Interprocedural Analysis Based on PDAs

Helmut Seidl

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

Christian Fecht

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

Abstract. We systematically explore the design space of constraint systems for interprocedural analysis both of imperative and logic languages. Our framework is based on a small-step operational semantics where both the concrete and the abstract operational semantics are formalized by means of (input-free) *pushdown automata*. We consider the analysis problem of *derivability* and present constraint systems for a corresponding *relational analysis* with *forward* as well as *backward* (intraprocedural) accumulation. We abstract these constraint systems to obtain constraint systems for corresponding *functional* analyses. We clarify the relative precision of forward versus backward accumulation in presence of more or less complex intraprocedural control-flow graphs.

1 Introduction

Static analysis aims at computing statements about the runtime behavior of a program without actually executing the program. We propose a general framework for interprocedural analysis based on a *small-step* operational semantics [9, 36]. In order to derive an analysis engine, our framework proceeds in three steps. First, the concrete operational semantics is *simulated* by an abstract operational semantics [36]. To this end, we only "abstract" data and the operations on data. Especially, we *maintain* the control structure, implying that this abstract operational semantics has the same "structure" as the concrete one. The analysis problems have thus been transformed to determine properties of this abstract operational semantics. In the second step, a constraint system is selected which (more or less precisely) characterizes the property to be analyzed. In the last step, a constraint solver is added to compute a (least) solution of the constraint system.

In this paper, we concentrate on the first two steps. We do this with respect to the analysis problem of determining the *effects* of procedures. This analysis is also called *derivability analysis* [38]. An analogous treatment of other analysis problems e.g., *reachability* of program states is possible as well. Indeed, determining the *effects* of procedures can be based on an abstraction of a *denotational semantics* of the program as in [11, 23, 29, 7] as well. Execution of programs, however, and reachability of program points with states are essentially *operational* notions. Therefore, a corresponding program analysis can only be conveniently defined as well as rigorously proven correct if it is based on the abstraction of a *small-step* operational semantics in the sense of [9, 36]. For Prolog, corresponding frameworks have been proposed, e.g., by Bruynooghe [5], Nilsson [31–33] and Fecht [15]. Opposed to Bruynooghe, Nilsson's as well as our semantics for Prolog are very similar in spirit to a corresponding small-step operational semantics for imperative languages with procedures [24, 2, 4].

Since the very beginning of the design of programming languages with procedures, it has been observed that pushdowns are the adequate data structure to implement subroutine calls and recursion [34, 30, 27, 35]. Besides being therefore the natural choice at hand for the operational semantics, pushdown automata also offer the opportunity to apply constructions and theorems for context-free languages well-known since the sixties. We therefore formalize the both the concrete operational semantics of our procedural languages as well as the abstract one by means of input-free *pushdown* *automata* (pda's). For imperative languages, abstract pda's have also been considered by Knoop and Steffen in [26]. In contrast to [26], we systematically exploit the potential of the pda-formalism offered by Formal Language Theory. Applying the *triple construction* [8, 37, 13] to the pda (no matter whether it is concrete or abstract) for a given program, we elegantly derive constraint systems whose least solutions *precisely* characterize derivability. They are the basis for *relational analysis* of programs. No such constraint systems are contained in [26]. Relational analysis has been called `OLDT`-*resolution* by Van Hentenryck et al. and investigated for Prolog [18, 19]. The coincidence of the abstract operational semantics (defined through pda's) and relational analysis derived by Formal Language Theory, can be interpreted as general "interprocedural coincidence theorem". Abstracting the constraint systems of relational analysis, we obtain constraints systems for *functional* analysis. Current analyzer generators [1, 14, 15] both of imperative and logic languages are mostly based on functional analysis, however, with *forward* accumulation meaning that the effect of procedures are successively accumulated starting from the entry points of procedures. Opposed to that, constraint systems with *backward accumulation* successively collect the effects of procedures in a backward fashion starting from the exit points of procedures. This idea has been suggested by Van Hentenryck et al. as one optimization for `OLDT`-resolution [18]. To the best of our knowledge, functional analysis with backward accumulation has not been considered so far.

The overall structure of our paper is as follows. Section 2 introduces the standard notion of pushdown automata (pda's). Section 3 introduces the subclass of pushdown automata relevant for program analysis and how it is related to the well-known notion of interprocedural control-flow graphs. As an example, Section 4 presents a pda-based small-step operational semantics for (a pure subset of) Prolog. Section 5 introduces the notion of "simulation" between pda's. Section 6 systematically derives constraint systems with forward accumulation, whereas Section 7 systematically derives constraint systems with backward accumulation. Especially, we prove that relational analysis (either with forward accumulation or backward accumulation) is "optimal" in the sense that the respective analysis problems are precisely solved. Section 8 compares the constraint systems with forward accumulation with the corresponding ones using backward accumulation w.r.t. precision.

2 Pushdown Automata

An input-free *pushdown automaton* M (accepting with empty pushdown) is a triple $(D, \Gamma, \vdash)$ where Γ is the set of *pushdown symbols*, D is the set of *states* (or *values*) and $\vdash$ is the *transition relation*. The set of *configurations*[1] of M equals the set $D \times \Gamma^*$. $\vdash$ consists of the following three types of transitions:

$$\begin{array}{ll} (d_1, A) \vdash (d_2, \epsilon) & \quad (\mathit{pop}) \\ (d_1, A) \vdash (d_2, B) & \quad (\mathit{shift}) \\ (d_1, A) \vdash (d_2, B_1 B_2) & \quad (\mathit{push}) \end{array}$$

for $d_1, d_2 \in D$ and $A, B, B_1, B_2 \in \Gamma$. While shifts correspond to intraprocedural computation steps, pushs and pops are used to model calls to and returns from procedures.

Relation $\vdash$ is extended to a relation $\vdash_M$ on configurations by $(d_1, w_1) \vdash_M (d_2, w_2)$ iff $w_1 = Aw$ and $w_2 = vw$ with $(d_1, A) \vdash (d_2, v)$. If no confusion may arise, we will also skip the index M at $\vdash$.

We are interested in the following two questions:

Derivability: Given (d, A), to determine the set of all d' with $(d, A) \vdash^* (d', \epsilon)$;
Same-Level Reachability: Given (d, A) and B, to determine the set of all d' with $(d, A) \vdash^* (d', B)$.

The first question asks for the functional behavior of pushdown symbols, notably procedures. The second question can be thought of as the pda formulation of *intraprocedural* reachability.

