Optimisation Techniques for Combining Constraint Solvers

Stephan Kepser
CIS, Universität München
Oettingenstr. 67
80538 München, Germany
kepser@cis.uni-muenchen.de

Jörn Richts
Theoretische Informatik
RWTH Aachen
52056 Aachen, Germany
richts@informatik.rwth-aachen.de

Abstract

In recent years, techniques that had been developed for the combination of unification algorithms for equational theories were extended to combining constraint solvers. These techniques inherited an old deficit that was already present in the combination of equational theories which makes them rather unsuitable for practical use: The underlying combination algorithms are highly non-deterministic. This paper is concerned with the practical problem of how to optimise the combination method of Baader and Schulz. We present an optimisation method, called the *deductive* method, which uses specific algorithms for the components to reach certain decisions deterministically. We also give a strategy how to select an order of non-deterministic decisions. Run time tests of our implementation indicate that the optimised combination method yields combined decision procedures that are efficient enough to be used in practice.

1 Introduction

One idea behind constraint solving is to use specialised formalisms and inference mechanisms to solve domain-specific tasks. In many applications, however, one is faced with a complex combination of different problems. Therefore constraint solvers tailored to solving a single problem can only be applied, if it is possible to combine them with others. Concrete examples of the combination of constraint solvers can be found, e.g., in [9, 8]. In a recent paper [3], Baader and Schulz present a general method for the combination of constraint systems. Their method is applicable to a large class of structures, the so-called quasi-free structures. Quasi-free structures comprise many important infinite non-numerical solution domains such as (quotient) term algebras [13], rational tree algebras [7], vector spaces, hereditarily finite wellfounded and non-wellfounded lists, sets [1] and multi sets as well as certain types of feature structures [16]. The combined solution domain the authors present in [3], the so called free

*This work was funded by the "Schwerpunkt Deduktion" of the *Deutsche Forschungsgemeinschaft* (DFG) and was supported by the Esprit working group 22457 – CCL II of the European Union.

amalgamated product, has the characterising property of being the most general combination in the sense that every other combined domain contains a homomorphic image of it.

The question of how to combine specialised methods was first discussed in the field of unification theory (see [4] for an overview). Equational unification algorithms, which can be seen as an instance of constraint solvers, were built into resolution-based theorem provers and rewriting engines to improve their handling of equality. Since the unification problems occurring in these applications usually contain function symbols from various equational theories, the question of how to combine equational unification algorithms became important. For algorithms that compute complete sets of unifiers for equational theories over disjoint signatures, this problem was solved by Schmidt-Schauß [14] and Boudet [5]. With the development of constraint-based approaches to theorem proving and rewriting, the interest in combining unification algorithms extended towards combinations of decision procedures, for which Baader and Schulz [2] finally presented a general algorithm.

As a generalisation of the one given in [2], the algorithm for combining constraint solvers in [3] inherits the old weakness of being so highly non-deterministic that it is of very limited practical use. The aim of this paper is to provide optimisation techniques for the combination algorithm by Baader and Schulz that make the combination of constraint solvers practically usable, that are yet general enough to be applicable to a large class of constraint solvers. The method we propose is the so called deductive method which is based on the insight that many decisions of the combination algorithm do not really need to be made non-deterministically, but can rather be deduced on the base of the constraint domains involved, the input problem and other decisions made earlier. In our deductive combination method the component solvers are consulted to gain information on what further steps can be made deterministically. This obviously requires component solvers capable of doing so. The strength of this combination method lies in the interchange of information between the component algorithms. The impact of this interchange is highlighted by the fact that, although developed for the more general case, our combination algorithm turns out to be an implementation of the *PTIME* combination algorithm given in [15, 12] for a special subclass of constraint solvers. We also present a selection strategy for choosing the next non-deterministic decision. In order to detect unsolvability of a single component faster, we first make all non-deterministic decisions relative to one component before we proceed to the next one. The run time tests we present in this paper show the enormous effect of our optimisation methods making us confident that combination of constraint solvers is feasible in practice.

In this paper, we present our combination method as an algorithm for combining constraint solvers, but our optimisation techniques are nevertheless useful for the special case of equational unification. Moreover our method can be directly extended to compute complete sets of unifiers.

2 Preliminaries

Quasi-free Structures and the Free Amalgamated Product

A signature Σ consists of a set Σ_F of function symbols and a disjoint set Σ_P of predicate symbols (not containing "="), each of fixed arity. Σ-structures over the carrier set A are denoted by $\mathfrak{A}^\Sigma$. Σ-terms $(t, t_1, \ldots)$ and atomic Σ-formulae (of the form $t_1 = t_2$, or of the form $p(t_1, \ldots, t_n)$) are built as usual from Σ and a countable set of variables $\mathcal{V}$. A Σ-formula φ is written in the form $\varphi(v_1, \ldots, v_n)$ in order to indicate that the set $Var(\varphi)$ of free variables of φ is a subset of $\{v_1, \ldots, v_n\}$. A mapping $\sigma : \mathcal{V} \to A$ from the set of variables to the carrier set of $\mathfrak{A}^\Sigma$ is called an *assignment*. A *constraint problem over signature* Σ is a set of atomic Σ-formulae. An assignment σ is a *solution* for a constraint problem Γ in $\mathfrak{A}^\Sigma$ iff $\varphi(\sigma(v_1), \ldots, \sigma(v_n))$ becomes true in $\mathfrak{A}^\Sigma$ for all formulae $\varphi(v_1, \ldots, v_n) \in \Gamma$.

