Goal–Directed Completion using SOUR Graphs*

Christopher Lynch †

April 30, 1996

Abstract

We give the first Goal-Directed version of the Knuth Bendix Completion Procedure. Our procedure is based on Basic Completion and SOUR Graphs. There are two phases to the procedure. The first phase, which runs in polynomial time, compiles the equations and the goal into a constrained tree automata representing the completed system, and a set of constraints representing goal solutions. The second phase starts with the goal solutions and works its way back to the original equations, solving constraints along the way.

1 Introduction

The Knuth Bendix Completion Procedure [7] is the best known procedure for solving word problems and equational unification problems. One reason that it works as well as it does is that it is based on a well-founded ordering. In many cases, this allows it to convert a set of equations into a decision procedure to solve the word problem for its equational theory. This cannot always work, because the word problem is undecidable. The Completion Procedure works by starting with the initial set of equations, continually generating new equations, at each point checking to see if it has discovered a solution to the goal. No matter what the goal is, it always generates the same equations. There has been much research to improve the efficiency of the Completion Procedure, but up until now, nobody has found a way to use the goal to direct the search for solutions of the goal. In this paper, we give the first method for doing that.

The fact that Knuth-Bendix completion is not goal directed means that if a goal is false, and the set of equations has an infinite canonical system, then Knuth-Bendix completion will run forever. A goal directed system could detect that certain critical pairs are not necessary to solve a given goal, and halt and say the goal is not true. It is possible to give heuristics for constructing critical pairs so that equations that are related to the goal are created first. But in order to have a complete system, all critical pairs much be eventually created. So this changes the order in which critical pairs are created, but all of them must be created eventually. It could be argued that the completion procedure can be combined with a narrowing (or rewriting) procedure so that new equations are immediately involved in inferences with the goal. In this sense the completion procedure is more goal sensitive. But it is not goal directed, because it is still the case that all critical pairs must be constructed for a false goal. Also, it is possible to encode the goal as an equation itself, and prove the goal in the completion procedure. However, this process is equivalent to the process of interleaving completion with narrowing, and therefore must construct all critical pairs. In fact, it is worse, because it does not give any priority to the goal. Until now, there have been no modifications to Knuth Bendix completion to allow the process to stop when a goal is false and the canonical set of equations is infinite.

Consider the equational theory $E = \{f(f(x)) = g(f(x))\}$. Suppose we want to know if a and b are equivalent modulo E. The Completion Procedure will generate infinitely many equations from E,

*This document was created using Xy-pic[13]. Thanks to Kristoffer Rose for his help with Xy-pic

†INRIA Lorraine et CRIN, Campus Scientifique, BP 101, 54602 Villers-lës-Nancy cedex, France, Phone: (33) 83 59 30 88, Fax : (33) 83 27 83 19, email: lynch@loria.fr

but never halt and say the goal is not true, although it is trivial to see that the goal does not follow. The General E-Unification Procedure [14] is an alternative to Completion. It starts from the goal, and creates new subgoals by using the equations. But it has a drawback that it cannot incorporate orderings and make things smaller at each point. Given the theory $E = \{f(a) = a\}$, and the goal $g(a) = h(a)$, it will replace $g(a)$ by $g(f(a))$ by $g(f(f(a)))$ etc. It never halts and says "no". Orderings are crucial for converting theories into decision procedures.

For logic programming, there is a procedure which solves goals by building up from the facts until a goal is reached. If those bottom-up proofs are reversed, we have SLD Resolution, a goal-directed procedure. This is possible because an initial rule is involved in every inference. But in the completion procedure, we cannot turn the proof around and work backwards, because inferences may not involve initial equations, and we have no idea what equations will be derived. In our approach, we handle this by schematizing the equations that will be derived, using constrained tree automata. Recently, there have been some papers showing how to schematize terms (see [6] for references). However these approaches do not show how the schematizations can be created.

Our approach is based on Basic Completion [1, 11], which works by saving unification problems in equational constraints instead of applying most general unifiers. It is even possible to save the ordering problems in constraints [12]. In [9], we defined a graphical theorem proving approach, based on Basic Paramodulation. This procedure was refined to be implemented as a Completion Procedure in [10]. The initial problem is stored in a graph with terms represented as dags. The edges forming the dag are called *subterm edges*, and the edges for the equations are called *rewrite edges*. Inferences are transformations on the graph, which create new subterm and rewrite edges, labelled with the constraint and renaming used in the inference. Inference do not increase the number of nodes in the graph. The graphs are called *SOUR graphs*, because the edges in the graph represent **S**ubterm, **O**rdering, **U**nification, and **R**ewrite relations.

