An Even Faster Solver for General Systems of Equations

Christian Fecht

Universität des Saarlandes
Postfach 151150
D-66041 Saarbrücken
Tel.: +49–681–302–5573
fecht@cs.uni-sb.de

Helmut Seidl

Fachbereich IV – Informatik
Universität Trier
D-54286 Trier
Tel.: +49–651–201–2835
seidl@psi.uni-trier.de

Abstract. We present a new algorithm which computes a partial approximate solution for a system of equations. It is *local* in that it considers as few variables as necessary in order to compute the values of those variables we are interested in, it is *generic* in that it makes no assumptions on the application domain, and it is *general* in that the algorithm does not depend on any specific properties of right-hand sides of equations. For instance, monotonicity is not required. However, in case the right-hand sides satisfy some weak monotonicity property, our algorithm returns the (uniquely defined) least solution.
The algorithm meets the best known theoretical worstcase complexity of similar algorithms. For the application of analyzing logic languages, it also gives the best practical results on most of our real world benchmark programs.

1 Introduction

In numerous application areas the information one is interested in can be specified most conveniently by systems of *equations* $x = f_x, x \in V$, where V is a (usually finite) set of unknowns. Important examples include the description of first and follow sets for grammars in the area of parser generation [18], control flow resp. data flow information for imperative programs [13, 10, 15], abstract interpretation [6] of functional and logic languages, and system verification [8, 4].

Given system S of equations, the main goal in all applications consists in efficiently computing a *solution* over some complete lattice D, i.e., an assignment of the variables of S to values in D in such a way that the left-hand side and the right-hand side of each equation evaluate to the same value. If the right-hand sides denote *monotonic* functions, system S is guaranteed to have a *least* solution which usually is also the best information to be obtained. Surprisingly enough, monotonicity of right-hand sides can not always be assured. Well–known and important program analyses introduce *non–monotonic* right-hand sides. In a non-monotonic setting, system S need not have any solution at all. The best we can hope for in this general case is an *approximate solution*, i.e., a variable assignment where the values of left-hand sides either equal the corresponding

right-hand sides or exceed them. A *solver* is an algorithm which, given system S of equations, tries to compute a non-trivial approximate solution. If system S is monotonic, the solver should return the least solution, i.e., should have *maximal precision*.

Unfortunately, most solvers presented so far have been developed and presented in an application dependent way, often using different notation. Therefore, it is hard to capture the essentials of the solver algorithms and the key ideas underlying different optimization strategies. Since it is difficult to compare these algorithms, the same algorithmic ideas and optimizations have been reinvented by different people for different applications. Also, specialized algorithms do not allow for reusable implementations. On the contrary, introducing both efficient and application independent solvers offers a lot of promising possibilities. The algorithmic ideas can be pointed out more clearly and are not superseded by application specific aspects. Correctness for the solver can therefore be proven more easily. Once proven correct, a general purpose algorithm can be instantiated to different application domains. Thus, for the overall correctness of the application it simply remains to check whether or not system S correctly models the problem to be analyzed. Reasoning about the approximation process itself can be totally abandoned.

Recently, two efficient application independent solvers have attracted attention, namely *topdown solver* **TD** of Le Charlier and Van Hentenryck [2] and an enhanced version **W** of Kildall's worklist algorithm [13, 12, 17]. The first one has been successfully used to implement analyzers for logic languages [3] and behaves extremely well in this application area – although a better worstcase complexity can be proven for the second one.

In this paper we present a new application independent solver **WRT** which has the same worstcase complexity as **W** but whose instantiation as an analyzer of logic programs additionally outperforms topdown solver **TD** on almost all our benchmark programs. Similar to the known solvers **W** and **TD**, our new solver **WRT** is both *local* and guided by *dynamic* dependencies between variables. The reason is that often in practice the set of all variables is very large where at the same time the subset of variables whose value one is interested in is rather small. In model checking, for instance, one only wants to determine whether or not the initial system state satisfies a given property. A *local* solver, therefore, tries to compute the values only of as few variables as necessary in order to compute the values of the interesting variables. For a local solver, precomputation of all variable dependencies as in the original global version of **W** [13] is no longer feasible. It may even happen that variable dependencies change unpredictably during execution of the algorithm. Therefore, dependencies between variables have to be determined and changed *dynamically* during the execution process.

The overall structure of our paper is as follows. In the first three sections we introduce basic concepts for our exposition. Especially, we introduce the notion of *weak monotonicity* which is more liberal than ordinary monotonicity but still ensures both existence of a least solution together with maximal precision of our algorithms. The following three sections succinctly present solvers **W**, **TD** and

WRT. We use here an ML–style language as algorithmic paradigm. We included descriptions of **W** and **TD** to demonstrate that not only new algorithm **WRT** but also the existing ones benefit from this kind of approach. Section 8 presents our test application, namely abstract interpretation of logic programs, and points out how, based on this technology, efficient Prolog analyzers can be generated. Finally, Section 9 summarizes results from our practical experiments. Especially, it compares all three solvers and concludes.