Σ-homomorphisms and Σ-endomorphisms are defined as usual, see e.g., [13]. With $End_{\mathfrak{A}^\Sigma}$ we denote the monoid of all endomorphisms of $\mathfrak{A}^\Sigma$, with composition as operation.

We will now introduce the solution domains for constraint solving we consider here, namely quasi-free structures. Quasi-free structures, a generalisation of free structures, were introduced by Baader and Schulz [3]. We consider a fixed Σ-structure $\mathfrak{A}^\Sigma$.

Let A_0, A_1 be subsets of $\mathfrak{A}^\Sigma$. Then A_0 *stabilises* A_1 iff all elements m_1 and m_2 of $End_{\mathfrak{A}^\Sigma}$ that coincide on A_0 also coincide on A_1. For $A_0 \subseteq A$ the *stable hull* of A_0 is the set $\mathrm{SH}^{\mathfrak{A}}(A_0) := \{a \in A \mid A_0 \text{ stabilises } \{a\}\}$.

$\mathrm{SH}^{\mathfrak{A}}(A_0)$ is always a Σ-substructure of $\mathfrak{A}^\Sigma$, and $A_0 \subseteq \mathrm{SH}^{\mathfrak{A}}(A_0)$. The stable hull of A_0 can be larger than the Σ-subalgebra generated by A_0.

The set $X \subseteq A$ is an *atom set* for $\mathfrak{A}^\Sigma$ if every mapping $X \to A$ can be extended to an endomorphism.

Definition 2.1 A countably infinite Σ-structure $\mathfrak{A}^\Sigma$ is a *quasi-free structure* iff $\mathfrak{A}^\Sigma$ has an infinite atom set X where every $a \in A$ is stabilised by a finite subset of X. We denote this quasi-free structure by $(\mathfrak{A}^\Sigma, X)$.

The class of quasi-free structures contains many important non-numerical infinite solution domains. For example, all free structures (see, e.g., [13]), rational tree algebras ([7]), feature structures with arity ([16]), domains with nested, finite or rational lists (rational lists are used in Prolog III, see [8]), and domains with nested, finite or rational sets ([1]). In each case, we have to take the non-ground variant since we assume the existence of a countably infinite set of atoms. For details we refer to [3].

A fundamental property of quasi-free structures is the following: for each $a \in A$ there exists a *unique minimal* finite set $Y \subseteq X$ such that $a \in \mathrm{SH}^{\mathfrak{A}}(Y)$. The *stabiliser* of $a \in A$, $\mathrm{Stab}^{\mathfrak{A}}(a)$, is the unique minimal finite subset Y of X such that $a \in \mathrm{SH}^{\mathfrak{A}}(Y)$. The stabiliser of $A' \subseteq A$ is the set $\mathrm{Stab}^{\mathfrak{A}}(A') := \bigcup_{a \in A'} \mathrm{Stab}^{\mathfrak{A}}(a)$.

We extend the notions *regular* and *collapse-free*, known from equational unification, to quasi-free structures.

Definition 2.2 A quasi-free structure $(\mathfrak{A}^\Sigma, X)$ is called *collapse-free*, iff every endomorphism maps non-atoms to non-atoms, i.e., $m(a) \in A \setminus X$ for all $m \in$

$End_{\mathfrak{A}^\Sigma}$ and all $a \in A \setminus X$. The quasi-free structure $(\mathfrak{A}^\Sigma, X)$ is *regular*, iff for all $m \in End_{\mathfrak{A}^\Sigma}$ and all $a \in A : \mathrm{Stab}^{\mathfrak{A}}(m(a)) = \mathrm{Stab}^{\mathfrak{A}}(m(\mathrm{Stab}^{\mathfrak{A}}(a)))$.

In [3], Baader and Schulz present a combined solution domain of two or more quasi-free structures, the so-called free amalgamated product, which is characterised amongst all considerable combined solution domains as being the most general in the sense that every domain contains a homomorphic image of it. The authors also provide a construction method to obtain the free amalgamated product of arbitrary quasi-free structures. If $(\mathfrak{A}_1^{\Sigma_1}, X), \ldots, (\mathfrak{A}_n^{\Sigma_n}, X)$ are n quasi-free structures over disjoint signatures, we write $\mathfrak{A}_1^{\Sigma_1} \otimes \ldots \otimes \mathfrak{A}_n^{\Sigma_n}$ for their free amalgamated product. If the quasi-free structures one combines are quotient term algebras of equational theories over disjoint signatures, then their free amalgamated product is the quotient term algebra of the theory over the union of the axiom sets.

In this paper, we investigate "mixed" constraint problems. For $i = 1, \ldots, n$ ($n \geq 2$), let Σ_i be pairwise disjoint signatures and let $(\mathfrak{A}_i^{\Sigma_i}, X)$ be a quasi-free structure over signature Σ_i. A "mixed" constraint problem is a conjunction of atomic formulae over the joined signature $\Sigma_1 \cup \ldots \cup \Sigma_n$. A constraint problem Γ is in *decomposed form*, if Γ has the form $\bigcup_{i=1}^n \Gamma_i$ where each Γ_i is a pure constraint problem over the signature Σ_i. Any constraint problem Γ can be transformed into a constraint problem in decomposed form that is solvable, iff the original problem is solvable, by a simple deterministic preprocessing step (variable abstraction, see [2]). In the following, we will therefore always assume that a constraint problem is in decomposed form $\bigcup_{i=1}^n \Gamma_i$.

