ELSEVIER The Journal of Logic Programming 43 (2000) 123–156

THE JOURNAL OF LOGIC PROGRAMMING

www.elsevier.com/locate/jlpr

Interprocedural analyses: a comparison

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

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

Received 18 September 1997; received in revised form 30 April 1999; accepted 14 June 1999

Abstract

We present a framework for program analysis of languages with procedures which is general enough to allow for a comparison of various approaches to interprocedural analysis. Our framework is based on a small-step operational semantics and subsumes both frameworks for imperative and for logic languages. We consider *reachability analysis*, that is, the problem of approximating the sets of program states reaching program points. We use our framework in order to clarify the impact of several independent design decisions on the precision of the analysis. Thus, we compare intraprocedural *forward accumulation* with intraprocedural *backward accumulation*. Furthermore, we consider both *relational* and *functional* approaches. While for relational analysis the accumulation strategy makes no difference in precision, we prove for functional analysis that forward accumulation may lose precision against backward accumulation. Concerning the relative precision of relational analyses and corresponding functional analyses, we exhibit scenarios where functional analysis does not lose precision. Finally, we explain why even an enhancement of functional analysis through disjunctive completion of the underlying abstract domain may sometimes lose precision against relational analysis. © 2000 Elsevier Science Inc. All rights reserved.

Keywords: Interprocedural analysis; Pushdown automata; OLDT-resolution; Coincidence theorems

1. Introduction

Static analysis aims at computing statements about the runtime behavior of a program without actually executing the program. Program analysis can be used to guarantee validity of semantic preconditions for optimizing program transformations. Since this type of analysis is to be included into real application systems, namely compilers or program development environments, it should be both fully automatic and sufficiently efficient. No user is willing to wait for hours (or even for minutes) to

* Corresponding author. Tel.: +49-651-201-2835; fax: +49-651-201-3822.
E-mail address: seidl@psi.uni-trier.de (H. Seidl).

receive the answer to her query or get her program compiled. In general, however, the answer for every non-trivial question about programs is uncomputable. Therefore, the "art" of program analysis consists in finding acceptable compromises: providing sufficiently "precise" answers in a reasonable amount of time. This paper studies some design choices together with their impacts on precision in the area of analysis of imperative procedural languages as well as logic languages like Prolog.

The main analysis question we are interested in asks for the set of values *reaching* certain program points when the program is executed. Such an analysis is also called *reachability analysis*. Since the programming languages we are interested in provide the concept of *procedures*, it turns out to be useful as an auxiliary question to ask for the *effects* of procedures. Let us call this type of analysis *effect analysis*. Determining the effects of procedures can be based on an abstraction of the *denotational semantics* of the program as in Refs. [13,32,41,8,9]. Execution of programs, however, and reachability of program points with states are essentially *operational* notions. As an illustration, consider the following (admittedly contrived) program:

$$s \leftarrow p(a),\ s \qquad\qquad p(b) \leftarrow \qquad\qquad p(X) \leftarrow$$

The set of (finite) SLD evaluation trees for s is empty, since evaluation of s never terminates. Nonetheless during this non-terminating computation, predicate p is iteratively called with input substitution $\{X \mapsto a\}$. A meaningful reachability analysis therefore is obliged to report this fact (or its abstractions). It is exactly for this kind of situation where a *small-step* operational semantics has been favored [11,46]. The small-step operational semantics of recursion is most naturally modeled by *pushdown automata* (PDAs). Related concepts have successfully been used for imperative languages [49,33,2,4] as well as for Prolog [39,16,17].

In order to derive an analysis engine, this framework proceeds in three stages. In the first stage, the concrete operational semantics is *simulated* by an abstract operational semantics [46]. Ideally, while preserving the control structure, we would only "abstract" data and the operations on data. Then, our analyzer simply needs to execute this abstract version of the program. Abstraction of data, however, introduces loss of information implying that evaluation of conditions may no longer allow us to determine precisely the branches taken at runtime. A safe choice at hand therefore is to abstract conditional branching by non-determinism. The somewhat surprising consequence is that, despite the apparent differences in the concrete semantics, the abstract operational semantics of deterministic procedural languages like C and the non-deterministic procedural language Prolog are (conceptually) identical.

This first derivation step of the framework has transformed analysis problems for the concrete semantics into corresponding analysis problems for the abstract operational semantics. In the second stage, a system of (in-)equations is selected which (more or less precisely) characterizes the property to be analyzed. In the last stage, a standard solver is applied to compute a (least) solution of the system of (in-)equations.

Such solvers have systematically been studied in Refs. [6,20,8,22,23]. Therefore, we concentrate here on the second stage, that is, the design decisions for an analysis once the abstract operational semantics of a program is fixed. The main contributions of our paper are:

- By giving a unified framework applicable to imperative as well as logic languages we hope to bridge the gap between two communities that usually do not know much about each other.
- By basing our framework on a small-step operational semantics, we both derive and prove correct not only effect analysis but also reachability analysis – a topic which has been often treated quite carelessly.
- Besides presenting known analyses we also succeed in deriving completely new ones.

The main motivation, however, for our unified framework is to enable us to elegantly compare different approaches w.r.t. their relative precisions. Thus:

- We compare two approaches to reachability analysis: both start from a system of inequations for effect analysis; while the first one extends the system of inequations by adding further variables and inequations, the second one leaves the system unchanged and applies *local* fixpoint computation instead.
- We clarify the impact of the design decision of using "backward accumulation" instead of usual "forward accumulation".
- We derive scenarios where computing with sets ("relational analysis") gains nothing over computing with individual abstract values ("functional analysis"); and
- We explain why enhancing a functional analysis by applying disjunctive completion to the abstract domain as suggested in Ref. [24] may still lose precision against relational analysis.

The overall structure of our paper is as follows. Section 2 introduces interprocedural control-flow graphs which are used as a meta-language for program analysis. As an example, Section 3 presents a translation of (a pure subset of) Prolog into this meta-language. Section 4 introduces our notion of pushdown automata and shows how they can be used as a formal model for the operational semantics. Section 5 introduces the notion of "simulation" between PDAs. Sections 7 and 8 consider systems of inequations for effect analysis. Section 7 systematically derives systems of inequations with forward accumulation, whereas Section 8 systematically derives systems of inequations with backward accumulation. In particular, 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 9 is devoted to reachability analysis. We present two ideas: first, reachability through application of local solvers, and second, reachability through extension of systems of inequations for effect analysis. We determine that, in general, reachability analysis based on local solvers is more precise than reachability based on extensions of systems of inequations for functional effect analysis. Section 10 compares the systems of inequations with forward accumulation and those using backward accumulation w.r.t. precision. In Section 11 we study scenarios where relational analysis essentially computes no more information than functional analysis. To this end we critically review a scenario already suggested for imperative languages, as well as present new scenarios where coincidence can be proved. One of these scenarios turns out to be especially well-tailored for the analysis of Prolog programs. Finally, the comparison of relational analysis with functional analysis, enhanced with disjunctive completion is included in Section 12.

This work has been presented as a tutorial at ETAPS'98. A condensed version of Sections 2–10 is presented in Ref. [48]; a short abstract of Section 12 occurs as Ref. [47].

2. The meta-language

In this paper, we consider programming languages where programs can be separated into a specification of *control-flow* and a specification of *data-flow*. Control-flow defines a set of program points together with those sequences of program points which are (formally) possible program executions. In the presence of procedures, control-flow is most conveniently described through *interprocedural control-flow graphs* (interprocedural CFGs). Given a specification of control-flow, data-flow indicates what kind of operations are executed during program execution when passing from one program point to another. Here, such a specification is called *behavior*.

Formally, an interprocedural control-flow graph consists of a finite set Proc of procedures together with a collection $\mathcal{G}_p, p \in \mathsf{Proc}$, of disjoint *intraprocedural* control-flow graphs. We assume that there is one special procedure main with which program execution starts. The intraprocedural control-flow graph $\mathcal{G}_p$ of a procedure p consists of the following components:

- a set N_p of *program points*;
- a special *entry point* $s_p \in N_p$; for simplicity, we denote s_p by p itself;
- a set of *exit points* $R_p \subseteq N_p$;
- a set of edges $E_p \subseteq N_p \times N_p$;
- a subset $C_p \subseteq E_p$ of *call edges* where for each $e \in C_p$, call e denotes the procedure called at this edge.

Let Point and Return denote the sets of all program points and all exit points, respectively. Let Edge, Call and Basic denote the set of all edges, the set of call edges, and the set of remaining edges, respectively. Edges from Basic are also called *basic computation steps*. Let D denote the set of possible data values or *program states*. Then computation on data can succinctly be represented through the (possibly partial) functions

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

The functions Entry and Exit specify how data are passed to the called procedure and returned from it. The function Trans specifies the *transfer functions* for basic computation steps whereas the function Comb corresponds to *combine* in Ref. [33] or $\mathcal{R}$ in Refs. [37,36]. It combines the result returned by the called procedure with the value before the call yielding the value after the call. This binary function allows us to model local variables of procedures conveniently. Let us call the collection of the functions Entry, Exit, Trans and Comb *behavior*.

This program representation has been used for instance by Jones and Muchnick for imperative programs consisting of a finite set of non-nested procedure definitions where each procedure may have local variables and use value as well as result parameters (as in AlgolW) [33]. A similar class of programs is dealt with in Refs. [13,37]. Bourdoncle extends this approach to deal with nested procedure declarations, global variables and reference parameters as in Pascal [2]. Extensions including procedure parameters are considered in Ref. [4]. First-order functional languages (with tail call optimization) are considered by Debray and Proebsting in Ref. [18]. One further instance of this meta-language is *normalized Prolog* [8,20].

3. Prolog

A normalized (pure) Prolog program consists of a set Pred of *predicates*, a main predicate $\mathsf{main} \in \mathsf{Pred}$ together with a set Clause of *clauses*. Each clause is of the form $p(X_1, \ldots, X_k) \leftarrow \alpha$ where p is a predicate, and α consists of a sequence of *goals*. A goal in α either is a *literal* like $q(X_{i_1}, \ldots, X_{i_m})$ (with predicate q and pairwise distinct variables X_{i_j}) from Lit or a basic goal from $\mathsf{Builtin}$ like $X_i = t$ or **true**. Thus, $\alpha \in (\mathsf{Lit} \cup \mathsf{Builtin})^*$.

In order to extract control- and data-flow from such programs, we view each predicate p as a procedure whose procedure body is given by the clauses for p. Thus, the set of program points consists of all predicates p together with all items $u = [h \leftarrow \alpha.\beta]$, $\alpha, \beta \in (\mathsf{Lit} \cup \mathsf{Builtin})^*$, where $h \leftarrow \alpha\beta \in \mathsf{Clause}$. The item u is an exit point if $\beta \equiv \epsilon$ (the empty sequence). The set of edges is given by all pairs $(p, [h \leftarrow .\alpha])$ where $h \leftarrow \alpha$ is a clause for p, together with all pairs $e = ([h \leftarrow \alpha.g\beta], [h \leftarrow \alpha g.\beta])$. If g is a goal of the form $q(X_{j_1}, \ldots, X_{j_m})$, then e is a call edge which calls procedure/predicate q, that is, $q = \mathsf{call}\, e$. All other edges are basic computation steps.

Given a set D of possible program states, the semantics of normalized Prolog programs is specified through the following functions [8,20]:

$[\![.]\!]$	:	$\mathsf{Builtin} \to D \to D$	*meaning of basic goals*
restrG	:	$\mathsf{Lit} \to D \to D$	*restriction to locals in literal*
ci	:	$\mathsf{Lit} \to D \to D$	*parameter passing into predicate*
extC	:	$\mathsf{Clause} \to D \to D$	*initialization of selected clause*
restrC	:	$\mathsf{Clause} \to D \to D$	*restriction to head variables*
co	:	$\mathsf{Lit} \to D \to D$	*parameter passing back into literal*
extG	:	$\mathsf{Lit} \to D \times D \to D$	*effect of call*

The function $[\![.]\!]$ assigns a meaning to basic goals, restrG restricts the current state to the variables occurring in the corresponding goal; extC extends the current state to take all clause variables into account; restrC restricts the current state to the variables occurring in the head of the clause; extG describes how the state after the call is determined from the state returned by the predicate and the state before the call; finally, ci and co model the effect of variable renaming. Given these functions, the behavior can be obtained by:

$$\begin{array}{lll} \mathsf{Trans}\ e\ d & = [\![b]\!]\ d & \text{if } e = ([h \leftarrow \alpha.b\beta], [h \leftarrow \alpha b.\beta]),\ b \in \mathsf{Builtin} \\ \mathsf{Trans}\ e\ d & = \mathsf{extC}\ (h \leftarrow \alpha)\ d & \text{if } e = (p, [h \leftarrow .\alpha]),\ h \equiv p(\ldots) \\ \mathsf{Exit}\ r\ d & = \mathsf{restrC}\ (h \leftarrow \alpha)\ d & \text{if } r \equiv [h \leftarrow \alpha.] \end{array}$$

Furthermore for $e = ([h \leftarrow \alpha.g\beta], [h \leftarrow \alpha g.\beta])$ with $g \equiv p(\ldots) \in \mathsf{Lit}$,

$$\begin{array}{ll} \mathsf{Entry}\ e\ d & = \mathsf{ci}\ g\ (\mathsf{restrG}\ g\ d) \\ \mathsf{Comb}\ e\ (d_1, d) & = \mathsf{extG}\ g\ (\mathsf{co}\ g\ d_1, d) \end{array}$$

Example 1. Consider the program

$$\mathsf{main} \leftarrow X = a,\ p(X), \mathsf{main} \qquad p(X) \leftarrow X = b \qquad p(X) \leftarrow$$

as presented in Section 1. In order to meet our syntactical restrictions, we renamed the main procedure s as main and made the unifications $X = a$ and $X = b$ explicit.

