Propagating Differences: An Efficient New Fixpoint Algorithm for Distributive Constraint Systems

Christian Fecht[1] and Helmut Seidl[2]

[1] Universität des Saarlandes, Postfach 151150, D-66041 Saarbrücken, Germany,
fecht@cs.uni-sb.de
[2] Fachbereich IV – Informatik, Universität Trier, D-54286 Trier, Germany,
seidl@psi.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 [17] as well as for control-flow analysis for functional languages [13] 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 [15, 16, 25] for Prolog.

1 Introduction

Efficient application independent local solvers for general classes of constraint systems have been successfully used in program analyzers like GAIA [9, 6], PLAIA [20] or GENA [10, 11] for Prolog and 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advantages of application independence are obvious: the algorithmic ideas can be pointed out more clearly and are not superseded by application specific aspects.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 can therefore be proven more easily. Once proven correct, the solver then can be instantiated to different application domains – thus allowing for reusable implementations. For the overall correctness of every such 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.

In [12], 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 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.

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 *topdown*-solver **TD** of Le Charlier and Van Hentenryck [5, 12] a related technique has been suggested and practically applied to the analysis of Prolog, by Englebert et al. in [9].

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 [22]. 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., $f\ (s_1 \cup s_2) = f\ s_1 \cup f\ s_2$. An extension to n-ary f is contained in [2]. A general combination, however, of this principle with continuations and local worklist solvers seems to be new. 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 [12]. Thus, we are free to add timestamps or just depth-first priorities to obtain even more competitive fixpoint algorithms (see [12]). 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 [17], control-flow analysis for higher-order functional languages [13], and abstract `OLDT`-resolution as considered in [15, 16] 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 [18, 19]. 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 binary operators and 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 differential fixpoint algorithm $\mathbf{WR}_\Delta$. Conventional worklist solver **WR** 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. The results of section 9 are sufficient to derive fast

algorithms for framework *IDE* (section 10) as well as control-flow analysis (section 11). Framework *IDE* has been proposed by Horwitz, Reps and Sagiv for interprocedural analysis of imperative programs and applied to interprocedural *linear constant propagation* [17]. A variant of control-flow analysis ("set-based analysis") has been proposed by Heintze for the analysis of ML [13]. Another variant, even more in the spirit of the methods used here, has been applied in [26] to a higher-order functional language with call-by-need semantics to obtain a termination analysis for *deforestation*. In section 12 we extend 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 [15, 16, 25] (section 13). As an example implementation, we integrated an enhanced version of fixpoint algorithm $\mathbf{WR}_\Delta$ into program analyzer generator GENA for Prolog [10, 11] 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. Le Charlier and Van Hentenryck in [5] and Fecht and Seidl in [12] 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 [12] 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, 12], however, our new algorithms take also into account *how* right-hand sides are evaluated.

3 Computation Trees

Operationally, every evaluation of right-hand side t on variable assignment σ 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 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:

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

Clearly, ct x can be viewed as an abbreviation of ct $\langle x, \lambda d.d\rangle$. Representations of equivalent computation trees can 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 (mostly even without occurrences of *binary* operators g). The computation tree t for expression e can be chosen as e itself if $e \in D \cup V$. For $e \equiv f\, x$, we set $t = \langle x, f\rangle$ and for $e \equiv g\,(x_1, x_2)$, $t = \langle x_1, C\rangle$ where $C\, d = \langle x_2, C_d\rangle$ and $C_d\, d' = g\,(d, d')$. □

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 for right-hand sides. In the second place, however, ct's (resp. their generalization as considered in section 12) can be viewed as the *conceptual* intermediate representation for our fixpoint iterators to rely on, meaning, that evaluation of right-hand sides should provide for every access to 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 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 $\mathcal{S}$. 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 [8, 12], the constraint systems for interprocedural analysis of (imperative or logic) languages often are not monotonic but just *weakly* 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 "$\leq$" 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 "=". We have:

Proposition 2. 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 coincides with what was called "weakly monotonic" in [12] – adapted to constraint systems. It is a derived property here since we started out from *syntactic* properties of computation trees (not just their meanings as in [12]). As in [12] we conclude:

