t_1 \in \tilde{t}_1 \wedge t_2 \in \tilde{t}_2 \colon \tilde{t}_1 \sqcap \tilde{t}_2 = \text{GCG}(t_1, t_2) \tag{2.73}$$

Proof: We know from algorithm 4.1 and section 4.1.3 that a GCG always exists. □

The bottom element in this semi-lattice is the equivalence class consisting of terms of the form X, where X is a variable.

Since the meet-operation ($\sqcap$) of this lattice is anti-unification, we might suspect that a join-operation ($\sqcup$) could be defined as unification, but as example 2.3.6 shows, a unificator does not always exist. In section 2.7 however, we extend the semi-lattice to a lattice.

Lemma 2.6.2 The terms consisting of exactly one variable in the category **Term** are all *initial* objects, since if X is a variable, then

$$\forall t \exists \theta \colon t = X\theta \tag{2.74}$$

Proof: Indeed, we may choose $\theta = \{X/t\}$. $\square$

2.7 A lattice.

If we want to extend the semi-lattice to a lattice, we must "invent" a super-l.u.b., i.e. an element greater than all terms. Such a term conveys no information, and is called the null term: Ω.

The counterpart to Ω is the universal term $\mathcal{A}$, which in our case is the equivalence class $\tilde{X}$, i.e. terms consisting of just one variable.

Introducing Ω we must note that the definition of $\tilde{\trianglelefteq}$ does not hold when Ω is involved, i.e. there is no θ such that $\tilde{t}\theta = \Omega$, even if $\tilde{t} \mathrel{\tilde{\trianglelefteq}} \Omega$.

In a sense the lattice we have defined is upside down, since Ω intuitively should be the least element of the lattice. This is due to the definition of $\trianglelefteq$ in example 2.5.1, which is the one used in [Plo69]. However in [Rey69] the definition is the opposite giving a lattice with Ω at the bottom. In that lattice the terms *least common instance* and *greatest common generalization* make better sense.

2.8 Conclusion

I hope to have shown that the concept of unification leads to a beautiful and coherent mathematical theory. It is nice to be able to present this in terms of category theory, but a bit unsatisfactory, since there is no indication of what it can be used for. In [AM75] it says

Category theory is the mathematician's attempt to lay bare some of the underlying principles common to diverse fields in the mathematical sciences. It has become, as well, an area of pure mathematics in its own right.

What I have done is just a few exercises in pure mathematics, but I would have liked it to have had some element of applied mathematics as well. The next two chapters on algorithms for unification and anti-unification will compensate for that.

Chapter 3

Algorithms for Unification

There exist many unification algorithms, see e.g. [Kni89, page 96].

I will present two unification algorithms from the litterature:

- The “classical” as presented in [Rob65].
- The “linear” as presented in citepatweg:lu and [dC86].

In addition I modify the classical unification algorithm with the technique called “Dynamical Programming”, to obtain an algorithm that in some cases are better than the other two. This is demonstrated in tables 3.1 and 3.2, where I compare runtimes of the algorithms.

In all algorithms I assume for simplicity that only two terms are to be unified.

3.1 Robinsons algorithm

The algorithm presented below is closer to a real implementation than the one in [Rob65, page 32], and the one in [Fra86]. Neither of the two can be implemented directly, so I have chosen to formulate it as a recursive algorithm, where the fundamental structure can be used in an implementation.

Algorithm 3.1.1 *Robinsons unification algorithm.*
Input: Two trees, t_1 and t_2 to be unified.
Output: A most general unificator θ, or a message that no unificator exists.

Method:
Step 1: Set $\theta' = \{\}$. Instead of changing the input trees whenever a substitution is made, we must look at the trees through θ' using this function:

$$\alpha(\theta', t) = \begin{cases} f(\alpha(\theta', t_1), \ldots, \alpha(\theta', t_n) & \text{if } t = f(t_1, \ldots, t_n) \\ \alpha(\theta', t') & \text{if } t = v \text{ and } v/t' \in \theta' \\ v & \text{if } t = v \text{ and } v/t' \notin \theta' \end{cases} \tag{3.1}$$

This function will also be used to produce the final θ (if this is needed).
Step 2: Traverse t_1 and t_2 simultaneously. Assume the nodes considered are u in t_1 and v in t_2.
Step 2a: If $u = v$, then simultaneously traverse the sons (if any) of u and v.
Step 2b: If $u \neq v$, then check:

a. If both nodes are function-symbols output *"No unificator"* and stop.
b_1. If u is a variable: if u occurs in v, output *"No unificator"* and stop. If u does not occur in v set $\theta' = \theta' \cup \{u/v\}$.
b_2. If v is a variable: if v occurs in u, output *"No unificator"* and stop. If v does not occur in u set $\theta' = \theta' \cup \{v/u\}$.

Step 3: If a final θ is needed it is produced by unioning all $v/\alpha(\theta', t)$ for all $v/t \in \theta'$.

θ will now contain the unificator – unless, of course, it has stopped with the message *"No unificator"*. □

Lemma 3.1.2 Algorithm 3.1 will stop, and if the two terms have a unificator, θ will be an MGU.

