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 two optimisation methods, called the *iterative* and the *deductive* method. The iterative method reorders and localises the non-deterministic decisions. The deductive method uses specific algorithms for the components to reach certain decisions deterministically. 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 [11, 10]. In a recent paper [2], 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

[*] 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.

comprise many important infinite non-numerical solution domains such as (quotient) term algebras [16], rational tree algebras [9], vector spaces, hereditarily finite wellfounded and non-wellfounded lists, sets [1] and multi sets as well as certain types of feature structures [21]. The combined solution domain the authors present in [2], the so called free 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 [18] and rewriting engines [13] 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 [19] and Boudet [5]. With the development of constraint-based approaches to theorem proving [7, 17] and rewriting [15], the interest in combining unification algorithms extended towards combinations of decision procedures, for which Baader and Schulz [3] finally presented a general algorithm.

As a generalisation of the one given in [3], the algorithm for combining constraint solvers in [2] 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 and are still general enough to be applicable to a large class of constraint solvers. The methods we propose are the so called iterative and deductive method. The iterative method reorders the non-deterministic decisions. 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 deductive method 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 [20] for a special subclass of constraint solvers. 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., [16]. 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 [2]. 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 of $\mathfrak{A}^\Sigma$.

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.,

[16]), rational tree algebras ([9]), feature structures with arity ([21]), domains with nested, finite or rational lists (rational lists are used in Prolog III, see [10]), and domains with nested, finite or rational sets ([1]) are quasi-free structures. For details we refer to [2].

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

Note that $m(\mathrm{Stab}^{\mathfrak{A}}(a))$, the image of $\mathrm{Stab}^{\mathfrak{A}}(a)$ under m, can contain non-atoms; therefore we have to apply $\mathrm{Stab}^{\mathfrak{A}}$ again.

Baader and Schulz [2] 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 paiwise 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 free algebras defined by equational theories over disjoint signatures, then their free amalgamated product is the free algebra defined by 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 [3]). 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* $\mathcal{U} := \{x \mid \exists i,j : i \neq j, x \in \mathit{Var}(\Gamma_i) \cap \mathit{Var}(\Gamma_j)\}$. 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 $Lab : \mathcal{U}/_\Pi \rightarrow \{\Sigma_1, \ldots, \Sigma_n\}$ 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 of $\mathcal{U}$ into $\mathfrak{A}_i^{\Sigma_i}$ 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 $x, y \in \mathcal{U}$:

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

Intuitively speaking, item two guarantees that all variables receiving 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 functions and their orders differ at most in the ordering of variables with 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 [2], 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 [3].

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 :

[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 $\mathcal{U}$.

$\mathcal{U}/_\Pi \to \{\Sigma_1, \ldots, \Sigma_n\}$.
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:

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

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:

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

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 to tell 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* $x \dot{\neq} y \in D$, *and*
   either $x \dot{<} y \in D$ *or* $y \dot{<} x \in D$ *if* $x \dot{\mapsto} \Sigma_i, y \not\dot{\mapsto} \Sigma_i \in D$, *and*
   $x \dot{\mapsto} \Sigma_i \in D$ *for one* Σ_i.

3 Iterative Decomposition

The Principle

A major disadvantage of the original method is late detection of failure. Suppose the input problem consists of constraint problems of five different components and that the second sub constraint problem – and thus the whole problem – is unsolvable. The original method always makes all decisions for all constraint problems. In order to detect the insolvability of the second component, all decisions for all the following components must be considered as well before testing solvability. Thus the whole search tree of the remaining constraint problems must be considered before the algorithm establishes that at any leaf of this tree the second component is unsolvable, independently of the decisions made for later components.

Avoiding this problem is the main goal of the iterative decomposition method: components are solved iteratively, one component at a time. All decisions in the non-deterministic steps are made locally, for the current component only, and after that, this component is tested for solvability. So we start by non-deterministically choosing a variable identification, a labelling, and an ordering that solves the first component problem. And we proceed from one component constraint problem to another by making the choices necessary to solve the next component problem while respecting previously made choices. If it turns out that previously made choices make the current component problem unsolvable, we have to backtrack to the previous component problem and try another set of choices. Making choices locally just for one component problem means the following. We identify or discriminate variables of the current component problem, only. We label variables of the current component problem, and furthermore we only determine whether a variable receives the signature of the current component problem as label or whether it is treated as a constant in this component. And just the variables of the current component problem are ordered.

The advantages of the iterative decomposition are twofold. Firstly, iterative decomposition remedies the disadvantage of late detection of insolvability as described above. If a component problem is unsolvable, this is detected when trying to solve this component problem. Therefore no decisions about later component problems will be made.

Secondly, the search space is reduced as compared to the original algorithm by avoiding certain superfluous choices. Even under the assumption that all component problems of the input constraint problem are interrelated, there are variable identifications and orderings that are not needed. For example, if two variables do not occur commonly in one component problem after all identifications being made, then ordering them either way does not affect solvability. Since iterative decomposition can make decisions only on variables that occur together in at least one component problem, these superfluous choices will not be made.

