Journal of Automated Reasoning **11**: 185–212, 1993.

Competing for the *AC*-Unification Race

ALEXANDRE BOUDET
LRI, CNRS URA 410, Université Paris-Sud, Bât 490, 91405 Orsay Cedex, France

(Received: 18 November 1992)

Abstract. We describe our implementation of the unification algorithm for terms involving some associative-commutative operators plus free function symbols described by Boudet *et al.* The first goal of this implementation is efficiency, more precisely competing for the *AC Unification Race*. Although our implementation has been designed for good performance when applied to non-elementary *AC*-unification problems, it is also very efficient on elementary problems. Our implementation, written in C and running on Sun workstations, is to be compared with the implementations in LISP, on Symbolics LIPS machines.

Key words. Unification, equation solving, logic programming, automated theorem proving, term rewriting.

1. Introduction

Robinson [28] coined the term *unification* for the process of solving an equation in a term algebra. Ever since, unification has been playing a key role in automated deduction and logic programming. Plotkin [27] suggested that some properties could be taken into account while solving equations, namely, the properties expressible by an equational theory.

Let $\mathscr{F}$ be a set of function symbols and $\mathscr{X}$ a denumerable set of variables. $\mathscr{T}(\mathscr{F},\mathscr{X})$ is the free $\mathscr{F}$-algebra over $\mathscr{X}$. An *equational axiom* is a pair $\langle s,t\rangle$ of terms of $\mathscr{T}(\mathscr{F},\mathscr{X})$, denoted by $s=t$. Given a set $E=\{s_1=t_1,\ldots,s_n=t_n\}$ of equational axioms, the corresponding *equational theory* $=_E$ is the least congruence containing all the instances of the axioms. A theory E is *consistent* if the set of its equivalence classes is not reduced to a singleton; E is *regular* if the set of variables of s and t are the same for each $s=t\in E$ and *collapse-free* if it has no axiom $x=s$ where x is a variable and s is not equal to x.

Unification in an equational theory E consists in finding the substitutions σ that make an equation valid in the class of models of E. In other words, σ is an *E-solution* of the equation $s\stackrel{?}{=}t$ iff

$$E\models s\sigma=t\sigma$$

or, equivalently, by using Birkhoff's completeness theorem [2],

$$s\sigma=_E t\sigma.$$

The problem of the existence of such a solution is undecidable in general, but there are *ad hoc* algorithms for many theories of interest, including associativity-commutativity which is probably the most frequently encountered one [8, 14, 15, 17, 21, 22, 25, 30].

We describe an implementation in C of the algorithm for associative-commutative unification given in [5]. Several choices contribute to the efficiency of the program:

- Variable sharing allows to have one unique occurrence of each variable in the system and to use *DAG solved forms* instead of solved forms. This saves space and turns out to be essential for unification in the free theory. The size of a most general unifier may be exponential with respect to the original problem, while the size of a DAG solved form is linear. The reader is referred to [20] for a study of solved forms *vs.* DAG solved forms.
- AC-unification requires branching when an elementary AC-unification problem has several solutions. The way this branching is implemented is essential for the efficiency. We have chosen to save the problem *physically*, that is by copying a byte array, rather than copying a symbolic expression.
- Solving a subproblem associated with an associative-commutative operator rather than one equation at a time is also crucial. It may be that one equation has an intractable number of solutions while the whole subproblem is solvable in practice.

  For instance, the problem $x + x + x + x \stackrel{?}{=} x_1 + x_2 + x_3 + x_4 \wedge x + x \stackrel{?}{=} x_1 + x_2$ has 487 most general unifiers, computed in a very short time by our program, while the first equation alone has 34 359 607 481 most general unifiers! For further evidence of the advantage of the *system-solving approach*, see [1].
- For elementary AC-unification, it is necessary to solve linear Diophantine equations. The algorithm described in [5, 11] solves efficiently a system of linear Diophantine equations, and it is easy to implement. When the problem has few solutions, the efficiency of AC unification heavily depends on the efficiency of this step. When solving the equations one at a time, the number of variables at each step is equal to the number of minimal solutions of the subproblem solved so far. Evidence of the advantage of solving a whole system at one time can be found in [11].
- When an elementary AC-unification problem has many solutions, most of the time is spent in combining the minimal solutions of the system of linear Diophantine equations and most of all, in constructing each solution. The care with which these two steps were implemented is the main reason for the satisfactory efficiency of the program.

The paper is organized as follows: Section 2 describes the problem, gives some basic definitions, and recalls the algorithm for combining several AC theories and the free theory. Section 3 describes the data structure that we use for unification problems, and Section 4 shows how the equations of the original problem are stored into this data structure. Section 5 briefly describes how the occur-check is performed. The most important part of the implementation deals with elementary AC-unification. This is described in Section 6, together with the algorithm for solving systems of linear Diophantine equations. Section 7 deals with the branching that results because AC theories are not unitary and the implementation of the failure rules. Finally, Section 8 gives some benchmarks for elementary AC problems taken

from [7], as well as new benchmarks for non-elementary problems. Our notations are consistent with [13]; for example, $t(p)$ denotes the symbol in t at the position p, $t(\Lambda)$ the top function symbol of t, $t[s]_p$ the term t where the subterm at position p is replaced by s. The subterm at position p in t is denoted by $t|_p$. The notation $t[x]$ simply means that x occurs in t. The set of variables of a term t (resp. a unification problem P) is denoted by $Var(t)$ (resp. $Var(P)$). The syntactic equality will be denoted by $\equiv$.

2. The Problem and the Algorithm

2.1. UNIFICATION PROBLEMS

Following Martelli and Montanari [26] and Kirchner [22], we envision a unification algorithm as a simplification process that transforms a unification problem until a solved form is reached, from which a most general solution is easily obtained.

Rather than considering systems of equations (or multiequations) and disjunctions of systems, we consider unification problems as a particular case of *equational problems* [10], with no negations nor universal quantifiers.

DEFINITION. A *unification problem* is a first-order predicate calculus formula involving only one binary predicate $\stackrel{?}{=}$, built up using disjunction, conjunction, and existential quantifiers only.

- T is the *trivial* unification problem.
- F is the *unsolvable* unification problem.
- An *atomic* unification problem is an equation $s \stackrel{?}{=} t$.[1]
- If P_1 and P_2 are unification problems and x is a variable, then $P_1 \wedge P_2$, $P_1 \vee P_2$ and $(\exists x)P_1$ are unification problems.

The E-solutions of a unification problem P are the substitutions σ that make the formula valid when the predicate $\stackrel{?}{=}$ is interpreted as the equational theory $=_E$.

DEFINITION. Let E be an equational theory.

- The unification problem F has no E-solution.
  The unification problem T has the E-solution σ for any substitution σ.
- An *E-solution* of $s \stackrel{?}{=} t$ is a substitution σ such that $s\sigma =_E t\sigma$.
- An *E-solution* of $P_1 \wedge P_2$ (resp. $P_1 \vee P_2$) is a substitution σ that is an *E-solution* of P_1 and (resp. or) P_2.
- An *E-solution* of $(\exists x)P$ is a substitution σ, such that there exists a term t, such that σ is an E-solution of $P\{x \mapsto t\}$.

Two E-unification problems are *E-equivalent* (or simply equivalent if E is clear from the context) if they have the same sets of solutions.

The explicit use of existential quantifiers in the syntax makes the set of variables of the original problem irrelevant. Partial correctness can be proven by just showing

that the rules preserve the sets of solutions, without regard to the original problem. The importance of existential quantifiers is illustrated by the following example:

Example 1. Let E be the free theory. The unification problems $P_1 \equiv x \stackrel{?}{=} y$ and $P_2 \equiv x \stackrel{?}{=} x' \wedge y \stackrel{?}{=} x'$ are not equivalent since $\sigma = \{x \mapsto a, y \mapsto a, x' \mapsto b\}$ is a solution of P_1, but not of P_2. On the other hand, P_1 and the unification problem $P_3 \equiv (\exists x')x \stackrel{?}{=} x' \wedge y \stackrel{?}{=} x'$ are E-equivalent for any equational theory E.

As shown by the above example, an advantage of this definition is that it gives a clear notion of what the new variables introduced in the unification process are, specifically, nothing but existentially quantified variables.

2.2. COMPLETE SETS OF SOLVED FORMS

It may be that a unification problem has a *most general unifier*, which is a particular solution that somehow expresses all the other solutions:

DEFINITION. Let P be a unification problem and V the set of the free variables occurring in P. The substitution σ is a *most general E-unifier* of P if

- σ is an E-solution of P;
- for all E-solutions θ of P, there exists a substitution ρ such that $\forall x \in V$, $x\theta =_E x\sigma\rho$.

DEFINITION. The unification problem P is in *solved form* if $P \equiv T$, or $P \equiv F$, or

$$P \equiv (\exists y_1, \ldots, y_p)x_1 \stackrel{?}{=} t_1 \wedge \cdots \wedge x_n \stackrel{?}{=} t_n,$$

where $\{x_1, \ldots, x_n\} \cap \{y_1, \ldots, y_p\} = \emptyset$ and x_i occurs only once[2] in P for $1 \leqslant i \leqslant n$.

Solved forms are essentially substitutions, and they represent their most general unifier.

LEMMA 1. *Let* $P \equiv (\exists y_1, \ldots, y_p)x_1 \stackrel{?}{=} t_1 \wedge \cdots \wedge x_n \stackrel{?}{=} t_n$ *be a unification problem in solved form. The substitution* $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ *is a most general E-unifier of P, for any consistent equational theory E.*

A major problem with solved forms is that their size may be exponential with respect to the size of the original unification problem, even in the free theory.[3] We prefer to use DAG solved forms [20], which are more compact and allow *variable sharing*. Evidence of the efficiency of variable sharing is given in [12].

DEFINITION. A unification problem P is in *DAG solved form* if $P \equiv T$, or $P \equiv F$, or

$$P \equiv (\exists y_1, \ldots, y_p)x_1 \stackrel{?}{=} t_1 \wedge \cdots \wedge x_n \stackrel{?}{=} t_n,$$

where $\forall i, j \in [1, \ldots, n]$

1. $x_i \neq x_j$ for $i \neq j$.
2. $x_i \in Var(t_j) \Rightarrow i > j$.

3. $x_i \notin \{y_1, \ldots, y_p\}$ or
   $\exists j < i$ such that $x_j \notin \{y_1, \ldots, y_p\}$ and $x_i \in Var(t_j)$.

The first condition is already required in solved forms, the second one expresses acyclicity, and the last one is a usefulness condition, preventing the use of new variables if not necessary.

It is easy to deduce a solved form from a DAG solved form: one simply needs to apply the two following rules as long as possible.

Rep (Replacement)
$(\exists y_1, \ldots, y_n) x \stackrel{?}{=} s \wedge P \Rightarrow (\exists y_1, \ldots, y_n) x \stackrel{?}{=} s \wedge P\{x \mapsto s\}$
if $x \in Var(P)$, and $x \notin Var(s)$, and $s \notin \mathscr{X}$ or
$s, e \in Var(P)$, and $x \not\equiv s$, and x is existentially quantified, or s is free.

