ELSEVIER

Science of Computer Programming 35 (1999) 137–161

Science of Computer Programming

www.elsevier.nl/locate/scico

A faster solver for general systems of equations

Christian Fecht [a], Helmut Seidl [b,*]

[a] *Universität des Saarlandes, Postfach 151150, D-66041 Saarbrücken, Germany*
[b] *Fachbereich IV - Informatik, Universität Trier, D-54286 Trier, Germany*

Abstract

We present a new algorithm which computes a partial approximate solution for a system of equations. It is *local* in that it considers just as many 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 theoretical worstcase complexity known for similar algorithms. For the application of analyzing logic languages, it also gives the best practical results on most of our real-world benchmark programs.

Keywords: Generic local fixpoint iteration; Worklist algorithms; Abstract interpretation; Program analysis

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 [24], control flow resp. data flow information for imperative programs [14, 18, 21], abstract interpretation [8] of functional and logic languages, and system verification [1, 10, 20].

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

* Corresponding author.

0167-6423/99/$ - see front matter
PII: S0167-6423(99)00009-X

cannot 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 for S. If system S is monotonic, the solver should return the least solution.

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 the system of equations 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 [5] and an enhanced version **W** of Kildall's worklist algorithm [17, 18, 23]. The first one has been successfully used to implement analyzers for logic languages [6] 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 most of our benchmark programs. Similar to the known solvers **W** and **TD**, our new solver **WRT** is both *local* and guided by *dynamic* dependences 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 dependences as in the original global version of **W** [18] is no longer feasible. It may even happen that variable dependences change unpredictably during execution of the algorithm. Therefore, dependences 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 describes further optimizations and enhancements of solver algorithms. Section 9 explains how the implementation of generic solvers is supported by the programming language SML. Section 10 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 11 summarizes results from our practical experiments. Especially, it compares all three solvers and concludes.

A preliminary version of this paper appeared in [13].

2. Systems of equations

Assume D is a complete lattice. Operationally, D is given as an abstract datatype consisting of a set of values (denoted by D as well) together with a designated least element $\bot$, an equality predicate "$=$", and a binary least upper bound operation "$\sqcup$". Note that 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 [5, 17, 23].

A *system of equations* S is given as a pair $(V, \mathscr{F})$ where V denotes a set of variables and $\mathscr{F}$ 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", $\mathscr{F}$ 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 implementing 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 implementing 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 [5], we present our solvers in a very general setting. Nevertheless, we insist on the following three further assumptions:

(i) set V of variables is always finite;
(ii) the complete lattice has finite height h, i.e., every strictly increasing sequence contains at most $h + 1$ elements;
(iii) evaluation of right hand sides is always terminating.

All three assumptions are satisfied in many applications. In [5], 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.

With system S we associate function $G_S : (V \to D) \to V \to D$ defined by $G_S \sigma x = f_x \sigma$. Then the solutions and approximate solutions of S are given by the fixpoints and *post-fixpoints*, respectively, of G_S. In general, sequence $G_S^n \perp$, $n \geqslant 0$, ($\perp$ denotes the minimal variable assignment mapping every x onto $\perp$) 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 the 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

$$\sigma_1\langle 0\rangle = 1, \quad \sigma_1\langle 1\rangle = 1, \quad \sigma_1\langle 2\rangle = 0,$$

$$\sigma_2\langle 0\rangle = 1, \quad \sigma_2\langle 1\rangle = 2, \quad \sigma_2\langle 2\rangle = 0.$$

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 "$\leqslant$" 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 $x \leqslant x'$ implies $\sigma x \sqsubseteq \sigma x'$. G_S (and equally well S) is called *weakly monotonic* with respect to variable ordering "$\leqslant$" iff G_S has the following properties:

- if $x \leqslant y$ 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 "$\leqslant$" given by $\langle d_1\rangle \leqslant \langle d_2\rangle$ iff $d_1 \sqsubseteq d_2$. Also, both the systems of equations investigated by Jørgensen in his paper on fixpoints in finite function spaces [17] and those systems derived in Section 10 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 $\langle p, \beta\rangle$ where p is a predicate symbol and β an abstract substitution. Similar to the small example above, the ordering is given by $\langle p_1, \beta_1\rangle \leqslant \langle p_2, \beta_2\rangle$ 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*:

(i) *sequence $G_S^n \perp$, $n \geqslant 0$, is ascending*;

(ii) *S has a least approximate solution μ which is also a solution of S and monotonic. μ is given by $\mu = G_S^n \perp$ for some $n \leqslant h \cdot \#V$.*

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 $S = (V, \mathscr{F})$ where the set V of variables is tremendously large. One way to deal with large sets of variables is *dynamic partitioning* as described in [2]. 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. Indeed, a similar idea for first-order functional languages has been called *minimal function graph* by Jones and Mycroft [16].

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 σx, $x \in V$. 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 define $dep(f, \sigma)$ as the set of all variables y for which values σy are accessed during the evaluation of f on input σ. In Appendix A, we examplify this concept for a simple expression language which, nevertheless, is expressive enough to formalize, e.g., the abstract semantics of Prolog. In fact, it is this notion of variable dependence which does not refer to f considered as a function but to f considered as an algorithm. To make it precise, we therefore need to refer to the *operational* semantics of right-hand sides (see the proof of Theorem 4 in Appendix B).

Given a partial variable assignment $\sigma : V \rightsquigarrow D$, the *dependence graph* $G(\sigma)$ (relative to σ) is defined as follows. The set of nodes of $G(\sigma)$ is given by V whereas the set of edges consists of all pairs (y, x) such that $f_x\,\sigma$ is defined and $y \in dep(f_x, \sigma)$. Variable y is said to *influence* variable x relative to σ iff there is a path in $G(\sigma)$ from y to x. Let $X \subseteq V$ denote the set of variables which we are interested in. Then (partial)

variable assignment σ is called *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:

(i) σ is X-stable;

(ii) 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*.

