A Semantic Matching Algorithm: Analysis and Implementation

Hui Shi*

Universität Bremen, FB3 Informatik, Bibliothekstr. 4, 28359 Bremen, Germany
E-mail: shi@informatik.uni-bremen.de

Abstract. A decision procedure for a class of semantic matching problems was proposed in [3], but it yielded efficiency problems, principally because of redundancies. We present in this paper a new semantic matching algorithm for a restricted class of convergent rewrite systems, its theoretical properties, and an efficient implementation. This class of rewrite systems is particularly interesting for functional programming and for program manipulations. To make the algorithm efficient, two techniques are introduced: *dependency analysis* and *sharing analysis*. The possibility to divide the semantic matching problems into three complexity classes *linear*, *polynomial*, and *decidable* is also an advantage of our approach. A number of test results shed some light on the efficiency of an ML implementation in the theorem prover Isabelle.

1 Introduction

Semantic matching is a useful mechanism. It is well-known that any strategy for finding a complete set of matches with respect to a first-order theory may not terminate, even when the theory is presented as a convergent rewrite system. A decision procedure has been proposed in [3] for a class of convergent rewrite systems. However, it is not always decidable, whether a rewrite system belongs to this class or not, nor efficiency problems have been considered there.

In this paper we modify the class of rewrite systems presented in [3] in three aspects. First, we study a class of *constructor-based* rewrite systems in consideration of the efficience of its semantic matching procedure and the application in functional programming and program manipulations [8]. Second, we treat variable-dropping and variable-preserving rewrite systems uniformly. Third, it is always decidable whether a rewrite system belongs to this class or not. In order to get a decidable semantic matching procedure, we restrict constructor-based rewrite systems to *acyclic* ones.

The following rewrite system defines the maximum function 'max' over the natural numbers with the constructors '0' and 's' and the height function 'ht' over binary trees with the constructors '$empty$' and '$node$' forming a tree from two trees.

$$\begin{array}{llll}
max(0,y) & \rightarrow y & ht(empty) & \rightarrow 0 \\
max(s(x),0) & \rightarrow s(x) & ht(node(x,y)) & \rightarrow s(max(ht(x),ht(y))) \\
max(s(x),s(y)) & \rightarrow s(max(x,y)) & &
\end{array}$$

* Research partially supported by BMBF Project *UniForM*.

Although the algorithm described in Section 3 has some nice properties, such as completeness and termination, after implementing the algorithm in SML/NJ [5] directly, we were very surprised and also disappointed at how inefficient it could be even for some simple matching problems. For instance, it takes more than half an hour to compute all the possible trees of height 4. A combinatorial investigation shows that there are 651 such trees! For practical use, additional techniques to improve the algorithm are indispensable. The techniques we applied are:

– dependency analysis – independent subgoals will be solved independently; and
– sharing analysis – a subgoal will never be solved more than once.

After applying these techniques, the algorithm has been obviously improved: all 651 trees of height 4 are produced within 2 minutes. A number of test results presented in Section 5 give an intuitive view on the performance gain.

An additional benefit of theses techniques is that they make it possible to classify matching problems according to their complexities: linear, polynomial, and decidable. Such a refinement classification has rarely been done in the literature about semantic unification or semantic matching, though it is important for promoting the application of them.

The rest of this paper is organized as follows. Section 2 introduces the basic definitions and notations that will be used in this paper. Section 3 presents a class of convergent rewrite systems, the matching algorithm, and its properties. The improvement techniques are discussed in Section 4. In Section 5 the algorithm will be analysed and some test results will be presented. We conclude in Section 6.

2 Preliminaries

In this section we briefly review the relevant basic notations, terminology, and results. For surveys of this area, see [2, 7].

Terms are constructed from a given set of function symbols and variables. We normally use s, t, l, and r for terms, and x, y, z for variables. A *ground* term is one containing no variables. A term is said to be *linear* if no variables occur more than once. The *set of variables* occurring in a term t is denoted by $\mathcal{V}(t)$.

A precise formalism for describing a subterm is obtained through the notion of *occurrence*, which describes the path from the root to the subterm. We just briefly review the notation, details can be found in [2]. The set of all occurrences in a term t is denoted by $\mathcal{O}(t)$. The letters p and q stand for occurrences. The root occurrence is ε. The *subterm* of t at occurrence p will be denoted as $t|_p$. The replacement of a subterm of t at occurrence p with s will be written as $t[s]_p$. The *depth* of a term t is the length of the longest occurrence of t, denoted as $|t|$.

A *Substitution* σ is a function denoted by $\{(x_1, s_1), \cdots, (x_n, s_n)\}$. The *domain* and the *range* of σ are defined as $dom(\sigma) = \{x_1, \cdots, x_n\}$ and $ran(\sigma) = \mathcal{V}(s_1) \cup \cdots \cup \mathcal{V}(s_n)$ respectively. Renamings are special substitutions, which are functions from variables to variables. The composition of two substitutions σ and τ, denoted as $\sigma \circ \tau$, and the restriction of a substitution σ to a set of variables W, written as $\sigma[W]$, are defined as usual: $\sigma \circ \tau\ (x) = \sigma(\tau(x))$ and $\sigma[W]\ (x) = \sigma(x)$ if $x \in W$, x

otherwise. We say σ is more general than τ, denoted as $\sigma \leq \tau$, if there exists some ρ such that $\rho \circ \sigma = \tau$.