The Algorithm

Before we present the algorithm, we have to define a condition when all choices for one component constraint problem have been made. Recall that $\mathcal{U}_i$ denotes the set of combination variables of problem Γ_i.

Definition 3.1 A decision set D is *complete for component* i, iff for all variables $x, y \in \mathcal{U}_i$

either $x \doteq y \in D$ or $x \dot{\neq} y \in D$, and
either $x \dot{\mapsto} \Sigma_i \in D$ or $x \dot{\not\mapsto} \Sigma_i \in D$, and
either $x \dot{<} y \in D$ or $y \dot{<} x \in D$ if $x \dot{\mapsto} \Sigma_i, y \dot{\not\mapsto} \Sigma_i \in D$.

In the following description, we collect previously made decisions in the form of sets of decisions D_i. Each set D_i will be a consistent closed set of decisions collecting the choices we have made so far. Define $D_0 := \mathit{Clo}(\emptyset)$, i.e., the initial set of decisions is trivial.

For component problems $i := 1$ to n repeat the following steps

Step 1: Variable Identification

Choose a partition Π amongst the variables $\mathcal{U}_i$. Define $D'_{i,=} := \{x \doteq y \mid x \equiv_\Pi y \Pi\}$ and $D'_{i,\neq} := \{x \dot{\neq} y \mid x \not\equiv_\Pi y \Pi\}$. The partition Π must be chosen in such a way, that $D_{i-1} \cup D'_{i,=} \cup D'_{i,\neq}$ is consistent. This means that previously made identifications and discriminations must be observed.

Step 2: Labelling

Choose some set $V \subseteq \mathcal{U}_i$ to form the labelling decision set $D'_{i,Lab} := \{x \dot{\mapsto} \Sigma_i \mid x \in V\} \cup \{x \dot{\not\mapsto} \Sigma_i \mid x \in \mathcal{U}_i \setminus V\}$ in such a way that $D_{i-1} \cup D'_{i,=} \cup D'_{i,\neq} \cup D'_{i,Lab}$ is consistent. Therefore labels are assigned to whole classes of the partition Π, and a label can only be assigned to variables that have not yet received one.

Step 3: Ordering

Choose a set of ordering decisions $D'_{i,<} \subset \{x \dot{<} y, y \dot{<} x \mid x, y \in \mathcal{U}_i \text{ and } x \dot{\mapsto} \Sigma_i, y \dot{\not\mapsto} \Sigma_i \in D'_{i,Lab}\}$ such that each pair $x, y \in \mathcal{U}_i$ with distinct labels is ordered and $D_{i-1} \cup D'_{i,=} \cup D'_{i,\neq} \cup D'_{i,Lab} \cup D'_{i,<}$ is consistent. This implies amongst other things that the order is non-cyclic and that previous ordering decisions are respected.

Define D_i as the closure of $D_{i-1} \cup D'_{i,=} \cup D'_{i,\neq} \cup D'_{i,Lab} \cup D'_{i,<}$. Define $D_i|_{\mathcal{U}_i} \subseteq D_i$ as the subset of D_i that contains only decisions over the variable set $\mathcal{U}_i$.

Step 4: Testing the Component Problem Γ_i with Decision Set If there is a Σ_i-substitution that solves $(\Gamma_i, D_i|_{\mathcal{U}_i})$, continue with the next component problem. Otherwise choose another set of decisions. If no other choice is left for the current component problem Γ_i, backtrack over components $i-1, \ldots, 1$, i.e., try another choice in the preceding components.

Proposition 3.2 *The input problem Γ is solvable, iff there is a set D_n such that for each $i = 1, \ldots, n$ the component problem with decision set $(\Gamma_i, D_i|_{\mathcal{U}_i})$ is solvable.*

Note that testing $(\Gamma_i, D_i|_{\mathcal{U}_i})$ for solvability can be performed by the same component algorithms as are used in the original algorithm.

We will now give syntactic criteria for when an extension is consistent.

Lemma 3.3 *In Step 1, $D_{i-1} \cup D'_{i,=} \cup D'_{i,\neq}$ is consistent, iff the following two conditions are true: both $x \equiv_\Pi y\Pi$ if $x \doteq y \in D_{i-1}$, and $x \not\equiv_\Pi y\Pi$ if $x \not\doteq y \in D_{i-1}$ for all $x, y \in \mathcal{U}_i$.*

Proof. If $D_{i-1} \cup D'_{i,=} \cup D'_{i,\neq}$ is consistent, then clearly the two conditions hold. For the inverse direction, D_{i-1} and $D'_{i,=} \cup D'_{i,\neq}$ are consistent. So the only way inconsistencies can arise by $d \in D_{i-1}$ and $\neg d \in D'_{i,=} \cup D'_{i,\neq}$ for some decision d. This can only happen by either $x \doteq y \in D_{i-1}$ and $x \not\equiv_\Pi y\Pi$ or $x \not\doteq y \in D_{i-1}$ and $x \equiv_\Pi y\Pi$ for some $x, y \in \mathcal{U}_i$. ∎