Proof: The proof for the "unification algorithm" in [Fra86] can be used directly. □

Corollar 3.1.3 When two terms has a unificator they will also have a most general unificator. □

Example 3.1.4 Let's run the algorithm on the two terms in example 2.3.2:

$$\{p(X, f(X), Y), p(g(Z), W, W)\} \tag{3.2}$$

The roots of the two trees are identical, so we look at the first two sons. One is a variable, X, the other, $g(Z)$, is a term without X, so we start setting $\theta = \{X/g(Z)\}$. We have the same situation for the second sons, so we extend θ to $\{X/g(Z), W/f(X)\}$. The third sons are both variables – it doesn't matter which variable we choose to substitute for the other, so let's extend θ to $\{X/g(Z), W/f(X), Y/W\}$. However we forgot to use α, so we should have written:

$$\{X/g(Z), W/f(g(Z)), Y/f(g(Z))\} \tag{3.3}$$

which is identical to (2.29). □

Example 3.1.5 It is clear that by using α will ensure that θ is an idempotent substitution. □

3.1.1 In more detail

The algorithm can be specified in more detail. The function OccursIn(v,t) is a simple recursive function that returns true if variable v is found in term t.

```
proc traverse($t_1$,$t_2$)
$t'_1 := \alpha(\theta', t_1)$
$t'_2 := \alpha(\theta', t_2)$
if $t'_1 \neq t'_2$ then
   if IsVar($t'_1$) then
      if OccursIn($t'_1, t'_2$) then
         stop "No unificator"
      $\theta' := \theta' \cup \{t'_1/t'_2\}$
   elsif IsVar($t'_2$) then
      if OccursIn($t'_2, t'_1$) then
         stop "No unificator"
      $\theta' := \theta' \cup \{t'_2/t'_1\}$
```

```
        else
            stop "No unificator"
        endif
    else
        traverse($t'_1$.son[$i$],$t'_2$.son[$i$]), $1 \leq i \leq t'_1$.#sons
    endif

$\theta' := \{\}$
traverse($t_1$,$t_2$)
$\theta := \{\}$
$\theta := \theta \cup \{v/\alpha(\theta', t)\},\ \forall v/t \in \theta'$
```

In an actual implementation we will use α only when necessary, and not just uncritical at the start.

3.1.2 Occurcheck and timecomplexity

Example 3.1.6 Robinsons algorithm has exponential runtime on some terms. With the terms

$$\begin{array}{l} p(f(X_1, X_1), f(X_2, X_2), \ldots, f(X_n, X_n)) \\ p(X_2, \ldots, X_{n+1}) \end{array} \tag{3.4}$$

the algorithm will first set $\theta' = \{X_2/f(X_1, X_1)\}$, and then extend θ' successively with $X_{i+1}/f(X_i, X_i)$. To do this the algorithm must make the *occurcheck* for X_{i+1} in $f(X_i, X_i)$. When this term has X_i substituted, and then X_{i-1}, X_{i-2}, ..., the resulting tree will have $O(2^i)$ nodes. The size of input is $O(n)$, so in this case the algorithm is exponential. $\square$

Example 3.1.7 Some implementations of logic programming avoid this problem simply by omitting the occurcheck. This has a strange effect on e.g. these terms:

$$\{f(X), X\} \tag{3.5}$$

Normally this will result in the message "No unification" from step 2b.b_2 in the algorithm above, but if we omit the occurcheck, we will get the unificator

$\theta' = \{X/f(X)\}$. Using α will give an infinite substitution:

$$\theta = \{X/f(f(f(\ldots)))\} \tag{3.6}$$

which, in a sense, is a correct answer, since

$$f(X)\theta = f(f(f(\ldots))) = X\theta \tag{3.7}$$

□

Omitting the occurcheck will also make normal processing faster. [Col82] mentions that concatenation of two lists can be done in linear time without occurcheck, but requires $O(n^2)$ time with occurcheck.

The occurcheck itself it not the only source to a potential exponential runtime for Robinsons algorithm. As we shall see in example 3.2.1 and in table 3.2, another set of terms can be constructed that will exhibit the exponential runtime, even without the occurcheck.

Whether it is a reasonable choice to omit the occurcheck in an implementation is beyond the scope of this work. The reader might want to consult [Cou83] for a formal discussion of infinite trees.

Instead of omitting the occurcheck every time, it is possible to analyze a logic program (e.g. in PROLOG) in advance to avoid almost all unnecessary occurchecks, as is stated in [Bee88]. The method employed might be useful when compiling a PROLOG program to an abstract PROLOG machine.

3.2 An algorithm using "Dynamical Programming"

If we look at Robinsons algorithm, and example 3.1.6, we see that the term $f(X_i, X_i)$ will be checked for variables X_{i+1}, X_{i+2},. . .,X_{n+1}. Thus we are doing almost the same work several times. This suggests that we look at the programming principle *Dynamical Programming*.

The first time we look for a variable in $f(X_i, X_i)$ we might as well collect the names of all variables present, and associate this set of variables with the node representing $f(X_i, X_i)$. Next time this term is considered we only have to look at this set of variables.

A set of variables is not static. Whenever a new substitution, X/t, is made, all sets of variables V with X present must be updated: $V = V \backslash \{X\} \cup var(t)$. To perform this update we have to recursively traverse the two terms to be unified. Whenever we meet a node with a set of variables without X, we need not go further down the tree.