2 Systems of Equations

Assume $\mathcal{D}$ is a complete lattice. Operationally, $\mathcal{D}$ is given as an abstract datatype consisting of a set of values D together with a designated least element $\bot$, an equality predicate "=", and a binary least upper bound operation "$\sqcup$". Note that $\mathcal{D}$ possibly supports other kinds of (monotonic) operations which may be used by right-hand sides of equations. These, however, are not used by our generic solvers. Especially, they do not depend on any implementation of the partial ordering relation "$\sqsubseteq$" on D as opposed to those in [17, 12, 2].

A *system of equations* S is given as a pair $(V, \mathcal{F})$ where V denotes a set of variables and $\mathcal{F}$ denotes (a representation of) the right-hand sides f_x for every $x \in V$. Right hand sides f_x are meant to be (not necessarily monotonic) mappings of variable assignments in $V \to D$ to values in D. In case when set V is "small", $\mathcal{F}$ simply may consist in a collection of corresponding function definitions. In the interesting case however where set V is big, the f_x are only *implicitly* given through some total function F of type $F : V \to ((V \to D) \to D)$. The right-hand side f_x for $x \in V$ then is obtained by $f_x = F\ x$.

In the sequel, we do not distinguish between the algorithm realizing a function f in $(V \to D) \to D$ and the function itself. The only assumption we make is that the only way the algorithm realizing f has access to variable assignment σ provided as its parameter is to call σ on variables $x \in V$.

Similar to Le Charlier and Van Hentenryck in [2], we present our solvers in a very general setting. Nevertheless, we insist on the following three further assumptions:

1. set V of variables is always finite;
2. complete lattice D has finite height h, i.e., every strictly increasing sequence contains at most $h + 1$ elements;
3. evaluation of right-hand sides is always terminating.

All three assumptions are satisfied in many applications. In [2], however, it is pointed out how such assumptions still can be relaxed (at least to some extent). It is for clarity of presentation, that we refrain from doing so as well.

3 Approximate Solutions

A variable assignment $\sigma : V \to D$ is called

– *solution* for S if $\sigma\ x = f_x\ \sigma$ for all $x \in V$.
– *approximate solution* for S if $\sigma\ x \sqsupseteq f_x\ \sigma$ for all $x \in V$.

Note that by definition, every solution is also an approximate solution, and that every system S has at least one approximate solution, namely the trivial one mapping every variable to $\top$, the top element of D. In general, we are interested in computing a "good" approximate solution, i.e., one which is as small as possible or, at least, non–trivial.

For function $G : (V \to D) \to V \to D$ define the n–th iterate of G by $G^0\bot = \bot$ (where $\bot$ denotes the minimal variable assignment mapping every x onto $\bot$), and for $n > 0$, $G^n\bot = G(G^{n-1}\bot)$. With system S we associate function $G_S : (V \to D) \to V \to D$ defined by $G_S\ \sigma\ x = f_x\ \sigma$. In general, sequence $G_S^n\bot, n \geq 0$, may not even be ascending. If we are lucky, all right-hand sides f_x are monotonic. Then G_S is monotonic as well, and therefore has a least fixpoint which is also the least (approximate) solution of S. Often, however, we are less lucky and right-hand sides f_x are not monotonic in general. As a consequence, function G_S is also not monotonic.

Example: Consider complete lattice $D = \{0 \sqsubset 1 \sqsubset 2\}$ and system S with variables $V = \{\langle d\rangle \mid d \in D\}$ and right-hand sides $f_{\langle d\rangle}\ \sigma = \sigma\ \langle\sigma\ \langle d\rangle\rangle$. Right-hand sides of this kind are common when analyzing programs with procedures or functions. Variables $\langle d\rangle$ and their values in the solution of the system represent the input-output behavior of a given procedure p. Thus, nesting of procedure calls introduces "indirect addressing" of variables. Now consider variable assignments σ_1, σ_2 where

$$\begin{array}{lll} \sigma_1\ 0 = 1 & \sigma_1\ 1 = 1 & \sigma_1\ 2 = 0 \\ \sigma_2\ 0 = 1 & \sigma_2\ 1 = 2 & \sigma_2\ 2 = 0 \end{array}$$

Clearly, $\sigma_1 \sqsubseteq \sigma_2$. However, $G_S\ \sigma_1\ \langle 1\rangle = 1$, but $G_S\ \sigma_2\ \langle 1\rangle = 0$. Hence, G_S is not monotonic. □

Even if function G_S of system S is not monotonic in general, there might be some partial ordering "$\leq$" on V, such that G_S is monotonic at least on monotonic variable assignments. As usual, variable assignment $\sigma : V \to D$ is called monotonic iff $x \leq x'$ implies $\sigma\ x \sqsubseteq \sigma\ x'$. G_S (and equally well S) is called *weakly monotonic* with respect to variable ordering "$\leq$" iff G_S has the following properties:

– If $x \leq y$ then for all monotonic σ, $f_x\ \sigma \sqsubseteq f_y\ \sigma$;
– If $\sigma_1 \sqsubseteq \sigma_2$ and at least one of the variable assignments σ_i is monotonic then for every $x \in V$, $f_x\ \sigma_1 \sqsubseteq f_x\ \sigma_2$.