Corollary 3. If $\mathcal{S}$ 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*, finally, 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 solver **WR** we consider is a local worklist algorithm enhanced with recursive descent into new variables (Fig. 1). Solver **WR** is an adaption of a simplification of solver **WRT** in [12] to constraint systems. Opposed to **WRT**, no time stamps 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 $\sigma\ x$. The set of constraints to be reevaluated is kept in data structure W, called *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.

```
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**.

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 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, right-hand sides t of constraints r are *not* evaluated on σ directly. 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 4. 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 *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 (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 right-hand side of constraint $x \sqsupseteq t$, subtree $t' = \langle y, C\rangle$ is reached, we 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.

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 obtains elements from $V \times \mathcal{T}(V, D)$. In order to have continuations explicitly available for insertion into infl-sets, we change the functionality of the argument of $[\![.]\!]$ (and hence also $[\![.]\!]$) by passing down the current continuation into the argument. We define:

$$[\![d]\!]\ \sigma' = d \qquad [\![x]\!]\ \sigma' = \sigma'\ (\lambda d.d)\ x \qquad [\![\langle x, C\rangle]\!]\ \sigma' = [\![C\ (\sigma'\ C\ x)]\!]\ \sigma'$$

where $\sigma'\ C\ x = \sigma\ x$. Using this modified definition of the semantics of computation trees we finally adapt the functionality of Eval. 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 $\sigma\, y$. We obtain:

Theorem 5. Algorithm $\mathbf{WR}_C$ is a solver. □

A similar optimization for topdown solver **TD** [5, 12] in the context of analysis of Prolog programs has been called *clause prefix optimization* [9]. As far we know, an *application independent* exposition for worklist based solvers has not considered before.

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, we may initiate reevaluation just for the *increment*. This optimization is especially 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

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

Example 6. If $D = 2^A$, A a set, we may choose for diff set difference.

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

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:

$$\mathsf{dep}(C\ d_1, \sigma) \cup \mathsf{dep}(C\ d_2, \sigma) \supseteq \mathsf{dep}(C\ d, \sigma) \quad \text{and} \quad [\![C\ d_1]\!]\ \sigma \sqcup [\![C\ d_2]\!]\ \sigma \sqsupseteq [\![C\ d]\!]\ \sigma$$

```
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. 2. Algorithm $\mathbf{WR}_\Delta$.

In interesting applications, $\mathcal{S}$ is even monotonic and variable dependencies are "static", i.e., independent of σ. Furthermore, equality holds in the second inclusion. This is especially the case if right-hand sides are given through expressions as in Example 1, where 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 infl y after use. The resulting *differential* worklist algorithm with recursive descent into new variables ($\mathbf{WR}_\Delta$ for short) is given in Figure 2.

Theorem 7. For distributive $\mathcal{S}$, $\mathbf{WR}_\Delta$ computes an X-stable partial least solution. □

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 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 et al. [17] and applied, 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 for 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 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:

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

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 6, we obtain:

Theorem 8. 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 $\mathcal{O}(|p| \cdot |M|^3)$. □

The complexity bound in Theorem 8 should be compared with $\mathcal{O}(|p| \cdot |M|^5)$ which can be derived for **WR**. By saving factor $|M|^2$, we find the same complexity as has been obtained in [17] 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 [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., [23, 24, 21, 26]. Let A denote the set of subexpressions of p and $D = 2^A$. Then cfa for a lazy language as in [26] 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:

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

for $a \in A$. 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. 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 set diffrence. 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 $I = \mathcal{O}(\log\ s)$.

Cfa introduces $\mathcal{O}(|p|^2)$ constraints of the form $y \supseteq (a \in x_1); x_2$. Inorder to avoid creation of (a representation of) all these in advance, we introduce the following additional optimization. We start iteration with constraint system $\mathcal{S}_0$ lacking all constraints of this form. 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, $r\ x\ (d_1 \cup d_2) = (r\ x\ d_1) \cup (r\ x\ d_2)$. Whenever variable x is incremented by Δ, we add all constraints from $r\ x\ \Delta$ to the current constraint system by 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, we obtain:

Theorem 9. 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. Here, it results in an efficient algorithm which is comparable to the one proposed by Heintze in [13].

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. 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:

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

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:

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

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 10. Assume expressions e are given by the grammar:

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

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 $e \equiv g\,(e_1, e_2)$. □

For partial ordering "$\leq$" on set V of variables, we define relation "$\leq$" on gct's 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 the right-hand sides of constraint system $\mathcal{S}$ all 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. 2, cor. 3 as well as Theorem 4 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 $[\![.]\!]$ by defining:

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

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:

$$\mathsf{app}\,[]\;d\;\sigma' = d \qquad \mathsf{app}\,(C{:}\gamma)\;d\;\sigma' = \mathsf{app}\,\gamma\,([\![C\;d]\!]\;\sigma'\;\gamma)\;\sigma'$$

The resulting value then gives the new contribution to the value of σ for x. As for $\mathbf{WR}_\Delta$ for ordinary evaluation trees, infl-sets cannot be emptied after use. Carrying the definition of distributivity from section 9 over to gct's, we obtain:

Theorem 11. For distributive $\mathcal{S}$ with gct's as right-hand sides, algorithm $\mathbf{WR}_\Delta$ computes an X-stable partial least solution. □

As an advanced application of gct's, we consider abstract OLDT-resolution [15, 16, 25].