SOUR Graphs can be adapted to a goal directed completion procedure. When we add a new edge to the graph, instead of inheriting the actual constraints and renamings, we just give a reference to the constraints and renamings it is built up of. So there are only finitely many edges added to the graph, in fact only polynomially many, and completion halts in polynomial time (we call this a compilation). It creates a schematization of the completed set of equations and a set of constraints representing goal solutions. After the compilation halts, the procedure starts with the goal solutions and words backwards from the goal to the initial equations, solving constraints along the way. If our technique is restricted to narrowing, we could consider the work of [5, 2] to be special cases. We consider our work to be an extension of [8], where it was first shown how to decide the word problem for ground terms in polynomial time, with a congruence closure algorithm. We show that it is possible to perform congruence closure in polynomial time, even for non-ground terms. The difference is that the result of the congruence closure is a constraint which now must be solved.

The paper is as follows. After some brief definitions we define how to compile a set of equations into a SOUR graph, in polynomial time. Next we give the semantics of SOUR graphs, to show that it truly represents the completion of a set of equations. Then we give inference rules to solve the constraints of the SOUR graph. In the last section, we discuss ways to use the procedure as a decision procedure. The proofs are missing in this abstract, but they are straightforward.

2 Preliminaries

We present necessary definitions briefly. We refer the reader to [4] for a more detailed exposition. Terms are defined inductively from a set of functions symbols. If f is a function symbol of *arity* p, and $t_1, \cdots, t_p$ are terms then $t = f(t_1, ...t_p)$ is a *term*, and we say $top(t) = f$. We consider *variables* and *constants* as arity 0 function symbols. Unifiers and substitutions are defined as usual. A *renaming* is an injective substitution ρ such that every variable is mapped to a variable. The renaming *id* is the identity substitution. A *fresh renaming* is a renaming which maps all variables to new variables. In

SOUR graphs, renamings will be written in the form ρ_i, where i is a positive integer. They can be understood to mean that $x\rho_{i_1}\cdots\rho_{i_m}$ is the same as $x\rho_{j_1}\cdots\rho_{j_m}$ if and only if $m = n$ and $i_k = j_k$ for all k, $1 \leq k \leq n$. For this, we use the concept of a *fresh number*, which means an index number that has not previously appeared.

The symbol $\approx$ (resp. $\not\approx$) is a binary symbol, written in infix notation, representing semantic equality (resp. disequality). The 0-ary symbol $\Box$ will represent solutions to equations. An object is *ground* if it contains no variables. We say $s\sigma \approx t\sigma$ is an *instance* of $s \approx t$. if $s\sigma$ and $t\sigma$ are ground. Let EQ be a set containing equations. We define EQ^* to be the reflexive, symmetric, transitive and congruence closure of all the instances of equations $s \approx t$ in EQ.

Let EQ be a set of equalities and disequalities. We define a function Sub so that $Sub(EQ)$ is the set of subterms in EQ. If $t = f(t_1, \cdots, t_n)$ with $n \geq 0$, then $Sub(t) = \{t\} \cup \bigcup_{1\leq i\leq n} Sub(t_i)$. We define $Sub(s \approx t) = Sub(s) \cup Sub(t)$. A disequation $s \not\approx t$ is viewed as a term with s the first argument of $\not\approx$ and t the second, so $Sub(s \not\approx t) = \{s \not\approx t\} \cup Sub(s) \cup Sub(t)$. We define $Sub(EQ) = \bigcup_{eq\in EQ} Sub(eq)$.

We assume that $<$ is a reduction ordering, total on ground terms. The symbol $\doteq$ is a binary symbol that represents syntactic equality, and $\prec$ is a symbol representing the ordering $<$. A constraint is a conjunction of $\doteq$ and $\prec$ expressions. A constraint $s \doteq t$ is true if $s = t$, $s \prec t$ is true if $s < t$, $E_1 \wedge E_2$ is true if E_1 and E_2 are true, and $\top$ is always true. We say that a substitution σ is a *solution* of a constraint φ if $\varphi\sigma$ is true. If there is a substitution σ such that $\varphi\sigma$ is true, then φ is satisfiable. Otherwise it is *unsatisfiable*. We say $\sigma = mgs(\varphi)$ (σ is a *most general solution* of φ) if σ is a solution of φ, and $\sigma \leq \theta$ for all solutions θ of φ. See [3, 12] for algorithms to find the mgs of a constraint, where $\prec$ represents a lexicographic path ordering.

Let EQ be a set of equations and disequations, where no two disequations in EQ have any variables in common. A set of constraints C is a *complete set of solutions* for EQ if (i) for every solution σ of some $\varphi \in C$, there is a disequation $s \not\approx t$ in EQ such that $s\sigma \approx t\sigma \in EQ^*$, and (ii) for every disequation $s \not\approx t$ in EQ and substitution σ such that $s\sigma \approx t\sigma \in EQ^*$, there is a constraint $\varphi \in C$ such that σ is a solution of φ.