A *rewrite rule* is an oriented pair of terms, written as $l \rightarrow r$ such that $\mathcal{V}(r) \subseteq \mathcal{V}(l)$. A *rewrite system* is a set of such rules. A rewrite rule is *left linear* if its left hand side is linear; a rewrite system is called *left linear* if all its rules are left linear. $l \rightarrow r$ is a *variable-preserving* rule if $\mathcal{V}(l) = \mathcal{V}(r)$, otherwise it is *variable-dropping*. Two rules $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ are *overlapping*, if there exists a substitution σ such that $\sigma(\tau(l_1|_p)) = \sigma l_2$ for some occurrence p and renaming τ. A rewrite system $\mathcal{R}$ is *non-overlapping* if any two rules of $\mathcal{R}$ are not overlapping. A function symbol f is said to be a *defined function* (short: *function*) with respect to a rewrite system $\mathcal{R}$ if there is a rule in $\mathcal{R}$ with f as the top-level symbol of its left hand side; if there is no such rule, then f is called a *constructor*. In the sequel, f and g will be used for functions, c for constructors.

Several well-known concepts about rewrite systems like *normal form*, *terminating*, *confluent*, and *convergent* will also be used in this paper, details can be found in [2, 7]. We write $s \rightarrow t$, if s rewrites to t in one step; $s \rightarrow^* t$, if s rewrites to t in zero or more steps; $s \rightarrow^! t$, if $s \rightarrow^* t$ and t is a normal form. In addition, we are only interested in convergent rewrite systems, i.e., every term can be normalized uniquely. A substitution σ is said to be *normalized* if $\sigma(x)$ is a normal form for each $x \in dom(\sigma)$. Only normalized substitutions are considered in this paper.

A *matching pair* is an ordered pair of terms (s, t). A *matching problem* Γ is a set of matching pairs: $\{(s_1, t_1), \cdots, (s_n, t_n)\}$. If all t_i $(1 \leq i \leq n)$ are ground terms, then Γ is called a *ground matching problem*. $\mathcal{V}(\Gamma) = \mathcal{V}(s_1) \cup \cdots \cup \mathcal{V}(s_n)$.

As an abbreviation, we represent lists of terms using a "vector" notation, so that $f(s_1, \cdots, s_n)$ will be written as $f(\overline{s_n})$, and we shall even use $\{\overline{(x_n, s_n)}\}$ for the substitution $\{(x_1, s_1), \cdots, (x_n, s_n)\}$, and $\{\overline{(s_n, t_n)}\}$ for the matching problem $\{(s_1, t_1), \cdots, (s_n, t_n)\}$.

The set $\mathcal{M}$ of substitutions is a *complete set of minimal R-matches for Γ* if the following holds:

- for all $\sigma \in \mathcal{M}$, $dom(\sigma) \subseteq \mathcal{V}(\Gamma)$;
- for all $\sigma \in \mathcal{M}$ and $(s, t) \in \Gamma$, $\sigma(s) \rightarrow_R^* t$;
- if τ is a substitution and $\tau(s) \rightarrow_R^* t$ for every $(s, t) \in \Gamma$, then there is a $\sigma \in \mathcal{M}$ such that $\sigma \leq \tau$; and
- for any two substitutions σ, $\sigma' \in \mathcal{M}$, $\sigma \not\leq \sigma'$.

3 Definitions and algorithm

A rewrite system is *constructor-based* if it is left linear, non-overlapping, and has no defined function symbols appearing in the inner part of the left hand side of any rule. Some of the functional languages primarily based on recursive equations, such as HOPE, ML, and Haskell, fall into this category when restricted to defining first-order non-overlapping functions. Well known results (e.g. [4, 6]) ensure that complete set of matches can always be found for such class of semantic matching problems, but it is not necessarily finite. For example, the matching problem $\{(x - y, s(0))\}$ may have infinitely many matches: $\{(x, s(0)), (y, 0)\}$, $\{(x, s(s(0))), (y, s(0))\}$,

$\cdots$, where '$-$' is defined on the natural numbers in the usual way:

$$x - 0 \to x,\ 0 - y \to 0,\ s(x) - s(y) \to x - y$$

To solve this problem, the concept of *non-decreasing functions with respect to a suitable property* is introduced in [3]. Unfortunately, it is not possible to always decide whether a function is non-decreasing with respect to a property P, even for some simple suitable property like depth. Moreover, according to this definition all functions containing variable-dropping rules, such as the function *len* over lists, are decreasing with respect to depth.

$$len(nil) \to 0,\ \ len(x : x_s) \to s(len(x_s)),$$

where *nil* and : are constructors of lists.