3.2.1 In more detail

We need to change two things in Robinsons algorithm: The OccursIn-function, and the updating of θ'. We also extend each function-node with a boolean: Calculated, stating whether we have already calculated the variables of this subtree, and with a set of variables: Variables, containing the variables of this subtree (if calculated).

```
**func** OccursIn($v$,$t$)
**if** IsVar($t$) **then**
    **return** $v = t$
**else**
    CalculateVariables($t$)
    **return** $v \in t$.Variables
**endif**
```

where

```
**proc** CalculateVariables($t$)
**if** IsFunc($t$) $\wedge \neg t$.Calculated **then**
    $t$.Variables := {}
    $1 \leq i \leq t$.#sons:
        **if** IsVar($t$.son[$i$]) **then**
            $t$.Variables := $t$.Variables $\cup$ $t$.son[$i$]
        **else**
```

```
            CalculateVariables($t$.son[$i$])
            $t$.Variables := $t$.Variables $\cup$ $t$.son[$i$].Variables
        **endif**
    **endif**
```

Whenever we add v/t to θ' we must perform UpdateVariables(t_i,v,$var(t)$), $1 \leq i \leq 2$, where

```
    **proc** UpdateVariables($t$,$v$,$s$)
    **if** IsFunc($t$) $\wedge$ $t$.Calculated $\wedge$ $v \in t$.Variables **then**
        $t$.Variables := $t$.Variables $\backslash v \cup s$
        UpdateVariablesInTree($t$.Sons[$i$],$v$,$s$), $1 \leq i \leq t$.#sons
    **endif**
```

3.2.2 Timecomplexity

Example 3.2.1 As we can see from table 3.1 this algorithm is much faster than Robinsons algorithm on the terms from example 3.1.6. It is even faster than the one presented in section 3.3! Unfortunately it still has exponential growth on this set of terms, since it needs to compare exponentially larger, but identical, trees:

$$\begin{aligned} & a(p(f(X_1,X_1),f(X_2,X_2),\ldots,f(X_n,X_n)), \\ & \quad q(f(X_1,X_1),f(X_2,X_2),\ldots,f(X_n,X_n))) \qquad (3.8) \\ & a(p(X_2,\ldots,X_{n+1}),q(X_2,\ldots,X_{n+1})) \qquad (3.9) \end{aligned}$$

As long as we are trying to unify the p-terms, nothing is changed. When we get to the q-terms we have made substitutions for variables $X_2,\ldots,X_{n+1}$, so the algorithm proceeds to compare larger and larger subtrees, $f(X_1,X_1)$, $f(f(X_1,X_1),f(X_1,X_1))$, etc., all of which are identical. The last tree will have $O(2^n)$ nodes, and thus the algorithm will run in exponential time. The same argument holds for Robinsons algorithm without occurcheck, and table 3.2 illustrates this fact beautifully. □

We observe again that the reason for exponential runtime is redundant work, since we "know" that the trees mentioned above are identical. An algorithm with less than exponential runtime needs to know more about common structures in the terms to be unified. The algorithm in the next section is a solution to that.

3.3 The linear algorithm by Paterson and Wegner

This algorithm was presented in [PW78], and later clarified in [dC86], and solves the problem of exponential runtime. This is more complicated, and slower for “small” terms, than Robinsons algorithm.

The first requirement for the algorithm to work is to have the input represented, not as trees, but as reduced DAGs, where common subexpressions are represented as a single subgraph. In practice this requirement can be relaxed, so that only terminals need to be represented uniquely. However, the more structure-sharing available, the faster the algorithm will run.

In our case, where we only consider two terms to be unified, we will thus have a DAG with two roots (a root being a node with indegree 0).

3.3.1 Equivalence classes

We define an equivalence relation between the nodes of our DAG, which we can view as a reduced graph, with the following properties:

1. Whenever two function nodes are equivalent, their sons (in proper pairs) are equivalent.
2. In an equivalence class, there is at most one function symbol (or in our case: function symbol and arity).
3. The equivalence classes has a partial order, inherited from the partial order of the DAG. This insures that the reduced graph is acyclic.

An equivalence relation with this property is called *valid.* [PW78] proves this lemma:

Lemma 3.3.1 Nodes u and v are unifiable if and only if there is a valid equivalence relation with $u \equiv v$. In the affirmative case there is a unique minimal valid equivalence relation.

Proof: See [PW78, p. 161–162]. □

This suggests that we shall try to construct equivalence classes to see if the two roots are in the same equivalence class, in which case they are unifiable – and hopefully constructing a unificator at the same time. A simple algorithm for doing this is:

Algorithm 3.3.2 Test a pair of nodes u and v for unifiability.
Step 1. Set $u \equiv v$.
Step 2. As long as we have a pair of nodes r and s with $r \equiv s$, without knowing the same about a pair of corresponding sons, r' and s', set $r' \equiv s'$.
Step 3. If the relation $\equiv$ is valid according to 2 and 3 above, then u and v are unifiable, otherwise they are not. □

This algorithm has one (small) problem in step 2, since the best equivalence handling routine, *union-find*, uses time $O(n\alpha(n))$, where α is the functional inverse of Ackerman's function.

This is very close (as close as you can get) to a linear algorithm, since α is a function with extremely slow growth. Even if n is the number of elementary particles in the universe, $\alpha(n) \leq 5$. This algorithm was presented by Huet in 1976 as an *almost linear* algorithm.