A *constrained term*, (or equation or disequation) $t\,[\![\,\varphi\,]\!]$ is a pair of a term (or equation or disequation) t and a constraint φ. An equation or disequation eq is viewed as the constrained equation or disequation $eq\,[\![\,\top\,]\!]$. The meaning of $t_1\,[\![\,\varphi_1\,]\!] \doteq t_2\,[\![\,\varphi_2\,]\!]$, is $t_1 \doteq t_2 \wedge \varphi_1 \wedge \varphi_2$. The meaning of $f(t\,[\![\,\varphi\,]\!])\,[\![\,\Psi\,]\!]$ is $f(t)\,[\![\,\varphi \wedge \Psi\,]\!]$ The Basic Completion inference rules are the following.

Critical Pair

$$\frac{s \approx t\,[\![\,\varphi_1\,]\!] \qquad u[s'] \approx v\,[\![\,\varphi_2\,]\!]}{u[t\rho] \approx v\,[\![\,s\rho \doteq s' \,\wedge\, s \succ t \,\wedge\, u[s'] \succ v \,\wedge\, \varphi_1\rho \,\wedge\, \varphi_2\,]\!]}$$

where ρ is a renaming substitution and $s' \notin Var$.

Narrowing

$$\frac{s \approx t\,[\![\,\varphi_1\,]\!] \qquad u[s'] \not\approx v\,[\![\,\varphi_2\,]\!]}{u[t\rho] \not\approx v\,[\![\,s\rho \doteq s' \,\wedge\, s \succ t \,\wedge\, \varphi_1\rho \,\wedge\, \varphi_2\,]\!]}$$

where ρ is a renaming substitution and $s' \notin Var$.

Equation Resolution

$$\frac{u \not\approx v\,[\![\,\varphi\,]\!]}{\Box\,[\![\,u \doteq v \,\wedge\, \varphi\,]\!]}$$

If EQ is a set of equations[1] and disequations, let $Clos(EQ)$ represent the closure of EQ under the Basic Completion inference rules. We have the following soundness and completeness result from [12].

[1]Throughout the paper, we assume we are given a set of equations closed under symmetricity.

Theorem 1 *Let EQ be a set of equations and disequations. Let $C = \{\varphi \mid \Box [\![\varphi]\!] \in Clos(EQ)\}$. Then C is a complete set of solutions for EQ.*

3 SOUR Graphs

In this section we show how to create a SOUR graph. An initial SOUR graph is created as the dag representation of a set of equations. From the initial representation a compilation process is performed that creates a schematization of the completed system. The schematization adds new edges to the SOUR graph, but does not add nodes.

A SOUR graph contains nodes which are labelled uniquely by the subterms in the initial set of equations. Each node represents a set of constrained terms. There are two types of edges in a SOUR graph, *subterm edges* and *rewrite edges*. Subterm edges are directed edges labelled by an index number. Some subterm edges are also labelled with a set of pairs, each pair consisting of a subterm edge and a rewrite edge. Rewrite edges are directed edges, some of which are labelled by a set of pairs, each pair consisting of two rewrite edges. Edges that are not labelled with a set of pairs are called *initial* edges. Initially a graph only contains initial edges. As the compilation process is performed, new edges are created due to pairs of existing edges. The sets of pairs for an edge represents the different ways the edge could be created. Therefore, a compiled graph contains a history of its origins. Let u and v be two nodes in the graph. Then there is at most one rewrite edge from u to v. If a rewrite edge exists from u and v, we will call it (R, u, v). If u is labelled by a term with an n-ary top symbol, then for all i such that $1 \leq i \leq n$ there is at most one subterm edge from u to v labelled with index i. If $i < 1$ or $i > n$ then there are no subterm edges from u to v labelled with i. If a subterm edge exists from u to v labelled with index i, we will call it (S, i, u, v). Also, every graph has an additional set of pairs called $Goal$ associated with it, each pair consisting of two subterm edges.

Given a set EQ of equations and disequations, we create an initial SOUR graph in the following way from the subterms of EQ. For every element t of $Sub(EQ)$, there is a node in the graph labelled with t. We create a set of subterm edges $S(EQ)$ such that if $f(t_1, \cdots, t_n) \in Sub(EQ)$, then, for each t_i, there is an initial subterm edge $e \in S(EQ)$ from the node labelled with $f(t_1, \cdots, t_n)$ to the node labelled with t_i. This edge is labelled with index i. We also create a set $R(EQ)$ of rewrite edges such that if $s \approx t \in EQ$, then there is an initial rewrite edge from the node labelled with s to the node labelled with t. There is also an initial rewrite edge from the node labelled with t to the node labelled with s. We add both edges, but in inferences, we will use ordering constraints, which will be unsatisfiable for edges ordered the wrong way. In figures, we will sometimes draw unoriented rewrite edges, since they always appear in both directions. We say that the SOUR graph composed of edges of $S(EQ)$ and $R(EQ)$ is the *initial SOUR graph* of EQ. In Figure 1, we give the initial SOUR graph of $ffx \approx gfx$. This gives us the usual dag representation of a set of equations. One point that is important to understand is that when we define the semantics, only the top symbol of a term labelling a node will be used. The rest of the term was only used to aide us in defining the graph, so in future figures, only the top symbol will be written.