Observe that monotonicity is a special case of weak monotonicity where the variable ordering is equality. System S of the example above is weakly monotonic w.r.t. variable ordering "$\leq$" given by $\langle d_1\rangle \leq \langle d_2\rangle$ iff $d_1 \sqsubseteq d_2$. Also, both the systems of equations investigated by Jørgensen in his paper on fixpoints in finite function spaces [12] and those systems derived in Section 8 from a generic abstract interpretation framework for logic programs are weakly monotonic. In the latter case, the variables of the equation system are of the form (p, β) where p is a predicate symbol and β an abstract substitution. Similar to the small example

above, the ordering is given by $(p_1, \beta_1) \leq (p_2, \beta_2)$ iff $p_1 = p_2$ and $\beta_1 \sqsubseteq \beta_2$. We have:

Fact 1 If G_S is weakly monotonic then the following holds:

1. Sequence $G_S^n \bot, n \geq 0$, is ascending;
2. S has a least approximate solution μ which is also a solution of S and monotonic. μ is given by $\mu = G_S^n \bot$ for some $n \leq h \cdot \#V$. □

Furthermore, it will turn out that weak monotonicity is sufficient for our solvers not only to compute approximate solutions but precisely the minimal solutions according to Fact 1.

4 Partial Variable Assignments

Assume we are given a system of equations $S = (V, \mathcal{F})$ where the set V of variables is tremendously large. One way to deal with large sets of variables is *functional partitioning* as described in [1]. If, however, we are only interested in the values for a rather small subset X of variables, we could try to compute the values of an approximate solution only for variables from X and all those variables y that "influence" values for variables in X. In the following we are going to make this idea precise.

Consider some function $f : (V \to D) \to D$. Evaluation of f on its argument σ does not necessarily consult *all* values $\sigma\ x, x \in V$. Therefore, evaluation of f may also be defined for some *partial* variable assignment $\sigma : V \rightsquigarrow D$. Now assume evaluation of f on σ succeeds. Then we can define set $dep(f, \sigma)$ as the set of all variables y such that the evaluation of f on input σ accesses value $\sigma\ y$.[1]

As a consequence we have, $f\ \sigma = f\ \sigma'$ also for every variable assignment $\sigma' : V \rightsquigarrow D$ as long as $dep(f, \sigma) \subseteq dom(\sigma')$ and σ' agrees with σ on $dep(f, \sigma)$.

Given a partial variable assignment $\sigma : V \rightsquigarrow D$, we put up a *dependence graph* $G(\sigma)$ (relative to σ). The node set of $G(\sigma)$ is given by V itself whereas the set of edges consists of all pairs (y, x) such that $f_x\ \sigma$ is defined and $y \in dep(f_x, \sigma)$. Then variable y is said to *influence* variable x relative to σ iff there is a path in $G(\sigma)$ from y to x. Let furthermore $X \subseteq V$ denote the set of variables which we are interested in. Then we call (partial) variable assignment σ *X–stable* iff for every $y \in V$ influencing some $x \in X$ relative to σ, $f_y\ \sigma$ is defined with $\sigma\ y \sqsupseteq f_y\ \sigma$.

Using this terminology, our goal can be precisely stated as follows:
Given: system S and set X of interesting variables,
Compute: a partial assignment σ with the following properties:

1. σ is X–stable;

[1] In fact, it is this definition which does not really refer to f considered as a function but to f considered as an algorithm. Therefore, it is not really a formal one. To make it precise it would be necessary to formally introduce the programming language to express f and its operational semantics – which we happily refrained to do.

2. If S is weakly monotonic w.r.t. some variable ordering and μ is its least (approximate) solution, then $\sigma\, y = \mu\, y$ for all y influencing some variable in X (relative to σ).

In other words, our algorithm when given S and X should not only return an X–stable partial variable assignment but also should behave "well" in a "well–behaving" context. An algorithm with these properties is called *solver*.

In order to formally compare different solvers with respect to their worstcase complexities we additionally make the following assumptions:

1. We only count the number of evaluations of right-hand sides f_x.
2. The sizes of all sets $dep(f, \sigma)$ are bounded by some constant.

In our applications we found the second one to be a reasonable abstraction. The first assumption, on the contrary, means that we ignore organizational overhead like computing the transitive closure of a certain relation in **TD** or maintaining a priority queue in **WRT**. This is at least justified when (as in our example application) the overall running time is drastically dominated by the calculations in lattice D.

5 The Worklist Solver W

The first and simplest algorithm we consider is solver **W** (Fig. 1). Variants of it were proposed by Jørgensen for the demand driven evaluation of systems of (possibly recursive) first order function definitions in [12] and by Vergauwen, Wauman and Lewi in [17] in a monotonic setting. **W** extends the usual worklist algorithm for systems with statically known variable dependencies, e.g., the one of Kildall [13] and his followers, to the case where system S is dynamically constructed and dependencies between variables may vary.

The algorithm proceeds as follows. The set of variables yet to be evaluated is kept in data structure W, called *worklist*. It is initialized with the set X of variables in which we are interested. For every variable x considered so far, we (globally) maintain the current value $\sigma\ x$ together with a set $infl(x)$ of certain variables y such that the evaluation of f_y (on σ) may access value $\sigma\ x$ or more formally, x may be contained in $dep(f_y, \sigma)$.

As long as W is nonempty, the algorithm iteratively extracts some variable x from W and evaluates right-hand side f_x of x on the current partial variable assignment σ. If the least upper bound of the old value $\sigma\ x$ and $f_x\ \sigma$ is different from the old value (and hence larger!), the value of σ for x is updated. Since the value for x has changed, the values of σ for all $y \in infl(x)$, may no longer be valid; therefore, they are added to W. Afterwards, $infl(x)$ is reset to $\emptyset$.