The main idea in Paterson & Wegmans algorithm is to process the equivalence classes in a special order, finishing so-called *root classes* first.

Definition 3.3.3 *Root class.* An equivalence class consisting of roots (nodes with indegree 0) is called a *root class.* □

The first root class to be considered is the one between the two roots in the DAG. When this is finished its nodes are deleted, and new roots and root classes emerge. The core of the algorithm is to keep track of this. When two terms have a unifier, there is a root class, due to:

Lemma 3.3.4 ([PW78, page 161]) Any nonempty equivalence relation satisfying criteria 3 in the definition of validity has a root class.

Proof: An equivalence class, which is maximal with respect to the partial ordering must be a root class. □

If at some point in the algorithm we have nodes left, and no root classes, we can conclude that the terms are not unifiable. At this point we can see how an implicit occurcheck is made: If we were to unify a variable, say X, with an expression involving this variable, say $f(X)$, the algorithm would produce a cycle in the reduced graph. No nodes in this cycle could ever become a root, and thus a member of a root class.

All we need now is an efficient algorithm for manipulating root classes, and this is given in the next section, where the root classes are represented as stacks. Nodes are not explicitly deleted, but only marked so.

At one point [dC86, page 85] disagrees with [PW78]. [dC86] states that when the algorithm *creates links between corresponding sons in s and r*, it may create more than one link between two nodes, and argues:

> *It will take more than linear time to prevent a double link between x and y (unless we unrealistically assume available a square matrix M of booleans, initialized with false values, where $M(i, j)$ indicates whether there is an edge between node i and j).*

A small change in the algorithm is then suggested, that will permit multiple links between a pair of nodes. However, as we shall see in section 4.2.1, it is quite realistic to manipulate a square matrix in linear time (but using quadratic space), and it is quite possible that [PW78] have thought about this, although they do not mention it explicitly. In the next section I have adopted the "small change", as this is the easiest to use.

3.3.2 The algorithm in detail

Assume each node has been assigned a set representing it's fathers, i.e.: s.Father. This can be done by a simple traversal of the DAG (a node may have more than one father in a DAG).

```
**proc** Finish($r$)
**var**
    $S$: Stack
**if** $\neg r$.Finished **then**
    **if** $r$.Pointer $\neq \perp$ **then**
        **stop** "No unificator"
    **else**
        $r$.Pointer := $r$
        Init($S$)
        Push($S, r$)
        **while** $|S| > 0$ **do**
            $s$ := Pop($S$)
            **if** IsFunc($s$) $\wedge$ IsFunc($r$) $\wedge$ $r$.Symbol $\neq$ $s$.Symbol **then**
                **stop** "No unificatior"
            **endif**
            $\forall t \in s$.Father:
                Finish($t$)
            $\forall t \in s$.Link:
                **if** $t$.Finished $\vee$ $t = r$ **then**
                    **skip**
                **elsif** $t$.Pointer $= \perp$ **then**
                    $t$.Pointer := $r$
                    Push($S, t$)
                **elsif** $t$.Pointer $\neq r$ **then**                              NB[1]
                    **stop** "No unificator"
                **else**
                    **skip**
                **endif**
            **if** $s \neq r$ **then**
```

[1]The "small change" mentioned: this ensures that we may have multiple links between nodes.

```
            **if** IsVar($s$) **then**
                $\sigma = \sigma\{s/r\}$
            **elsif** IsFunc($s$) **then**
                *create links between corresponding sons in $s$ and $r$*
            **endif**
            $s$.Finished := true                                NB[2]
        **endif**
    **endwhile**
    $r$.Finished := true
  **endif**
**endif**

*Create link between the two roots of the trees*
$\sigma = \epsilon$
$\forall f \in$ FunctionNodes:
    Finish($f$)
$\forall v \in$ VariableNodes:
    Finish($v$)
```

The substitution σ made by this algorithm is an ordered substitution, and can – if needed – be converted to an unordered ("normal") substitution by the algorithm in [dC86, pages 86–88]. The details are omitted from this report.

3.4 Implementations

I have implemented the 3 algorithms presented above, and I will present a few runs on two different kinds of test-terms: Those from example 3.1.6 (T_n), and those from example 3.2.1 (S_n):

$$\begin{aligned} T_n &= \{p(f(X_1,X_1),\ldots,f(X_n,X_n)),p(X_2,\ldots,X_{n+1})\} && (3.10)\\ S_n &= \{a(p(f(X_1,X_1),\ldots,f(X_n,X_n)), \\ &\qquad q(f(X_1,X_1),\ldots,f(X_n,X_n))), && (3.11)\\ &\quad a(p(X_2,\ldots,X_{n+1}),q(X_2,\ldots,X_{n+1})\} \end{aligned}$$

[2]In [dC86, page 90] this statement is misplaced.

For Robinsons algorithm I have also tested a version without occurcheck.

My implementations are not very sofisticated, and probably not very efficient either. But they can be used to illustrate the time-complexity of the 3 algorithms, and to estimate when Paterson & Wegners linear algorithm should be used instead of Robinsons.

In Robinsons algorithm and in the dynamical algorithm I have only calculated θ', as the construction of θ itself would always have resulted in exponential runtime, since it is exponential in size.

The linear algorithm does not use a reduced DAG – only terminals are represented uniqely.

The implementations have been made in TurboPascal 5.0 on an IBM-PC-clone (running 12 MHz) – the sources are not included in this report.