We formally compare the different solvers with respect to their worstcase complexities. To do so, we make the following assumptions:

(i) We only count the number of evaluations of right-hand sides f_x.

(ii) For every x, the total number of variables accessed during evaluations of right-hand side f_x is bounded by c.

The first assumption 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.

In general concerning the second assumption, we just know that $c \leqslant N$ where N is the total number of variables considered by the solver. For our example operational semantics, however, given in Appendix A, we find $c \leqslant d^{\mathcal{O}(1)}$ where d is the maximal number of updates of variables σx, and we even get $c \leqslant \mathcal{O}(1)$ for systems of equations with *static* variable dependences. The latter means that sets $dep(f, \sigma)$ are independent of σ. In our applications (and for our solvers), we found results close to the static case, i.e., the sets of variable dependences did change but "not very often".

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 [17] and by Vergauwen, Wauman and Lewi in [23] in a monotonic setting. **W** extends the usual worklist algorithm for systems with statically known variable dependences, e.g., the one of Kildall [18] and his followers, to the case where system S is dynamically constructed and dependences 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 σx together with a set $infl(x)$ of certain variables y such that the

```
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**.

evaluation of f_y (on σ) may access value σx or more formally, x may be contained in $dep(f_y, \sigma)$.

As long as W is non-empty, 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 σx 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_x may access; second, we have to determine all y such that $f_x \sigma$ depends on σy. 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 is not the case, y is added to the domain of σ and σy is set to some safe initial value (e.g., $\perp$). Also, variable $infl(y)$ 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 $y \in dom(\sigma)$ or not), x is added to $infl(y)$, and the value of σ for y (which is now always defined) is returned.

Remark. We use accumulating updates of entries σx 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 $infl(x)$ are emptied after use which is also the case in Vergauwen, Wauman and Lewi's variant, but not in Jørgensen's.

Generalizing the proof of [17], we find:

Theorem 2. (i) *Algorithm* **W** *is a solver.*

(ii) **W** *terminates after at most* $\mathcal{O}(c \cdot d \cdot N)$ *steps where* $N \leqslant \#V$ *is the number of considered variables and* $d \leqslant h$ *is the maximal number of updates of* σx *for some variable* x.

6. The topdown solver TD

Algorithm **TD** (Fig. 2) was proposed by Le Charlier and Van Hentenryck [5] and applied by them to the analysis of logic programs [6]. Algorithm **TD** is an improved version of an algorithm of Bruynooghe et al. [4].

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 implement this idea, the algorithm must take precaution that no new iteration is started for variable y provided

(i) one iteration process for y has already been initiated; or

(ii) iteration for y will not result in a new value for y, i.e. y is stable with respect to the current σ.

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 $dom(\sigma)$, σx 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 σx (and hence strictly larger), then the value of σ for x is updated; moreover all values of σ for variables (possibly) affected by this update are *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 $infl(x)$

```
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
  let t = infl(x) in
    infl(x) := ∅;
    foreach y ∈ t do Stable := Stable − {y}; destabilize(y) od
  end
end;

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

Fig. 2. Algorithm **TD**.

and for every $y \in I$ also all elements from $infl(y)$. Now destabilization means that all variables y in $infl^+(x)$ are removed from *Stable*. Additionally, all their sets $infl(y)$ are reset 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 σy.

Remark. Our presentation of **TD** is closely related to that of Le Charlier and Van Hentenryck in [5]. We only removed all monotonicity constraints (which to maintain can be costly in practice). Also – at least to our taste – our treatment of variable dependences is much more elegant.

Similar to [5] we find:

Theorem 3. (i) *Algorithm* **TD** *is a solver.*

(ii) **TD** *terminates after at most* $\mathcal{O}(d \cdot N^2)$ *steps where* $N \leqslant \#V$ *is the number of considered variables and* $d \leqslant h$ *is the maximal number of updates of* σx *for some variable* x.

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 11). Theoretically, it can be proven to be "optimal" for the case of acyclic variable dependences – 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 strongly connected 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 static systems of equations with *acyclic* variable dependences.