In the following, we are going to introduce a new definition of *non-decreasing functions*, which solves the above two problems.

Definition 1 (Subterm property) A *subterm property* P is a measure of terms, along with a well-founded total ordering $\rhd$ which compares values of P, such that P is strictly larger, under $\rhd$, for terms than for its subterms.

Definition 2 (Non-decreasing function) Let f be a function defined by a constructor-based rewrite system $\mathcal{R}$, P a subterm property. f is said to be *non-decreasing* (with respect to P) (short: *non-decreasing*), if for any $f(\overline{l_n}) \to r \in \mathcal{R}$ the following holds: $P(l_i) \not\rhd P(r)$ for any $1 \leq i \leq n$, and any other function in r is non-decreasing. Otherwise, f is called *decreasing* (with respect to P) (short: *decreasing*).

According to this definition, *len* is a non-decreasing function with respect to depth along with the greater-than function on the natural numbers. The following function *half* is decreasing, since $|s(s(0))| > |s(0)|$.

$$half(s(s(0))) \to s(0),\ \ half(s(s(s(x)))) \to s(half(s(x)))$$

Although the definition of non-decreasing functions has been extended to cover functions with variable-dropping rules, we can still prove the following theorem.

Theorem 1 Let s be a normal form, in which all functions are non-decreasing with respect to some subterm property P. If $\sigma(s) \to^! t$ and for any other $\sigma'(s) \to^! t$, $\sigma' \not\leq \sigma$, then $P(\sigma x) \not\rhd P(t)$ for any $x \in dom(\sigma)$.

Proof We proof the theorem by an induction on the structure of s.

- Let s be a constant or a variable. Then the theorem holds trivially.
- Let s be of the form $c(\overline{s_n})$. Then t must have the form $c(\overline{t_n})$, where each t_i ($1 \leq i \leq n$) is a normal form. By induction assumption, we can prove the theorem.
- Let s be of the form $f(\overline{s_n})$, and $\sigma(s_i) \to^! s'_i$ for all $1 \leq i \leq n$. We prove that $P(s_i) \not\rhd P(t)$ by an induction on rewriting steps.
  - Let $f(\overline{s'_n}) = t$. Then the theorem holds trivially.

- Let $f(\overline{s'_n}) \to t'$ with some rule $f(\overline{l_n}) \to r$. Then there exists a substitution θ such that $\theta(l) = f(\overline{s'_n})$ and $t' = \theta(r)$. If $f(\overline{l_n}) \to r$ is variable-preserving, then $P(\theta(l_i)) \not\triangleright P(\theta(r))$, since f is a non-decreasing function. By induction assumption, the assertion holds. If $f(\overline{l_n}) \to r$ is a variable-dropping rule, the assertion holds by induction assumption and the fact that σ is a minimal substitution such that $\sigma(s) \to^! t$.

□

Now we can define a class of *acyclic rewrite systems*, for which semantic matching is always decidable.

Definition 3 (Acyclic rewrite system) Let P be a subterm property. A constructor-based rewrite system $\mathcal{R}$ is called *acyclic* (with respect to P), if each $f(\overline{l_n}) \to r \in \mathcal{R}$ satisfies the following conditions:

- the top-level symbol of r is a variable or a constructor; and
- there is no function in r nested below any decreasing function with respect to P.

According to this definition, $-$ is not defined by an acyclic rewrite system, since the right hand side of its third rule begins with $-$; nor is the following function *isEq*:

$$isEq(0,0) \to s(0), \;\; isEq(s(x), s(y)) \to s(half(isEq(x,y))),$$

because *isEq* occurs below the decreasing function *half* on the right hand side of the second rule. On the contrary, the rewrite systems defining *max* and *ht*, *len*, and *half* are all acyclic, so are the following two rewrite systems:

$$\begin{array}{llll} 0 + y & \to y & 0 * y & \to 0 \\ s(x) + y & \to s(x + y) & s(x) * 0 & \to 0 \\ & & s(x) * s(y) & \to s(y + (x * s(y))) \end{array}$$

$$\begin{array}{llll} nil +\!\!+ y_s & \to y_s, & (x : x_s) +\!\!+ y_s & \to x : (x_s +\!\!+ y_s) \end{array}$$

Instead of using an existing narrowing strategy, such as basic narrowing or innermost narrowing, we introduce a new narrowing strategy — *structural narrowing.* We use $mgu(t_1, t_2)$ to denote the *most general (syntactical) unifier* of t_1 and t_2.

Definition 4 (Structural narrowing) A non-variable term t is *structurally narrowable* into t' with respect to $\mathcal{R}$, if there is a renaming $l \to r$ of a rule in $\mathcal{R}$ such that, t and l have no common variables and are unifiable, and $t' = mgu(t,l)(r)$. It is denoted as $t \leadsto_{l\to r} t'$. A term that is not structurally narrowable is called *structurally normalized.*

In fact, structural narrowing is a lazy narrowing strategy, which always occurs at the top-level of a term, i.e., considers the term as a whole. We give some examples to show the idea.