However, right-hand side f_x is not evaluated on σ directly. There are two reasons for this. First, σ may not be defined for all variables y the algorithm for f may access; second, we have to determine all y such that $f_x\ \sigma$ depends on $\sigma\ y$. Therefore, f_x is applied to auxiliary function $\lambda y.\, eval(x, y)$. When applied to variables x and y, *eval* first checks whether σ is indeed defined for y. If this

```
fun eval(x : V, y : V) : D
begin
  if y ∉ dom(σ) then σ(y) := ⊥; infl(y) := ∅; W := W ∪ {y} fi;
  infl(y) := infl(y) ∪ {x};
  return σ(y)
end;
begin
  σ := ∅; infl := ∅; W := ∅;
  forall x ∈ X do σ(x) := ⊥; infl(x) := ∅; W := W ∪ {x} od;
  while W ≠ ∅ do
    choose an x ∈ W; W := W − {x};
    let new = σ(x) ⊔ f_x (λy. eval(x, y) ) in
        if σ(x) ≠ new then σ(x) := new; W := W ∪ infl(x); infl(x) := ∅ fi
    end
  od;
  return σ
end
```

Fig. 1. Algorithm **W**.

is not the case, y is added to the domain of σ and $\sigma\ y$ is set to some safe initial value (e.g., $\bot$). Also, variable $\mathit{infl}(y)$ is created and initialized with $\emptyset$. Finally, since y has not yet been considered its future evaluation is initiated by adding y to W. In any case (i.e., whether $y \in \mathit{dom}(\sigma)$ or not), x is added to $\mathit{infl}(y)$, and the value of σ for y (which is now always defined) is returned.

Remarks: We use accumulating updates of entries $\sigma\ x$ in order to handle non–monotonic systems as well. The elegant use of function *eval* is made possible by our ML–style algorithmic paradigm: on the one hand we rely on partial applications and on the other hand on side effects. In contrast to Vergauwen, Wauman and Lewi's formulation, our version also works in a non–monotonic setting. In contrast to Jørgensen's algorithm we need not make any assumptions on the nature of right-hand sides. Also, we need not ensure monotonicity of σ which in practice turns out to be rather costly. Last but not least, in our version sets $\mathit{infl}(x)$ are emptied after use (which is also the case in Verhauwen, Wauman and Lewi's variant, but not in Jørgensen's).

Generalizing the proof of [12], we find:

Theorem 1. *1.* **W** *terminates after at most* $O(h \cdot N)$ *steps where* $N \leq \#V$ *is the number of considered variables.*
2. Algorithm **W** *is a solver.* □

6 The Topdown Solver TD

Algorithm **TD** (cf. Fig. 2) originally was proposed by Le Charlier and Van Hentenryck [2] and applied by them to the analysis of logic programs [3].

According to its name, solver **TD** proceeds in a topdown fashion. The basic idea is as follows: If variable y is accessed during the evaluation of a right-hand side, the value of y is not just returned as in algorithm **W**. Instead, **TD** first tries to compute the best possible approximation for y. In order to realize this idea, the algorithm must take precaution that no new iteration is started for variable y provided

1. one iteration process for y has already be initiated; or
2. iteration for y will not result in a new value for y, i.e. y is stable with respect to the current state.

Therefore, **TD** maintains (additionally to σ and *infl*) two extra sets, namely *Called* and *Stable*. A variable y is in *Called* iff a computation for y has been started but not yet terminated. $y \in$ *Stable* means the iteration for y has been completed and since then value $\sigma\ y'$ has not changed for any variable y' possibly influencing y.

Execution of **TD** starts by iteratively calling procedure *solve* for all x whose values we are interested in. In contrast to algorithm **W**, procedure *solve* when applied to variable x, does not evaluate the right-hand side f_x just once but iteratively continues to reevaluate f_x until $x \in$ *Stable*. In detail, procedure *solve* first checks whether its argument x is in *Called* or in *Stable*. In this case, *solve* immediately returns. Otherwise, it proceeds as follows. If x has not been considered so far, x is added to $dom(\sigma)$, $\sigma\ x$ is initialized with $\bot$; also variable $infl(x)$ is created and initialized with $\emptyset$. Then x is added to set *Called*. It follows an accumulating iteration on x. Finally, x is removed again from *Called*.

The accumulating iteration on x first adds x to set *Stable* (which is kind of an optimistic decision which in the sequel may need revision). Now, right-hand side f_x is evaluated and the least upper bound of the result with the old value of σ for x is computed. If this value *new* is different from the old value $\sigma\ x$ (and hence strictly larger), then the value of σ for x is updated and all values of σ for variables (possibly) affected by this update *destabilized*. This is repeated until x remains stable, i.e., x has not been removed from *Stable* during the last iteration.

For destabilization, the set $infl^+(x)$ of (possibly) affected variables is computed by taking the transitive closure of set $infl(x)$, i.e., the smallest set I containing $infl(x)$ and for every $y \in I$ also $infl(y)$. Now destabilization means that all variables in $infl^+(x)$ are removed from *Stable*. Finally, procedure *destabilize* resets set $infl(x)$ to $\emptyset$.