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 dependences. It does not, however, guarantee that (still in case of static variable dependences) iteration is performed in one strongly connected component after the other. The best idea therefore might be to add something like an algorithm detecting strongly connected components "on the fly". Since dependences vary over time in that dependences 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 added to W and afterwards reset to $\emptyset$. Then the top element is popped from *Stack*.

```
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.

The modification now is that procedure *solve* need not return immediately. Instead if *Stack* is non-empty, 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 greater 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 in $dom(\sigma)$, function *eval* not only initializes both values σy and $infl(y)$, but additionally calls $solve(y)$ before returning current value σy.

Remark. Solver **WRT** has no equivalent either in Le Charlier and Van Hentenryck's paper [5], Jørgensen's work [17] or Vergauwen, Wauman and Lewi's overview [23]. It can be seen as an effort to combine the nice features both of 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 **WDFS** can be obtained from **WRT** if each variable obtains a new time stamp only once, namely when it is added to $dom(\sigma)$. Algorithm **WDFS** can be seen as the dynamic version of *priority-queue iteration* in [15]. Two variables are arranged by **WDFS** according to the *first* dependency between them that was established by the algorithm. This is a reasonable choice in case of static variable dependences. However, in the dynamic case dependences may arbitrarily change during the fixpoint iteration, meaning that any worklist algorithm with fixed variable priorities may show poor performance.

A proof of the following theorem is contained in Appendix B.

Theorem 4. (i) *Algorithm* **WRT** *is a solver.*

(ii) **WRT** *terminates after at most* $\mathcal{O}(c \cdot d \cdot N)$ *steps where* $N \leqslant \#V$ *is the number of considered variables and* $d \leqslant h$ *is the maximal number of updates of* σx *for some variable* x.

The difference between the iteration strategies of solvers **TD** and **WRT** can already be illustrated for equation systems with *static* variable dependences. Here, execution of **TD** corresponds to an iteration according to a *dual* weak topological ordering. *Weak topological orderings* (wtos) are generalizations of topological orderings of acyclic directed graphs to directed graphs containing cycles. They have been suggested by Bourdoncle in [3] as specifications of iteration strategies. *Dual* weak topological orderings (dwtos) differ from wtos only in that they specify repeat-loops for iteration in components instead of while-loops as wtos do. Observe, however, that the dwto for **TD** to guide iteration is not determined beforehand but realized *on the fly* during execution of the algorithm.

Consider, e.g., a system of equations with variable dependences as depicted in Fig. 4 where x is the only initially interesting variable. Then the dwto of **TD** is given by $(z\,(y)\,x)$ with the following meaning. **TD** repeatedly iterates on the strongly connected component z, y, x. Each iteration first visits z, then descends to a subiteration on the subcomponent consisting of variable y, followed by evaluation of variable x.

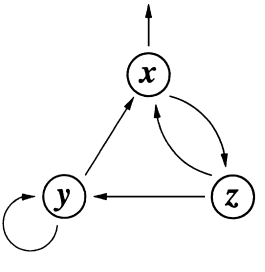

Fig. 4. An example dependence graph.

The whole iteration is completed as soon as variable x is stable. For a more formal treatment of dwtos and their relationship to **TD** see Appendix C.

In contrast, solver **WRT** (similar to other worklist-based solvers) potentially shows a much more irregular treatment of strongly connected components. For the given (admittedly tiny) example, it proceeds in two phases. In the first phase, analogously to **TD**, it descends to z, continues with an iteration on y until stabilization, followed by an evaluation of x. If after this phase x has received a non-$\perp$-value, the second phase is initiated. The second phase repeatedly executes the following block. It starts with an iteration on the two variables x and z. Whenever this led to a new value for z, a further iteration on y follows. If by the latter iteration, the value of y has changed, x is recomputed. Finally if the value of x has changed, a new iteration of the whole block is initiated.

Clearly, it is debatable whether in this example a separate iteration on the subcomponent x,z reduces the overall necessary number of evaluations. One obvious advantage of **WRT** over **TD**, however, is the availability of "short-cuts". If, e.g., the value of x has changed at the end of the execution of one iteration block of the second phase of **WRT**, but reevaluation of z does not lead to a new value, then fixpoint iteration immediately terminates. Opposed to that, if solver **TD** finds a new value of x then the whole component is destabilized. Consequently, all variables have to be reevaluated.

Finally, it should be noted that the motivation in [3] was to find a small subset of variables where *widenings* should be placed at in order to enforce termination even in presence of infinite ascending chains. Given a wto (or a dwto as in the example), widenings only have to be placed at "heads of components", i.e., at just one variable per specified loop. In our case these heads are y and x. It remains as an interesting open problem to exhibit "good" heuristics for the placement of widenings also in the setting of worklist-based local solvers.

8. Improvements and extensions

A more precise treatment of (possible) variable dependences is possible if we additionally maintain sets $dep(x)$ containing the set of variables accessed during the last evaluation of f_x.

A first strategy tries to avoid reevaluation of variables initiated by variable dependences which are no longer valid. Whenever $\sigma\, y$ has changed its value, only those elements $x \in infl(y)$ are now put into worklist W which have accessed variable y during the *last* evaluation of their right-hand sides, i.e., for which $y \in dep(x)$. This improvement can be added to solvers **WRT** and **WDFS**. Accordingly, procedure *destabilize* of topdown solver **TD** can be modified to recursively destabilize only along edges in the *current* dependence graph, i.e., along those (y,x) where $x \in infl(y)$ as well as $y \in dep(x)$.

A more aggressive strategy tries to avoid reevaluation of variables which (at the current stage of iteration) are *dead*. A variable y is *dead* at a certain point in the iteration process if no interesting variable transitively depends on it (w.r.t. the current variable assignment). To the contrary, *live* variables y are *reachable* from the interesting variables. Since dead variables do not influence the values of the interesting variables, reevaluation of dead variables should be avoided. *Search space reduction* tries to detect dead variables and remove them from the iteration process. Search space reduction is automatically taken care of by topdown solver **TD**. As an extra optimization, it can be added to the worklist-based solvers. Whenever a variable y is extracted from the worklist for reevaluation, its liveness is checked first. If y is definitively dead, reevaluation is abandoned. Observe that although y may be dead at one point of iteration, it eventually may become alive at another. In this case, the right-hand side of y must be reevaluated.

Liveness of variables is a *global* property. Therefore, it may be too expensive to compute liveness exactly. In analogy to reference counting for garbage collection, approximate liveness information can be obtained from the *infl*-sets. To this end, it makes sense to keep the *infl*-sets as small as possible. Assume variable x with current set $dep(x) = A_1$ is reevaluated and during this reevaluation the variables from set A_2 have been accessed. Then x should be removed from all sets $infl(y), y \in A_1 \backslash A_2$. If $infl(y) = \emptyset$, then y is definitively dead. In this case, also all occurrences of y in sets $infl(z), z \in dep(y)$, can be removed.

9. Implementation

We briefly point out how reusable implementations of solvers can be obtained in the programming language SML. Any implementation of a solver must represent systems of equations in some way, i.e. it has to implement variables, complete lattices, assignments $\sigma : V \to D$, right-hand sides $f_x : (V \to D) \to D$, and V-indexed families $(f_x)_{x \in V}$ of right-hand sides.

Variables and lattices are implemented by structures which meet the signature VARIABLE and LATTICE (Fig. 5). For efficiency reasons, we require every variable to have a unique identifier which should be a nonnegative number. This unique identifier of a variable is accessed via the function id. An implementation of complete lattices must provide a type lattice, the bottom element bottom of the lattice, an equality function same, and the least upper bound operation lub.

With these structures one can represent the remaining concepts as SML values of the following types.