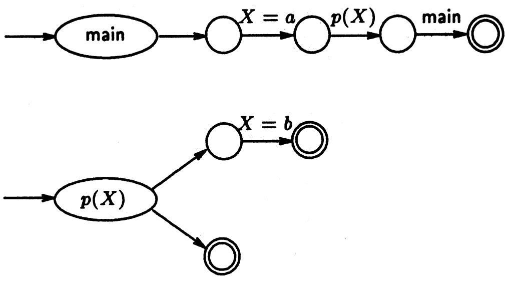

Fig. 1. The interprocedural CFG for Example 1.

The corresponding (interprocedural) control-flow graph is given in Fig. 1. There, we also attached the corresponding goals of edges as labels. This is convenient since, according to our generic semantics of Prolog, the behavioral functions for edges can directly be generated from these labels. Also, we listed the formal parameters of procedures inside the entry nodes. We should have added the sets of local variables to the corresponding declaration edges as well. The latter, however, has been omitted for better readability of the diagrams. Similar decorations of control-flow graphs with pieces of source code are useful for imperative languages as well.

In order to reduce the number of arguments, we will write in the following the first arguments to behavioral functions (that is, arguments of Return or Edge) as subscripts. Thus, we will write for instance $\mathsf{Exit}_r\, d$ instead of $\mathsf{Exit}\; r\; d$ and $\mathsf{Comb}_e\; (d_1, d_2)$ instead of $\mathsf{Comb}\; e\; (d_1, d_2)$.

4. The operational semantics: pushdown automata

For the (small-step) operational semantics of a program we rely on *pushdown automata* (PDAs for short) [49,33,39,16,37]. There are several ways how such pushdown automata can be defined. Lang [39] for instance follows the tradition in parsing theory. He views intermediate configurations as sequences of stack symbols and the set of transitions as *left-rewrite system*. For a recent presentation, see also Ref. [17]. Instead, we insist on an explicit distinction between the current program state and stack symbols. In our presentation, stack symbols either represent the current program point (in which case they occur on top of the stack) or stacked activation records. Stacked activation records correspond to called, but not yet completed procedures. This distinction allows us to view procedures as transformers on *states* – a fact onto which many existing (efficient) analysis algorithms are based.

Therefore, we define a *pushdown automaton* as a triple $M = (D, \Gamma, \vdash)$ where Γ is the set of *stack symbols*, D the set of *states* (or *values*) and $\vdash$ is the *transition relation*. The set of *configurations* of M equals the set $D \times \Gamma^*$. The transition relation consists of the following three types of transitions:

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

where $d_1, d_2 \in D$ and $A, B, B_1, B_2 \in \Gamma$. While shifts correspond to intraprocedural computation steps, pushes and pops are used to model calls to and returns from procedures. Note that pushes not only add new elements to the stack but also modify the former topmost symbol. Indeed, this feature is essential for our operational semantics. The 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 skip the index M at $\vdash_M$.

Consider a program, represented through an interprocedural CFG together with a behavior as in Section 2. The PDA which implements the operational semantics uses $\Gamma = \mathsf{Point} \cup (D \times \mathsf{Call})$ as set of stack symbols. The set of transitions obtained from the behavior is defined by:

$$\begin{array}{lll}
\textit{Push}: & (d,u) \vdash (d', p\ [d,e]) & \text{if } e = (u,\_) \in \mathsf{Call}, p = \mathsf{call}_e,\ d' = \mathsf{Entry}_e\ d \\
\textit{Pop}: & (d,r) \vdash (d', \epsilon) & \text{if } r \in \mathsf{Return} \text{ and } d' = \mathsf{Exit}_r\ d \\
\textit{Shift}: & (d,u) \vdash (d', v) & \text{if } e = (u,v) \in \mathsf{Basic} \text{ and } d' = \mathsf{Trans}_e\ d \\
\textit{Unpack}: & (d_1, [d_2,e]) \vdash (d', v) & \text{if } e = (\_,v) \in \mathsf{Call} \text{ and } d' = \mathsf{Comb}_e\ (d_1,d_2)
\end{array}$$

Observe that we split the set of shift transitions into one set involving program points and one involving unpacking of stacked frames $[d_2, e]$. Each pop is immediately followed by an unpack step. Separating the return from procedures into two steps conveniently allows us to distinguish between the effects of procedures and the effects of calls. All configurations reachable from some initial configuration (d_0, main) are of the form $(d, A\ [d_1, e_1] \ldots [d_m, e_m])$ where $d, d_1, \ldots, d_m \in D$, $e_1, \ldots, e_m \in \mathsf{Call}$ and $A = [d', e'] \in D \times \mathsf{Call}$ or $A = u \in \mathsf{Point}$. In the latter case, u corresponds to the program point in the currently active procedure, whereas each $[d_j, e_j]$ represents the stacked frame for a call whose evaluation has been started but not yet completed.

4.1. Limitations and extensions

Our approach to the construction of a small-step operational semantics is no longer applicable if more complicated forms of control-flow must be considered. Therefore, it fails for back-tracking in the presence of side effects on some global state, for instance through arbitrary assert and retract (see Ref. [9] or Ref. [38] for a generalization in this direction). It also fails in the presence of multiple threads which communicate through channels or shared variables.

Also, it is not appropriate if the procedure called at a call edge cannot be statically determined. A simple example is given by normalized Prolog programs as in Section 3 extended with *computed goals*, that is, goals which themselves can be variables. Other instances of this scheme are procedure parameters as in Pascal [3], higher-order functional languages [19] or dynamic method dispatches as in Java. Such dynamically determined procedure calls, however, can be dealt with through a minor extension of the PDA approach. We simply allow our PDA to select the procedures to be called depending on the current state. Technically, we enhance the functionality of our function call:

$$\mathsf{call} : \mathsf{Call} \to D \to 2^{\mathsf{Proc}}$$

Observe that we allow several procedures to be called in a certain state (or even none). The *push* transitions for an edge $e = (u, \_) \in \mathsf{Call}$ then look like:

Push : $(d,u) \vdash (d',p\ [d,e])$ if $p \in \mathsf{call}_e\ d$ and $d' = \mathsf{Entry}_e\ d$

If the set of called procedures depends on the state, we feel free to reckon call among the behavioral functions as well.

4.2. Questions about program behavior

The most general questions to be asked about program behavior refer to properties of execution *paths*. A well-known analysis of that type for imperative programs is to determine whether along every execution path reaching a program point the value of a certain expression has been computed and still is available. Here, however, we only consider analyses referring to properties of program states at *program points*. More specifically, we are interested in the following two questions:

Effect: Given a program point v and a value d, to determine the set of all d' such that $(d,v) \vdash^* (d',\epsilon)$;
Reachability: Given an initial state d_0, to determine for every program point v the set of all states d such that $(d_0,\mathsf{main}) \vdash^* (d,v\pi)$ for some $\pi \in \Gamma^*$.

The first question is concerned with the functional behavior of program points, notably procedures. The second question is concerned with the sets of values arriving at program points. If for a certain procedure p and $d \in D$, we are only interested in determining for every program point v of p the set of values d' such that $(d,p) \vdash^* (d',v)$, then we also speak of *same-level reachability analysis*. Given the effects of procedures, same-level reachability should be thought of as the PDA formulation of *intraprocedural* reachability.

The restriction to analyses of states at program points is not as limiting as it may seem. Many path problems (in particular, the problem of availability of expressions) can be attacked by our approach through the construction of a suitable *instrumented* operational semantics where information about execution paths is explicitly recorded in the current program state. In one extreme case, we could store the complete execution history.

5. From concrete PDAs to abstract PDAs

In general, reasoning about a PDA M for the concrete operational semantics is not tractable. Therefore, we prefer to reason about some PDA $M^\sharp$ for a simplified abstract operational semantics – leaving us with the additional obligation to guarantee that the information extracted from the analysis of $M^\sharp$ is meaningful also for the original PDA M. The simplest method to do so is to establish a *simulation relation* between the PDAs M and $M^\sharp$.

Assume that we are given a set $D^\sharp$ of abstract values together with a relation $\Delta \subseteq D \times D^\sharp$. Let us call relation Δ *simulation* if Δ is *left-total*, that is, *every* concrete value $d \in D$ is simulated by at least one abstract value in $D^\sharp$. Usually, the set of abstract values is partially ordered where the ordering "$\sqsubseteq$" reflects the quality of approximation, that is, $d\ \Delta\ a$ and $a \sqsubseteq a'$ implies $d\ \Delta\ a'$. If this is the case, we call 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 commutes with arbitrary least upper bounds [12,11]. Then

$$\alpha X = \bigsqcup_{d \in X} \alpha\{d\} \quad \text{for all } X \subseteq D$$

and a compatible simulation relation Δ_α can be obtained by

$$d \Delta_\alpha a \quad \text{iff} \quad \alpha\{d\} \sqsubseteq a$$

We extend the notion of "simulation" to (partial) functions and PDAs. 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 $f(d_1, \ldots, d_k) \,\Delta\, f^\sharp(a_1, \ldots, a_k)$. We do not make any assumptions w.r.t. monotonicity (or continuity) of concrete or abstract functions. We only require abstract functions to be total and the simulation relation to be respected.

Assume now that M and $M^\sharp$ are PDAs for the same interprocedural CFG with sets of states D and $D^\sharp$, respectively. PDA M is *simulated* by PDA $M^\sharp$ (abbreviated: $M \,\Delta\, M^\sharp$) iff

1. the behavioral functions of M are simulated by the behavioral functions of $M^\sharp$; and (in case of dynamic procedure calls)
2. $\mathsf{call}_e\, d \subseteq \mathsf{call}^\sharp_e\, a$ whenever $d \,\Delta\, a$.

Thus given $M \,\Delta\, M^\sharp$, we can simulate each execution step of M by a corresponding execution step of $M^\sharp$. Therefore, we obtain:

Theorem 2. *If the PDA M is simulated by the PDA $M^\sharp$ and $d \,\Delta\, a$ then for every $p \in \mathsf{Proc}$ and $v \in N_p$,*
1. $(d, v) \vdash^*_M (d', \epsilon)$ *implies* $(a, v) \vdash^*_{M^\sharp} (a', \epsilon)$ *for some a' with $d' \,\Delta\, a'$;*
2. $(d, p) \vdash^*_M (d', v)$ *implies* $(a, p) \vdash^*_{M^\sharp} (a', v)$ *for some a' with $d' \,\Delta\, a'$;*
3. $(d, \mathsf{main}) \vdash^*_M (d', v\pi)$ *implies* $(a, \mathsf{main}) \vdash^*_{M^\sharp} (a', v\pi^\sharp)$ *for some a' and $\pi^\sharp$ with $d' \,\Delta\, a'$.*

By Theorem 2, all three analysis problems: effect, same-level reachability as well as reachability for M can be approximately solved by (approximatively) solving these problems for $M^\sharp$.

6. From PDAs to algorithms

The PDA for the abstract operational semantics may be directly used to determine (approximate) answers to our analysis problems. A major difficulty therein is that, in the presence of recursive procedures, the number of reachable configurations is infinite – even when the set of abstract states is finite. Hence, a naive implementation based on direct execution of PDAs may never terminate. In order to obtain a safe result after a finite number of steps, the PDA must be equipped with a detection method of (possible) non-termination together with some widening strategy for this case.

Another systematic way of avoiding non-termination consists of approximating the infinitely many configurations of the PDA by finitely many more abstract ones *beforehand*. This is the basic idea of the *call-string* approach of Sharir and Pnueli [49]. A configuration is abstracted by recording the $k + 1$ topmost stack elements. In the extreme case $k = 0$, every context except the current program point is ignored.

In the imperative world, such algorithms have been called "context-insensitive". Context-insensitive algorithms have also been proposed for Prolog (see for instance Ref. [43]). Interesting other choices of abstractions of pushdown configurations are suggested in Ref. [42].