It remains to explain that (similar as in **W**) procedure *solve* does not directly evaluate right-hand side f_x on σ but on auxiliary function $\lambda y.eval(x, y)$. Function *eval* when applied to parameters x and y first calls $solve(y)$. Then it adds x to set $infl(y)$ and, finally, returns $\sigma\ y$.

Remarks: Our presentation of **TD** is closely related to that of Le Charlier and Van Hentenryck in [2]. We only removed all monotonicity constraints (which to maintain can be costly in practice). Also – at least to our taste – our treatment of variable dependencies is much more elegant.

Similar to [2] we find:

```
proc solve(x : V)
begin
  if not(x ∈ Stable or x ∈ Called)
  then
    if x ∉ dom(σ) then σ(x) := ⊥; infl(x) := ∅ fi;
    Called := Called ∪ {x};
    repeat
      Stable := Stable ∪ {x};
      let new = σ(x) ⊔ f_x (λy. eval(x, y)) in
        if σ(x) ≠ new then σ(x) := new; destabilize(x) fi
      end
    until x ∈ Stable;
    Called := Called − {x}
  fi
end;

fun eval(x : V, y : V) : D
begin
  solve(y); infl(y) := infl(y) ∪ {x}; return σ(y)
end

proc destabilize(x : V)
begin
  foreach y ∈ infl⁺(x) do
    Stable := Stable − {y};   infl(y) := ∅
  od;
  infl(x) := ∅
end;

begin
  σ := ∅; Stable := ∅; Called := ∅; infl := ∅;
  foreach x ∈ X do solve(x) od;
  return σ
end
```

Fig. 2. Algorithm **TD**

Theorem 2. *1.* **TD** *terminates after at most* $O(h \cdot N^2)$ *steps where* $N \leq \#V$ *is the number of considered variables.*
2. Algorithm **TD** *is a solver.* □

7 The Time Stamps Solver WRT

It turns out that – despite its theoretically worse worstcase complexity – solver **TD** performs extremely well (see, e.g., our numbers in Section 9). Theoretically, it can be proven to be "optimal" for the case of acyclic variable dependencies – which is not the case, e.g., for solver **W**. In our example application, deficiencies occur only for large and medium sized input programs to be analyzed like *aqua-c*, *chat-parser* or *readq* of our benchmark suite. We conclude that **TD** does not

adequately treat larger strong components in the dependence graph.

Opposed to that, solver **W** – despite its theoretically better worstcase complexity – gives worse practical results. One source of inefficiency clearly is its insufficient treatment of variables y newly encountered during evaluation of some right-hand side f_x. Evaluation of f_x simply proceeds while assuming $\bot$ as value for $\sigma\, y$. Computing a better value for $\sigma\, y$ is postponed. It follows that the (possibly) new value for x is most likely to be recomputed later on. Therefore, **W** cannot be proven optimal even for systems of equations with *acyclic* static variable dependencies.

It is for this reason that we propose two improvements to solver **W**. First, we add recursion (the "**R**") in order to compute a better initial value for $\sigma\, y$ than $\bot$ in case y is newly encountered. This modification already guarantees optimality for the case of acyclic static variable dependencies. It does not, however, guarantee that (still in case of static variable dependencies) iteration is performed in one strong component after the other. The best idea therefore might be to add something like an algorithm detecting strong components "on the fly". Since dependencies vary over time in that dependencies both can be added and removed, such an approach seems not very practical. The second best idea is to use *time stamps* (the "**T**").

The resulting algorithm is presented in Fig. 3. Additionally to the data structures of solver **W**, we maintain for every $x \in dom(\sigma)$ its time stamp $time(x)$ which is a positive integer. It records the last time $solve(x)$ has been called. Accordingly, worklist W now is organized as a (max) priority queue where the priority of an element is given by its time stamp. Moreover, we need a stack *Stack* for the time stamps of variables in the recursion stack.

Solver **WRT** works as follows. Initially, variables $x \in X$ are put into worklist W. While doing so, each such variable is equipped with a new time stamp. Then the main loop consists in extracting the variable with maximal time stamp from W and applying procedure *solve* to it until W is empty.

Procedure *solve* when applied to variable x first checks whether $x \in dom(\sigma)$. If not, x is removed from W (if it has been there) and added to the domain of σ, and $\sigma\, x$ and $infl(x)$ are initialized with $\bot$ and $\emptyset$, respectively. In any case, x now receives the next time stamp; this value is pushed onto *Stack*. Similar to **W**, the least upper bound is computed of the old value of σ for x and f_x evaluated on σ. If this new value is different from the old one, σ is updated, the variable set $infl(x)$ is included into W and afterwards reset to $\emptyset$. Then the top element is popped from *Stack*.

The modification now is that procedure *solve* need not return immediately. Instead if *Stack* is nonempty, we compare the current top of *Stack* t and the time stamps of the elements in W and apply *solve* to all those variables y in W whose times are less than t.