```
assignments : V.variable -> L.lattice
f_x         : (V.variable -> L.lattice) -> L.lattice
(f_x)_{x∈V} : V.variable -> (V.variable -> L.lattice) -> L.lattice
```

```
signature VARIABLE =
  sig
    type variable
    val id : variable -> int
  end
signature LATTICE =
  sig
    type lattice
    val bottom : lattice
    val lub    : lattice * lattice -> lattice
    val same   : lattice * lattice -> bool
  end
```

Fig. 5. The abstract interface of variables and complete lattices.

```
signature SOLVER =
  sig
    structure V : VARIABLE
    structure L : LATTICE
    val solve : (V.variable -> (V.variable -> L.lattice) ->L.lattice)
                -> V.variable list -> (V.variable * L.lattice) list
  end
```

Fig. 6. The interface to the generic solver.

A solver is implemented by an SML functor Solver with the following head:

```
functor Solver(structure V: VARIABLE
               structure L: LATTICE): SOLVER
```

Functor Solver takes structures V and L as inputs and returns the actual solver which has signature SOLVER (Fig. 6). There is only one function solve which is called with two arguments: solve f varList. Thereby, f represents the system of equations and varList is the list of interesting variables. The call solve f varList returns the approximate solution as a list of variable-value pairs. The user can customize the solver to a specific application by implementing structures Variable and Lattice as defined by his application and then applying functor Solver to them.

10. Generic abstract interpretation of logic programs

As an example application for our implementations of solvers **W**, **TD**, **WRT** and **WDFS** the first author integrated the four solvers into a tool (GENA) [11, 12] for generating Prolog analyzers from specifications. The generated analyzers are based on the principle of abstract interpretation [6, 9, 19]. More specifically, the analyzers are based on the generic abstract interpretation framework for logic programs of Le Charlier and Van Hentenryck [6]. This framework assumes logic programs to be *normalized*. The set of *normalized goals* is inductively defined by the following rules:

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

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 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 $\mathscr{A}$ is a function $[\![P]\!]_{\mathscr{A}} : Pred \times Asub \to Asub$. It assigns an abstract success substitution to every abstract call. The abstract semantics $[\![P]\!]_{\mathscr{A}}$ is defined denotationally by means of functions $solveP$, $solveC$, and $solveG$ which define the meaning of predicates, clauses, and goals, respectively.

$$
\begin{array}{l}
solveP :: (Pred \times Asub \to Asub) \to (Pred \times Asub \to Asub) \\
solveP\ \sigma\ \langle p, \beta \rangle = \bigsqcup \{ solveC\ \sigma\ \langle c, \beta \rangle \mid c \in Clause \text{ and head predicate of } c \text{ is } p \} \\
\\
solveC :: (Pred \times Asub \to Asub) \to (Clause \times Asub \to Asub) \\
solveC\ \sigma\ \langle p(X_1, \ldots, X_n) \leftarrow G, \beta_{in} \rangle = restrC(\beta_{exit}, n) \\
\quad \text{where } \beta_{exit} = solveG\ \sigma\ \langle G, \beta_{entry} \rangle \\
\qquad \beta_{entry} = extC(\beta_{in}, p(X_1, \ldots, X_n) \leftarrow G) \\
\\
solveG :: (Pred \times Asub \to Asub) \to (Goal \times Asub \to Asub) \\
solveG\ \sigma\ \langle \textbf{true}, \beta \rangle = \beta \\
solveG\ \sigma\ \langle X_i = t, \beta \rangle = aunify(\beta, X_i, t) \\
solveG\ \sigma\ \langle p(\bar{X}), \beta \rangle = extG(\beta, \bar{X}, \sigma \langle p, restrG(\beta, \bar{X}) \rangle) \\
solveG\ \sigma\ \langle (G_1, G_2), \beta \rangle = solveG\ \ \sigma\ \ \langle G_2, solveG\ \ \sigma\ \ \langle G_1, \beta \rangle \rangle
\end{array}
$$

The abstract semantics $[\![P]\!]_{\mathscr{A}}$ 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 $[\![P]\!]_{\mathscr{A}}$ that is needed in order to evaluate $[\![P]\!]_{\mathscr{A}} \langle p, \beta \rangle$ for an initial abstract call $\langle p, \beta \rangle$. 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 $[\![P]\!]_{\mathscr{A}}$ as a system of equations:

$$V = Pred \times Asub, \qquad D = Asub, \qquad f_x \sigma = solveP\ \sigma\ x.$$

Since the abstract semantics assigns an abstract substitution to every abstract call $\langle p, \beta \rangle$, the variables of the system of equations are exactly the abstract calls $Pred \times Asub$. The

lattice of values of the system is the lattice of abstract substitutions. The right-hand side function associated with variable $\langle p, \beta \rangle$ is simply $\lambda\sigma.\ solveP\,\sigma\,\langle p, \beta \rangle$. 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 "$\leqslant$" on the set of variables from Section 3 defined by $\langle p_1, \beta_1 \rangle \leqslant \langle p_2, \beta_2 \rangle$ 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** is easily obtained by rewriting the above denotational semantics as SML code. **ASem** is parametrized on abstract domains and on generic equation solvers. As generic equation solvers, we considered solvers **W**, **TD**, **WRT** and **WDFS**. 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 papaer to report only on our numbers found for **POS**. Further numbers for various sharing domains can be found in [12]. **POS** is a conceptually simple and elegant abstract domain to compute *groundness* information for Prolog programs where abstract substitutions are represented by Boolean functions [7, 22]. 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 **WDFS** into **ASem**+**POS** we obtained four analyzers for Prolog.