$x - y \leadsto_{x_1 - 0 \to x_1} x_1$
$x - y \leadsto_{s(x_1)-s(y_1)\to x_1 - y_1} x_1 - y_1 \leadsto_{s(x_2)-s(y_2)\to x_2-y_2} x_2 - y_2 \leadsto_{s(x_3)-s(y_3)\to x_3-y_3} \cdots$

The first narrowing sequence terminates with x_1 which is structurally normalized, while the second narrowing sequence does not terminate. Structural narrowing is particularly effective for solving matching problems with respect to constructor-based systems, since if t is structurally narrowable with $l \to r$ and $\sigma = mgu(t, l)$, $\sigma(x)$ contains no function symbols for any $x \in dom(\sigma)$.

We present our matching algorithm as a set of transformations for transforming a ground matching problem into an explicit representation of a solution. A pair (x, s) in a matching problem Γ is in *solved form* and x is called a *solved variable*, if x is a variable that occurs neither in s nor anywhere else in Γ. A matching problem is in *solved form* if all its pairs are in solved form. For a given matching problem Γ, the algorithm *succeeds* when there is a sequence of transformations terminating with a matching problem in solved form, say σ, in which case σ is a match of the initial problem Γ, fails when there exist no sequences of transformations that terminate with a matching problem in solved form. In the case of failure, Γ is not matchable. In the sequel, we write $l \to r \in \mathcal{R}$ to denote a renaming of a rule in $\mathcal{R}$, whose variables do not occur in other related terms. It is assumed that terms in a matching problem are always automatically normalized.

(Elimination) finds a partial solution for a variable. If x occurs in Γ,

$$\{(x, t)\} \cup \Gamma \Longrightarrow \{(x, t)\} \cup \sigma\Gamma, \quad \text{where } \sigma = \{(x, t)\}.$$

(Decomposition) breaks a matching pair down into simpler ones. If $c \in \mathcal{C} \cup \mathcal{F}$,

$$\{(c(\overline{t_n}), c(\overline{s_n}))\} \cup \Gamma \Longrightarrow \{\overline{(t_n, s_n)}\} \cup \Gamma.$$

(Narrowing) realizes structural narrowing of a pair. If each t_i $(1 \leq i \leq n)$ contains no function symbols,

$$\{(f(\overline{t_n}), s)\} \cup \Gamma \Longrightarrow \sigma \cup \{(\sigma r, s)\} \cup \sigma\Gamma, \quad \text{where } l \to r \in \mathcal{R}, \sigma = mgu(f(\overline{t_n}), l).$$

(Deletion) removes a matching pair playing no role in solutions. If $x \notin \mathcal{V}(\Gamma)$,

$$\{(t, x)\} \cup \Gamma \Longrightarrow \Gamma.$$

(Abstraction) abstracts a matching pair. If the head of $f(\overline{t_n})|_p$ is a function symbol and $p \neq \varepsilon$,

$$\{(f(\overline{t_n}), s)\} \cup \Gamma \Longrightarrow \{(f(\overline{t_n})[x]_p, s), (f(\overline{t_n})|_p, x)\} \cup \Gamma, \quad \text{where } x \text{ is a new variable.}$$

As usual, $\Longrightarrow^*$ and $\Longrightarrow^+$ denotes the reflexive-transitive and transitive closure of $\Longrightarrow$ respectively. The first three transformations are very similar to a standard semantic matching algorithm. The use of structural narrowing requires two additional transformations (Deletion) and (Abstraction). The rule (Abstraction) ensures the completeness of the algorithm. (Deletion) removes the matching pairs which are introduced by applying variable-dropping rules.

The rule (Narrowing) can not be used to the matching problem $\{((x+y)+z), u)\}$ directly. Applying (Abstraction) to it, we get the new problem $\{(x_1 + z, u), (x + y, x_1)\}$. Now, (Narrowing) can be applied to both pairs. The next example shows the application of the transformation (Deletion). Let $\{(len((x + y) : zs), s(0))\}$ be a matching problem. After applying (Abstraction) and (Narrowing) with the rewrite rule $len(x : xs) \rightarrow s(len(xs))$, we get the new problem $\{(s(len(zs)), s(0)), (x + y, z)\}$. Because the rewrite rule is variable-dropping, so the second pair can be removed by (Deletion) without loss of the completeness.

Theorem 2 For any matching problem Γ over an acyclic constructor-based rewrite system, the algorithm terminates with a complete set of minimal matches of Γ.

Proof The proofs of the termination and the completeness are similar to the proofs in [3]. A complete proof of this theorem can be found in [12]. We present here a proof of minimality. The only two rules to be considered are (Elimination) and (Narrowing). The application of any other rule does not influence the matches.