It remains to explain that, as with solvers **W** or **TD**, evaluation of right hand side f_x in *solve* is not performed directly on the current value of σ but on auxiliary function $\lambda y.eval(x, y)$. Function *eval* is the same as function *eval* of algorithm **TD**, i.e., when applied to variables x and y where y is not yet contained

```
proc solve(x : V)
begin
  if x ∉ dom(σ) then W := W − {x}; σ(x) := ⊥; infl(x) := ∅ fi;
  time(x) := nextTime();
  push(Stack, time(x));
  let new = σ(x) ⊔ f_x (λy.eval(x, y)) in
    if new ≠ σ(x)
    then σ(x) := new; W := W ∪ infl(x); infl(x) := ∅ fi
  end;
  pop(Stack);
  if not(isEmpty(Stack))
  then while W ≠ ∅ and top(Stack) < time(max(W))
       do let y = max(W) in
            W := W − {y}; solve(y)
          end
       od;
  fi
end;

fun eval(x : V, y : V) : D
begin
  if y ∉ dom(σ) then solve(y) fi;
  infl(y) := infl(y) ∪ {x};
  return σ(y)
end;

begin
  σ := ∅; time := ∅; Stack := empty; infl := ∅; W := X;
  foreach x ∈ X do time(x) := nextTime() od;
  while W ≠ ∅ do
      let x = max(W) in
        W := W − {x}; solve(x)
      end
  od;
  return σ
end
```

Fig. 3. Algorithm **WRT** with time stamps

in $dom(\sigma)$, function *eval* not only initializes both values $\sigma\, y$ and $infl(y)$, but additionally calls $solve(y)$ before returning current value $\sigma\, y$.

Remarks: Solver **WRT** has no equivalent either in Le Charlier and Van Hentenryck's paper [2], Jørgensen's work [12] or Vergauwen, Wauman and Lewi's overview [17]. It can be seen as an effort to combine the nice features of both solver **W** and solver **TD**. From **TD** it inherits the recursive descent on variables not yet considered whereas from **W** it receives the more flexible treatment of strongly connected components in the (dynamically changing) dependence graph. A simpler algorithm **W-DFS** can be obtained from **WRT** if each vari-

able obtains a new time stamp only once, namely when it is added to $dom(\sigma)$. Algorithm **W-DFS** can be seen as the dynamic version of *priority-queue iteration* in [11]. Two variables are arranged by **W-DFS** according to the *first* dependency between them that was established by the algorithm. This is a reasonable choice in case of *static* variable dependencies. However, in the dynamic case dependencies may arbitrarily change during the fixpoint iteration, meaning that any worklist algorithm with fixed variable priorities may show poor performance.

Theorem 3. *1.* **WRT** *terminates after at most* $O(h \cdot N)$ *steps where* $N \leq \#V$ *is the number of considered variables.*
2. Algorithm **WRT** *is a solver.*

8 Generic Abstract Interpretation of Logic Programs

As an example application for our implementations of solvers **W**, **TD**, **WRT** and **W-DFS** the first author integrated the four solvers into a tool (GENA) [9] for generating Prolog analyzers from specifications. The generated analyzers are based on the principle of abstract interpretation [7, 14, 3]. More specifically, the analyzers are based on the generic abstract interpretation framework for logic programs of Le Charlier and Van Hentenryck [3]. This framework assumes logic programs to be *normalized*. The set of *normalized goals* is inductively defined by the following rules:

$$\begin{array}{lll} G ::= \mathbf{true} & & \\ \quad \mid\ X_i = t & & X_i \notin vars(t) \\ \quad \mid\ p(X_{i_1}, \ldots, X_{i_n}) & & i_1, \ldots, i_n \text{ are distinct indices} \\ \quad \mid\ G_1, G_2 & & \end{array}$$

where $X_1, X_2, \ldots$ are *program variables*, t ranges over terms built up from program variables by formal applications of function symbols, and p denotes predicate symbols (of arities n). A *normalized program* is a set of *normalized clauses* $p(X_1, \ldots, X_n) \leftarrow G$. If a clause contains m variables, these variables are necessarily $X_1, \ldots, X_m$.

The abstract semantics is parametrized on an *abstract domain*. The abstract domain provides a finite complete lattice $(Asub, \sqcup, \bot)$ of abstract substitutions and functions for abstract unification (*aunify*), procedure entry (*restrG*, *extC*), and procedure exit (*restrC*, *extG*). The abstract semantics of P with respect to a given abstract domain $\mathcal{A}$ is a function $[\![P]\!]_{\mathcal{A}} : Pred \times Asub \to Asub$. It assigns an abstract success substitution to every abstract call. The abstract semantics $[\![P]\!]_{\mathcal{A}}$ is defined denotationally by means of functions *solveP*, *solveC*, and *solveG* which define the meaning of predicates, clauses, and goals, respectively.