Lemma 3.4 *In Step 2, $D_{i-1} \cup D'_{i,=} \cup D'_{i,\neq} \cup D'_{i,Lab}$ is consistent, iff the following two conditions are true: both $[x]_\Pi \subseteq V$ for all $x \in V$, and $V \cap \{x \mid \exists j < i : x \dot{\mapsto} \Sigma_j \in D_{i-1}\} = \emptyset$.*

Proof. If $D_{i-1} \cup D'_{i,=} \cup D'_{i,\neq} \cup D'_{i,Lab}$ is consistent, then clearly the two conditions hold. For the inverse direction, $D_{i-1} \cup D'_{i,=} \cup D'_{i,\neq}$ and $D'_{i,Lab}$ are consistent. There are two ways inconsistencies can arise. There can be some decision d such that $d \in D_{i-1} \cup D'_{i,=} \cup D'_{i,\neq}$ and $\neg d \in D'_{i,Lab}$. Or the inconsistency occurs when forming the closure of $D_{i-1} \cup D'_{i,=} \cup D'_{i,\neq} \cup D'_{i,Lab}$. The former case can only happen if there is an $x \in V$ such that $x \dot{\mapsto} \Sigma_j \in D_{i-1}$ for some $j < i$. The latter case occurs only, when $x \not\dot{\mapsto} \Sigma_i \in D'_{i,Lab}$ and $x \dot{\mapsto} \Sigma_i \in Clo(D_{i-1} \cup D'_{i,=} \cup D'_{i,\neq} \cup D'_{i,Lab}) \setminus (D_{i-1} \cup D'_{i,=} \cup D'_{i,\neq} \cup D'_{i,Lab})$. This happens, when $x \doteq y \in D_{i-1} \cup D'_{i,=} \cup D'_{i,\neq}$ and $y \dot{\mapsto} \Sigma_i \in D'_{i,Lab}$. Thus there is a y such that $y \in V$ but $[y]_\Pi \not\subseteq V$. ∎

Correctness and Completeness

We presume the correctness and completeness of the original decomposition with basic optimisations, as stated in Proposition 2.4.

Lemma 3.5 *For each i with $1 \leq i \leq n$, the decision set $D_i|_{\mathcal{U}_i}$ is closed, consistent and complete for component i.*

Proof. $D_i|_{\mathcal{U}_i}$ is consistent as a subset of the consistent set D_i. $D_i|_{\mathcal{U}_i}$ is closed, because it is the reduction of the closed set D_i that contains all decisions over variables $\mathcal{U}_i$.

Let $x, y \in \mathcal{U}_i$. Then either $x \doteq y \in D_i|_{\mathcal{U}_i}$ or $x \dot{\neq} y \in D_i|_{\mathcal{U}_i}$ due to Step 1 of the algorithm. And either $x \dot{\mapsto} \Sigma_i \in D_i|_{\mathcal{U}_i}$ or $x \dot{\not\mapsto} \Sigma_i \in D_i|_{\mathcal{U}_i}$ due to Step 2. If $x \dot{\mapsto} \Sigma_i, y \dot{\not\mapsto} \Sigma_i \in D_i|_{\mathcal{U}_i}$ then immediately by Step 3 either $x \dot{<} y \in D_i|_{\mathcal{U}_i}$ or $y \dot{<} x \in D_i|_{\mathcal{U}_i}$. Therefore $D_i|_{\mathcal{U}_i}$ is complete for component i by Definition 3.1. ■

Proposition 3.6 *If for all i with $1 \leq i \leq n$ there exists a Σ_i-substitution σ_i that solves $(\Gamma_i, D_i|_{\mathcal{U}_i})$, then the input problem Γ is solvable.*

Proof. D_n is consistent by definition. Define the following generalised linear constant restriction $L = (\Pi, Lab, <_L)$ by

- $x \equiv_\Pi y\Pi$, iff $x \doteq y \in D_n$,
- $Lab(x) = \begin{cases} \Sigma_i, & \text{if } x \dot{\mapsto} \Sigma_i \in D_n, \\ \Sigma_n, & \text{otherwise;} \end{cases}$
- $<_L$ is given by any consistent extension of
  $x <_L y$, if $x \dot{<} y \in D_n$
  that orders each two variables with different labels.

L satisfies D_n, and if σ_i solves $(\Gamma_i, D_i|_{\mathcal{U}_i})$ then σ_i solves (Γ_i, L). Thus the input problem Γ is solvable due to correctness of the original algorithm (Proposition 2.4). ■

Now, we proceed to show completeness of the algorithm. The aim is to show the following

Proposition 3.7 *If the input problem Γ is solvable, then Γ is solvable by iterative decomposition.*

We will prove this proposition using the completeness of the original algorithm with basic optimisations. Due to the completeness of the original algorithm, if the input problem is solvable, there exists a generalised linear constant restriction L such that the output tuples $((\Gamma_i, L))_{1 \leq i \leq n}$ are solvable. This generalised linear constant restriction is used to guide the choices that will be made in each iteration of the iterative method.