After the initial SOUR graph, is created, we must compile it into a schematization of the completed system by adding edges corresponding to inferences. Before we define how to perform the compilation, we define what it means for two nodes to *top unify*. Two nodes top unify if they represent the same term, ignoring constraints and renamings (which we describe below). Formally, if u and v are nodes, then u and v *top unify* if (i) u or v is labelled by a variable symbol, or (ii) u and v are labelled by the same n-ary function symbol and for all i, $1 \leq i \leq n$, there is an edge e_i from u to some u_i and an edge $e_i{}'$ from v to some v_i such that both e_i and $e_i{}'$ are labelled with index i, and u_i and v_i top unify. In order for terms represented by nodes to unify, the nodes must top unify.

We will define an inference procedure that detects certain patterns in the graph and adds a new edge based on each pattern. We first define a function RUR so that if $e_1 = (R, v_1, v_2)$ and $e_2 = (R, v_3, v_4)$ are edges such that (i) v_1 and v_3 top unify, (ii) v_1 is not labelled by a variable, and (iii) $e_3 = (R, v_2, v_4)$

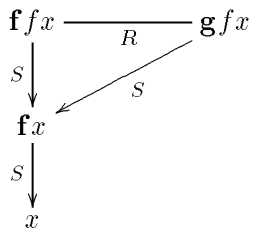

Figure 1: Initial SOUR graph

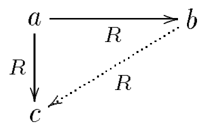

Figure 2: RUR edge

is not an initial edge, then $e_3 = RUR(e_1, e_2)$. We say that e_3 is created by an *RUR transformation.* Creating edge e_3 corresponds to performing a critical pair between the equation represented by e_1 and the equation represented by e_2, at the root of the left hand sides. See Figure 2 to see a critical pair between $a \approx b$ and $a \approx c$. The new edge is represented by a dotted line. After adding the new edge, the graph represents the original equations plus the equation added by performing the critical pair.

Let SUR be a function so that if $e_1 = (S, i, v_1, v_2)$ and $e_2 = (R, v_3, v_4)$ are edges such that (i) v_2 and v_3 top unify, (ii) v_2 is not labelled by a variable, and (iii) $e_3 = (S, i, v_1, v_4)$ is not an initial edge, then $e_3 = SUR(e_1, e_2)$. Then e_3 is created by an *SUR transformation.* Creating edge e_3 corresponds to performing a inference from the equation represented by e_2 into the term represented by e_1, below the root. Figure 4 shows a critical pair from $a \approx c$ into $fa \approx b$. Figure 3 is a more complicated example. In Figure 3, the curved edge going from left to right represents the following inference:

$$\frac{ffx \approx gfx \qquad ffx \approx gfx}{fgfx\rho_1 \approx gfx \, [\![\, ffx\rho_1 \doteq fx \, \wedge \, ffx \succ gfx \,]\!]}$$

More precisely, what we have shown in the figure represents the *skeleton* part of the conclusion of this inference, i.e., everything except for the renaming and the constraint. Later, we will see how the renaming and constraint will be represented, by virtue of the label on the new edge. The conclusion can be simplified to $fgfx\rho_1 \approx gfx \, [\![\, fx\rho_1 \doteq x \,]\!]$

The line that goes from the edge labelled by g to itself represents the following inference:

$$\frac{ffx \approx gfx \qquad fgfx\rho_1 \approx gfx \, [\![\, fx\rho_1 \doteq x \,]\!]}{fggfx\rho_2 \approx gfx \, [\![\, ffx\rho_2 \doteq fx\rho_1 \, \wedge \, ffx \succ gfx \, \wedge \, fx\rho_1 \doteq x \,]\!]}$$

With the addition of these two edges, we have completely schematized the completion of $ffx \approx gfx$, which is an infinite set of equations.

We also define a function *Goal* so that if $e_1 = (S, 1, v, v_1)$ and $e_2 = (S, 2, v, v_2)$ are edges such that (i) v_1 and v_2 top unify, and (ii) v is labelled by a disequation $s \not\approx t$, then $Goal(e_1, e_2)$ is called a *goal transformation.* This corresponds to a possible solution of a goal. When we define the semantics, we will see that this represents a constraint that must be solved to solve the goal.