- Let (x, s) be a matching pair in a matching problem. Then the minimal match of the pair is $\{(x, s)\}$. That is to say, the (Elimination) transformation produces a minimal partial solution.
- Let $(f(\overline{t_n}), s)$ be a matching pair in a matching problem, $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ be two rules that can be applied to narrow it. If $\sigma_1 = mgu(f(\overline{t_n}), l_1)$ and $\sigma_2 = mgu(f(\overline{t_n}), l_2)$, then $\sigma_1 \not\leq \sigma_2$, since $\mathcal{R}$ is a non-overlopping rewrite system.

□

4 Improvement

We present in this section two techniques *dependency analysis* and *sharing analysis* to improve our algorithm.

Dependency analysis is motivated by the partial ordering for first-order unification problems from Martelli and Montanari [9]. One reason to cause our algorithm inefficient is the random selection of matching pairs to apply structural narrowing. Especially, the use of the transformations (Narrowing) and (Abstraction) may transform ground matching problems into non-ground ones. See the following sequence of the (Abstraction) transformations:

$$\{(max(ht(t_1), ht(t_2)), s(s(0)))\} \Longrightarrow^+ \{(max(x, y), s(s(0))), (ht(t_1), x), (ht(t_2), y)\}$$

The right hand side of the transformation is not a ground matching problem, which contains new variables x and y. Solving $(ht(t_1), x)$ at first without knowing x could be very inefficient through many useless narrowing steps. Instead, the pair $(max(x, y), s(s(0)))$, whose right term contains no variables that occur in the left terms of other pairs, should be considered first.

Definition 5 (Solvable matching pair) A matching pair (s, t) is called *dependent on* (s', t') in Γ, denoted as $(s, t) \prec (s', t')$, if $\mathcal{V}(t) \cap \mathcal{V}(s') \neq \emptyset$. A matching pair $(s, t) \in \Gamma$ is *solvable*, if it is not in solved form and there is no $(s', t') \in \Gamma$ such that $(s, t) \prec (s', t')$.

A solvable matching pair can be solved independently. We can treat a solvable pair as a new matching problem, solve it using the matching algorithm, and then substitute its matches into the original problem. For instance, to solve $\{(max(x, y),$

$s(s(0))), (ht(t_1), x),\ (ht(t_2), y)\}$, we shall first solve $\{(max(x, y),\ s(s(0)))\}$. One of the matches is $\{(x, s(s(0))), (y, s(s(0)))\}$. Substituting it into the original problem we get the new problem:

$$\{(x, s(s(0))), (y, s(s(0))), (ht(t_1), s(s(0))), (ht(t_2), s(s(0)))\}$$

We add an additional condition to the transformation (Narrowing) such that it can only be applied to solvable matching pairs.

For proving Theorem 2 the order of applying transformations is unimportant. If we can show that there always exists a solvable matching pair during solving a ground matching problem, unless it is not matchable, we can conclude that the theorem still holds with the additional condition on the transformation (Narrowing). At first, we present some lemmas.

Lemma 1 Let t be any term and l a linear term such that $\mathcal{V}(t) \cap \mathcal{V}(l) = \emptyset$. If $\sigma = mgu(t, l)$, then

- σx is linear for any $x \in \mathcal{V}(t)$;
- $\mathcal{V}(\sigma x_1) \cap \mathcal{V}(\sigma x_2) = \emptyset$ for any x_1 and $x_2 \in \mathcal{V}(t)$.

Proof By an induction on the structure of l.

- If l is a variable, then $\sigma = \{(x, t)\}$, the lemma holds obviously.
- If l is a constant, then $ran(\sigma) = \emptyset$.
- If l has the form $f(l_1, l_2)$ (for simplicity), f is a constructor or a function symbol. t must have the form $f(t_1, t_2)$. Otherwise, they are not unifiable. Let $\sigma_1 = mgu(t_1, l_1)$ and $\sigma_2 = mgu(\sigma_1 t_2, \sigma_1 l_2)$, then $\sigma = \sigma_2 \circ \sigma_1$. Because $\mathcal{V}(l_1) \cap \mathcal{V}(l_2) = \emptyset$, $\mathcal{V}(t) \cap \mathcal{V}(l) = \emptyset$, and $dom(\sigma_1) \cup ran(\sigma_1) = \mathcal{V}(t_1) \cup \mathcal{V}(l_1)$ (see [10]), we have $dom(\sigma_1) \cap \mathcal{V}(l_2) = \emptyset$ and $\sigma_1 l_2 = l_2$. By induction assumption, it is easy to show that σx is linear for any $x \in \mathcal{V}(t)$. Now, we prove $\mathcal{V}(\sigma x_1) \cap \mathcal{V}(\sigma x_2) = \emptyset$ for any x_1 and $x_2 \in \mathcal{V}(t)$. The case of x_1 and $x_2 \in \mathcal{V}(t_1)$ or x_1 and $x_2 \in \mathcal{V}(t_2)$ can be proved directly by induction assumption. If $x_1 \in \mathcal{V}(t_1)$ and $x_2 \in \mathcal{V}(t_2)$, we have $\sigma x_2 = \sigma_2 x_2$ and $\sigma x_1 = \sigma_2(\sigma_1 x_1)$. Since $x_2 \notin \mathcal{V}(\sigma_1 x_1)$, so $\mathcal{V}(\sigma x_1) \cap \mathcal{V}(\sigma x_2) = \emptyset$ by induction assumption.

