Fast Algorithms for Uniform Semi-Unification

Alberto Oliart*
Laboratorio Nacional de Informática Avanzada, A.C.
Rébsamen 80, Xalapa, Veracruz, Mexico
aoliart@lania.mx

Wayne Snyder
Boston University Computer Science Department
111 Cummington Street
Boston, MA 02215
snyder@cs.bu.edu

January 16, 2003

Abstract

Uniform semi-unification is a simple combination of matching and unification defined as follows: Given two terms s and t, do there exist substitutions σ and ρ such that $s\sigma\rho = t\sigma$? We present two algorithms for this problem based on Huet's unification closure method, one producing (possibly) non-principal solutions, and one producing principal solutions. For both we provide a precise analysis of correctness and asymptotic complexity. Under the uniform cost RAM model (counting assignment, comparison, and arithmetic operations as primitive) our first algorithm is asymptotically as fast as Huet's method, $O(n\,\alpha(n))$, where α is the functional inverse of Ackermann's function. Under a model which counts assignments and comparisons of pointers, and arithmetic operations on bits, the cost is $O(n^2\,\alpha(n)^2)$. Producing principal solutions is more complex, however, and our second algorithm runs in $O(n^2\,\alpha(n)^2)$ and $O(n^2\,log^2(n\alpha(n))\,loglog(n\,\alpha(n))\,\alpha(n)^2)$ under these two models.

1 Preliminaries

This this section we will present the basic notions necessary for the remainder of the paper, and motivate our study of asymptotically fast algorithms for semiunification. In general we follow the standard notations established by [4], and provide here only a brief review of the most important ideas; readers requiring a more complete introduction to unification are referred to the comprehensive survey [2]

We work with first-order terms $\mathcal{T}(\mathcal{F}, \mathcal{X})$ (denoted by the symbols s, t, u, and v) over a fixed signature $\mathcal{F}$ (denoted by f, g, and h) and set of variables $\mathcal{X}$ (represented by w, x, y, and z). When

*Supported by CoNACyT.

it simplifies the notation, we write unary function symbols without parenthesis, e.g., $h(f(h(x), a))$ would be written as $hf(hx, a)$. By $s[t]$ we indicate that s contains a distinguished subterm t, and we extend this notation to sets of equations. A *substitution* (represented by σ and ρ) is a function from variables to terms almost everywhere equal to the identity. The application of a substitution to a term is represented in the form $s\sigma$. The composition $\sigma\rho$ is the function which maps each x to $(x\sigma)\rho$. We denote the n-fold composition of ρ by ρ^n. If σ_1 and σ_2 are substitutions then we say that $\sigma_1 \leq \sigma_2$ iff there is a substitution θ such that $\sigma_2 = \sigma_1\theta$.

We say that a term s *matches onto* t iff there exists a substitution ρ (a *matcher*) such that $s\rho = t$. Matchers, when they exist, are unique. A *unification problem* is a pair of terms denoted $s =^? t$, and has a solution (or *unifier*) σ iff $s\sigma = t\sigma$. Unifiers are, in general, not unique, and it is well known that a unification problem has a unifier iff it has a *most general unifier*, i.e., a unifier σ such that for any other unifier θ of s and t, $\sigma \leq \theta$.

Semi-unification is a combination of matching and unification formally defined as follows.

Definition 1 *A* semi-unification instance *is a set of inequalities* $\{s_1 \leq^? t_1, \ldots s_n \leq^? t_n\}$ *and is solvable if there exist substitutions* $\rho_1, \ldots, \rho_n$ *and* σ *such that* $s_i\sigma\rho_i = t_i\sigma$ *for each* i*. Since the* ρ_i *can be uniquely generated once* σ *is given, we call* σ *the solution or* semi-unifier *of the instance. A semi-unifier* σ *is* principal *if for any other semi-unifier* σ' *we have* $\sigma \leq \sigma'$*. A semi-unification problem is* uniform *when* $n = 1$.

Semi-unification has applications in term rewriting, type checking for programming languages, proof theory, and computational linguistics (see [9] for a comprehensive list of references). Although it has a simple definition, it has proved remarkably difficult to analyze precisely. In its general form the problem has been shown to be undecidable [9], with an exceedingly difficult proof. The uniform case is decidable, and various authors have given algorithms (see Section 8), however, a careful analysis of the asymptotic complexity of the problem has not been performed. In this paper we present a two decision procedures based on the Huet unification closure method. The first can produce a (possibly non-principal) solution, or fail if no solution exists, in $O(n\,\alpha(n))$ under the uniform cost RAM model (which counts only assignments, comparisons, and additions and subtractions of integers), where n is the number of symbols in the the problem instance. Thus, the complexity of deciding a uniform semi-unification instance and producing a solution is asymptotically equivalent to deciding an instance of standard unification using Huet's unification closure algorithm. However, as we explain below, arithmetic is a significant part of the algorithm, and it is also interesting to consider a model which counts assignment and comparison of pointers and arithmetic operations on bits. Under this model, the cost of the first algorithm is $O(n^2\,\alpha(n)^2)$. However, the first algorithm produces solutions which may not be principal; if principal solutions are desired, we must do additional work, and in particular must perform GCD to keep weights on the links in reduced form. This boosts the cost, and the resulting algorithm runs in $O(n^2\,\alpha(n)^2)$ under the uniform-cost model and $O(n^2\,log^2(n\alpha(n))\,loglog(n\,\alpha(n))\,\alpha(n)^2)$ under the bit-cost RAM model. Both algorithms run in quadratic space.

2 A Naive Decision Procedure by Transformations

The basic idea of the algorithms we present in this paper (due to [8]) is to transform a semiunification problem $s \leq^? t$ into a kind of unification problem on $\varphi(s) =^? t$, where φ is a placeholder for the matching substitution ρ. The unification algorithm used is different from standard unification because in the presence of the placeholder φ one must be careful about applying substitutions (i.e., the Variable Elimination rule) and because the failure condition related to the occurs check is more delicate.

In this section we explore the logical properties of the problem by first considering how the rule-based approach to standard unification (see [2]) must be modified to account for these differences.[1] The algorithm begins with an instance $s \leq^? t$ and either fails (if no solution exists) or terminates with a set of pairs of terms from which a solution (possibly non-principal) may be extracted. By considering how the primitive operations of this rule-based approach may be efficiently performed, we develop the efficient methods presented later.

Before presenting the rules, we must formalize the notion of a placeholder for a matching substitution.

Definition 2 *Let φ be a unary functional symbol not in $\mathcal{F}$. The set of φ-terms is $\mathcal{T}_\varphi = \mathcal{T}(\mathcal{F} \cup \{\varphi\}, \mathcal{X})$. For any term t, $\varphi^0 t = t$ and $\varphi^{i+1} t = \varphi(\varphi^i t)$.*

Let ρ be a substitution over terms in $\mathcal{T}$ and let t be a φ-term. Then

$$t\{\rho\} = \begin{cases} x & \text{if } t = x \\ f(s_1\{\rho\}, \ldots, s_n\{\rho\}) & \text{if } t = f(s_1, \ldots, s_n) \\ \rho^i(t'\{\rho\}) & \text{if } t = \varphi^i t' \end{cases}$$

Thus, we may think of $t \in \mathcal{T}_\varphi$ as a term that has a (single) unknown substitution applied to some of its subterms, and $t'\{\rho\}$ is the term in $\mathcal{T}$ that results when an actual substitution ρ is instantiated for φ and applied to those subterms.

In fact, we are generally only interested in the effect of substitutions on variables, and so we will distribute φ down into subterms using rewrite rules of the form

$$\varphi(f(s_1, \ldots, s_n)) \longrightarrow f(\varphi(s_1), \ldots, \varphi(s_n))$$

for every $f \in \mathcal{F}$. The normal form of a term t under these rules will be denoted by $\widehat{t}$. In other words, we may push the ocurrences of φ down until they either disappear (at constants) or can be pushed no farther (at variables). Clearly, $t\{\rho\} = \widehat{t}\{\rho\}$.

For example, if $t = \varphi(f(x, \varphi(f(y, z))))$ and $\rho = \{x \mapsto a, y \mapsto hy\}$, then

$$t\{\rho\} = f(a, f(hhy, z))$$

[1] The rules presented in this section are essentially the same as those in [8], however, they are worth presenting again, in the interests of being relatively self contained.

and

$$\widehat{t} \;=\; f(\varphi x, f(\varphi^2 y, \varphi^2 z)).$$

To ensure termination, we will also need a well-founded ordering on φ-terms.

Definition 3 *Let $>$ be a fixed total ordering on $\mathcal{V}ar(s,t)$. Then for any φ-terms $\varphi^n s$ and $\varphi^m t$, $\varphi^n s > \varphi^m t$ iff*

1. *$s, t \in \mathcal{X}$ and $s > t$, or*
2. *$s, t \in \mathcal{X}$ and $t = s$ and $n > m$, or*
3. *$t \in \mathcal{X}$ and $s \notin \mathcal{X}$.*

This ordering is not total, but will suffice for termination of the rule set, which we now define.

Definition 4 The set of rules S *Let E be a set of equations on φ-terms, let s, t, and u be φ-terms, and $\succ$ be a total ordering on $\mathcal{V}ar(s,t,u)$.*

Push*:*

$$E \cup \{\varphi(f(s_1,\ldots,s_n)) \stackrel{?}{=} t\} \Rightarrow E \cup \{f(\varphi(s_1),\ldots,\varphi(s_n)) \stackrel{?}{=} t\}$$
$$E \cup \{s \stackrel{?}{=} \varphi(f(t_1,\ldots,t_n))\} \Rightarrow E \cup \{s \stackrel{?}{=} f(\varphi(s_1),\ldots,\varphi(s_n))\}$$

Decompose*:*

$$E \cup \{f(s_1,\ldots,s_n) \stackrel{?}{=} f(t_1,\ldots,t_n)\} \Rightarrow E \cup \{s_1 \stackrel{?}{=} t_1, \ldots s_n \stackrel{?}{=} t_n\}$$

Transitivity*:*

$$E \cup \{s \stackrel{?}{=} t,\ t \stackrel{?}{=} u\} \Rightarrow E \cup \{s \stackrel{?}{=} u,\ s \stackrel{?}{=} t,\ t \stackrel{?}{=} u\}$$

Orient*:*

$$E \cup \{s \stackrel{?}{=} t\} \Rightarrow E \cup \{t \stackrel{?}{=} s\} \text{ if } t \succ s$$

Substitute*:*

$$E \cup \{s[\varphi^n x] \stackrel{?}{=} t,\ \varphi^n x \stackrel{?}{=} u\} \Rightarrow E \cup \{s[u/\varphi^n x] \stackrel{?}{=} t,\ \varphi^n x \stackrel{?}{=} u\}$$

Fail*:*

$$E \cup \{f(\ldots) \stackrel{?}{=} g(\ldots)\} \Rightarrow \perp \text{ if } f \neq g$$

$$E \cup \{\rho^n x \stackrel{?}{=} f(\ldots,\rho^m x,\ldots)\} \Rightarrow \perp \text{ if } n \leq m$$

By $\cup$ we mean *multiset* union and by $s[u/\varphi^n x]$ we mean that u is substituted for an occurrence of the term $\varphi^n x$ in s.