	Robinson w/occurcheck	Robinson wo/occurcheck	Dynamical	Linear
T_1	0.15	0.06	0.00	0.40
T_2	0.11	0.00	0.05	0.44
T_3	0.27	0.05	0.15	0.55
T_4	0.27	0.00	0.22	0.49
T_5	0.66	0.06	0.22	0.88
T_6	1.38	0.12	0.33	1.01
T_7	2.77	0.22	0.33	1.15
T_8	5.38	0.11	0.33	1.32
T_9	10.80	0.10	0.43	1.76
T_{10}	21.63	0.00	0.55	1.63
T_{11}	43.04	0.11	0.30	1.73
T_{12}	86.23	0.00	0.58	1.92
T_{13}	172.42	0.10	0.55	2.14
T_{14}	344.89	0.17	0.71	2.25
T_{15}	689.48	0.11	0.76	4.28

Table 3.1: Runtime in seconds for 100 unifications.

	Robinson w/occurcheck	Robinson wo/occurcheck	Dynamical	Linear
S_1	0.11	0.16	0.17	0.66
S_2	0.17	0.16	0.17	0.97
S_3	0.39	0.22	0.50	1.28
S_4	1.10	0.00	0.78	1.33
S_5	1.63	1.08	1.32	1.94
S_6	3.41	2.27	2.42	2.19
S_7	6.82	2.80	4.33	2.68
S_8	13.75	8.39	8.68	2.85
S_9	27.36	16.82	16.87	3.18
S_{10}	54.77	33.62	33.51	3.40
S_{11}	109.85	67.27	66.56	3.83
S_{12}	219.82	134.34	132.77	4.10
S_{13}	439.33	268.68	265.13	4.22
S_{14}	878.57	537.29	529.79	4.74
S_{15}	1757.31	1074.50	1059.36	5.11

Table 3.2: Runtime in seconds for 100 unifications.

In table 3.1 Robinsons algorithm is indeed exponential: Whenever the input size grows with 4 nodes (from T_i to T_{i+1}) the time doubles. The linear algorithm looks linear, and the dynamical algorithm seems to be much better.

However in table 3.2 the dynamical algorithm grows exponential, as expected. We can also see that it is not only the occurcheck that makes Robinsons algorithm exponential on some terms.

In table 3.1 the *breakeven-point* for Robinsons algorithm and the linear algorithm seems to be around T_5, and similarly in table 3.2 around S_5. If a reduced DAG was used, the linear algorithm would perform better in table 3.2.

Considering that the terms T_5 and S_5 are not typical for logic programming, the linear algorithm does not look like a good candidate for a basic unification algorithm. The dynamical algorithm however seems to be slightly better than

Robinsons, so it might be considered, even if it is exponential in the worst case.

It is difficult to get an accurate measurement of small timeslices on an IBM-PC, and this accounts for the small fluctuations in the tables (e.g. the entries indicating 0.00 seconds are obviously wrong).

Chapter 4

Algorithms for Anti-Unification

The concept of anti-unification and (almost identical) algorithms for anti-unification were presented by Plotkin [Plo69] and Reynolds [Rey69] in 1969. I will present Reynolds algorithm (and mention how Plotkins differ) – this algorithm uses quadratic time. Two new linear algorithms, invented together with my supervisor Gudmund Frandsen, are presented (the first, however, using quadratic space).

Finally I present a parallel algorithm for anti-unification proposed by Kuper et al. [KMPP88].

Again, for simplicity, we only want to anti-unify two terms.

4.1 Anti-unification algorithm (Reynolds)

Algorithm 4.1.1 *Anti-unification algorithm.*
Input: Two terms represented as trees t_1 and t_2.
Output: Two sets of substitutions θ_1 and θ_2.
Method: Set θ_1 and θ_2 empty. Simultaneously traverse t_1 and t_2 in preorder. Whenever different subtrees, s_1 and s_2, are encountered, we look in θ_1 and θ_2 to see if we have already made a substitution with these subtrees. If not, we invent a new variable, say Z, and add Z/s_1 to θ_1 and Z/s_2 to θ_2. □

In addition to updating θ_1 and θ_2 we could produce the tree t_G from figure 2.3 while traversing t_1 and t_2.

Plotkins algorithm differs from Reynolds in that whenever we make a new substitution, we not only add it to θ_1 and θ_2, but also replace all occurrencies of s_1 and s_2 with Z, whenever s_1 and s_2 occur at "the same place" in t_1 and t_2. It seems like an unpractical way to do it, but is obviously the same as the method in Reynolds.

4.1.1 In more detail

We can write the algorithm in more detail, using a pseudo-language. This algorithm is extended to produce the t_G as well.

```
func traverse($s_1$,$s_2$)
if $s_1 \neq s_2$ then
   if $(Z_1/s_1) \in \theta_1$ & $(Z_2/s_2) \in \theta_2$ & $Z_1 = Z_2$ then
      return $Z_1$
   else
      $Z$ := newvariable()
      $\theta_i := \theta_i \cup (Z/s_i)$, $1 \leq i \leq 2$
      return $Z$
   endif
else
   $t$ := copynode($s_1$)
   $t$.son[$i$] := traverse($s_1$.son[$i$],$s_2$.son[$i$]), $1 \leq i \leq s_1$.#sons
   return $t$
endif

$\theta_i$ := {}, $1 \leq i \leq 2$
$t_G$ := traverse($t_1$,$t_2$)
```

4.1.2 Timecomplexity

The time to traverse the trees is $O(n)$, where n is the number of nodes in the trees. At each node we may have to search θ_1 and θ_2, the size of each cannot exceed n. This search involves comparing subtrees, each with a maximum of n nodes, but the total number of nodes in the trees stored in θ_1 and θ_2 cannot exceed n, so with a simple linear search we have a total time of $O(n^2)$. We will later show an algorithm with timecomplexity $O(n)$.