In this presentation we will not follow these two approaches. Instead we systematically explore the design space of algorithms that avoid both kinds of further abstraction steps. The key idea is to attach to every procedure p a *transformer* on states denoting the *effect* of p on the state when p is called. Note that this approach has been pioneered by Sharir and Pnueli as well [49].

Our main technique for computing these transformers consists of finding (and later-on solving) suitable systems of *inequation* over some complete lattice D. An alternative and (essentially) equivalent approach relies on equations. We prefer inequations here, since they directly emerge from the graphical representation of programs as interprocedural CFGs and liberate us from ugly considerations about degenerated situations. In general, the inequations we consider are of the form $x \sqsupseteq t$ where $x \in \mathsf{Var}$ (Var the set of variables in the system) and t denotes a function $[\![t]\!] : (\mathsf{Var} \to D) \to D$ mapping variable assignments to values. The variables in inequations correspond to pieces of information we are interested in, whereas right-hand sides represent how variables are influenced by other variables. A *model* or *solution* of a system $\mathscr{C}$ of inequations is an assignment $\sigma : \mathsf{Var} \to D$ such that $\sigma\, x \sqsupseteq [\![t]\!]\, \sigma$ for all inequations $x \sqsupseteq t$ of $\mathscr{C}$. Clearly, every system of inequations has at least one solution, namely, the trivial one, mapping every variable to $\top$. Here we are interested in as small solutions as possible: the smaller a value, the higher is the precision. Each system $\mathscr{C}$ of inequations considered in this paper has a unique *least solution* which is denoted by $[\![\mathscr{C}]\!]$.

A typical inequation could state that the transformer f subsumes the composition of the transformers g_1 and g_2. This fact can be denoted as

$$f \sqsupseteq g_1 \circ g_2 \tag{1}$$

Equivalently, we can express this fact by the following *set* of inequations:

$$fd \sqsupseteq g_1(g_2 d), \quad d \in D \tag{2}$$

The first representation is well-suited in case where the transformers f, g_1 and g_2 can be succinctly represented as a whole and operations on these like composition can be computed efficiently. Indeed, this is quite frequently the case for imperative languages. Transformers according to the *Gen-Kill*-approach to data-flow analysis [25], for instance can be represented by pairs of sets. In case succinct representations are not known, however, representation (2) suggests to determine an unknown transformer by computing its *function graph* [34], that is, the set of its argument-result pairs. This can be made explicit by introducing a separate variable, say $y_{f,d}$, for every application fd, with f a (yet unknown) transformer and d a possible argument, which is going to receive the value of application fd. Then we obtain the set of inequations

$$y_{f,d} \sqsupseteq y_{g_1, y_{g_2,d}}, \quad d \in D \tag{3}$$

They are again of the form $x \sqsupseteq t$. The nested application has mutated into *indirect* indexing: the argument d_1 for which we need to determine the return value of g_1 is obtained through variable $y_{g_2,d}$. The advantage of formulation (3) is that not all variables $y_{f,d}$ of a transformer f necessarily contribute to the questions the analysis is going to answer. Therefore, the least solution of the possibly huge system of inequations needs to be only *partially* computed. This idea is exploited by *local solvers* [6,22,23] (see Section 9).

Instead of using the somewhat clumsy representation (3) which is good for implementation but shifts everything one level down into the indices, we prefer to stick to the more readable representation (2) and keep in mind that in fact each transformer application represents a variable. Analogously, we will also feel free to avoid for other index sets I the construction of sets of variables like $\{y_i \mid i \in I\}$ and use I *directly* as a set of variables.

Let us begin with an (auxiliary) effect analysis. There are two independent design choices for systems of inequations:

1. whether to compute effect information *for all* program points ("backward accumulation") or whether to compute the effects of procedures only and to use same-level reachability for accumulating these intraprocedurally ("forward accumulation");
2. whether to use set-valued transformers from D to 2^D as values for our procedure variables ("relational analysis") or ordinary transformers from D to D ("functional analysis").

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

7. Forward accumulation

The basic idea of *forward accumulation* is that computing the effect of procedures is reduced to solving another analysis problem, namely, *same-level reachability*. The effect of a procedure p on some input d is (conceptually) determined in two stages. First, we determine the values reaching exit points r through paths starting at the entry point of p; second, we apply Exit_r to determine the possible return values. We first explain this idea for the relational case. Its functional realization is considered in the subsequent subsection.

7.1. Relational analysis

Let D denote a set of values. In order to simplify our inequations, we introduce the auxiliary functionals $\mathscr{M}$ and $\mathscr{E}$ which take functions from $D \to D$ and $D \to 2^D$, respectively, map them over a set and collect all possible results:

$$\begin{array}{lll} \mathscr{M} : (D \to D) \to 2^D \to 2^D & \mathscr{M} f X & = \{fx \mid x \in X\} \\ \mathscr{E} : (D \to 2^D) \to 2^D \to 2^D & \mathscr{E} f X & = \bigcup \{fx \mid x \in X\} \end{array}$$

Furthermore, we introduce functionals $H_e^* : (D \to 2^D) \to D \to 2^D$, $e \in \mathsf{Call}$, which take the effect of a procedure and compute the corresponding effect of the call, that is,

$$H_e^* \, f \, d = \{\mathsf{Comb}_e \, (d_1, d) \mid d_1 \in f(\mathsf{Entry}_e \, d)\}$$

for $f : D \to 2^D$ and $d \in D$. Thus, the application $(H_e^* f)$ combines the value d before the call with all possible return values of f on $\mathsf{Entry}_e d$.

According to the strategy of forward accumulation, we use two kinds of *set-valued* variables for every $d \in D$, namely, variables $pd, p \in \mathsf{Proc}$, for the effects of procedures on input d, and variables $\langle u, d\rangle$, $u \in \mathsf{Point}$, for same-level reachability from the respective procedure entry (when entered with state d). The system $\mathscr{R}_f$ is defined by:

$$\begin{array}{lll r}
p\,d & \supseteq \mathscr{M}\ \mathsf{Exit}_r\ \langle r, d\rangle & \text{if } r \in R_p & (1)\\
\langle p, d\rangle & \supseteq \{d\} & \text{if } p \in \mathsf{Proc} & (2)\\
\langle v, d\rangle & \supseteq \mathscr{M}\ \mathsf{Trans}_e\ \langle u, d\rangle & \text{if } e = (u, v) \in \mathsf{Basic} & (3)\\
\langle v, d\rangle & \supseteq \mathscr{E}\ (H_e^* p)\ \langle u, d\rangle & \text{if } e = (u, v) \in \mathsf{Call}, p = \mathsf{call}_e & (4)
\end{array}$$

for all $d \in D$. Intuitively, the inequations are obtained as follows. Assume that we want to determine the set of possible return values for a procedure p on some input d. Then we first determine the sets $\langle u, d\rangle$ of values arriving at the program points u of p. Clearly, the value d arrives at the entry point of p (remember that this is denoted by p as well) – yielding the inequations of line (2). If $e = (u, v)$ is an edge in Basic, then all values $\mathsf{Trans}_e\ d'$ arrive at v where d' arrives at u – yielding the inequations of line (3). Accordingly, if $e = (u, v)$ is an edge in Call with p' as called procedure, then all the values in $(H_e^* p')\,d'$ arrive at v for every d' arriving at u – yielding the inequations of line (4). Having thus determined the set of values arriving at a return point r of p, we obtain return values for p by applying Exit_r to each of these values – yielding line (1).

In case of dynamic procedure calls, line (4) for call edge $e = (u, v)$ must be replaced by

$$\langle v, d\rangle \supseteq \bigcup\{H_e^*\ p\,x \mid x \in \langle u, d\rangle, p \in \mathsf{call}_e x\} \qquad (4')$$

Since the meaning of procedures are *relations*, we call an analysis based on (partially) computing the least solution of $\mathscr{R}_f$ *relational*. Here, we follow Ref. [31] where the term "relational" has been used in a similar way. Note, however, that in Ref. [14] "relational analysis" has been used instead to describe an abstraction technique for direct products – meaning that tuples of concrete elements are not abstracted by tuples of abstract elements but by relations, that is, *sets* of tuples of abstract elements.

We have the following theorem:

Theorem 3. *For all* $p \in \mathsf{Proc}$, $v \in N_p$ *and* $d \in D$,
1. $[\![\mathscr{R}_f]\!]\ p\,d = \{d' \in D \mid (d, p) \vdash^* (d', \epsilon)\}$;
2. $[\![\mathscr{R}_f]\!]\ \langle v, d\rangle = \{d' \in D \mid (d, p) \vdash^* (d', v)\}$.

Theorem 3 can be interpreted as a first and general Interprocedural Coincidence Theorem stating that the effects of procedures (as defined through the operational semantics) are precisely characterized by the least solution of system $\mathscr{R}_f$. It should be emphasized that Theorem 3 contains no assumptions on the nature of D or the monotonicity/continuity of behavioral functions. We only rely on preserving the simulation relation. Thus, an effective analysis engine is already obtained in a case where the set of values is finite. For Prolog, solving a system of inequations in the spirit of $\mathscr{R}_f$ has been called OLDT-resolution and is practically evaluated by Van Hentenryck et al. [26]. The idea of OLDT-resolution goes back to research by Tamaki and Sato [45] on tabled resolution for Prolog, see also Ref. [35]. A similar coincidence result concerning the formal relationship to a PDA-based operational semantics has been described by Lang [39] (see also Ref. [16]). New algorithms for relational analysis are proposed in Ref. [23].

7.2. Functional analysis

Relational analysis computes with sets. For efficiency reasons we often would like to replace this precise but expensive description by a less precise but (hopefully) computationally more tractable one. To study such kinds of (implicitly applied) *widenings* [12], let us make the following additional assumptions:

1. D is a complete lattice of abstract values,
2. the simulation relation is compatible, and
3. all behavioral functions are total and monotonic where (in case of dynamic procedure calls),
4. $d_1 \sqsubseteq d_2$ implies $\mathsf{call}_e d_1 \subseteq \mathsf{call}_e d_2$.

Slightly less restrictive assumptions for which essentially the same techniques can be applied are explained in Ref. [7]. The basic idea is to approximate occurring sets by one of their upper bounds. The best choice (w.r.t. precision) is to use *least* upper bounds here. Thus, we obtain a system $\mathscr{F}_f$ for *functional analysis with forward accumulation* by replacing in $\mathscr{R}_f$ all $\supseteq$ and all $\cup$ with $\sqsupseteq$ and $\bigsqcup$, respectively. The resulting system of inequations now speaks about values in D only. Accordingly, we have to replace functional H_e^* from the last subsection with the (much simpler) functional $H_e : (D \to D) \to D \to D$ defined as

$$H_e\ f\ d = \mathsf{Comb}_e(f(\mathsf{Entry}_e\ d), d)$$

The new system $\mathscr{F}_f$ is then given by:

$$\begin{array}{llll}
p\ d & \sqsupseteq \mathsf{Exit}_r\langle r, d\rangle & \text{if } r \in R_p & (1) \\
\langle p, d\rangle & \sqsupseteq d & \text{if } p \in \mathsf{Proc} & (2) \\
\langle v, d\rangle & \sqsupseteq \mathsf{Trans}_e\langle u, d\rangle & \text{if } e = (u, v) \in \mathsf{Basic} & (3) \\
\langle v, d\rangle & \sqsupseteq H_e\, p\langle u, d\rangle & \text{if } e = (u, v) \in \mathsf{Call}, p = \mathsf{call}_e & (4)
\end{array}$$

for all $d \in D$. In case of dynamic procedure calls, line (4) for call edge $e = (u, v)$ must now be replaced by

$$\langle v, d\rangle \sqsupseteq \bigsqcup\{H_e\, p\ \langle u, d\rangle \mid p \in \mathsf{call}_e\langle u, d\rangle\} \qquad (4')$$

Current interprocedural analyzers of imperative languages [1,36] perform functional analysis with forward accumulation.

If D is finite, the least solution of $\mathscr{F}_f$ can be computed by chaotic fixpoint iteration (possibly with "needed information only") as suggested by Cousot and Cousot [13]. For imperative languages, (functional forward accumulating) effect analysis based on worklist solvers has been proposed already by Sharir and Pnueli [49] 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 [23]. In case D is infinite, approximation methods like those of Bourdoncle [3] may be applicable. If F is the lattice of possibly occurring transformers $D \to D$, $\mathscr{F}_f$ can also be viewed as system of equations over F. In this case, "ordinary" fixpoint methods may be applied, see Refs. [37,36] or [28–30] for instances of this idea for imperative languages. For logic languages, this approach has been suggested in Refs. [32,40]. 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.

By fixpoint induction, we prove:

Theorem 4. *For all $p \in \mathsf{Proc}$, $v \in N_p$ and $d \in D$,*
1. $\bigsqcup\,([\![\mathscr{R}_f]\!]\ p\ d) \sqsubseteq [\![\mathscr{F}_f]\!]\ p\ d$;
2. $\bigsqcup\,([\![\mathscr{R}_f]\!]\ \langle v, d\rangle) \sqsubseteq [\![\mathscr{F}_f]\!]\ \langle v, d\rangle$.