EQE (Existential Quantifiers Elimination)
$(\exists x, y_1, \ldots, y_n) x \stackrel{?}{=} s \wedge P \Rightarrow (\exists y_1, \ldots, y_n) P$
if $x \notin Var(P) \cup Var(s)$.

It is trivial to show that these two rules preserve the sets of solutions and that they terminate, starting with a DAG solved form. This allows us to compute complete sets of DAG solved forms, rather than complete sets of most general unifiers as usually defined (see, e.g., [20]).

DEFINITION. A *complete set of DAG solved forms* of a unification problem P is a set $\{P_1, \ldots, P_n\}$ of unification problems in DAG solved form such that P and $P_1 \vee \cdots \vee P_n$ are equivalent. A problem P' is a *DAG solved form of* P if P' is a member of a complete set of DAG solved forms of P.

Note that not all theories have finite complete sets of (DAG) solved forms, nor even complete sets of (DAG) solved forms, since solved forms correspond to unifiers. However, the rules given in Section 2.3 apply to *finitary* theories, that is, theories having finite complete sets of (DAG) solved forms.

Rather than introducing disjunctions in the unification problems, we prefer to give the rules in a non-deterministic form. This is just a matter of notation, since it is actually enough to be able to solve conjunctions of equations and to collect all the solutions when non-deterministic branching occurs. More precisely, we will consider unification problems in prenex form, with no disjunctions:

$$(\exists y_1, \ldots, y_n) s_1 \stackrel{?}{=} t_1 \wedge \cdots \wedge s_n \stackrel{?}{=} t_n$$

modulo the commutativity of $\stackrel{?}{=}$, the associativity-commutativity and idempotence of $\wedge$, and the rewrite rules $P \wedge T \to P$, $P \wedge F \to F$.

2.3. ASSOCIATIVE-COMMUTATIVE UNIFICATION

The associative-commutative theory of a binary function symbol $+$ is the theory presented by the two axioms $+(+(x, y), z) = +(x, +(y, z))$ (associativity) and

$+(x,y) = +(y,x)$ (commutativity). We aim at solving equations in the theory $E = E_0 \cup E_1 \cup \cdots \cup E_n$ where E_0 is the free theory over a set $\mathscr{F}$ of free function symbols, and for $1 \leqslant i \leqslant n$, E_i is the associative-commutative theory of the symbol $+_i$. Two different problems arise:

- Solving equations in $\mathscr{T}(\mathscr{F}_i, \mathscr{X})$, modulo E_i, where $\mathscr{F}_i$ is the singleton $\{+_i\}$ (or possibly $\{+_i\} \cup \mathscr{C}$ where $\mathscr{C}$ is a set of free constants, for a better efficiency).
- Combining the unification algorithms for the theories $E_0, E_1, \ldots, E_n$, in order to get an E-unification algorithm.

The first problem has been independently solved by Stickel [30] and Livesey and Siekman [25]. It requires solving linear Diophantine equations [5, 11, 15, 16, 18, 24, 23]. We implemented the algorithm described in [5, 11] which has the advantage of solving whole systems of linear Diophantine equations. From an implementation point of view, the most difficult part is the combining of the minimal solutions of the linear Diophantine equations.

The second problem is a difficult one because termination is hard to prove: The termination of Stickel's algorithm [30] remained an open problem for nine years until Fages [14] gave an adequate complexity measure.[4] We use the rules of [5] for unification in a combination of regular, collapse-free theories. These rules terminate; and the notion of *shared variables* gives, besides a simple termination proof, a bound on the number of calls to elementary AC unification.

We recall briefly the algorithm presented in [5], for combining unification algorithms for regular collapse-free theories. Actually, the regular collapse-free case is more general than what we need for AC theories. We shall extend, in Section 4, some of the rules for our specific combination problem. In particular, the occur-check and conflict rules may be extended.

Before giving the rules, we will define a data structure for unification problems in a combination of regular collapse-free theories.

DEFINITION. A *proper equation* is an equation $s \stackrel{?}{=} t$, where s and t are not both variables. A term t is *pure* in the theory E_i if $t \in \mathscr{T}(\mathscr{F}_i, \mathscr{X})$. An equation $s \stackrel{?}{=} t$ is *pure* in the theory E_i if $s, t \in \mathscr{T}(\mathscr{F}_i, \mathscr{X})$. Non-pure terms or equations are called *heterogeneous*. A non-variable, proper subterm u of a term t is an *alien subterm of* t if the top function symbol of u does not belong to the same theory as the function symbol immediately above u in t.

To apply the rules, we write a unification problem as follows:

$$P \equiv (\exists y_1, \ldots, y_p) P_V \wedge P_H \wedge P_0 \wedge P_1 \wedge \cdots \wedge P_n,$$

where P_V is the conjunction of the *non-proper equations* $x \stackrel{?}{=} y$ of P; P_H is the conjunction of the *heterogeneous equations* of P; and P_i is the conjunction of the proper equations of P, pure in the theory E_i.

The first rule is a classical one. Since the goal is to reduce unification in a

combination of theories to the unification in each elementary theory, it is natural to split heterogeneous terms so as to make a pure term appear. The rule is the following:

VA (VariableAbstraction)
$s[u]_p \stackrel{?}{=} t \Rightarrow (\exists x) \quad s[x]_p \stackrel{?}{=} t \wedge x \stackrel{?}{=} u$
if u is an alien subterm of $s[u_p]$ at position p. (x is a new variable.)

Variable abstraction preserves the sets of solutions according to the semantics of existential quantifiers. As an example, the problem $x +_1 (y +_2 z) \stackrel{?}{=} u +_1 v$ becomes $(\exists w) x +_1 w \stackrel{?}{=} u +_1 v \wedge w \stackrel{?}{=} y +_2 z$, after applying variable abstraction.

After one or more applications of variable abstraction, it is possible that some equations will be pure in a theory E_i. The pure subproblem P_i can be solved by using the following rule:

***E*-Res**
$P_i \Rightarrow P_i'$
if P_i is not in a (DAG) solved form and P_i' a (DAG) solved form of P_i.

For instance, a DAG solved form of $P_1 \equiv x +_1 w \stackrel{?}{=} u +_1 v$ is (among the seven DAG solved forms of a complete set)

$$P_1' \equiv (\exists z_1, z_2, z_3) x \stackrel{?}{=} z_1 \wedge w \stackrel{?}{=} z_2 +_1 z_3 \wedge u \stackrel{?}{=} z_1 +_1 z_2 \wedge v \stackrel{?}{=} z_3.$$

After applying ***E*-Res** with this DAG solved form, the problem one obtains is

$$(\exists w, z_1, z_2, z_3) \underbrace{x \stackrel{?}{=} z_1 \wedge v \stackrel{?}{=} z_3}_{P_V'} \wedge \underbrace{w \stackrel{?}{=} z_2 +_1 z_3 \wedge u \stackrel{?}{=} z_1 +_1 z_2}_{P_1'} \wedge \underbrace{w \stackrel{?}{=} y +_2 z}_{P_2'}.$$

Clearly, ***E*-Res** is sound (i.e., it introduces no new solutions) but surprisingly, the completeness (i.e., no solutions are lost) is not straightforward. A complete proof can be found in [3]. If P_i has no solution, its only (DAG) solved form is F. This rule is non-deterministic: If a complete set of (DAG) solved forms is not a singleton, then the rule must be applied with all its elements.

In a combination of collapse-free theories over disjoint sets of function symbols, the equations $s \stackrel{?}{=} t$, where s and t have their top function symbols belonging to different theories, have no solutions. This statement is easily proven by contradiction and induction on the alleged proof. This yields the correctness of the following two rules:

Conflict 1
$s \stackrel{?}{=} t \Rightarrow F$
if $s(\Lambda) \in \mathscr{F}_i$, and $t(\Lambda) \in \mathscr{F}_j$, and $i \neq j$.

Conflict 2
$x \stackrel{?}{=} s \wedge x \stackrel{?}{=} t \Rightarrow F$
if $s(\Lambda) \in \mathscr{F}_i$, and $t(\Lambda) \in \mathscr{F}_j$, and $i \neq j$.

Note that in our example, the rule **Conflict 2** applies to the two equations $w \stackrel{?}{=} z_2 +_1 z_3$ and $w \stackrel{?}{=} y +_2 z$ of the problem yielded by ***E*-Res**.

In the case of unification with several AC symbols plus free operators, we shall extend these rules to take into account the operator clashes in the free theory (see Section 4).

The standard occur-check is not complete in a combination of regular collapse-free theories: The equation $x \stackrel{?}{=} f(x)$ has the solution $\{x \mapsto a\}$, in the regular collapse-free theory $RC = \{a = f(a)\}$. However, a *compound cycle* (i.e., involving operators from different theories) has no solutions. Hence we have the following rule:

Check*
$P \Rightarrow F$
if P contains a compound cycle $x_1 \stackrel{?}{=} s_1[x_2] \wedge x_2 \stackrel{?}{=} s_2[x_3] \wedge \cdots \wedge x_n \stackrel{?}{=} s_n[x_1]$, with $s_i(\Lambda) \in \mathscr{F}_h$, and $s_j(\Lambda) \in \mathscr{F}_k$, and $h \neq k$ for some $i,j \in [1,\ldots,n]$.

For our implementation, we shall extend this rule (in Section 5), since our combination is a *simple theory*, that is, a theory in which the equations of the form $x \stackrel{?}{=} s$, where s is a non-variable term containing x have no solutions.

To compute a DAG solved form, the classical term replacement (or variable elimination) rule is not needed. All we need is a restriction of the rule to non-proper equations:

Var-Rep
$(\exists z_1,\ldots,z_n)x \stackrel{?}{=} y \wedge P \Rightarrow (\exists z_1,\ldots,z_n)x \stackrel{?}{=} y \wedge P\{x \mapsto y\}$
if $x,y \in Var(P)$, and x is existentially quantified or y is free.

The variable replacement rule, **Var-Rep**, is very important because it handles the interactions between the different pure subproblems. Indeed, solving a subproblem P_i with the rule ***E*-Res** may yield a non-proper equation $x \stackrel{?}{=} y$. If **Var-Rep** is then applied to this equation, it is possible that a previously solved subproblem P_j becomes unsolved, because x and y have been identified.

Finally, the last rule **EQE** removes useless existential quantifiers. It is clear that it preserves the sets of solutions.

EQE (Existential Quantifiers Elimination)
$(\exists x, y_1,\ldots,y_n)x \stackrel{?}{=} s \wedge P \Rightarrow (\exists y_1,\ldots,y_n)P$
if $x \notin Var(P) \cup Var(s)$.