4.1.3 Termination and correctness

It is clear that the algorithm will eventually terminate, since it is just a (possibly shortened) preorder traversal of a tree.

It is also clear that $t_G\theta_1 = t_1$ and $t_G\theta_2 = t_2$, since every difference in t_1 and t_2 is "ironed" out with a suitable substitution in θ_1 and θ_2.

From (2.52) we see that

$$t_G \trianglelefteq t_1 \ \wedge \ t_G \trianglelefteq t_2 \tag{4.1}$$

Suppose

$$t'_G \trianglelefteq t_1 \ \wedge \ t'_G \trianglelefteq t_2 \tag{4.2}$$

We need to show that $t'_G \trianglelefteq t_G$. Suppose this is *not* the case. Then there must be two different, but corresponding[1] nodes in t'_G and t_G, say u' and u, where u' is a function-node. The nodes in t_1 and t_2 corresponding with u' must be function-nodes with the same function-symbol, due to (4.2). But in that case the algorithm would have left u unchanged in t_G, which is a contradiction.

4.1.4 Comment

If we skip the search in θ_1 and θ_2, we will still get a generalization, but not necessarily a least generalization, and will only use $O(n)$ time.

[1]See definition 4.4.1.

4.2 Linear algorithm 1

Looking at Reynolds algorithm for anti-unification in section 4.1, we see that it might be useful to have better control over common subexpressions. This problem is the same encountered in Robinsons algorithm for unification, and Paterson & Wegman solves this using a reduced DAG, see section 3.3. We propose to do the same.

Assume the input is represented as reduced DAGs, d_1 and d_2, for the two terms in question. We shall investigate how a reduced DAG can be build in section 4.2.3.

Each node in d_i is assigned a unique number in the range $1 \dots |d_i|$. This can be done with a simple traversal of the DAGs in linear time[2].

The algorithm is now similar to Reynolds. We traverse d_1 and d_2 in the same way as we would do, were they trees. Whenever we meet a difference, we need to know if we have made such a substitution already (same as in Reynolds). In our case the two sub-DAGs can be identified by the numbers assigned to their nodes (say n_1 and n_2), so we need an efficient datastructure for pairs of numbers.

4.2.1 "Lazy Initialization"

[AHU74, exercise 2.12, page 71] presented an algorithm for maintaining a subset of numbers in the range $[1, \dots, N]$ in time proportional to the size of the subset, but with space proportional to N. The main trick is to use *lazy initialization* of the range.

We need an array A, with room for all elements in the range $[1, \dots, N]$, and an array B, with room for the maximum number of elements in the subset, say k. We may possibly also want an array C with k elements containing values associated with the numbers in the subset. A contains pointers to B,

[2]Linear in the number of nodes in the original trees

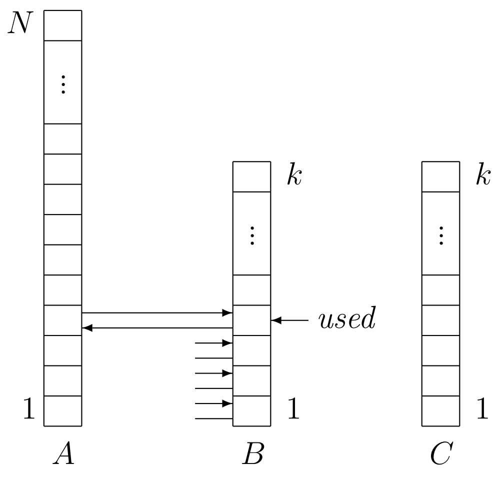

Figure 4.1: Datastructures for "Lazy Initialization".

and B contains pointers to A. See figure 4.1. The variable *used* indicates which part of B (and C) is in use.

Init: Only entries of B in the range $[1, \ldots, \mathit{used}]$ are valid, so the only thing we need to do to initialize the whole datastructure, is to set *used* to zero. We may need to allocate A, B and C.

Member: To see if i is a member of the set, we must look up $b := A[i]$. A was not initialized, so b can be anything. First we check if $1 \leq b \leq \mathit{used}$. If this is not the case, then i is not in the set, otherwise we proceed to check if $B[b] = i$. If this is not the case, then i is not in the set, otherwise it is.

Insert: To insert i into the set (if it is not already there), we must increase the variable *used* by one, and set $A[i] := \mathit{used}$ and $B[\mathit{used}] := i$.

All 3 operations are done in constant time. If we also needed to delete a member of the set, j, we could just set $A[j] := 0$. To reuse the space in B we could build a list of free locations.

We have these operations:

Init(S, N, k)	$O(1)$
Member(S, i)	$O(1)$
Insert(S, i)	$O(1)$

provided $1 \leq i \leq N$ and Insert is called no more than k times.