We now define what it means for a set of equations on φ-terms to have a solution.

Definition 5 *A set of equations on φ-terms $E = \{s_1 =^? t_1, \ldots, s_n =^? t_n\}$ has a solution if there exist substitutions σ and ρ such that $s_1\sigma\{\rho\} = t_1\sigma\{\rho\}$, ..., $s_n\sigma\{\rho\} = t_n\sigma\{\rho\}$. We say that σ is a solution to E. The set of all solutions of E is denoted by $Sol(E)$.*

A set of equations on φ-terms which is not a failure set, but which no rule applies, is called a *solution set*. The following result can be easily shown (see [8]).

Theorem 1 *Let $E = \{\varphi(s) =^? t\}$. If s and t are semi-unifiable, then any application of the above rules to E will result in a solution set; otherwise any application of the rules will result in $\perp$.*

We illustrate how this naive decision procedure works with two examples.

Example 1: $f(x, f(y, z)) \leq^? f(f(z, x), x)$
We define the ordering among variables as $z > y > x$.

The algorithm starts with the equation

$$\varphi(f(x, f(y, z))) \stackrel{?}{=} f(f(z, x), x),$$

which becomes, after pushing on the left side,

$$f(\varphi(x), f(\varphi(y), \varphi(z))) \stackrel{?}{=} f(f(z, x), x).$$

After decomposition we get the equations

$$\varphi(x) \stackrel{?}{=} f(z, x), \quad x \stackrel{?}{=} f(\varphi(y), \varphi(z)).$$

Substituting in first equation using the second and then pushing on the left, we obtain:

$$f(\varphi^2(y), \varphi^2(z)) \stackrel{?}{=} f(z, f(\varphi(y), \varphi(z))), \quad x \stackrel{?}{=} f(\varphi(y), \varphi(z)).$$

Finally, after applying Push, Decompose, and Orient, we get

$$x \stackrel{?}{=} f(\varphi(y), \varphi^3(y)), \quad \varphi^4(y) \stackrel{?}{=} f(\varphi(y), \varphi^3(y)), \quad z \stackrel{?}{=} \varphi^2(y).$$

No more rules apply and hence the two terms are semi-unifiable.

Example 2: $g(f(x, y), f(y, z)) \leq^? g(z, x)$
After pushing on the left we have the equation

$$g(f(\varphi(x), \varphi(y)), f(\varphi(y), \varphi(z))) \stackrel{?}{=} g(z, x).$$

After applying Decompose and Orient, we obtain:

$$z \stackrel{?}{=} f(\varphi(x), \varphi(y)), \quad x \stackrel{?}{=} f(\varphi(y), \varphi(z)).$$

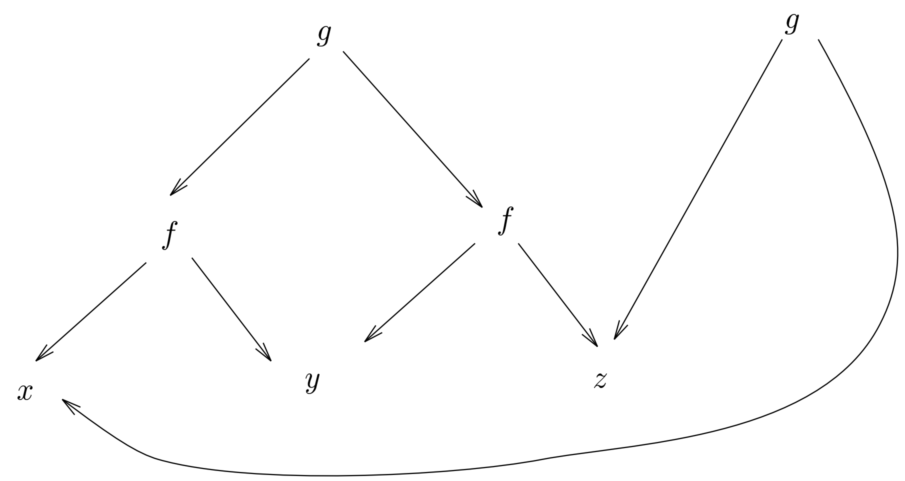
Using the first equation to substitute into the second, and pushing on the right side, we have:

$$z \stackrel{?}{=} f(\varphi(x), \varphi(y)), \quad x \stackrel{?}{=} f(\varphi(y), f(\varphi^2(x), \varphi^2(y)))$$

and by virtue of the second equation this reduces to $\perp$. Hence the terms are not semi-unifiable.

3 A Fast Decision Procedure

The method presented in section 2 is obviously not very efficient: duplication of variable occurrrences leads to duplication of terms, substitution is time-consuming, and the best order in which to apply the rules is not clear. As in the case of standard unification, we may develop a more efficient algorithm by representing terms as directed acyclic graphs (DAGs) with unique occurrences of variable nodes, and using Union-Find data structures; both are necessary to achieve almost-linear time and both can be translated into the case of semi-unification very naturally.[2] For example, the instance $\Gamma = \{g(f(x,y), f(y,z)) \leq^? g(z,x)\}$ used in Example 2 would have a DAG representation as follows:



We assume that the reader is familiar with this standard data structure for terms, and refer those needing more background to the survey [2].

[2] Using DAGs without Union-Find results in a quadratic upper bound for standard unification [3]; the use of Union-Find to achieve almost linear time is due to [7]. In Section 8 we will explain in some detail why the Huet algorithm is the most natural approach for fast semi-unification. For the application of DAGs and Union-Find to Robinson's original algorithm see [14].

3.1 The Semi-Unification Algorithm Based on Unification Closure

The algorithm presented in this section[3] builds equivalence classes of subterms in DAGs by adding pointers ("class links") between the nodes on the graph; thus, as in the Huet method, equivalence classes of terms to be unified are represented by a tree of pointers, with one term (the "representative") at the root. However, in our case, the algorithm also has to account for the matching substitution (represented by φ). The class links account for the application of the matching substitution with two weights, the source weight and the target weight, which count the number of applications of φ on each side of the directed link in a similar way to the algorithm of chapter 2. A class link of the form $r \xrightarrow{n \quad m} s$ says that the term r is a member of the class whose representative is s, and that $\rho^n(r) = \rho^m(s)$, in other words, a link represents an equation between φ-terms. In addition, the data structure for the links has a Boolean flag (or "mark") that is only used for the proof of correctness of the algorithm. The links are added to the graph (and perhaps later marked), more or less as equations are added and removed in the naive algorithm presented above.

We now present our algorithm in detail. Our graph nodes have the following properties :

1. *class* : indicates the class representative for that node. This contains two weights, a source cost and a target cost. At the beginning it is initialized to point to itself, with both weights equal to 0.
2. *size* : Indicates the size of the class when the node is a class representative.
3. *children* : List of the subterms of the term in the original DAG.
4. *self_loop* : Indicates the presence of a self-loop. It is initialized to false for all nodes in the original DAG.
5. *in_stack* : Indicates whether the node is in the stack used for finding cycles. Initialized to **false**.
6. *processed* : Indicates whether the node has already been processed in the finding of cycles.
7. *cycle_cost* : Indicates the cost of the traversed path when searching for cycles.
8. *func* : The functional symbol in a node. It is *null* if the node represents a variable.

Each link has associated with it a Boolean flag called *mark*, which is set to **false** when it is first created, and may be set to **true** sometime during the algorithm. We say that a link is marked if the *mark* flag is **true**. In addition, we maintain a global list **LSF** which contain the nodes that are involved in self-loops (i.e., have links that start and end at the same node); this is critical in the decision procedure, as we see below.

The algorithm may now be given. Our driver function **SU** first calls function **Semiunify**; if **Semiunify** returns true it then calls function **Cycle**. This function returns true if both **Semiunify** and **Cycle** return true.

[3] This paper is based on the first-author's Ph.D. thesis [13] and the reader is referred to that document for a more extensive development of the two algorithms presented in this paper.

```
boolean SU (term s, term t) {    // s ≤ t ?
    return Semiunify(s, 1, t, 0) AND Cycle(s, t)
}
```

Function **Semiunify** returns true if there are no symbol clashes. As a side effect it adds links to the DAG that represents the two terms. After this, the resulting graph is not necessarily a DAG. The links added by the algorithm to the DAG represent membership in a given class of nodes. To do this it uses the well-known almost-linear *Union-Find* algorithm. The Union function is embedded in the **Semiunify** function, and **Find** is defined below.

```
boolean Semiunify(term s, int n, term t, int m) {

  (n1, m1, s') = Find(s);  // Find link s --n1  m1--> s' to representative       (1)
  (n2, m2, t') = Find(t);  // Find link t --n2  m2--> t' to representative       (2)

  // Check for symbol clash

  if (func(s') != func(t') and both are non-null)
      return false;

  // Determine new path between s' and t'

  (w1, w2) = Getpath((m2, n2), (m, n), (n1, m1));

  // Check for self-loop.

  if (s' == t') {        // Union not necessary, but have self-loop
      if (w1 ≠ w2)
          self_loop(s') = true;
      return true;                                                              (3)
  } else {          // Classes distinct, take union
      // Assume, wlog, that size(s') ≥ size(t'), so s' is new
      // representative. The link to be added here is t' --w2  w1--> s'.
      // The case for size(s') < size(t') is analogous.

      class(t') = (w2, w1, s');
      size(s') = size(s') + size(t');
      if (func(s') == null) and (func(t') != null)
          func(s') = func(t');    // func(t') is new function node
      else if (func(s') != null) and (func(t') != null) {
          // Calculate link between function nodes and push down
          (p1, q1, s') = Find(func(s'));  // s' and t' are unchanged            (4)
          (p2, q2, t') = Find(func(t'));                                        (5)
          (k1, k2) = GetPath(((p1, q1), (w1, w2), (q2, p2)));
          return Sulist(subterms(func(s')), k1, subterms(func(t')), k2);        (6)
      }
```

} }

Function **Sulist** traverses the list of subterms of a DAG node and returns true if **Semiunify** returns true for each subterm.

Function **Getpath** returns the weights for a new link to be added to the graph. It receives the weights for three links, which represent the links to be collapsed. The new link goes from the node that is at the source of the first link to the node that is the target of the last link. Given the pairs of integers $(m_2, n_2), (m, n), (n_1, m_1)$, the pair of weights returned by **Getpath** is (w_1, w_2), where

$$\begin{aligned} w_1 &= max\{m_2, m_2 - n_2 + m, m_2 - n_2 + m - n + n_1\} \\ w_2 &= w_1 - ((((m_2 - n_2) + m) - n + n_1) - m_1) \end{aligned}$$

This corresponds to deriving an equality $\rho^{w_1}(s') = \rho^{w_2}(t')$ from the equalities $\rho^{m_1}(s') = \rho^{n_1}(s)$, $\rho^n(s) = \rho^m(t)$, and $\rho^{m_2}(t') = \rho^{n_2}(t)$. Such an inference is sound according to an appropriate set of rules for equational inference which preserve solvability but not principality of solutions extracted. This is because the set includes a rule for cancellation of ρ; e.g., $s\sigma\rho^2 = t\sigma\rho$ is solvable iff $s\sigma'\rho = t\sigma'$ is, however the solution to the second is larger (in fact it is $\sigma\rho$). For a decision procedure this is not an issue, although in order to extract principal solutions, we will need to modify the graph to recover the original links (see below), which accounts for its higher complexity.

