Nordic Journal of Computing

PROPAGATING DIFFERENCES: AN EFFICIENT NEW FIXPOINT ALGORITHM FOR DISTRIBUTIVE CONSTRAINT SYSTEMS

Christian Fecht
Fachbereich Informatik
Universität des Saarlandes, Postfach 151150, D-66041 Saarbrücken
fecht@cs.uni-sb.de

Helmut Seidl
FB IV - Informatik
Universität Trier, D-54286 Trier
seidl@uni-trier.de

Abstract. Integrating *semi-naive* fixpoint iteration from deductive data bases [3, 2, 4] as well as continuations into worklist-based solvers, we derive a new application independent local fixpoint algorithm for distributive constraint systems. Seemingly different efficient algorithms for abstract interpretation like those for linear constant propagation for imperative languages [16] as well as for control-flow analysis for functional languages [12] turn out to be instances of our scheme. Besides this systematizing contribution we also derive a new efficient algorithm for abstract OLDT-resolution as considered in [14, 15, 24] for Prolog.

1. Introduction

Efficient application independent local solvers for general classes of constraint systems have been successfully used in program analyzers like GAIA [8, 6], PLAIA [19] or GENA [9, 10] for Prolog or PAG [1] for imperative languages. The advantages of application independence are obvious: the algorithmic ideas can be pointed out more clearly and are not superseded by application specific aspects.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, it then can be instantiated to different application domains. Thus, for the overall correctness of the application it simply remains to check whether or not the constraint system correctly models the problem to be analyzed. Reasoning about the solution process itself can be totally abandoned. Therefore, opposed to specialized algorithms, application independent solvers allow for reusable implementations.

In [11], we considered systems of equations of the form $x = f_x$ (x a variable) and tried to minimize the number of evaluations of right-hand sides f_x during the solution process. Accordingly, we viewed these as (almost) atomic actions. In practical applications, however, like the abstract interpretation of Prolog programs, right-hand sides represent complicated functions. In

Received October 1998.

this paper, we therefore try to minimize not just the number of evaluations but the overall work on right-hand sides. Clearly, improvements in this direction can no longer abstract completely from algorithms implementing right-hand sides. Nonetheless, we aim at optimizations in an as application independent setting as possible.

We start by observing that right-hand sides f_x of defining equations $x = f_x$ often are of the form $f_x \equiv t_1 \sqcup \ldots \sqcup t_k$ where the t_i represent independent contributions to the value of x. We take care of that by considering now systems of *constraints* of the form $x \sqsupseteq t$. Having adapted standard worklist-based equation solvers to such constraint systems, we investigate the impact of two further optimizations. First, we try to avoid identical subcomputations which would contribute nothing new to the next iteration. Thus, whenever a variable y accessed during the last evaluation of right-hand side t has changed it's value, we try to avoid reevaluation of t as a whole. Instead, we resume evaluation just with the access to y.

In order to do this in a clean way, we adapt the model of (generalized) *computation trees*. We argue that many common expression languages for right-hand sides can easily and automatically be translated into this model. This model has the advantage to make *continuations*, i.e., remaining parts of computations after returns from variable look-ups, explicit. So far, continuations have not been used in connection with worklist-based solver algorithms. Only for the *topdown*-solver **TD** of Charlier and Hentenryck [5, 11] a related technique has been suggested and practically applied to the analysis of Prolog, by Englebert et al. in [8].

In case, however, computation on larger values is much more expensive than on smaller ones, continuation based worklist solvers can be further improved by calling continuations not with the *complete* new value of the modified variable but just its *increment*. This concept clearly is an instance of the very old idea of optimization through *reduction in strength* as considered, e.g., by Paige [21]. A similar idea has been considered for recursive query evaluation in deductive databases to avoid computing the same tuples again and again [3, 4]. *Semi-naive* iteration, therefore, propagates just those tuples to the respective next iteration which have been newly encountered. Originally, this optimization has been considered for rules of the form $x \supseteq f\ y$ where x and y are mutually recursive relations and unary operator f is distributive, i.e., commutes with "$\cup$":

$$f\ (s_1 \cup s_2) = f\ s_1 \cup f\ s_2$$

An extension to n-ary f is contained in [2]. A general combination, however, of this principle with continuations and worklist solvers seems to be new. In order to make the combination work, we need an operator diff which when applied to abstract values d_1 and d_2 determines the necessary part from $d_1 \sqcup d_2$ given d_1 for which reevaluation should take place. We then provide a set of sufficient conditions guaranteeing the correctness of the resulting algorithm.

Propagating differences, is orthogonal to the other optimizations of worklist solvers considered in [11]. Thus, we are free to add timestamps or just depth-first priorities to obtain even more competitive fixpoint algorithms (see [11] for precise definitions of these ideas). We refrained from doing so to keep the exposition as simple as possible.

We underline generality as well as usefulness of our new fixpoint algorithm by giving three important applications, namely, distributive framework *IDE* for interprocedural analysis of imperative languages [16], control-flow analysis for higher-order functional languages [12], and abstract `OLDT`-resolution as considered in [14, 15] for Prolog. In the first two cases, we obtain similar complexity results as for known special purpose algorithms. Completely new algorithms are obtained for abstract `OLDT`-resolution.

Another effort to exhibit computational similarities at least between control-flow analysis and certain interprocedural analyses has been undertaken by Reps and his coworkers [17, 18]. It is based on the graph-theoretic notion of *context-free language reachability*. This approach, however, is much more limited in its applicability than ours since it does not work for lattices which are *not* of the form $D = 2^A$ for some un-ordered base set A.

The paper is organized as follows. In sections 2 through 6 we introduce our basic concepts. Especially, we introduce the notions of computation trees and weak monotonicity of computation trees. In the following three sections, we successively derive the differential fixpoint algorithm $\mathbf{WR}_\Delta$. The conventional worklist solver **WR** (with recursion) is introduced in Section 7. Continuations are added in Section 8 to obtain solver $\mathbf{WR}_C$ from which we obtain algorithm $\mathbf{WR}_\Delta$ in Section 9. Correctness of algorithm $\mathbf{WR}_\Delta$ is proven under reasonable additional assumptions on the constraint system (Appendix A). The results of Section 9 are sufficient to derive fast algorithms for framework *IDE* (Section 10 and Appendix B) as well as control-flow analysis (Section 11 and Appendix C). Framework *IDE* has been proposed by Horwitz, Reps and Sagiv for interprocedural analysis of imperative programs and applied to interprocedural *linear constant propagation* [16]. A variant of control-flow analysis ("set-based analysis") has been proposed by Heintze for the analysis of ML [12]. Another variant, even more in the spirit of the methods used here, has been applied in [25] to a higher-order functional language with call-by-need semantics to obtain a termination analysis for *deforestation*. In Section 12 we extend the applicability of algorithm $\mathbf{WR}_\Delta$ further by introducing *generalized computation trees*. This generalization takes into account independence of subcomputations. Especially, it allows to derive new algorithms for abstract `OLDT`-resolution [14, 15, 24] (Section 13). As an example implementation, we integrated an enhanced version of fixpoint algorithm $\mathbf{WR}_\Delta$ into program analyzer generator GENA for Prolog [9, 10] and practically evaluated the generated analyzers on our benchmark suite of large Prolog programs. The results are reported in Section 14.

2. Constraint Systems

Assume D is a complete lattice of *values*. A *constraint system* $\mathcal{S}$ with set variables V over lattice D consists of a set of constraints $x \sqsupseteq t$ where left-hand side $x \in V$ is a variable and t, the right-hand side, represents a function $[\![t]\!] : (V \to D) \to D$ from variable assignments to values. In case when the number of constraints is "small", $\mathcal{S}$ can be represented explicitly by listing its constraints. In many interesting cases, however, (especially when set V is big) the set of constraints for a variable is only *implicitly* given through some total function F of type $F : V \to 2^{\mathbf{rhs}}$. The set F_x of right-hand sides t for $x \in V$ then is obtained by $F_x = F\ x$.

Le Charlier and Van Hentenryck in [5] and Fecht and Seidl in [11] presented their solvers in a setting which was (almost) independent of the implementation of right-hand sides. In this paper, we insist on a general formulation as well. As in [11] we assume that

(1) set V of variables is always finite;

(2) complete lattice D has finite height;

(3) evaluation of right-hand sides is always terminating.

In contrast to [5, 11], however, our new algorithms also take into account *how* right-hand sides are evaluated.

3. Computation Trees

Operationally, the evaluation of a right-hand side on a variable assignment can be viewed as a sequence alternating between variable lookups and internal computations which, eventually, terminates to return the result. Formally, this type of evaluation can be represented as a D-branching *computation tree* (ct for short) of finite depth. The set $\mathcal{T}(V, D)$ of all (uniformly bounded) computation trees is the least set $\mathcal{T}$ containing $d \in D$, $x \in V$ together with all pairs $\langle x, C\rangle$ where $x \in V$ and $C : D \to \mathcal{T}$. Given $t \in \mathcal{T}(V, D)$, function $[\![t]\!] : (V \to D) \to D$ *implemented* by t and set $\mathsf{dep}(t, \_)$ of variables *accessed* during evaluation of t are given by:

$$\begin{array}{lcllcl} [\![d]\!]\ \sigma & = & d & \mathsf{dep}(d, \sigma) & = & \emptyset \\ [\![x]\!]\ \sigma & = & \sigma\ x & \mathsf{dep}(x, \sigma) & = & \{x\} \\ [\![\langle x, C\rangle]\!]\ \sigma & = & [\![C\ (\sigma\ x)]\!]\ \sigma & \mathsf{dep}(\langle x, C\rangle, \sigma) & = & \{x\} \cup \mathsf{dep}(C\ (\sigma\ x), \sigma) \end{array}$$

Note that ct x can be viewed as an abbreviation of ct $\langle x, \lambda d.d\rangle$.