4.2.2 In more detail

We can now specify the algorithm in more detail. We use "lazy initialization" to represent a set of pairs in the usual way, i.e. (i, j) maps to $(i-1) * \max_1 + j - 1$, with $N = \max_1 * \max_2$. The array C contains a variablename in each entry, and we extend Insert and Member with a parameter for a variablename. Assume $\text{inx}(i, j) := (i-1) * \max_1 + j - 1$

```
    **func** traverseDAG($s_1$,$s_2$)
    **if** $s_1$.name $\neq$ $s_2$.name **then**
       **if** Member($S$,inx($s_1$.num,$s_2$.num),$Z$) **then**
          **return** $Z$
       **else**
          $Z$ := newvariable()
          $\theta_i := \theta_i \cup (Z/s_i)$, $1 \leq i \leq 2$
          Insert($S$,inx($s_1$.num,$s_2$.num),$Z$)
          **return** $Z$
       **endif**
    **else**
       $t$ := copynode($s_1$)
       $t$.son[$i$] := traverse($s_1$.son[$i$],$s_2$.son[$i$]), $1 \leq i \leq s_1$.#sons
       **return** $t$
    **endif**

 $\theta_i := \{\}$, $1 \leq i \leq 2$
 $d_i$ := DAGify($t_i$), $1 \leq i \leq 2$
 max$_i$ := numberDAG($d_i$), $1 \leq i \leq 2$
 Init($S$,max$_1$*max$_2$,minimum(|$t_1$|,|$t_2$|))
 $t_G$ := traverseDAG($d_1$,$d_2$)
```

4.2.3 Reduced DAG

We observe that Paterson & Wegners algorithm doesn't really need a reduced DAG to work. It only needs that terminal nodes are shared, although the algorithm will be faster the more sharing of structure there is.

In our case we cannot restrict the DAG to this simple form, so we must investigate how a reduced DAG may be built from a tree.

Algorithm 4.2.1 *Reduced DAG.* Assume that

- names of variables and functions are represented by a fixed-length bit-string.
- father-pointers have been assigned to all nodes.
- that each node knows its number among its brothers.

Input: A tree.
Output: A reduced DAG.

Method:
Step 1: Assign to each node in the tree the length, l, of the longest path to a terminal node (i.e. the height of the subtree). Collect all nodes with the same l-value.

It is obvious that common subtrees must have the same l-value.

Step 2: Starting with terminal nodes ($l = 0$), use a radix-sort[3] to sort the names. Nodes with identical names are now adjacent in the list, and can be collapsed into one node. Doing this a son must modify its pointer in its father.

Step 3: For each additional level ($l > 0$) we do a similar operation: Make a sorting label for each node by concatenating the name of the node with

[3]Also called bucket-sort or pocket-sort. A beloved child has many names.

the *addresses*[4] of its sons. Sort this list with a radix-sort, and collapse nodes with identical sorting labels as before. When all levels have been processed we are finished. (Step 2 is in fact just a special case of step 3.) □

A radix-sort is linear in the number of items and in the length of the bitstrings sorted. In step 3 we sort strings with length $b + na$, where b is the bitlength of a name, n is the maximum number of sons and a is the bitlength of an address, which makes this a non-linear operation in a standard radix-sort.

A careful implementation of the radix-sort can be linear in the number of bits sorted, e.g. by having the length of the sorting labels as primary sorting-key. In this way the total time spent in algorithm 4.2.1 is linear in the number of bits in the input tree.

4.3 Linear algorithm 2

"Lazy initialization" in linear algorithm 1 was needed to check if we have already made a certain substitution. A different approach is this:

Whenever we make a substitution, we construct the new variablename by concatenating the numbers of the two nodes in the DAG (see section 4.2). This will automatically produce the same names when needed. We will likely make duplicate substitutions, but these can be removed by a radix-sort after the traversal of the DAG.

Then we can rename the new variables, either by giving them their number in the sorted list, or by some other automatic name-generation.

This method is also linear and does not require quadratic space.

[4]or other unique node-numbers.

4.4 A parallel algorithm

In the litterature parallel algorithms are often presented in terms of the *parallel random-access machine (PRAM)*. For the details of this the reader is referred to [CLR90]. I assume in the following that we use a CREW PRAM.

4.4.1 The algorithm by Kuper et al.

Kuper et al. [KMPP88] has proposed a parallel algorithm for anti-unification, which I sketch below. There are a few basic definitions needed:

Definition 4.4.1 An *edgelabel* is an integer associated with an edge, and has value i if the edge connects a father with its i'th son. If u is a node in the tree, *Path(u)* is the sequence of edgelabels from the root to u. Two nodes u and v are *corresponding* if their paths from the roots are identical, i.e. $\mathit{Path}(u)=\mathit{Path}(v)$. □

Algorithm 4.4.2 *Parallel anti-unification.*
Input: Two terms represented as trees, t_1 and t_2.

Output: A term t_G being the anti-unification of t_1 and t_2.

Method: Assume u from t_1 and v from t_2
Step 1: Find the corresponding nodes in t_1 and t_2.