The set of rules $\mathscr{S} = \{$**VA**, ***E*-Res**, **Conflict1**, **Conflict2**, **Check***, **Var-Rep**, **EQE**$\}$ preserves the sets of solutions. It terminates for every input P and the irreducible problems are DAG solved forms. Hence $\mathscr{S}$ implements a complete unification algorithm for the combination of regular collapse-free theories.

We now show how our rules compute a solution for the problem given above,

when **Conflict2** does not apply after ***E*-Res**:

$$x +_1 (y +_2 z) \stackrel{?}{=} u +_1 v$$

VA $\Rightarrow$

$$(\exists w) x +_1 w \stackrel{?}{=} u +_1 v \wedge w \stackrel{?}{=} y +_2 z$$

***E*-Res** $\Rightarrow$

$$(\exists w, w_1, w_2, w_3) w \stackrel{?}{=} w_3 \wedge u \stackrel{?}{=} w_1 \wedge x \stackrel{?}{=} w_1 +_1 w_2 \wedge v \stackrel{?}{=} w_2 +_1 w_3 \wedge w \stackrel{?}{=} y +_2 z$$

Var-Rep $\Rightarrow$

$$(\exists w, w_1, w_2, w_3) w \stackrel{?}{=} w_3 \wedge u \stackrel{?}{=} w_1 \wedge x \stackrel{?}{=} w_1 +_1 w_2 \wedge v \stackrel{?}{=} w_2 +_1 w \wedge w \stackrel{?}{=} y +_2 z$$

Var-Rep $\Rightarrow$

$$(\exists w, w_1, w_2, w_3) w \stackrel{?}{=} w_3 \wedge u \stackrel{?}{=} w_1 \wedge x \stackrel{?}{=} u +_1 w_2 \wedge v \stackrel{?}{=} w_2 +_1 w \wedge w \stackrel{?}{=} y +_2 z$$

EQE $\Rightarrow$

$$(\exists w, w_1, w_2) u \stackrel{?}{=} w_1 \wedge x \stackrel{?}{=} u +_1 w_2 \wedge v \stackrel{?}{=} w_2 +_1 w \wedge w \stackrel{?}{=} y +_2 z$$

EQE $\Rightarrow$

$$(\exists w, w_2) x \stackrel{?}{=} u +_1 w_2 \wedge v \stackrel{?}{=} w_2 +_1 w \wedge w \stackrel{?}{=} y +_2 z$$

The last problem is irreducible, and it is a DAG solved form. The solved form

$$(\exists w_2) x \stackrel{?}{=} u +_1 w_2 \wedge v \stackrel{?}{=} w_2 +_1 (y +_2 z)$$

can be obtained by applying the replacement rule, followed by the rule **EQE** to the equation $w \stackrel{?}{=} y +_2 z$. One can check that $\{x \mapsto u +_1 w_2 \wedge v \mapsto w_2 +_1 (y +_2 z)\}$ is a solution of the original problem.

3. The Data Structure

The fundamental notion for the choice of the data structure is the notion of *variable sharing*. Variable sharing and the use of DAG solved forms avoid having solved forms whose size may be exponential with respect to the size of the input, in the free theory. Hence the first implementation choice:

CHOICE 1. *A variable x may not be duplicated in the memory.*

Variables will be referenced by pointers, and a variable position in a term will be a pointer to that variable. A non-variable term t will be implemented as a pair whose first item is its top function symbol $t(\Lambda)$ and whose second item is the list of the subterms $t|_1, \ldots, t|_{arity(t(\Lambda))}$.

Moreover, we have chosen an *AC*-flat representation such that two occurrences of a same *AC* symbol one immediately above the other are forbidden, *AC* operators having a variable arity.

CHOICE 2

- *A term of* $T(\{+_i\} \cup \mathscr{C}, \mathscr{X}) \backslash (\mathscr{C} \cup \mathscr{X})$ *is of the form* $+_i(s_1, \ldots, s_n)$, *where* $n \geqslant 2$ *and* $s_i \in \mathscr{C} \cup \mathscr{X}$ *for* $i \in [1, \ldots, n]$.
- *A term* t *of* $T(\{+_1, \ldots, +_n\} \cup \mathscr{F}_0, \mathscr{X})$, *where* $+_1, \ldots, +_n$ *are AC symbols and* $\mathscr{F}_0$ *is a set of free operators, is of one of the following forms:*
  - t *is a variable (pointer).*
  - $t(\Lambda) \in \mathscr{F}_0$, *and* t *is represented by the pair* $(t(\Lambda), l)$, *where* l *is the list of the pointers to* $t|_1, \ldots, t|_{arity(t(\Lambda))}$.
  - $t(\Lambda) = +_i$, *and* t *is represented by the pair* $(t(\Lambda), l)$, *where* l *is a list of pointers to* $t|_1, \ldots, t|_m$, *with* $m \geqslant 2$ *and* $t|_j(\Lambda) \neq +_i$ *for* $j \in [1, \ldots, n]$.

Given choice 1, we can collect all the variables of the system into some data structure, which is a list in our implementation. This allows access to the set of all the variables of the problem.

We introduce the notion of quasi-solved variable:

DEFINITION. An equation $x \stackrel{?}{=} s$, where x is a variable, is called a *quasi-solved equation.* The variable x is *quasi-solved within the unification problem* P if it occurs at most once as a side of a quasi-solved equation $x \stackrel{?}{=} s$.

Our two first implementation choices allow a special representation for the quasi-solved equations.

CHOICE 3. *The physical implementation of a variable* x *contains, besides the symbol* 'x', *a pointer field to a term called* value field *of* x. *If* x *is quasi-solved, the equation* $x \stackrel{?}{=} s$ *will be physically represented by the presence of a pointer to the term* s *in the value field of* x. *We will say that* x has a value s.

As an example, the unification problem $z \stackrel{?}{=} f(y, x, x) \wedge x + y \stackrel{?}{=} g(z, y)$ will be represented, according to our implementation choices, as shown in Figure 1. If there is a non-proper equation $x \stackrel{?}{=} y$, it is allowed that x has a pointer to y in its value field, but it is necessary to avoid having a cycle of such non-proper equations. Thus we have a notion of *representative* for the classes of the equivalence $=_V$ generated by the non-proper equations.

DEFINITION (Representative of a variable). If a variable x has a value field containing a pointer to a non-variable term or a *null* pointer (i.e., a particular value that addresses no location in the memory), then x is its own *representative.* Otherwise, if x has in its value field a pointer to another variable y, then the representative of x is the representative of y.

The parsing of the equations of the initial problem is done by a parser generated by YACC, which puts the equations into a stack and performs the variable sharing. On top of the head function symbol and the list of immediate subterms, a non-variable term contains a field, containing its size, which is initialized at parsing time. The variables will have an extra counter field which will be used for the occur-check.

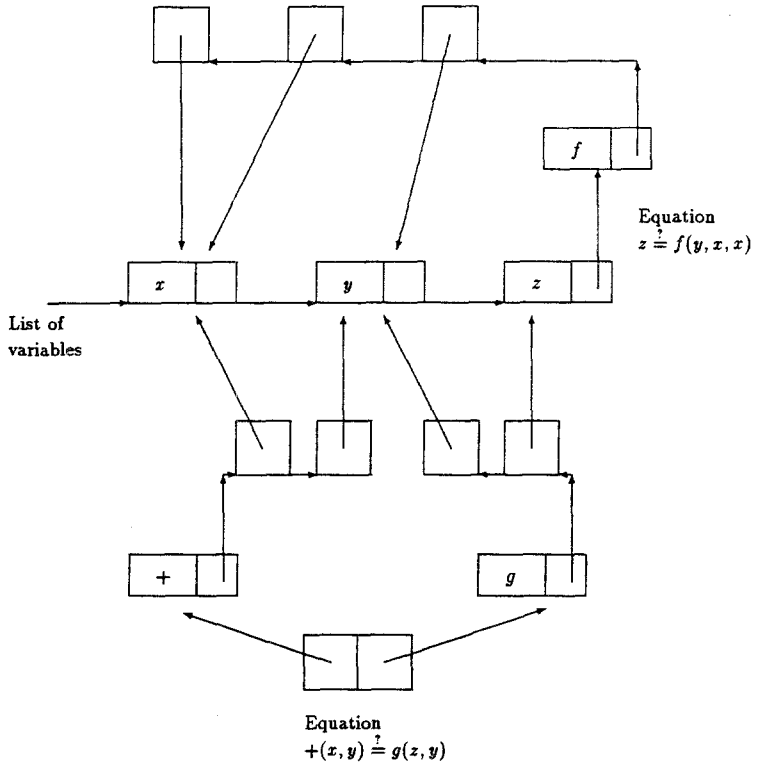


Fig. 1. Internal representation of the equation $x + y \stackrel{?}{=} g(z, y)$ and of the quasi-solved equation $z \stackrel{?}{=} f(y, x, x)$.

This field is initialized with the value 0. The value fields of the variables contain a null pointer at the beginning.

The program takes, as input, a stack (conjunction) of equations whose variables are shared, and the list of the variables of the problem. It returns a DAG solved form represented by the list of the variables whose value field has been modified in order to point to their values. Another global variable is the list (conjunction) of pure *AC* problems (i.e., the subproblems involving only one *AC* operator, free constants, and variables). To improve the efficiency, the equations with a same free function symbol on both sides will be decomposed. Hence, the subproblem P_0 associated with the free theory contains only quasi-solved equations. The program is not a functional program: It works only by side effects.

4. Insertion of the Equations into the Data Structure

In this section we show how to insert a unification problem into our data structure. Some additional transformations are used, whose soundness and completeness are straightforward. As long as the equations stack returned by the parser is not empty, one equation is popped and is processed as follows.

If possible, apply the first rule for the free theory:

Decomposition
$f(s_1, \ldots, s_n) \stackrel{?}{=} f(t_1, \ldots, t_n) \Rightarrow s_1 \stackrel{?}{=} t_1 \wedge \cdots \wedge s_n \stackrel{?}{=} t_n$
if f is a free function symbol.

If the rule applies, the resulting equations are pushed onto the stack. **Clash** is also a classical rule for free function symbols, and the rule extends to AC operators.

Clash
$f(s_1, \ldots, s_n) \stackrel{?}{=} g(t_1, \ldots, t_m) \Rightarrow F$
if $f, g \in \mathscr{F}_0 \cup \{+_1, \ldots, +_n\}$ with $f \neq g$.

In the case where f and g belong to two different signatures, **Clash** implements the rule **Conflict1** of the combination algorithm. The rule **Conflict2** will possibly apply when inserting quasi-solved equations, as shown below.

Section 7 will expose how the failure rules that return F are implemented. For the moment we will just say that the program fails.

If none of these two rules applies to the top equation in the stack, two cases may occur:

- the top equation is of the form $+(s_1, \ldots, s_n) \stackrel{?}{=} +(t_1, \ldots, t_m)$, where $+$ is an AC symbol, or
- the top equation is a quasi-solved equation $x \stackrel{?}{=} s$.