[1] We assume that the input tape usually attached to pda's always contains the empty word ϵ. Also every transition consumes just ϵ. Therefore, we omit all components referring to input.

We observe that (for application in an optimizing compiler) it suffices to compute derivability information for A only for those values d for which (d, A) is reachable from the initial configuration (or abstractions of this set). In this case, we speak of *local derivability* analysis as considered by Sharir and Pnueli [39] for imperative languages and, e.g., by Le Charlier and Van Hentenryck [6, 7] or Fecht and Seidl [14, 16] for Prolog.

3 Specifying PDAs

We put the following restrictions on the structure of pda's. Let D denote the set of states. Let Item denote a finite set of *items* (independent of D). Item is the disjoint union of sets Call, Return and Point where Call is the set of *call* items, Return is the set of *return* items and Point collects the remaining program points. Let $\mathsf{Proc} \subseteq \mathsf{Point}$ denote a finite set of *procedure entry points* (or *procedures*) where $S \in \mathsf{Proc}$ is the *main* procedure of the program. The called procedure of a call item can be obtained by means of function $proc : \mathsf{Call} \to \mathsf{Proc}$ whereas the successor item of a call is obtained by $next : \mathsf{Call} \to \mathsf{Item}$. Let $\mathsf{Edge} \subseteq \mathsf{Point} \times \mathsf{Item}$ denote the set of all pairs (P, A) between which *intraprocedural* computation steps are possible. Item, $proc$, $next$, Edge together with S can be viewed as a general form of *interprocedural control-flow graph* (interprocedural CFG). For convenience we additionally assume that set Item is the disjoint union of collections $\mathsf{Item}_p, p \in \mathsf{Proc}$, the *intra-procedural* program points of procedures p. Furthermore,

1. $p \in \mathsf{Item}_p$ and p has no in-going edge;
2. $P \in \mathsf{Item}_p$ and $(P, A) \in \mathsf{Edge}$ implies $A \in \mathsf{Item}_p$;
3. $C \in \mathsf{Call} \cap \mathsf{Item}_p$ and $A = next\ C$ implies $A \in \mathsf{Item}_p$.

The 4-tuple $\mathcal{G}_p = (\mathsf{Item}_p, E_p, p, \mathcal{R}_p)$ where

$$\begin{aligned} E_p &= (\mathsf{Edge} \cap \mathsf{Item}_p^2) \cup \{(C, next\ C) \mid C \in \mathsf{Call} \cap \mathsf{Item}_p\} \\ \mathcal{R}_p &= \mathsf{Item}_p \cap \mathsf{Return} \end{aligned}$$

is also called *intraprocedural* control-flow graph for p.

Computation on data is succinctly represented through (possibly partial) functions

$\mathsf{Entry} : \mathsf{Call} \to D \to D$	*procedure entry*
$\mathsf{Exit} : \mathsf{Return} \to D \to D$	*procedure exit*
$\mathsf{Trans} : \mathsf{Edge} \to D \to D$	*intraprocedural computation step*
$\mathsf{Comb} : \mathsf{Call} \to (D \times D) \to D$	*resume after call*

Functions Entry and Exit specify how data are passed into the called procedure resp. returned back. Function Trans specifies for every edge (P, A) the corresponding *transfer function*. Function Comb corresponds to *combine* in [24] resp. $\mathcal{R}$ in [26, 25]. It combines the value before a call with the result returned by the called procedure. This binary function allows to model local variables of procedures conveniently. Let us call the collection of the four functions Entry, Exit, Trans and Comb *behavior*. In the following, to reduce the number of arguments, we will write the first arguments to behavioral functions (i.e., those in Item resp. Edge) also as subscripts. The pda M describing the operational semantics then is completely specified by the interprocedural CFG together with a behavior. The set Γ of pushdown symbols is obtained by $\Gamma = \mathsf{Item} \cup (D \times \mathsf{Call})$, while its configurations are of the form $(d, A\ [d_1, C_1] \ldots [d_m, C_m])$ where $d, d_1, \ldots, d_m \in D$, $C_1, \ldots, C_m \in \mathsf{Call}$ and $A \in \mathsf{Item}$. Here, A corresponds to the program point in the currently active procedure call, whereas the $[d_j, C_j]$ represent stacked frames for calls whose evaluation has been started but not yet completed. The set of transitions is given by:

$$\begin{array}{lll} \textit{Push}: & (d, C) \vdash (\mathsf{Entry}_C\ d, p\ [d, C]) & \text{if } p \equiv proc\ C \\ \textit{Pop}: & (d, R) \vdash (\mathsf{Exit}_R\ d, \epsilon) & \\ \textit{Shift}: & (d, P) \vdash (\mathsf{Trans}_{(P,A)}\ d, A) & \\ \textit{Unpack}: & (d_1, [d_2, C]) \vdash (\mathsf{Comb}_C\ (d_1, d_2), A) & \text{if } A \equiv next\ C \end{array}$$

where $C \in \mathsf{Call}$, $P \in \mathsf{Point}, R \in \mathsf{Return}$ and $A \in \mathsf{Item}$.

Note that we splitted the set of shift transitions into one set involving ordinary program points and one involving unpacking of stacked frames $[d_2, C]$. Non-deterministic choices can only occur between different shifts where different items A in the right-hand sides are involved.

In [24], Jones and Muchnick consider imperative programs consisting of a finite set of non-nested procedure definitions where each procedure may have local variables, use value as well as result parameters (as in Algol60) and show that their operational semantics can be formalized by pda's. The same class of programs is dealt with in [11, 26]. In [2], Bourdoncle has shown that pda's are sufficient even if nested procedure declarations, global variables and reference parameters are allowed [2].

It turns out that the operational semantics of (the pure subset of) Prolog with SLD resolution can be described by pda's as well.

4 A Small-Step Operational Semantics for Logic Programs

A normalized logic program in the sense of [7, 12][2] consists of a set Pred of *predicates* together with a set Clause of predicate *definitions* and a main predicate $main \in \mathsf{Pred}$. A predicate definition for predicate p can be denoted by $p(X_1, \ldots, X_k) \leftarrow e$ where e consists of a sequence of *goals*. Goal g may either be a call like $p(X_{i_1}, \ldots, X_{i_k})$ (with variables X_{i_j} pairwise distinct) from Goal or a basic goal from Basic like $X = t$ or **true**. Thus, $e \in (\mathsf{Goal} \cup \mathsf{Basic})^*$.