Representations of equivalent computation trees can efficiently be obtained for various expression languages.

Example 1. Assume right-hand sides are given by

$$e \ ::= \ d \mid x \mid f\ x \mid g\,(x_1, x_2)$$

where, d denotes an element in D, and f and g represent monotonic functions $D \to D$ and $D^2 \to D$, respectively. For simplicity, we do not distinguish

(notationally) between these symbols and their respective meanings. Standard intra-procedural data-flow analyzes for imperative languages naturally introduce constraint systems of this simple type (usually even without occurrences of *binary* operators g). Given such an expression e, the corresponding computation tree t can be chosen as:

$$\begin{array}{lcll} t & = & e & \text{if} \quad e \in D \cup V \\ t & = & \langle x, f \rangle & \text{if} \quad e \equiv f\,x \\ t & = & \langle x_1, C \rangle & \text{if} \quad e \equiv g\,(x_1, x_2) \text{ where} \\ & & C\ d_1 = \langle x_2, C_{d_1} \rangle \text{ and } C_{d_1}\ d_2 = g\,(d_1, d_2) & \end{array}$$

□

Further examples of useful expression languages together with their translations into (generalized) computation trees can be found in Sections 11, 12, and 13. It should be emphasized that we do not advocate ct's as *specification language* for right-hand sides. In the first place, ct's serve as an abstract notion of algorithm allowing for a convenient representation of fixpoint iterators. In the second place, however, ct's (resp. their generalization as considered in Section 12) can be viewed as the *conceptual* intermediate representation which our algorithms rely on, meaning, that evaluation of right-hand sides should provide for every access to constraint variable y some representation C of the remaining part of the evaluation. As in Example 1, such C typically consists of a tuple of values together with information about the reached program point.

4. Solutions

Variable assignment $\sigma : V \to D$ is called *solution* for $\mathcal{S}$ if $\sigma\ x \sqsupseteq [\![t]\!]\ \sigma$ for all constraints $x \sqsupseteq t$ in $\mathcal{S}$. Note that every system $\mathcal{S}$ has at least one solution, namely the trivial one mapping every variable to $\top$, the top element of D. In general, we are interested in computing a "good" solution, i.e., one which is as small as possible or, at least, non–trivial. With system $\mathcal{S}$ we associate function $G_{\mathcal{S}} : (V \to D) \to V \to D$ defined by $G_{\mathcal{S}}\ \sigma\ x = \bigsqcup\{[\![t]\!]\ \sigma \mid x \sqsupseteq t \in \mathcal{S}\}$. If we are lucky, all right-hand sides t represent monotonic functions. Then $G_{\mathcal{S}}$ is monotonic as well, and therefore has a least fixpoint which is also the least solution of $\mathcal{S}$. As observed in [11], the constraint systems for interprocedural analysis of (imperative or logic) languages often are not monotonic but just *weakly* monotonic.

Example 2. Consider complete lattice $D = \{0 \sqsubset 1 \sqsubset 2\}$ and system $\mathcal{S}$ with variables $V = \{y_d \mid d \in D\}$ and constraints $y_d \sqsupseteq f_d$ where $[\![f_d]\!]\ \sigma = \sigma\ y_{(\sigma y_d)}$. Right-hand sides of this kind are common when analyzing programs with procedures or functions. Variables y_d 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

$$\sigma_1\ y_0 = 1 \qquad \sigma_1\ y_1 = 1 \qquad \sigma_1\ y_2 = 0$$
$$\sigma_2\ y_0 = 1 \qquad \sigma_2\ y_1 = 2 \qquad \sigma_2\ y_2 = 0$$

Clearly, $\sigma_1 \sqsubseteq \sigma_2$, but $G_{\mathcal{S}}\ \sigma_1\ y_1 = 1$ and $G_{\mathcal{S}}\ \sigma_2\ y_1 = 0$. Hence, $G_{\mathcal{S}}$ is not monotonic. □

5. Weak Monotonicity of Computation Trees

Assume we are given a partial ordering "$\leq$" on variables. Variable assignment $\sigma : V \to D$ is called *monotonic* if for all $x_1 \leq x_2$, $\sigma\ x_1 \sqsubseteq \sigma\ x_2$. On computation trees we define a relation "$\leq$" inductively by:

- $\bot \leq t$ for every t; and $d_1 \leq d_2$ if $d_1 \sqsubseteq d_2$;
- $x_1 \leq x_2$ as ct's if also $x_1 \leq x_2$ as variables;
- $\langle x_1, C_1\rangle \leq \langle x_2, C_2\rangle$ if $x_1 \leq x_2$ and $C_1\ d_1 \leq C_2\ d_2$ for all $d_1 \sqsubseteq d_2$.

Constraint system $\mathcal{S}$ is called *weakly* monotonic iff for every $x_1 \leq x_2$ and constraint $x_1 \sqsupseteq t_1$ in $\mathcal{S}$, some constraint $x_2 \sqsupseteq t_2 \in \mathcal{S}$ exists such that $t_1 \leq t_2$. $\mathcal{S}$ is called *monotonic* if it is weakly monotonic for variable ordering "=".

Example 3. Recall the constraints from Example 2. A natural variable ordering is given by $y_{d_1} \leq y_{d_2}$ iff $d_1 \sqsubseteq d_2$. For $d \in D$, right-hand side f_d can be represented by ct $t_d = \langle y_d, C\rangle$ with $C\ d' = y_{d'}$. Clearly, $t_{d_1} \leq t_{d_2}$ provided $d_1 \sqsubseteq d_2$. We find: $[\![t_{d_1}]\!]\,\sigma_1 \sqsubseteq [\![t_{d_2}]\!]\,\sigma_2$ whenever $d_1 \leq d_2$, $\sigma_1 \sqsubseteq \sigma_2$ and at least one of the σ_i is monotonic. □

The observation from Example 3 can be generalized.

Proposition 1. Assume σ_1, σ_2 are variable assignments where $\sigma_1 \sqsubseteq \sigma_2$ and at least one of the σ_i is monotonic. Then $t_1 \leq t_2$ implies:

(1) $\mathsf{dep}(t_1, \sigma_1) \leq \mathsf{dep}(t_2, \sigma_2)$;

(2) $[\![t_1]\!]\ \sigma_1 \sqsubseteq [\![t_2]\!]\ \sigma_2$. □

Here, $s_1 \leq s_2$ for sets $s_1, s_2 \subseteq V$ iff for all $x_1 \in s_1$, $x_1 \leq x_2$ for some $x_2 \in s_2$. Semantic property 2 of Prop. 1 coincides with what has been called "weakly monotonic" in [11] – adapted to constraint systems instead of equation systems. It is a derived property here since we started out from *syntactic* properties of computation trees, i.e., algorithms (not just their meanings as in [11]). As in [11] we conclude:

Corollary 1. Assume $\mathcal{S}$ is weakly monotonic. Then:

(1) If σ is monotonic, then $G_{\mathcal{S}}\ \sigma$ is again monotonic;

(2) $\mathcal{S}$ has a unique least solution μ given by $\mu = \bigsqcup_{j\geq 0} G_{\mathcal{S}}^j\ \bot$.
Especially, this least solution μ is monotonic. □

6. Local Fixpoint Computation

Assume set V of variables is tremendously large while at the same time we are only interested in the values for a rather small subset X of variables. Then we should try to compute the values of a solution only for variables from X and all those variables y that "influence" values for variables in X. This is the idea of *local* fixpoint computation.

Evaluation of computation tree t on argument σ does not necessarily consult *all* values $\sigma\ x, x \in V$. Therefore, evaluation of t may succeed already for *partial* $\sigma : V \rightsquigarrow D$. If evaluation of t on σ succeeds, we can define the set $\mathsf{dep}(t, \sigma)$ accessed during this evaluation in the same way as in Section 3 for complete functions. Given partial variable assignment $\sigma : V \rightsquigarrow D$, variable y *directly* influences (relative to σ) variable x if $y \in \mathsf{dep}(t, \sigma)$ for some right-hand side t of x. Let then *"influencing"* denote the reflexive and transitive closure of this relation. Partial variable assignment σ is called *X–stable* iff for every $y \in V$ influencing some $x \in X$ relative to σ, and every constraint $y \sqsupseteq t$ in $\mathcal{S}$ for y, $[\![t]\!]\ \sigma$ is defined with $\sigma\ y \sqsupseteq [\![t]\!]\ \sigma$. A *solver* computes for constraint system $\mathcal{S}$ and set X of variables an X-stable partial assignment σ; furthermore, if $\mathcal{S}$ is weakly monotonic and μ is its least solution, then $\sigma\, y = \mu\, y$ for all y influencing some variable in X (relative to σ).

7. The Worklist Solver with Recursion

The first and simplest solver **WR** we consider is a local worklist algorithm enhanced with recursive descent into variables not yet considered (see Fig. 1). Solver **WR** is an adaption of a simplification of solver **WRT** in [11] to constraint systems. Opposed to **WRT**, no time stamps (the "**T**") are maintained.