Only variables occurring in more than one component system Γ_i have to be considered by the combination algorithm. Hence we define the set of *shared variables* $\mathcal{U} := \{x \mid \exists i, j : i \neq j, x \in Var(\Gamma_i) \cap Var(\Gamma_j)\}$. The combination algorithm presented in the next section imposes some restrictions on the shared variables in order to prevent conflicts between the solutions of the component structures (like a variable being assigned to different elements by solutions of different structures). The solutions of the component problems Γ_i have to obey these so-called *linear constant restrictions*.

Definition 2.3 A *linear constant restriction* $L = (\Pi, Lab, <_L)$ for variables $\mathcal{U}$ consists of a partition[1] Π of $\mathcal{U}$, a labelling function $Lab : \mathcal{U}/_\Pi \to \{\Sigma_1, \ldots, \Sigma_n\}$ and a linear order $<_L$ on $\mathcal{U}/_\Pi$. We use $Lab(x)$ and $x <_L y$ instead of $Lab([x]_\Pi)$ and $[x]_\Pi <_L [y]_\Pi$.
An assignment σ is a *solution* for the *constraint problem with linear constant restrictions* (Γ_i, L) in $(\mathfrak{A}_i^{\Sigma_i}, X)$, iff it is a solution for Γ_i and for each $x, y \in \mathcal{U}$:

- $\sigma(x) = \sigma(y)$ if $x \equiv_\Pi y$,
- $\sigma(x) \in X$ if $Lab(x) \neq \Sigma_i$, and
- $\sigma(x) \notin \mathrm{Stab}^{\mathfrak{A}}(\sigma(y))$ if $Lab(x) \neq \Sigma_i, Lab(y) = \Sigma_i, y <_L x$.

By item two, all variables that receive a label different from Σ_i are treated as constants by σ. By item three, the use of these constants in σ is further restricted in order to prevent cycles. Two linear constant restrictions L_1 and L_2 over $\mathcal{U}$ are called *equivalent*, if they have identical partitions and labelling

[1] The equivalence relation induced by Π is denoted by $\equiv_\Pi$, $[x]_\Pi$ is the equivalence class of a variable x, and $\mathcal{U}/_\Pi$ is the set of all equivalence classes of variables in $\mathcal{U}$.

functions and their orders differ at most in ordering variables of identical label. This definition induces an equivalence relation on all linear constant restrictions for a given set of variables $\mathcal{U}$. If L_1 and L_2 are equivalent and an assignment σ solves (Γ, L_1), then σ also solves (Γ, L_2).

The Original Combination Algorithm

In the following we describe the combination algorithm given by Baader and Schulz in [3], where one can find the details. Here we give a straightforward generalisation of this algorithm to the case where more than two structures are combined. Additionally, we include basic optimisations similar to those described in [2].

Let Γ be a constraint problem in decomposed form. We assume the constraints in Γ are connected by shared variables, i.e., there is no partition $\Gamma = \Gamma' \cup \Gamma''$ where Γ' and Γ'' do not have variables in common. Otherwise Γ' and Γ'' can be solved separately. The algorithm consists of three non-deterministic steps which result in a linear constant restriction for the constraint problem.
Step 1: Variable identification Non-deterministically choose a partitioning Π of $\mathcal{U}$.
Step 2: Labelling Non-deterministically choose a labelling function $Lab : \mathcal{U}/_\Pi \to \{\Sigma_1, \ldots, \Sigma_n\}$.
Step 3: Ordering Non-deterministically choose a linear order $<_L$ on $\mathcal{U}/_\Pi$.
$L = (\Pi, Lab, <_L)$ constitutes a linear constant restriction. Note that for each equivalence class of linear constant restrictions, it suffices to choose just one member. The output tuple determined by these three steps is $((\Gamma_1, L), \ldots, (\Gamma_n, L))$.

Theorem 2.4 *The input problem Γ has a solution in the free amalgamated product $\mathfrak{A}_1^{\Sigma_1} \otimes \ldots \otimes \mathfrak{A}_n^{\Sigma_n}$, if and only if there exists an output tuple $((\Gamma_1, L), \ldots, (\Gamma_n, L))$ such that for each $i = 1, \ldots, n$, the constraint problem with linear constant restriction (Γ_i, L) has a solution in $\mathfrak{A}_i^{\Sigma_i}$.*

Decision Sets

The original algorithm makes all non-deterministic decisions first, and only thereafter it calls the component algorithms to determine whether the input problem with the thus chosen constant restriction is solvable. Our optimisations interleave these two parts. Hence we have to deal with linear constant restrictions which are only partially specified, i.e., restrictions representing the choices already made but making no statements about the decisions still open. In order to describe these partial constant restrictions and to have a framework for describing our optimisations on a formal level we introduce the notion of decision sets. A decision describes a single non-deterministic choice. There exist five different types of decisions.

Definition 2.5 Let $\mathcal{U}$ be the set of variables. A *decision* is an expression of the form $x \doteq y$, $x \not\doteq y$, $x \dot{\leq} y$, $x \dot{\mapsto} \Sigma_i$, or $x \dot{\not\mapsto} \Sigma_i$, where $x, y \in \mathcal{U}$ and $1 \leq i \leq n$. The decision $x \dot{<} y$ is used as an abbreviation for $x \dot{\leq} y, x \not\doteq y$.

We speak about sets of decisions (for a set of variables $\mathcal{U}$) which are—as usual—read conjunctively. In order to represent the two options when making a non-deterministic choice, we define the negation of a decision.