In the first case, **VA** is applied, if necessary, so as to obtain a pure equation in $T(\{+\} \cup \mathscr{C}, \mathscr{X})$ which is added to the corresponding pure AC subproblem. The other equations yielded by **VA** are pushed onto the stack.

For quasi-solved equations $x \stackrel{?}{=} s$, several cases are to be considered.

4.1. QUASI-SOLVED EQUATIONS $x \stackrel{?}{=} +_i(s_1, \ldots, s_n)$

When the equation on top of the stack is a quasi-solved equation $x \stackrel{?}{=} s$, with $s \notin \mathscr{X}$ and $s(\Lambda) = +_i$, the rule **VA** is applied if necessary so as to obtain a pure equation $x \stackrel{?}{=} s_1$ with $s_1 \in \mathscr{T}(\{+_i\} \cup \mathscr{C}, \mathscr{X}) \backslash \mathscr{X}$. The other equations yielded by **VA** are stacked.

- If x has a null pointer in its value field, a pointer to s_1 is put in its value field. This transformation does nothing but represent a quasi-solved equation according to our implementation choices.
- If the variable x has a value with a top function symbol other than $+_i$, the rule **Conflict2** applies, yielding F.
- If x has a value field pointing to a term t whose top function symbol is $+_i$, the equation $s \stackrel{?}{=} t$ is stacked. This transformation corresponds to the following rule, which obviously preserves the solution sets and leaves x quasi-solved.

AC-Merge
$x \stackrel{?}{=} s \wedge x \stackrel{?}{=} t \Rightarrow x \stackrel{?}{=} s \wedge s \stackrel{?}{=} t$
if s and t are terms in $\mathscr{T}(\{+_i\} \cup \mathscr{C}, \mathscr{X})\backslash\mathscr{X}$.

- If the variable x has a value field pointing to another variable, then the equation $y \stackrel{?}{=} s_1$ is inserted rather than $x \stackrel{?}{=} s_1$, where y is the representative of x. This last case corresponds to the transformation

Representative
$x \stackrel{?}{=} y \wedge x \stackrel{?}{=} s_1 \Rightarrow x \stackrel{?}{=} y \wedge y \stackrel{?}{=} s_1$
if y is the representative of x.

4.2. QUASI-SOLVED EQUATIONS $x \stackrel{?}{=} f(s_1, \ldots, s_n)$

The equations $x \stackrel{?}{=} s$, where the top function symbol of s is a free symbol, are processed in a similar way. The only difference is that the merge must take into account the size of the terms in order to ensure termination. The rule **Merge** uses the size field of the terms initialized at parsing time:

Merge
$x \stackrel{?}{=} s \wedge x \stackrel{?}{=} t \Rightarrow x \stackrel{?}{=} s \wedge s \stackrel{?}{=} t$
if s and t are terms of $\mathscr{T}(\mathscr{F}_0 \cup \mathscr{C}, \mathscr{X})\backslash\mathscr{X}$ and $|s| \leqslant |t|$.

4.3. NON-PROPER EQUATIONS $x \stackrel{?}{=} y$

The variable replacement rule is built in via the notion of representative. If an equation is of none of the forms examined so far, then it must be a non-proper equation $x \stackrel{?}{=} y$. Again there are several cases:

- If x and y have the same representative, there is no need to do anything. This corresponds to the trivially correct transformation

$$x \stackrel{?}{=} x_1 \wedge x_1 \stackrel{?}{=} x_2 \wedge \cdots \wedge x_n \stackrel{?}{=} y \wedge y \stackrel{?}{=} x$$
$$\Rightarrow x \stackrel{?}{=} x_1 \wedge x_1 \stackrel{?}{=} x_2 \wedge \cdots \wedge x_n \stackrel{?}{=} y.$$

- Otherwise, let x' and y' be the representatives of x and y, respectively.
  - If x' has no value, a pointer to y' is put into the value field of x'. This corresponds to the transformation

$$x \stackrel{?}{=} x_1 \wedge \cdots \wedge x_n \stackrel{?}{=} x' \wedge y \stackrel{?}{=} y_1 \wedge \cdots \wedge y_n \stackrel{?}{=} y' \wedge x \stackrel{?}{=} y$$
$$\Rightarrow x \stackrel{?}{=} x_1 \wedge \cdots \wedge x_n \stackrel{?}{=} x' \wedge y \stackrel{?}{=} y_1 \wedge \cdots \wedge y_n \stackrel{?}{=} y' \wedge x' \stackrel{?}{=} y',$$

    which obviously preserves the sets of solutions.
  - If y' has no value, we proceed symmetrically.
  - If x' and y' have values with different top function symbols, there is a clash, and

the program fails. This implements the rule **Conflict2** of the combination algorithm.

- If x' and y' have respective values s and t, with the same associative-commutative top function symbol $+_i$, the equation $s \stackrel{?}{=} t$ is added to the subproblem P_i.
- Finally, if x' and y' have respective values s and t, with the same free top function symbol with $|s| \leqslant |t|$, the pointer to t in the value field of y' is replaced by a pointer to x', and the equation $s \stackrel{?}{=} t$ is stacked. If $|s| > |t|$, we proceed symmetrically.

After this preliminary process, a problem equivalent to the original one is obtained where

1. all the quasi-solved equations $x \stackrel{?}{=} s$ are represented by a pointer to s in the value field of x;
2. all the equations are pure;
3. each variable has a representative (possibly itself), and only the representatives may have a non-variable value; and
4. the subproblem P_0 associated with the free theory contains only quasi-solved equations.

If there is no cycle in the subproblem P_0, then P_0 is in a DAG solved form. Otherwise, the problem has no solution, since the combination of the free theory and some associative-commutative theories is a *simple* theory, i.e., a theory where the equations $s \stackrel{?}{=} t[s]$ between a term and one of its proper subterms have no solutions. The counter field of the variables is used for the occur-check.

5. Occur-check

The fact that we have a simple theory allows us to replace the rule **Check*** of the general combination algorithm by the following:

Occur-Check
$x_1 \stackrel{?}{=} s_1[x_2] \wedge x_2 \stackrel{?}{=} s_2[x_3] \wedge \cdots \wedge x_n \stackrel{?}{=} s_n[x_1] \Rightarrow F$
if there exists $i \in [1, \ldots, n]$ such that $s_i \notin \mathscr{X}$.

The occur-check is performed by means of a topological sort with respect to the *predecessor relation.*

DEFINITION. The variable x is a *predecessor* of the variable y if there exists an equation $x \stackrel{?}{=} s[y]$, where $s \notin \mathscr{X}$, in the problem where each variable has been replaced by its representative. If y has n occurrences in s, x is a *predecessor of y of multiplicity n.* Symmetrically, y is a *successor of x of multiplicity n.*

We actually look for a cycle in the problem where all the variables have been replaced by their representatives.

In each of the transformations described above, the counter fields of the variables

are maintained with the sum of the multiplicities of their predecessors. This is done as follows:

- When a pointer to a term s is set in the value field of a variable x for the first time, the term s is walked through, and at every variable position, the counter of the representative of the variable is incremented.
- When a value pointer to a term t is removed, the representatives of the variables of t are decremented similarly.
- When a pointer to a term s in the value field of a variable x is replaced by a pointer to another variable y, the counter of y is incremented by the value of the counter of x, and the counter of x is set to 0.

The occur-check proceeds as follows:

1. The representatives are counted in the list of the variables of the problem, and the representatives that have no predecessor (having the value 0 in their counter field) are stacked.
2. If the stack is empty, there is a cycle because every variable has a predecessor. Otherwise, a variable x is popped from the stack: This variable cannot belong to a cycle. Now having a proof that x belongs to no cycle, x can be removed from the graph of the successor relation without changing the existence or non-existence of a cycle. This is done by decrementing the counters of the representatives of the successors of x (which are available through the value field). When a variable has its counter decremented to 0, it cannot belong to any cycle, and it is stacked.
3. If as many variables have been popped as the number of representatives computed in 1, there is no cycle; else continue to 2.

If there is a cycle, the program fails; otherwise P_0 is in DAG solved form. We still have to implement the rule ***E*-Res** for the pure subproblems in an AC theory.

6. Elementary *AC*-Unification

Before solving the subproblem P_i, associated with an AC symbol $+_i$, the problem is walked through, and the variables are replaced by their representatives. In the following x^n will denote the term

$$\underbrace{x +_i \cdots +_i x}_{n \text{ times}}.$$

Remember that P_i contains two types of equations: the quasi-solved equations which are represented by pointers in the value fields of the variables, and the equations between two non-variable terms. Since there has been an occur-check for the quasi-solved equations, the conjunction of the quasi-solved equations of P_i is a problem in DAG solved form. We will take advantage of the general fact that if σ_1 is a most general solution of P_1, then the solutions of $P_1 \wedge P_2$ are the substitutions $\sigma_1\sigma_2$,

where σ_2 is a solution of $P_2\sigma_1$. Therefore, we only need to solve the non-quasi-solved equations of P_i, provided that the most general unifier of the quasi-solved ones has been applied to P_i. This is done by applying, as long as possible, the following rule:

E-Rep
$s[x] \stackrel{?}{=} t \wedge x \stackrel{?}{=} s_1 \Rightarrow s[s_1] \stackrel{?}{=} t \wedge x \stackrel{?}{=} s_1$
if $s[x] \stackrel{?}{=} t$ and $x \stackrel{?}{=} s_1$ are equations of P_i.

This rule terminates because the occur-check guarantees acyclicity.

6.1. SOLVING SYSTEMS OF LINEAR DIOPHANTINE EQUATIONS

Before we give the algorithm for elementary AC-unification, we show how to efficiently solve systems of linear Diophantine equations.

Consider the system $\mathscr{S}$ of homogeneous linear Diophantine equations

$$\begin{aligned} a_{11}x_1 + \cdots + a_{1m}x_m &= 0 \\ \vdots \qquad\qquad \vdots \quad & \vdots \\ a_{n1}x_1 + \cdots + a_{nm}x_m &= 0 \end{aligned}$$

with n equations and m unknowns, where the a_{ij} in $\mathbb{Z}$.

We are interested in the set of the non-zero solutions of $\mathscr{S}$ in $\mathbb{N}^m$ which are minimal wrt the ordering $>^m$ on $\mathbb{N}^m$, defined by $(a_1, \ldots, a_m) >^m (b_1, \ldots, b_m)$ if $a_i \geqslant b_i$ for $1 \leqslant i \leqslant m$, and there exists $j \in [1, \ldots, m]$ s.t. $a_j > b_j$.

Such a set always exists and it is finite. There exist several algorithms for computing $Sol(D)$ [9, 18, 23], but we will use the new one given in [5, 11] which extends [9] to the solving of a whole system, rather than of one equation at a time. A complete proof can be found in [5, 11].