In the light of Theorem 3, we find that the left-hand sides here represent what has been called the "interprocedural meet over all path" solution (interprocedural MOP) to the analysis problem [49,37,28,29]. Theorem 4 guarantees safety of functional analysis (with forward accumulation) relative to this interprocedural MOP respectively relative to relational analysis – thus implying overall safety of functional effect analysis.

8. Backward accumulation

Surprisingly enough, there is an alternative approach to interprocedural program analysis which, although (to a certain extent) more natural, has not attracted much attention so far: *backward* accumulation. Here, effect information is not only computed for procedures but for *every* program point. The idea is to view every program point u as a separate procedure. The procedure body of u essentially consists of the outgoing edges (u, v) followed by a call to v. We explain this second approach analogously to Section 7.

8.1. Relational analysis

Let D denote a set of values. For every $d \in D$, we introduce set-valued variables $u\ d, u \in \mathsf{Point}$, which are going to collect the effects of program points on input d. The system $\mathscr{R}_b$ is then defined by:

$$\begin{array}{lll l} r\ d & \supseteq \{\mathsf{Exit}_r\ d\} & \text{if } r \in \mathsf{Return} & (1)\\ u\ d & \supseteq v\ (\mathsf{Trans}_e\ d) & \text{if } e = (u, v) \in \mathsf{Basic} & (2)\\ u\ d & \supseteq \mathscr{E}\ v\ (H_e^* pd) & \text{if } e = (u, v) \in \mathsf{Call}, p = \mathsf{call}_e & (3) \end{array}$$

for all $d \in D$.

In case of dynamic procedure calls, line (3) for call edge $e = (u, v)$ should be replaced by

$$u\ d \supseteq \mathscr{E}\, v(\cup\{(H_e^* pd) \mid p \in \mathsf{call}_e d\}) \qquad (3')$$

In a certain sense, (the least solution of) system $\mathscr{R}_b$ can be viewed as the *big-step* operational semantics corresponding to the small-step operational semantics of the program. Intuitively, the inequations of $\mathscr{R}_b$ are obtained as follows. Assume we want to determine for some program point u and some incoming value d the set of all values to be returned at intraprocedurally reachable return points. If the program point in question equals a return point r, this set should contain $\mathsf{Exit}_r\ d$ – yielding the inequations from line (1). If u has an outgoing edge $e = (u, v)$ which is a basic computation step, then we must collect all values which are returned for v on input $\mathsf{Trans}_e\ d$ – yielding the inequations from line (2). Finally, if u has an outgoing edge $e = (u, v)$ which is a call to the procedure p, then we should collect all values which are returned for v on values arriving at v after the call to p – yielding the inequations from line (3).

Since the meaning of program points are *relations* represented by mappings $D \to 2^D$, we call an analysis based on (partially) computing the least solution of $\mathscr{R}_b$ *relational* with backward accumulation. We have:

Theorem 5. *For all* $u \in \mathsf{Point}$ *and* $d \in D$, $[\![\mathscr{R}_b]\!]\ u\ d = \{d' \in D \mid (d,u) \vdash^* (d',\epsilon)\}$.

Theorem 5 presents a second general Interprocedural Coincidence Theorem stating that the effects of program points (as defined through the operational semantics) are precisely characterized by the least solution of $\mathscr{R}_b$. Since no assumptions are made on the nature of D or the monotonicity/continuity of behavioral functions, an effective analysis engine is obtained in case where the set of values is finite. For Prolog, $\mathscr{R}_b$ has been proposed and practically evaluated on some programs by Van Hentenryck et al. [26].

8.2. Functional analysis

As for forward accumulation, let us now study the impact of widenings. Therefore, we again additionally assume that D is a complete lattice of abstract values, that the simulation relation is compatible and all behavioral functions are total and monotonic where (in case of dynamic procedure calls) $d_1 \sqsubseteq d_2$ implies $\mathsf{call}_e\ d_1 \subseteq \mathsf{call}_e\, d_2$. From $\mathscr{R}_b$ we obtain a system for functional analysis with backward accumulation by replacing all $\supseteq$ and all $\bigcup$ with $\sqsupseteq$ and $\bigsqcup$. The new system of inequations uses the same variables as $\mathscr{R}_b$ which now receive values in D (instead of values in 2^D). This system $\mathscr{F}_b$ is given by:

$$\begin{array}{lll r}
r\ d & \sqsupseteq \mathsf{Exit}_r\ d & \text{if } r \in \mathsf{Return} & (1)\\
u\ d & \sqsupseteq v\ (\mathsf{Trans}_e\ d) & \text{if } e = (u,v) \in \mathsf{Basic} & (2)\\
u\ d & \sqsupseteq v\ (H_e\ p\ d) & \text{if } e = (u,v) \in \mathsf{Call}, p = \mathsf{call}_e & (3)
\end{array}$$

for all $d \in D$. In case of dynamic procedure calls, line (3) for call edge $e = (u,v)$ should be replaced by

$$u\ d \sqsupseteq v\ (\bigsqcup\{(H_e p d) \mid p \in \mathsf{call}_e d\}) \quad (3')$$

Analogously to Theorem 4, we find:

Theorem 6. *For all* $u \in \mathsf{Point}$ *and* $d \in D$, $\bigsqcup\ ([\![\mathscr{R}_b]\!]\ u\ d) \sqsubseteq [\![\mathscr{F}_b]\!]\ u\ d$.

Theorem 6 implies overall safety of functional analysis with backward accumulation. To the best of our knowledge, functional analysis with backward accumulation has not attracted attention for program analysis so far.

9. Reachability analysis

For an initial state d_0, we would like to determine for every program point v, the set of all $d \in D$ such that $(d_0, \mathsf{main}) \vdash^* (d, v\pi)$ for some π. There are many ways to obtain reachability analyses. Typically, they build on effect information (conceptually) computed beforehand by solving one of the systems of inequations considered in Sections 7 or 6. Here, we restrict ourselves to two approaches. One idea is to *extend* a system of inequations for effect analysis by further variables and inequations. Another approach, however, tries to avoid the construction of an auxiliary system of inequations. Instead, it extracts reachability information from the *variable dependences* of systems for effect analysis. We start with this second approach.

9.1. Variable dependences

Let us introduce the following proviso:

(R) For every procedure p and and every program point $v \in N_p$, there is a path in $\mathcal{G}_p$ from v to a return point of p.

Proviso (R) states that intraprocedural CFGs are “well-formed” in the sense that all syntactical dead ends have been removed.

Let $\mathcal{C}$ denote a system of inequations with a (not necessarily least) solution σ. Assume furthermore that we are given a set X of “interesting” variables. We consider the set $\mathsf{dep}(X,\sigma)$ of variables on which the values of σ for $x \in X$ (recursively) depend. Thus, $\mathsf{dep}(X,\sigma)$ is the least set Y of variables such that $X \subseteq Y$, and whenever $x \in Y$, $x \sqsupseteq t$ is an inequation in $\mathcal{C}$, and the evaluation of t for σ accesses variable y, then also $y \in Y$. Recall from Section 6 that for our systems of inequations, this set indeed may depend on σ. We obtain:

Theorem 7.
1. *Assume that σ is the least solution of $\mathcal{R}_b$. Then*

$$\{v\ d \mid \exists \pi : (d_0, \mathsf{main}) \vdash^* (d, v\pi)\} = \mathsf{dep}(\{\mathsf{main}\ d_0\}, \sigma)$$

2. *Assume that proviso (R) holds, and σ is the least solution of $\mathcal{R}_f$. Then for every program point v of procedure p,*

$$\{d \in D \mid \exists \pi : (d_0, \mathsf{main}) \vdash^* (d, v\pi)\} = \cup\{\sigma\langle v, d_1\rangle \mid p\ d_1 \in \mathsf{dep}(\{\mathsf{main}\ d_0\}, \sigma)\}$$

The correspondence between reachability and variable dependences according to $\mathcal{R}_b$ is surprisingly simple: they coincide. For $\mathcal{R}_f$, the correspondence is more complicated – and additionally relies on proviso (R). The intuitive idea for $\mathcal{R}_f$ is as follows. Analogous to the case of backward accumulation, the set of states for which a procedure p is called can be determined *directly* through variable dependences. The states reaching arbitrary program points of p can then be recovered through same-level reachability. The following example explains why the additional assumption (R) in Theorem 7 (2) is necessary.

Example 8. Consider the trivial program whose interprocedural control-flow graph is shown in Fig. 2. The exit point 1 of procedure main has no ingoing edge. Let us determine the variable dependences introduced by forward accumulation. According to line (1) in $\mathcal{R}_f$, the value of the transformer for main on some input value d_0 depends on the variable $\langle 1, d_0\rangle$, that is, the set of abstract values arriving at 1. Since there is no ingoing edge for 1, no further variables can be added to $\mathsf{dep}(\{\mathsf{main}\,d_0\}, \sigma)$ – independent of how σ looks like. Thus, variable dependences alone fail to report the reachability of some call to the procedure p.

The situation for backward accumulation is different. According to line (3) of $\mathcal{R}_b$, the application $\mathsf{main}\,d_0$ formally depends on the result of $p\ d_1$ for suitable d_1 and therefore also on applications $2\ d_2$ and $0\ d_3$ for certain $d_2, d_3 \in D$. Especially, all

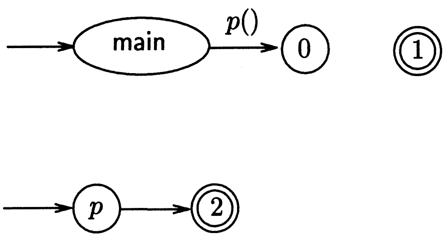

Fig. 2. The control-flow graph for example 8.

reachability information, namely, d_0 for main, d_1 for p and d_2 and d_3 for program points 2 and 0, respectively, are reported.

A theorem similar to Theorem 7 can be obtained for functional analyses.

Theorem 9.
1. *Assume that* σ *is a solution of* $\mathcal{F}_b$ (*not necessarily the least one*). *Then for every program point* v, $(d_0, \mathsf{main}) \vdash^* (d, v\pi)$ *for some* π *implies* $d \sqsubseteq d_1$ *for some* d_1 *with* $v d_1 \in \mathsf{dep}(\{\mathsf{main}\, d_0\}, \sigma)$.
2. *Assume that* σ *is a solution of* $\mathcal{F}_f$ (*not necessarily the least one*). *Then for every program point* v *of the procedure* p, $(d_0, \mathsf{main}) \vdash^* (d, v\pi)$ *for some* π *implies* $d \sqsubseteq \sigma\langle v, d_1\rangle$ *for some* d_1 *with* $p\ d_1 \in \mathsf{dep}(\{\mathsf{main}\, d_0\}, \sigma)$.

We conclude that in case of forward accumulation, we obtain a safe approximation to all values reaching a program point v of a procedure p by computing

$$\rho v = \bigsqcup\{\sigma\langle v, d\rangle \mid p d \in \mathsf{dep}(\{\mathsf{main}\ d_0\}, \sigma)\}$$

Note that the formulations of Theorems 7 and 9 suggest that reachability analysis be implemented in two phases. First, an effect analysis is performed whose result is used in a second phase to determine the set $\mathsf{dep}(\{\mathsf{main}\ d_0\}, \sigma)$ and then reachability. Such a post-processing phase is described, for instance, by Le Charlier and Van Hentenryck [8].

In fact, this second phase can be avoided. In Ref. [49], Sharir and Pnueli suggest a reachability analysis based on computing the least solution of their system of (in-) equations for effect analysis *on demand.* The appealing idea is that the task of determining a safe approximation of the set of variables on which the interesting variables depend can be delegated to the exploration of the variable space through *local solving* [21]. Starting from a set X of interesting variables, local solvers try to compute a partial solution σ which is defined only for a (hopefully) small superset of $\mathsf{dep}(X, \sigma)$. Thus, we can safely replace the latter set with the entire domain of σ. We only have to guarantee that all variables $\langle v, d\rangle$, v a program point of the procedure p, are contained in the domain of σ whenever the variable $p\ d$ has been included. For local solvers like those considered in Refs. [6,22,23] this is – under proviso (R) – always the case.

Summarizing, the resulting algorithm (with forward accumulation) proceeds as follows. It maintains for every program point v a variable $\langle v\rangle$ which has been initialized with $\bot$. In particular, since $\mathsf{Proc} \subseteq \mathsf{Point}$, we thus also have introduced variables $\langle p\rangle$ for every procedure p. Then the algorithm performs local effect analysis starting from the single interesting variable main d_0. Whenever during this computation some variable $\langle v, d\rangle$ receives a new value, this value is (as a side effect) also added to $\langle v\rangle$. For a procedure p, this means that whenever p is called on d, the variable $\langle p, d\rangle$ receives value d implying that d is added to $\langle p\rangle$ as well. Finally, when the ultimate value for main d_0 has been obtained, the variables $\langle v\rangle$ contain safe approximate reachability information.