Function **Find**(s) is from the fast Union-Find algorithm, but it also calculates the weights of the new compressed links constructed in a manner similar to GetPath. It returns the class representative of the class of term s together with the weights that correspond to the compressed path from s to the class representative. The function also compresses the path to the representative of any other node that is in the original path from s.

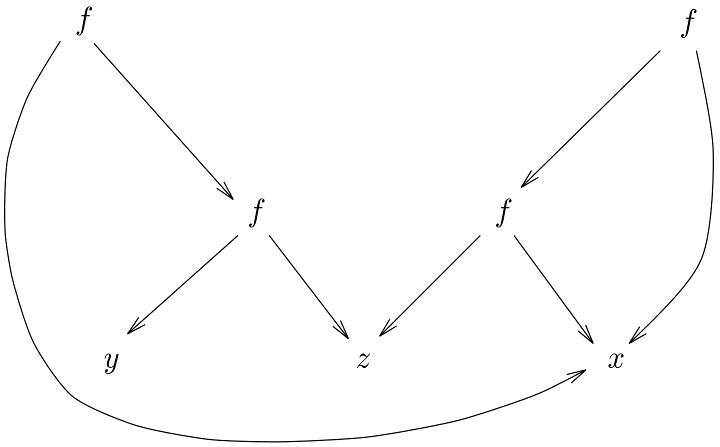
Function **Cycle** checks for bad cycles in the terminal graph which indicate the non-existence of solutions. To find cycles this function uses the well-known linear algorithm, while performing a calculation similar to that used in GetPath to derive equational consequences of the equivalence links followed. If a cycle $x \xrightarrow{n \quad m} x$ is found, where $m \leq n$, then this indicates that the system implies a bad equation $\rho^m(x) = \rho^n(f(\ldots x \ldots))$, which is sufficient for the non-existence of solutions (cf. [8]). Secondly, if a cycle of any kind involving a node with a self-loop is found, then this is also sufficient for non-existence, since this implies the existence of an equational consequence $\rho^p(s) = \rho^q(s)$, which can be used to pump the exponents in the cyclical equation to produce a bad equation. If neither of these conditions holds, it can be shown that the graph has a solution, and so **Cycle** returns true.

4 Examples

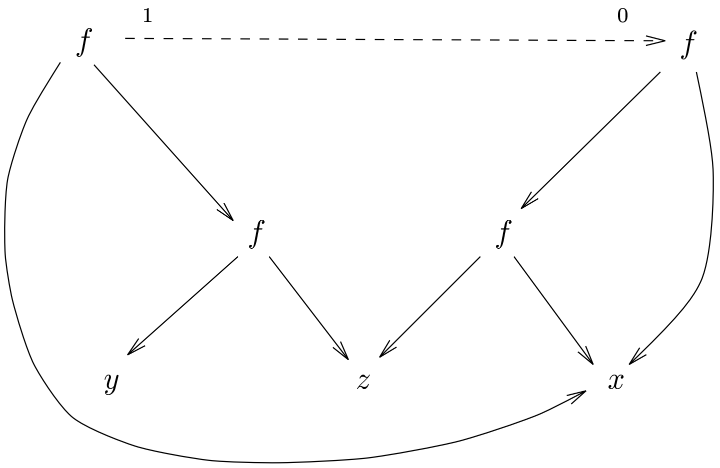
We now demonstrate the decision procedure on the two examples presented previously, plus one more that shows how instances may fail to semi-unify if a self-loop is produced during the algorithm.

4.1 Example 1 Revisited

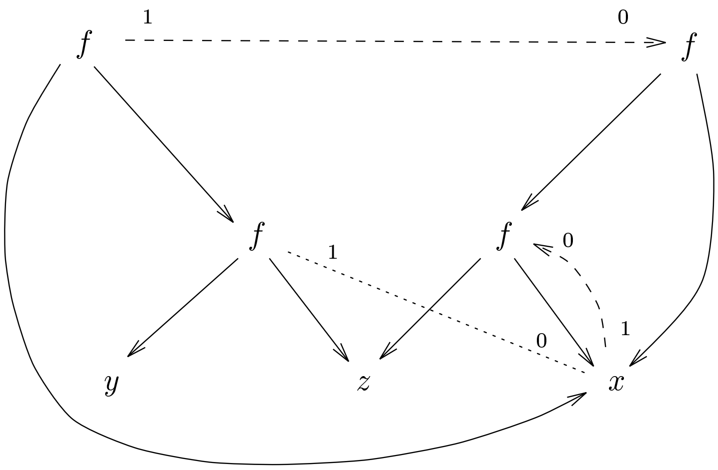
This shows how a problem may have an instance of an occurs check equation and still be semi-unifiable. The DAG representing the problem $\{f(x, f(y,z)) \leq^? f(f(z,x),x)$ is as follows:



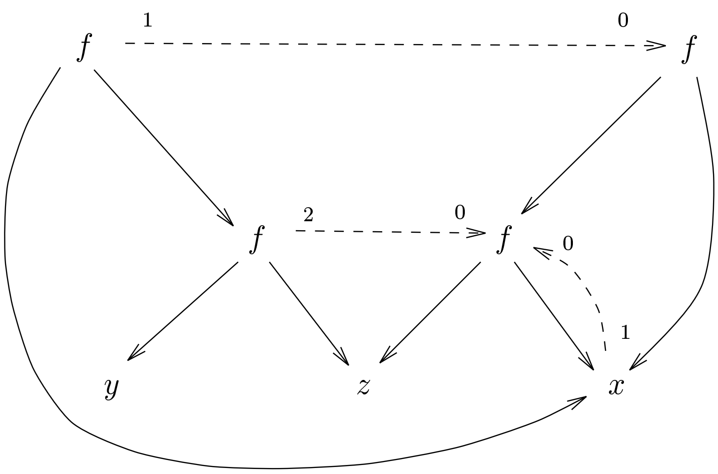
In these diagrams, we will indicate a call to **Semiunify** on two nodes as an equivalence link, denoted by an undirected dotted line, which indicates that the two nodes are about to be put in the same equivalence class, distinguishing it from an actual link between a node and the representative for its class (as before, a dashed arrow). After the first call to **Semiunify** we have an equivalence link between the two top nodes; one of the two nodes is chosen as representative (arbitrarily, since they are both function nodes) and this link turns into a pointer to a class representative:



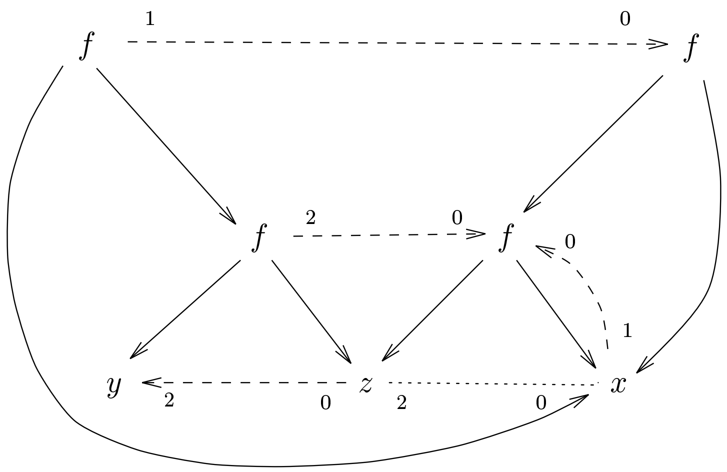
Now the link between the top nodes is pushed down by the recursive calls to **Semiunify** and a link is placed on the left subterms x and $f(z,x)$ and pointed to the latter term as representative. A link is then placed on the right subterms; no weights have yet changed in any of these arcs. Here is a view of the graph before the representative is chosen:



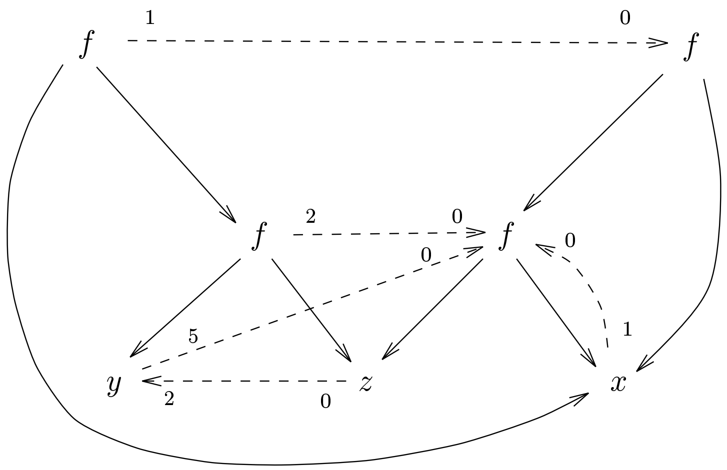
Since x has a pointer to a class representative, then the equivalence link to x must be repointed to this representative, by adding a new arrow, with new weights calculated along the path from $f(y,z)$ to $f(z,x)$:



At this point, the new link between $f(y,z)$ and $f(z,x)$ must be pushed down into the subterms. The link is pushed down to z and y, and since neither has a representative, one is arbitrarily chosen, and then the subterms y and x are linked; here is the diagram before the representative is chosen for this last link:



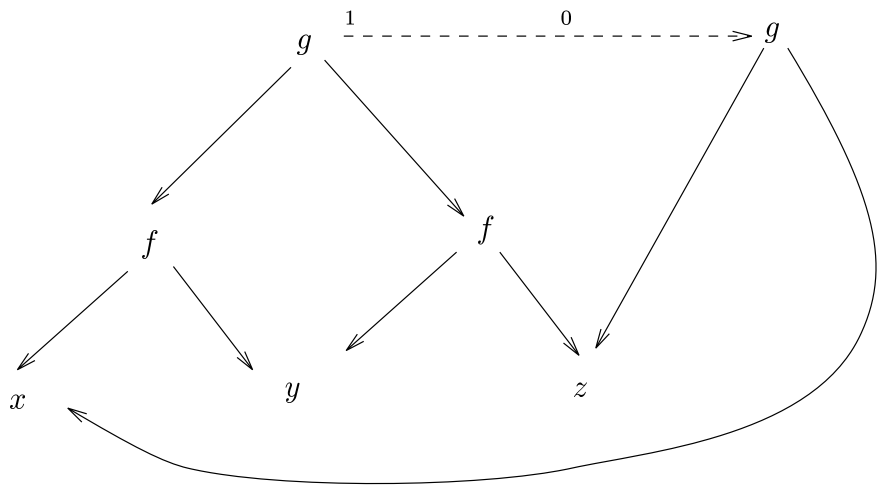
Now the equivalence link between z and x must be repointed so that it joins the representatives of these two terms, and this means building a representative pointer between y and $f(z, x)$; since the class of the latter is larger (with 3 terms) than the class of the former (with 2 terms), the latter is chosen as the new representative. A representative pointer is built, with new weights calculated along the path y, z, x, $f(z, x)$:



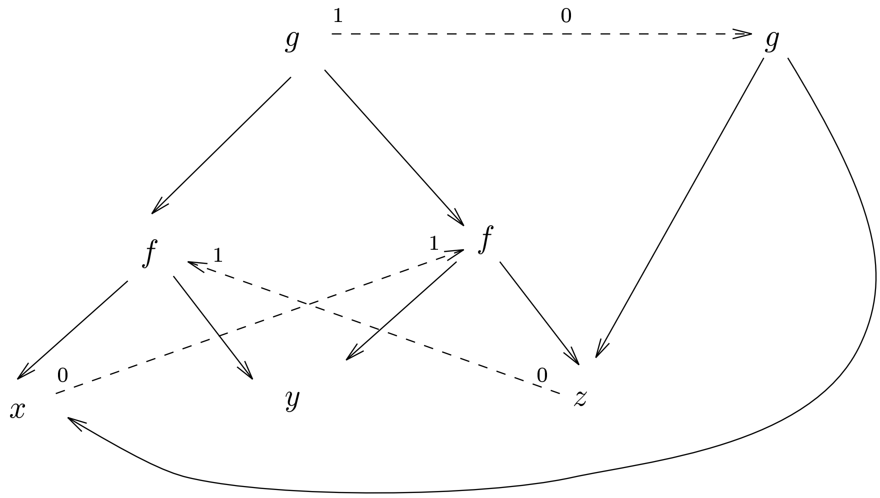
No other arcs are added to the DAG after this, and the algorithm would proceed to check for cycles. In fact, the DAG has a cycle, from the link $x \xrightarrow{1 \quad\quad 0} f(z, x)$. Because the cost on x is greater than the cost on $f(z, x)$ this is not a bad cycle, and therefore the instance has a solution. The decision procedure would return **true**.