Definition 3.8 Let $L = (\Pi, <_L, Lab)$ be a generalised linear constant restriction. Define

> the set of equality decisions
> $D_{\downarrow=} := \{x \doteq y \mid x \equiv_\Pi y\Pi \text{ and } \exists i \leq n : x, y \in \mathcal{U}_i\}$,
> the set of disequality decisions
> $D_{\downarrow\neq} := \{x \dot{\neq} y \mid x \not\equiv_\Pi y\Pi \text{ and } \exists i \leq n : x, y \in \mathcal{U}_i\}$,
> the set of labelling decisions
> $D_{\downarrow Lab} := \{x \dot{\mapsto} \Sigma_i \mid Lab(x) = \Sigma_i \text{ and } x \in \mathcal{U}_i\}$,
> the set of ordering decisions as the set
> $D_{\downarrow<} := \{x \dot{<} y \mid x <_L y, \exists j : x, y \in \mathcal{U}_j, (Lab(x) = \Sigma_j,$
> $Lab(y) \neq \Sigma_j) \text{ or } (Lab(x) \neq \Sigma_j, Lab(y) = \Sigma_j)\}$.

Set $D_{\downarrow L}$, the *decision set induced by* L, as the closure of the union $D_{\downarrow=} \cup D_{\downarrow\neq} \cup D_{\downarrow<} \cup D_{\downarrow Lab}$.

Lemma 3.9 *$D_{\downarrow L}$ is a closed consistent set.*

Lemma 3.10 *Let Γ_i be a constraint problem. Let $L = (\Pi, Lab, <_L)$ be a linear constant restriction and $D_{\downarrow L}$ the decision set induced thereby. Then (Γ_i, L) is solvable , if and only if $(\Gamma_i, D_{\downarrow L}|_{\mathcal{U}_i})$ is solvable, where $D_{\downarrow L}|_{\mathcal{U}_i}$ is $D_{\downarrow L}$ restricted to decisions over variables $\mathcal{U}_i$.*

Proof. If (Γ_i, L) is solvable, then $(\Gamma_i, D_{\downarrow L}|_{\mathcal{U}_i})$ is solvable, because the decision set $D_{\downarrow L}|_{\mathcal{U}_i}$ induced by L contains only a subset of the decisions of L.

For the inverse direction, suppose σ solves $(\Gamma_i, D_{\downarrow L}|_{\mathcal{U}_i})$. If for $x, y \in \mathcal{U}_i$: $\sigma(x) = \sigma(y)$, then $x \doteq y \in D_{\downarrow L}|_{\mathcal{U}_i}$ and therefore $x \equiv_\Pi y\Pi$.
Now let $x \equiv_\Pi y\Pi$. Then $x \doteq y \in D_{\downarrow L}|_{\mathcal{U}_i}$ by definition of $D_{\downarrow L}$ and therefore $\sigma(x) = \sigma(y)$.

Let for $y \in \mathcal{U}_i$: $Lab(y) = \Sigma_j$ with $j \neq i$. If $y \in \mathcal{U}_j$, then $y \dot{\mapsto} \Sigma_j \in D_{\downarrow L}|_{\mathcal{U}_i}$. If $y \notin \mathcal{U}_j$, then there is no k such that $y \dot{\mapsto} \Sigma_k \in D_{\downarrow L}|_{\mathcal{U}_i}$. In both cases $y \dot{\mapsto} \Sigma_i \notin D_{\downarrow L}|_{\mathcal{U}_i}$. Therefore $\sigma(x) \in X$ as demanded.

Let for $x, y \in \mathcal{U}_i$: $Lab(x) = \Sigma_j, Lab(y) = \Sigma_i, j \neq i$ and $\sigma(x) \in \mathrm{Stab}(\sigma(y))$. Then $y \dot{\mapsto} \Sigma_i \in D_{\downarrow L}|_{\mathcal{U}_i}$; and $x \dot{\not\mapsto} \Sigma_i \in D_{\downarrow L}|_{\mathcal{U}_i}$ according to the same argument as in the previous paragraph. Therefore $x \dot{<} y \in D_{\downarrow L}|_{\mathcal{U}_i}$ and $x <_{L|_{\mathcal{U}_i}} y$ by definition of $D_{\downarrow L}|_{\mathcal{U}_i}$. ■

We now have to show that $D_{\downarrow L}$ is a potential decision set calculated by the iterative decomposition.