For every encountered variable x, algorithm **WR** (globally) maintains the current value $\sigma\ x$ together with a set $\mathsf{infl}\ x$ of constraints $z \sqsupseteq t$ such that evaluation of t (on σ) may access value $\sigma\ x$ or, more formally, x may be contained in $\mathsf{dep}(t, \sigma)$. The set of constraints to be reevaluated is kept in data structure W, called a *worklist*. Initially, W is empty. The algorithm starts by initializing all variables from set X by calling procedure Init. Procedure Init when applied to variable x first checks whether x has already been encountered, i.e., is contained in set dom. If this is not the case, x is added to dom, $\sigma\ x$ is initialized to $\bot$ and set $\mathsf{infl}\ x$ of constraints potentially influenced by x is initialized to $\emptyset$. Then a first approximation for x is computed by evaluating all right-hand sides t for x and adding the results to $\sigma\ x$. If a value different from $\bot$ has been obtained, all elements from set $\mathsf{infl}\ x$ have to be added to W. After that, set $\mathsf{infl}\, x$ is emptied.

Initializing variables $x \in X$ through Init, may have inserted constraints into worklist W. Therefore, as long as W is nonempty, the algorithm now iteratively extracts constraints $x \sqsupseteq t$ from W and evaluates right-hand side t on current partial variable assignment σ. If $[\![t]\!]\ \sigma$ is not subsumed by $\sigma\, x$, the value of σ for x is updated. Since the value for x has changed, the

```
dom = ∅; W = ∅;
forall (x ∈ X) Init(x);
while (W ≠ ∅) {
    x ⊒ t = Extract(W);
    new = [[t]] (λy.Eval(x ⊒ t, y));
    if (new ⋢ σ x) {
        σ x = σ x ⊔ new;
        W = W ∪ infl x;
        infl x = ∅;
    }
}
void Init(V x) {
    if (x ∉ dom) {
        dom = dom ∪ {x};
        σ x = ⊥; infl x = ∅;
        forall (x ⊒ t ∈ S)
            σ x = σ x ⊔ [[t]] (λy.Eval(x ⊒ t, y));
        if (σ x ≠ ⊥) {
            W = W ∪ infl x;
            infl x = ∅;
        }
    }
}
D Eval(Constraint r, V y) {
    Init(y);
    infl y = infl y ∪ {r};
    return σ y;
}
```

Fig. 1: Algorithm **WR**.

constraints r in infl x may no longer be satisfied by σ; therefore, they are added to W. Afterwards, infl x is reset to $\emptyset$.

However, the right-hand sides t are *not* evaluated on σ directly either in Init or in the main loop on the worklist. There are two reasons for this. First, σ may not be defined for all variables y the evaluation of t may access; second, we have to determine all y accessed by the evaluation of t on σ. Therefore, t is applied to auxiliary function $\lambda y.\mathsf{Eval}(r, y)$. When applied to constraint r and variable y, Eval first initializes y by calling Init. Then r is added to infl y, and the value of σ for y (which now is always defined) is returned.

THEOREM 1. Algorithm **WR** is a solver. □

8. The Continuation Solver

Solver **WR** contains inefficiencies. Consider constraint $x \sqsupseteq t$ where, during evaluation of t, value $\sigma\, y$ has been accessed at subtree $t' = \langle y, C\rangle$ of t. Now assume $\sigma\, y$ obtains new value *new*. Then reevaluation of the *complete* right-hand side t is initiated. Instead of doing so, we would like to initiate

reevaluation only of subtree $C\,new$. Function C in subtree t' can be interpreted as (a syntactic representation of) the *continuation* where reevaluation of t has to proceed if the value of σ for y has been returned. We modify solver **WR** therefore as follows:

- Whenever during evaluation of the right-hand side of constraint $x \sqsupseteq t$, subtree $t' = \langle y, C\rangle$ is reached, we do not just access value $\sigma\ y$ and apply continuation C but additionally add (x, C) to the infl-set of variable y.
- Whenever $\sigma\, y$ has obtained a new value, we add to W all pairs $(x, \langle y, C\rangle)$, $(x, C) \in \mathsf{infl}\, y$, to initiate their later reevaluations.

Therefore, the infl-sets of resulting algorithm $\mathbf{WR}_C$ now contain elements from $V \times \mathbf{Cont}$ where $\mathbf{Cont} = D \to \mathcal{T}(V, D)$, whereas worklist W contains elements from $V \times \mathcal{T}(V, D)$. In order to have continuations explicitly available for insertion into infl-sets, we slightly change the functionality of the argument of $[\![.]\!]$ (and hence also $[\![.]\!]$) by passing down the current continuation into the argument. Thus, we now define:

$$\begin{array}{lcl} [\![d]\!]\ \sigma' & = & d \\ [\![x]\!]\ \sigma' & = & \sigma'\ (\lambda d.d)\ x \end{array} \qquad\qquad [\![\langle x, C\rangle]\!]\ \sigma' \;=\; [\![C\ (\sigma'\ C\ x)]\!]\ \sigma'$$

where $\sigma'\ C\ x = \sigma\ x$. Observe that the auxiliary function σ' just returns values of variables as given by variable assignment σ. Besides identifiers x, σ' consumes one further argument, namely, a continuation C. The extra argument C supplied to σ' is simply ignored – at least during the evaluation of right-hand sides according to $[\![.]\!]$. It becomes important, however, for the fixpoint algorithm since there σ' is replaced by the (partially applied) function Eval. As for solver **WR**, function Eval not only returns the values of variables (as σ') but additionally produces side effects at variable accesses. According to the changed functionality of σ', side effects now may also depend on the current formal continuation C.

Thus, the new function Eval consumes three arguments, namely variables x and y together with continuation C. Here, variable x represents the left-hand side of the current constraint, y represents the variable whose value is being accessed, and C is the current continuation. Given these three arguments, Eval first calls Init for y to make sure that $\sigma\, y$ as well as $\mathsf{infl}\, y$ have already been initialized. Then it adds (x, C) to set $\mathsf{infl}\, y$. Finally, it returns $\sigma\, y$. The resulting worklist solver $\mathbf{WR}_C$ with recursion and continuations is contained in Fig. 2. We obtain:

THEOREM 2. Algorithm $\mathbf{WR}_C$ is a solver. □

A similar optimization for the topdown solver **TD** [5, 11] in the context of analysis of Prolog programs has been called *clause prefix optimization* [8]. As far as we know, an *application independent* exposition of the method as well as its integration into worklist based solvers has not been considered before.

```
dom = ∅; W = ∅;
forall (x ∈ X) Init(x);
while (W ≠ ∅) {
    (x, t) = Extract(W);
    new = [[t]] (λC, y.Eval(x, C, y));
    if (new ⋢ σ x){
        σ x = σ x ⊔ new;
        forall ((x', C') ∈ infl x)
            W = W ∪ {(x', ⟨x, C'⟩)};
        infl x = ∅;
    }
}
void Init(V x){
    if (x ∉ dom) {
        dom = dom ∪ {x};
        σ x = ⊥; infl x = ∅;
        forall (x ⊒ t ∈ S))
            σ x = σ x ⊔ [[t]] (λC, y.Eval(x, C, y));
        if (σ x ≠ ⊥)
            forall ((x', C') ∈ infl x)
                W = W ∪ {(x', ⟨x, C'⟩)};
            infl x = ∅;
    }
}
D Eval(V x, Cont C, V y) {
    Init(y);
    infl y = infl y ∪ {(x, C)};
    return σ y;
}
```

Fig. 2: Algorithm $\mathbf{WR}_C$.

9. The Differential Fixpoint Algorithm

Assume variable y has changed its value. Instead of reevaluating all trees $\langle y, C\rangle$ from set infl of y with the new value, as solver $\mathbf{WR}_C$ eventually does, we may initiate reevaluation just for the *increment*. This optimization is helpful, if computation on "larger" values is much more expensive than computations on "smaller" ones. The increase of values is determined by some function $\mathsf{diff} : D \times D \to D$ satisfying

$$d_1 \sqcup \mathsf{diff}(d_1, d_2) \quad = \quad d_1 \sqcup d_2$$

EXAMPLE 4. If $D = 2^A$ for some set A (ordered by set inclusion), then there is a natural choice for diff, namely set difference.

If $D = M \to R$, M is a finite set and R a finite complete lattice, $\mathsf{diff}(f_1, f_2)$ can be defined as the mapping $f : M \to R$ given by $f\ v = \bot$ if $f_2\ v \sqsubseteq f_1\ v$ and $f\ v = f_2\ v$ otherwise. □

In order to make our idea work, we have to impose further restrictions onto the structure of right-hand sides t. We call $\mathcal{S}$ *distributive* if $\mathcal{S}$ is weakly

monotonic and for every subterm $\langle x, C\rangle$ of right-hand sides of $\mathcal{S}$, d_1, d_2, and d such that $d = d_1 \sqcup d_2$ and arbitrary variable assignment σ,

(1) $\mathsf{dep}(C\ d_1, \sigma) \cup \mathsf{dep}(C\ d_2, \sigma) \supseteq \mathsf{dep}(C\ d, \sigma)$;

(2) $[\![C\ d_1]\!]\ \sigma \sqcup [\![C\ d_2]\!]\ \sigma \sqsupseteq [\![C\ d]\!]\ \sigma$.

In interesting applications, $\mathcal{S}$ is even monotonic and variable dependencies are "static", i.e., independent of variable assignments. Furthermore, equality holds in item 2. This is especially the case if right-hand sides are given through expressions as in Example 1 when all operators f and g are distributive in each of their arguments.

In order to propagate increments, we change solver $\mathbf{WR}_C$ as follows. Assume $\sigma\ y$ has obtained a new value which differs from the old one by Δ and $(x, C) \in \mathsf{infl}\ y$.