□

Following from this lemma and using the fact that each constructor-based rewrite system is left linear, the following lemma can be proved by an induction on the steps of transformations.

Lemma 2 Let Γ be a ground matching problem and $\Gamma \Longrightarrow^* \Gamma'$.

- If $(t, s) \in \Gamma'$, then s is linear.
- If $(t_1, s_1), (t_2, s_2) \in \Gamma'$, then $\mathcal{V}(s_1) \cap \mathcal{V}(s_2) = \emptyset$.

Lemma 3 Let Γ be a ground matching problem and $\Gamma \Longrightarrow^* \Gamma'$, then the dependent relation $\prec$ is well-founded on the matching pairs in Γ'.

Proof By an induction on transformations. The theorem holds for the case $\Gamma' = \Gamma$, since Γ is a ground matching problem. Let $\Gamma_1 \Longrightarrow \Gamma_2$ with the transformation (Elimination), (Abstraction), or (Narrowing), since any other rule does not change the dependent relation between matching pairs.

– (Elimination). Let $\sigma = \{(x, t)\}$, (s_1, s_2) and (s_1', s_2') be two pairs in Γ_1. We show that if $(s_1, s_2) \not\prec (s_1', s_2')$, i.e., $\mathcal{V}(s_2) \cap \mathcal{V}(s_1') = \emptyset$, then $(\sigma s_1, \sigma s_2) \not\prec (\sigma s_1', \sigma s_2')$. There are three cases to be considered:
  - If $x \notin \mathcal{V}(s_2)$ and $x \notin \mathcal{V}(s_1')$, then $\mathcal{V}(\sigma s_2) \cap \mathcal{V}(\sigma s_1') = \emptyset$.
  - If $x \in \mathcal{V}(s_2)$, but $x \notin \mathcal{V}(s_1')$, because $\mathcal{V}(t) \cap \mathcal{V}(s_1') = \emptyset$ ((x, t) is solvable), so $\mathcal{V}(\sigma s_2) \cap \mathcal{V}(\sigma s_1') = \mathcal{V}(\sigma s_2) \cap \mathcal{V}(s_1') = \emptyset$.
  - If $x \notin \mathcal{V}(s_2)$, but $x \in \mathcal{V}(s_1')$, because $\mathcal{V}(t) \cap \mathcal{V}(s_2) = \emptyset$ (lemma 2), so $\mathcal{V}(\sigma s_2)$ $\cap\, \mathcal{V}(\sigma s_1') = \mathcal{V}(s_2) \cap \mathcal{V}(\sigma s_1') = \emptyset$.

  That is, $\mathcal{V}(\sigma s_2) \cap \mathcal{V}(\sigma s_1') = \emptyset$.
– (Abstraction). Let $\{(f(\overline{t_n}), s)\} \cup \Gamma \Longrightarrow \{(f(\overline{t_n})[x]_p, s), (f(\overline{t_n})|_p, x)\} \cup \Gamma$ be a transformation step using (Abstraction). For any $(s_1, s_2) \in \Gamma$, $(f(\overline{t_n})|_p, x) \not\prec (s_1, s_2)$, since x is a new variable; if $(s_1, s_2) \not\prec (f(\overline{t_n}), s)$, then $(s_1, s_2) \not\prec (f(\overline{t_n})[x]_p, s)$ and $(s_1, s_2) \not\prec (f(\overline{t_n})|_p, x)$. So, if $\prec$ is well-founded on $\{(f(\overline{t_n}), s)\} \cup \Gamma$, it is also well-founded on $\{(f(\overline{t_n})[x]_p, s), (f(\overline{t_n})|_p, x)\} \cup \Gamma$.
– (Narrowing). The proof is similar to the first case.

□

Theorem 3 Let Γ be a ground matching problem and $\Gamma \Longrightarrow^* \Gamma'$, then Γ' contains at least one solvable matching pair as long as it is not in solved form.

Proof Follows from the above lemma. □

During solving a matching problem, there are often some *similar matching pairs* that are identical under renaming. The matching pairs $(ht(t_1), s(s(0)))$ and $(ht(t_2), s(s(0)))$ are similar in the above new matching problem. Usually, all these matching pairs will be solved independently, because our matching algorithm does not "remember" the matching pairs which have been solved already. The introduction of *sharing analysis* can avoid such redundant computation of similar matching pairs. We simply define a *solution list* for each given matching problem, in which solved pairs with their matches will be recorded. In the above example, the solution list contains matches of $(max(x, y), s(s(0)))$, $(max(x, y), s(0))$, $(max(x, y), 0)$, $(ht(t_1), 0)$, $(ht(t_1), s(0))$ and $(ht(t_1), s(s(0)))$. The number of matching pairs to be solved is now 6 instead of more than 60. Without solution list, solving $(ht(t_1), s(s(0)))$, for instance, contains again to solve $(max(x, y), s(0))$, $(ht(t_1), s(0))$, and $(ht(t_1), 0)$.