Lemma 3.11 *Let (Γ, L) be a solvable component problem with decision set L. Then the induced decision set $D_{\downarrow L}$ can be constructed by the iterative decomposition, i.e., $D_{\downarrow L} = D_n$.*

Proof. In each component i, we make the following choices. Two variables $x, y \in \mathcal{U}_i$ are identified according to $D_{\downarrow L}$, that is, iff $x \doteq y \in D_{\downarrow L}$, then $x \doteq y \in D'_{i,=}$; iff $x \not\doteq y \in D_{\downarrow L}$, then $x \not\doteq y \in D'_{i,\neq}$. Iff $x \dot{\mapsto} \Sigma_i \in D_{\downarrow L}$, then $x \dot{\mapsto} \Sigma_i \in D'_{i,Lab}$. Iff $x \dot{<} y \in D_<$, then $x \dot{<} y \in D'_{i,<}$.

Claim 1: For $0 \leq i \leq n$: D_i is consistent and $D_i \subseteq D_{\downarrow L}$.
Proof of Claim 1:
$D_0 = \emptyset$ is obviously consistent and a subset of $D_{\downarrow L}$.
Let $i > 0$. $D_{i-1} \subseteq D_{\downarrow L}$ by hypothesis. $D'_{i,=}, D'_{i,\neq}, D'_{i,Lab}$ and $D'_{i,<}$ are subsets of $D_{\downarrow L}$ by definition, thus $D_{i-1} \cup D'_{i,=} \cup D'_{i,\neq} \cup D'_{i,Lab} \cup D'_{i,<}$ is consistent, because it is a subset of the consistent set $D_{\downarrow L}$. D_i defined as the closure of the above union is a subset of $D_{\downarrow L}$ by monotonicity of the closure operator and consistent, because it is a subset of a consistent set.

Claim 2: $D_{\downarrow L} = D_n$.
Proof of Claim 2:
$D_n \subseteq D_{\downarrow L}$ by Claim 1.
Let $x \doteq y \in D_{\downarrow L}$, then $x \doteq y \in Clo(D_{\downarrow =})$. $D_{\downarrow =} = \bigcup_{i=1}^n D'_{i,=}$ by definition, thus $Clo(D_{\downarrow =}) = Clo(\bigcup_{i=1}^n D'_{i,=}) \subseteq D_n$.
Let $x \not\doteq y \in D_{\downarrow L}$. Then, by definition, $x \not\doteq y \in Clo(D_{\downarrow =} \cup D_{\downarrow \neq})$. $D_{\downarrow =} \subseteq D_n$ by the above. If $w \not\doteq z \in D_{\downarrow \neq}$, then there is a j such that $w, z \in \mathcal{U}_j$, and thus $w \not\doteq z \in D'_{j,\neq}$. Therefore $D_{\downarrow \neq} \subseteq D_n$. Thus $x \not\doteq y \in D_n$, since D_n is closed.
Let $x \dot{\mapsto} \Sigma_i \in D_{\downarrow L}$ for some i. Then $x \in \mathcal{U}_i$ by definition, and therefore $x \dot{\mapsto} \Sigma_i \in D'_{i,Lab} \subseteq D_n$.
Concerning the ordering, $D_{\downarrow <} = \bigcup_{i=1}^n D'_{i,<}$ by definition. $Clo(D_{\downarrow =} \cup D_{\downarrow \neq}) = Clo(\bigcup_{i=1}^n D'_{i,=} \cup \bigcup_{i=1}^n D'_{i,\neq})$ by the above. Now $x \dot{<} y \in D_{\downarrow L}$ implies $x \dot{<} y \in Clo(D_{\downarrow <} \cup D_{\downarrow =} \cup D_{\downarrow \neq}) = Clo(\bigcup_{i=1}^n D'_{i,<} \cup \bigcup_{i=1}^n D'_{i,=} \cup \bigcup_{i=1}^n D'_{i,\neq}) \subseteq D_n$.

Claim 3: All of the above choices of the sets $D'_{i,=}, D'_{i,\neq}, D'_{i,Lab}, D'_{i,<}$ are valid as steps in iterative decomposition.
Proof of Claim 3:
That all of these choices can be made consistently, is shown by Claim 1.
For variable identification, the partitioning is directly given by the equivalence classes that result when restricting the equality and disequality decisions of $D_{\downarrow L}$ to the variables of a particular component problem, as done in $D'_{i,=}$ and $D'_{i,\neq}$.
For labelling, the generalised linear constant restrictions ensure that each variable receives only one label and that classes of variables that are identified receive one and the same label.
For ordering, $D'_{i,<}$ contains only ordering decisions on variables of component i. It respects the variable identification, because the generalised linear constant restriction $(\Pi, <_L, Lab)$ does so. And, by definition, each pair $x, y \in \mathcal{U}_i$ of variables where one has component i as label while the other has not is ordered in $D'_{i,<}$. ■