- Instead of adding $(x, \langle y, C\rangle)$ to W (as for $\mathbf{WR}_C$), we add (x, C, Δ). Thus, now worklist W contains elements from $V \times \mathbf{Cont} \times D$.
- If we extract triple (x, C, Δ) from W, we evaluate ct $C\ \Delta$ to obtain a (possibly) new increment for x.

In contrast, however, to $\mathbf{WR}_C$ and $\mathbf{WR}$, it is no longer safe to empty sets $\mathsf{infl}\ y$ after use.

EXAMPLE 5. Consider a single constraint

$$x \sqsupseteq f\ x$$

where f is distributive. After initialization of $\sigma\ x$ through init, we have $\sigma\ x = f\ \bot = d_1$ and $\mathsf{W} = \{(x, C, d_1)\}$ where continuation C equals f. Assume, we have emptied the the influence set $\mathsf{infl}\ x$ of x in init. Then evaluation of the triple (x, C, d_1) results in an update of x – without triggering further insertions into worklist W. Consequently, fixpoint iteration terminates after two iterations – independently of whether a solution has been obtained or not.

If the influence set is not emptied in init but emptied during the main loop on the worklist, then fixpoint iteration may not stop after two iterations, but certainly after three – which cannot be generally correct. □

The resulting *differential* worklist algorithm with recursive descent into new variables and persistent influence sets ($\mathbf{WR}_\Delta$ for short) is given in Fig. 3. We would like to emphasize that, in contrast to the case of worklist solvers, it is not obvious (at least to us) how propagation of differences could possibly be added to the topdown solver.

Remark: An even more sophisticated implementation may collect all triples (x, C, Δ_i) in W with identical first two components and combine them into one single triple (x, C, Δ) where $\Delta = \bigsqcup \Delta_i$.

A proof of the following theorem can be found in Appendix A.

THEOREM 3. If $\mathcal{S}$ is distributive, then algorithm $\mathbf{WR}_\Delta$ computes an X-stable partial least solution. □

```
dom = ∅; W = ∅;
forall (x ∈ X) Init(x);
while (W ≠ ∅) {
    (x, C, Δ) = Extract(W);
    new = [[C Δ]] (λC, y.Eval(x, C, y));
    if (new ⋢ σ x){
        Δ = diff(σ x, new);
        σ x = σ x ⊔ new;
        forall ((x', C') ∈ infl x)
            W = W ∪ {(x', C', Δ)};
    }
}
void Init(V x){
    if (x ∉ dom) {
        dom = dom ∪ {x};
        σ x = ⊥; infl x = ∅;
        forall (x ⊒ t ∈ S))
            σ x = σ x ⊔ [[t]] (λC, y.Eval(x, C, y));
        if (σ x ≠ ⊥)
            forall ((x', C') ∈ infl x)
                W = W ∪ {(x', C', (σ x))};
    }
}
D Eval(V x, Cont C, V y) {
    Init(y);
    infl y = infl y ∪ {(x, C)};
    return σ y;
}
```

Fig. 3: Algorithm $\mathbf{WR}_{\Delta}$.

Note that we did not claim algorithm $\mathbf{WR}_{\Delta}$ to be a solver: and indeed this is not the case. Opposed to solvers **WR** and $\mathbf{WR}_C$, algorithm $\mathbf{WR}_{\Delta}$ may *fail* to compute (interesting parts of) the least solution for constraint systems which are not distributive.

10. The Distributive Framework *IDE*

As a first application, let us consider the distributive framework *IDE* for interprocedural analysis of imperative languages as suggested by Horwitz, Reps and Sagiv in [16]. *IDE* is a quite general distributive framework for interprocedural analysis of imperative languages. Horwitz, Reps and Sagiv apply it, e.g., to *linear constant propagation*. Framework *IDE* assigns to program points elements from lattice $D = M \to L$ of program states, where M is some finite base set (e.g., the set of currently visible program variables), and L is a lattice of abstract values.

The crucial point of program analysis in framework *IDE* consists in determining summary functions from $D \to D$ to describe effects of procedures. The lattice of possible transfer functions for statements as well as summary

functions for procedures in *IDE* is given by $\mathcal{F} = M^2 \to \mathcal{R}$ where $\mathcal{R} \subseteq L \to L$ is assumed to be a lattice of distributive functions of (small) finite height (e.g., 4 for linear constant propagation) which contains $\lambda x.\bot$ and the identity and is closed under composition. The transformer in $D \to D$ defined by $f \in \mathcal{F}$ is given as

$$[\![f]\!]\ \eta\ v' = \bigsqcup_{v \in M} f\ (v, v')\ (\eta\ v)$$

Clearly, $[\![f]\!]$ is distributive, i.e., $[\![f]\!]\ (\eta_1 \sqcup \eta_2) = [\![f]\!]\ \eta_1 \sqcup [\![f]\!]\ \eta_2$. Computing the summary functions for procedures in this framework boils down to solving a constraint system $\mathcal{S}$ over $\mathcal{F}$ where right-hand sides e are of the form:

$$e \quad ::= \quad f \mid x \mid f \circ x \mid x_2 \circ x_1$$

where $f \in \mathcal{F}$. Since all functions in $\mathcal{F}$ are distributive, function composition $\circ : \mathcal{F}^2 \to \mathcal{F}$ is distributive in each argument. Thus, constraint system $\mathcal{S}$ is a special case of the constraint systems from example 1. Therefore, we can apply $\mathbf{WR}_\Delta$ to compute the least solution of $\mathcal{S}$ efficiently – provided operations "$\circ$" and "$\sqcup$" can be computed efficiently. Using a diff-function similar to the last one of Example 4, we obtain:

THEOREM 4. If operations in $\mathcal{R}$ can be executed in time $\mathcal{O}(1)$, then the summary functions for program p according to interprocedural framework *IDE* can be computed by $\mathbf{WR}_\Delta$ in time $\mathcal{O}(|p| \cdot |M|^3)$. □

For a proof of Theorem 4 see appendix Appendix B. The complexity bound in Theorem 4 should be compared with bound $\mathcal{O}(|p| \cdot |M|^5)$ which trivially can be obtained using solver $\mathbf{WR}$. By saving factor $\mathcal{O}(|M|^2)$, we find the same complexity bounds for $\mathbf{WR}_\Delta$ as have been obtained in [16] for a special purpose algorithm.

11. Control-Flow Analysis

Control-flow analysis (cfa for short) is an analysis for higher-order functional languages possibly with recursive data types [12, 13]. Cfa on program p tries to compute for every expression t occurring in p a superset of expressions into which t may evolve during program execution, see, e.g., [22, 23, 20, 25].

Let A denote the set of subexpressions of p and $D = 2^A$. Then cfa for a lazy language as in [25] can be formalized through a constraint system $\mathcal{S}$ over domain D with set V of in variables $y_t, t \in A$, where right-hand sides of constraints consist of expressions e of one of the following forms:

$$e \quad ::= \quad \{a\} \mid x \mid (a \in x_1); x_2$$

where $a \in A$. Here, we view $(a \in x_1); x_2$ as specification of ct $\langle x_1, C\rangle$ where $C\ d = \emptyset$ if $a \notin d$ and $C\ d = x_2$ otherwise. For a precise description how such

constraints arise, see Appendix C. Let us assume set V of variables is just ordered by equality. Then $\mathcal{S}$ is not only monotonic but also distributive. As function diff, we simply choose: $\mathsf{diff}(d_1, d_2) = d_2 \backslash d_1$. With these definitions, algorithm $\mathbf{WR}_\Delta$ can be applied.

Let us assume that the maximal cardinality of an occurring set is bounded by $s \leq |p|$. Furthermore, let I denote the complexity of inserting a single element into a set of maximally s elements. In case, for example, we can represent sets as bit vectors, $I = \mathcal{O}(1)$. In case, the program is large but we nevertheless expect sets to be sparse we may use some suitable hashing scheme to achieve approximately the same effect. Otherwise, we may represent sets through balanced ordered trees giving extra cost $I = \mathcal{O}(\log\, s)$.

Now let us estimate the amount of time the algorithm spends with an individual constraint of the form $y \supseteq (a \in x_1); x_2$. Let $d_1, \ldots, d_k$ be the increments produced by the algorithm for $\sigma\ x_1$. By definition, $|d_1| + \ldots + |d_k| \leq s$. Since these increments are element-disjoint at most one of them triggers the evaluation of x_2. Thus, the overall amount of work spent with this constraint is bounded by $\mathcal{O}(2 \cdot s \cdot I) = \mathcal{O}(s \cdot I)$. Since at most $|p|^2$ such constraints arise, this results in a total amount of $\mathcal{O}(|p|^2 \cdot s \cdot I)$. Since there are at most $|p|$ other constraints and the time spent with each of these is bounded by $\mathcal{O}(s \cdot I)$, we achieve time complexity $\mathcal{O}(|p|^2 \cdot s \cdot I)$.

Indeed, this first algorithm still contains inefficiencies. It creates (a representation of) $\mathcal{O}(|p|^2)$ constraints of the form $y \supseteq (a \in x_1); x_2$ in advance – although potentially only few of these will result in a non-trivial contribution. For this type of constraints, however, we can introduce the following optimization. We start with constraint system $\mathcal{S}_0$ lacking all constraints of the form $y \supseteq (a \in x_1); x_2$. Instead, we introduce function $r : V \to D \to 2^{\mathbf{constraints}}$ which, depending on the value of variables, returns the set of constraints to be added to the present constraint system. r is given by:

$$r\ x\ d = \{y \supseteq x_2 \mid a \in d, y \supseteq (a \in x); x_2 \in \mathcal{S}\}$$

Thus especially, $r\ x\ (d_1 \cup d_2) = (r\ x\ d_1) \cup (r\ x\ d_2)$.