Sharing analysis is different from matching modulo commutativity, e.g., $(x + 0, s(0))$ and $(0 + x, s(0))$ are not similar matching pairs. In addition, sharing analysis can sometimes be expensive, if the number of matching pairs is quite large and there are rarely similar pairs. We believe that sharing analysis is useful for solving matching problems containing functions that are defined through commutative functions. ht and $*$ are examples of such functions. They are defined through the commutative functions max and $+$ respectively. We take the matching problem $\{(max(x, y), s(s(0))), (ht(t_1), x), (ht(t_2), y)\}$ as an example. If $\{(x, s_1), (y, s_2)\}$ is a match of the matching pair $max(x, y)$, so is $\{(x, s_2), (y, s_1)\}$, since max is commutative. We should solve the matching problems $\{(ht(t_1), s_1), (ht(t_2), s_2)\}$ and $\{(ht(t_1), s_2), (ht(t_2), s_1)\}$. Applying sharing analysis only one of them will be solved, since their matching pairs are pairwise similar.

5 Algorithm analysis and test results

Dependency analysis also offers a direct way to analyse the algorithm. For simplicity, we only consider narrowing steps. Let $f(\overline{t_n})$ and t be two normal forms, where f is a function symbol and $t_1, \cdots, t_n$ contain no function symbols. To solve the matching pair $(f(\overline{t_n}), t)$, we should consider the following factors:

- The number M_f of rules defining f. There are at most M_f equations that can be used to narrow the matching pair. For instance, $M_{max} = 3$, $M_{ht} = 2$.
- The multiset S_r of the occurrences of function symbols in the right hand side of $l \rightarrow r$. It decides matching pairs in the new problem after applying (Narrowing) with the rule and (Abstraction). For instance,

$$S_y = S_{s(x)} = S_0 = \emptyset,\ S_{s(max(x,y))} = \{max\},\ S_{s(max(ht(x),ht(y)))} = \{max, ht, ht\}$$

- The depth of t, which provides the upper bound on times that the transformation (Narrowing) can be applied. Because after each (Narrowing) transformation, it may follow some (Decomposition) transformation, which reduces $|t|$ at least one. On the other hand, the application of (Abstraction) introduces dependent pairs. Since we only consider functions defined by acyclic rewrite systems, if (t_1, s_1) is dependent on (t_2, s_2), then $|\sigma s_1| \leq s_2$ (see theorem 1), where σ is a minimal match of (t_2, s_2).

Coarsely, any matching problem associated with an acyclic rewrite system is decidable. According to the above analysis, it is possible to make a refinement classification. Let f be a function defined by the set of rules $\{l_1 \rightarrow r_1, \cdots, l_k \rightarrow r_k\}$ in an acyclic rewrite system $\mathcal{R}$, i.e., $M_f = k$. We use $\mathcal{T}_f(m)$ to denote the upper bound of the narrowing steps to solve the matching pair $(f(\overline{t_n}), t)$ with $|t| = m$, which can be computed by the following two equations:

$$\mathcal{T}_f(0) = k \qquad (1)$$
$$\mathcal{T}_f(m+1) = k + \Sigma_{1 \leq i \leq k}(\Pi_{g \in S_{r_i}}(\mathcal{T}_g(m))) \qquad (2)$$

where Σ and Π are the finite addition and multiplication of the natural numbers. If the length of $|t| = 0$, we only need k narrowing steps, since there are at most k rules can be applied, see equation (1). If $|t| = m+1$, in addition to k narrowing steps, we still need to solve all new problems associated with functions in S_{r_i} for each $l_i \rightarrow r_i$ $(1 \leq i \leq k)$, see equation (2).

Definition 6 (Classification) Let f be a function symbol defined by an acyclic rewrite system. If $\mathcal{T}_f(m)$ can be presented as a linear expression, we call f a *linear function*. If $\mathcal{T}_f(m)$ can be presented as a polynomial expression, then f is called a *polynomial function*. Otherwise, f is a *decidable function*.

Surprisingly, there are a lot of useful functions, such as $+$, max, len, and $++$, are linear functions. The use of linear functions is unproblematic in practice. Unfortunately, there are still a lot of functions that are neither linear nor polynomial, such as $*$ and ht. We doubt the existence of a matching algorithm with polynomial complexity. For instance, the number of trees increases exponentially with respect to the height. In practice, solving matching problems with such kind of functions can be very inefficient.

The matching algorithm and its improved versions have been implemented in SML/NL [5]. We use the data structures for terms in the theorem prover Isabelle [11].

A number of tests have been done on a SUN SPARC-Station-20 with 64 megabytes of memory. In order to find out how much performance gain we can get, we have measured the CPU-time-costs in seconds on more than 30 examples. Figure 1 presents some test results. Column $\mathcal{T}_0$ is for the original algorithm, column $\mathcal{T}_1$ for the algorithm with dependency analysis, and column $\mathcal{T}_2$ for the algorithm with both dependency analysis and sharing analysis. '–' stands for tests consuming too much time to finish within 4 hours. '[' and ']', and '0', '1', '2', $\cdots$ are used to present lists and the natural numbers shortly. We can draw the following conclusions, which are in accordance with the analysis at the beginning of this section.