Definition 2.6 Let d be a decision. Its *negation* $\neg d$ is defined as follows:

$$\begin{aligned} \neg x \doteq y &:= x \not\doteq y, & \neg x \not\doteq y &:= x \doteq y, \\ \neg x \dot{\mapsto} \Sigma_j &:= x \not\dot{\mapsto} \Sigma_j, & \neg x \not\dot{\mapsto} \Sigma_j &:= x \dot{\mapsto} \Sigma_j, \\ \neg x \dot{\leq} y &:= y \dot{<} x. \end{aligned}$$

These rules of negation reflect the three non-deterministic steps of the algorithm: Two variables have to be identified or treated as different variables; each variable has to be treated as a variable or like a constant in a particular component system; and two variables with distinct labels have to be ordered in one way or the other. In the following we formally define this correspondence between sets of decisions and linear constant restrictions.

Definition 2.7 Let $\mathcal{U}$ be a set of variables. A linear constant restriction $L = (\Pi, Lab, <_L)$ over $\mathcal{U}$ *satisfies* a decision set D, if the following holds:

$$\begin{aligned} x \equiv_\Pi y \text{ if } x \doteq y \in D, & \qquad x \not\equiv_\Pi y \text{ if } x \not\doteq y \in D, \\ Lab(x) = \Sigma_i \text{ if } x \dot{\mapsto} \Sigma_i \in D, & \qquad Lab(x) \neq \Sigma_i \text{ if } x \not\dot{\mapsto} \Sigma_i \in D, \\ x <_L y \text{ or } x \equiv_\Pi y \text{ if } x \dot{\leq} y \in D. & \end{aligned}$$

The set of linear constant restrictions satisfying D is denoted by $\mathcal{L}(D)$. A set D is called *inconsistent* if $\mathcal{L}(D) = \emptyset$.

So, the decisions are interpreted by a linear constant restriction in a straightforward way. We can now use decision sets to represent constraint problems with partially specified linear constant restrictions.

Definition 2.8 A *constraint problem with decision set* (Γ, D) consists of a constraint problem Γ together with a set of decisions D. An assignment σ is a *solution* of (Γ, D) if σ is a solution of (Γ, L) for some $L \in \mathcal{L}(D)$.

Since decision sets represent linear constant restrictions, they inherit some properties like $\dot{<}$ representing an ordering. This is reflected by the following definition.

Definition 2.9 A decision set D is called *closed* if $D = \{d \mid \text{ every } L \in \mathcal{L}(D) \text{ satisfies } \{d\}\}$.

This definition implies that for each decision set D there is exactly one closed set which is equivalent to D; this set is called the *closure* of D. This closure can be computed efficiently; one has to consider that $\doteq$ denotes a congruence, $\dot{<}$ stands for an ordering, and $x \dot{\mapsto} \Sigma_i$ represents a functional relation. For example, a closure always contains $x \doteq x$ for all variables $x \in \mathcal{U}$, the two decisions $x \doteq y \in D$ and $y \dot{<} z \in D$ imply that $x \dot{<} z$ is in the closure of D, and the closure of $\{x \dot{\mapsto} \Sigma_i\}$ contains $x \not\dot{\mapsto} \Sigma_j$ for all $i \neq j$. In the following we will always assume that sets of decisions are closed, i.e., when adding decisions to a set we assume that the closure is formed immediately.

We need a criterion when a set of decisions already represents one linear constant restriction, i.e., when no more decisions have to be made.

Definition 2.10 A set of decisions D is *complete*, if all linear constant restriction in $\mathcal{L}(D)$ are equivalent.

From this definition and the one above it follows that there is a one-to-one correspondence between the equivalence classes of linear constant restrictions over $\mathcal{U}$ and closed and complete sets of decisions for $\mathcal{U}$. In order to test inconsistency and completeness of decision sets by an algorithm, we need a syntactic formulation of these properties. This is provided by the following lemma.

Lemma 2.11

1. *A closed set of decisions D is inconsistent iff $d \in D$ and $\neg d \in D$ for some decision d.*
2. *A closed and consistent set of decisions D (for variables $\mathcal{U}$) is complete iff for all $x, y \in \mathcal{U}$*
   either $x \doteq y \in D$ *or* $x \not\doteq y \in D$, *and*
   either $x \dot{<} y \in D$ *or* $y \dot{<} x \in D$ *if* $x \dot{\mapsto} \Sigma_i, y \not\dot{\Rightarrow} \Sigma_i \in D$, *and* $x \dot{\mapsto} \Sigma_i \in D$ *for one* Σ_i.

3 Deductive Method

In this section we will show how information deduced from the component systems and their individual structures can be used to prune the search space. The power of the method lies in the interchange of this information between the components.

Interchanging Decisions

A severe disadvantage of the original combination algorithm is that all non-deterministic decisions are made blindfoldedly without respecting the requirements that the component structures may impose. For example, if a component structure $\mathfrak{A}_i$ is collapse-free and the problem contains an equation $x = f(\ldots y \ldots)$ where $f \in \Sigma_i$, then x must receive label Σ_i. If $\mathfrak{A}_i$ is also regular then the problem is unsolvable if $y \not\dot{\Rightarrow} \Sigma_i \in D$ and $x \dot{<} y \in D$. Hence the algorithm can choose $x \dot{\mapsto} \Sigma_i \in D$ deterministically and take into account that $y \not\dot{\Rightarrow} \Sigma_i \in D$ implies $y \dot{<} x \in D$.