4.2 Example 2 Revisited

The DAG for the two terms was given at the beginning of Section 3. The φ is applied to the g node that represents the head of the left term in Γ, so the first link to be added is between those two nodes. This represents the first set of equations in the process, which contains only one equation, namely $\varphi(g(f(x,y),f(y,z))) =^? g(z,x)$. This means that both nodes belong to the same equivalence class, and we arbitrarily choose the right hand side node as the representative of the class. The graph at this point appears as follows:



In the naive algorithm, we would push down the φ using Push and then Decompose the equation; in the graph algorithm the same effect may be obtained by pushing down the link (with its weights) onto the subterms. After this step, we have this graph:



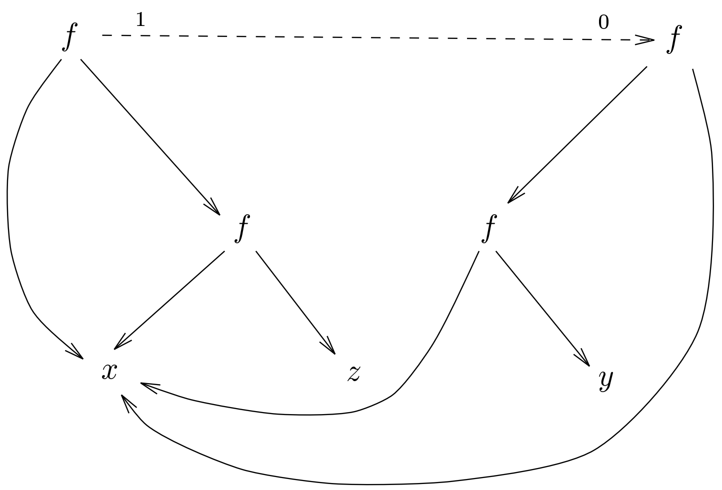
As already noted in Example 2, this particular instance has no solution, and the problem is a occurrence of the second failure condition (a kind of occurs check). This is reflected in the graph as a so-called bad cycle among the links. There are actually two such bad cycles in this graph:

one of them starts at x, and the other one in z. The loop from x is given by following the link $x \xrightarrow{0 \quad 1} f$, then following the subterm link from f to z, then the link $z \xrightarrow{0 \quad 1} f$, and finally back to x through the corresponding subterm link.

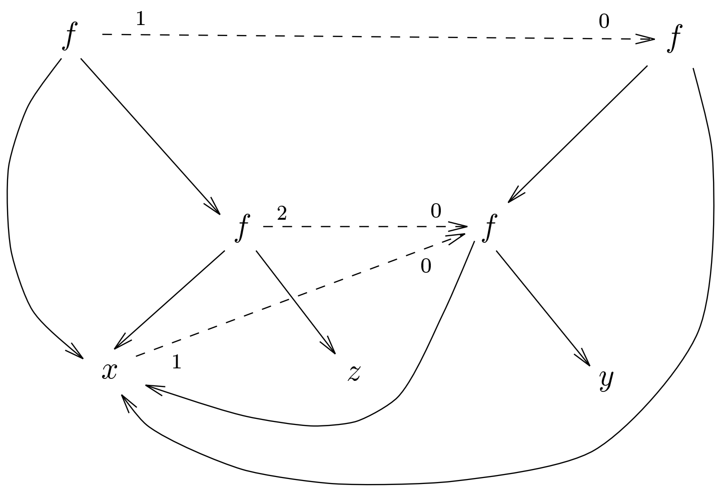
However, not all occurs checks indicate non-semi-unifiability (i.e., when $n > m$ in the second failure condition in our naive algorithm), and not all cycles in the graph are bad. The weights in the links play a role in the cycles, and in this particular case, if we traverse every weighted link, subtracting the target weight (second value) from the source weight (first value), when we return to x we have an accumulated negative value, which means that this is a bad cycle, which means that the instance has no solution. Instead of finding a bad occurs check using a kind of equational rewriting, we do it here using graph traversal and arithmetic on the weights.

4.3 Example 3: A Cycle with a Self-Loop

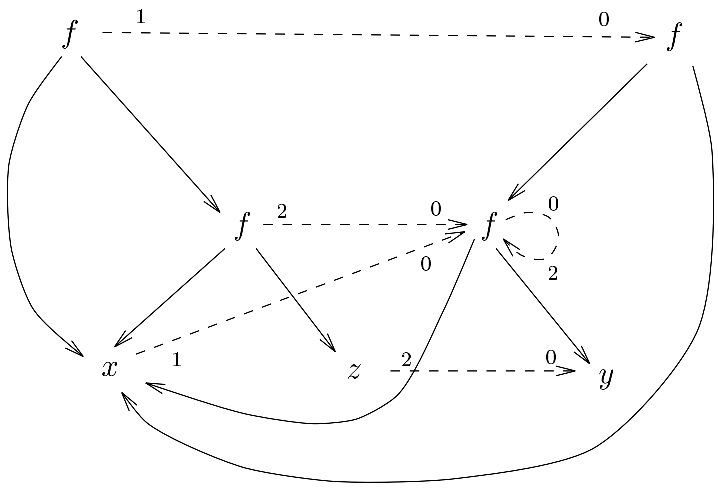
In the previous example we saw an instance that has no solution because of a bad cycle, we now present an example with no solution because of a self-loop. This is in a sense equivalent to a bad cycle, because the self-loop (considered equationally) can be used to rewrite any cycle into a bad cycle by pumping the exponent of one side of the equation. Our instance here is $f(x, f(x, z)) \leq^? f(f(x, y), x)$, corresponding to the following DAG:



After pushing down the links into the subterms and repointing, we obtain:



Pushing down the link between $f(x,z)$ and $f(x,y)$ into the left subterms gives us a self-loop on x, which is repointed and thereby moved up to $f(x,y)$. Pushing down onto the right subterms gives us the completed DAG:



There is a good loop $x \xrightarrow{1 \quad\quad 0} f(x,y)$, however the self-loop participates in this good loop, so that the decision procedure would return **false**, as the instance is not semi-unifiable.

5 Correctness of the Decision Procedure

To prove the correctness of the decision procedure we interpret the class links on the graph as equations over φ-terms. This means that at the end of the procedure we are left with a set of equations, and we show that this final set of equations has a solution if and only if the instance of semi-unification given as input to the algorithm has a solution. We use the term "semi-unification graph" (or, in what follows, simply "graph") for any graph obtained at any point during the procedure described above with an arbitrary instance Γ as input. We first show that the algorithm terminates.

Lemma 1 *If $\Gamma = \{s \leq^? t\}$ is an instance of semi-unification, then the call* $\mathbf{SU}(s,t)$ *terminates.*

Proof: The call to function **SU** produces calls to functions **Semiunify** and **Cycle**, however, the latter is the well known algorithm to find cycles in a graph and obviously terminates.

The only non-trivial case is the recursive function **Semiunify**. We first note that functions **Find** and **Getpath**, which are called on the body of **Semiunify** obviously terminate. Thus the only issue are the recursive calls to **Semiunify** itself. Let us associate with each call to this function the number of equivalence classes in the graph before the call. Note that equivalence classes are only joined, never split. The first call to the function is associated with the number of nodes in the graph, and before each recursive call through **Sulist** in line (6), the (formerly distinct) classes of the terms s and t are joined, reducing the measure. Hence the function terminates. $\square$

Each class link of the form $t \xrightarrow{n \qquad m} s$, where s and t are subterms of the graph, represents an equation on φ-terms of the form $\varphi^n t =^? \varphi^m s$. At each step of the process, a class link is added to the graph, or a class link is substituted with another. Each of these new graphs represents a new set of equations on φ-terms, so the process can be interpreted as building the final set of equations through intermediate sets of equations, starting from a set with only one equation.

Definition 6 *For any graph G, the set of equations represented by G is*

$$\begin{aligned} E(G) = \quad & \{\varphi^n s =^? \varphi^m t \mid \textit{there is an unmarked arc } s \xrightarrow{n \qquad m} t \textit{ in } G\} \cup \\ & \{\varphi^n s =^? \varphi^m s \mid \textit{a self-loop was added to } s \textit{ with weights } m, n\} \end{aligned}$$

Given an instance of semi-unification $\Gamma = \{t \leq^? u\}$, the set of equations associated with the graph after the first call to **Semiunify** is $\{\varphi t =^? u\}$, which obviously has a solution if and only if Γ has one. The graph changes with every call to **Semiunify**, **Find** and **Cycle**. To show that the resulting changes on the corresponding sets of equations preserves solvability (to be defined below), we now define a set of transformations on the sets of equations, show that these transformations preserve solvability, and that changes made to the graph by the above-mentioned functions are equivalent to finite sequences of the transformations on the sets of equations. These transformations are based on the transformations described in section 2. For technical reasons we need to mark some parts of a term in an equation with the symbol $^!$, which does not affect the term in any way. For example, if s is a term, then the marked version is denoted by $s^!$.

Definition 7 *Let E be a set of equations on φ-terms, and let s, t, and u be φ-terms. We define the following sequence of transformations.*

- $E \cup \{s =^? t\} \Rightarrow_0 E \cup \{t =^? s\}$.
- $E \cup \{s =^? t\} \Rightarrow_1 E \cup \{\widehat{t} =^? \widehat{s}\}$.