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 [15]. 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 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 [25]):

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

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 basic computation steps, passing of actual parameters into procedures 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 with the local state before the call [25]. 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 for variable $x\,a, a \in A$, are obtained from the set of constraint schemes for x by instantiating $\bullet$ 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 forms for elements s_a of S_d correspond to the three possible forms for f in expressions $\mathcal{E}\ f\ e$, i.e,

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

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 $\mathbf{WR}_\Delta$. Let us assume that applications $g\,a$ and insertion into sets can be executed in time $\mathcal{O}(1)$, whereas evaluation of $d \,\square\, a$ needs time $\mathcal{O}(\#d)$. Then:

Theorem 12. 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 $N \leq |p| \cdot \#A$ is the number of considered variables, and $s \leq \#A$ is the maximal cardinality of occurring sets. □

$\mathbf{WR}_\Delta$ saves an extra factor $\mathcal{O}(s^2)$ over solver $\mathbf{WR}$. An algorithm with similar savings has been suggested by Horwitz et al. [18]. Their graph-based algorithm, however, is neither able to treat *binary* operators nor deeper nested right-hand sides as ours.

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 [15, 25]. Here, subset $d \subseteq A$ is called lower iff $d = d{\downarrow}$. If all occurring functions f as well as operators $\square$ are monotonic, then 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

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

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.

14 Practical Experiments

We have adapted the fastest general-purpose equation solver from [12], namely **WDFS** (for a distinction 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**, 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 (cf. [12]). To solver $\mathbf{WDFS}^{Con}$ we added propagation of differences (the "Δ") in the same way as we added propagation of differences to solver **WR** in section 9. All fixpoint algorithms have been integrated into GENA [10, 11]. 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 [27, 7]. Its abstract substitutions are pairs of bdd's and graphs over variables. Thus, we maintain anti-chains of such elements. The generated analyzers were run on large real-world programs. `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 1, are the absolute runtimes in seconds (including system time) obtained for SML-NJ, version 109.29, on a Sun 20 with 64 MB main memory.

Comparing the three algorithms for OLDT-resolution, we find that all of these have still quite acceptable runtimes (perhaps with exeption of `aqua-c`) where algorithm $\mathbf{WDFS}^{Con}_{\Delta}$ almost always outperforms the others. Compared with equation solver $\mathbf{WDFS}^{Equ}$, algorithm $\mathbf{WDFS}^{Con}_{\Delta}$ saves approximately 40% of the runtime where usually less than half of the gain is obtained by maintaining constraints. The maximal relative gain of 48% could be observed for program `readq` where no advantage at all could be drawn out of constraints. Opposed to that, for Stefan Diehl's interpreter for action semantics `action`, propagation of differences did not give (significantly) better numbers than considering constraints alone – program `ann` even showed a (moderate) decrease in efficiency (factor 3.32). Also opposed to the general picture, constraints for `aqua-c` resulted in an improvement of 25% only – of which 9% was lost through propagation of differences! This slowdown is even more surprising, since it could not be confirmed with analyzer runs on `aqua-c` for other abstract domains. Table 2 reports the runtimes found for `aqua-c` on domain CompCon. Abstract domain CompCon analyzes whether variables are bound to atoms or are composite. For CompCon, constraints introduced a slowdown of 18% whereas propagation of differences resulted in a gain of efficiency by 38% over $\mathbf{WDFS}^{Equ}$.

program	$\mathbf{WDFS}^{Equ}$	$\mathbf{WDFS}^{Con}$	$\mathbf{WDFS}^{Con}_{\Delta}$
action.pl	32.97	19.37	19.35
ann.pl	1.36	1.62	4.52
aqua-c.pl	1618.00	1209.00	1361.00
b2.pl	2.41	2.14	1.82
chat.pl	77.73	67.94	53.14
chat-parser.pl	29.62	27.72	17.28
chess.pl	0.40	0.38	0.37
flatten.pl	0.36	0.34	0.26
nand.pl	0.47	0.38	0.32
readq.pl	14.96	15.09	7.85
sdda.pl	0.73	0.59	0.50

Table 1. Comparison of $\mathbf{WDFS}^{Equ}$, $\mathbf{WDFS}^{Con}$, and $\mathbf{WDFS}^{Con}_{\Delta}$ with PS+POS+LIN.

program	$\mathbf{WDFS}^{Equ}$	$\mathbf{WDFS}^{Con}$	$\mathbf{WDFS}^{Con}_{\Delta}$
aqua-c.pl	214.67	252.43	133.61

Table 2. Comparison of $\mathbf{WDFS}^{Equ}$, $\mathbf{WDFS}^{Con}$, and $\mathbf{WDFS}^{Con}_{\Delta}$ with CompCon.