11. 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. *b*2 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 r.h.s. 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

Table 1
Experimental evaluation of **W**, **TD**, **WRT** and **WDFS**

Program	Time				# r.h.s.			
	W	**TD**	**WRT**	**WDFS**	**W**	**TD**	**WRT**	**WDFS**
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

solvers **WRT** and **WDFS**. 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 **WDFS** turn out to be very similar. Note, however, that **WDFS** – despite (or because of) its static treatment of variable priorities – 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 **WDFS** 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 which are not based on (more or less cleverly guided) iteration.

Acknowledgements

We are indebted for valuable discussions on fixpoint iteration strategies to Andreas Neumann and the German Scholar Olympic Team for Computer Science. We also thank the anonymous referee for insisting on a clarification of the relationship to weak topological orderings.

Appendix A. An example semantics for right-hand sides

Assume the set V of variables is given by $V = P \times D$ where P denotes a (finite) set of names for unary transformers in $D \to D$. Variable $\langle p, d \rangle$ can be thought of as the entry d of the value table for p. Such kinds of variables occur both in the analysis of Prolog as of imperative languages. In both applications, right-hand side $f_{\langle p,d \rangle}$ for variable $\langle p, d \rangle$ is specified by means of a pair $\langle e_p, d \rangle$ where e_p is an expression constructed according to the following grammar:

$$
\begin{array}{lll}
e ::= & \mathsf{self} & (\textit{projection}) \\
\mid & d' & (\textit{basic value}) \\
\mid & \square(e_1, \ldots, e_k) & (\textit{operator application}) \\
\mid & \langle p', e \rangle & (\textit{function call})
\end{array}
$$

Here, d' and $\square$ denote fixed elements from D resp. fixed operators from $D^k \to D$. For convenience, we do not distinguish (notationally) between these symbols and their meanings. The (operational) semantics for $\langle e, d \rangle$ defines a mapping $[\![\langle e, d \rangle]\!] : (V \to D) \to D$. Using recursive descent, $[\![\langle e, d \rangle]\!]\sigma$ is evaluated according to the following rules:

$$
\begin{array}{ll}
[\![\langle \mathsf{self}, d \rangle]\!]\sigma & = d \\
[\![\langle d', d \rangle]\!]\sigma & = d' \\
[\![\langle \square(e_1, \ldots, e_k), d \rangle]\!]\sigma & = \square([\![\langle e_1, d \rangle]\!]\sigma, \ldots, [\![\langle e_k, d \rangle]\!]\sigma) \\
[\![\langle \langle p, e \rangle, d \rangle]\!]\sigma & = \sigma\langle p, [\![\langle e, d \rangle]\!]\sigma \rangle
\end{array}
$$

By induction on the structure of expression e, we find that the set $dep(\langle e, d \rangle, \sigma)$ of variables which are accessed when evaluating $[\![\langle e, d \rangle]\!]\sigma$ is given by:

$$
\begin{array}{ll}
dep(\langle \mathsf{self}, d \rangle, \sigma) & = \emptyset \\
dep(\langle d', d \rangle, \sigma) & = \emptyset \\
dep(\langle \square(e_1, \ldots, e_k), d \rangle, \sigma) & = dep(\langle e_1, d \rangle, \sigma) \cup \cdots \cup dep(\langle e_k, d \rangle, \sigma) \\
dep(\langle \langle p, e \rangle, d \rangle, \sigma) & = dep(\langle e, d \rangle, \sigma) \cup \{\langle p, [\![\langle e, d \rangle]\!]\sigma \rangle\}
\end{array}
$$

Appendix B. Proof of Theorem 4

For $j \geqslant 0$, let σ_j, infl_j, R_j and W_j denote the values of σ, infl, the recursion stack and the worklist, respectively, after the jth update of values σx. Furthermore, let $N \leqslant \#V$ denote the number of variables considered by **WRT**, $d \leqslant h$ the maximal number of updates of variables, and c the maximal number of variables accessed to evaluate some right-hand side. We start by analyzing the complexity of algorithm **WRT**. Worklist W receives new elements only if an update of some value σy has occurred. For y, this may happen at most d times. Thus, every variable x depending on y is added at most d times to worklist W. By our assumptions, every variable x may depend at most on c variables. It follows that x is added at most $c \cdot d$ times to W. Hence, there are at

most $c \cdot d \cdot N$ insertions into W. Furthermore, we observe that for all $j \geqslant 0$:

(1) $dom(\sigma_j) \subseteq dom(\sigma_{j+1})$;

(2) whenever $x \in dom(\sigma_j)$ then $(\sigma_i x)_{i \geqslant j}$ is an ascending chain.

Since any evaluation of a right-hand side either corresponds to a variable taken from W or a new variable, we conclude that algorithm **WRT** performs at most $\mathcal{O}(c \cdot d \cdot N)$ evaluations of right-hand sides. Note here, that, instead of factor c in the estimation of the complexity we could as well have taken an upper bound to the sizes of *infl*-sets of variables changing their values – a value, which we also found to be small in our applications (however for different reasons than c).