$solveP :: (Pred \times Asub \to Asub) \to (Pred \times Asub \to Asub)$
$solveP\ \sigma\ (p, \beta) = \bigsqcup\{solveC\ \sigma\ (c, \beta) \mid c \in Prog \text{ and head predicate of } c \text{ is } p\}$

$$
\begin{array}{l}
\mathit{solveC} :: (\mathit{Pred} \times \mathit{Asub} \to \mathit{Asub}) \to (\mathit{Clause} \times \mathit{Asub} \to \mathit{Asub}) \\
\mathit{solveC}\ \sigma\ (p(X_1, \ldots, X_n)\ \leftarrow\ G,\ \beta_{in}) = \mathit{restrC}(\beta_{exit},\ n) \\
\quad \text{where } \beta_{exit}\ = \mathit{solveG}\ \sigma\ (G,\ \beta_{entry}) \\
\qquad\quad \beta_{entry}\ = \mathit{extC}(\beta_{in},\ p(X_1, \ldots, X_n)\ \leftarrow\ G)
\end{array}
$$

$$
\begin{array}{l}
\mathit{solveG} :: (\mathit{Pred} \times \mathit{Asub} \to \mathit{Asub}) \to (\mathit{Goal} \times \mathit{Asub} \to \mathit{Asub}) \\
\mathit{solveG}\ \sigma\ (\mathbf{true},\ \beta) \quad\ = \beta \\
\mathit{solveG}\ \sigma\ (X_i = t,\ \beta) \quad\ = \mathit{aunify}(\beta,\ X_i,\ t) \\
\mathit{solveG}\ \sigma\ (p(\bar{X}),\ \beta) \quad\ = \mathit{extG}(\beta,\ \bar{X},\ \sigma\ (p,\ \mathit{restrG}(\beta,\ \bar{X}))) \\
\mathit{solveG}\ \sigma\ ((G_1,\ G_2),\ \beta)\ = \mathit{solveG}\ \sigma\ (G_2,\ \mathit{solveG}\ \sigma\ (G_1,\ \beta))
\end{array}
$$

The abstract semantics $[\![P]\!]_{\mathcal{A}}$ is defined as the least fixpoint of function *solveP*. Existence of the least fixpoint follows from the facts that *Asub* is a finite complete lattice and that all abstract operations are monotonic.

In goal-dependent analyses of logic programs, we are only interested in that part of $[\![P]\!]_{\mathcal{A}}$ that is needed in order to evaluate $[\![P]\!]_{\mathcal{A}}(p,\ \beta)$ for a given initial abstract call $(p,\ \beta)$. Given this part of the abstract semantics, one can compute, for instance, call modes for those predicates in P which are valid with respect to all concrete calls described by the initial abstract call. Therefore, we rephrase $[\![P]\!]_{\mathcal{A}}$ as a system of equations:

$$V = \mathit{Pred} \times \mathit{Asub} \qquad D = \mathit{Asub} \qquad f_x\ \sigma = \mathit{solveP}\ \sigma\ x$$

Since the abstract semantics assigns an abstract substitution to every abstract call $(p,\ \beta)$, the variables of the system of equations are exactly the abstract calls $\mathit{Pred} \times \mathit{Asub}$. The lattice of values of the system is the lattice of abstract substitutions. The right-hand side function associated with variable $(p,\ \beta)$ is simply $\lambda\sigma.\ \mathit{solveP}\ \sigma\ (p,\ \beta)$. Unfortunately, this system of equations is not monotonic. This is a consequence of the fact that semantic function *solveG* is not monotonic in general. However, we can consider ordering "$\leq$" on the set of variables from Section 3 defined by $(p_1,\ \beta_1) \leq (p_2,\ \beta_2)$ iff $p_1 = p_2$ and $\beta_1 \sqsubseteq \beta_2$. It turns out that for every program the corresponding system of equations is weakly monotonic – at least if the abstract operations *aunify, restrG, extG, restrC* and *extC* are monotonic.

The core of the implementation of analyzer generator GENA is module **ASem** which implements the above system of equations. The implementation of **ASem** was easily obtained by rewriting the above denotational semantics as SML code. **ASem** is parametrized on abstract domains and on general equation solvers. As general equation solvers, we considered solvers **W**, **TD**, **WRT** and **W-DFS**. In case of solver **W**, the worklist is implemented as a stack. In order to get complete program analyzers, the first author also implemented various abstract domains. For lack of space, we confined ourselves in this paper to report only on our numbers found for **Pos**[2] [5, 16]. **Pos** is a conceptually simple and elegant abstract domain to compute *groundness* information for Prolog programs

[2] Similar results have been obtained also with other abstract domains, e.g., for sharing analysis.

program	*time*				*#rhs*			
	W	**TD**	**WRT**	**W-DFS**	**W**	**TD**	**WRT**	**W-DFS**
action	16.25	3.23	3.25	3.30	390	198	198	198
aqua-c	147.40	57.40	51.26	53.56	11135	3797	3529	3517
ann	0.44	0.23	0.22	0.23	237	110	107	107
b2	1.29	0.72	0.70	0.68	1000	437	418	418
chat	21.96	12.11	10.23	10.45	2946	1625	1276	1281
chat-parser	4.86	3.10	2.08	1.85	1149	751	501	500
flatten	0.27	0.10	0.10	0.11	207	69	67	67
nand	0.58	0.39	0.35	0.36	148	67	67	67
peep	0.23	0.11	0.11	0.12	67	42	42	41
press	0.66	0.39	0.34	0.36	461	245	245	245
read	0.51	0.26	0.29	0.25	171	88	92	89
readq	0.91	0.55	0.51	0.51	408	239	201	195
scc	0.10	0.09	0.08	0.08	61	36	35	35
sdda	0.21	0.14	0.14	0.14	164	82	82	82

Table 1. Experimental evaluation of **W**, **TD**, **WRT** and **W-DFS**

where abstract substitutions are represented by Boolean functions. Recall that logical variable X is ground with respect to substitution ϑ if $X\vartheta$ does not contain any variable. By plugging implementations of **W**, **TD**, **WRT** and **W-DFS** into **ASem** + **Pos** we obtained four analyzers for Prolog.