To do this we could compute the *Path*-values in the two trees, and then sort these values. *Path*-values can be $O(n \log n)$ bits long, but since a tree with n nodes has exactly n different paths from the root, we can code each path with numbers from 1 to n (using $\log n$ bits). To do this we proceed as follows (we assume each node has a pointer, Father, to its father in the tree):

Initially w.Code is the edgelabel between w and its father.

a. Let u.Code be the concatenation of the strings u.Father.Code and u.Code.

b. Since u.Code is now $2 \log n$ bits long we sort the numbers, and replace u.Code by its number in the sorted list.

c. Then we use the classical technique of *pointer jumping*[5] and sets w.Father := w.Father.Father, and continue with step a.

This process is repeated until all Father-pointers are nil.

The number of steps in this process is at most $O(\log n)$, since the height of the tree can at most be n. Each step involves sorting a list of n numbers of size $O(\log n)$ bits, and Cole, [Col86], gives an algorithm for doing this in $O(\log n)$ time using n processors. The total time of this process is thus $O(\log^2 n)$ using n processors.

At this point all corresponding nodes have the same Code, due to this lemma:

Lemma 4.4.3 [KMPP88, lemma 14]
Let u and v be nodes from the trees t_1 and t_2, respectively. Upon termination of step 1, u.Code = v.Code if and only if u and v are corresponding nodes.

Proof: This is easily verified by induction – the argument can be found in [KMPP88, page 118] $\square$

Step 2: If u and v are corresponding nodes with the same function- or variablename, we may just output this name.

Step 3: If u and v have different labels we must invent a new variablename, but we must choose the same variablename for all other pairs of nodes with the same subtrees.

Kuper et al. propose to solve this using an injective function $\rho(t_u, t_v)$, where t_u and t_v are the subtrees with roots u and v.

This injective function can be computed like this:

[5]Also called *recursive doubling.*

a. For each corresponding pair of nodes, u_i and v_i, with different labels, a new tree, w_i, is constructed with t_{u_i} as its left son and t_{v_i} as its right son.

b. We number the nodes in w_i from 1 to $|w_i|$ in a deterministic fashion, e.g. in preorder. Let W_i be an array of length $|w_i|$. All nodes will now write their label in the position in W_i corresponding to their number.

c. The W_i's are now string representations of the w_i's, so if we can sort them we can find identical the w_i's that must have the same labels. Since the W_i's are long ($O(n)$), we use a technique similar to the one in step 1, to produce and sort encodings of $\log n$ bits.

d. Identical strings are adjacent in the sorted list, and can be given the same label.

The numbering in preorder is an exercise in [CLR90, page 701, exercise 30.1-6]. Using n processors, steps a, b and d take $O(\log n)$ time, while step c takes $O(\log^2 n)$.

Step 4: In fact we have done more work than needed, since only *maximally different* nodes, i.e. corresponding nodes with different labels, where no ancestors also have this property. So sons of nodes that has been relabelled, must be deleted. $\Box$

This parallel algorithm, which takes $O(\log^2 n)$ time on n processors, can be simulated on a sequential machine in time $O(n \log^2 n)$.

In the parallel case we can reduce the runtime by increasing the number of processors needed. If we only sort the strings in step 1 every i'th iteration, and use $n2^i$ processors, the total time can be reduced to $O((log^2 n)/i)$, provided $1 \leq i \leq \log n$.

Bibliography

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[AM75] Michael A. Arbib and Ernest G. Manes. *Arrows, Structures, and Functors. The Categorical Imperative.* Academic Press, 1975.

[Bee88] Joachim Beer. The occur-check problem revisited. *Journal of Logic Programming*, 5:243–261, 1988.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. Algorithms for parallel computers. In *Introduction to Algorithms*, chapter 30, pages 688–729. The MIT Press, 1990.

[Col82] A. Colmerauer. Prolog and infinite trees. In W. K. Clark and S. A. Tarnlund, editors, *Logic Programming.* Academic Press, New York, 1982.

[Col86] Richard Cole. Parallel merge sort. In *Proc. Twenty-Seventh Annual IEEE Symposium on Foundations of Computer Science*, pages 511–516. IEEE, 1986.

[Cou83] B. Courcelle. Fundamental properties of infinite trees. *Theor. Comput. Sci.*, 25:95–129, 1983.

[dC86] Dennis de Champaux. About the Paterson-Wegman linear unification algorithm. *Journal of Computer and System Sciences*, 32:79–90, 1986.

[Fra86] Gudmund Frandsen. *Notes from the Course: Logic Programming, Autumn 1986.* 1986. In Danish.

[KMPP88] G. M. Kuper, K. W. McAloon, K. V. Palem, and K. J. Perry. Efficient parallel algorithms for anti-unification and relative complement. In *Proc. Third Annual IEEE Symposium on Logic in Computer Science*, pages 112–120. IEEE, 1988.

[Kni89] Kevin Knight. Unification: A multidisciplinary survey. *ACM Computing Surveys*, 21:93–124, 1989.

[LMM86] J-L. Lassez, M. J. Mahler, and K. Marriot. Unification revisited. In M. Boscarel, L. Carlucci Aiello, and G. Levi, editors, *Foundations of Logic and Functional Programming*, pages 67–113. Springer Verlag, 1986. LNCS 306.

[MB79] Saunders MacLane and Garret Birkoff. *Algebra.* Collier Macmillan, 1979.

[Plo69] Gordon D. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. 1969.

[PW78] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.

[Rey69] John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In *Machine Intelligence*, volume 5, pages 135–151. 1969.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12:23–41, January 1965.