Now, we start $\mathbf{WR}_\Delta$ on $\mathcal{S}_0$ of size $\mathcal{O}(|p|)$. Whenever variable x is incremented by Δ, we add all constraints from $r\ x\ \Delta$ to the current constraint system by calling Init for their left-hand sides and then inserting them into worklist W. For cfa, each application $r\ x\ \Delta$ can be evaluated in time $\mathcal{O}(|\Delta|)$. Thus, if the cardinalities of all occurring sets is bounded by s, at most $\mathcal{O}(|p| \cdot s)$ constraints of the form $y \supseteq x$ are added to $\mathcal{S}_0$. Each of these introduces an amount $\mathcal{O}(s \cdot I)$ of work. Therefore, this optimization brings down overall complexity to $\mathcal{O}(|p| \cdot s^2 \cdot I)$. Summarizing, we obtain:

THEOREM 5. If s is the maximal cardinality of occurring sets, the least solution of constraint system $\mathcal{S}$ for cfa on program p can be computed by the optimized $\mathbf{WR}_\Delta$ algorithm in time $\mathcal{O}(|p| \cdot s^2 \cdot I)$. □

The idea of dynamic extension of constraint systems is especially appealing and clearly can also be cast in a more general setting.

The algorithm from Theorem 5 can be viewed as a variant of the algorithm proposed by Heintze in [12]. Using both dynamic variable dependencies as well as dynamic constraint systems, it has provably (almost) linear runtime – provided the cardinalities of occurring sets are sufficiently small and sets are implemented efficiently.

12. Generalized Computation Trees

In practical applications, certain subcomputations for right-hand sides turn out to be independent. For example, the values for a set of variables may be accessed in any order if it is just the least upper bound of returned values which matters. This type of independence allows for much more aggressive forms of the differential approach. In order to describe such kinds of phenomena formally, we introduce set $\mathcal{GT}(V, D)$ of *generalized computation trees* (gct's for short). Gct's t are given by:

$$t \quad ::= \quad d \mid x \mid S \mid \langle t, C \rangle$$

where $S \subseteq \mathcal{GT}(V, D)$ is finite and $C : D \to \mathcal{GT}(V, D)$. Thus, we not only allow *sets* of computation trees but also (sets of) computation trees as *selectors* of computation trees. Given $t \in \mathcal{GT}(V, D)$, function $[\![t]\!] : (V \to D) \to D$ implemented by t as well as set $\mathsf{dep}(t, \_)$ of variables accessed during evaluation are defined by:

$$\begin{array}{lcllcl}
[\![d]\!]\,\sigma & = & d & \mathsf{dep}(d, \sigma) & = & \emptyset \\
[\![x]\!]\,\sigma & = & \sigma\, x & \mathsf{dep}(x, \sigma) & = & \{x\} \\
[\![S]\!]\,\sigma & = & \bigsqcup\{[\![t]\!]\,\sigma \mid t \in S\} & \mathsf{dep}(S, \sigma) & = & \bigcup\{\mathsf{dep}(t, \sigma) \mid t \in S\} \\
[\![\langle t, C\rangle]\!]\,\sigma & = & [\![C\,([\![t]\!]\,\sigma)]\!]\,\sigma & \mathsf{dep}(\langle t, C\rangle, \sigma) & = & \mathsf{dep}(t, \sigma) \cup \mathsf{dep}(C\,([\![t]\!]\,\sigma), \sigma)
\end{array}$$

While *sets* of trees conveniently allow to make independence of subcomputations explicit (see our example application in section 13), nesting of trees into selectors eases the translation of deeper nesting of operator applications.

Example 6. Assume expressions e are given by the grammar:

$$e \quad ::= \quad d \mid x \mid f\, e \mid g\,(e_1, e_2)$$

where $d \in D$ and f and g denote monotonic functions in $D \to D$ and $D^2 \to D$, respectively. The gct t_e for e can then be constructed by:

- $t_e = e$ if $e \in D \cup V$;
- $t_e = \langle t_{e'}, \lambda d.f\, d\rangle$ if $e \equiv f\, e'$;
- $t_e = \langle t_{e_1}, C\rangle$ with $C\, d_1 = \langle t_{e_2}, \lambda d_2.g\,(d_1, d_2)\rangle$ if $e \equiv g\,(e_1, e_2)$. □

At the expense of introducing additional variables, constraint system $\mathcal{S}$ using gct's can always be transformed into an equivalent constraint system $\mathcal{S}'$ using just ordinary ct's as right-hand sides. To this end, we only need to add all gct's s to the set of variables which represent sets or selectors not contained

in $D \cup V$. For these new variables s, we then add constraints $s \sqsupseteq t$ whenever $t \in s$ and $s \sqsupseteq \langle t, C\rangle$ whenever $s \equiv \langle t, C\rangle$. Resulting right-hand sides are gct's without sets and with selectors which are either constants or variables, and therefore can equivalently be represented by ct's.

EXAMPLE 7. Assume $x \sqsupseteq \langle\{x_1, x_2\}, C\rangle$ is a constraint of $\mathcal{S}$. Viewing set $s = \{x_1, x_2\}$ as an auxiliary variable, we add the two constraints: $s \sqsupseteq x_1, s \sqsupseteq x_2$. □

At least theoretically, this normalization is always viable. Normalization, however, has two disadvantages. First, solving the normalized constraint system is *computationally* different from solving the original one. Second, a (possibly) huge amount of auxiliary variables is introduced. This second draw-back is especially problematic since the overhead of dynamicly creating variables is by no means negligible in practice. In our experience, efficiency of program analyzers, e.g. for Prolog, crucially depends on *minimizing* the number of variables to be maintained by the fixpoint algorithm.

We conclude that it is of *practical* importance to allow generalized ct's as right-hand sides. In the following we extend the notions of "weak monotonicity" and "distributivity" to constraint systems where right-hand sides are gct's.

For partial ordering "$\leq$" on set V of variables, we define relation "$\leq$" on gct's inductively by:

- $\bot \leq t$ for every t; and $d_1 \leq d_2$ if $d_1 \sqsubseteq d_2$;
- $x_1 \leq x_2$ as gct's if also $x_1 \leq x_2$ as variables;
- $S_1 \leq S_2$ if for all $t_1 \in S_1$, $t_1 \leq t_2$ for some $t_2 \in S_2$;
- $\langle t_1, C_1\rangle \leq \langle t_2, C_2\rangle$ if $t_1 \leq t_2$ and for all $d_1 \sqsubseteq d_2$, $C\, d_1 \leq C\, d_2$.

Now assume that all right-hand sides of constraint systems $\mathcal{S}$ are given through gct's. Then $\mathcal{S}$ is called *weakly monotonic* iff for every $x_1 \leq x_2$ and constraint $x_1 \sqsupseteq t_1$ in $\mathcal{S}$ some constraint $x_2 \sqsupseteq t_2$ in $\mathcal{S}$ exists such that $t_1 \leq t_2$. With these extended definitions Prop. 1, Cor. 1 as well as Theorem 1 hold. Therefore, algorithm **WR** is also a solver for constraint systems where right-hand sides are represented by gct's.

Function C in $t = \langle t', C\rangle$ can now only be interpreted as a representation of the continuation where reevaluation of t has to start if the evaluation of subtree t' on σ has terminated. t' again may be of the form $\langle s, C'\rangle$. Consequently, we have to deal with *lists* γ of continuations. Thus, whenever during evaluation of t an access to variable y occurs, we now have to add pairs (x, γ) to the infl-set of variable y. As in Section 8, we therefore change the functionality of $[\![.]\!]$ as well as the argument of $[\![.]\!]$ and hence also $[\![.]\!]$ by defining:

$$\begin{array}{lcllcl} [\![d]\!]\,\sigma'\,\gamma & = & d & [\![S]\!]\,\sigma'\,\gamma & = & \bigsqcup\{[\![t]\!]\,\sigma'\gamma \mid t \in S\} \\ [\![x]\!]\,\sigma'\,\gamma & = & \sigma'\,\gamma\,x & [\![\langle t, C\rangle]\!]\,\sigma'\,\gamma & = & [\![C\,([\![t]\!]\,\sigma'\,(C{:}\gamma))]\!]\,\sigma'\,\gamma \end{array}$$

where $\sigma'\,\gamma\,x = \sigma\,x$. The goal here is to avoid reevaluation of whole set S just because one of its elements has changed its value. Therefore, we propagate list γ arriving at *set* S of trees immediately down to each of its elements.

Now assume $\sigma\,y$ has changed its value by Δ. Then we add all triples (x, γ, Δ) to W where $(x, \gamma) \in \mathsf{infl}\,y$. Having extracted such a triple from the worklist, the new solver applies list γ to the new value Δ. The iterative application process is implemented by:

$$\begin{array}{lcl} \mathsf{app}\ []\ d\ \sigma' & = & d \\ \mathsf{app}\ (C{:}\gamma)\ d\ \sigma' & = & \mathsf{app}\ \gamma\ (\llbracket C\ d \rrbracket\ \sigma'\ \gamma)\ \sigma' \end{array}$$

The resulting value then gives the new contribution to the value of σ for x.