Let e_i be the vector $(0, \ldots, 1, \ldots, 0)$ of $\mathbb{N}^m$, where the value 1 is at the ith position. The *defect* of the vector $\mathbf{v} = (v_1, \ldots, v_n)$ of $\mathbb{N}^m$ is the vector

$$\mathscr{D}(\mathbf{v}) = (a_{11}v_1 + \cdots + a_{1m}v_m, \ldots, a_{n1}v_1 + \cdots + a_{nm}v_m)$$

of $\mathbb{Z}^n$. Clearly, $\mathbf{v}$ is a solution iff $\mathscr{D}(\mathbf{v}) = (0, \ldots, 0)$.

Let $\mathbf{v}$ be a (non-zero) solution. We can write $\mathbf{v} = e_{i_1} + \cdots + e_{i_k}$, where $i_j \in [1, \ldots, m]$ for $1 \leqslant j \leqslant k$. Let $\mathbf{u}_0 = (0, \ldots, 0)$ and $\mathbf{u}_j = \mathbf{u}_{j-1} + e_{i_j}$ for $1 \leqslant j \leqslant k$ (we have $\mathbf{u}_k = \mathbf{v}$). With each member of this sequence, we associate its defect in $\mathbb{Z}^n$. Since $\mathbf{v}$ is a solution, $\mathscr{D}(\mathbf{u}_k) = \mathscr{D}(\mathbf{v}) = (0, \ldots, 0)$. Hence, for any (non-zero) solution, we have a 'geometrical proof', which is a sequence of vectors of $\mathbb{Z}^n$, starting and ending at the origin, each vector being obtained from the previous by adding the defect of a vector e_{i_j}. Now, a very simple geometrical remark allows a dramatic restriction of the search space for building every such proof:

Remark. Let $\mathbf{u}_j$ ($j \geqslant 1$) be a vector in the sequence that is not a solution. Every solution greater than $\mathbf{u}_j$ (wrt $>^m$) must be greater on at least one component a, such that $\mathscr{D}(\mathbf{u}_j + e_a)$ lies in the open half space delimited by the affine hyperplane orthogonal to $\mathscr{D}(\mathbf{u}_j)$ and containing the origin.[5]

This is so because otherwise the Euclidian norm of $\mathscr{D}(\mathbf{u}_{j+1}), \ldots, \mathscr{D}(\mathbf{u}_k)$ would always remain greater than or equal to that of $\mathscr{D}(\mathbf{u}_j)$. Hence, every solution greater than $\mathbf{u}_j$ can be constructed, starting from $\mathbf{u}_j$, by increasing one component satisfying this constraint. In other words, every solution $\mathbf{v} = (v_1, \ldots, v_m)$ of $\mathscr{S}$ s.t. $v_1 + \cdots + v_m = k$ corresponds with a sequence

$$\mathbf{u}_0 = (0, \ldots, 0),\ \mathbf{u}_1 = \mathbf{u}_0 + e_{j_1}, \ldots, \mathbf{u}_k = \mathbf{u}_{k-1} + e_{j_k} = \mathbf{v}$$

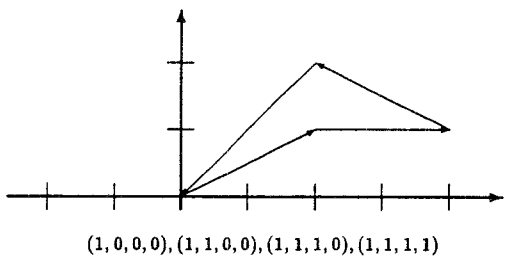
such that $\mathscr{D}(\mathbf{u}_j) \cdot \mathscr{D}(e_{i_{j+1}}) < 0$ for $1 < j < k$. A *son* of $\mathbf{u}$ is a vector $\mathbf{u} + e_j$ such that $\mathscr{D}(\mathbf{v}) \cdot \mathscr{D}(e_j) < 0$. The solutions greater than a vector $\mathbf{v}$ are greater than or equal to a son of $\mathbf{v}$. This is illustrated in Figure 2, which shows the sequences of defects corresponding to two sequences of vectors leading to the same solution.

The algorithm searches $\mathbb{N}^m \backslash \{(0, \ldots, 0)\}$, taking into account the geometrical restriction. It starts with the set

$$\mathscr{V}_1 = \{e_1, \ldots, e_m\}$$

of vectors and the set

$$\mathscr{M}_1 = \{e_i | 1 \leqslant i \leqslant m \ \& \ \mathscr{D}(e_i) = (0, \ldots, 0)\}$$



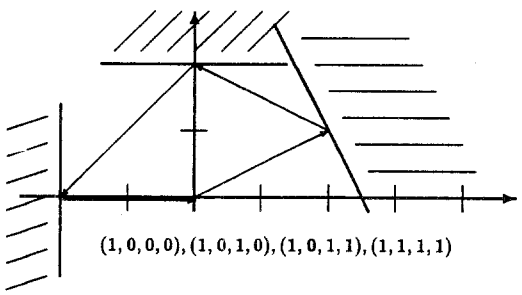


Fig. 2. Two sequences of defects corresponding to the solution $(1, 1, 1, 1)$ of the system.

$$2x_1 + 2x_2 - 2x_3 - 2x_4 = 0$$
$$x_1 \qquad + x_3 - 2x_4 = 0$$

Only the second will be constructed by the algorithm since the first one violates the geometrical restriction.

of minimal solutions. For $k > 1$, it computes the sets

$$\mathscr{V}_{k+1} = \{\mathbf{v} + e_j | \mathbf{v} \in \mathscr{V}_k \backslash \mathscr{M}_k \ \& \ \forall \mathbf{u} \in \mathscr{M}_k \mathbf{v} + e_j \not>^m \mathbf{u} \ \& \ \mathscr{D}(\mathbf{v}) \cdot \mathscr{D}(e_j) < 0\}$$

$$\mathscr{M}_{k+1} = \mathscr{M}_k \cup \{\mathbf{v} \in \mathscr{V}_{k+1} | \mathscr{D}(\mathbf{v}) = (0, \ldots, 0)\}.$$

The algorithm returns the first $\mathscr{M}_h$ such that $\mathscr{V}_h$ is empty. An additional constraint avoids redundancy: when a vector $\mathbf{v}$ has several sons $\mathbf{v} + e_{j_1}, \ldots, \mathbf{v} + e_{j_p}$, every j_lth component of $\mathbf{v} + e_{j_h}$ for $1 \leqslant l < h \leqslant p$ can be *frozen*. A vector with a frozen component j cannot be increased by e_j. This restriction preserves the completeness since all the solutions greater than $\mathbf{v}$ on their j_lth component will descend from $\mathbf{v} + e_{j_l}$. Therefore it is of no use to consider them again among the descendents of $\mathbf{v} + e_{j_h}$.

We explain below how the solutions of a system of linear Diophantine equations are used for elementary AC unification.

6.2. UNIFICATION IN $\mathscr{T}(\{+_i\} \cup \mathscr{C}, \mathscr{X})$

We recall some well-known results for the elementary AC-unification [22, 25, 30], and we illustrate them with an example that will be developed through the rest of this section.

Let P be the elementary AC unification problem $v +_i v +_i v \stackrel{?}{=} a +_i y +_i c +_i v$, where a and c are free constants. For the sake of efficiency, we include $\mathscr{C}$ into the signature associated with the AC theory of $+_i$.

DEFINITION. Let $s \stackrel{?}{=} t$ be a pure equation in $\mathscr{T}(\{+_i\} \cup \mathscr{C}, \mathscr{X})$. The *defect of the variable* x (*resp. of the constant* c) *with respect to the equation* $s \stackrel{?}{=} t$, denoted $d(x, s \stackrel{?}{=} t)$ (resp. $d(c, s \stackrel{?}{=} t)$), is the number of occurrences of x (resp. c) in s, minus its number of occurrences in t.

Let $P \equiv s_1 \stackrel{?}{=} t_1 \wedge \cdots \wedge s_n \stackrel{?}{=} t_n$ be a pure unification problem in $\mathscr{T}(\{+_i\} \cup \mathscr{C}, \mathscr{X})$. The *defect of the variable* x (*resp. the constant* c) *with respect to* P, denoted $d(x, P)$ (resp. $d(c, p)$), is the vector $(d_1, \ldots, d_n)$, where d_i is the defect of x (resp. c) with respect to the equation $s_i \stackrel{?}{=} t_i$. In our example, $d(v, P) = (2)$ and $d(c, P) = (-1)$.

Let $P \equiv t_1 \stackrel{?}{=} t'_1 \wedge \cdots \wedge t_p \stackrel{?}{=} t'_p$ be an elementary AC unification problem. Let $\{u_1, \ldots, u_n\}$ be the set of the variables and free constants occurring in P. For $i \in [1, \ldots, n]$, we associate with u_i an integer variable x_i, and we associate with the unification problem P, the system D of linear Diophantine equations

$$\begin{aligned} d(u_1, t_1 \stackrel{?}{=} t'_1)x_1 + d(u_2, t_1 \stackrel{?}{=} t'_1)x_2 + \cdots + d(u_n, t_1 \stackrel{?}{=} t'_1)x_n &= 0 \\ \vdots \qquad\qquad \vdots \qquad\qquad\qquad \vdots \qquad\quad &\vdots \\ d(u_1, t_p \stackrel{?}{=} t'_p)x_1 + d(u_2, t_p \stackrel{?}{=} t'_p)x_2 + \cdots + d(u_n, t_p \stackrel{?}{=} t'_p)x_n &= 0. \end{aligned}$$

In our example, $\{u_1, \ldots, u_4\} = \{v, a, y, c\}$. The integer variables x_1, x_2, x_3 and x_4 are associated with v, a, y and c, respectively.

A solution of D is a vector $s = (d_1, \ldots, d_n)$ of $\mathbb{N}^n$ such that for $j \in [i, \ldots, p]$

$$\sum_{i=1}^{n} d(u_i, t_j \stackrel{?}{=} t'_j) d_i = 0.$$

We will denote by $s(i)$ the ith component d_i of s. In the example, D is reduced to the equation $2x_1 - x_2 - x_3 - x_4 = 0$, and the vector $s = (2, 1, 3, 0)$ is a solution of D.

A solution s of D is said to be *minimal* if there exists no nonzero solution strictly smaller than s for the Cartesian product ordering on $\mathbb{N}^n$. The solution $(2, 1, 3, 0)$ is not minimal because $(1, 1, 1, 0)$ is a smaller solution and is actually a minimal one.

Let $Sol(D)$ be the set of positive, non-null, minimal solutions of D. In our example, we have

$$Sol(D) = \{(1, 2, 0, 0), (1, 0, 2, 0), (1, 0, 0, 2), (1, 1, 1, 0), (1, 1, 0, 1), (1, 0, 1, 1)\}.$$

A variable x_i of D is *constrained* if it is associated with a constant u_i of $\{u_1, \ldots, u_n\}$. The constrained variables are x_2 and x_4.