To specify the (small-step) operational semantics of a given program by an input-free *pushdown automaton* M as described in the last section, we first introduce the interprocedural CFG corresponding to the program. The set of procedures is given by the set of predicates Pred. The set of items consists of all predicates p together with all program points $[h \leftarrow \alpha.\beta]$ where $h \leftarrow \alpha\beta \in \mathsf{Clause}$. Furthermore,

$$\begin{array}{ll}\mathsf{Call} & = \{[h \leftarrow \alpha.g\beta] \mid g \in \mathsf{Goal}, h \leftarrow \alpha g \beta \in \mathsf{Clause}\} \\ \mathsf{Return} & = \{[h \leftarrow \alpha.] \mid h \leftarrow \alpha \in \mathsf{Clause}\} \\ \mathsf{Point} & = \mathsf{Pred} \cup \{[h \leftarrow \alpha.b\beta] \mid b \in \mathsf{Basic}, h \leftarrow \alpha b \beta \in \mathsf{Clause}\}\end{array}$$

The set Edge of possible intraprocedural computation steps consists of all (P, A) where

- $P \equiv p \in \mathsf{Proc}$ and $A \equiv [p(X_1, \ldots, X_n) \leftarrow .\alpha]$; or
- $P \equiv [h \leftarrow \alpha.b\beta]$ and $A \equiv [h \leftarrow \alpha b.\beta]$ for $b \in \mathsf{Basic}$.

Functions *proc* and *next* are defined by:

$$\begin{array}{ll}proc[h \leftarrow \alpha.g\beta] = p & \text{if } g \equiv p(X_{i_1}, \ldots, X_{i_n}) \\ next[h \leftarrow \alpha.g\beta] = [h \leftarrow \alpha g.\beta] & \end{array}$$

It remains to define the behavior of M. Let D denote a set of *values*. The semantics of normalized logic programs is specified through the following four functions [7, 12]:

$$\begin{array}{lll}[\![.]\!] & : \mathsf{Basic} \to D \to D & \textit{meaning of basic goals} \\ \mathsf{restrG} & : \mathsf{Goal} \to D \to D & \textit{parameter passing into predicate} \\ \mathsf{extC} & : \mathsf{Clause} \to D \to D & \textit{initialization of selected clause} \\ \mathsf{restrC} & : \mathsf{Clause} \to D \to D & \textit{parameter passing back into predicate} \\ \mathsf{extG} & : \mathsf{Goal} \to D^2 \to D & \textit{effect of call}\end{array}$$

$[\![.]\!]$ assigns a meaning to basic goals, restrG describes how values are passed to the formal parameters of predicates, extC initializes the evaluation of the selected clause, restrC describes how the formal

[2] It is for simplicity that we restrict ourselves to normalized programs here. A similar operational semantics can be easily set up also for *non-normalized* programs.

parameters are affected by the clause whereas extG describes the effect of the call depending on the return value of the predicate and the value before the call.

Given these functions, the behavior of pda M is obtained by:

$$\begin{array}{lll}
\mathsf{Entry}\ C & = \mathsf{restrG}\ g & \text{if } C \equiv [h \leftarrow \alpha.g\beta] \\
\mathsf{Trans}\ (P,A) & = [\![b]\!] & \text{if } P \equiv [h \leftarrow \alpha.b\beta],\ A \equiv [h \leftarrow \alpha b.\beta] \\
\mathsf{Trans}\ (p,A) & = \mathsf{extC}\ (h \leftarrow \alpha) & \text{if } h \equiv p(X_1,\ldots,X_n),\ A \equiv [h \leftarrow .\alpha] \\
\mathsf{Exit}\ R & = \mathsf{restrC}\ (h \leftarrow \beta) & \text{if } R \equiv [h \leftarrow \beta.] \\
\mathsf{Comb}\ C & = \mathsf{extG}\ g & \text{if } C \equiv [h \leftarrow \alpha.g\beta]
\end{array}$$

Example 1. For the following running example assume that our Prolog dialect provides also *complex* goals of the form $(g_1; g_2)$ where ";" represents the OR operator. Then consider predicate definition $p \leftarrow (a;p)\ b$. Let us introduce the following abbreviations:

$$\begin{array}{lll}
I = [p \leftarrow .(a;p)\ b] & A = [p \leftarrow (.a;p)\ b] & R = [p \leftarrow (a;p)\ b.] \\
C = [p \leftarrow (a;.p)\ b] & B = [p \leftarrow (a;p)\ .b] &
\end{array}$$

Then the interprocedural CFG is given by (cf. Figure 1):

$$\begin{array}{ll}
\mathsf{Item} & = \{p, I, A, B, C, R\} \\
\mathsf{Call} & = \{C\} \quad \text{where} \quad \mathit{proc}\ C = p,\ \ \mathit{next}\ C = B \\
\mathsf{Return} & = \{R\} \\
\mathsf{Point} & = \{I, A, B\} \\
\mathsf{Edge} & = \{(p,I), (I,A), (I,C), (A,B), (B,R)\}
\end{array}$$

and the transition relation $\vdash$ is given by:

$$\begin{array}{lll}
(d,R) \vdash (\mathsf{Exit}_R\ d, \epsilon) & (d,I) \vdash (\mathsf{Trans}_{(I,A)}\ d, A) & (d,A) \vdash (\mathsf{Trans}_{(A,B)}\ d, B) \\
(d,p) \vdash (\mathsf{Trans}_{(p,I)}\ d, I) & (d,I) \vdash (\mathsf{Trans}_{(I,C)}\ d, C) & (d,B) \vdash (\mathsf{Trans}_{(B,R)}\ d, R) \\
(d,C) \vdash (\mathsf{Entry}_C\ d, p\ [d,C]) & (d_1,[d,C]) \ \vdash\ (\mathsf{Comb}_C\ (d_1,d), B) &
\end{array}$$

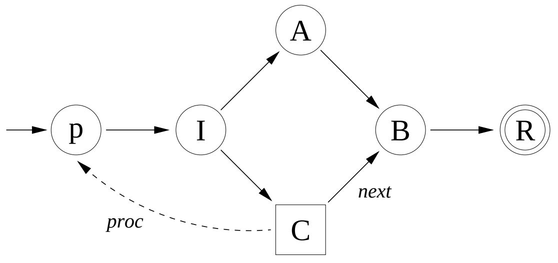

Fig. 1. The interprocedural CFG for $p \leftarrow (a;p)\ b$.