	Matching pairs	$\mathcal{T}_0$	$\mathcal{T}_1$	$\mathcal{T}_2$
1	$(x +\!\!+ [0] +\!\!+ z, [])$	0.22	0.22	0.23
2	$(x +\!\!+ [0] +\!\!+ z, [1, 0, 2])$	0.35	0.34	0.33
3	$(x +\!\!+ y +\!\!+ z, [0, 1, 2, 3])$	0.57	0.58	0.54
4	$(ht(node(x, y)), 1)$	0.26	0.20	0.23
5	$(x * y, 1)$	0.26	0.28	0.26
6	$(x +\!\!+ y +\!\!+ z +\!\!+ w, [0, 1, 2, 3, 4, 5])$	61.10	3.39	3.54
7	$(x +\!\!+ y +\!\!+ z +\!\!+ w, [0, 1, 2, 2, 1, 0])$	53.62	2.97	3.60
8	$(ht(node(x, y)), 3)$	30.24	2.70	2.62
9	$(ht(node(x, y)), 4)$	–	94.57	92.50
10	$(x * y, 12)$	7.64	3.99	1.91
11	$(x * y, 36)$	493.35	91.22	51.11
12	$(x * y, 48)$	–	247.48	139.69

Fig. 1. Test results

- For simple matching problems, all these algorithms need less than one second. This can been see from the first 5 examples.
- In general, the more complicated matching problems are, the better the algorithms with dependency analysis are. The second group contains examples, for which algorithms with dependency analysis get about 93% speedup in execution time.
- For some matching problems the performance gain through both dependency analysis and sharing analysis is very significant. This can be seen from the examples in the third test group. The algorithm with only dependency analysis gets from 50% to 85% speedup in execution time. The algorithm with both dependency analysis and sharing analysis gets about 46% speedup over the algorithm with only dependency analysis.

6 Conclusion

We have presented a new semantic matching algorithm. Our implementation shows that the new algorithm can solve many matching problems efficiently. The implementation will be used in a prototypical program transformation system [8, 12].

Now, we are considering an efficient implementation of a second-order semantic matching algorithm. Because of the complexities of the second-order matching and the semantic matching, efficiency is particularly important to make it practically useful. In addition, the number of matches of a given matching problem may be very large. Usually only a few of them, or even only one of them, are interesting in practice. How to make use of context information or additional restrictions on matching problems during matching process will be considered.

Acknowledgement. We thank the anonymous referees for their corrections and comments.

References

1. Bockmayr, B.: A Note on a Canonical Theory with Undecidable Unification and Matching Problem. In *Journal of Automated Reasoning,* Vol. 3, 379-381, 1987.
2. Dershowitz, N., Jouannaud, J.-P.: Rewrite System. In I. van Leeuwen, editor, *Formal Models and Semantics, Handbook of Theoretical Computer Science, Vol. B,* 243-320. The MIT press, 1990.
3. Dershowitz, N., Mitra, S. and Sivakumar, G.: Decidable Matching for Convergent Systems. In *Proc. of 11th International Conference on Automated Deduction,* LNCS 607, 589-602, 1992.
4. Fay, M.: First Order Unification in an Equational Theory. In *Proc. 4th Workshop on Automated Deduction* (1979), 161-167.
5. Harper, R., MacQueen, D., Miller, R.: Standard ML. LFCS Report Series, ECS-LFCS-86-2. Dept. of Computer Science, University of Edinburgh, 1986.
6. Hullot, J.-M.: Canonical Forms and Unification. In *Proc. 5th Workshop on Automated Deduction* (1980), 318-334.
7. Klop, J. W.: Term Rewriting Systems. In Abramski, S., Gabbay, Dov. M. and Maibaum, T.S.E. (eds): *Handbook of Logic in Computer Science, Chapter 1.* Oxford Science Publications, (1992).
8. Krieg-Brückner, B., Liu, J., Shi, H. and Wolff, B.: Towards Correct, Efficient and Reusable Transformational Developments. In Broy, M. and Jähnichen, S. (ed.): *KORSO, Correct Software by Formal Methods*, LNCS 1009, 1995.
9. Martelli, A. and Montanari, U.: An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems*, Vol 4, No. 2, (1982), 258-282.
10. Middeldorp, A. and Hamoen, E.: Completeness Results for Basic Narrowing. In *Proc. 3rd International Conference on Algebraic and Logic Programming*, LNCS 632, 244-258, (1992).
11. Paulson, Lawrence C.: *Isabelle – A Generic Theorem Prover.* LNCS 828, Springer-Verlag, 1994.
12. Shi, H.: *Extended Matching with Applications to Program Transformation.* Ph.D. thesis, Universität Bremen, 1994.