9.2. Adding inequations

Alternatively, we obtain reachability analyses by extending systems of inequations for effect analysis. For every program point v, a new variable $\langle v\rangle$ is introduced which is going to receive reachability information for v. For simplicity we restrict ourselves to systems of inequations with forward accumulation. Similar ideas work for systems with backward accumulation as well.

In case of relational analysis (with forward accumulation), variable $\langle v\rangle$ is going to receive the *set* of all values d arriving at v. Therefore, we extend $\mathcal{R}_f$ by adding the following inequations:

$$\begin{array}{lll l}
\langle \mathsf{main}\rangle & \supseteq \{d_0\} & \text{for initial value } d_0 \in D & (5)\\
\langle p\rangle & \supseteq \mathcal{M}\ \mathsf{Entry}_e\ \langle u\rangle & \text{if } e = (u, \_) \in \mathsf{Call}, p = \mathsf{call}_e & (6)\\
\langle v\rangle & \supseteq \mathcal{E}\ (\lambda d.\langle v, d\rangle)\ \langle p\rangle & \text{if } v \in N_p, v \neq p & (7)
\end{array}$$

Here, the somewhat strange expression $\mathcal{E}\ (\lambda d.\langle v, d\rangle)\ \langle p\rangle$ denotes the union of all sets $\langle v, d\rangle, d \in \langle p\rangle$. Let us denote the resulting system of equations by $\bar{\mathcal{R}}_f(d_0)$. Line (5) says that the initial actual parameter d_0 for main arrives at the entry point of main; line (6) says that the set of values arriving at a procedure p can be obtained from the sets of all values arriving at calls to p by application of the corresponding Entry-functions; finally, line (7) says that the set of all values arriving at a program point v of the procedure p is the union of all sets $\langle v, d\rangle$ where d arrives at p.

In case of functional reachability, we are interested in one single value for every program point which describes *all* possibly reaching values simultaneously. Thus, the variables $\langle v\rangle$ now receive values from D (which is assumed to be a complete lattice). The system of inequations for functional reachability is obtained from $\mathcal{F}_f$ by adding the inequations:

$$\begin{array}{lll l}
\langle \mathsf{main}\rangle & \sqsupseteq d_0 & \text{for initial value } d_0 \in D & (5)\\
\langle p\rangle & \sqsupseteq \mathsf{Entry}_e\ \langle u\rangle & \text{if } e = (u, \_) \in \mathsf{Call}, p = \mathsf{call}_e & (6)\\
\langle v\rangle & \sqsupseteq \langle v, \langle p\rangle\rangle & \text{if } v \in N_p, v \neq p & (7)
\end{array}$$

Let us denote the resulting system by $\bar{\mathcal{F}}_f(d_0)$. In case of dynamic procedure calls, line (6) for call edge $e = (u, v)$ in the systems $\bar{\mathcal{R}}_f(d_0)$ and $\bar{\mathcal{F}}_f(d_0)$ must be replaced with lines (6′) and (6″), respectively:

$$\begin{array}{lll}
\langle p\rangle & \supseteq \{\mathsf{Entry}_e\ x \mid x \in \langle u\rangle,\ p \in \mathsf{call}_e x\} & (6')\\
\langle p\rangle & \sqsupseteq \bigsqcup\{\mathsf{Entry}_e\ \langle u\rangle \mid p \in \mathsf{call}_e\langle u\rangle\} & (6'')
\end{array}$$

By fixpoint induction, we find:

Theorem 10. *For all* $p \in \mathsf{Proc}$ *and* $v \in N_p$,
1. $[\![\bar{\mathcal{R}}_f(d_0)]\!]\ \langle v\rangle = \{d \in D \mid \exists \pi : (d_0, \mathsf{main}) \vdash^* (d, v\pi)\}$.
2. $\bigsqcup\ ([\![\bar{\mathcal{R}}_f(d_0)]\!]\ \langle v\rangle) \sqsubseteq [\![\bar{\mathcal{F}}_f(d_0)]\!]\ \langle v\rangle$.

The system $\bar{\mathscr{F}}_f(d_0)$ is the method at hand when summary functions for procedures are computed as a whole as in the frameworks [32,36,28,29]. It may *lose*, however, precision against a reachability analysis through local solvers, simply because just one value per program point is maintained to describe the whole reachability information. Formally, we have:

Theorem 11. *Let $\rho : \mathsf{Point} \to D$ denote the reachability information extracted from variable dependences from $\mathscr{F}_f$ (and the least solution) according to the preceding subsection. Then*
1. *For every program point v, $\rho v \sqsubseteq [\![\bar{\mathscr{F}}_f(d_0)]\!]\langle v\rangle$;*
2. *The inclusion of (1) can be strict.*

10. Forward versus backward accumulation

In this section, we compare the systems of inequations for effect analysis with forward and backward accumulation w.r.t. precision. Corresponding results also hold for the systems of inequations for reachability analysis. From Theorems 3 and 5, we obtain:

Theorem 12. *For every $p \in \mathsf{Proc}$ and $d \in D$, $[\![\mathscr{R}_f]\!]\; p\; d = [\![\mathscr{R}_b]\!]\; p\; d$.*

While the least solutions for the systems of relational analysis with forward and backward accumulation contain the same information, this does no longer hold for the corresponding systems of functional analysis.

Systems $\mathscr{F}_f$ as well as $\mathscr{F}_b$ use transformers from $D \to D$ to represent the effects of procedures. They differ, however, in the way how these effects are accumulated *intraprocedurally*. Functional analysis with forward accumulation abstracts the set of all values intraprocedurally reaching some program point by just one value. On the contrary, functional analysis with backward accumulation keeps all these values distinct. Computing least upper bounds is delayed until the very end. Accordingly, we find:

Theorem 13. *For every $p \in \mathsf{Proc}$ and $d \in D$ $[\![\mathscr{F}_b]\!]\; p\; d \sqsubseteq [\![\mathscr{F}_f]\!]\; p\; d$.*

In order to show that "$\sqsubseteq$" in Theorem 13 cannot generally be replaced with "$=$", we give an example where forward accumulation *loses* precision 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. For the following example assume that our Prolog dialect provides also *complex* goals of the form $(g_1; g_2)$ where ";" represents the OR operator. Then consider the predicate definition

$$p \leftarrow (a;p), b$$

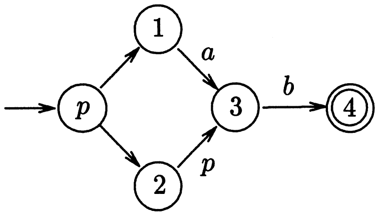

Fig. 3. The interprocedural CFG for $p \leftarrow (a;p),\ b$.

where a and b are basic goals. The corresponding control-flow graph is shown in Fig. 3. Let us consider the complete lattice $D = \{\bot < d_1, d_2 < \top\}$ where

$$\begin{array}{l} [\![a]\!] = \mathsf{Trans}_{(1,3)} = Id \\ [\![b]\!] = \mathsf{Trans}_{(3,4)} = f \end{array} \quad \text{where} \quad f\ x = \begin{cases} d_2 & \text{if } x \in \{d_1, d_2\} \\ x & \text{otherwise} \end{cases}$$

In order to simplify calculations, we furthermore define

$$\mathsf{Exit}_4 = \mathsf{Entry}_{(2,3)} = \mathsf{Trans}_{(p,1)} = \mathsf{Trans}_{(p,2)} = Id$$

$$\mathsf{Comb}_{(2,3)}\ (x_1, x) = x_1$$

Starting with the value $\bot$ for the application $p\,d_1$, a fixpoint algorithm for $\mathscr{F}_f$ may compute the following approximations:

$$\sigma_f^{(1)}\ p\,d_1 = f\,(d_1 \sqcup \bot) = f\ d_1 = d_2$$

$$\sigma_f^{(2)}\ p\,d_1 = f\,(d_1 \sqcup d_2) = f\,\top = \top = [\![\mathscr{F}_f]\!]p d_1$$

For system $\mathscr{F}_b$ the following approximations are computed:

$$\sigma_b^{(1)}\ p\,d_1 = (f\,d_1) \sqcup (f\ \bot) = d_2 \sqcup \bot = d_2$$

$$\sigma_b^{(2)}\ p\,d_1 = (f\,d_1) \sqcup (f\,d_2) = d_2 \sqcup d_2 = d_2 = [\![\mathscr{F}_b]\!]p d_1$$

Thus, $\mathscr{F}_b$ returns a more precise result for $p\ d_1$ than $\mathscr{F}_f$.

Therefore, functional analysis with forward accumulation may *lose* precision against functional analysis with backward accumulation. There is, however, a simple structural condition which enforces equivalence between the two. Let us call the intraprocedural CFG $\mathscr{G}_p$ *simple* if it is a tree with root p, and *complex* otherwise.

Theorem 15. *Assume all intraprocedural CFGs are simple. Then for every* $p \in \mathsf{Proc}$ *and* $d \in D$, $[\![\mathscr{F}_f]\!]\ p\ d = [\![\mathscr{F}_b]\!]\ p\ d$.

A proof of Theorems 13 and 15 is included in Appendix A. By Theorem 15, precision may be only lost when analyzing programs with *complex* intraprocedural control-flow graphs. These are typical for imperative programs and "real" Prolog programs which may contain complex goals as in Example 14. No precision is lost at *normalized* Prolog programs as introduced in Section 3, since these introduce simple intraprocedural control-flow graphs only.

11. Coincidence theorems for functional analysis

In the previous section, we investigated the impact of the accumulation strategy on precision. In this section we compare relational analysis with functional analysis. Practical comparisons of these two approaches w.r.t. precision and efficiency for Prolog are presented by Hentenryck et al. [27] and Fecht [21]. While the first one experiments with small programs and very complicated domains, the latter one analyzes quite large programs (for instance, the source code of the Aquarius Prolog compiler) but with less complicated domains (essentially POS and various domains for sharing).

Here, we are interested in scenarios where *provably* no precision is lost, that is, "$\sqsubseteq$" in Theorems 4, 6 and 10 can be replaced by equality ("coincidence"). Such scenarios promise the best of two worlds: the precision of relational analysis as well as the efficiency of their functional abstractions. In their seminal paper on interprocedural data-flow analysis of imperative programs [49], Sharir and Pnueli claim coincidence in case all transfer functions (are continuous[1] and) respect binary least upper bounds. Here is a trivial example showing that this does not suffice.[2]

Example 16. Consider the following C program:

```
int x;
main() { q(); x = 1; }
   q() { q() }
```

The corresponding interprocedural CFG is shown in Fig. 4. Assume that we want to perform interprocedural *copy-constant* propagation for this program. Then we can use the complete lattice

$$D = \{\bot \sqsubset 1 \sqsubset \top\}$$

where $d \in D$ reports the value of the (global) variable x. Value $\bot$ denotes that x has not yet been initialized, whereas value $\top$ denotes that the value of x is not known. For the assignment edge $e = (1,2)$, we obtain $\mathsf{Trans}_e\ d = 1$ – independent of d. Clearly, Trans_e preserves binary lubs. Now consider the program point 2 immediately after execution of the assignment, that is, at the return point of procedure main. Since 2 is not reachable by any execution path of the (abstract) operational semantics, the interprocedural MOP for 2 equals $\bigsqcup\emptyset = \bot$. Opposed to that, the solution computed by the functional approach of Sharir and Pnueli assigns some value to the program point before the assignment implying that, no matter what this value is, value 1 will be assigned to program point 2 – which therefore differs from the interprocedural MOP. A similar defect may even occur in *intraprocedural* analysis if conditions are taken into account – which is indispensable for any reasonable constant propagation [11,1]. Then reachability of a program point by some (abstract) execution path can no longer be (easily) reduced to graph reachability.

[1] In fact, Sharir and Pnueli consider coincidence just for finite domains. Therefore, they only demand monotonicity.

[2] A similar example has independently been found by Horwitz et al. [44].

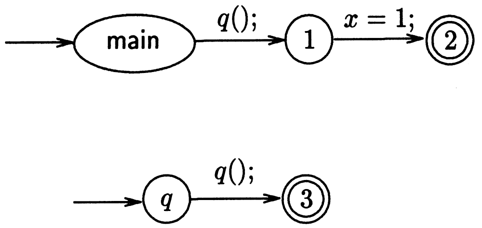

Fig. 4. The interprocedural CFG for Example 16.

Coincidence in the framework of Sharir and Pnueli as well as in the frameworks of their followers Knoop and Steffen [37,36] relies on the additional (silent) assumption that every program point is formally reachable by at least one execution path. The situation is somewhat different in the framework of Horwitz et al. [28–30]. They fail to compute $\perp$ for program point 2 of our example as well, but they do not claim to do so either. Instead of referring to interprocedural execution paths of the operational semantics, they solely rely on "interprocedurally valid control-flow paths". Then they introduce *extra* control-flow edges such that (at least formally) interprocedural reachability is guaranteed whenever the call graph is connected [44].