5 Simulating PDAs

Assume we are given a set $D^\sharp$ of abstract values together with a *simulation* relation $\Delta \subseteq D \times D^\sharp$. We assume that *every* concrete value $d \in D$ should possibly be simulated by at least one abstract value in $D^\sharp$. Therefore, Δ should be *left-total*. Usually, set $D^\sharp$ of abstract values is partially ordered where the ordering "$\sqsubseteq$" reflects the quality of approximation, i.e., $d\ \Delta\ a$ and $a \sqsubseteq a'$ implies $d\ \Delta\ a'$. If this is the case, we call simulation relation Δ *compatible*. Even more popular are analyses where $D^\sharp$ is a complete lattice, and Δ is given by means of an *abstraction* function $\alpha : 2^D \to D^\sharp$ which is surjective and commutes with arbitrary least upper bounds [10, 9]. Then especially, $\alpha X = \bigsqcup_{d \in X} \alpha\{d\}$ for all $X \subseteq D$. Given such an abstraction function, a compatible simulation relation Δ_α can be obtained by: $d\ \Delta_\alpha\ a$ iff $\alpha\{d\} \sqsubseteq a$.

We extend the notion of "simulation" to subsets $X \subseteq D$ and (partial) functions. If $X \subseteq D$ and $X^\sharp \subseteq D^\sharp$ then X is simulated by $X^\sharp$ (denoted: $X\ \Delta\ X^\sharp$) iff for all $d \in X$, $x\ \Delta\ a$ for some $a \in X^\sharp$. If $f : D^k \to D$ is a (possibly partial) function and $f^\sharp : (D^\sharp)^k \to D^\sharp$ is a total function, then f is *simulated* by $f^\sharp$ (denoted: $f\ \Delta\ f^\sharp$) iff $(d_1, \ldots, d_k) \in \mathrm{dom}(f)$, and $d_i\ \Delta\ a_i$ for all i implies $f(d_1, \ldots, d_k)\ \Delta\ f^\sharp(a_1, \ldots, a_k)$. Note here, that we do not make any assumptions w.r.t. monotonicity (or continuity) of concrete or abstract functions. We only demand abstract functions to be total and the simulation relation Δ to be respected.

Assume now M and $M^\sharp$ are pda's with sets of states D and $D^\sharp$, respectively. Pda M is *simulated* by pda $M^\sharp$ (abbreviated: $M\ \Delta\ M^\sharp$) iff their interprocedural CFGs are identical and the behavioral functions of M are simulated by the behavioral functions of $M^\sharp$. Furthermore, configuration $c \equiv (d_0, A\ [d_1, C_1] \ldots [d_m, C_m])$ of M is *simulated* by configuration $c^\sharp$ of $M^\sharp$ (abbreviated: $c\ \Delta\ c^\sharp$) iff $c^\sharp \equiv (a_0, A\ [a_1, C_1] \ldots [a_m, C_m])$ with $d_i\ \Delta\ a_i$ for all i.

Proposition 2. *1. Assume $M\ \Delta\ M^\sharp$, and c_1 and $c_1^\sharp$ are configurations of M and $M^\sharp$, respectively, with $c_1\ \Delta\ c_1^\sharp$. Then $c_1 \vdash_M c_2$ implies $c_1^\sharp \vdash_{M^\sharp} c_2^\sharp$ for some configuration $c_2^\sharp$ of $M^\sharp$ with $c_2\ \Delta\ c_2^\sharp$.*
2. Simulation between pda's is transitive. □

Theorem 3. *If pda M is simulated by pda $M^\sharp$ and $d\ \Delta\ a$ then for every $p \in \mathsf{Proc}$ and $A \in \mathsf{Item}_p$,*

1. $\{d' \in D \mid (d, A) \vdash_M^* (d', \epsilon)\}\ \Delta\ \{a' \in D^\sharp \mid (a, A) \vdash_{M^\sharp}^* (a', \epsilon)\}$;
2. $\{d' \in D \mid (d, p) \vdash_M^* (d', A)\}\ \Delta\ \{a' \in D^\sharp \mid (a, p) \vdash_{M^\sharp}^* (a', A)\}$. □

By Theorem 3, we conclude that – via simulation relation Δ – the analysis problems: derivability and same-level reachability for pda M are translated into the corresponding analysis problems for abstract pda $M^\sharp$. Therefore, it suffices to solve these problems (approximately) for abstract pda's.

6 Constraint Systems with Forward Accumulation

Our main technique consists in finding and transforming suitable constraint systems for the respective analysis problems. A *constraint system* $\mathcal{C}$ in variables from set X over complete lattice D consists of a set of constraints of the form $x \sqsupseteq e$ where $x \in X$ is a variable and e is denotes a function $[\![e]\!] : (X \to D) \to D$. A *model* or *solution* of $\mathcal{C}$ is a map $\sigma : X \to D$ such that $\sigma\ x \sqsupseteq [\![e]\!]\ \sigma$ for all constraints $x \sqsupseteq e$ of $\mathcal{C}$. All constraint systems $\mathcal{C}$ to be considered have a unique *least solution* which is denoted by $[\![\mathcal{C}]\!]$. Often we are going to put up constraint systems where variables p in fact are going to be bound to functions $D_1 \to D_2$. Instead of considering such functions as a whole we use the standard trick to divide variable p into a set of variables one for each entry in the value table for p. For convenience, we denote these individual variables by $p\ d$, $d \in D_1$. Note that we allow such formal applications within right-hand sides of constraints as well. If $e \equiv p\ e'$ then we define $[\![e]\!]\ \mu = \mu\ p\ ([\![e']\!]\ \mu)$.

When deriving constraint systems for derivability we are confronted with with several independent design choices:

1. whether we compute derivability information just for procedures and use same-level reachability for accumulating intraprocedural effects ("forward accumulation") or whether we compute derivability information *for all* items ("backward accumulation");
2. whether we use set-valued transformers from $D \to 2^D$ as values for our procedure variables ("relational analysis") or just ordinary transformers from $D \to D$ ("functional analysis").