Let us now verify that variable dependences are maintained correctly. To reason formally about variable dependences, we assume that every evaluation of a right-hand side f_x has the form $\pi = \gamma_0 \lambda_1 \ldots \gamma_{k-1} \lambda_k \gamma_k$ where γ_j are internal computations, λ_j are look-ups of variables y_j, and γ_k eventually returns the result. Clearly, which variable intermediately is looked up depends on the current state of the computation as well as the remaining part of the computation depends on the value received through the look-up. If π has been completed on variable assignment σ then set $dep(f_x, \sigma)$ consists of all variables looked up by $\lambda_1, \ldots, \lambda_k$, namely, $dep(f_x, \sigma) = \{y_1, \ldots, y_k\}$.

For every variable x put onto the recursion stack, a prefix π_x of such a computation for the corresponding right-hand side f_x has already been executed. Then the following invariant can be verified:

(3) If x is not contained in R_j then $y \in dep(f_x, \sigma_j)$ implies that either $x \in infl_j(y)$ or $x \in W_j$.

If x is contained in R_j and y is looked up in π_x then either $x \in infl_j(y)$ or $x \in W_j$ or y is the last variable looked up in π_x, and x is immediately followed by y in R_j.

Note that the first part of this invariant is the usual one for worklist-based solvers whereas the second one takes care of the recursion stack.

Using this invariant, we then deduce that

(4) If x is contained in $dom(\sigma_j)$ but neither contained in W_j nor in R_j then f_x is defined for σ_j and $\sigma_j\, x \sqsupseteq f_x\, \sigma_j$.

Next, we would like to verify that evaluation of every right-hand side f_x indeed terminates. Recall that we only demanded f_x to terminate on *variable assignments*. Thus, in order to guarantee termination, we have to prove that the same values are returned for *all* accesses to a variable y during one evaluation.

To this end, consider a prefix $\pi = \gamma_0 \lambda_1 \ldots \gamma_{k-1} \lambda_k$ of the computation induced by the evaluation of f_x. Assume λ_k is an access to variable y, and Y is the set of all variables accessed by $\lambda_1, \ldots, \lambda_{k-1}$ for which procedure *solve* has been called. Furthermore, let $\bar{Y}$ denote the set of variables whose evaluation has been initiated by the evaluation of variables in Y. Then the following holds before evaluating λ_k:

– $\bar{Y}$ is the set of all variables whose timestamps exceed the timestamp of x;
– no variable from $\bar{Y}$ is contained in the worklist.

Now consider the evaluation of the variable access λ_k:

- if σ for y is defined, then the value is directly accessed; no reevaluation of variables is initiated;
- if σ for y has not been defined so far, then procedure *solve* for y is called; during this call, *solve* may iteratively be called for y but never for variables which have already been defined before the first call $solve(y)$.

From the above statements we especially deduce that none of the values of variables once accessed during evaluation of right-hand side f_x is changed during this evaluation.

Now let $\bar{\sigma}$ denote the partial variable assignment after termination of algorithm **WRT**. We want to prove that $\bar{\sigma}$ is X-stable. Since eventually both the recursion stack and the worklist are empty, we conclude from (4) that for every $y \in dom(\bar{\sigma})$, $f_y\,\bar{\sigma}$ is defined and $\bar{\sigma}\,y \sqsupseteq f_y\,\bar{\sigma}$. Hence, $\bar{\sigma}$ is $dom(\bar{\sigma})$-stable and therefore also X-stable.

Finally assume that set V of variables is equipped with some partial ordering "$\leqslant$" and S is weakly monotonic relative to "$\leqslant$". Let μ denote the least solution of S. By induction on j we find:

(5) $\sigma_j\,y \sqsubseteq \mu\,y$ for all $y \in dom(\sigma_j)$.

Note that validity of (5) crucially depends on the weak monotonicity of S.

Assume again that $\bar{\sigma}$ is the final partial variable assignment computed by algorithm **WRT**. Define set of variables Y as the complement of $dom(\bar{\sigma})$. Let us now start algorithm **WRT** on set $X \cup Y$ (instead of X) where all variables in Y receive time stamps *smaller* than those in X, and call the resulting variable assignment τ. **WRT** on $X \cup Y$ proceeds as **WRT** on X until W contains just the variables from Y. At that moment, the recursion stack is empty. Since $\bar{\sigma}$ is $dom(\bar{\sigma})$-stable no variable from $dom(\bar{\sigma})$ ever will be put into W or the recursion stack R anymore. We conclude that τ is a total variable assignment where $\tau\,y = \bar{\sigma}\,y$ for all $y \in dom(\bar{\sigma})$.

It remains to show that τ is equal to least solution μ of S. By (5) (instantiated with $X \cup Y$ as set of interesting variables) we definitely know that $\tau \sqsubseteq \mu$. Since $\tau\,y \sqsupseteq f_y\,\tau$ for all $y \in V$, τ is an approximate solution of S. Since μ is the least approximate solution, we have $\mu \sqsubseteq \tau$ as well and equality follows.

This completes the proof of Theorem 4. Note that an analogous argumentation also proves correctness and the same complexity statement also for **WDFS**. □

Appendix C. Dual weak topological orderings

In this section, we formally define *dual* weak topological orderings, corresponding fixpoint iteration strategies and prove that topdown solver **TD** when applied to systems of equations with static variable dependences iterates according to (a slight variation of) such a strategy.