$Sol(D)$ may be restricted to the solutions where all constrained variables have value at most 1 and where two different constrained variables are not both non-null. In the example, it becomes $Sol(D) = \{(1, 0, 2, 0), (1, 1, 1, 0), (1, 0, 1, 1)\}$.[6]

To each solution $s_j = (d_1, \ldots, d_n)$ of $Sol(D)$ we associate a term v_j which is

- a new variable if s_j has the value zero in each component corresponding to a constrained variable x_i;
- the constant c corresponding to the only constrained variable x_i such that $d_i = 1$ otherwise.

In the example, the new variable v_1 is associated with $s_1 = (1, 0, 2, 0)$, the constant a with $s_2 = (1, 1, 1, 0)$, and the constant c with $s_3 = (1, 0, 1, 1)$.

DEFINITION. A potential solution of P is a subset $p = \{s_1, \ldots, s_q\}$ of $Sol(D)$. A potential solution $\{s_1, \ldots, s_q\}$ is *suitable* if

- for all $i \in [1, \ldots, n]$, $\sum_{j=1}^{q} s_j(i) \neq 0$ (p is large enough); and
- for each constrained variable x_i, $\sum_{j=1}^{q} s_j(i) < 2$ (p is small enough).

The potential solution $\{s_1, s_3\}$ is not suitable because $s_1(2) + s_3(2) = 0$. The potential solution $\{s_1, s_2, s_3\}$ is suitable.

We associate the substitution

$$\{u_i \mapsto v_1^{s_1(i)} +_i \cdots +_i v_q^{s_q(i)} | u_i \in \mathscr{X},\ i \in [1, \ldots, n]\}$$

with every suitable potential solution $\{s_1, \ldots, s_q\}$.

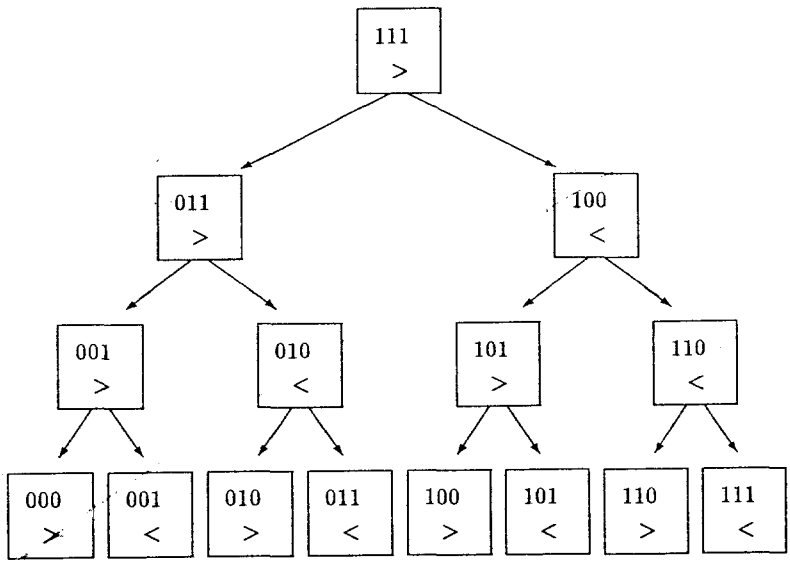
Then the substitutions associated with the suitable potential solutions form a complete set of unifiers of P (see [25, 30]). The suitable potential solution $\{s_1, s_2, s_3\}$ yields the solution

$$\{v \mapsto v_1 +_i a +_i c,\ y \mapsto v_1 +_i v_1 +_i a +_i c\}$$

of the original elementary AC unification problem.

From the implemental point of view, the major problem with elementary AC unification is that the number of potential solutions is $2^{|Sol(D)|}$. It is then necessary to efficiently extract from $2^{Sol(D)}$ the set of suitable potential solutions, that is, that are 'large enough' and 'small enough' at one time. For this purpose, we use the binary trees defined by Hullot [19] which extend the binomial trees of Vuillemin [31].

We recall the method on an example. Let $A = \{a_1, a_2, a_3\}$ be a set. Every subset of A is encoded by a bit vector representing its characteristic function. The bit vector (b_1, b_2, b_3) represents the subset $\{a_i | b_i = 1\}$. We then build the following tree:



This tree has the following three properties:

1. Every subtree rooted at a node, containing the subset p and the symbol $>$, contains only nodes whose associated subset q is smaller than or equal to p, with respect to the inclusion ordering.
2. Every subtree rooted at a node, containing the subset p and the symbol $<$, contains only nodes whose associated subset q is greater than or equal to p, with respect to the inclusion ordering.
3. Every leaf containing the subset p and the symbol $>$ (resp. $<$) has an ancestor containing the same subset p and the symbol $<$ (resp. $>$). The only exception is the leaf $((0,0,0), >)$ which is always small enough.

To determine which potential solutions are suitable (that is, which elements of $2^{Sol(D)}$ are large enough and small enough), a depth-first search of such a tree is made, testing whether the element at each node is large enough (if the node contains the symbol $>$) or small enough (if the node contains the symbol $<$). Thanks to the

Input: $P \equiv t_1 \stackrel{?}{=} t'_1 \wedge \cdots \wedge t_p = t'_p$, where $t_i, t'_i \in \mathscr{T}(\{+_i\} \cup \mathscr{C}, \mathscr{X})$.

1. Associate an integer variable x_i with each variable or constant u_i of P, x_i being *constrained* if u_i is a constant.
2. Compute the set $Sol(D)$ of minimal, non-negative, integral solutions of the system

$$\begin{array}{llll} d(u_1, t_1 \stackrel{?}{=} t'_1)x_1 + d(u_2, t_1 \stackrel{?}{=} t'_1)x_2 + \cdots + d(u_n, t_1 \stackrel{?}{=} t'_1)x_n & = 0 \\ \vdots & \vdots & \vdots & \vdots \\ d(u_1, t_p \stackrel{?}{=} t'_p)x_1 + d(u_2, t_p \stackrel{?}{=} t'_p)x_2 + \cdots + d(u_n, t_p \stackrel{?}{=} t'_p)x_n & = 0 \end{array}$$

of linear Diophantine equations.
3. Remove from $Sol(D)$ the solutions which have a value strictly greater than 1 for a constrained variable, or a non-zero value for two different constrained variables.
4. Associate with each s_i in $Sol(D)$ a term v_i that is (i) a new variable if s_i has the value 0 for every constrained variable, or (ii) the constant $c \in \mathscr{C}$ associated with the only constrained variable that has value 1 in s_i.
5. For every subset $\{s_1, \ldots, s_q\}$ of $Sol(D)$ such that (i) for all $i \in [1, \ldots, n]$, $\sum_{j=1}^{q} s_j(i) \neq 0$, and (ii) for each constrained variable x_i, $\sum_{j=1}^{q} s_j(i) < 2$, yield the solution $\{u_i \mapsto v_1^{s_1(i)} +_i \cdots +_i v_q^{s_q(i)} \mid u_i \in \mathscr{X}, i \in [1, \ldots, n]\}$ of P.

Fig. 3. Elementary *AC*-unification algorithm.

properties 1 and 2, the search of a subtree rooted at a node where the test failed is avoided. Thanks to the property 3, when a test succeeds on a leaf, the corresponding bit vector codes a suitable potential solution. This tree is obtained by merging two binomial trees [31], and the reader is referred to [19] for the construction of such trees.

6.3. FAST IMPLEMENTATION OF THE TESTS

It is natural to encode the subsets of a finite set by a bit vector. We are going to take advantage of the fact that bitwise logic operations are extremely fast on a computer.

Let D be the system of linear Diophantine equations associated with the problem P, and $Sol(D) = \{s_1, \ldots, s_q\}$ be the set of its positive minimal solutions. Let us associate with each variable $x_i \in \{x_1, \ldots, x_n\}$ of D, the vector of q bits $v_i = (b_1, \ldots, b_q)$ where $b_j = 1$ iff $s_j(i) \neq 0$. Then the subset p of $Sol(D)$, represented by the bit vector $(p_1, \ldots, p_q)$, is big enough if and only if $(p \,\&\, v_i) \neq (0, \ldots, 0)$ for all $i \in [1, \ldots, n]$, where & represents the bitwise logic operation *and*.

Similarly, we test whether the componentwise sum of a subset p of $Sol(D)$ is different from 0 on at most one component corresponding to a constrained variable. That is to say, for every constrained variable x_i, $p \,\&\, v_i$ has at most one non-null bit. A bit vector represented by an unsigned integer n has at most a non-null bit if n is 0 or a power of 2 or, equivalently, if $(n \,\&\, (n-1)) = 0$.

These remarks give an extremely fast method for enumerating the suitable potential solutions:

- Compute the set $\{e_1, \ldots, e_n\}$ of integers representing the bit vectors $v_1, \ldots, v_n$ associated with the variables $x_1, \ldots, x_n$ of D.

- Extract from this set the subset $\{c_1, \ldots, c_m\}$ of the integers associated with constrained variables.
- A subset p of $Sol(D)$ is large enough if for $i \in [1, \ldots, n]$, $p \,\&\, e_i \neq 0$.
- A subset p of $Sol(D)$ is small enough if for $i \in [1, \ldots, m]$, $(p \,\&\, c_i) \,\&\, ((p \,\&\, c_i) - 1) = 0$.
- Perform a depth-first search of Hullot's binary tree (with a stack), and test the subsets of $Sol(D)$, represented by unsigned integers, with the test 'large enough' (resp. 'small enough') if the node contains the symbol $>$ (resp. $<$).
- If a test fails at a node, do not search the subtree rooted at this node. If a test succeeds at a leaf containing the potential solution p, then p is suitable.

It is easy to generate Hullot's tree by using a stack whose items contain a test ($<$ or $>$), a bit vector (unsigned integer coding the subset to be tested), and an integer representing the height of the tree to be generated. Moreover, the sons of a node are easily computed with bitwise logic operations and shifting on unsigned integers. The height of the stack used for generating Hullot's binary tree is bounded by the number of solutions in $Sol(D)$ plus one; hence the required memory can be allocated in advance.

Note that for these tests to be really efficient, the length of the bit vectors is limited to the number of bits in the words of the machine. In our case, the machines have 32-bit words; therefore, we are limited to the problems having up to 32 solutions to their associated Diophantine system (that is, 2^{32} potential solutions).

6.4. CONSTRUCTION OF THE UNIFIERS

The equation $x +_i x +_i x = x_1 +_i x_2 +_i x_3 +_i x_4$, where $+_i$ is associative-commutative, has 1 044 569 most general unifiers. It is crucial not to spend too much time on building each solution. In the preceding section, we saw how to determine efficiently what the solutions are. We will now show how to build them fast. We keep developing our example: The following table gives the solutions associated with the two suitable potential solutions of $2^{Sol(D)}$.