We start our exposition with the more conventional approach of *forward accumulation.*

6.1 Relational Analysis

Let M be an abstract pda over set D of abstract values. According to the strategy of forward accumulation, constraint system $\mathcal{R}_f(M)$ introduces set-valued variables $p\ d, p \in \mathsf{Proc}$, for derivability of procedures as well as set-valued variables $\langle A, d\rangle$, $A \in \mathsf{Item}_p$, for same-level reachability from (d, p), i.e., the respective procedure entry (entered with state d). $\mathcal{R}_f(M)$ is defined by:

$$\begin{array}{lll}
p\ d \supseteq \mathsf{Exit}^*_R\ \langle R, d\rangle & \text{if } R \in \mathcal{R}_p & (1) \\
\langle p, d\rangle \supseteq \{d\} & \text{if } p \in \mathsf{Proc} & (2) \\
\langle A, d\rangle \supseteq \mathsf{Trans}^*_{(P,A)}\ \langle P, d\rangle & \text{if } (P, A) \in \mathsf{Edge} & (3) \\
\langle A, d\rangle \supseteq \mathcal{E}\ (\lambda x.\mathsf{Comb}^*_C\ (p\ (\mathsf{Entry}_C\ x), x))\ \langle C, d\rangle & \text{if } p = \mathit{proc}\ C,\ A = \mathit{next}\ C & (4)
\end{array}$$

where $\mathcal{E} : (D \to 2^D) \to 2^D \to 2^D$ is the usual extension function for the power set, defined by

$$\mathcal{E}\ f\ X = \bigcup_{x\in X} f\ x$$

Furthermore,

$$\begin{array}{ll}
\mathsf{Comb}^*_C\ (X, d) & = \{\mathsf{Comb}_C\ (x, d) \mid x \in X\} \\
\mathsf{Exit}^*_R\ X & = \{\mathsf{Exit}_R\ x \mid x \in X\} \\
\mathsf{Trans}^*_{(P,A)}\ X & = \{\mathsf{Trans}_{(P,A)}\ x \mid x \in X\}
\end{array}$$

Intuitively, constraint system $\mathcal{R}_f(M)$ works as follows. Assume we would like to determine the set of possible return values for procedure p on input d. Then first, we determine the sets $\langle A, d\rangle$ of all values arriving at all program points A of p. Clearly, value d arrives at the entry point of p – giving constraints of line (2). Accordingly, if (P, A) is an in-going edge of A, P an ordinary program point, then all values $\mathsf{Trans}_{(P,A)}\ d'$ arrive at A where d' arrives at P – giving constraints of line (3). Now assume A is the next program point after call C to some procedure p'. Then, for every element x arriving at C, we proceed as follows. We first determine the set of return values of p' for $\mathsf{Entry}_C\ x$; then we combine each element x_1 of this set with value x before the call, i.e., compute $\mathsf{Comb}_C\ (x_1, x)$ to obtain a new value to arrive at A – which gives us constraints of line (4). Having thus finally determined the set of values arriving at some return point R of p, we obtain return values for p by applying Exit_R to each of these values – giving line (1).

Example 4. Let us consider the pda from Example 1. Then constraint system $\mathcal{R}_f(M)$ is given by:

$$\begin{array}{lll}
p\ d \supseteq \mathsf{Exit}^*_R\ \langle R, d\rangle & \langle A, d\rangle \supseteq \mathsf{Trans}^*_{(I,A)}\ \langle I, d\rangle & \langle I, d\rangle \supseteq \mathsf{Trans}^*_{(p,I)}\ \langle p, d\rangle \\
\langle p, d\rangle \supseteq \{d\} & \langle C, d\rangle \supseteq \mathsf{Trans}^*_{(I,C)}\ \langle I, d\rangle & \langle R, d\rangle \supseteq \mathsf{Trans}^*_{(B,R)}\ \langle B, d\rangle \\
\multicolumn{3}{l}{\langle B, d\rangle \supseteq \mathsf{Trans}^*_{(A,B)}\ \langle A, d\rangle,\ \mathcal{E}\ (\lambda x.\mathsf{Comb}^*_C\ (p\ (\mathsf{Entry}_C\ x), x))\ \langle C, d\rangle}
\end{array}$$

Theorem 5. *For all $p \in \mathsf{Proc}$ and $A \in \mathsf{Item}_p$,*

1. $[\![\mathcal{R}_f(M)]\!]\ p\ d = \{d' \in D \mid (d, p) \vdash^* (d', \epsilon)\}$;
2. $[\![\mathcal{R}_f(M)]\!]\ \langle A, d\rangle = \{d' \in D \mid (d, p) \vdash^* (d', A)\}$. □

Since here the (abstract) meaning of procedures are *relations*, we call an analysis based on (partially) computing the least solution of $\mathcal{R}_f(M)$ *relational*. Theorem 5 then presents a first and general Interprocedural Coincidence Theorem stating that derivability for the *abstract* operational semantics M is precisely characterized by the least solution of constraint system $\mathcal{R}_f(M)$.

It should be emphasized that Theorem 5 contains no assumptions on the nature of D or the monotonicity/continuity of behavioral functions. We only rely on preservation of simulation relation Δ. Thus, an effective analysis engine is already obtained in case set D of abstract values is finite. For Prolog, a constraint system in the spirit of $\mathcal{R}_f(M)$ has been suggested for `OLDT`-resolution and practically evaluated by Van Hentenryck et al. in [18]. The precise relationship, however, to some operational semantics has not been formally clarified. New algorithms for relational analysis are also proposed in [17].

6.2 Functional Analysis

Clearly, for efficiency reasons we often would like to replace a precise but too large description by a less precise but (hopefully) computationally more tractable one. To study such kinds of (often implicitly applied) *widenings* [10], let us now additionally assume that D is a complete lattice, Δ is compatible and all behavioral functions of M are total and continuous. From $\mathcal{R}_f(M)$ we obtain constraint system $\mathcal{F}_f(M)$ in the same variables but over D if we replace all $\supseteq$ and all $\bigcup$ with $\sqsupseteq$ and $\bigsqcup$. Thus, $\mathcal{F}_f(M)$ is given by:

$$\begin{array}{lll}
p\ d & \sqsupseteq \mathsf{Exit}_R\ \langle R, d\rangle & \text{if } R \in \mathcal{R}_p \\
\langle p, d\rangle & \sqsupseteq d & \\
\langle A, d\rangle & \sqsupseteq \mathsf{Trans}_{(P,A)}\ \langle P, d\rangle & \text{if } (P, A) \in \mathsf{Edge} \\
\langle A, d\rangle & \sqsupseteq \mathsf{Comb}_C\ (p\ (\mathsf{Entry}_C\ \langle C, d\rangle), \langle C, d\rangle) & \text{if } p = \mathit{proc}\ C,\ A = \mathit{next}\ C
\end{array}$$

An analysis based on $\mathcal{F}_f(M)$ is called *forward functional*. All current interprocedural analyses (of imperative languages) performing functional analysis we know of [39, 26, 1] use *forward* accumulation.