Our approach allows us to refine the known scenario for coincidence as well as to derive completely new coincidence theorems. Hence, we are able to deduce the first coincidence result for a functional analysis of logic programs, namely, of groundness analysis based on abstract domain POS [5,10]. Here, we only report our coincidence results for functional analysis with forward accumulation. The same scenarios, however, guarantee coincidence for functional analysis with backward accumulation as well.

For the purposes of this section let us consider programs without dynamic procedure calls where the set of values is a complete lattice. A function $f : D_1 \to D_2$, (D_1, D_2 complete lattices) is called

- *strict* iff $f\ \perp = \perp$;
- *finitely distributive* iff $f\ (d_1 \sqcup d_2) = f\ d_1 \sqcup f\ d_2$ for all $d_1, d_2 \in D_1$;
- *completely distributive* iff $f(\bigsqcup X) = \bigsqcup\{f x \mid x \in X\}$ for every $X \subseteq D_1$.

A strict and finitely distributive function being continuous also preserves arbitrary least upper bounds, and therefore is even completely distributive.

We start by exhibiting a master scenario which will be sufficient for proving coincidence. Later on we will give more concrete instances of this very general scheme. Our master scenario consists of the following properties:

Master Scenario: We are given a complete sublattice $\mathcal{F} \subseteq D \to D$ of completely distributive functions which contains the constant function $\lambda x.\ \perp$ as well as the identity $\lambda x.x$ and is closed under composition. Furthermore,

1. All unary behavioral functions are completely distributive;
2. $\mathsf{Trans}_e \in \mathcal{F}$ for all $e \in \mathsf{Basic}$;
3. Comb_e is completely distributive in its first argument;
4. If r is an exit point of the procedure called at edge e, then $H_e(\mathsf{Exit}_r \circ f) \in \mathcal{F}$ whenever $f \in \mathcal{F}$.

The sublattice $\mathcal{F}$ serves as the set of all possible descriptions of same-level reachability, that is, all transformers transforming the value d at the entry point of a procedure to the value $\langle v, d\rangle$ reaching program point v of this procedure. These transformers should be completely distributive. Moreover, they should be obtained in a "distributive way" meaning that the operators used for their construction are "well-behaving". In particular, this results in the restrictions on Comb_e, H_e, and Exit_r as expressed in (3) and (4).

Observe that our Master Scenario does not demand Entry_e or Exit_r to be elements of $\mathscr{F}$; and indeed, in our Scenarios II and III this will rarely be the case. We obtain our Master Coincidence Theorem for functional analysis:

Theorem 17. *Assume the Master Scenario is satisfied. Then for all $p \in \mathsf{Proc}$, $u \in N_p$ and $d \in D$,*

$$\begin{aligned}
&\bigsqcup\{d' \in D \mid (d,p) \vdash^* (d',\epsilon)\} = \bigsqcup([\![\mathscr{R}_f]\!]\ p\ d) = [\![\mathscr{F}_f]\!]\ p\ d \\
&\bigsqcup\{d' \in D \mid (d,p) \vdash^* (d',u)\} = \bigsqcup([\![\mathscr{R}_f]\!]\ \langle u,d\rangle) = [\![\mathscr{F}_f]\!]\ \langle u,d\rangle \\
&\bigsqcup\{d' \in D \mid \exists\pi : (d_0,\mathsf{main}) \vdash^* (d',u\pi)\} = \bigsqcup([\![\bar{\mathscr{R}}_f(d_0)]\!]\ \langle u\rangle) = [\![\bar{\mathscr{F}}_f(d_0)]\!]\ \langle u\rangle
\end{aligned}$$

For a proof see Appendix B. Theorem 17 is not easily applicable since property (4) of the Master Scenario relies on a quite complicated interplay between Entry_e, Exit_r and Comb_e. Therefore, we derive more concrete scenarios.

Scenario I:

1. All unary behavioral functions are completely distributive;
2. Comb_e is continuous where additionally $\mathsf{Comb}_e(\bot,d) = \bot$ and whenever $\bot \neq d_1,d_2$, $\mathsf{Comb}_e(d_1 \sqcup d_2, d'_1 \sqcup d'_2) = \mathsf{Comb}_e(d_1,d'_1) \sqcup \mathsf{Comb}_e(d_2,d'_2)$.

Please note that our Scenario I is quite restrictive with the special value $\bot$. In contrast, the authors in Refs. [49,37,28,29,36,30], require behavioral functions to be *finitely* distributive only – nothing concerning $\bot$ is asked for. All these papers, therefore, treat the non-terminating program from Example 16 overly conservative (w.r.t. the operational semantics).

Luckily, there is a *general technique* to overcome their deficiency. Let D denote a complete lattice. Then we introduce the *lifted* complete lattice $D_\bot = \{\bot\} \cup D$ where $\bot \sqsubset d$ for all $d \in D$. Assume that $f : D \to D$ is continuous and finitely distributive, but not strict. We extend f to a continuous function $f_\bot : D_\bot \to D_\bot$ defined by $f_\bot\ \bot = \bot$ and $f_\bot\ d = f\ d$ otherwise. Then $f_\bot$ is finitely distributive as well as strict (and hence completely distributive). Accordingly, we extend a continuous, finitely distributive function $g : D \times D \to D$ to the binary continuous function $g_\bot : D_\bot \times D_\bot \to D_\bot$ by $g_\bot(d_1,d_2) = \bot$ whenever $d_1 = \bot$ or $d_2 = \bot$ and $g_\bot(d_1,d_2) = g(d_1,d_2)$ otherwise. Then $g_\bot$ satisfies properties (2) and (3) of Scenario I. For a semantic justification of lifting, we observe:

Proposition 18. *Assume that from a given PDA M with set of states D, we constructed a PDA $M_\bot$ with state set $D_\bot$ by taking the same interprocedural CFG and replacing every behavioral function f with $f_\bot$. Then for every $u \in \mathsf{Point}$ and stack w the following holds*:

1. *For all $d_1,d_2 \in D$, $(d_1,u) \vdash^*_M (d_2,w)$ iff $(d_1,u) \vdash^*_{M_\bot} (d_2,w)$.*
2. *$(d,u) \vdash^*_{M_\bot} (\bot,w)$ iff $d = \bot$.*

The new abstract value $\bot$ is meant to simulate the *empty set* of concrete values, that is, $x \Delta \bot$ for *no* concrete value x. Adding $\bot$ neither hinders nor helps the PDA – it does help, however, system $\mathscr{F}_f$ (as well as system $\bar{\mathscr{F}}_f(d_0)$) to keep track of empty sets. Note that once such an additional abstract value $\perp$ is available, it may also be used to mark certain (abstract) transitions as not viable – without sacrificing the totality of behavioral functions.

Summarizing, let us assume that we are given a complete lattice D together with behavioral functions which are continuous and but only finitely distributive as in Refs. [37,36]. Then (on malicious programs) coincidence between the results of relational analysis and functional analysis may fail. In order to avoid this failure we suggest to enhance the analysis by lifting D to $D_\perp$ and replacing all behavioral functions f with $f_\perp$. The resulting PDA will satisfy the conditions of Scenario I – implying that relational analysis will be as precise as functional analysis.

Distributivity of the unary behavioral functions is a common phenomenon in the analysis of imperative programs. Also, many important interprocedural analyses rely on Comb-functions which essentially are least upper bounds of two unary finitely distributive functions [29,30,36] and are therefore themselves finitely distributive. Surprisingly, the situation for logic languages looks quite different. In fact, we do not know of *any* non-trivial analysis of logic programs where Scenario I can be applied.

As an important example, consider groundness analysis based on abstract domain POS [5,10]. In this case, the unary behavioral functions Entry_e, Exit_r and Trans_e are all completely distributive. The binary functions Comb_e (corresponding to extG), however, are essentially given by "$\sqcap$", the greatest lower bound operator. Operator "$\sqcap$" is strict in each argument – but usually fails to satisfy property (3) of Scenario I. To see this, assume that there are elements $a, b \in D$ with $a \sqsubset b$. Then

$$\sqcap((a,b) \sqcup (b,a)) = \sqcap(b,b) = b \neq a = (\sqcap(a,b)) \sqcup (\sqcap(b,a))$$

Therefore, we searched for further instances of the Master Scenario. For the following two scenarios let us generally assume that we are given completely distributive *renaming* functions $\mathsf{ci}_e, \mathsf{co}_e : D \to D$ ("copy-in", "copy-out") with

- $\mathsf{co}_e\ (\mathsf{ci}_e\ x) = x$;
- Entry_e can be factored $\mathsf{Entry}_e\ x = \mathsf{ci}_e\ (\mathsf{In}_e\ x)$.

Recall that renaming functions with these two properties have already been introduced in Section 3 for Prolog. There, the function In_e is given by the restriction to the variables in the current goal. Renaming functions, however, are also useful for imperative languages to conveniently model the passing of arguments into formal parameters.

Scenario II: The greatest lower bound operator "$\sqcap$" is completely distributive in each argument. As set $\mathscr{F}$ we then choose all $f : D \to D$ with $f\ \perp = \perp$ and $f\ x = (a \sqcap x) \sqcup b$ for all $x \neq \perp$ where a, b are suitable coefficients in D. Moreover,
1. In_e and all unary behavioral functions besides Entry_e are in $\mathscr{F}$;
2. For non-$\perp$-arguments, the Comb-functions rely on "$\sqcup$" and "$\sqcap$" to combine return values with the state before the call: $\mathsf{Comb}_e(x_1, x) = \perp$ for $x_1 = \perp$; otherwise for suitable $a_1, a_2, a_3 \in D$, $\mathsf{Comb}_e(x_1, x) = a_0 \sqcup (a_1 \sqcap \mathsf{co}_e\ x_1) \sqcup (a_2 \sqcap x) \sqcup (a_3 \sqcap \mathsf{co}_e\ x_1 \sqcap x)$.
3. The copy-out functions co_e commute with greatest lower bounds: $\mathsf{co}_e\ (a \sqcap b) = (\mathsf{co}_C\ a) \sqcap (\mathsf{co}_C\ b)$ for all $a, b \in D$.

Scenario III: We are given a binary operator $* : D \times D \to D$ which is associative and completely distributive in each argument where $\top * d = d$. Then set $\mathscr{F}$ is given by all functions f of the form $f\,x = a * x$ for some $a \in D$. Moreover,

1. $\mathsf{Trans}_e \in \mathscr{F}$ for all $e \in \mathsf{Basic}$;
2. $\mathsf{Comb}_e(x_1, x) = (\mathsf{co}_e\, x_1) * x$ where $\mathsf{co}_e\,(a * b) = (\mathsf{co}_e\, a) * (\mathsf{co}_e\, b)$ for all $a, b \in D$;
3. The functions In_e and Exit_r are completely distributive where $(\mathsf{In}_e\, x) * x = x$; and for $p = \mathsf{call}\ e$ and $r \in R_p$, $\mathsf{Exit}_r\,(x_1 * (\mathsf{Entry}_e\, x)) = (\mathsf{Exit}_r\, x_1) * (\mathsf{Entry}_e\, x)$.

Scenario II is quite common in data-flow analysis of imperative programs. Its assumptions are met for instance by an analysis of *truly uninitialized variables*. Scenario III is closely related to the notion of *condensing* for Prolog [32]. There, our operator "$*$" corresponds to *abstract instantiation*.

Example 19. Let us consider groundness analysis with POS. Assume that the edge e corresponds to the literal $g \equiv p(X_{j_1}, \ldots, X_{j_k})$. Then the variable renamings $\mathsf{ci}_e, \mathsf{co}_e : \mathrm{POS} \to \mathrm{POS}$ are given by

$$\mathsf{ci}_e\,\phi = \phi[X_{j_i} \mapsto X_i]_{i=1}^{k} \qquad \mathsf{co}_e\,\phi = \phi[X_i \mapsto X_{j_i}]_{i=1}^{k}$$

Then $\mathsf{In}_e = \mathsf{RestrG}\ g$ restricts to the variables $X_{j_1}, \ldots, X_{j_k}$ occurring in g, whereas Exit_r, r an exit point of p, restricts to the variables $X_1, \ldots, X_k$.

The abstract instantiation operator "$*$" for POS equals "$\wedge$", the logical AND which (for POS) coincides with the greatest lower bound. Indeed, the effect of the unification $X = t$ onto an abstract substitution ϕ is $a \wedge \phi$ where

$$a \;\equiv\; X \leftrightarrow \bigwedge_{Y \in V} Y \quad (V \text{ the set of variables occurring in } t)$$

With these definitions, the identities of Scenario III can be verified.