Proof of Proposition 3.7.
Let Γ be solvable. By Proposition 2.4 there exists a generalised linear constant restriction $L = (\Pi, <_L, Lab)$ such that the output tuples $((\Gamma_i, L))_{1 \leq i \leq n}$ have a solution. By Lemma 3.10 the output tuples $((\Gamma_i, D_{\downarrow L}|_{\mathcal{U}_i}))_{1 \leq i \leq n}$ with the induced decision set $D_{\downarrow L}$ are solvable. By Lemma 3.11, there exists a set of choices of the iterative algorithm such that the decision set $D_{\downarrow L}$ is constructed thereby. ∎

4 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 $y \dot{\not\mapsto} \Sigma_i \in D$ and $x \dot{<} y \in D$. Hence the algorithm can choose $x \dot{\mapsto} \Sigma_i \in D$ deterministically and take into account that $y \dot{\not\mapsto} \Sigma_i \in D$ 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 determines 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 4.1 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 $D' \supseteq D$ such that (Γ, D') is solvable, that is, iff

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

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 5 for a discussion on how deductive component algorithms co-operate with standard ones.

Figure 1 shows the combination algorithm. Like before we present the method as a non-deterministic algorithm, i.e., the algorithm contains non-deterministic steps for which both alternatives have to be regarded. In the algorithm, D denotes the current set of decisions. The termination condition in case of success is that the set of decisions is complete, as given in Lemma 2.11.

Proposition 4.2 *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.

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

```
$D := \emptyset$
loop: Repeat
    Deduce consequences:
    Repeat
        For each system $i$
            call the component algorithm of system $i$ to calculate
            new consequences $C$ of $(\Gamma_i, D)$,
            set the new current set of decisions $D := D \cup C$.
            If $D$ is inconsistent
                break loop.        /* exit from outer loop */
    Until no component algorithm computes new decisions.
    If $D$ is 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 complete.
Return $D$.
```

Figure 1: The deductive combination algorithm

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, which entails that all steps can be made deterministically. In [20], Schulz gives a general description of a *PTIME* combination algorithm for certain equational theories. This algorithm can be extended to the combination of quasi-free structures. 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 [20]. 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 deterministically. This deterministic behaviour shows the great impact of interchanging decisions between component algorithms.

5 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., [10]). But standard unification 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. This is not the place to give detailed descriptions of these algorithms. 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 $x \neq y$, $x \dot{\mapsto} \Sigma_i$, $x \not\dot{\mapsto} \Sigma_i$, 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 $x \neq y$ 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 similarly 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 [14], 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 appears neither 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 [14] 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 $x \not\mapsto \Sigma_{ACI}$ 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 $x \mathrel{\dot{<}} y$ 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 [14]). 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: $x \mathrel{\dot{\mapsto}} \Sigma_{ACI}$ is returned if $P_{x,\bar{y}}$ is set to False and $x \mathrel{\dot{\neq}} y$ 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 $x \mathrel{\dot{<}} y$ 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.

Other Component Algorithms

Here, we would like to present at least the basic ideas for the other component algorithms. The theory $A = \{x + (y + z) = (x + y) + z\}$, i.e., the theory of an associative function symbol $+$ is basically the theory of free word equations. The deductive component algorithm translates the input into word equations and simplifies them. The simplification steps allow the computation of new identification, labelling and ordering information. This is an example of a deductive component algorithm which does not compute all consequences. Hence we need to call the standard algorithm for A-unification with linear constant restrictions in the end.

For the theory $AC = \{x + (y + z) = (x + y) + z; x + y = y + x\}$, i.e., the theory of an associative and commutative function symbol $+$, the deductive algorithm is based on [22]. First, the set of minimal solutions of the homogeneous Diophantine equations corresponding to the unification problem is computed. Some of these solutions can be deleted with the help of the existing decisions. From the remaining set of solutions, information about labelling, ordering and identification can be deduced.

The set of minimal solutions has to be recomputed when new identification decisions occur. This might seem to be a drawback at first glance, since computing the solutions of Diophantine equations can be a time-consuming task; but it cannot be worse than in the original combination algorithm, i.e., Diophantine equations are not solved more often, since this happens at most once for every partition of variables. Unfortunately, the number of minimal solutions of the Diophantine equations can be exponential in the size of the unification problem. But at least we do not need to compute complete sets of unifiers, which can even be doubly-exponential in number.

The algorithms for the free theory and for the theories AC and ACI have in common that they behave like decision procedures for unification with linear constant restrictions if called with a complete set of decisions, i.e., they return a correct and complete answer. Therefore the final test does not need to compute anything; it can simply return the result achieved by the corresponding deductive component algorithm.

Rational Trees and Feature Structures

As examples of a quasi-free structures which are not an equational theories the author implemented rational tree algebras and feature structures of the Smolka and Treinen variety [21]. The algorithm for rational tree algebras is a simple extension of the algorithm for syntactic unification. The occurs-check has to be left out and the computation of new decisions is a bit more complicated since certain cyclic solution which are impossible in the free theory have to be taken into account.