If D is finite, the least solution of $\mathcal{F}_f(M)$ can be computed by chaotic fixpoint iteration (possibly with "needed information only") as suggested by Cousot and Cousot [11]. For imperative languages, (functional forward accumulating) derivability analysis based on worklist solvers has been proposed already by Sharir and Pnueli [39] and is used by Alt and Martin [1]. For logic languages, the application of *generic local* solvers has been advocated by Le Charlier and Van Hentenryck [6] and Fecht and Seidl [16]. In case D is infinite, approximation methods like those of Bourdoncle [3] may be applicable. If F is the lattice of possibly occurring transformers $D \to D$, $\mathcal{F}_f(M)$ can also be viewed as system of equations over F. In this case, "ordinary" fixpoint methods may be applied, see [26, 25] or [20–22] for instances of this idea for imperative languages. For logic languages, this approach has been suggested in [23, 28]. Prerequisite always is that every (occurring) function $f \in F$ is succinctly representable and that the necessary operations on F, especially composition, "$\sqcup$" and equality, are efficiently computable.

Example 6. Let us consider the pda from Example 1. Then constraint system $\mathcal{F}_f(M)$ is given by:

$$\begin{array}{lll}
p\ d \sqsupseteq \mathsf{Exit}_R\ \langle R, d\rangle & \langle A, d\rangle \sqsupseteq \mathsf{Trans}_{(I,A)}\ \langle I, d\rangle & \langle I, d\rangle \sqsupseteq \mathsf{Trans}_{(p,I)}\ \langle p, d\rangle \\
\langle p, d\rangle \sqsupseteq d & \langle C, d\rangle \sqsupseteq \mathsf{Trans}_{(I,C)}\ \langle I, d\rangle & \langle R, d\rangle \sqsupseteq \mathsf{Trans}_{(B,R)}\ \langle B, d\rangle \\
\multicolumn{3}{l}{\langle B, d\rangle \sqsupseteq \mathsf{Trans}_{(A,B)}\ \langle A, d\rangle,\ \mathsf{Comb}_C\ (p\ (\mathsf{Entry}_C\ \langle C, d\rangle), \langle C, d\rangle)}
\end{array}$$

Theorem 7. *For all $p \in \mathsf{Proc}$, $A \in \mathsf{Item}_p$ and $d \in D$,*

1. $\bigsqcup\ (\llbracket\mathcal{R}_f(M)\rrbracket\ p\ d) \sqsubseteq \llbracket\mathcal{F}_f(M)\rrbracket\ p\ d$;
2. $\bigsqcup\ (\llbracket\mathcal{R}_f(M)\rrbracket\ \langle A, d\rangle) \sqsubseteq \llbracket\mathcal{F}_f(M)\rrbracket\ \langle A, d\rangle$. □

In light of Theorem 5, we find that the left-hand sides here represent what has been called "interprocedural merge over all path" solution (interprocedural MOP) to the analysis problem [39, 26, 20, 21]. Thus, Theorem 7 guarantees safety of functional analysis (with forward accumulation) relative to this interprocedural MOP resp. relative to relational analysis – thus implying overall safety of functional derivability analysis.

7 Constraint Systems with Backward Accumulation

Surprisingly enough, there is an alternative approach to program analysis which, although (to a certain extent) more natural, has not attracted much attention so far, namely, *backward* accumulation. Here, derivability information is not just computed for procedures but for *every* item of pda M. We explicate this second approach analogously to section 6. In subsequent section 8 we then relate these two approaches w.r.t. precision.

7.1 Relational Analysis

As for forward accumulation, we start out from abstract pda M over set D of abstract values. According to the strategy of backward accumulation, constraint system $\mathcal{R}_b(M)$ now introduces just one type of set-valued variables, namely $A\ d, A \in \mathsf{Item}$, for derivability of every item. $\mathcal{R}_b(M)$ is then defined by:

$$\begin{array}{lll} R\ d \supseteq \{\mathsf{Exit}_R\ d\} & \text{if } R \in \mathsf{Return} & (1) \\ P\ d \supseteq A\ (\mathsf{Trans}_{(P,A)}\ d) & \text{if } (P,A) \in \mathsf{Edge} & (2) \\ C\ d \supseteq \mathcal{E}\ A\ (\mathsf{Comb}^*_C\ (p\ (\mathsf{Entry}_C\ d), d)) & \text{if } p = \mathit{proc}\ C,\ A = \mathit{next}\ C & (3) \end{array}$$

In a certain sense, (the least solution of) constraint system $\mathcal{R}_b(M)$ can be viewed as the *big-step* operational semantics corresponding to pda M. The intuition behind constraint system $\mathcal{R}_b(M)$ is as follows. Assume we would like to determine for some program point and some value d arriving at A the set of all values to be returned at intra-procedurally reachable return points. If the program point in question itself equals a return point R, this set clearly contains $\mathsf{Exit}_R\ d$ – giving constraints from line (1). If it equals an ordinary program point P (no call) and (P, A) is an out-going edge of P, then all values are returned for P on input d which are returned for A on input $\mathsf{Trans}_{(P,A)}\ d$ – giving constraints from line (2). Finally consider call C of procedure p where the next program point after C is A. Then all values are returned for C which are returned for A on values arriving at A after call C on input d. The latter set, however, consists of all values $\mathsf{Comb}_C\ (d_1, d)$ where d_1 is one of the values returned by procedure p on input $\mathsf{Entry}_C\ d$. Overall, this gives the constraints from line (3).

Example 8. Let us consider the pda from Example 1. Then constraint system $\mathcal{R}_b(M)$ is given by:

$$\begin{array}{lll} p\ d \supseteq I\ (\mathsf{Trans}_{(p,I)}\ d) & A\ d \supseteq B\ (\mathsf{Trans}_{(A,B)}\ d) & I\ d \supseteq A\ (\mathsf{Trans}_{(I,A)}\ d),\ C\ (\mathsf{Trans}_{(I,C)}\ d) \\ R\ d \supseteq \{\mathsf{Exit}_R\ d\} & B\ d \supseteq R\ (\mathsf{Trans}_{(B,R)}\ d) & C\ d \supseteq \mathcal{E}\ B\ (\mathsf{Comb}^*_C\ (p\ (\mathsf{Entry}_C\ d), d)) \end{array}$$

Theorem 9. *For all* $A \in \mathsf{Item}$, $[\![\mathcal{R}_b(M)]\!]\ A\ d = \{d' \in D \mid (d, A) \vdash^* (d', \epsilon)\}$. □

Here, the (abstract) meaning of procedures are *relations* represented by mappings $D \to 2^D$. Therefore, we call an analysis based on (partially) computing the least solution of *backward relational*. Theorem 9 then presents a second general Interprocedural Coincidence Theorem stating that derivability for the *abstract* operational semantics M is also precisely characterized by the least solution of constraint system $\mathcal{R}_b(M)$.