As the example shows, some decisions that have been deduced earlier in one component can be used to deduce new decisions in another one. This possible interplay between different structures suggests to use a method where component algorithms computing new decisions are called alternately in the beginning of the combination algorithm and whenever a non-deterministic choice has been made: Starting with some initial decisions, each component algorithm computes new decisions; these new decisions are added to the current set of decisions, which is used when calling the other component algorithms. When this process comes to an end because no new decisions can be deduced, the next non-deterministic choice has to be made by the combination algorithm. After this choice the process of computing new consequences can be started again. At any step of computing the consequences, a component algorithm may return that its subproblem has become unsolvable with the current set of decisions. Thereby, unsolvable branches of the search tree can be detected earlier.

Obviously, this method requires new component algorithms that are capable of computing consequences implied by the component structures, the problem,

and the decisions computed so far. A structure for which such an algorithm does not exist can still be used in this method, but it cannot contribute to the deductive process. It is clearly the quality of the deductive component algorithms that decides the amount of optimisation achieved. The optimisations of our component algorithms go quite beyond using only syntactic properties of structures as in the example above. The goal is to deduce as much information as is possible with a reasonable effort.

The Algorithm

First we define the task of the new deductive component algorithms. Their input is a pure constraint problem and a set of decisions which need not be complete. The result is a set of decisions that follows from the constraint problem and the input decisions. If the input is unsolvable, the result may also be an inconsistent set of decisions.

Definition 3.1 Let (Γ, D) be a constraint problem with decision set. The decision set C is a *consequence* of (Γ, D), iff C is contained in every complete decision set $D' \supseteq D$ with (Γ, D') is solvable, i.e.,

$$C \subseteq \bigcap\{D' \mid D \subseteq D', D' \text{ is complete, and } (\Gamma, D') \text{ is solvable}\}.^2$$

Note that $C = \emptyset$ is always a consequence and that the consequence need not be inconsistent if (Γ, D') is unsolvable for all complete extensions D' of D. Therefore, the standard algorithms for constraint solving with linear constant restrictions must be called in the end when a complete set of decisions is reached. See Section 4 for a discussion on how deductive component algorithms co-operate with standard ones.

Now we can describe the combination algorithm. The termination condition in case of success is that the set of decisions is complete, as given in Lemma 2.11. In the following, D denotes the current set of decisions, initialised with $D := \emptyset$.

Repeat
  Deduce consequences:
  Repeat
    For each system i,
      call the component algorithm of system i to calculate
      new consequences C of (Γ_i, D),
      set the new current set of decisions $D := D \cup C$.
  Until D is inconsistent or no component algorithm computes new decisions.
  If D is consistent and not complete
    Select next choice:
    Select a decision $d \notin D$ such that $D \cup \{d\}$ is consistent.
    Non-deterministically choose either
    $D := D \cup \{d\}$ or
    $D := D \cup \{\neg d\}$.
Until D is inconsistent or complete.
Return D.

[2] $\bigcap\{\}$ is the (inconsistent) set of all decisions over $\mathcal{U}$.

Proposition 3.2 *The input problem Γ is solvable, iff the algorithm computes a consistent set D such that for each $i = 1, \ldots, n$ the constraint problem with decision set (Γ_i, D) is solvable.*

Again, testing (Γ_i, D) for solvability can be performed by the component algorithms used in the original combination algorithm. Since a consequence is a decision that is contained in every solvable complete decision set, it is clear that we prune those branches of the search space that are unsolvable. Hence correctness of the algorithm is an immediate consequence of the correctness of the original combination algorithm in Theorem 2.4.

The deductive method additionally allows to reduce certain redundancies in the search space. We can prune some solvable branches that would only lead to redundant solutions. For example, let $\Gamma_1 = \{x = a, y = a\}$ and $\Gamma_2 = \{z = x+y\}$ where $+$ is associative and commutative. Clearly, $Lab(x) = Lab(y) = \Sigma_1$ and $Lab(z) = \Sigma_2$. And the order must be such that x and y are below z. But there are two different partitions that lead to a solution: We can identify x and y or leave them different. The resulting solution looks the same in both cases. Hence we compute only one partition. Other, more complicated examples occur in ordering decisions. It turns out, that sometimes it is useful to order variables of the same label to avoid the computation of superfluous orders that only lead to redundant solutions. A longer discussion of this side issue would be beyond the limited scope of this paper.

Deterministic Combination

It is interesting to observe that there exists a class of constraint systems for which the deductive combination algorithm has *PTIME* complexity. In [15], Schulz gives a general description of a *PTIME* combination algorithm for certain equational theories. This algorithm is extended to the combination of quasi-free structures in [12]. The class of structures that are deterministically combinable is quite restricted. Currently, only unitary regular collapse-free structures are known to belong to it.

Although our deductive component algorithm is designed for the general case, it turns out to be an implementation of the deterministic algorithm when applied to component algorithms satisfying the conditions imposed in [15] and [12]. Our component algorithms for unification in the empty theory, for rational tree algebras, and for feature structures meet these conditions. Thus, when applied to these structures, our combination algorithm runs without backtracking. This deterministic behaviour shows the great impact of interchanging decisions between component algorithms.