9 Comparison and Conclusion

The generated analyzers were tested on large real-world programs. The program *aqua-c* (16.000 lines of code), for example, is the source code of Peter Van Roy's Aquarius Prolog compiler. Programs *read* and *readq* are Prolog readers. *b2* is a large mathematical program and *chat* (5.000 lines of code) is Warren's chat-80 system. The analyzers were run on a Sun 20 with 64MB main memory with SML-NJ-109. Table 1 shows the results of our experiments. Column *time* gives the total running times of the analyzers (in seconds) including system and garbage collection times. Column *rhs* shows how often right-hand sides of the equation system have been evaluated during the solution process.

As a first result, it should be noted that our versions of all four solvers do reasonably well on all given benchmarks. Thus, the *absolute* numbers indicate that analysis engines generated by GENA are efficient enough to be included into production quality compilers. Secondly, we find that the maximal advantage one solver gains over the other on our benchmark programs is approximately a factor of 5 (which indeed is still moderate). Not surprisingly, worklist solver **W** turns out to be the least efficient. Top-down solver **TD** is amazingly fast even despite its bad (theoretical) worst-case complexity. On programs *aqua-c*, *chat* or *chat-parser*, **TD** is outperformed by our new solvers **WRT** and **W-DFS**. The gain in efficiency here ranges between 6% and 30%. There is just one program, namely *read*, where **TD** needs less evaluations of right-hand sides than **WRT**. On all other programs **WRT** evaluates equally many right-hand sides or less. On *chat* and *chat-parser*, the savings are more than 20%. The runtimes of the

two solvers **WRT** and **W-DFS** turn out to be very similar. Note, however, that **W-DFS** – despite (or because of) its static treatment of variable dependencies – is sometimes a tick faster. In fact, the dynamic treatment of variable priorities of solver **WRT** pays off in reducing the number of evaluations of right-hand sides only for program *chat*. Nevertheless we suggest to use solver **WRT** instead of **W-DFS** since it seems to be more robust against non-well-behaving inputs.

To summarize, we found new application independent local solvers for general systems of equations. Both in theory and practice, these algorithms favorably compete with existing algorithms of the same kind, namely worklist solver **W** and topdown solver **TD**. Secondly, we showed that based on such general algorithms, very efficient program analyzers can be generated. Further directions of research must include evaluation of such kind of solvers also in other areas of application. Also, we would like to investigate methods of constructing solutions with are not based on (more or less cleverly guided) iteration.

Acknowledgements. We are indepted for valuable discussions on fixpoint iteration strategies to Andreas Neumann and the German Scholar Olympic Team for Computer Science.

References

1. F. Bourdoncle. *Abstract Interpretation by Dynamic Partitioning.* Journal of Functional Programming, **2**(4), 1992.
2. B. Le Charlier and P. Van Hentenryck. *A Universal Top-Down Fixpoint Algorithm.* Technical report 92–22, Institute of Computer Science, University of Namur, Belgium, 1992.
3. B. Le Charlier and P. Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. *TOPLAS*, 16(1):35–101, 1994.
4. R. Cleaveland and B. Steffen. A Linear-Time Model Checking Algorithm for the Alternation-Free Modal Mu-Calculus. In *CAV'91.* Springer, LNCS 575, 1991.
5. A. Cortesi, G. Filé, and W. Winsborough. Prop revisited: Propositional formulas as abstract domain for groundness analysis. In *LICS'91*, 322–327, Amsterdam, The Netherlands, 1991.
6. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL'77*, 238–252, 1977.
7. P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2):103–179, 1992.
8. A. Dicky. An Algebraic and Algorithmic Method of Analysing Transition Systems. *TCS*, 46:285–303, 1986.
9. Christian Fecht. GENA – a Tool for Generating Prolog Analyzers from Specifications. *SAS'95*, 418–419. Springer Verlang, LNCS 983, 1995.
10. M.S. Hecht. *Flow Analysis of Computer Programs.* Amsterdam: Elsevier North-Holland, 1977.
11. S. Horwitz, A. Demers and T. Teitelbaum. An Efficient General Iteration Algorithm for Dataflow Analysis. *Acta Informatica*, **24**, 679–694, 1987.
12. N. Jørgensen. Finding Fixpoints in Finite Function Spaces Using Neededness Analysis and Chaotic Iteration. In *SAS'94*, 329–345. Springer, LNCS 864, 1994.

13. G.A. Kildall. A Unified Approach to Global Program Optimization. In *POPL'73*, 194–206, 1973.
14. K. Marriott, H. Søndergaard, and N.D. Jones. Denotational Abstract Interpretation of Logic Programs. *ACM Transactions of Programming Languages and Systems*, 16(3):607–648, 1994.
15. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Application*, 189–233. Prentice-Hall, 1981.
16. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Evaluation of the domain Prop. *The Journal of Logic Programming*, 23(3):237–278, 1995.
17. B. Vergauwen, J. Wauman, and J. Lewi. Efficient fixpoint computation. In *SAS'94*, 314–328. Springer, LNCS 864, 1994.
18. R. Wilhelm and D. Maurer. *Compiler Construction*. Addison-Wesley, 1995.