We introduced feature structures as examples of quasi-free structures in the paragraph following definition 2.1. The implementation employs techniques for integrating record like data types (as feature structures) into logic programming frameworks developed by Van Roy, Mehl and Scheidhauer [24]. Upon first call, the internal graph-like representation of the feature theory is constructed and used to calculate new identification, labelling and ordering information. This representation needs to be constructed only once. Later on, new incoming identification information does not trigger a complete new setup, rather starts a feature structure unification of the the two structures pending below the newly identified variables. Additional information can be read out of the new structures, if unification succeeds. Incoming labelling or ordering information triggers no unification. Labelling information can help to deduce more information on the ordering. The algorithm is designed in such a way that it behaves like a decision procedure for feature constraint problems with linear constant restrictions when called with a complete set of decisions.

6 Integrating the Deductive and Iterative Method

The two methods described above can easily be integrated. The iterative method is a selection strategy for non-deterministic steps, while the deductive method deduces deterministic consequences from the decisions already made. Therefore integration is achieved by plugging the iterative selection strategy into the deductive algorithm. The combined method looks as follows. Suppose component constraint problems Γ_1 to Γ_{i-1} are solved, the current decision set is D, and D is not complete for component i, the current component. Select a decision $d \notin D$ over the variables of component i such that $D \cup \{d\}$ is consistent. Nondeterministically choose d or its negation and add it to D. Compute consequences and add them to D. If D is still not complete for component i, select the next decision for this component. If D is complete and $(\Gamma_i, D|_{\mathcal{U}_i})$ is solvable, proceed to the next component problem. Otherwise perform backtracking and make an alternative choice for one of the decisions made so far.

The method to compute consequences of a non-deterministic decision should be amended to the new selection strategy as follows. Components that are already solved cannot contribute any new decisions. Consequently only components that still have non-deterministic choices left open are consulted.

7 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 [12]. 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 them. In order to test our algorithms with examples that occur in practice we used the REVEAL theorem prover [8]. For some example theorems, we collected all unification problems that are generated and solved by REVEAL while proving this theorem. These theorems (and

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

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 unification problems
Bktrk	Number of backtracking steps
i+d	Iterative selection strategy in deductive method
ded	Deductive method
i+d-, ded-	Same as i+d/ded, but AC-component replaced by one that uses only collapse-freeness and regularity
it	Iterative selection strategy in original algorithm
orig	Original unoptimised algorithm

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 [3]; 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.

Set	Equations	term-depth	# *ACI*	Ded+Iter time	Ded+Iter bktrk	Ded time	Ded bktrk
1	199/98	6	3	816	1953	81	152
2	200/99	6	3	232	780	>1h	
3	199/101	6	3	330	800	1158	1982
4	200/127	6	3	58	250	42	110
5	200/97	6	3	1362	3971	141	401
6	200/113	6	3	>1h		103	295
7	200/112	6	3	676	2217	189	689
8	200/100	5	0	19	1	19	1
9	200/90	5	0	67	33	75	33
10	200/95	5	0	16	1	15	1
11	200/87	5	0	20	7	21	10
12	200/89	5	0	21	8	21	8
13	200/99	5	1	32	50	31	30
14	200/93	5	1	21	47	26	22
15	200/109	5	1	154	394	3931	12335
16	200/116	5	1	26	50	30	31
17	200/107	5	2	319	1116	83	147
18	200/106	5	2	1250	2627	44	107
19	200/95	5	2	178	462	58	169
20	200/108	5	2	99	414	43	159

Legend: The signature of these problems consists of 2 *A*, 2 *AC*, 0–3 *ACI* and several free function symbols. Equations: number of equations in set and number of solvable equations; term depth: maximal depth of terms; # *ACI*: Number of *ACI*-function symbols in signature; Ded+Iter: deductive combination with iterative selection strategy; Ded: deductive combination with a selection strategy that chooses all identifications first; bktrk: number of backtracking steps.

Table 2: Run time of randomly generated example sets

In order to get more examples, we developed a test set generator. With it, one generates sets of random combined unification problems over signatures containing several function symbols from different theories. Certain means were taken to ensure that about half of the generated problems are solvable.

Table 2 presents some run time results for these randomly generated problem sets. The signature contains 2 A, 2 AC, 0–3 ACI and several free function symbols. The problems are that complex that a use of a combination method different from the deductive combination makes no sense at all.

It is interesting to observe that with these problems, the iterative selection strategy is not always the best choice. There are examples (sets 2, 3, and 15) in which the iterative selection strategy is superior. On the other hand, in the sets 1, 5, 6, and 18 it is much worse than a strategy which firstly settles all variable identification and discrimination decisions for all component problems. It is currently not clear what the conditions are under which one should choose the iterative selection strategy, and when to rather use the other strategy. The presence of several collapsing theories is important, but there are several collapsing theories both in those examples where the iterative selection strategy works well and in those where it flounders. In all these examples, it seems important to make the "right" decisions first, but there is at current no way to state what the "right" decisions are.