4 Component Algorithms

In order to prune the search space significantly, new component algorithms are needed for the deductive method. When designing these algorithms one should take into account the special way in which they are called. Constraint solvers are usually designed to work incrementally (e.g., [8]). But standard unifica-

tion algorithms are "one shot" algorithms: they are started only once with all information they need given and compute final results. Deductive component algorithms must be able to cope with partial information and deliver a meaningful but not necessarily the final result. More importantly, when receiving new information the algorithms should not restart computation from scratch but rather continue on the base of their prior internal states. Otherwise, the search space would be partially shifted from the combination algorithm to the deductive component algorithms. The same holds for the standard component algorithms for problems with linear constant restrictions that perform a complete test at the end of the combination algorithm: they should take into account the information already computed by the corresponding deductive component algorithms.

Note that there is no need for completeness in the deductive component algorithm: the algorithm need not compute all decisions implied by the input and it need not return an inconsistent set if the problem is unsolvable. Thus an algorithm returning always the empty set would be correct, though it would not contribute to the deductive process. This, however, enables us to use every structure that is suitable for the original algorithm. In the other extreme it might not be advisable to compute new decisions at any cost; there should be a careful consideration between optimisations of the combination algorithm resulting from new decisions and a higher complexity of the deductive component algorithm.

We have developed deductive component algorithms for the free theory, A, AC, and ACI and for rational trees and feature structures. A detailed description of these algorithms would be beyond the limited space of this paper. In the following, we outline the ideas underlying the algorithms for the free theory, a theory in which one can deduce many decisions, and for ACI as a more complicated example.

Syntactic Unification

The deductive algorithm for the free theory is based on the quasi-linear algorithm described in [4] where terms and unifiers are represented as directed acyclic graphs. We assume that the reader is familiar with this representation. When the deductive component algorithm is called for the first time, the dag is built, which is then used again for all further calls of this component algorithm. Decisions of the form $x \not\doteq y$, $x \dot{\mapsto} \Sigma_i$, $x \dot{\not\mapsto} \Sigma_i$, or $x \dot{\leq} y$ do not initiate any computation. Only identification decisions $x \doteq y$ cause a call of the corresponding unification procedure, which updates the existing dag. The decision set to be returned by the component algorithm can be computed from the dag: $x \doteq y$ is returned if x and y are identified in the dag; $x \dot{\mapsto} \Sigma_{Free}$ is returned if x is connected to a non-variable term; $x \dot{<} y$ is returned if x can be reached from y. Additionally $x \not\doteq y$ is returned if x and y are *certainly not unifiable*. The algorithm does not test real unifiability of x and y since it would be too costly to do this for all pairs of variables; instead it tests if the variables are connected to non-variable terms with different topsymbol. The dag is also used by the decision procedure for problems with linear constant restrictions. This

algorithm works exactly like the deductive component algorithm, except that it does not compute a decision set but returns solvable or unsolvable.

The deductive algorithm for rational trees works similar to this algorithm. It does not perform an occur-check and it returns $x \dot{<} y$ only if x can be reached from y and y has been labelled by another structure.

The Theory *ACI*

In the theory of Abelian monoids, *ACI*, the binary function symbol $+$ is associative, commutative and idempotent. In [11], an algorithm was given that decides solvability of *ACI*-unification with constants. The main idea is to set up Horn clauses which describe the solvability of the equations. The Horn clauses are built from propositional variables $P_{x,a}$ which are true iff the constant a does not occur in a solution for the variable x. A clause $P_{x,a} \wedge P_{y,a} \Rightarrow \mathsf{False}$ means that the problem is unsolvable if a does neither appear in x nor in y, or equivalently: if we can deduce that a does not occur in x, then it must appear in y.

We extend the algorithm given in [11] for our situation where the set of variables and constants is not fixed in the beginning. By this, we prevent that new Horn clauses have to be set up when a new labelling decision is made. Let $\mathcal{V}_{ACI}$ be the set of variables in Γ_{ACI}; note that there are no constants in Γ_{ACI}. We introduce a new constant $\bar{x}$ for each variable $x \in \mathcal{V}_{ACI}$ and construct two types of Horn clauses:

- $\bigwedge_{y \in \mathcal{V}_{ACI}} P_{x,\bar{y}} \Rightarrow \mathsf{False}$ for each variable $x \in \mathcal{V}_{ACI}$,
- $P_{x_1,\bar{y}} \wedge \ldots \wedge P_{x_k,\bar{y}} \Leftrightarrow P_{u_1,\bar{y}} \wedge \ldots \wedge P_{u_l,\bar{y}}$
  for each $y \in \mathcal{V}_{ACI}$ and each equation $x_1 + \ldots + x_k = u_1 + \ldots + u_l \in \Gamma_{ACI}$.

The first type of clauses guarantees that the solution for each variable contains at least one constant. The second type represents the equations of Γ_{ACI}: if a constant does not appear on the left hand side, it must not appear on the right hand side, and vice versa. A decision $x \dot{\not\to} \Sigma_{ACI}$ introduces the Horn clauses $P_{x,\bar{x}} \Rightarrow \mathsf{False}$ and $\Rightarrow P_{x,\bar{y}}$ for each $y \in \mathcal{V}_{ACI}$ with $y \neq x$, i.e., the propositional variables are set to False and True, respectively. The effect of these clauses is that $\bar{x}$ is the only constant that appears in x, i.e., x is identified with $\bar{x}$ and is treated like a constant by the algorithm. A decision $x \dot{<} y$ causes the atom $P_{x,\bar{y}}$ to be set to True.