As in Section 9, we call constraint system $\mathcal{S}$ with gct's as right-hand sides *distributive* iff $\mathcal{S}$ is weakly monotonic and for every subtree $\langle t, C\rangle$ of right-hand sides, d_1, d_2, and d such that $d = d_1 \sqcup d_2$ and variable assignment σ,

(1) $\mathsf{dep}(C\,d_1, \sigma) \cup \mathsf{dep}(C\,d_2, \sigma) \supseteq \mathsf{dep}(C\,d, \sigma)$;

(2) $\llbracket C\,d_1 \rrbracket\,\sigma \sqcup \llbracket C\,d_2 \rrbracket\,\sigma \sqsupseteq \llbracket C\,d \rrbracket\,\sigma$.

This definition naturally extends the corresponding definition for constraint systems with ordinary evaluation trees as right-hand sides. We obtain:

THEOREM 6. Assume $\mathcal{S}$ is a distributive constraint system where right-hand sides are given by gct's. Then algorithm $\mathbf{WR}_\Delta$ computes an X-stable partial least solution. □

As an advanced application of gct's, we consider abstract `OLDT`-resolution [14, 15, 24].

13. Abstract `OLDT`-Resolution

Given Prolog program p, abstract `OLDT`-resolution tries to compute for every program point x the set of (abstract) values arriving at x. Let A denote the set of possible values. Lattice D to compute in is then given by $D = 2^A$. *Coarse-grained* analysis assigns to each predicate an abstract state transformer $A \to D$, whereas *fine-grained* analysis additionally assigns transformers also to every program point [14]. Instead of considering transformers as a whole (as, e.g., in the algorithm for framework *IDE* in Section 10), transformer valued variable x is replaced by a *set* of variables, namely $x\,a, a \in A$, where $x\,a$ represents the return value of the transformer for x on input a. Thus, each variable $x\,a$ potentially receives a value from D. The idea is that, in practice, set A may be tremendously large, while at the same time each transformer is called only on a small number of inputs. Thus, in this application we explicitly rely on the *locality* of our fixpoint algorithms, i.e., their demand-driven exploration of the variable space.

To every transformer variable x the analysis assigns a finite set of *constraint schemes* $x\bullet \supseteq e$ where $\bullet$ formally represents the argument to the

transformer, and e is an expression built up according to the following grammar (see [24]):

$$\begin{aligned} e &::= \mathbf{s}\bullet \mid \mathcal{E}\, f\, e \\ f &::= \lambda a.\mathbf{s}\,(g\,a) \mid x \mid \lambda a.(x\,(g\,a) \,\square\, a) \end{aligned}$$

Here, $\mathbf{s} : A \to D$ denotes the singleton map defined by $\mathbf{s}\,a = \{a\}$, and $\mathcal{E} : (A \to D) \to D \to D$ denotes the usual extension function defined by $\mathcal{E}\ f\ d = \bigcup\{f\ a \mid a \in d\}$. Thus, expressions e are built up from $\mathbf{s}\bullet = \{\bullet\}$ by successive applications of extensions $\mathcal{E}\ f$ for three possible types of functions $f : A \to D$.

Unary functions $g : A \to A$ are used to model point-to-point transfer functions as well as passing of actual parameters into procedures (predicates) and returning of results, whereas binary operators $\square : D \times A \to D$ conveniently allow to model *local states* of procedures. They are used to combine the set of return values of procedures (predicates) with the local state before the call [24]. For Prolog, they are essentially given by *abstract instantiation* [7] extended to sets. In case of fine-grained analysis, every scheme for right-hand sides contains at most two occurrences of "$\mathcal{E}$", whereas coarse-grained analysis possibly introduces also deeper nesting.

The set of constraints of abstract `OLDT`-resolution for variable $x\,a, a \in A$, are obtained from the set of constraint schemes for x by instantiating formal parameter $\bullet$ with actual parameter a. The resulting right-hand sides can be implemented through gct's $t_{\{a\}}$. For $d \in D$, gct t_d is of the form $t_d = d$ or $t_d = \langle S_d, C\rangle$ such that $C\,d'$ returns some tree $t'_{d'}$, and $S_d = \{s_a \mid a \in d\}$. The possible forms for elements s_a of S_d correspond to the three possible forms for f in expressions $\mathcal{E}\ f\ e$, i.e,

$$s_a ::= \mathbf{s}\,(g\,a) \mid x\,a \mid \langle x\,(g\,a), \lambda d'.d' \,\square\, a\rangle$$

In the first case all elements s_a of selector S_d consist just of constants, in the second case of variable look-ups, and in the third case of operators applied to the results of variable look-ups.

Constraint system $\mathcal{S}$ for abstract `OLDT`-resolution then turns out to be monotonic as well as distributive. As operator diff, we choose: $\mathsf{diff}(d_1, d_2) = d_2 \backslash d_1$. Therefore, we can apply algorithm $\mathbf{WR}_\Delta$. The next theorem gives the complexity bound just for the case of fine-grained analysis, i.e., where right-hand sides contain at most two occurrences of extension function $\mathcal{E}$. For simplicity, we assume that applications $g\,a$ as well as insertion into sets can be executed in time $\mathcal{O}(1)$, and evaluation of $d \,\square\, a$ needs time $\mathcal{O}(\#d)$. Then:

THEOREM 7. Fine-grained `OLDT`-resolution for program p with abstract values from A can be executed by $\mathbf{WR}_\Delta$ in time $\mathcal{O}(N \cdot s^2)$ where $N \le |p| \cdot \#A$ is the maximal number of considered variables, and $s \le \#A$ is an upper bound for the cardinalities of occurring sets. □

$\mathbf{WR}_\Delta$ saves an extra factor $\mathcal{O}(s^2)$ over solver **WR**. An algorithm with similar savings has been suggested by Horwitz, Reps and Sagiv in [17]. Their graph-based algorithm, however, is neither able to treat *binary* operators nor deeper nested right-hand sides, i.e., coarse-grained analysis as ours. Therefore, their apporach is unable to implement, e.g., `OLDT`-resolution for Prolog.

In the usual application for program analysis, A is equipped with some partial abstraction ordering "$\sqsubseteq$", implying that set $d \subseteq A$ contains the same information as its lower closure $d{\downarrow} = \{a \in A \mid \exists a' \in d : a \sqsubseteq a'\}$. In this case, we may decide to compute with *lower* subsets of A right from the beginning [14]. Here, subset $d \subseteq A$ is called lower iff $d = d{\downarrow}$. If all occurring functions f as well as operators $\Box$ are monotonic, we can represent lower sets d by their maximal elements and do all computations just with such anti-chains. The resulting constraint system then turns out to be no longer monotonic. However, it is still *weakly* monotonic w.r.t. variable ordering "$\leq$" given by $x_1\, a_1 \leq x_2\, a_2$ iff $x_1 \equiv x_2$ and $a_1 \sqsubseteq a_2$. As function diff for anti-chains, we choose

$$\begin{aligned} \mathsf{diff}(d_1, d_2) &= d_2 \backslash (d_1{\downarrow}) \\ &= \{a_2 \in d_2 \mid \forall a_1 \in d_1 : a_2 \not\sqsubseteq a_1\} \end{aligned}$$

Again, we can apply the differential worklist algorithm. Here, we found no comparable algorithm in the literature. $\mathbf{WR}_\Delta$ beats conventional solvers for this application (see Section 14). A simple estimation of the runtime complexity, however, is no longer possible since even large sets may have succinct representations by short anti-chains. Also, adding a new element to a set may now take time proportional to the length of the corresponding anti-chain.

14. Practical Experiments

We have adapted the fastest general-purpose equation solver from [11], namely **WDFS** (called $\mathbf{WDFS}^{Equ}$ here), to constraint systems giving general-purpose constraint solver $\mathbf{WDFS}^{Con}$. Solver $\mathbf{WDFS}^{Con}$ is similar to solver **WR** as presented in Section 7, but additionally maintains priorities on variables and, before return from an update of σ for variable x, evaluates all constraints $y \sqsupseteq t$ from the worklist where y has higher priority as the variable below x (see [11] for a detailed description of this idea for equation solvers). The effect of this strategy is that solver $\mathbf{WDFS}^{Con}$ proceeds strong component by strong component in the (current) variable dependence graph.

To solver $\mathbf{WDFS}^{Con}$ we added continuations as well as propagation of differences (the "Δ") in the same way as we added continuations and propagation of differences to solver **WR** in Section 12.

In order to remove multiples from data structures $\mathsf{infl}\, y$ as well as W, we generally have to rely on a computable equality predicate on continuations. Since our continuations are syntactical objects, this is possible but may be expensive. At the risk, therefore, of potentially evaluating the same

continuation several times, we avoided this overhead in our implementation by simply implementing occurring sets $\mathsf{infl}\, y$ as well as W through *lists*.

All fixpoint algorithms have been integrated into GENA [9, 10]. GENA is a generator for Prolog program analyzers written in SML. We generated analyzers for abstract OLDT-resolution for PS+POS+LIN which is Søndergaard's pair sharing domain enhanced with POS for inferring groundness [26, 7]. Its abstract substitutions are pairs of bdd's and graphs over variables. Thus, our fixpoint algorithms maintain anti-chains of such elements. The generated analyzers were run on large real-world programs that are significantly bigger and more complex than the programs from the GAIA benchmark suite. `aqua-c` (about 560KB) is the source code of an early version of Peter van Roy's Aquarius Prolog compiler. `chat` (about 170KB) is David H.D. Warren's chat-80 system. The numbers reported in Table I, are the absolute runtimes in seconds (including system time) obtained for SML-NJ, version 110.7, on a PentiumPro with 200 MH and 128 MB main memory.