Another observation is that there is no simple, e.g., linear, connetion between the run time and the number of backtracking steps. Obviously, some backtracking steps require a lot of time, because they appear high up in the search tree, while others that are close to the leaf nodes of the search tree have a very small influence on the run time.

8 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 the theory 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.

References

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

[2] Franz Baader and Klaus U. Schulz. Combination of Constraint Solvers for Free and Quasi-Free Structures. Technical Report CIS-Bericht-96-90, CIS, Universität München, 1996.

[3] Franz Baader and Klaus U. Schulz. Unification in the Union of Disjoint Equational Theories: Combining Decision Procedures. *Journal of Symbolic Computation*, 21:211–243, 1996.

[4] Franz Baader and Jörg H. Siekmann. Unification Theory. In Dov M. Gabbay, Christopher J. Hogger, and John Alan Robinson, editors,

Handbook of Logic in Artificial Intelligence and Logic Programming, volume 2, pages 41–125. Oxford University Press, 1994.

[5] Alexandre Boudet. Combining Unification Algorithms. *Journal of Symbolic Computation*, 16(6):597–626, 1993.

[6] Alan Bundy, editor. *Automated Deduction, Proceedings CADE-12, Nancy, France*, LNAI 814. Springer-Verlag, 1994.

[7] Hans-Jürgen Bürckert. A Resolution Principle for Clauses with Constraints. In *Stickel* [23], pages 178–192, 1990.

[8] Ta Chen and Siva Anantharaman. STORM: A Many-to-one Associative-commutative Matcher. In Jieh Hsiang, editor, *Rewriting Techniques and Applications, Proceedings RTA-95*, LNCS 914, pages 414–419. Springer-Verlag, 1995.

[9] Alain Colmerauer. Equations and Inequations on Finite and Infinite Trees. In Institute for New Generation Computer Technology, editor, *Proceedings of the 2nd International Conference on Fifth Generation Computing Systems*, pages 85–99, Tokyo, 1984. Ohmsha et al.

[10] Alain Colmerauer. An Introduction to PROLOG III. *Communications of the ACM*, 33:69–90, 1990.

[11] Agostino Dovier, Alberto Policriti, and Gianfranco Rossi. Integrating lists, multisets, and sets in a logic programming framework. In Franz Baader and Klaus U. Schulz, editors, Frontiers of Combining Systems, *Proceedings of the 1st Int. Workshop, FroCoS'96.* Kluwer Academic Publishers, 1996.

[12] Xiaorong Huang, Manfred Kerber, Michael Kohlhase, Erica Melis, Dan Nesmith, Jörn Richts, and Jörg H. Siekmann. KEIM: A Toolkit for Automated Deduction. In *Bundy* [6], pages 807–810, 1994.

[13] Jean-Pierre Jouannaud and Hélène Kirchner. Completion of a Set of Rules modulo a set of Equations. *SIAM Journal on Computing*, 15:1155–1195, 1986.

[14] Deepak Kapur and Paliath Narendran. Complexity of Unification Problems with Associative-Commutative Operators. *Journal of Automated Reasoning*, 9:261–288, 1992.

[15] Claude Kirchner and Hélène Kirchner. Constrained Equational Reasoning. In Gaston H. Gonnet, editor, *Proceedings of SIGSAM 1989 International Symposium on Symbolic and Algebraic Computation: ISSAC'89*, pages 382–389. ACM Press, 1989.

[16] Anatolij Ivanovič Mal'cev. *The Metamathematics of Algebraic Systems.* Edited by Benjamin Franklin Wells, volume 66 of *Studies in Logic.* North-Holland Publishing Company, 1971.

[17] Robert Nieuwenhuis and Albert Rubio. *AC*-superposition with Constraints: No *AC*-unifier Needed. In Alan Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction, Nancy, France*, LNAI, pages 545–559. Springer, 1994.

[18] Gordon D. Plotkin. Building-in Equational Theories. *Machine Intelligence*, 7:73–90, 1972.

[19] Manfred Schmidt-Schauß. Unification in a Combination of Arbitrary Disjoint Equational Theories. *Journal of Symbolic Computation*, 8(1,2):51–99, 1989.

[20] Klaus U. Schulz. Combining Unification and Disunification Algorithms—Tractable and Intractable Instances. Technical Report CIS-Bericht-96-99, CIS, Universität München, 1996.

[21] Gert Smolka and Ralf Treinen. Records for Logic Programming. *Journal for Logic Programming*, 18(3):229–258, 1994.

[22] Mark E. Stickel. A Unification Algorithm for Associative-Commutative Functions. *Journal of the ACM*, 28(3):423–434, 1981.

[23] Mark E. Stickel, editor. *Automated Deduction, Proceedings CADE-10*, LNAI 449, Berlin, Germany, 1990. Springer–Verlag.

[24] Peter Van Roy, Michael Mehl, and Ralf Scheidhauer. Integrating efficient records into concurrent constraint programming. In *8th International Symposium on Programming Languages, Implementations, Logic, and Programs (PLILP96), Aachen*, September 1996.