A *dual weak topological ordering* γ of directed graph $g = (V, E)$ (dwto) is defined inductively by:

- if $V = \emptyset$, then $\gamma = \varepsilon$;
- if $V = \{x\}$ and $E = \emptyset$, then $\gamma = x$;

- if g is strongly connected containing at least one edge, then $\gamma = (\gamma' x)$ is dwto of g for every $x \in V$ and every γ' which is a dwto of the subgraph of g on $V \setminus \{x\}$.
- otherwise, let $C_1, \ldots, C_k$ denote a topological ordering of the strongly connected components of g. Then $\gamma = \gamma_1 \ldots \gamma_k$ is a dwto of g whenever for every $i = 1, \ldots, k, \gamma_i$ is a dwto w.r.t. C_i.

A dwto $(\gamma' x)$ is also called *component* and x *head* of the component.

Let S denote a system of equations with variables from V and static variable dependences, and assume all variables are initialized with $\bot$. Then every dwto γ of the variable dependence graph of S naturally defines a fixpoint iteration strategy for S as follows.

$\gamma = x\,\gamma'$	Recompute x; do γ'
$\gamma = (\gamma_1\, x)\,\gamma'$	repeat do $\gamma_1\, x$ until x is stable; do γ'
$\gamma = \varepsilon$	;

Note that the difference between dwtos as considered here and wtos as considered by Bourdoncle in [3] implies that we use here a repeat-loop for iteration in components – opposed to while-loops for wtos.

Now we are able to state formally the relationship between solver **TD** and dwtos.

Theorem 5. *Assume S is a static system of equations with variables from V and set of interesting variables X. W.l.o.g. no variable in V is superfluous, i.e., any $y \in V$ influences some $x \in X$. Then there is a dwto γ of the variable dependence graph of S such that the sequence of completions of evaluations of right-hand sides produced by* **TD** *on S equals (a permutation of) the sequence of completions produced by iteration according to γ.*

Observe here that we only made assertions about "completions of evaluations of right-hand sides". This is reasonable since (opposed to iteration according to dwtos) iteration with **TD** does not execute right-hand sides *atomicly* but may recursively descend to recomputations of other variables at variable look-ups.

Proof (*Sketch*)**.** That the iteration strategy of solver **TD** can be specified by a dwto follows, intuitively, from the observation that **TD** *statically* decomposes iterations in strongly connected components into subiterations.

To select a specific dwto, two strategies are needed:

(i) how to select a topological ordering of strongly connected components;

(ii) how to select a head inside a strongly connected component.

For the first strategy **TD** relies on the left-to-right ordering of predecessors of a node according to the temporal ordering of variable accesses. To exhibit the second strategy, let us consider a strongly connected component S. We distinguish two cases.

First, assume that all variables x outside S influencing variables in S are already stable or on the recursion stack. Furthermore, assume that $x \in S$ is the first variable of S for which *solve* has been called. By putting x onto the recursion stack, **TD** selects x as head of component S. Indeed, solver **TD** then starts a subiteration on $S - \{x\}$. Having finished with that, **TD** reevaluates x. If a new value is obtained, destabilization removes all variables in S from set *stable*, and the whole procedure repeats.

The behavior of **TD** is more complicated, if during iteration on component S some variable y is encountered, which has not been considered so far. Due to static variable dependences, however, this may only happen during the *first* iteration on S. Then solver **TD** interrupts the iteration on S to start a subiteration on all variables by which y is influenced. Having finished this subiteration, **TD** resumes iteration on S by accessing the final value of y. Observe that it is this demand-driven treatment of so far unknown variables which makes **TD** inducing a sequence of recomputations of variables which is not exactly the sequence corresponding to a dwto but just a *permutation* of it.

Summarizing, the dwto followed by solver **TD** (upto permutation) can be specified as follows.

Assume we are given a sequence π of DFS-trees of the dependence graph g where an individual DFS-tree t is denoted by $t = \langle \pi_1 x \rangle$ with node x at the root and π_1 as the sequence of DFS-trees whose roots are predecessors of x. Dwto $\gamma[\pi]$ selected by **TD** for π is inductively defined by:

(i) If $\pi = \varepsilon$ (empty sequence of DFS-trees), then $\gamma[\pi] = \varepsilon$.

(ii) If $\pi = t\,\pi', t$ an individual DFS-tree, then $\gamma[\pi] = \gamma[t]\,\gamma[\pi']$.

(iii) Now assume $t = \langle \pi\, x \rangle$ is an individual DFS-tree with root x and sequence of subtrees $\pi = t_1 \ldots t_n$. If x forms a trivial strong component without edges, then $\gamma[t] = \gamma'\, x$ where $\gamma' = \gamma[\pi]$.

Otherwise, every t_j is decomposed into an upper fragment s_j consisting of all nodes in t_j which are strongly connected to x and the sequence π_j of remaining DFS-trees. In case, no node in t_j has an edge to x, we set $s_j = \varepsilon$ and $\pi_j = t_j$. Then $\gamma[t] = \gamma\,(\gamma'\, x)$ where $\gamma = \gamma[\pi_1 \ldots \pi_n]$ and $\gamma' = \gamma[s_1 \ldots s_n]$.