Now we are ready to show how to compile the graph. If EQ is a set of equations, we define $COMP(EQ)$ to be the graph $G = (V, E, Goal)$ where V is the set of nodes labelled by $Sub(EQ)$, and E and *Goal* are the smallest set of edges and goal transformations such that (i) $S(EQ) \cup R(EQ) \subseteq E$, (ii) if there is an SUR transformation $e_3 = SUR(e_1, e_2)$ for edges $e_1, e_2 \in E$, then $e_3 \in E$ and (e_1, e_2) is in the set of pairs of e_3, (ii) if there is an RUR transformation $e_3 = RUR(e_1, e_2)$ for edges $e_1, e_2 \in E$, then $e_3 \in E$ and (e_1, e_2) is in the set of pairs of e_3, (ii) if there is a *Goal* transformation

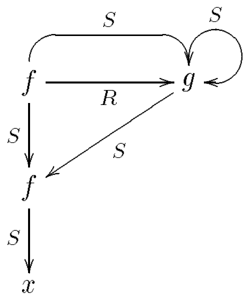

Figure 3: SUR edges

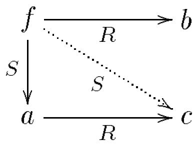

Figure 4: SUR edge

$e_3 = Goal(e_1, e_2)$ for edges $e_1, e_2 \in E$, then $e_3 \in E$ and (e_1, e_2) is in the set of pairs of e_3.

This completely defines the SOUR graph. What is left to do is to define how to interpret the SOUR graph to represent the schematization of a completed set of equations and the solutions to the goal. We also need to say how to use this interpretation to solve the goal constraints in a goal directed fashion. But first we must point out that $COMP(EQ)$ can be computed in polynomial time in the size of the initial set of equations. Clearly the initial graph can be created in polynomial time. There is a simple algorithm to compute $COMP(EQ)$ from the initial SOUR graph. The algorithm is the following. At any point, if there is an SUR transformation $e_3 = SUR(e_1, e_2)$ that has not performed, then we perform it. We perform it by checking if e_3 exists. If it does not exist, we add it with set of term pairs $\{(e_1, e_2)\}$. If it does exist, we add (e_1, e_2) to the set of term pairs. We do the same thing for RUR and Goal transformations. We keep performing these transformations until we have done them all. This algorithm will halt in polynomial time. First, notice that it is possible to calculate if a transition exists and find it in polynomial time. We only need to check all the pairs of subterm edges, pairs of subterm and rewrite edges, and pairs of rewrite edges. The number of such pairs is quadratic in the size of the graph. For each pair, we check the top unification problem. But top unification can, in fact be performed on the whole graph in time quadratic in the size of the graph, if answers are saved as they are calculated, and no calculation is performed twice. This is true, because we only have to top unify all pairs of edges. Therefore, each step of the algorithm is quadratic. Now we must argue that the algorithm halts and only takes polynomially many steps. Note that each step in the algorithm either adds a new edge to the graph or increases the number of edges associated with a pair. Since the number of nodes of the graph does not grow, we cannot add more than quadratically many edges in the initial size of the graph, which is no bigger than the initial size of the problem. Furthermore, each edge can have at most one pair in its set for each pair of edges in the graph. Therefore the size of each set is $O(n^4)$ in the size of the original problem. That shows that the algorithm has only polynomially many steps, therefore it is polynomial. This is a big polynomial, but in practice it will be much smaller.

4 SOUR Graph Semantics

We need a semantics which shows that the initial SOUR graph of a set of equations and disequations represents the initial set of equations. We also need to show that a compiled SOUR graph of a set of equations represents a schematization of the completed set of equations, and a compiled SOUR graph of a set of equations and disequations represents the schematization of the completed set plus a set of recursive syntactic unification and ordering constraints representing possible solutions of the goal.

We define the semantics of nodes, and subterm and rewrite edges, inductively, in terms of each other. Each node and edge has a set of functions associated with it, which can be applied to certain types of arguments. Let e be an edge (or node). Then F_e will be the set of functions associated with

e. Each function in F_e will have the same type of inputs and output. If the functions in F_e take inputs of type $T_1, \cdots, T_n$, then we define $N_e = \{F(n_1, \cdots, n_p) \mid n_i \in T_i \text{ for all } i, 1 \le i \le p\}$. Functions will be written in typed lambda calculus notation. Therefore, each $n \in N_e$ is a typed lambda calculus term, and $Nf(n)$ is the normal form reached by reducing all the radices in n. Also, let T_r be the set of all fresh renamings.

If v is a node, then each member of N_v will be a constrained term. If e is a subterm edge, then each member of N_e will be a triple (φ, ρ, t), where φ is a constraint, ρ is a renaming, and t is a constrained term. We consider Con, Ren and $Term$ to be functions applied to members of N_e which select the corresponding field. If e is a rewrite edge, then each member of N_e will be of the form $(\varphi, \rho_1, \rho_2, t_1, t_2)$, where φ is a constraint, ρ_1 and ρ_2 are renamings, and t_1 and t_2 are constrained terms. We consider Con, Ren_1, Ren_2, $Term_1$ and $Term_2$ to be functions applied to members of N_e which select the corresponding fields.