The constraint problem with linear constant restrictions is solvable iff the set of Horn clauses is solvable. This can be tested efficiently by an algorithm which constructs a graph from the Horn clauses and propagates True and False through this graph (see [11]). The set of Horn clauses (and the corresponding constraint problem) is unsolvable if True meets False during this propagation. New decisions can be deduced from the atoms mapped to True or False: $x \dot{\mapsto} \Sigma_{ACI}$ is returned if $P_{x,\bar{y}}$ is set to False and $x \dot{\neq} y$ has been already deduced or if $P_{x,\bar{y}}$ and $P_{x,\bar{z}}$ have been set to False for three different variables x, y, and z. The decision $x \dot{<} y$ is returned if $P_{x,\bar{y}}$ has been set to False with $x \neq y$.

Like the dag for syntactic unification, the Horn clauses and the state of the propositional variables are stored and used again for each further call of the

component algorithm; only when a new identification decision $x \doteq y$ is deduced by another component algorithm, the clauses have to be set up anew.

5 A Selection Strategy for Non-deterministic Decisions

The deductive method is a method to calculate the consequences of a non-deterministic decision, once it is made. It does not state, how to select the next non-deterministic decision. In this section, we will describe a strategy for selecting the next decision called the *iterative strategy*. It is based on the insight that trying to find a set of decisions for the whole problem is best done by looking at one component at a time. We assume that the component systems are linearly ordered by some heuristics. One such heuristic is to place the more deterministic systems in front. The first non-deterministic choices are made for the first system only. And one proceeds to the next system after all choices for the first system are made and a set of decisions is found with which the first component problem is solvable. Suppose all decisions for the first k systems are already made. The next non-deterministic choice is made for system $k+1$, if there is one left. These choices are made locally, that means the decisions are made only on variables of system $k+1$, the labelling determines solely, if the variable will be assigned to the current system, or not; and an order is determined only between two variables, if one belongs to the current system while the other does not. Implicitly, we have already given a priority order of the decisions: first identification or discrimination decisions, then labelling decisions and finally ordering decisions for the current system. It should be clear that after each non-deterministic decision the deductive process is started to deduce its consequences.

The main effect of this selection strategy is that we proceed to non-deterministic choices for the next system only after we found a set of choices with which all previous systems are solvable. This leads to earlier detection of failure, when one component problem is unsolvable. As a side effect, the search space is reduced in that certain superfluous non-deterministic choices such as the ordering of two variables that do not occur together in at least one system are never made due to localisation of choices.

6 Tests

The combination method and component algorithms for the free theory, A, AC, and ACI as well as for rational tree algebras and feature structures are implemented[3] in COMMON LISP using the KEIM toolkit [10]. In the following we show some results of our optimisations. As already stated, the constraint solvers for rational tree algebras and feature structures are such that one can combine them even deterministically. Hence we do not present any test data for

[3]The implementation can be found at `http://www-lti.informatik.rwth-aachen.de/Forschung/unimok.html`.

them. In order to test our algorithms with examples that occur in practice we used the REVEAL theorem prover [6]. For some example theorems, we collected all unification problems that are generated and solved by REVEAL while proving this theorem. These theorems (and the corresponding set of unification problems) contain free function symbols and constants and one or two AC-symbols.

Table 1 gives an overview of the run time for some sets of unification problems. The first six lines contain all unification problems that have to be solved by REVEAL during the proof search or completion of the respective example. All examples except the first one contain two AC-symbols and several free symbols. The last three examples, containing several AC- and ACI-symbols, are added to demonstrate the potential of the iterative selection strategy. In order to see the effect of the iterative selection strategy on its own, we integrated it into the original algorithm (column 'it'). An empty cell in the columns indicates that the algorithm was aborted after one hour.

		Time in seconds						Bktrk	
Example	Size	i+d	ded	i+d-	ded-	it	orig	i+d	ded
Abelian group	29	3.7	3.7	5.0	5.0	11.6	17.2	4	4
Boolean ring	51	3.2	3.2	4.8	4.8	3.5	3.3	0	0
Boolean algebra	122	15.8	15.7	20.5	24.5			12	12
`exboolston`	87	12	12	948	997			17	14
`exgrobner`	1002	154	155	1442	1488			65	66
`exuqsl2`	404	109	108					74	74
AC*–ACI* 1	1	16	101	74	385	15		16	103
AC*–ACI* 2	1	31	407	393		841		13	205
AC*–ACI* 3	1	67	557			248		22	192

Legend: Size: Number of problems; i+d: Integration of Iterative and Deductive Method; ded: Deductive Method; i+d-: Iterative & Deductive Method, but AC-component replaced by one that uses only collapse-freeness and regularity; ded-: Deductive Method, but AC-component replaced by one that uses only collapse-freeness and regularity; it: Iterative selection strategy in original algorithm; orig: Original unoptimised algorithm; Bktrk: Number of backtracking steps.

Table 1: Run time of some example sets

We want to emphasise the differences between column 'ded' and 'ded-'. Column 'ded-' shows the run time of the algorithm when using only syntactic properties as described in [2]; a comparison with column 'ded' demonstrates the power of the deductive method and the deductive component algorithms. The run time decreases dramatically for most examples and some examples even cannot be solved in suitable time when using only syntactic properties.

The use of the iterative selection strategy does not lead to a performance increase in the deductive algorithm in the first six example sets, because these examples are too simple. They contain too few component theories. The last three examples show that the use of the iterative selection strategy can lead to a speed-up by more than one order of magnitude. The equations in these examples contain several AC and ACI-function symbols besides free function symbols. It is a general observation that the iterative selection strategy is

advantageous, if the number of systems is large or the deductive component algorithms do not deduce many decisions.