- $E \cup \{\varphi^n f(s_1, \ldots, s_n) =^? \varphi^m f(t_1, \ldots, t_n)\} \Rightarrow_2$
  $E \cup \{\varphi^n f(s_1, \ldots, s_i^!, \ldots, s_n) =^? \varphi^m f(t_1, \ldots, t_i^!, \ldots, t_n), \varphi^n s_i =^? \varphi^m t_i\}$
  for $1 \leq i \leq n$, *with* s_i, t_i *unmarked.*
- $E \cup \{\varphi s =^? \varphi t\} \Rightarrow_3 E \cup \{s =^? t\}$.
- $E \cup \{s =^? t\} \Rightarrow_4 E \cup \{\varphi s =^? \varphi t\}$.
- $E \cup \{s =^? t,\ t =^? u\} \Rightarrow_5 E \cup \{s =^? u,\ t =^? u\}$.
- $E \cup \{s =^? t,\ t =^? u\} \Rightarrow_6 E \cup \{s =^? t,\ s =^? u\}$.
- $E \cup \{f(s_1^!, \ldots, s_n^!) =^? f(t_1^!, \ldots, t_n^!)\} \Rightarrow_7 E$.
- $E \cup \{s[\varphi^n x] =^? t, \varphi^n x =^? u\} \Rightarrow_8 E \cup \{s[u/\varphi^n x] =^? t, \varphi^n x =^? u\}$.

Furthermore, let $\Rightarrow = \Rightarrow_0 \cup \Rightarrow_2 \cup \ldots \cup \Rightarrow_8$ *and let* $\Rightarrow_i^n$ *denote* n *applications of rule* $\Rightarrow_i$.

We say that $\Rightarrow_i$, $i \in \{1, \ldots, 8\}$, **preserves solvability** *if and only if* $E \Rightarrow_i E'$ *implies* $Sol(E) \subseteq Sol(E')$ *and* $Sol(E) = \emptyset$ *implies* $Sol(E') = \emptyset$.

The sense in which these transformations (corresponding to the actions of our algorithm) preserve solvability is make precise in the next result.

Lemma 2 **(a)** *If* $E \Rightarrow_i E'$, $i \in \{0, 1, 2, 4, 6, 7, 8\}$ *then* $Sol(E) = Sol(E')$.

(b) *For any substitutions* σ *and* ρ, *and* φ*-term*

$$\varphi^i(t), (\varphi^i t)\sigma\{\rho\} = (t\sigma\{\rho\})\rho_i = t\sigma\rho_i\{\rho\}.$$

(c) *If* $\sigma \in Sol(E \cup \{\varphi^{n+i} t =^? \varphi^{m+i} s\}$, *then* $\sigma' = \sigma\rho^i \in Sol(E \cup \{\varphi^n t =^? \varphi^m s\})$.

(d) *Let* $E = E' \cup \{\varphi^n t =^? \varphi^m s\}$. *Then* $E_1 = E' \cup \{\varphi^{n+i} s =^? \varphi^{m+i} t\}$ *has a solution if and only if* E *has a solution.*

(e) *If* $E \Rightarrow_3 E'$ *then* $Sol(E') \subseteq Sol(E)$.

(f) $\Rightarrow_4$ *preserves solvability.*

Proof: The only non-trivial part of this lemma is (c). Clearly we have

$$(\varphi^n t)\sigma'\{\rho\} = t\sigma'\rho^n\{\rho\} = t\sigma\rho^{n+i}\{\rho\} = (\varphi^{n+i} t)\sigma\{\rho\} =$$

$$(\varphi^{m+i} s)\sigma\{\rho\} = s\sigma\rho^{m+i}\{\rho\} = s\sigma'\rho^m\{\rho\} = (\varphi^m s)\sigma'\{\rho\}.$$

Let $\varphi^k u =^? \varphi^j v \in E$. Since $(\varphi^k u)\sigma\{\rho\} = (\varphi^j v)\sigma\{\rho\}$, then

$$(\varphi^k u)\sigma'\{\rho\} = u\sigma'\rho^k\{\rho\} = u\sigma\rho^{k+i}\{\rho\} = v\sigma\rho^{j+i}\{\rho\} = v\sigma'\rho^k\{\rho\} = (\varphi^j v)\sigma'\{\rho\}.$$

□

We now need to show that these transformations to the graph by function **Semiunify** can be explained in terms of the transformations defined above. The changes occur at points (1) - (6), of which (1), (2), (4) and (5) are all equivalent, since they are calls to function **Find**.

Lemma 3 *Assume that graph G has a the following unmarked links:*

$$t \xrightarrow{n_1 \quad m_1} s \xrightarrow{n_2 \quad m_2} u,$$

where u is the representative. Then, the call **Find***(t) generates the graph G_1, such that G_1 is equal to G everywhere except on the part shown, which is changed to:*

$$s \xrightarrow{n_2 \quad m_2} u \xleftarrow{m_3 \quad n_3} t$$

where $n_3 = max\{n_1, n_1 - m_1 + n_2\}$ and $m_3 = n_3 - (n_1 - m_1 + n_2) + m_2$, and $E(G_1)$ has a solution if and only if $E(G)$ has a solution.

Proof: Upon inspection of the code for function **Find** we observe that only one more call to **Find** is issued, specifically **Find**(s), which returns the triplet (u, n_2, m_2). After this, function **GetPath** is called with parameter the list $[(n_1, m_1), (n_2, m_2)]$. This call returns the values $n_3 = max\{n_1, n_1 - m_1 + n_2\}$ and $m_3 = n_3 - (n_1 - m_1 + n_2) + m_2$. Finally, the class link for node t is replaced by one that points to node u, which is the representative, adding the corresponding values as the weights. And this is the only change to the graph, so the graph G_1 is generated and it is equal to G except at the part shown.

To show that $E(G)$ has a solution if and only if $E(G_1)$ has one, we first note that there is a set of equations E_1 such that $E(G) = E_1 \cup \{\varphi^{n_1} t =^? \varphi^{m_1} s, \varphi^{n_2} s =^? \varphi^{m_2} u\}$ and $E(G_1) = E_1 \cup \{\varphi^{n_3} t =^? \varphi^{m_3} u, \varphi^{n_2} s =^? \varphi^{m_2} u\}$. We need to consider two cases: (1) $m_1 < n_2$, and (2) $m_1 \geq n_2$. For the first case, we note that $n_3 = max\{n_1, n_1 - m_1 + n_2\} = n_1 - m_1 + n_2$, and $m_3 = n_3 - (n_1 - m_1 + n_2) + m_2 = m_2$. Therefore we have that

$$E(G) \Rightarrow_4^{(n_2 - m_1)} E_1 \cup \{\varphi^{n_3} t \stackrel{?}{=} \varphi^{n_2} s, \varphi^{n_2} s \stackrel{?}{=} \varphi^{m_2} u\} \Rightarrow_5 E_1 \cup \{\varphi^{n_3} t \stackrel{?}{=} \varphi^{m_3} u, \varphi^{n_2} s \stackrel{?}{=} \varphi^{m_2} u\},$$

and we have that

$$E_1 \cup \{\varphi^{n_3} t \stackrel{?}{=} \varphi^{m_3} u, \varphi^{n_2} s \stackrel{?}{=} \varphi^{m_2} u\} = E(G_1).$$

Therefore $E(G)$ has a solution if and only if $E(G_1)$ has a solution.

For the second case we have that $n_3 = n_1$ and $m_3 = n_1 - (n_1 - m_1 + n_2) + m_2 = m_1 - n_2 + m_2$. We therefore have that

$$E(G) \Rightarrow_4^{(m_1 - n_2)} E_1 \cup \{\varphi^{n_1} t \stackrel{?}{=} \varphi^{m_1} s, \varphi^{m_1} s \stackrel{?}{=} \varphi^{m_3} u\},$$

and we also have that

$$E_1 \cup \{\varphi^{n_1} t \stackrel{?}{=} \varphi^{m_1} s, \varphi^{m_1} s \stackrel{?}{=} \varphi^{m_3} u\} \Rightarrow_5 E_1 \cup \{\varphi^{n_3} t \stackrel{?}{=} \varphi^{m_3} u, \varphi^{m_1} s \stackrel{?}{=} \varphi^{m_3} u\} \Rightarrow_3^{(m_1 - n_2)} E(G_1).$$

Therefore, since the transformations used preserve solvability, $E(G)$ has a solution if and only if $E(G_1)$ has a solution. □

Lemma 4 *A call to* **Find** *on a node on a graph G produces a graph G_1 such that $E(G)$ has a solution if and only if $E(G_1)$ has a solution.*

Proof: The proof of this lemma is by induction on the number of calls generated by the first call to **Find** using Lemma 3 □

Lemma 5 *Let $\Gamma = \{t \leq^? u\}$ be an instance of semi-unification. Then the call* **Semiunify**$(t, 1, u, 0)$ *returns a graph G_S such that Γ has a solution if and only if $E(G_S)$ has a solution.*

Proof: We prove the following by induction on the total number of calls to the function **Semiunify**:

> Let G be a semi-unification graph, and let t, u be nodes in the graph. Let $E = E(G) \cup \{\varphi^n s =^? \varphi^m t\}$. Then the graph G_S that results from the call **Semiunify**(s, n, t, m) is such that E has a solution if and only if $E(G_S)$ has a solution.

Semiunify calls itself recursively through a call to **Sulist**. Assume that E has a solution.

Base case: The function executes first lines (1) and (2). Let G_2 be the grpah after the execution of (2). By lemma 4 we have that E has a solution if and only if $E(G_2)$ has one. We call s' and t' the respective representatives returned by the call to **Find**. Since only one call is generated, either line (3) or line (4) is executed, but not (5), (6) and (7). If it executes (3) then a self loop with weights w_1 and w_2 is added to $s' = t'$. The resulting graph is G_S, and $E(G_S)$ contains the equation $\varphi^{w_1} s' =^? \varphi^{w_2} s'$.

We observe also that the equations $\varphi^{n_1} s =^? \varphi^{m_1} s'$ and $\varphi^{n_2} t =^? \varphi^{m_2} t'$ are in $E_2 = E(G_2) \cup \{\varphi^n s =^? \varphi^m t\}$. We now show that it is possible to go from E_2 to $E(G_S)$ using $\Rightarrow$. We first observe that

$$w_1 = max\{m_2, m_2 - n_2 + m, m_2 - n_2 + m - n + n_1\}$$

and

$$w_2 = w_1 - (m_2 - n_2 + m - n + n_1) + m_1.$$

We can easily check that $w_1 = max\{w_1', w_1' - w_2' + n_1\}$ and $w_2 = w_1 - (w_1' - w_2' + n_1) + m_1$, where $w_1' = max\{m_2, m_2 - n_2 + m\}$ and $w_2' = w_1' - (m_2 - n_2 + m) + n$. Using a similar argument to the one used in the proof of lemma 3, it is easy to check that E_S can be obtained from E_2 using a combination of $\Rightarrow_3, \Rightarrow_4$, etc. If line (4) is executed, then the argument is basically the same as the previous case, the only difference being that the equation added is not a self loop.

Induction Hypothesis. Assume that the lemma is true for fewer than n calls. To show that the lemma is true for n calls, we notice that the call **Semiunify**(s, n, t, m) does not execute line (3), and it must execute lines (1), (2), (4), (5), (6) and (7). Let G_6 be the graph after the execution of line (6). It is easy to show, using a very similar argument to the one used for the base case, that $E(G_6)$ has a solution if and only if E has one. We just need to show that G_S, the graph after the execution of line (7), which comes after a call to function **Sulist**, is such that $E(G_S)$ has a solution if and only if E has a solution.