To understand the complicated functions we are about to describe, we note that for each node v, the normal forms of all members of N_v represent terms whose dag representations are rooted at the node. For an edge e, the normal forms of the members of N_e represent the the constraints, renamings and terms that are built up by inferences. For the case of subterm edges, these apply to the final node of the edge. For rewrite edges, terms and renamings apply to both nodes. There is one constraint to apply to both nodes.

Let v be a node labelled by a p-ary function symbol f and $i_1, \cdots, i_p$ be integers.[2] Then $F_v = \{\lambda n_1 : N_{e_1}, \cdots, n_p : N_{e_p}.Term(v, i_1, \cdots, i_p) \mid$ for all j, e_j is an edge leading from v to some v_j with index $j\}$, where $Term(v, i_1, \cdots, i_p) = f(Term(n_{i_1})Ren(n_{i_1}), \cdots, Term(n_{i_p})Ren(n_{i_p}))\, [\![\bigcup_{1 \le j \le p} Con(n_{i_j})]\!]$.

For the rest of the definitions, let i, j, k be integers. Let $e = (S, l, u, v)$ be an initial subterm edge. Then $F_e = \{\lambda n : N_v.List(e, i, j, k)\}$, where $List(e, i, j, k) = (\top, id, n)$. Let $e = (R, u, v)$ be an initial rewrite edge. Then $F_e = \{\lambda n_1 : N_u, n_2 : N_v.List(e, i, j, k)\}$, where $List(e, i, j, k) = (\top, id, id, n_1, n_3)$.

Let $e = (S, l, u, v)$ be a non-initial subterm edge. Then $F_e = \{\lambda n_1 : N_{e_1}, n_2 : N_{e_2}, \rho_k : T_r.List(e, i, j, k)$ $\mid (e_1, e_2)$ is a pair in the pair set of $e\}$, where $List(e, i, j, k) = (Con(e, i, j, k), Ren(e, i, j, k), Term(e, i, j, k))$, such that $Con(e, i, j, k) = Con(n_i) \wedge Con(n_j)\rho_k \wedge Term_1(n_j)Ren_1(n_j)\rho_k \doteq Term(n_i)Ren(n_i) \wedge Term_1(n_j)Ren_1(n_j) \succ Term_2(n_j)Ren_2(n_j)$, and $Ren(e, i, j, k) = Ren_2(n_j)\rho_k$, and $Term(e, i, j, k) = Term_2(n_j)$.[3]

Let $e = (R, u, v)$ be a non-initial rewrite edge. Then $F_e = \{\lambda n_1 : N_{e_1}, n_2 : N_{e_2}, \rho_k : T_r.List(e) \mid (e_1, e_2)$ is a pair in the pair set of $e\}$, where $List(e, i, j, k) = (Con(e, i, j, k), Ren_1(e, i, j, k), Ren_2(e, i, j, k),$ $Term_1(e, i, j, k), Term_2(e, i, j, k))$, such that $Con(e, i, j, k) = Con(n_i) \wedge Con(n_j)\rho_k \wedge$ $Term_1(n_j)Ren_1(n_j)\rho_k \doteq Term_1(n_i)Ren_1(n_i) \wedge Term_1(n_i)Ren_1(n_i) \succ Term_1(n_j)Ren_1(n_j) \wedge$ $Term_1(n_j)Ren_1(n_j) \succ Term_j(n_j)Ren_2(n_j)$, and $Ren_1(e, i, j, k) = Ren_2(n_i)$, and $Ren_2(e, i, j, k) = Ren_2(n_j)\rho_k$, and $Term_1(e, i, j, k) = Term_2(n_i)$, and $Term_2(e, i, j, k) = Term_2(n_j)$.

If $e = Goal$, then $F_e = \{\lambda n_1 : N_{e_1}, n_2 : N_{e_2}.List(e, i, j, k) \mid (e_1, e_2)$ is a pair in the pair set of $e\}$, where $List(e, i, j, k) = Con(e, i, j, k)$, such that $Con(e, i, j, k) = Con(n_i) \wedge Con(n_j) \wedge Term(n_i)Ren(n_i)$ $\doteq Term(n_j)Ren(e_j)$.

For a graph $G = (V, E, Goal)$, let E_r be the set of all rewrite edges in R. We define $Eqs(G)$, $Deqs(G)$ and $Sols(G)$ to represent all equations, disequations and goal solutions given by G, according to the semantics. Formally, $Eqs(G) = \{Nf(s) \approx Nf(t) \mid (\varphi, \rho_1, \rho_2, s, t) \in N_e \text{ for some } e \in E_r\}$. Also, $Deqs(G) = \{Nf(n) \mid n \in N_v \text{ for some } v \in V \text{ where } top(v) \text{ is } \not\approx\}$. Additionally, $Sols(G) = \{Nf(n) \mid n \in N_e \text{ and } e = Goal\}$. The following are straightforward consequences of the definitions.