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 a common algorithmic idea in seemingly different fast special purpose algorithms like the one of Horwitz et al. for interprocedural framework *IDE* [17] of Heintze's algorithm for control-flow analysis [13]. Our exposition furthermore explains how such algorithms can be practically *improved* – namely by incorporating recursive descent into variables as well as timestamps [12]. Finally, our approach allowed to develop completely new efficient algorithms for abstract `OLDT`-resolution.

References

1. M. Alt and F. Martin. Generation of Efficient Interprocedural Analyzers with PAG. In *2nd SAS*, 33–50. LNCS 983, 1995.
2. I. Balbin and K. Ramamohanarao. A Generalization of the Differential Approach to Recursive Query Evaluation. *JLP*, 4(3):259–262, 1987.
3. F. Bancilhon and R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In *ACM SIGMOD Conference 1986*, 16–52, 1986.
4. F. Bancilhon and R. Ramakrishnan. Performance Evaluation of Data Intensive Logic Programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 12, 439–517. Morgan Kaufmann Publishers, 1988.
5. B. Le Charlier and P. Van Hentenryck. A Universal Top-Down Fixpoint Algorithm. Technical Report CS-92-25, Brown University, Providence, RI 02912, 1992.
6. B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM TOPLAS*, 16(1):35–101, 1994.
7. M. Codish, D. Dams, and E. Yardeni. Derivation of an Abstract Unification Algorithm for groundnes and Aliasing Analysis. In *ICLP*, 79–93, 1991.

8. P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Recursive Programs. In E.J. Neuhold, editor, *Formal Descriptions of Programming Concepts*, 237–277. North-Holland Publishing Company, 1977.
9. V. Englebert, B. Le Charlier, D. Roland, and P. Van Hentenryck. Generic Abstract Interpretation Algorithms for Prolog: Two Optimization Techniques and their Experimental Evaluation. *SPE*, 23(4):419–459, 1993.
10. C. Fecht. GENA - A Tool for Generating Prolog Analyzers from Specifications. *2nd SAS*, 418–419. LNCS 983, 1995.
11. C. Fecht. *Abstrakte Interpretation logischer Programme: Theorie, Implementierung, Generierung.* PhD thesis, Universität des Saarlandes, Saarbrücken, 1997.
12. C. Fecht and H. Seidl. An Even Faster Solver for General Systems of Equations. *3rd SAS*, 189–204. LNCS 1145, 1996. Extended version to appear in SCP'99.
13. N. Heintze. Set-Based Analysis of ML Programs. *ACM Conf. LFP*, 306–317, 1994.
14. N. Heintze and D.A. McAllester. On the Cubic Bottleneck in Subtyping and Flow Analysis. *IEEE Symp. LICS*, 342–351, 1997.
15. P. Van Hentenryck, O. Degimbe, B. Le Charlier, and L. Michel. Abstract Interpretation of Prolog Based on OLDT Resolution. Technical Report CS-93-05, Brown University, Providence, RI 02912, 1993.
16. P. Van Hentenryck, O. Degimbe, B. Le Charlier, and L. Michel. The Impact of Granularity in Abstract Interpretation of Prolog. *3rd WSA*, 1–14. LNCS 724, 1993.
17. S. Horwitz, T.W. Reps, and M. Sagiv. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *6th TAPSOFT*, 651–665. LNCS 915, 1995.
18. S. Horwitz, T.W. Reps, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. *22nd POPL*, 49–61, 1995.
19. D. Melski and T.W. Reps. Interconvertability of Set Constraints and Context-Free Language Reachability. *ACM SIGPLAN Symp. PEPM*, 74–89, 1997.
20. K. Muthukumar and M. V. Hermenegildo. Compile-Time Derivation of Variable Dependency Using Abstract Interpretation. *JLP*, 13(2&3):315–347, 1992.
21. H. Riis Nielson and F. Nielson. Infinitary Control Flow Analysis: A Collecting Semantics for Closure Analysis. *24th POPL*, 332–345, 1997.
22. R. Paige. Symbolic Finite Differencing – Part I. *3rd ESOP*, 36–56. LNCS 432, 1990.
23. J. Palsberg. Closure Analysis in Constraint Form. *ACM TOPLAS*, 17:47–82, 1995.
24. J. Palsberg and P. O'Keefe. A Type System Equivalent to Flow Analysis. *ACM TOPLAS*, 17:576–599, 1995.
25. H. Seidl and C. 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.
26. H. Seidl and M.H. Sørensen. Constraints to Stop Higher-Order Deforestation. *24th POPL*, 400–413, 1997.
27. H. Søndergaard. An Application of Abstract Interpretation of Logic Programs: Occur Check Reduction. *1st ESOP*, 327–338. LNCS 213, 1986.