If there are many most general solutions, the most expensive operation in

Potential solution		Associated substitution	
Subset	Code	Value of v	Value of y
$\{s_2, s_3\}$	$(0,1,1)$	$t_2 +_i t_3 \equiv a +_i c$	$t_2 +_i t_3 \equiv a +_i c$
$\{s_1, s_2, s_3\}$	$(1,1,1)$	$t_1 +_i t_2 +_i t_3 \equiv v_1 +_i a +_i c$	$t_1^2 +_i t_2 +_i t_3 \equiv v_1 +_i v_1 +_i a +_i c$

elementary AC unification is the construction of the solutions, or equivalently, the solved forms, from the suitable potential solutions. The method given above for the computation of the suitable potential solutions is so efficient that 95% of the time would be spent in memory allocation if the different solutions were constructed *ex nihilo*. To improve the efficiency, it is enough to build, in advance, all the subterms

$t_i^{s_i(j)}$ (that is, the terms $t_1 \equiv v_1, t_1^2 \equiv v_1 +_i v_1, t_2 \equiv a$ and $t_3 \equiv c$ in our example). Remember that the terms are implemented as pairs containing the top function symbol and the list of immediate subterms. If the addresses of the beginning and the end of the lists (v_1), (v_1, v_1), (a) and (c) are available, then the value of each variable can be constructed by only testing each bit of the corresponding suitable potential solution and moving some pointers: It is sufficient to compute for each variable x_i a list of triples containing for each solution s_j of $Sol(D)$, the corresponding unsigned integer 2^j, as well as the addresses of the beginning and the end of the list containing $s_j(i)$ occurrences of the term associated with s_j.

We can thus construct, for an elementary *AC* unification problem, a data structure containing all the useful information for computing and constructing all the solutions of the original problem. A function `next-sol` allows one to compute the next solution, if any, from such a data structure. After a call to `next-sol`, the stack for the generation of Hullot's tree is in such a state that the next call will provide a different solution. There is no need to compute all the solutions of an elementary *AC* unification problem at one time, which allows one to save space. Another advantage of this data structure is that it can easily be re-initialized by only re-initializing the stack. This makes it possible to enumerate several times the solutions of an elementary *AC* unification problem, while most of the computations are made only once.

We manage to compute the 1 044 569 most general unifiers of the equation $x +_i x +_i x \stackrel{?}{=} x_1 +_i x_2 +_i x_3 +_i x_4$ in less than 60 seconds on a Sun 4/330.

7. Branching and Failure

An implementation problem more directly concerning the combination algorithm given in Section 2.3 is how to handle the non-determinism introduced by the rule ***E*-Res**, when the subtheories are not unitary. This is the case of *AC* theories.

Applying ***E*-Res** to the subproblem P_i consists of replacing P_i by a solved form yielded by the function `next-sol`. Some other rules may then be applied (or ***E*-Res** itself to another subproblem), until a DAG solved form (which may be F in case of failure) is reached for the whole original problem P. After that, it is necessary to restore the state of the system and repeat the process with all the other most general solutions of P_i.

To make this backtracking possible, the most natural approach is to make a copy of the state of the problem before modifying it by applying the first solution of P_i, and restoring it before applying the second solution of P_i, and so on.

The copy of a complex data structure, involving variable sharing like ours, is expensive, and this operation may have to be repeated thousands or millions of times. A solution is to physically copy the whole state of the system, rather than copying a symbolic expression. To do this, we allocate a contiguous memory zone at the beginning, and all memory allocation (for pointers and for data) will be done in constant time (without any calls to the system). When non-deterministic

branching occurs, the part of the memory used so far is copied onto a stack, as a whole. C allows us to use a very fast routine for copying such a byte array. To restore the state of the system before the branching, it is sufficient to copy back the memory image located on top of the stack. The system will then be physically in the same state as before the branching. Memory images are saved in a stack because the branching may have an arbitrary depth. The rules that yield F essentially call a function, fail, that restores a memory image from the top of the stack.

Table I. Computation times for elementary AC problems.

		Lisp on Symbolics 36xx			C on Sun	
Problem	No. sol	Stickel	H. & S.	K. & Z.	Sun 4/330	Sun 3/260
$*(x,a,b) = *(u,c,d,e)$	2	0.018	0.009	0.012	ϵ	ϵ
$*(x,a,b) = *(u,c,c,d)$	2	0.011	0.010	0.011	ϵ	ϵ
$*(x,a,b) = *(u,c,c,c)$	2	0.008	0.010	0.008	ϵ	ϵ
$*(x,a,b) = *(u,v,c,d)$	12	0.047	0.031	0.031	ϵ	ϵ
$*(x,a,b) = *(u,v,c,c)$	12	0.032	0.030	0.029	ϵ	ϵ
$*(x,y,a) = *(u,u,v,w)$	300	0.622	0.766	0.507	0.017	0.050
$*(x,y,a) = *(u,u,v,v)$	216	0.347	0.473	0.314	0.017	0.017
$*(x,y,a) = *(u,u,u,c)$	92	0.166	0.197	0.140	ϵ	ϵ
$*(x,y,a) = *(u,u,u,v)$	196	0.323	0.464	0.276	ϵ	ϵ
$*(x,y,a) = *(u,u,u,u)$	124	0.163	0.291	0.160	ϵ	0.017
$*(x,y,z) = *(u,u,v,w)$	2,901	5.435	7.936	2.330	0.117	0.317
$*(x,y,z) = *(u,u,v,v)$	3,825	5.673	8.913	2.584	0.133	0.367
$*(x,y,z) = *(u,u,u,c)$	2,982	4.730	7.103	1.701	0.117	0.350
$*(x,y,z) = *(u,u,u,v)$	7,029	10.695	19.321	4.615	0.283	0.750
$*(x,y,z) = *(u,u,u,u)$	32,677	39.865	98.544	19.136	1.450	3.867
$*(x,x,a) = *(u,u,v,w)$	12	0.028	0.042	0.026	ϵ	ϵ
$*(x,x,a) = *(u,u,v,v)$	0	0.003	0.004	0.004	ϵ	ϵ
$*(x,x,a) = *(u,u,u,c)$	2	0.008	0.009	0.008	ϵ	ϵ
$*(x,x,a) = *(u,u,u,v)$	12	0.019	0.032	0.018	ϵ	ϵ
$*(x,x,a) = *(u,u,u,u)$	0	0.002	0.003	0.004	ϵ	ϵ
$*(x,x,y) = *(u,v,w,c)$	1,632	3.228	3.871	1.368	0.100	0.283
$*(x,x,y) = *(u,v,w,t)$	13,703	25.605	36.690	13.186	0.583	1.517
$*(x,x,y) = *(u,u,c,d)$	2	0.007	0.010	0.009	ϵ	ϵ
$*(x,x,y) = *(u,u,c,c)$	4	0.007	0.014	0.008	ϵ	ϵ
$*(x,x,y) = *(u,u,v,c)$	18	0.034	0.047	0.027	ϵ	ϵ
$*(x,x,x) = *(u,v,w,c)$	6,006	9.499	14.671	3.218	0.283	0.767
$*(x,x,x) = *(u,v,w,t)$	1,044,569	–	–	639.640	59.517	148.150
$*(x,x,x) = *(u,u,c,d)$	2	0.008	0.008	0.008	ϵ	ϵ
$*(x,x,x) = *(u,u,c,c)$	2	0.004	0.009	0.005	ϵ	ϵ
$*(x,x,x) = *(u,u,v,c)$	12	0.023	0.033	0.016	ϵ	0.017

This table gives some computation times for the implementations of the algorithms of Stickel, Herold and Siekmann, and Kapur and Zhang (in LISP), to be compared with those of our algorithm (in C). The computation times are given in seconds. The value ϵ means that the computation time is too small to be measured, the system answering 0 for durations under 17 milliseconds. The missing times (–) denote durations too long for the experiment to succeed. The data of the first three implementations are taken from [7].

8. Benchmarks

In the following, $*$, $+$, $\cdot$ denote AC operators, $f, g, \ldots, a, b, c, \ldots$ free function symbols and constants, and $x, y, z, u, v, w, t, x_1, \ldots$ variables. Table I presents some benchmarks taken from [7]. The table contains the elementary problems, the number of minimal solutions, the computation time for several implementations given in [7], and, in the last two columns, the computation time for our program on a Sun 4/330 and on a Sun 3/260. We have run the program several times for each problem, and we have kept the shortest time. The comparison of the execution times is not easy since the algorithms of Stickel, Herold–Siekmann, and Kapur–Zhang are implemented in LISP on LISP machines (Symbolics 36xx) with an instruction fetch unit.

Table II. Computation times for some non-elementary problems.

Problem	Number of solutions	Number of occur-checks	Time
$+(y_1, y_2) = +(y_3, y_4)$	7	0	ϵ
$*(x, x, x) = *(u, v, w, a)$	6,006	0	0.283
$f(+(y_1, y_2), *(x, x, x)) = f(+(y_3, y_4), *(u, v, w, a))$	42,042	0	2.667
$f(+(y_1, y_2), \cdot(z_1, z_2), *(x, x, x)) = f(+(y_3, y_4), \cdot(z_3, z_4), *(u, v, w, a))$	294,294	0	23.533
$*(x, x, x) = *(u, u, v, c)$	12	0	ϵ
$*(x, x, x) = *(u, u, v, y) \wedge y = f(z)$	12	13	ϵ
$*(x, x, x) = *(u, u, v, y) \wedge y = f(x)$	0	13	0.017
$*(x, x, y) = *(u, v, w, c)$	1,632	0	0.100
$*(x, x, y) = *(u, v, w, t) \wedge t = f(z)$	1,632	1,633	2.000
$*(x, x, y) = *(u, v, w, t) \wedge t = h(x, y)$	0	1,633	2.383
$*(x, x, x) = *(u, v, w, a)$	6,006	0	0.283
$*(x, x, x) = *(u, v, w, t) \wedge t = f(y)$	6,006	6,007	10.000
$*(x, x, x) = *(u, v, w, t) \wedge t = f(x)$	0	6,007	11.533
$*(x, y, z) = *(u, u, u, u)$	32,677	0	1.450
$*(x, y, z) = *(u, u, u, u) \wedge w = f(w)$	0	1	ϵ
$*(x, y, z) = *(u, u, u, u) \wedge w = f(v)$	32,677	32,678	31.283

This table allows the measurement of the part taken by the occur-checks (and backtrackings) in the computation times. If the problem involves non-constant free operators, an occur-check is done before solving the AC subproblems, and another occur-check each time a new AC solution is tested.

The last three examples suggest an optimization, which has not yet been implemented. When there are no *shared variables*, as defined in [5], there cannot be a compound cycle; hence the occur-check is not necessary (except at the beginning for the free theory). Taking this fact into account would reduce, by a factor of 20, the computation time for the last problem. Indeed, the last three problems have the same AC subproblem. In the first case, the whole problem is pure in the AC theory of $*$; therefore, occur-check and branching are not necessary, and 32 677 solutions are computed in 1.450 seconds. The second case is the same problem, with an additional cyclic equation $w \stackrel{?}{=} f(w)$, in the free theory. The program stops with failure in negligible time after a positive occur-check. Finally, the last problem still has the same AC equation, with an additional non-cyclic equation in the free theory. The program performs 32 678 occur-checks (and 32 677 context restorations) in 31 seconds. Since there are no shared variables, these occur-checks are useless, and the computation time could be the same as for the first case.