Theorem 2 *Let EQ be a set of equations and disequations. Let G be the initial graph of EQ. Then $Eqs(G) \cup Deqs(G) = EQ$.*

[2] It does not not matter which integers we pick for these definitions. They are only necessary so we can perform renamings in the inference rules in the next section.

[3] For this definition and the following two definitions, please refer to the inference rules at the end of section 2. We are only reproducing those constraints.

The next theorem shows that the compilation process schematizes the completed system and finds all the solutions to the goals. It is proved by the relationship between SOUR derivations and Basic Paramodulation derivations (also see [10]).

Theorem 3 *Let EQ be a set of equations and disequations. Let $G = COMP(EQ)$. Then $Sols(G)$ is a complete set of solutions for EQ.*

5 Goal Directed Completion

Now we describe how to solve the *Goal* constraint in $COMP(EQ)$. For this procedure, we save a constraint denoted by φ, a set of variable–type pairs denoted by VT, and a set of term equalities denoted by TE. The constraint is built up by the algorithm. Each member of VT is of the form $n : T$ where n is a variable and T is the type of n. Elements of TE are of the form $Term_1(n) = t$ or $Term_2(n) = t$, where t is a constrained term we are building.

Initially, $\varphi = Con(n)$, $VT = \{n : N_{Goal}\}$, and $TE = \emptyset$. The algorithm expands φ. At each point of the algorithm, we can perform an inference rule called *Definition Expansion* or an inference rule called *Term Expansion*, which modify φ, VT and TE. First we don't-care nondeterministically select an inference rule to perform and a variable from VT to expand. Then we don't-know nondeterministically select an expansion to perform. After the inference we deterministically perform some simplification steps. This is analogous to the SLD-Resolution rule in Logic Programming, where we don't-care nondeterministically select a subgoal to solve and then don't-know nondeterministically select a rule to resolve against the subgoal. As in SLD Resolution, Goal-Directed Completion may be implemented with a breadth-first search or with iterative deepening. A depth first search is not complete, because we could follow a single path and ignore the one that solves the goal.

The *Definition Expansion* rule replaces a variable by its definition:

Definition Expansion

$$\frac{\varphi[n] \qquad VT \cup \{n : N_e\} \qquad TE[n]}{\varphi[List(e,i,j,k)] \qquad VT \cup \{n : N_e, n_i : N_{e_1}, n_j : N_{e_2}\} \qquad TE[List(e,i,j,k)]}$$

where (e_1, e_2) is in the pair set of e, and i, j, and k are fresh numbers.

This mean that we select a variable n appearing in VT and replace n everywhere it appears in the constraint we are solving with renamed versions of the edges that create n. For example, this implies that we replace a field of the form $Con(n)$ by a renamed copy of its meaning. In the term equations, we do the same, effectively replacing $Term(e)$ by a renamed copy of its meaning.

The *Term Expansion* rule expands a term:

Term Expansion

$$\frac{\varphi[Term_m(n)] \qquad VT \cup \{n : N_e\} \qquad TE[Term_m(n)]}{\varphi[Term(v_m, i_1, \cdots, i_p)] \qquad VT \cup \{n : N_e, n_{i_1} : N_{e_1}, \cdots, n_{i_p} : N_{e_p}\} \qquad TE[Term(v_m, i_1, \cdots, i_p)]}$$

where (i), $m \in \{1, 2\}$, (ii) e goes from v_1 to v_2, (iii) for all j, e_j is an edge from v_m to some v_j with index j, and (iv) $i_1, \cdots, i_p$ are fresh numbers.

This says that if there is a term $Term_m(n)$ that appears in the constraint, we can find this node in the graph and expand the term, replacing the new constraints and renamings by renamed copies.

After each inference rule, we also need to perform simplification steps to solve the equational and ordering constraints which appear. For lack of space, we do not present the simplifications here (see Figure 3 in [3]). These transformations convert constraints to a normal form. We can then perform the algorithms given in [3, 12] to solve the constraints.[4] If the constraints are unsatisfiable we fail. If the constraints are satisfiable, we leave them in normal form and continue the goal solving procedure. Each branch of this procedure halts when no more inferences can be performed.

We want to show that these inference rules give us all the solutions to our goal. Let Φ_{EQ} be the set of all constraints φ such that (φ, VT, TE) is in the closure of the above inference rules over $COMP(EQ)$, and no inferences can be applied to (φ, VT, TE). We have the completeness theorem:

Theorem 4 *Let EQ be a set of equations and disequations. Then Φ_{EQ} is a complete set of solutions for EQ.*

The theorem is proved by showing that for each (φ, VT, TE), the set of all Definition Expansion inferences on a given variable, preserves the set of solutions. This is also true for Term Expansion.