Again, Theorem 9 contains no assumptions on the nature of D or the monotonicity/continuity of behavioral functions. We only rely on preservation of simulation relation Δ. Thus, an effective analysis engine is already obtained in case set D of abstract values is finite. For Prolog, constraint system $\mathcal{R}_b(M)$ has been proposed as an optimization to constraint system $\mathcal{R}_f(M)$ and practically evaluated on some programs by Van Hentenryck et al. in [18]. As in the case of forward accumulation, its relation to some kind of operational semantics has not yet been clarified.

7.2 Functional Analysis

As for forward accumulation, let us now study the impact of widenings. Therefore, we now additionally assume that D is a complete lattice, Δ is compatible and all behavioral functions of M are total and continuous. From $\mathcal{R}_b(M)$ we obtain constraint system $\mathcal{F}_b(M)$ in the same variables but over D if we replace all $\supseteq$ and all $\bigcup$ with $\sqsupseteq$ and $\bigsqcup$. Thus, $\mathcal{F}_b(M)$ is given by:

$$\begin{array}{ll} R\ d \sqsupseteq \mathsf{Exit}_R\ d & \text{if } R \in \mathsf{Return} \\ P\ d \sqsupseteq A\ (\mathsf{Trans}_{(P,A)}\ d) & \text{if } (P,A) \in \mathsf{Edge} \\ C\ d \sqsupseteq A\ (\mathsf{Comb}_C\ (p\ (\mathsf{Entry}_C\ d), d)) & \text{if } p = \mathit{proc}\ C,\ A = \mathit{next}\ C \end{array}$$

Let us call an analysis based on $\mathcal{F}_f(M)$ *backward functional.*

Example 10. Let us consider the pda from Example 1. Then constraint system $\mathcal{F}_b(M)$ is given by:

$$\begin{array}{lll} p\ d \sqsupseteq I\ (\mathsf{Trans}_{(p,I)}\ d) & A\ d \sqsupseteq B\ (\mathsf{Trans}_{(A,B)}\ d) & I\ d \sqsupseteq A\ (\mathsf{Trans}_{(I,A)}\ d)\ ,\ C\ (\mathsf{Trans}_{(I,C)}\ d) \\ R\ d \sqsupseteq \mathsf{Exit}_R\ d & B\ d \sqsupseteq R\ (\mathsf{Trans}_{(B,R)}\ d) & C\ d \sqsupseteq B\ (\mathsf{Comb}_C\ (p\ (\mathsf{Entry}_C\ d), d)) \end{array}$$

Theorem 11. *For all* $A \in \mathsf{Item}$ *and* $d \in D$, $\bigsqcup\ (\llbracket \mathcal{R}_b(M) \rrbracket\ A\ d) \sqsubseteq \llbracket \mathcal{F}_b(M) \rrbracket\ A\ d.$ □

Thus, Theorem 11 implies overall safety of backward functional analysis. To the best of our knowledge, functional analysis with backward accumulation has not attracted attention so far for program analysis.

8 Comparison of Forward and Backward Accumulation

In this section, we compare the constraint systems with forward and backward accumulation w.r.t. precision. From Theorems 5 and 9, we immediately obtain:

Theorem 12. *For every* $p \in \mathsf{Proc}$ *and every* $A \in \mathsf{Item}_p$, $\llbracket \mathcal{R}_f(M) \rrbracket\ p = \llbracket \mathcal{R}_b(M) \rrbracket\ p.$ □

While equivalence holds in the relational case, this holds no longer true for functional analysis. Constraint systems $\mathcal{F}_f(M)$ as well as $\mathcal{F}_b(M)$ use transformers from $D \to D$ to represent effects of procedures. They differ, however, dramatically in the way how *intra-procedurally* these effects are accumulated. Consider one call to procedure p for actual parameter d. Functional analysis with forward accumulation abstracts the set of all values intra-procedurally arriving at some program point by just one abstract value. On the contrary, functional analysis with backward accumulation keeps all these values distinct. Computing least upper bounds is delayed to the very end.

Theorem 13. *For every* $p \in \mathsf{Proc}$ *and every* $A \in \mathsf{Item}_p$, $\llbracket \mathcal{F}_b(M) \rrbracket\ p\ d \sqsubseteq \llbracket \mathcal{F}_f(M) \rrbracket\ p\ d.$ □

To show that "$\sqsubseteq$" in Theorem 13 cannot generally be replaced with "$=$", we give an example where forward accumulation loses information against backward accumulation. The example is *not* intended to represent an interesting program analysis but to exhibit a *minimal* situation where this loss in precision can be observed.

Example 14. Let us compare the constraint systems from Examples 6 and 10. Consider complete lattice $D = \{\bot < 1, 2 < \top\}$, and let us assume that $\llbracket a \rrbracket = \mathsf{Trans}_{(A,B)} = \mathit{Id}$ and $\llbracket b \rrbracket = \mathsf{Trans}_{(B,R)} = f$ where $f\ x = 2$ if $x \in \{1,2\}$ and $f\ x = x$ otherwise.

To simplify calculations, we let $\mathsf{Exit}_R = \mathsf{Entry}_C = \mathsf{Trans}_{(p,I)} = \mathsf{Trans}_{(I,A)} = \mathsf{Trans}_{(I,C)} = \mathit{Id}$ and $\mathsf{Comb}_C\ (d_1, d) = d_1$. Then $\llbracket \mathcal{F}_b(M) \rrbracket\ p\ 1 = 2 \sqsubset \top = \llbracket \mathcal{F}_f(M) \rrbracket\ p\ 1.$ □

There is, however, a graph-theoretical restriction on intra-procedural CFGs enforcing equivalence between forward and backward accumulation. Let us call intra-procedural CFG $\mathcal{G}_p$ for p *simple* if it is a tree with root p, and otherwise *complex*. Clearly, normalized Prolog programs as defined in Section 4 introduce simple intra-procedural CFGs only.

Theorem 15. *If all intra-procedural CFGs are simple, then for every* $p \in \mathsf{Proc}$ *and every* $A \in \mathsf{Item}_p$, $\llbracket \mathcal{F}_f(M) \rrbracket\ p\ d = \llbracket \mathcal{F}_b(M) \rrbracket\ p\ d.$ □

Thus, loss in precision may only occur for complex intraprocedural control-flow graphs, as are usual for imperative programs. It cannot be observed for *normalized* Prolog programs as introduced in Section 4 but very well for "real" Prolog programs which contain complex goals as in Example 1.