Observe that function **Sulist** simply takes two nodes of the graph and calls **Semiunify** on all descendants of both nodes from left to right. Let $s_1, \ldots, s_k$ and $t_1, \ldots, t_k$ be the descendants (subterms) of s and t respectively, i.e., $s = f(s_1, \ldots, s_k)$, and $t = f(t_1, \ldots, t_k)$ for some functional symbol f. We define, for $i \in \{1, \ldots, k\}$ the sets of equations

$$E_i = E(G_{(i-1)}) \cup \{\varphi^n s_i \stackrel{?}{=} \varphi^m t_i, f(s_1^!, \ldots, s_i^!, \ldots, s_k) \stackrel{?}{=} f(t_1^!, \ldots, t_i^!, \ldots, t_k)\},$$

where $G_{(i)}$ is the graph that results after the call **Semiunify**(s_i, n, t_i, m), and $G_{(0)} = G_6$.

By the induction hypothesis, and since each of the calls to **Semiunify** will generate less than n recursive calls, we have that E_i has a solution if and only if $E(G_{(i)})$ has a solution. Using rule $\Rightarrow_7$ we get the result we require. □

Proposition 1 *A set of equations E containing an equation of the form $\varphi^n(x) = t$, where $\widehat{t}$ is in the form $\widehat{t}\,[\varphi^m x]$, $n \leq m$, has no solution.*

Definition 8 *An equation of the form $\varphi^n x = t$, where $\widehat{t}$ has the form $\widehat{t}\,[\varphi^m x]$, $n \leq m$, is called* unsolvable.

Lemma 6 *Let $\Gamma = \{t \leq^? u\}$ be an instance of semi-unification, and let G and G_S correspond to the graphs associated with Γ and the graph resulting from the call to* **Semiunify** *on Γ. Then, if the call to* **Cycle** *on G_S returns false, then there is no solution for G_S.*

Proof: If **Cycle** returns false on G_S, then there is a bad cycle of the form

$$\varphi^{n_1} x_1 = \varphi^{m_1} s_1, \ldots, \varphi^{n_k} x_k = \varphi^{m_k} s_k$$

where $s_i[x_{i+1}]$ for $i \in \{1, \ldots . k-1\}$, and $s_k[x_1]$, or a cycle with a self loop, which is analogous to a bad cycle.

We now show, by induction on k, the number of equations in the cycle, that an unsolvable equation can be deduced from the equations in the cycle. For one equation, $\varphi^{n_1} x_1 = \varphi^{m_1} s_1$ with $s_1[x_1]$, we observe that during the execution of **Cycle** the value $c = cycle_cost(x_1)$ is greater than $c + n_1 - m_1$, which means that $n_1 \leq m_1$ and therefore the equation shown is unsolvable.

We assume the result true for all integers smaller than k, and prove it now for k. The first step is to reduce the equations $\varphi^{n_1} x_1 = \varphi^{m_1} s_1$ and $\varphi^{n_2} x_2 = \varphi^{m_2} s_2$. There are two cases that we have to consider. If $m_1 \geq n_2$, using $\Rightarrow_1$ we can get the equation $\varphi^{n_1} = \widehat{\varphi^{m_1} s_1} = s_1'$. We observe that $s_1'[\varphi^{n_2} x_2]$, and therefore we can apply rule $\Rightarrow_8$, to obtain the equation $\varphi^{n_1} x_1 = s_1'[\varphi^{m_1 - n_2 + m_2} s_2 / \varphi_1^m x_2]$.

We now have $k-1$ equations, and therefore we can conclude, by induction hypothesis, that an unsolvable equation can be derived. If $m_1 < n_2$ we first apply $\Rightarrow_3$ as many times as necessary to obtain the equation $\varphi^{n_1 + n_2 - m_1} x_1 = \varphi^{n_2} s_1$, and then we proceed as above. □

Lemma 7 *Let $\Gamma = \{t \leq^? u\}$ be an instance of semi-unification, and let G and G_S correspond to the graphs associated to Γ and the graph resulting from the call to* **Semiunify** *on Γ. Then, if the call to* **Cycle** *on G_S returns true, then there is a solution for G_S.*

Proof: We show this by constructing a solution for the set of equations $E(G_S)$. We first note that if the equations are oriented in such a way that the class representative is on the right hand side, then we have one equation for each of the variables appearing in Γ. The first step in constructing the solution is to apply rule $\Rightarrow_8$ to the set of equations until it can be applied no more. This is possible since there are no bad cycles in the graph. We also observe that the new set of equations obtained, we call it E_1, is such that $Sol(E_1) = Sol(E(G_S))$.

Once the above procedure is done, we know that no term appearing on the left-hand side of an equation appears on the left hand side of an equation. We now describe the procedure to build the solution. (This procedure is basically the same described by Kapur et. al. in [8]). We start with $\sigma = \rho = \emptyset$.

1. Eliminate all occurrences of terms of the form $\varphi^n(x)$ from the right sides of equations in E_1 as follows: While there is a term of the form $\varphi^n(\varphi(x))$ on the right side of an equation in E_1, make $\rho = \rho \cup \{x \mapsto x'\}$, where x' is a fresh variable, and replace $\varphi(x)$ with x in E_1.

2. Now build σ as follows, for every equation of the form $x = t$, where x is a variable make $x\sigma = t$.

3. Finish building ρ. For every remaining equation of the form $\varphi^n x = t$, we let $\rho = \rho \cup \{x \mapsto x_1, x_1 \mapsto x_2, \ldots, x_{n-2} \mapsto x_{n-1}, x_{n-1} \mapsto t\}$.

It is trivial to check that σ and ρ are a solution to E_1. $\square$

The correctness of the decision procedure now follows directly from lemmas 4, 5, 6 and 7.

Theorem 2 *Let $\Gamma = \{t \leq^? u\}$ be a semi-unification instance. Then the call to* **SU** *on Γ returns true if and only if Γ has a solution.*

5.1 Complexity Analysis

The decision procedure is based on the well known $O(n\alpha(n))$ unification algorithm of Huet, where $\alpha(n)$ is the inverse of Ackerman's function [1], and n is the number of symbols in the semi-unification instance. The central part is played by function **Semiunify** and it is easy to check that if the call is made with both numeric parameters equal to 0, then the algorithm would behave as Huet's algorithm for unification.

To analyze the complexity of the algorithm we count pointer assignments, comparisons of pointers or symbols, and primitive operations on bits. We consider the following points:

- **Semiunify** is called at most n times, where n is the number of symbols in the original equation to be solved;
- A sequence of $O(n)$ calls to Union (implicit in our algorithm) and **Find** can be performed with $O(n\,\alpha(n))$ assignments, comparisons, or additions of two *numbers*, where α is the functional inverse of Ackermann's function [1];

- All other operations add at most a constant number of assignments, comparisons, or additions/subtractions of two *numbers* to each call to **Semiunify**;
- The arithmetic operations of the algorithm may be analyzed as follows: if we start with two numbers of constant size (number of bits), and create a list of $O(m)$ new numbers by addition and subtraction of previous members of the list, we can create numbers of size at most $O(m)$; thus at each step we need to do at most $O(m)$ bit operations, which gives a total cost of $O(m^2)$ (in our algorithm, $m = n\,\alpha(n)$).

This gives us our complexity result.

Theorem 3 *Under the uniform-cost RAM model (counting assignments, comparisons, additions and subtractions), a call to* **Semiunify**(s,j,t,k)*, where the combined size of s and t is n symbols, costs $O(n\,\alpha(n))$. For a RAM model counting assignments and bit operations, the cost is $O(n^2\,\alpha(n)^2)$ assignments, comparisons, or bit operations.*

Note that a Union-Find problem can be reduced to a Semi-Unification problem in a trivial way, which shows that we cannot improve the $O(n\,\alpha(n))$ bound unless we can do the same for Union-Find, which is unlikely. What is interesting about this result is that the purely symbolic operations cost no more than for standard unification ($O(n\,\alpha(n))$; the dominate cost is for the arithmetic on weights. In Section 8 we compare this result with previous algorithms, which do not closely analyze the cost of the arithmetic.

6 Solution Extraction

6.1 An Example of Solution Extraction

The decision procedure described in Section 3 does not give enough information to construct the principal semi-unifier for a given instance. Consider the instance of the example in Section 4.1. In this case, the links added by the algorithm to the original DAG are $x \xrightarrow{1 \qquad 0} f(z,x)$, $y \xrightarrow{5 \qquad 0} f(z,x)$ and $z \xrightarrow{0 \qquad 2} y$. Considering these links as equations we can deduce that $x\rho\sigma = f(z,x)$, but this says nothing about $x\sigma$.

In following the execution of the decision procedure for this example we notice that there are three calls to the **Semiunify** function involving variable x. These calls represent links between x and the other terms involved. If an ordering is given to the variables, say $z > y > x$, and links are always assumed to be from greater to smaller, then the links associated with x during execution are $x \xrightarrow{1 \qquad 0} f(z,x)$, $x \xrightarrow{0 \qquad 1} f(y,z)$ and $z \xrightarrow{2 \qquad 0} x$. We can see these links as rewrite rules, and do a reduction. In this case, we can reduce the two links coming out of x, and keep the one with cost 0 on x. This operation would produce a rewrite rule between the two f terms, which can be ignored.

If all these possible links coming out of variables are considered, what we get is a system of rewrite rules that can be used to extract the solution of an instance of uniform semi-unification.

We are interested in obtaining the semi-unifier, since the matching substitution is determined by the semi-unifier.

The links obtained are:

$$z \xrightarrow{2 \qquad 0} x \qquad z \xrightarrow{0 \qquad 2} y$$

$$y \xrightarrow{5 \qquad 0} f(z,x)$$

$$x \xrightarrow{1 \qquad 0} f(z,x) \qquad x \xrightarrow{0 \qquad 1} f(y,z)$$

After simplifying the links from z:

$$z \xrightarrow{0 \qquad 2} y$$

$$y \xrightarrow{5 \qquad 0} f(z,x) \qquad y \xrightarrow{4 \qquad 0} x$$

$$x \xrightarrow{1 \qquad 0} f(z,x) \qquad x \xrightarrow{0 \qquad 1} f(y,z)$$

After simplifying y:

$$z \xrightarrow{0 \qquad 2} y$$

$$y \xrightarrow{4 \qquad 0} x$$

$$x \xrightarrow{1 \qquad 0} f(z,x) \qquad x \xrightarrow{0 \qquad 1} f(y,z) \qquad x \xrightarrow{1 \qquad 0} f(z,x)$$

Finally, after simplifying the links (rules) that correspond to x we have one link (rule) for each variable. The links that have a 0 as the weight on the side of the variable indicate the σ.

$$z \xrightarrow{0 \qquad 2} y \qquad y \xrightarrow{4 \qquad 0} x \qquad x \xrightarrow{0 \qquad 1} f(y,z)$$

Which says that $x\sigma = \rho(f(y,z)),\ y\sigma = y,\ z\sigma = y\rho\rho$.