We have defined an algorithm which starts with the potential goal solutions and works its way backwards to the original equations. But the inference rules can be modified so that we can start at any point and build from there in finding goal solutions. For instance, we could simulate the Knuth-Bendix algorithm by starting with the original equations and working toward the goal. Perhaps the most interesting algorithm is one that starts with the initial goal equation, instead of the solutions to the goal. This algorithm performs inferences to build up the goal solution, and it also works its way back to the original equations. It is also truly goal directed in that it starts from the goal and works its way backwards toward the original equations.

6 Decidability Results and Future Work

A natural question to show the usefulness of these ideas is to show how SOUR graphs are used to give a decision procedure in cases where Knuth-Bendix completion does not because the completed system is infinite. For instance, consider the equation $ffx \approx gfx$ and goal $a \not\approx b$ given in the introduction. Knuth-Bendix runs forever on this example. Our algorithm compiles the system in polynomial time. After the compilation is finished, there are no goal constraints. Therefore, the Definition Expansion and Term Expansion rules cannot be performed. And the system stops immediately and says that the goal is not true. So this is an example where our algorithm is superior to the Knuth-Bendix algorithm.

It would be even more interesting if we could show that there are some equational theories such that the Knuth-Bendix algorithm never halts for any false goals, but our system halts for all goals. For instance, consider the equational system $ffx \approx gfx$. First note that $\{fg^n fx \approx gg^n fx \mid n \geq 0\}$ is an infinite reduced system representing the same equational theory as $ffx \approx gfx$. Therefore Knuth-Bendix will not halt with the equation $ffx \approx gfx$ and a false goal. We would like to show that it does halt in our system

We believe we have given convincing evidence to show that SOUR graphs and the algorithms in this paper are a powerful technique for proving theories decidable. We are currently examining this, and we will give some results at the Unification Workshop. The constraints in $COMP(EQ)$ provide solutions to the goal that can be represented as recurrence equations on terms. We need to develop techniques to solve such recurrence equations. Other ways of developing this is to use recently developed constraint automata techniques, or possibly to use techniques developed in logic programming, since the constraints we give are similar to constraints that would be created by compiling a set of Horn clauses in a similar way.

References

[1] L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder. Basic Paramodulation. *Information and Computation Vol. 121, No. 2* (1995) pp. 172–192.

[4]We are assuming the ordering is the lexicographic path ordering

[2] J. Chabin, and P. Réty. Narrowing directed by a graph of terms. In *Proc. 4th Int. Conf. on Rewriting Techniques and Applications*, Lect. Notes in Computer Science, vol. 488, pp. 112–123, Springer-Verlag.

[3] H. Comon. Solving Symbolic Ordering Constraints. *International Journal of Foundations of Computer Science* **1** (1990) pp. 387–411.

[4] N. Dershowitz and J. -P. Jouannaud. Rewrite Systems. In J. van Leeuwen editor, *Handbook of Theoretical Computer Science B: Formal Methods and Semantics*, chapter 6, pp. 243–320. North-Holland, Amsterdam, 1990.

[5] N. Dershowitz and G. Sivakumar. Solving goals in equational languages. In *Proc. of the First Intl. Workshop on Conditional and Typed Rewriting Systems*, Lect. Notes in Comput. Sci., vol. 308, (1987) pp. 45–55.

[6] M. Hermann and R. Galbavý. Unification of Infinite Sets of Terms Schematized by Primal Grammars. To appear in *Theoretical Computer Science.*

[7] D. Knuth and P. Bendix. Simple Word Problems in Universal Algebras. In J. Leech, Ed. *Computational Problems in Abstract Algebras*, Oxford, Pergamon Press.

[8] D. Kozen. Complexity of Finitely Presented Algebras. PhD Thesis. Cornell University. 1977.

[9] C. Lynch. Paramodulation without Duplication. In *10th Intl IEEE Symposium on Logic in Computer Science*, (1995) pp. 167-177.

[10] C. Lynch, and P. Strogova. SOUR Graphs for Efficient Completion. CRIN Technical Report (1995).

[11] R. Nieuwenhuis and A. Rubio. Basic Superposition is Complete. In *Proc. European Symposium on Programming*, Lect. Notes in Computer Science, vol. 582, pp. 371–390, (1992) Springer-Verlag.

[12] R. Nieuwenhuis and A. Rubio. Theorem Proving with Ordering and Equality Constrained Clauses. *Journal of Symbolic Computation Vol. 19 No. 4* (1995) pp. 321–352.

[13] K. Rose. Xy-pic Reference Manual 3.0 edition. DIKU, University of Copenhagen. Universitetsparken 1, DK-2100 København Ø. June, 1995.

[14] W. Snyder. *A Proof Theory for General Unification.* Birkhauser Boston, Inc., Boston MA (1991).