Table III. Compared cost of the various subroutines.

Pb.	No. sol.	Time	Mon.	Point.	Stack	Bit	Back.	O.-C.	Alloc.
1	1,044,569	59.517	4.7	93.1	1.1	0.0	0.0	0.0	0.0
2	6,006	0.283	22.7	59.0	9.1	4.5	0.0	0.0	0.0
3	42,042	2.667	9.7	79.4	1.0	6.2	0.0	0.0	0.0
4	294,294	23.533	6.9	78.9	1.9	8.7	0.0	0.0	0.0
5	6,006	10.000	21.5	2.8	0.0	0.1	28.8	6.4	6.7
6	32,677	1.450	11.5	84.7	0.0	1.0	0.0	0.0	0.0
7	32,677	31.283	14.9	4.2	0.0	0.0	59.5	1.4	0.0

This table shows the relative percentages of time spent by the program in the most important subroutines. The monitoring is obtained using the -p option of the compiler. The first column gives the problem number:

1: $*(x,x,x) = *(u,v,w,t)$
2: $*(x,x,x) = *(u,v,w,a)$
3: $f(+(y_1,y_2,*(x,x,x)) = f(+(y_3,y_4),*(u,v,w,a))$
4: $f(+(y_1,y_2),\cdot(z_1,z_2),*(x,x,x)) = f(+(y_3,y_4),\cdot(z_3,z_4),*(u,v,w,a))$
5: $*(x,x,x) = *(u,v,w,t) \wedge t = f(y)$
6: $*(x,y,z) = *(u,u,u,u)$
7: $*(x,y,z) = *(u,u,u,u) \wedge w = f(v)$

The second and third columns give the execution time on a Sun 4/330, and the number of solutions, respectively. The other columns give the percentage of time spent in the most important subroutines:

Mon.: the monitoring itself.
Point.: moving the pointers to set a new solution.
Stack: the generation of Hullot's tree (generated with a stack).
Bit: the tests on the potential solutions (bitwise logic operations).
Back.: the backtracking (copying byte arrays).
O.-C.: the occur-check.
Alloc.: the memory allocation.

In Table II we also give some benchmarks for nonelementary unification problems. The execution time is measured on a Sun 4/330. If the different AC subproblems have no variables in common, then the occur-check is not necessary. Moreover, it is possible to combine several instances of the data structure for enumerating the solutions of an elementary problem. In this case, branching is not needed either. The efficiency of the program in such cases is very good. Table II also shows that most of the time may be spent in branching and occur-checks. Finally, Table III shows, for several problems, the relative cost of the various subroutines.

9. Conclusion

We have proved that it is possible to write an efficient implementation of AC-unification in C. The main advantages of C are the very good optimizations done by the compilers and the possibility to go deep enough into the internal representations of the data. On the other hand, C is not a very nice language for symbolic computation; hence we suggest that it should be used for implementing very

frequently used high-level primitives, for larger systems written in more advanced languages.

The data structure used in this program allows extension to theories other than *AC*. The combination part needs no modifications for adding other regular collapse-free subtheories: It is enough to have a unification algorithm for each of them.

The problem of combining arbitrary theories has been solved by Schmidt–Schauß [29], who gave a highly non-deterministic algorithm. In [4, 3], an algorithm is given that naturally extends the one implemented here. Adding non-regular or collapsing theories like *AC*1, *ACI*, *AC*0, and Boolean rings requires adaptation of the combination algorithm at two levels:

- The rules **Conflict1** and **Conflict2** are no longer complete, and theory clashes should be solved by using matching.
- The occur-check should not yield *F*, but compute the cycles that may be solved by using either matching or variable elimination.

For arbitrary theories, it is required to have besides the unification algorithm, a unification algorithm with free constants (i.e., a matching algorithm [6]), as well as a constant elimination algorithm.

Acknowledgements

I thank Quing Bin Pan, who taught me the basic notions of *C* and helped me to implement the data structure. Many thanks also to Judith Jakoubovitch, who implemented the routines for memory allocation and context saving and restoring, and who gave me the magic formula for the Boolean test 'small enough'. Pierre Dauchy showed me several times that it was not the machine that was faulty during nice debugging sessions. Last but not least, thanks to Jean-Pierre Jouannaud, who *really* wanted me to compete for the *AC* unification race.

Notes

[1] We make no distinction between the equations $s \stackrel{?}{=} t$ and $t \stackrel{?}{=} s$, which is consistent with the semantics of the predicate $\stackrel{?}{=}$.

[2] One may also require that all the variables in $t_1, \ldots, t_n$ are in $\{y_1, \ldots, y_p\}$. This is the so-called *protection of variables*.

[3] It is, for instance, the case for $f(x_1, x_2, \ldots, x_n) \stackrel{?}{=} f(h(x_2, x_2), h(x_3, x_3), \ldots, h(x_{n+1}, x_{n+1}))$.

[4] Stickel's algorithm goes back to '75 and Fages's proof to '84.

[5] That is, such that $\mathscr{D}(\mathbf{u}_j) \cdot \mathscr{D}(e_a) < 0$, where $\cdot$ is the usual scalar product in $\mathbb{Z}^n$.

[6] To improve the efficiency, the value of an integer variable corresponding to a variable having a value from another theory may be bounded by 1.

References

1. Adi, M. and Kirchner, C., 'Associative-commutative unification: The system solving approach', in Alfonso Miola (Ed.), *Proc. DISCO'90. LNCS 429*, Capri, Italy, April 1990. Springer-Verlag.
2. Birkhoff, G., 'On the structure of abstract algebras', *Proc. Cambridge Phil. Society*, **31** (1935).

3. Boudet, A., *Unification dans les mélanges de théories equationnelles.* Ph.D. thesis, Université de Paris-Sud, Orsay (February 1990).
4. Boudet, A., 'Unification in combinations of equational theories: An efficient algorithm, in *Proc. 10th Int. Conf. on Automated Deduction*, Kaiserslautern. Springer-Verlag, July 1990.
5. Boudet, A., Contejean, E. and Devie, H., 'A new *AC* unification algorithm with a new algorithm for solving diophantine equations', in *Proc. 5th IEEE Symp. Logic in Computer Science*, Philadelphia, June 1990.
6. Bürckert, H.-J., 'Matching, a special case of unification?' *J. Symbolic Computation*, **8**, 523–536 (1989).
7. Bürckert, H.-J., Hérold, A., Kapur, D., Siekmann, J. H., Stickel, M. E., Tepp, M. and Zhang, H., 'Opening the *AC*-unification race', *J. Automated Reasoning*, **4**(4), 465–474 (1988).
8. Christian, J. and Lincoln, P., 'Adventures in associative-commutative unification (a summary)', in *Proc. 9th Conf. on Automated Deduction*, Argonne, *LNCS* 310. Springer-Verlag (May 1988).
9. Clausen, M. and Fortenbacher, A., 'Efficient solution of linear Diophantine equations', Research Report 32/87, Univ. Karlsruhe, November 1987.
10. Comon, H., *Unification et disunification: Théorie et applications*, Thèse de Doctorat, I.N.P. de Grenoble, France (1988).
11. Contejean, E. and Devie, H., 'Résolution de systèmes linéaires d'équations Diophantiennes', *Comptes Rendus de l'Académie des Sciences, Paris*, **313**(1), 115–120 (1991).
12. Corbin, J. and Bidoit, M., 'A rehabilitation of Robinson's unification algorithm', *Information Processing Letters* 1983.
13. Dershowitz, N. and Jouannaud, J.-P. *Rewrite systems*, in J. van Leeuwen (ed), *Handbook of Theoretical Computer Science*, Vol. B, North-Holland (1990).
14. Fages, F., 'Associative-commutative unification', *J. Symbolic Computation*, **3**(3) (1987).
15. Fortenbacher, A., 'An algebraic approach to unification under associativity and commutativity', in *Proc. Rewriting Techniques and Applications 85*, Dijon, *LNCS* 202. Springer-Verlag, May 1985.
16. Guckenbiehl, Th. and Hérold, A., 'Solving linear Diophantine equations', Technical Report 85-IV-KL, SEKI, University of Kaiserslautern, Germany, 1985.
17. Herold, A., *Combination of unification algorithms in equational theories.* Ph.D. thesis, Universität Kaiserslautern, Kaiserslautern, Germany, 1987.
18. Huet, G., 'An algorithm to generate the basis of solutions to homogeneous linear diophantine equations', *Information Processing Letters*, **7**(3), April 1978.
19. Hullot, J.-M., 'Associative commutative pattern matching', in *Proc. 6th IJCAI* (Vol. I), Tokyo, pp. 406–412, August 1979.
20. Jouannaud, J.-P. and Kirchner, C., in J.-L. Lassez and G. Plotkin (eds), *Alan Robinson's Anniversary Book*, 'Solving Equations in Abstract algebras: A rule-based survey', MIT Press, 1990 (to appear).
21. Kapur, D. and Zhang, H., 'Rll: A rewrite rule laboratory', in *Proc. 9th Conf. on Automated Deduction*, Argonne, *LNCS* 310, pp. 768–769. Springer-Verlag, 1988.
22. Kirchner, C., *Méthodes et outils de conception systématique d'algorithmes d'unification dans les théories equationnelles.* Thèse d'Etat, Univ. Nancy, France, 1985.
23. Lambert, J. L., 'Une borne pour les générateurs des solutions entières positives d'une équation Diophantienne linéaire', *Comptes Rendus de l'Académie des Sciences de Paris*, **305** (1987). Série I.
24. Lankford, D., 'New non-negative integer basis algorithms for linear equations with integer coefficients', Unpublished manuscript, 1987.
25. Livesey, M. and Siekmann, J., 'Unification of bags and sets', Research report, Institut fur Informatik I, Univ. Karlsruhe, West Germany, 1976.
26. Martelli, A. and Montanari, U., 'An efficient unification algorithm', *ACM Transactions on Programming Languages and Systems*, **4**(2), 258–282 (1982).
27. Plotkin, G., 'Building in equational theories', in *Marchine Intelligence* 7, Edinburgh Univ. Press, 1972.
28. Robinson, J. A., 'A machine-oriented logic based on the resolution principle', *J. ACM*, **12**(1), 23–41 (1965).
29. Schmidt-Schauß, M., 'Unification in a combination of arbitrary disjoint equational theories', in *Proc. 9th Conf. on Automated Deduction*, Argonne, *LNCS* 310, Springer-Verlag, May 1988.
30. Stickel, M., 'A unification algorithm for associative-commutative functions', *J. ACM*, **28**(3), 423–434 (1981).
31. Vuillemin, J., 'A data structure for manipulating priority queues', *J. ACM*, **21**(4) (1978).