Here, the second case of item (iii) is the only one which establishes a new component of the resulting dwto. The new component consists of head x together with all nodes strongly connected to x.

We omit the enumeration of invariants for **TD** from which the correctness of our claim formally follows. Instead, we illustrate the dwto of **TD** for the example dependence graph in Fig. 7 which is the first example dependence graph from [3]. Assuming that 8 is the only (initially) interesting variable and a suitable ordering of ingoing edges, we obtain just one DFS-tree, namely:

$$t = \langle\langle\langle\langle\langle\langle\langle 1\rangle\, 2\rangle\, 3\rangle\, 4\rangle\, 5\rangle\, 6\rangle\, 7\rangle\, 8\rangle.$$

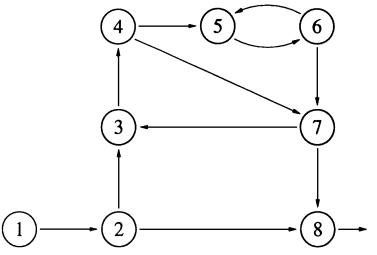

Fig. 7. The example dependence graph from [3], Fig. 1.

The construction of $\gamma[t]$ starts with variable 8. Since there is no edge $\langle 8, i\rangle$, $i \leqslant 8$, we obtain

$$\gamma[t] = \gamma[\langle \ldots 7\rangle]8.$$

Now consider subtree $t' = \langle \ldots 7\rangle$. Indeed, there is an edge (7,3). Therefore, we decompose the subtree $\langle \ldots 6\rangle$ into the upper fragment $s_1 = \langle\langle\langle 3\rangle 4\rangle 5\rangle 6\rangle$ and the remaining lower part $\pi_1 = \langle\langle 1\rangle 2\rangle$. Thus,

$$\gamma[\langle \ldots 7\rangle] = \gamma[\pi_1](\gamma[s_1]\,7).$$

Since $\gamma[\pi_1] = 1\,2$ and $\gamma[s_1] = 3\,4\,(5\,6)$, we finally obtain:

$$\gamma[t] = 1\,2\,(3\,4\,(5\,6)\,7)\,8. \qquad \square$$

References

[1] A. Arnold, P. Crubille, A linear time algorithm for solving fixpoint equations on transition systems, Inform. Process. Lett. 29 (1988) 57–66.
[2] F. Bourdoncle, Abstract interpretation by dynamic partitioning, J. Funct. Programm. 2 (4) (1922) 407–435.
[3] F. Bourdoncle, Efficient chaotic iteration strategies with widenings, in: Internat. Conf. on Formal Methods in Programming and Their Applications, Lecture Notes in Computer Science, Vol. 735, Springer, Berlin, 1993, pp. 128–141.
[4] M. Bruynooghe, G. Janssens, A. Callebaut, B. Demoen, Towards the global optimization of Prolog programs, in: 1987 Symp. on Logic Programming, SLP'87, 1987, pp. 192–204.
[5] B. Le Charlier, P. Van Hentenryck, A universal top-down fixpoint algorithm, Technical Report 92-22, Institute of Computer Science, University of Namur, Belgium, 1992.
[6] B. Le Charlier, P. Van Hentenryck, Experimental evaluation of a generic abstract interpretation algorithm for Prolog, ACM Trans. on Progr. Lang. and Systems (TOPLAS) 16 (1) (1994) 35–101.
[7] A. Cortesi, G. Filé, W. Winsborough, PROP revisited: propositional formulas as abstract domain for groundness analysis, in: 6th Annual IEEE Symp. on Logic in Computer Science (LICS'91), Amsterdam, The Netherlands, 1991, pp. 322–327.
[8] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: 4th Annual ACM Symp. on Principles of Progr. Languages (POPL'77), 1977, pp. 238–252.
[9] P. Cousot, R. Cousot, Abstract interpretation and application to logic programs, J. Logic Programm. 13 (2) (1992) 103–179.
[10] A. Dicky, An algebraic and algorithmic method of analysing transition systems, Theoret. Comput. Sci. 46 (1986) 285–303.

[11] C. Fecht, GENA – a tool for generating Prolog analyzers from specifications, in: Second Internat. Static Analysis Symp. (SAS'95), Lecture Notes in Computer Science, Vol. 983, Springer, Berlin, 1995, pp. 418–419.
[12] C. Fecht, Abstrakte Interpretation logischer Programme: Theorie, Implementierung, Generierung, Ph.D. thesis, Universität des Saarlandes, Saarbrücken, 1997.
[13] C. Fecht, H. Seidl, An even faster solver for general systems of equations, in: Third Internat. Static Analysis Symp. (SAS'96), Lecture Notes in Computer Science, Vol. 1145, Springer, Berlin, 1995, pp. 189–204.
[14] M.S. Hecht, Flow Analysis of Computer Programs, Elsevier, Amsterdam, 1977.
[15] S. Horwitz, A. Demers, T. Teitelbaum, An efficient general iteration algorithm for dataflow analysis, Acta Inform. 24 (1987) 679–694.
[16] N.D. Jones, A. Mycroft, Dataflow analysis of applicative programs using minimal function graphs, in: 13th Internat. ACM Symp. on Principles of Progr. Languages (POPL'86), 1986, pp. 296–306.
[17] N. Jørgensen, Finding fixpoints in finite function spaces using neededness analysis and chaotic iteration, in: First Internat. Static Analysis Symp. (SAS'94), Lecture Notes in Computer Science, Vol. 864, Springer, Berlin, 1994, pp. 329–345.
[18] G.A. Kildall, A unified approach to global program optimization, in: First ACM Symp. on Principles of Progr. Languages (POPL'73), 1973, pp. 194–206.
[19] K. Marriott, H. Søndergaard, N.D. Jones, Denotational abstract interpretation of logic programs, ACM Trans. on Programm. Languages and Systems (TOPLAS) 16 (3) (1994) 607–648.
[20] H. Seidl, Fast and simple nested fixpoints, Inform. Process. Lett. 59 (1996) 303–308.
[21] M. Sharir, A. Pnueli, Two approaches to interprocedural data flow analysis, in: S.S. Muchnick, N.D. Jones (Eds.), Program Flow Analysis: Theory and Application, Prentice-Hall, Englewood Cliffs, NJ, 1981, pp. 189–233.
[22] P. Van Hentenryck, A. Cortesi, B. Le Charlier, Evaluation of the domain PROP, J. Logic Programm. 23 (3) (1995) 237–278.
[23] B. Vergauwen, J. Wauman, J. Lewi, Efficient fixpoint computation, in: First Internat. Static Analysis Symposium (SAS'94), Lecture Notes in Computer Science, Vol. 864, Springer, Berlin, 1994, pp. 314–328.
[24] R. Wilhelm, D. Maurer, Compiler Construction, Addison-Wesley, Reading, MA, 1995.