Theorem 20. *Scenarios I, II and III are all instances of the Master Scenario.*

For Scenario III, we included a proof into Appendix B. To the best of our knowledge, no coincidence theorems for Scenarios II or III have occurred in the literature. From Theorem 20 and Example 19 we conclude for groundness analysis with POS that, if we are interested in POS information alone, functional groundness analysis through $\mathscr{F}_f$ (or $\bar{\mathscr{F}}_f(d_0)$) is as precise as relational groundness analysis through $\mathscr{R}_f$ (or $\bar{\mathscr{R}}_f(d_0)$).

12. Output subsumption and disjunctive completion

In this section, we study the effect of two further concepts, namely *output subsumption* and *disjunctive completion*. While output subsumption is a technique which may enhance efficiency of relational analysis without sacrificing precision, disjunctive completion has been suggested to improve functional analysis. For the rest of this section we assume that the set D of abstract values is a complete lattice, and that the simulation relation is compatible. We start with a discussion of output subsumption.

In the light of Theorem 2 we observe that $X \subseteq D$ does not contain more information than its *lower closure* $X{\downarrow} = \{d \in D \mid \exists x \in X : d \sqsubseteq x\}$. Therefore, we can as well design relational analyses which compute lower closures *directly*. This optimization is suggested in [26]. The idea is that finite lower sets can be efficiently represented by the anti-chains of their maximal elements.

By $\mathscr{P}(D) \subseteq 2^D$ we denote the set of all subsets $X \subseteq D$ with $X = X\!\downarrow$, ordered by set inclusion. For finite D, $\mathscr{P}(D)$ is also known as *Hoare powerdomain*. For infinite D, the Hoare powerdomain of D does *not* consist of all lower sets but only of lower sets which are *closed*, that is, additionally contain all lubs of directed subsets. Another concept related to our $\mathscr{P}(D)$ is the *disjunctive completion* of D in the sense of Refs. [15,24]. There, the disjunctive completion is obtained from 2^D by identifying subsets which have identical concretizations, that is, simulate identical sets of concrete elements. It can be viewed as a surjective image of $\mathscr{P}(D)$. Since it can never be more precise as $\mathscr{P}(D)$, we ignore these subtle distinctions and just work with $\mathscr{P}(D)$.

The mapping $X \mapsto X\!\downarrow$ is continuous and surjective where $(X \cup Y)\!\downarrow = (X\!\downarrow) \cup (Y\!\downarrow)$. Let us apply this operator to the systems $\mathscr{R}_f$ and $\bar{\mathscr{R}}_f(d_0)$, respectively. The resulting systems of inequations $\mathscr{R}_f\!\downarrow$ and $\bar{\mathscr{R}}_f(d_0)\!\downarrow$ then compute with lower sets from $\mathscr{P}(D)$. System $\mathscr{R}_f\!\downarrow$ is given by:

$$\begin{array}{lll}
p\ d \supseteq \mathscr{M}'\ \mathsf{Exit}_r\ \langle r, d\rangle & \text{if } r \in R_p & (1)\\
\langle p, d\rangle \supseteq \{d\}\!\downarrow & \text{if } p \in \mathsf{Proc} & (2)\\
\langle v, d\rangle \supseteq \mathscr{M}'\ \mathsf{Trans}_e\ \langle u, d\rangle & \text{if } e = (u, v) \in \mathsf{Basic} & (3)\\
\langle v, d\rangle \supseteq \mathscr{E}'\ (H'_e p)\ \langle u, d\rangle & \text{if } e = (u, v), p \in \mathsf{call}_e & (4)
\end{array}$$

for $d \in D$. System $\bar{\mathscr{R}}_f(d_0)\!\downarrow$ additionally contains the inequations:

$$\begin{array}{llll}
\langle \mathsf{main}\rangle & \supseteq \{d_0\}\!\downarrow & \text{for initial value } d_0 \in D & (5)\\
\langle p\rangle & \supseteq \mathscr{M}'\ \mathsf{Entry}_e\ \langle u\rangle & \text{whenever } e = (u, \_) \in \mathsf{Call}, p \in \mathsf{call}_e & (6)\\
\langle v\rangle & \supseteq \mathscr{E}'\ (\lambda x.\langle v, x\rangle)\ \langle p\rangle & \text{if } v \in N_p, v \neq p & (7)
\end{array}$$

The only differences to systems $\mathscr{R}_f$ and $\bar{\mathscr{R}}_f(d_0)$ are that the singleton sets in lines (2) and (5) have been replaced by their lower closures, and H^*_e is replaced with H'_e where

$$H'_e\ f\ d = \{\mathsf{Comb}_e(d_1, d) \mid d_1 \in f\,(\mathsf{Entry}_e\, d)\}\!\downarrow$$

Also, we had to substitute $\mathscr{M}$ and $\mathscr{E}$ with the corresponding functions

$$\mathscr{M}' : (D \to D) \to \mathscr{P}(D) \to \mathscr{P}(D) \qquad \mathscr{E}' : (D \to \mathscr{P}(D)) \to \mathscr{P}(D) \to \mathscr{P}(D)$$

for $\mathscr{P}(D)$. Recall that whenever X is given by a (hopefully small) set $X_0 \subseteq X$ with $X = X_0\!\downarrow$, and $f : D \to D$ and $h : D \to \mathscr{P}(D)$ are monotonic, then we can implement $\mathscr{M}'$ and $\mathscr{E}'$ by

$$\mathscr{M}'\ f\ X = \{f\ x \mid x \in X_0\}\!\downarrow \qquad \mathscr{E}'\ h\ X = \bigcup_{x \in X_0} h\ x$$

Using fixpoint induction we verify:

Theorem 21. *For all* $p \in \mathsf{Proc}$, $v \in N_p$ *and* $d \in D$,
1. $([\![\mathscr{R}_f]\!]\ \langle v, d\rangle)\!\downarrow = [\![\mathscr{R}_f\!\downarrow]\!]\ \langle v, d\rangle$,
2. $([\![\mathscr{R}_f]\!]\ p\ d)\!\downarrow = [\![\mathscr{R}_f\!\downarrow]\!]\ p\ d$, and
3. $([\![\bar{\mathscr{R}}_f(d_0)]\!]\ \langle v\rangle)\!\downarrow = [\![\bar{\mathscr{R}}_f(d_0)\!\downarrow]\!]\ \langle v\rangle$.

Theorem 21 is the formal justification why we safely may apply output subsumption to gain efficiency without sacrificing precision.

Let us now discuss the suggestion by Filé and Ranzato to enhance the precision of interprocedural analysis by replacing the set D of abstract values with the *disjunctive completion* of D [24]. Their observation is that functional analysis may lose precision by taking least upper bounds of sets of elements. This loss in precision can be avoided if the analysis no longer computes with individual elements but with (lower) sets. While relational analysis cannot take advantage of this enhancement (computing with elements simply is replaced with computing with singleton sets of elements), functional analysis may gain precision. Here, we argue that functional analysis enhanced with disjunctive completion is at most as precise as relational analysis – but may still sometimes *lose* precision. The reason is that a loss in precision for functional analysis may not only occur at least upper bounds but also when combining return values with states before the call.

Given the behavioral functions for D, the corresponding behavioral functions for $\mathscr{P}(D)$ are constructed as follows. The new unary behavioral functions $f_{\mathscr{P}} : \mathscr{P}(D) \to \mathscr{P}(D)$ are obtained from the corresponding unary functions $f : D \to D$ by $f_{\mathscr{P}}\, X = \{f\ x \mid x \in X\}\!\downarrow$. Accordingly, the new binary functions $g_{\mathscr{P}} : \mathscr{P}(D) \times \mathscr{P}(D) \to \mathscr{P}(D)$ are obtained from the corresponding old functions g by $g_{\mathscr{P}}(X_1, X_2) = \{g\ (x_1, x_2) \mid x_1 \in X_1, x_2 \in X_2\}\!\downarrow$. Let $\mathscr{F}_{\mathscr{P},f}$ and $\bar{\mathscr{F}}_{\mathscr{P},f}\{d_0\}$ denote the systems of inequations for functional effect analysis and functional reachability analysis, respectively, where the abstract domain is given by $\mathscr{P}(D)$. By fixpoint induction we verify:

Theorem 22. *For all* $p \in \mathsf{Proc}$, $v \in N_p$ *and* $X \in \mathscr{P}(D)$,
1. $\bigcup_{d \in X} ([\![\mathscr{R}_f\!\downarrow]\!]\ \langle v, d\rangle) \subseteq [\![\mathscr{F}_{\mathscr{P},f}]\!]\ \langle v, X\rangle$;
2. $\bigcup_{d \in X} ([\![\mathscr{R}_f\!\downarrow]\!]\ p\ d) \subseteq [\![\mathscr{F}_{\mathscr{P},f}]\!]\ p\ X$;
3. $[\![\bar{\mathscr{R}}_f(d_0)\!\downarrow]\!]\ \langle v\rangle \subseteq [\![\bar{\mathscr{F}}_{\mathscr{P},f}\{d_0\}]\!]\ \langle v\rangle$.

Theorem 23. *For all* $p \in \mathsf{Proc}$, $v \in N_p$ *and* $d \in D$,
1. $\bigsqcup([\![\mathscr{F}_{\mathscr{P},f}]\!]\ \langle v, \{d\}\!\downarrow\rangle) \sqsubseteq [\![\mathscr{F}_f]\!]\ \langle v, d\rangle$;
2. $\bigsqcup([\![\mathscr{F}_{\mathscr{P},f}]\!]\ p\ \{d\}\!\downarrow) \sqsubseteq [\![\mathscr{F}_f]\!]\ p\ d$;
3. $\bigsqcup([\![\bar{\mathscr{F}}_{\mathscr{P},f}\{d_0\}]\!]\ \langle v\rangle) \sqsubseteq [\![\bar{\mathscr{F}}_f(d_0)]\!]\ \langle v\rangle$.

By Theorem 22, no better information can be obtained through lower sets than through relational analysis. On the other hand by Theorem 23, some gain may be obtained against ordinary functional analysis.

Van Hentenryck et al. [26] have found examples where relational analysis is more precise than functional analysis. Let us construct an example which exhibits a situation where relational analysis is even more precise than functional analysis enhanced with lower sets.

Example 24. Consider the Prolog program

$$\begin{aligned} \mathsf{main} &\leftarrow a(X, Y, Z), p(X, Y, Z) \\ a(X, Y, Z) &\leftarrow X = Y \\ a(X, Y, Z) &\leftarrow X = Z \\ p(X, Y, Z) &\leftarrow \end{aligned}$$

The corresponding interprocedural control-flow graph is shown in Fig. 5. Assume that we want to compute *pair-sharing* information with the pair-sharing domain

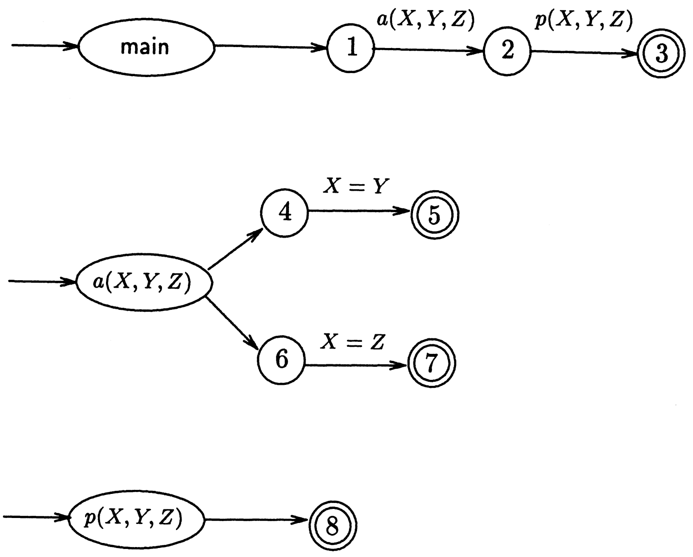

Fig. 5. The interprocedural CFG for Example 24.