Comparing the three algorithms for OLDT-resolution, we find that all of these have still quite acceptable runtimes (perhaps with exception of `aqua-c`) where algorithm $\mathbf{WDFS}_{\Delta}^{Con}$ almost always outperforms the others. Compared with equation solver $\mathbf{WDFS}^{Equ}$ on benchmarks with significant analysis times (more than 1 second), algorithm $\mathbf{WDFS}_{\Delta}^{Con}$ saves approximately 40% of the runtime where usually less than half of the gain is obtained by maintaining constraints. The maximal relative gain of 58% could be observed for program `readq`. On the other hand for Stefan Diehl's interpreter for action semantics `action`, propagation of differences gave only a minor improvement over considering constraints alone. Program `ann` even showed a (moderate) *decrease* in efficiency (78%) which seems to originate from the use of continuations. The same observation can be made for `aqua-c`. Here, constraints resulted in an improvement of 23% of which 9% was lost again through propagation of differences. This slowdown for `ann` and `aqua-c` did not occur during analyzer runs for other abstract domains we experimented with. Table II reports the runtimes found for these two benchmarks on domain CompCon. Abstract domain CompCon analyzes whether variables are bound to atoms or are composite. Abstract operations for CompCon are less expensive than for PS+POS+LIN but anti-chains tend to be longer. For CompCon, benchmark `ann` was fast enough that no significant differences between the solvers could be measured. For `aqua-c`, constraints introduced a slowdown of 20% whereas propagation of differences resulted in a gain in efficiency of 35% over $\mathbf{WDFS}^{Equ}$.

15. Conclusion

We succeeded to give an application independent exposition of two further improvements to worklist-based local fixpoint algorithms. This allowed us not only to exhibit common algorithmic ideas in seemingly different fast special purpose algorithms like the one of Horwitz et al. for interprocedural

program	$\mathbf{WDFS}^{Equ}$	$\mathbf{WDFS}^{Con}$	$\mathbf{WDFS}^{Con}_{\Delta}$
action.pl	14.41	8.14	6.99
ann.pl	0.87	0.79	1.55
aqua-c.pl	737.86	565.01	632.97
b2.pl	0.83	0.87	0.71
chat.pl	38.90	32.42	24.64
chat-parser.pl	13.87	12.89	8.13
chess.pl	0.20	0.23	0.23
flatten.pl	0.19	0.17	0.13
nand.pl	0.29	0.26	0.23
readq.pl	8.27	6.83	3.44
sdda.pl	0.38	0.32	0.24

TABLE I: Comparison of $\mathbf{WDFS}^{Equ}$, $\mathbf{WDFS}^{Con}$ and $\mathbf{WDFS}^{Con}_{\Delta}$ with abstract domain PS+POS+LIN.

program	$\mathbf{WDFS}^{Equ}$	$\mathbf{WDFS}^{Con}$	$\mathbf{WDFS}^{Con}_{\Delta}$
ann.pl	0.10	0.09	0.08
aqua-c.pl	64.67	77.66	42.10

TABLE II: Comparison of $\mathbf{WDFS}^{Equ}$, $\mathbf{WDFS}^{Con}$ and $\mathbf{WDFS}^{Con}_{\Delta}$ with abstract domain CompCon.

framework *IDE* [16] of Heintze's algorithm for control-flow analysis [12]. Thus, our exposition explains also how their algorithms can be *improved* – namely by incorporating ideas like the recursive descent into variables as well as the timestamps idea from [11]. Finally, our approach allowed to develop completely new efficient algorithms for abstract OLDT-resolution.

Several further questions remain for investigation. First, more practical experiments are needed to check the practical efficiency of our methods for other application domains. Here, the area of model-checking might be especially promising. Second, we would like to see whether the idea of dynamic constraint systems as introduced in Section 11 can be successfully applied in a less ad hoc way.

Appendix A. Proof of Theorem 3.

.

Let μ denote the least solution of $\mathcal{S}$.

Since every variable can be initialized at most once, we conclude that procedure Init can be called at most $\#V$ times. Let W_i, dom_i, $\mathsf{infl}_i\, y$, $\sigma_i\, y$ and Δ_i, respectively, denote the values of W, dom, $\mathsf{infl}\, y$, $\sigma\, y$, Δ and respectively, immediately after the i-th updates of σ and W. Then we have:

(1) The values for dom as well as $\sigma\ x$ and $\mathsf{infl}\ x$ (once defined) are non-decreasing.

(2) Whenever $\sigma\, y$ receives a new value through update i, then $\sigma_i\, y = \sigma_{i-1}\, y \sqcup \Delta_i$ and $(x, C, \Delta_i) \in \mathsf{W}_i$ for all $(x, C) \in \mathsf{infl}_i\, y$.

The number of updates of a single initialized variable is bounded by the height of lattice D. Therefore, the total number of updates is bounded. From this, we conclude that algorithm $\mathbf{WR}_\Delta$ indeed terminates.

Since in case of an update of σ for variable y, $\sigma_i\, y \sqsubseteq \mu\, y$ as well as $\Delta_i \sqsubseteq \mu\, y$, we deduce from weak monotonicity that

(3) $\sigma_i\, y \sqsubseteq \mu\, y$ for all i whenever $y \in \mathsf{dom}_i$.

Let σ^* resp. dom^* denote the final values for σ resp. dom. By property (3), $\sigma^*\ x \sqsubseteq \mu\, x$ for all $x \in \mathsf{dom}^*$. It remains to prove that σ^* is X-stable. Since $X \subseteq \mathsf{dom}^*$, we instead prove that σ^* is dom^*-stable. To this end, we prove:
Claim 1: If $x \sqsupseteq t$ is a constraint of $\mathcal{S}$ where $x \in \mathsf{dom}^*$, then evaluation of t on σ^* is defined with $\sigma^*\ x \sqsupseteq [\![t]\!]\, \sigma^*$.

In order to prove Claim 1, consider non-constant subtree occurrence[1] t' of t which has contributed to the value for x at least once. W.l.o.g. assume $t' = \langle y, C\rangle$. Then let $\rho\, t' = \{d_0, \ldots, d_k\}$ denote the set of values d_0 together with increments d_i such that evaluation of $C\, d_i$ has been initiated. Clearly, $d_0 \sqcup \ldots \sqcup d_k = \sigma^*\ y$.

Then the contribution of (subtrees of) t to $\sigma^*\ x$ is given by:

$$\begin{array}{lcl} [\![d]\!]\ \rho & = & d \\ [\![y]\!]\ \rho & = & d_0 \sqcup \ldots \sqcup d_k \\ & & \text{if} \quad \rho\ y = \{d_0, \ldots, d_k\} \\ [\![\langle y, C\rangle]\!]\ \rho & = & [\![C\ d_0]\!]\ \rho \sqcup \ldots \sqcup [\![C\ d_k]\!]\ \rho \\ & & \text{if} \quad \rho\ \langle y, C\rangle = \{d_0, \ldots, d_k\} \end{array}$$

According to this definition, $[\![t]\!]\, \rho$ is the least upper bound of all results for accessed leaves of t. Clearly, each such leave returns a value which is subsumed by σ^*.

Accordingly, define the set of variables accessed during all evaluations of subtree occurrences as:

$$\begin{array}{lcl} \mathsf{dep}(d, \rho) & = & \emptyset \\ \mathsf{dep}(y, \rho) & = & \{y\} \\ \mathsf{dep}(\langle y, C\rangle, \rho) & = & \{y\} \cup \mathsf{dep}(C\ d_0, \rho) \cup \ldots \cup \mathsf{dep}(C\ d_k, \rho) \\ & & \text{if} \quad \rho\ \langle y, C\rangle = \{d_0, \ldots, d_k\} \end{array}$$

These are precisely those variables which have been accessed by the fixpoint algorithm during computations on t. Therefore, $\mathsf{dom}^* \supseteq \mathsf{dep}(t, \rho)$. We conclude that Claim 1 is implied by the following claim.
Claim 2: If S is distributive, then for all subtrees t' of t with $[\![t']\!]\, \rho$ defined,

(1) $\mathsf{dep}(t', \rho) \supseteq \mathsf{dep}(t', \sigma^*)$;

[1] For notational convenience, we treat subtree *occurrences* as if they were subtrees themselves.

(2) $[\![t']\!]\,\rho \sqsupseteq [\![t']\!]\,\sigma^*$.

The proof of Claim 2 is by induction on the structure of t'. The only interesting case is $t' \equiv \langle y, C\rangle$ where $\rho\, t' = \{d_0, \ldots, d_k\}$. Then

$$\begin{array}{lcll} \mathsf{dep}(t', \rho) & = & \{y\} \cup \mathsf{dep}(C\, d_0, \rho) \cup \ldots \cup \mathsf{dep}(C\, d_k, \rho) & \\ & \supseteq & \{y\} \cup \mathsf{dep}(C\, d_0, \sigma^*) \cup \ldots \cup \mathsf{dep}(C\, d_k, \sigma^*) & \text{by ind. hypothesis} \\ & \supseteq & \{y\} \cup \mathsf{dep}(C\ (\sigma^*\, y), \sigma^*) & \text{by distributivity} \\ & = & \mathsf{dep}(t', \sigma^*) & \end{array}$$

Therefore, $[\![t']\!]\,\sigma^*$ is defined, and we can conclude:

$$\begin{array}{lcll} [\![t']\!]\,\rho & = & [\![C\, d_0]\!]\,\rho \sqcup \ldots \sqcup [\![C\, d_k]\!]\,\rho & \\ & \sqsupseteq & [\![C\, d_0]\!]\,\sigma^* \sqcup \ldots \sqcup [\![C\, d_k]\!]\,\sigma^* & \text{by ind. hypothesis} \\ & \sqsupseteq & [\![C\ (\sigma^*\, y)]\!]\,\sigma^* & \text{by distributivity} \\ & = & [\![t']\!]\,\sigma^* & \end{array}$$

This completes the proof of Claim 2 and thus also the proof of Claim 1 and Theorem 3. □

Appendix B. Proof of Theorem 4.