7 Related Work and Conclusion

The work that is most closely related to ours is the one by Boudet [5]. He presents an optimised algorithm for the combination of finitary equational theories. Our method is hence considerably more general, we are neither restricted to equational theories nor to structures for which minimal complete sets of solutions must be finite. But since combining unification algorithms is such an important instance of our methods, we want to compare the two approaches a bit more detailed. Boudet's algorithm computes a complete set of unifiers for each theory, subsequently treats arisen conflicts between the theories (like one variable getting assigned to different terms in different systems), and repeats these two steps until all conflicts have been solved. Thus there is an important difference in the way the non-determinism inherent in most constraint problems is handled. Our algorithm prophylactically makes a choice for all possible conflict situations before solving the component systems. – We showed that many of these choices can be made deterministically, but some have to be made non-deterministically. – Boudet follows another approach: his algorithm only makes a non-deterministic choice if a conflict actually arises. But as a drawback his approach introduces another source of non-determinism: in order to detect actual conflicts, the algorithm has to compute complete sets of unifiers for the component systems and it has to choose one of the unifiers non-deterministically if the computed set contains more than one solution. The set of unifiers can by very large, e.g., doubly-exponential in the number of variables of the input problem for AC.

Both algorithms have to perform several rounds of computation for the component systems, i.e., consequences (in our algorithm) or complete sets of unifiers (in Boudet's algorithm) have to be computed more than once for each component system. In our algorithm the constraint problem to be solved by a component has the same size in each round. In Boudet's algorithm the computation of a complete set of unifiers is based on the unifier found in the previous round. This means that the unification problem to be solved by a component theory can grow in each round, e.g., the number of variables in an AC-unifier can be exponential in the number of variables of the input problem. This can result in a higher worst-case complexity of Boudet's algorithm: It may well be non-elementary. And that, though the inherent complexity of combination is in NP. Our algorithm on the other hand has singly exponential complexity. Despite its high worst-case complexity, Boudet's algorithm performs quite well in many practical examples. It seems to be a promising line of research to try to integrate some of our optimisation ideas into Boudet's algorithm.

We presented an optimised algorithm for combining constraint solvers. Our empirical analysis indicates that the combined constraint solvers obtained this way can indeed be used in practice. It should be noted, however, that some of the non-determinism is inherent in the combination problem, which means

that even the best optimisation methods cannot avoid this complexity, unless the structures to be combined are severely restricted, as pointed out in the subsection on deterministic combination.

Acknowledgments: We would like to thank Franz Baader and Klaus Schulz for helpful discussions and comments on drafts of this paper.

References

[1] P. Aczel. *Non-wellfounded Sets*. Number 14 in CSLI Lecture Notes. CSLI, Stanford University, USA, 1988.

[2] F. Baader and K. U. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. *Journal of Symbolic Computation*, 21:211–243, 1996.

[3] F. Baader and K. U. Schulz. Combination of constraint solvers for free and quasi-free structures. *Theoretical Computer Science*, 192:107–161, 1998.

[4] F. Baader and J. H. Siekmann. Unification theory. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, Deduction Methodologies. Clarendon Press, Oxford, 1994.

[5] A. Boudet. Combining unification algorithms. *Journal of Symbolic Computation*, 16(6):597–626, 1993.

[6] T. Chen and S. Anantharaman. STORM: A many-to-one associative-commutative matcher. In J. Hsiang, editor, *Rewriting Techniques and Applications*, Proceedings of RTA-95, pages 414–419, Kaiserslautern, 1995. Springer LNCS 914.

[7] A. Colmerauer. Equations and inequations on finite and infinite trees. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 85–99, Tokyo, 1984. North-Holland Publishing Company.

[8] A. Colmerauer. An introduction to PROLOG III. *Commun. ACM*, 33:69–99, 1990.

[9] A. Dovier, A. Policriti, and G. Rossi. Integrating lists, multisets, and sets in a logic programming framework. In F. Baader and K. U. Schulz, editors, *Frontiers of Combining Systems*, Proceedings of FroCoS'96. Kluwer Academic Publishers, 1996.

[10] X. Huang, M. Kerber, M. Kohlhase, E. Melis, D. Nesmith, J. Richts, and J. Siekmann. KEIM: A Toolkit for Automated Deduction. In A. Bundy, editor, *Automated Deduction — CADE-12*, Proceedings, pages 807–810, Nancy, 1994. Springer LNAI 814.

[11] D. Kapur and P. Narendran. Complexity of unification problems with associative-commutative operators. *Journal of Automated Reasoning*, 9:261–288, 1992.

[12] S. Kepser. *Combination of Constraint Systems*. PhD thesis, CIS, Universität München, 1998. Available at `ftp://ftp.cis.uni-muenchen.de/pub/kepser/ccl/diss.ps.gz`.

[13] A. I. Mal'cev. *The Metamathematics of Algebraic Systems*, volume 66 of *Studies in Logic*. North-Holland Publishing Company, 1971.

[14] M. Schmidt-Schauß. Unification in a combination of arbitrary disjoint equational theories. *Journal of Symbolic Computation*, 8(1,2):51–99, 1989.

[15] K. U. Schulz. Combining unification- and disunification algorithms—tractable and intractable instances. Research Report CIS-Rep-96-99, CIS, LMU Munich, Germany, 1996.

[16] G. Smolka and R. Treinen. Records for logic programming. *Journal for Logic Programming*, 18(3):229–258, 1994.