6.2 The Solution-Extraction Procedure

The algorithm for extracting a principal solution is an extension of the decision procedure. A few more data structures are needed to extract the solution. Weights on the self loops are added, since they also are links between nodes in the graph, and the weights on them are needed to obtain the solution. Self loops on non-variable nodes have to be pushed to the descendant variable nodes. Also, a list of all links encountered during the decision procedure is kept to build σ. We also assume a total ordering $>$ among the variables. The following properties are added to the graph nodes:

1. *slw* : A pair of weights for a self loop. If t is a term, $slw.w1(t)$ denotes the first element of the pair and $slw.w2(t)$ the second.

2. *extract* : A list of links encountered during the decision procedure.

The extraction algorithm starts by pushing the self loops to the variables. It then pushes links between functional symbol nodes that have not been pushed before, which are those in the *extract* property of the nodes. Each pushed link is then removed from *extract*. At the end, each of the links in the **extract** property of a variable is viewed as a rewrite rule. These rewrite rules are reduced until there is only one rewrite rule per variable. These rewrite rules define the semi-unifier. Function **SU** calls function **Extract** when the call to **Semiunify** returns true.

The **Semiunify** function in the decision procedure has to be modified so that it initializes the properties **slw** and **extract** for each node. The change is basically to add to **extract** the link represented by the call. This is done right after the check for a symbol clash. Also, the link between the class representatives is added to the extract property of one of them, making sure that the links are not in the extract property already. The other functions used in the decision procedure of section 3 are the same. To save space, we do not show the modified procedure since it is basically the same as for the decision part.

The function **Extract** receives as a parameter a graph that corresponds to the Semi-Unification instance. It basically processes each variable in the graph until there is only one link associated with each variable, including self loop. This is possible given that function **Semiunify** has determined that there is a solution, and this means that there are no bad loops in the resulting graph. At this point the graph is solved, and the links from the variables indicate which is the semi-unifier σ.

Function **Pushlinks** takes all links between two functional nodes stored in the extract property of one of the nodes and pushes the links to the variables, so that simplification can take place. Function **Pushloops** takes the self loops in the graph and pushes them to the variables. If more than one self loop is found in a node they are simplified in the same way as in function **Semiunify** and only one is pushed.

The function **Simplify_var** takes the list of extraction links of a variable and simplifies it until only one extraction link is left. These links are then used in the extraction of σ (the semi-unifier) and ρ (the matching substitution). The function takes a pair of links and simplifies the one with the larger weight with the one with the smaller weight. This links are treated as rewrite rules.

The code for the function may now be given.

Simplify_var(*term t*)

    // First simplify the links on Extract, and then deal with a self loop.
    // First choose the good link, and then simplify the others.
    // We assume that the links in the extract property are sorted in ascending order using the
    // ordering of integer pairs induced by the normal integer order relation.
    if (*extract* is not empty) **then**
        Let l_1 be the smallest link in extract, with n_1 and m_2 as its costs
        for each link l_2 in *extract*

let n_2 and m_2 be the costs associated to l_2 // $n_1 \leq n_2$

// *The links are* $t \xrightarrow{n_1 \quad m_1} t_1$ *and* $t \xrightarrow{n_2 \quad m_2} t_2$.
// *Where* t_1 *and* t_2 *are terms, and* t *is a variable.*
Remove l_2 from *extract*
Simplify_rule(l_2, l_1, t)
end for
if (t has a *self loop*) **then** // *self-loop, simplify*
Let l be $t \xrightarrow{n \quad m} t_1$, the only link in *extract*
while $(slw.w1(t) \leq n)$ // $slw.w1(t) \geq slw.w2(t)$
$n := (n - slw.w1(t)) + slw.w2(t)$
end while
Let l_1 be a new link $t \xrightarrow{n_1 \quad m_1} t_1$, where
$m_1 := (slw.w1(t) - n) + m$
$n_1 = slw.w2(t)$

if $(n > n_1)$ **then**
Simplify_rule(l, l_1, t)
else
Simplify_rule(l_1, l, t)
Add l_1 to *extract(t)* and remove l.
end if
end if
end if
end Simplify_var

The function **Simplify_rule** simplifies the rewriting rule represented by t_1 using the one represented by t_2.

Simplify_rule$(link\ l_1, link\ l_2, term\ t)$

// *The two links are:* $t \xrightarrow{n_1 \quad m_1} t_1$ *and* $t \xrightarrow{n_2 \quad m_2} t_2$. *Simplification makes a link for* t_1 *and* t_2
// *If the two nodes are distinct functional nodes, then the link must*
// *be pushed to the variables. Otherwise a self-loop must be processed.*
if $(t_1 \neq t_2)$
$m := (n_1 - n_2) + m_2$;
$n := m_1$;
Add link $l = t_1 \xrightarrow{n \quad m} t_2$ to $extract(t_1)$.
if $(func(t_1) \neq null$ and $func(t_2) \neq null)$ // *Both are functional nodes.*
// *The link has to be pushed to the variables and processed,*
// *if there is no other link between the nodes. Otherwise ignore.*
if (there is no link between t_1 and t_2)
PushLink(l); // *Push the links to the variables for processing.*
end if
end if

else if $(t_1 = t_2)$ **then** // *This is a self loop, process*

if $(n \neq m)$ **then**

// If the costs are the same then this self loop must be ignored.
// Otherwise it has to be added if there is no self loop,
// or simplified if there is another self loop on the variable.

if (exists self loop on t_1)
// If there is a self loop then simplify the self loops.

$slw.w1(t_1) := gcd(abs(n - m), abs(slw.w1(t_1) - slw.w2(t_1)))$
if $(m < slw.w2(t_1))$
$slw.w1(t_1) := slw.w1(t_1) + m$
$slw.w2(t_1) = m$
else
$slw.w1(t_1) := slw.w1(t_1) + slw.w2(t_1)$

end if

else *// There is no previous self loop.*
Add a self loop on t_1
$slw.w1(t_1) := max\{n, m\}$
$slw.w2(t_1) := min\{n, m\}$

end if
end if

if (t_1 is a functional symbol) *// If the term is a functional symbol, push self loop*
Pushsubterm$(t_1, slw(t_1))$
end if

end if
end Simplify_rule

6.3 Correctness of Principal Solution Extraction

The proof of correctness of the principal solution extraction procedure is very simlar to the one for the decision procedure of section 3. We have to change the definition of the set of equations represented by the graph.

Definition 9 *For any graph G obtained from a semi-unification instance Γ through a call to function* **SU** *the set of equations represented by G is:*

$$E(G) = \begin{array}{l} \{\varphi^n s =^? \varphi^m t \mid \text{there is an unmarked arc } s \xrightarrow{n \quad m} t \text{ in } extract(s) \cup extract(t)\} \cup \\ \{\varphi^n s =^? \varphi^m s \mid \text{a self-loop was added to } s \text{ with weights } m, n\} \end{array}$$

The main difference between the two procedures is that the principal solution extraction keeps all the links encountered during the algorithm, and therefore the set of solutions is preserved. The set of equations that a graph represents is now given by the unmarked links stored in the extract property of the nodes.

We start by showing that the algorithm terminates. It is obvious that the new function **Semi-unify** ends since it is basically the same as in section 3.

Lemma 8 *A call to function* **Simplify_var** *terminates.*

Proof: The function simply takes all links on the extract property of a node and removes them one by one, until only one is left. The only function calls inside **Simplify_var** are to **Simplify_rule**, which obviously terminates if there are no self loops involved. If there is a self loop involved, the only problem would be to have a cycle preventing termination. This is not possible since the decision procedure would have detected a cycle with a self loop. □

The equation transformations defined in section 5 are now different, since we don't have to deal with the elimination of certain equations from the original set. The transformations are now basically the same as those in section 2. The transformations are:

Definition 10 *Let E be a set of equations on φ-terms, and let s, t, and u be φ-terms.*

- $E \cup \{s =^? t\} \Rightarrow_0 E \cup \{t =^? s\}$.
- $E \cup \{s =^? t\} \Rightarrow_1 E \cup \{\hat{t} =^? \hat{s}\}$.
- $E \cup \{\varphi^n f(s_1, \ldots, s_n) =^? \varphi^m f(t_1, \ldots, t_n)\} \Rightarrow_2$
  $E \cup \{f(s_1, \ldots, s_i^!, \ldots, s_n) =^? f(t_1, \ldots, t_i^!, \ldots, t_n), s_i =^? t_i\}$ *for some* $i \in \{1, \ldots, n\}$ *with* s_i *and* t_i *unmarked.*
- $E \cup \{s =^? t,\ t =^? u\} \Rightarrow_3 E \cup \{s =^? u,\ s =^? t\ t =^? u\}$.
- $E \cup \{f(s_1^!, \ldots, s_n^!) =^? f(t_1^!, \ldots, t_n^!)\} \Rightarrow_4 E$.
- $E \cup \{s[\varphi^n x] =^? t, \varphi^n x =^? u\} \Rightarrow_5 E \cup \{s[u/\varphi^n x] =^? t, \varphi^n x =^? u\}$.

Let $\Rightarrow = \Rightarrow_0 \cup \Rightarrow_1 \cup \ldots \cup \Rightarrow_5$.

The following may be proved easily by inspection of the rules.

Proposition 2 *If* $E \Rightarrow E'$ *then* $Sol(E) = Sol(E')$.

We now need to show that the sets of equations corresponding to the different graphs produced by the procedure described above can also be obtained through some sequence of transformations, starting from the set of equations that corresponds to the first graph.

The proof that procedure **Semiunify** produces a graph G whose set of equations $E(G)$ is equivalent to the first graph is very similar to the one presented in section 5. The main difference here is that in this case no links, and therefore no equations, are removed. We need to show that a call to **Simplify_var** preserves solutions.

Lemma 9 *Let* $\Gamma = \{t \leq^? u\}$ *be an instance of seminunification. Then the call* **Semiunify**$(t, 1, u, 0)$ *returns a graph* G_S *such that* σ *is a solution to* Γ *if and only if* σ *is a solution to* $E(G_S)$*.*

Proof: The proof is straightforward using the proof of correctness of **Semiunify** in section 5. □

The case for function **Cycle** is the same, and we omit the proof.

We now need to show that the solution extraction part produces a solution to the corresponding set of equations, and therefore a solution for the semi-unification instance. We prove this by pointing out that all the push operations preserve solutions, since they correspond to decomposition operations in the set of equations, and the final solution extraction is done by simple substitution.

Proposition 3 *Let* Γ *be an instance of semi-unification, and let* G *be the graph obtained from calling functions* **Semiunify** *and* **Cycle** *on* Γ*. A call to function* **Pushlinks** *returns a graph* G' *such that* $Sol(E(G)) = Sol(E(G'))$*.*

Proposition 4 *Let* Γ *be an instance of semi-unification, and let* G *be the graph obtained from calling functions* **Semiunify** *and* **Cycle** *and* **Pushlinks** *on* Γ*. A call to function* **Pushloops** *returns a graph* G' *such that* $Sol(E(G)) = Sol(E(G'))$*.*

Proposition 5 *Let* Γ *be an instance of semi-unification, and let* G *be the graph obtained from calling functions* **Semiunify** **Cycle** **Pushlinks** *and* **Pushloops** *on* Γ*. Then a call to function* **Simplify_var** *returns a graph* G' *such that* $Sol(G) = Sol(G')$*.*

The last proposition determines the correctness of function **Simplify_rule**.