Values $\sigma\, x$ maintained by $\mathbf{WR}_\Delta$ can be implemented as arrays (indexed with $M \times M$), while increments to be propagated can be implemented as lists of its non-$(\lambda x.\bot)$ elements. "$\sqcup$" on $M^2 \to \mathcal{R}$ can be computed componentwise whereas composition "$\circ$" is given by

$$(f_2 \circ f_1)\,(v_1, v_2) = \bigsqcup_{v \in M} f_2\,(v, v_2) \circ f_1\,(v_1, v)$$

We use a diff-function similar to the last one of Example 4, i.e., we define for $\pi \in M^2$,

$$\mathsf{diff}(f_1, f_2)\,\pi = \begin{cases} \lambda x.\bot & \text{if } f_2\,\pi \sqsubseteq f_1\,\pi \\ f_2\,\pi & \text{otherwise} \end{cases}$$

Assume $f = f_2 \circ f_1$, and $\Delta : M^2 \to \mathcal{R}$ contains at most l non-$(\lambda x.\bot)$-elements which are given as a list. Then the following holds:

- The list of non-$(\lambda x.\bot)$-entries of $\Delta \circ f_1$ can be computed in time $\mathcal{O}(l \cdot |M|)$.
- If f has already been computed, then $f' = f_2 \circ (f_1 \sqcup \Delta) = f \sqcup (f_2 \circ \Delta)$ together with the non-$(\lambda x.\bot)$-entries of the increment $\Delta' = \mathsf{diff}(f, f')$ can be computed in time $\mathcal{O}(l \cdot |M|)$.

We conclude that the maximal time spent on any right-hand side is bounded by $\mathcal{O}(|M|^3)$. It follows that $\mathbf{WR}_\Delta$ computes the least solution in time $\mathcal{O}(|\mathcal{S}| \cdot |M|^3)$. Since constraint system $\mathcal{S}$ for *IDE* has size linear in the program, Theorem 4 follows. □

Appendix C. CFA for an Example Language

Let us consider the higher-order functional language given by the grammar:

$$\begin{array}{lcl} t & ::= & v \mid \lambda v.t \mid c\ t_1 \ldots t_n \mid f \mid t\ t' \\ & & \mid \mathbf{case}\ t_0\ \mathbf{of}\ q_1 \to t_1; \ldots; q_k \to t_k \\ q & ::= & c\ v_1 \ldots v_n \\ p & ::= & \mathbf{letrec}\ f_1 = t_1; \ldots; f_m = t_m\ \mathbf{in}\ t \end{array}$$

(where $n \geq 0, k > 0, m > 0$). Here, c, v, and f range over names for constructors, variables, and functions, respectively, whereas t, q, and p range over terms, patterns, and programs. Moreover, we assume that constructor applications are always saturated. The semantics of the language is call-by-need.

The set $\mathcal{S}_p$ of constraints generated for cfa on program p is then given by:

- $y_t \supseteq \{t\}$ for all $t \in A$;
- $y_t \supseteq (a \in y_{t_1}); y_{t'}$ if $t \equiv t_1\ t_2$ and $a \equiv \lambda v.t'$;
- $y_v \supseteq (a \in y_{t_1}); y_{t_2}$ whenever $t_1\ t_2$ is a subexpression of p and $a \equiv \lambda v.t$;
- $y_f \supseteq y_t$ for every definition $f = t$;
- $y_t \supseteq (a \in y_{t_0}); t_j$ whenever $t \equiv \mathbf{case}\ t_0\ \mathbf{of}\ q_1 \to t_1; \ldots; q_m \to t_m$, $q_j \equiv c\ v_1 \ldots v_n$ and $a \in A$ is of the form $a \equiv c\, s_1 \ldots s_n$;
- $y_{v_i} \supseteq (a \in y_{t_0}); y_{s_i}$ if $q_j \equiv c\ v_1 \ldots v_n$ for some **case**-expression $\mathbf{case}\ t_0\ \mathbf{of}\ q_1 \to t_1; \ldots; q_m \to t_m$ of p and $a \equiv c\ s_1 \ldots s_n \in A$.

References

[1] Martin Alt and Florian Martin. Generation of Efficient Interprocedural Analyzers with PAG. In *Proceedings of 2nd Static Analysis Symposium (SAS)*, pages 33–50. LNCS 983, 1995.

[2] Isaac Balbin and Kotagiri Ramamohanarao. A Generalization of the Differential Approach to Recursive Query Evaluation. *Journal of Logic Programming (JLP)*, 4(3):259–262, 1987.

[3] Francois Bancilhon and Raghu Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In *ACM SIGMOD Conference 1986*, pages 16–52, 1986.

[4] Francois Bancilhon and Raghu Ramakrishnan. Performance Evaluation of Data Intensive Logic Programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 12, pages 439–517. Morgan Kaufmann Publishers, 1988.

[5] Baudouin Le Charlier and Pascal Van Hentenryck. A Universal Top-Down Fixpoint Algorithm. Technical Report CS-92-25, Brown University, Providence, RI 02912, 1992.

[6] Baudouin Le Charlier and Pascal Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions of Programming Languages and Systems (TOPLAS)*, 16(1):35–101, 1994.

[7] Michael Codish, Dennis Dams, and Eyal Yardeni. Derivation of an Abstract Unification Algorithm for groundnes and Aliasing Analysis. In *International Conference on Logic Programming (ICLP)*, pages 79–93, 1991.

[8] Vincent Englebert, Baudouin Le Charlier, Didier Roland, and Pascal Van Hentenryck. Generic Abstract Interpretation Algorithms for Prolog: Two Optimization Techniques and their Experimental Evaluation. *Software – Practice and Experience*, 23(4):419–459, 1993.

[9] Christian Fecht. GENA - A Tool for Generating Prolog Analyzers from Specifications. In *Proceedings of 2nd Static Analysis Symposium (SAS)*, pages 418–419. LNCS 983, 1995.

[10] Christian Fecht. *Abstrakte Interpretation logischer Programme: Theorie, Implementierung, Generierung.* PhD thesis, Universität des Saarlandes, Saarbrücken, 1997.

[11] Christian Fecht and Helmut Seidl. An Even Faster Solver for General Systems of Equations. In *Proceedings of 3rd Static Analysis Symposium (SAS)*, pages 189–204. LNCS 1145, 1996. Long version to appear in SCP'99.

[12] Nevin Heintze. Set-Based Analysis of ML Programs. In *ACM Conference on Lisp and Functional Programming (LFP)*, pages 306–317, 1994.

[13] Nevin Heintze and David A. McAllester. On the Cubic Bottleneck in Subtyping and Flow Analysis. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 342–351, 1997.

[14] Pascal Van Hentenryck, Olivier Degimbe, Baudouin Le Charlier, and Laurent Michel. Abstract Interpretation of Prolog Based on OLDT Resolution. Technical Report CS-93-05, Brown University, Providence, RI 02912, 1993.

[15] Pascal Van Hentenryck, Olivier Degimbe, Baudouin Le Charlier, and Laurent Michel. The Impact of Granularity in Abstract Interpretation of Prolog. In *Proceedings of Static Analysis, 3rd International Workshop (WSA)*, pages 1–14. LNCS 724, 1993.

[16] Susan Horwitz, Thomas W. Reps, and Mooly Sagiv. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. In *Proceedings of 6th International Conference on Theory and Practice of Software Development (TAPSOFT)*, pages 651–665. LNCS 915, 1995.

[17] Susan Horwitz, Thomas W. Reps, and Mooly Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 49–61, 1995.

[18] David Melski and Thomas W. Reps. Interconvertability of Set Constraints and Context-Free Language Reachability. In *ACM SIGPLAN Symposium of Partial Evaluation and Semantic-Based Program Manipulation (PEPM)*, pages 74–89, 1997.

[19] K. Muthukumar and Manuel V. Hermenegildo. Compile-Time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming (JLP)*, 13(2&3):315–347, 1992.

[20] Hanne Riis Nielson and Flemming Nielson. Infinitary Control Flow Analysis: A Collecting Semantics for Closure Analysis. In *Proceedings of 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 332–345, 1997.

[21] Robert Paige. Symbolic Finite Differencing – Part I. In *Proceedings of 3rd European Symposium on Programming (ESOP)*, pages 36–56. LNCS 432, 1990.

[22] Jens Palsberg. Closure Analysis in Constraint Form. *ACM Transactions on Programming Languages and Systems*, 17:47–82, 1995.

[23] Jens Palsberg and Patrick O'Keefe. A Type System Equivalent to Flow Analysis. *ACM Transactions on Programming Languages and Systems*, 17:576–599, 1995.

[24] Helmut Seidl and Christian Fecht. Interprocedural Analysis Based on PDAs. Technical Report 97-06, University Trier, 1997. Extended Abstract in: *Verification, Model Checking and Abstract Interpretation.* A Workshop in Assiciation with ILPS'97.

[25] Helmut Seidl and Morten Heine Sørensen. Constraints to Stop Higher-Order Deforestation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 400–413, 1997.

[26] Harald Søndergaard. An Application of Abstract Interpretation of Logic Programs: Occur Check Reduction. In *Proceedings of 1st European Symposium on Programming (ESOP)*, pages 327–338. LNCS 213, 1986.