9 Conclusion

We have developed a general framework for the analysis of procedural languages based on a small-step operational semantics with pda's. We then explored four extreme points in the design space of constraint systems derivable from (abstract) pda's. We started with two forms of relational analysis which both lose no information against the abstract operational semantics but differ in their treatment of procedure bodies. From these two, we obtained constraint systems for functional analysis by abstracting sets of values with their respective upper bounds. Clearly, a more graceful degradation in precision is here possible as well. Only some of the reported constraint systems have been considered so far. Especially, functional analysis with *backward* accumulation seems to be new.

The present paper clearly focussed onto the similarities behind the analysis of imperative and logic languages. We have demonstrated that (at least in principle) in both areas the same ideas are applicable. On the contrary, we feel that there are clear *pragmatic* differences. Algorithms which have been found efficient for the analysis of Prolog need not necessarily yield the best results also for imperative languages and vice versa. A sorrow comparison of the two language classes in this respect remains for future work.

Acknowledgement: We thank Reinhold Heckmann for semantic advice and Martin Alt, Jens Knoop, Florian Martin, Tom Reps and Reinhard Wilhelm for valuable discussions about the mysteries of interprocedural analysis of imperative languages.

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. Francois Bourdoncle. Interprocedural Abstract Interpretation of Block-Structured Languages with Nested Procedures, Aliasing and Recursivity. In *International Workshop on Programming Language Implementation and Logic Programming (PLILP)*, pages 307–323. LNCS 456, 1990.
3. Francois Bourdoncle. Abstract Interpretation by Dynamic Partioning. *Journal of Functional Programming*, 2(4):407–435, 1992.
4. Francois Bourdoncle. *Sémantiques des Langages Impératifs d'Ordre Supérieur et Interprétation Abstraite*. PhD thesis, École Polytechnique, Paris, 1992.
5. Maurice Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10(1/2/3/4):91–124, 1991.
6. Baudouin Le Charlier and Pascal Van Hentenryck. A Universal Top-Down Fixpoint Algorithm. Technical Report CS-92-25, Brown University, Providence, RI 02912, 1992.
7. 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.
8. Noam Chomsky. Context-free Grammars and Pushdown Storage. *Quart. Prog. Dept.*, 65:187–194, 1962.
9. Patrick Cousot. Semantic Foundations of Program Analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.
10. Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of 4th ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.
11. Patrick Cousot and Radhia Cousot. Static Determination of Dynamic Properties of Recursive Programs. In E.J. Neuhold, editor, *Formal Descriptions of Programming Concepts*, pages 237–277. North-Holland Publishing Company, 1977.
12. 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.
13. J. Evey. *The Theory and Application of Pushdown Store Machines.* PhD thesis, Harvard University, Cambridge, Mass., 1963.
14. 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.

15. Christian Fecht. *Abstrakte Interpretation logischer Programme: Theorie, Implementierung, Generierung.* PhD thesis, Universität des Saarlandes, Saarbrücken, 1997.
16. 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.
17. Christian Fecht and Helmut Seidl. Propagating Differences: A New Efficient Fixpoint Algorithm for Distributive Constraint Systems. Technical Report 97-13, Universität Trier, 1997.
18. 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.
19. 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.
20. Susan Horwitz, Thomas W. Reps, and Mooly Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of 22nd ACM Symposium on Principles of Programming Languages (POPL)*, pages 49–61, 1995.
21. 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.
22. Susan Horwitz, Thomas W. Reps, and Mooly Sagiv. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theoretical Computer Science*, 167(1&2):131–170, 1996.
23. Dean Jacobs and Anno Langen. Static Analysis of Logic Programs for Independent AND Parallelism. *Journal Logic Programming*, 13:291–314, 1992.
24. Neil D. Jones and Steven S. Muchnick. A Flexible Approach to Interprocedural Data Flow Analysis and Programs with Recursive Data Structures. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages (POPL)*, pages 66–74, 1982.
25. Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Towards a Tool Kit for the Automatic Generation of Interprocedural Data Flow Analyses. *Journal of Programming Languages*, 4:211–246, 1996.
26. Jens Knoop and Bernhard Steffen. The Interprocedural Coincidence Theorem. In *4th International Conference on Compiler Construction*, pages 125–140. LNCS 641, 1992.
27. P. Landin. The Mechanical Evaluation of Expressions. *Computer Journal*, 6(4):158–165, 1963.
28. Kim Marriott and Harald Søndergaard. Precise and Efficient Groundness Analysis for Logic Programs. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 2:181–196, 1993.
29. Kim Marriott, Harald Søndergaard, and Neil D. Jones. Denotational Abstract Interpretation of Logic Programs. *ACM Transactions of Programming Languages and Systems (TOPLAS)*, 16(3):607–648, 1994.
30. John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine. *CACM*, 3(4):184–195, 1960.
31. Ulf Nilsson. Towards a Framework for the Abstract Interpretation of Logic Programs. In *International Workshop on Programming Language Implementation and Logic Programming (PLILP)*, pages 68–82. LNCS 348, 1988.
32. Ulf Nilsson. *A Systematic Approach to Abstract Interpretation of Logic Programs.* PhD thesis, Department of Computer Science and Information Science, Linköping University, Sweden, 1989.
33. Ulf Nilsson. Systematic Semantic Approximations of Logic Programs. In *International Workshop on Programming Language Implementation and Logic Programming (PLILP)*, pages 293–306. LNCS 456, 1990.
34. W.L. Van Der Poel. Dead Programmes for a Magnetic Drum Automatic Computer. *Applied Scientific Research (B)*, 3:190–198, 1953.
35. B. Randell and L.J. Russell. *Algol 60 Implementation.* Academic Press, New York, 1964.
36. David A. Schmidt. Abstract Interpretation of Small-Step Semantics. In Mads Dam, editor, *Proceedings of the 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages.* LNCS 1192, June 1996.
37. Marcel Paul Schützenberger. On Context-free Languages and Pushdown Automata. *Information and Control*, 6:246–264, 1963.
38. Helmut Seidl and Christian Fecht. Interprocedural Analysis Based on PDAs. Technical Report 97-06, Universität Trier, 1997.
39. Micha Sharir and Amir Pnueli. Two Approaches to Interprocedural Data Flow Analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.