Lemma 10 *Let* Γ *be an instance of semi-unification, and let* G *be the graph obtained from calling functions* **Semiunify** **Cycle** **Pushlinks**, **Pushloops** *and* **Simplify_var** *on* Γ*. Then a call to function* **Simplify_rule** *returns a graph* G' *such that* $Sol(G) = Sol(G')$*.*

Proof: The function **Simplify_rule** simply applies the rules $\Rightarrow_5$ and $\Rightarrow_2$. □

Finally, we need to show how to read the semi-unifier from the resulting graph, and show that it actually is a principal semi-unifier.

The final graph, after the call to **Simplify**, contains one link in the extract property for each variable. It is from these links that the solution is extracted. This is a solution to the set of equations represented by the graph, and therefore it is a solution to the semi-unification instance given as input.

The semi-unifier σ can be read in the following triangular form: For each variable x, such that the extract property contains the link $x \xrightarrow{0 \qquad t} n$ make $x\sigma = \widehat{\varphi^n t}$. The matching substitution, ρ, can also be read in this form, but in this case it is given by the links that have something different from 0 as the source cost, so a link $x \xrightarrow{n \qquad m} t$ with $n \neq 0$, we do:

$$x\rho = x_1, \ldots, x_{n-1}\rho = \widehat{\varphi^m t},$$

where the x_i, $x \in X$ are fresh variables that appear nowhere else, and $x\sigma = x$.

6.4 Complexity Analysis

The algorithm presented in this section can be divided into two parts. The first one is very similar to the algorithm presented in section 3 and the other part consists of all the processing that takes place in the actual finding of the solution. These two parts are done in parallel, but here we discuss them separately.

The first part is the decision algorithm plus some extra processing to deal with self-loops. This extra processing makes the part more expensive than before. The analysis is very similar to the one presented in section 5.1, and goes as follows:

- **Semiunify** is called at most n times, where n is the number of symbols in the original equation to be solved;
- A sequence of $O(n)$ calls to Union (implicit in our algorithm) and **Find** can be performed with $O(n\,\alpha(n))$ assignments, comparisons, or additions of two *numbers*, where α is the functional inverse of Ackermann's function [1];
- All other operations add at most a constant number of assignments, comparisons, or additions/subtractions of two *numbers* to each call to **Semiunify**;
- The bit-cost of the arithmetic operations of the algorithm may be analyzed as follows: if we start with two numbers of constant size (number of bits), and create a list of $O(m)$ new numbers by addition, subtraction and Greatest Common Divisor (GCD) of previous members of the list, we can create numbers of size at most $O(m)$. Subtraction and addition have a cost of $O(m)$, while it has been shown that GCD can be performed in $O(m\,\log^2(m)\,\log\log(m))$ operations on bits, thus at each step we need to do at most $O(m)$ operations, which give a total cost of $O(m^2\,\log^2(m)\,\log\log(m))$ (where $m = n\,\alpha(n)$).
- If we are using the uniform-cost model, then there can be at most $O(n\,\alpha(n))$ steps involving addition, subtraction, or GCD. The GCD of two m-bit numbers can be found in O(m) arithmetic operations. Hence the total cost of the arithmetic is $O(n^2\,\alpha(n)^2)$ operations.

The second part of the process deals with the links in the extract property of each sub-term. These links must be pushed down to the variables for processing, and more links may be generated during this processing at the variable level. Since the procedure does not allow more than one link to be pushed per pair of functional nodes, the worst case is $O(n^2)$ added to the cost of the decision procedure. The other significant process in the extraction part is the actual simplification of links. There may be, in total, at most $O(n^2)$ links that need processing, taking $O(n^2)$.

In either cost model, the GCD operations dominate the other costs of the algorithm. This gives us the following complexity result:

Theorem 4 *Under the uniform-cost model, the algorithm to produce principle solutions runs in* $O(n^2\,\alpha(n)^2)$. *Under the bit-oriented RAM model the cost is*

$$O(n^2\ \log^2(n\,\alpha(n))\ \log\log(n\,\alpha(n))\,\alpha(n)^2)$$

assignments, comparisons, or bit operations.

7 On the Relationship between Unification and Semi-Unification

The basic approach taken in this paper is to encode semi-unification instances as unification instances over φ-terms, apply the Huet unification closure algorithm while keeping track of the number of φ's at each subterm, and end up with a solved form from which a solution can be extracted. This does not produce principal solutions, but a more complex version of the same basic idea, which keeps the DAG in a certain minimal form, does produce principal solutions.

This approach suggests that we might try to use other algorithms for standard unification, e.g., Robinson's simple algorithm Corbin-Bidoit rehabilitation of Robinson's algorithm [3], or Patterson-Wegman's linear-time algorithm.

Unfortunately, adapting such algorithms to semi-unification is not straight-forward, because in fact one needs to be able to unify φ-terms in which the standard occur check fails. For example, the instance $x \leq^? f(x)$ has Id as a semi-unifier (and so ρ would be $\{x \mapsto f(x)\}$). In other words, one needs an algorithm which can find unifiers for rational trees (terms represented by graphs which may have cycles), and none of the algorithms mentioned above, except for Huet's, have this property. Indeed, our first (doomed) attempt at a fast semi-unification algorithm was based on the Patterson-Wegman approach, but we were not able to prove termination, nor to find a way of producing principal solutions. The difficulty is that Patterson-Wegman uses a extension of the subterm order to equivalence classes to find the optimal ordering for the steps of the algorithm, and in the case of cyclical terms no such ordering exists. Similarly, ensuring termination when modifying Robinson's or Corbin-Bidoit's algorithm for cyclical terms is not straight-forward, and since cycles lead to the generation of non-principal solutions, we believe it would be especially difficult to adapt such algorithms to produce principal solutions. Hence the approach using Huet's algorithm for rational trees appears to be the optimal approach, both in terms of asymptotic complexity and in terms of the practical complexity of the algorithm.

8 Previous Work

The semi-unification problem was introduced in the late 70's by Lankford and Musser [10]. Purdom presented an algorithm for uniform semi-unification in [16], but his algorithm, as observed in [8], is incorrect. Other decision procedures for the uniform case can be found in [5, 6], in [15], and in [11]. F. Henglein showed that his algorithm, which finds principal solutions, is in PSPACE.

The first rigorous treatment of an efficient algorithm for the uniform case was given by Kapur et al. in [8]. In this paper, an (exponential time) algorithm based on completion (given above in Section 2), and a method for extracting solutions, was presented, and this leads to a graph-based decision procedure which is shown to terminate in polynomial time; the procedure does not produce solutions. It was conjectured (Paliath Narendran, personal communication) that the algorithm ran in cubic time under the uniform cost RAM model, but this was never proved.

The possibility of applying the Corbin-Bidoit method [3] to φ-terms was explored by Ružička in [17]. The algorithm is fairly simple to implement and analyze, and is shown to run in quadratic time (under the uniform cost RAM model). However, it does not produce principal solutions, and unfortunately (for reasons discussed in the previous section) there appears to be a serious bug. We conjecture that this can be patched, but we do not know whether this would still run in quadratic time. If we suppose that it did, then under the (perhaps more reasonable) RAM model counting assignments and bit operations, it should be possible to retrofit fast algorithms for arithmetic (e.g., GCD) into his approach, giving a complexity of $O(n^3 log^2 n)$, however we have not confirmed this. Do do not think it is possible to modify the algorithm in any reasonable way to produce principal solutions.

This paper represents the core of the first author's thesis [13], and was first published in preliminary form in [12].

9 Conclusion

We have presented the fastest algorithms to date for uniform semi-unification, based on the unification closure method for standard unification. Our decision procedure is asymptotically faster than our method for generating principal solutions. In the context of other algorithms for uniform semi-unification, these results show that the unification closure method for standard unification can be adapted for semi-unification, and the cost in terms of symbolic operations (assignment and comparison of pointers) is the same for both problems. However, semi-unification introduces arithmetic (most critically, the use of GCD) for counting the number of ρ's applied to subterms, and this arithmetic dominates the cost of the symbolic operations. Since it is not possible to adapt the Patterson-Wegman approach to semi-unification, we surmise that our approach is the fastest possible under the uniform cost RAM model; it may be possible to trim the bound for the bit-oriented RAM model somewhat by a more precise analysis of the arithmetic operations required, however we leave this as a subject for future research.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.

[2] F. Baader and W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook on Automated Deduction*, volume 1, chapter 8, pages 445–533. Elsevier, 2001.

[3] J. Corbin and M. Bidoit. A rehabilitation of Robinson's unification algorithm. In *Proceedings of the 9th World Computer Congress (IFIP'83)*, pages 909–914. Elsevier, 1983.

[4] N. Dershowitz and J.-P. Jouannaud. Notations for rewriting. *Bulletin of the EATCS*, (43):162–172, 1991.

[5] F. Henglein. Type inference and semi-unification. In *ACM Conference on Lisp and Functional Programming*, pages 184–197. ACM, 1988.

[6] Fritz Henglein. *Polymorphic Type Inference and Semi-Unification*. PhD thesis, Rutgers University, April 1989. Available as NYU Technical Report 443, May 1989, from New York University, Courant Institute of Mathematical Sciences, Department of Computer Science, 251 Mercer St., New York, N.Y. 10012, USA.

[7] G. Huet. *Résolution d'Equations dans les Langages d'Ordre* $1, 2, \ldots, \omega$. PhD thesis, Université de Paris VII, 1976.

[8] D. Kapur, D. Musser, P. Narendran, and Stillman J. Semi-unification. In *Proc. of 8-th Conference on Foundations of Software Technology and Theoretical Computer Science*, Dec. 1988.

[9] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. *Information and Computation*, 102(1):83–101, January 1993.

[10] D.S. Lankford and D.R. Musser. A finite termination criterion. Unpublished Draft, USC Information Sciences Institute, 1978.

[11] H. Leiss. Semi-unification and type inference for polymorphic recursion. Technical Report INF-2-ASE-5-89, Siemens, Munich, Germany, 1989.

[12] A. Oliart and W. Snyder. A fast algorithm for semi-unification. In *CADE-15*, volume 1421 of *LNAI*, pages 239–253. Springer-Verlag, 1998.

[13] Alberto Oliart. *Uniform Semi-Unification*. PhD thesis, Boston University, 1998.

[14] I. Privara and P. Ružička. An almost linear Robinson unification algorithm. *Acta Informatica*, 27(1):61–71, 1989.

[15] P. Pudlák. On a unification problem related to Kreisel's conjecture. *Commentationes Mathematicae Universitatis Carolinae*, 29(3):551–556, 1988.

[16] P.W. Purdom. Detecting looping simplifications. In P. Lescanne, editor, *Proceedings 2nd Conference on Rewriting Techniques and Applications (RTA'87), Bordeaux, France*, volume 250 of *Lecture Notes in Computer Science*, pages 54–61. Springer, Berlin, May 1987.

[17] P. Ružička. An efficient decision algorithm for the uniform semi-unification problem. In A. Tarlecki, editor, *Proceedings 16th International Symposium on Mathematical Foundations of Computer Science (MFCS'91)*, volume 520 of *LNCS*, pages 415–425. Springer-Verlag, September 1991.