PS of Søndergaard [50]. Let us determine the value returned by main on input $\emptyset$. For the program point 2 before the call to p we obtain:

$$\begin{aligned}[\![\mathscr{R}_f{\downarrow}]\!]\ \langle 2,\emptyset\rangle &= \{\mathsf{Comb}_{(1,2)}\ (s_1,\emptyset), \mathsf{Comb}_{(1,2)}\ (s_2,\emptyset)\}{\downarrow} = \{s_1,s_2\}{\downarrow}\\ &= [\![\mathscr{F}_{\mathscr{P},f}]\!]\ \langle 2,\{\emptyset\}{\downarrow}\rangle\end{aligned}$$

where $s_1 = \{(X,Y)\}$ and $s_2 = \{(X,Z)\}$. Thus, the same set of values arrives for both analyses. A difference, however, occurs in the way how the call to p is treated. We have

$$[\![\mathscr{R}_f{\downarrow}]\!]\ p\ d = \{d\}{\downarrow} \quad [\![\mathscr{F}_{\mathscr{P},f}]\!]\ p\ X = X$$

Thus, we calculate for the program point 3:

$$\begin{aligned}[\![\mathscr{R}_f{\downarrow}]\!]\ \langle 3,\emptyset\rangle &= \{\mathsf{Comb}_{(2,3)}\ (s_1,s_1), \mathsf{Comb}_{(2,3)}\ (s_2,s_2)\}{\downarrow}\\ &= \{s_1,s_2\}{\downarrow}\\ [\![\mathscr{F}_{\mathscr{P},f}]\!]\ \langle 3,\{\emptyset\}{\downarrow}\rangle &= \mathsf{Comb}_{(2,3),\mathscr{P}}\ (\{s_1,s_2\}{\downarrow},\{s_1,s_2\}{\downarrow})\\ &= \{\mathsf{Comb}_{(2,3)}\ (s_1,s_1), \mathsf{Comb}_{(2,3)}\ (s_2,s_1), \mathsf{Comb}_{(2,3)}\ (s_2,s_2)\}{\downarrow}\\ &= \{s_1,s,s_2\}{\downarrow}\end{aligned}$$

where $s = \{(X,Y),(X,Z),(Y,Z)\}$. We conclude that, according to relational analysis, sharing of Y and Z is excluded. Opposed to that, functional analysis with $\mathscr{P}(\mathrm{PS})$ cannot exclude concrete substitutions with sharing between Y and Z, due to the extra element s. Thus, $[\![\mathscr{F}_{\mathscr{P},f}]\!]\ \langle 3,\{\emptyset\}{\downarrow}\rangle$ is strictly less precise than $[\![\mathscr{R}_f{\downarrow}]\!]\ \langle 3,\emptyset\rangle$.

All intraprocedural CFGs of our counter example are simple. Therefore, the same loss of precision would have occurred if we used functional analysis enhanced with lower sets and backward accumulation instead. Precision is possibly lost since for $X \in \mathscr{P}(D)$ and $f : D \to \mathscr{P}(D)$, the sets

$$\{\mathsf{Comb}_e\ (d_1,d) \mid d \in X, d_1 \in f\ d\}\!\downarrow \text{ and } \{\mathsf{Comb}_e\ (d_1,d) \mid d \in X, d_1 \in \mathscr{E}\ f\ X\}\!\downarrow$$

need not be equal.

13. Conclusion

We have presented a general framework for program analysis based on a small-step operational semantics. In order to describe procedures operationally, we relied on pushdown automata. We explored several extreme points in the design space of systems of inequations derivable from (abstract) PDAs. We also explained how algorithms for reachability analysis can be derived from systems of inequations for effect analysis.

Then we applied our general framework to exhibit the impact of design choices onto precision. In particular, we compared forward accumulation with backward accumulation. Surprisingly, while the accumulation strategy had no effect on precision for relational systems of inequations, a degradation of precision could be observed in the functional case. We also investigated the relative precision of relational versus functional systems of inequations. We exhibited scenarios where no precision is lost. Finally, we critically reviewed the general technique of disjunctive completion in the context of interprocedural analysis and compared it with relational analysis.

The present paper clearly focused onto the conceptual similarities of imperative and logic languages which allowed us to design a common framework for program analyses. Thus, at least in principle, the same algorithms are applicable in both areas. On the contrary, we feel that there are clear *pragmatic* differences. Algorithms which have been found efficient for the analysis of Prolog need not yield the best results also for imperative languages and vice versa. A detailed comparison of the two language classes in this respect remains for future work.

Acknowledgements

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, as well as the anonymous referees for numerous valuable comments on earlier drafts and important pointers to the literature.

Appendix A. Proof of Theorems 13 and 15

For simplicity let us assume that the behavioral functions are not only monotonic but even continuous. The proof for the general case is an extension of our method using transfinite fixpoint induction.

For $a \in \{b, f\}$, let us define variable assignments $\sigma_a^{(j)}, j \geqslant 0$, by:

- $\sigma_a^{(0)}$ maps all variables to $\bot$.
- In order to construct $\sigma_a^{(j+1)}$, remove all uses of transformers p, $p \in \mathsf{Proc}$, in right-hand sides of $\mathscr{F}_a$ by replacing them with functions $\sigma_a^{(j)}\ p$. Then define $\sigma_a^{(j+1)}$ the least solution of the resulting system of inequations.

The right-hand sides of the inequations for computing the respective next approximation $\sigma_a^{(j+1)}$ do no longer contain procedure calls. We observe:

Claim 1: For every a, $\sigma_a^{(j)}, j \geq 0$, is a non-decreasing sequence of variable assignments with $\bigsqcup_j \sigma_a^{(j)} = [\![\mathscr{F}_a]\!]$.

Let us now fix some $j > 0$, and define for every $e = (u, v) \in \mathsf{Call}$, a transformer $h_{e,a} : D \to D$ by $h_{e,a} x = \bigsqcup\{H_e\,(\sigma_a^{(j-1)}\ p) x \mid p \in \mathsf{call}_e x\}$. Let $\mathscr{S}_b$ denote the system of inequations

$$\begin{array}{lll} r\ d & \sqsupseteq \mathsf{Exit}_r\ d & \\ u\ d & \sqsupseteq v\ (\mathsf{Trans}_e\ d) & \text{if } e = (u,v) \in \mathsf{Basic} \\ u\ d & \sqsupseteq v\ (h_{e,b}\ d) & \text{if } e = (u,v) \in \mathsf{Call} \end{array}$$

Accordingly, let $\mathscr{S}_f$ denote system

$$\begin{array}{lll} p\ d & \sqsupseteq \mathsf{Exit}_r\ \langle r, d\rangle & \text{if } r \in R_p \\ \langle p, d\rangle & \sqsupseteq d & \text{if } p \in \mathsf{Proc} \\ \langle v, d\rangle & \sqsupseteq \mathsf{Trans}_e\ \langle u, d\rangle & \text{if } e = (u,v) \in \mathsf{Basic} \\ \langle v, d\rangle & \sqsupseteq h_{e,f}\ \langle u, d\rangle & \text{if } e = (u,v) \in \mathsf{Call} \end{array}$$

Here, all right-hand sides depend on at most one variable. Clearly, $\sigma_a^{(j)} = [\![\mathscr{S}_a]\!]$.

For program points u, v let $\mathsf{Path}(u, v)$ denote the set of all *intraprocedural* paths from u to v. For a path π we can define the effect $[\![\pi]\!]_a : D \to D$ of π (w.r.t. to system $\mathscr{S}_a$) by $[\![\epsilon]\!]_a\ x = x$, $[\![e\ \pi']\!]_a\ x = [\![\pi']\!]_a\ (\mathsf{Trans}_e\ x)$ if $e = (u, v) \in \mathsf{Basic}$ and $[\![e\ \pi']\!]_a\ x = [\![\pi']\!]_a\ (h_{e,a}\ x)$ if $e \in \mathsf{Call}$. We claim:

Claim 2: Assume $v \in N_p$. Then

1. $[\![\mathscr{S}_b]\!]\ v\ d = \bigsqcup_{r \in R_p} \bigsqcup_{\pi \in \mathsf{Path}(v,r)} \mathsf{Exit}_r\ ([\![\pi]\!]_b\ d)$.
2. $[\![\pi]\!]_f\ x \sqsubseteq [\![\mathscr{S}_f]\!]\ \langle v, x\rangle$ for every path $\pi \in \mathsf{Path}(p, v)$.

Thus, the least solution of $\mathscr{S}_b$ is nothing but the *intraprocedural merge over all paths*. The same does not hold for $\mathscr{S}_f$. Since $\sigma_a^{(j)} = [\![\mathscr{S}_a]\!]$, we conclude that $\sigma_b^{(j)}\ p \sqsubseteq \sigma_f^{(j)}\ p$ for all j. This completes the proof of Theorem 13. In case of tree-shaped intraprocedural control-flow graphs, "$\sqsubseteq$" in Claim 2 can be replaced with equality implying Theorem 15.

Appendix B. Proofs of Theorems 17 and 20

Let us start with a proof of our Master Coincidence Theorem 17. For $j \geqslant 0$, let $\sigma^{(j)}$ denote the jth approximation to the least solution of $\bar{\mathscr{F}}_f(d_0)$. By induction on j, we first prove for all $j \geqslant 0$:

1. for every program point u, some $f \in \mathscr{F}$ exists such that $\sigma^{(j)}\ \langle u, x\rangle = f\,x$;
2. for every procedure p with set R of exit points, functions $f_r \in \mathscr{F}$, $r \in R$ exist such that $\sigma^{(j)} p x = \bigsqcup_{r \in R} \mathsf{Exit}_r\ (f_r x)$.

Then let $\bar{\sigma}^{(j)}, j \geqslant 0$, denote the jth approximation to the least solution of $\bar{\mathscr{R}}_f(d_0)$. It remains to prove that for all $j \geqslant 0$,

1. for every program point u, $\bigsqcup \bar{\sigma}^{(j)}\ \langle u, x\rangle = \sigma^{(j)}\ \langle u, x\rangle$;
2. for every procedure p, $\bigsqcup \bar{\sigma}^{(j)} p\, x = \sigma^{(j)} p\, x$;
3. for every program point u, $\bigsqcup \bar{\sigma}^{(j)}\ \langle u\rangle = \sigma^{(j)}\ \langle u\rangle$.

For $j = 0$, these assertions are clearly satisfied. For $j > 0$, we show that for every inequation, every outside lub can be pushed "inside". To get an idea, how the proof proceeds, consider $e = (u, v) \in \mathsf{Call}$ which calls p. For the inequations of $\langle v, x\rangle$ in $\bar{\mathscr{R}}_f(d_0)$ and $\bar{\mathscr{F}}_f(d_0)$ corresponding to e, we calculate:

$$\begin{aligned}
&\bigsqcup \mathscr{E}\ (H_e^*(\bar{\sigma}^{(j-1)} p))\ (\bar{\sigma}^{(j-1)} \langle u, x\rangle) \\
&\quad = \bigsqcup \bigcup \{H_e^*(\bar{\sigma}^{(j-1)} p)\, d \mid d \in \bar{\sigma}^{(j-1)} \langle u, x\rangle\} && \text{(B.1)} \\
&\quad = \bigsqcup \{\bigsqcup H_e^*(\bar{\sigma}^{(j-1)} p)\, d \mid d \in \bar{\sigma}^{(j-1)} \langle u, x\rangle\} && \text{(B.2)} \\
&\quad = \bigsqcup \{H_e(\bigsqcup \bar{\sigma}^{(j-1)} p)\, d \mid d \in \bar{\sigma}^{(j-1)} \langle u, x\rangle\} && \text{(B.3)} \\
&\quad = \bigsqcup \{H_e(\sigma^{(j-1)} p)\, d \mid d \in \bar{\sigma}^{(j-1)} \langle u, x\rangle\} && \text{(B.4)} \\
&\quad = H_e(\sigma^{(j-1)} p)\, (\bigsqcup \bar{\sigma}^{(j-1)} \langle u, x\rangle) && \text{(B.5)} \\
&\quad = H_e(\sigma^{(j-1)} p)\, (\sigma^{(j-1)} \langle u, x\rangle) && \text{(B.6)}
\end{aligned}$$

Here, line (3) follows from complete distributivity of Comb_e in its first argument; line (4) follows by induction hypothesis for p and $j - 1$; line (5) follows since $H_e(\sigma^{(j-1)} p)$ is in $\mathscr{F}$ and therefore completely distributive; line (6) follows again by induction hypothesis – here for $\langle u, x\rangle$. □

Let us turn to a proof of Theorem 20. We only consider Scenario III. Let $\mathscr{F}$ denote the set of all functions of the form $f\,x = a * x$, $a \in D$. By the assumptions on "$*$", $\mathscr{F}$ contains the constant $\bot$-function (select $a = \bot$) and the identity (select $a = \top$). Since "$*$" is completely distributive in the first argument, $\mathscr{F}$ is a complete lattice. Furthermore, since "$*$" is associative, $\mathscr{F}$ is also closed under composition. From the remaining properties, we only consider Property (4).

Assume $e \in \mathsf{Call}$ is a call to the procedure p. Let r be an exit point of p, and let $f \in \mathscr{F}$ be given by $f\,x = a * x$. Then

$$\begin{aligned}
H_e(\mathsf{Exit}_r \circ f)\,x &= \mathsf{Comb}_e(\mathsf{Exit}_r(f(\mathsf{Entry}_e x)), x) \\
&= (\mathsf{co}_e(\mathsf{Exit}_r(a * (\mathsf{Entry}_e x)))) * x \\
&= b * (\mathsf{co}_e(\mathsf{Entry}_e x)) * x \\
&= b * (\mathsf{In}_e x) * x \\
&= b * x \qquad \text{where} \\
b &= \mathsf{co}_e(\mathsf{Exit}_r\, a)
\end{aligned}$$

Thus, $H_e(\mathsf{Exit}_r \circ f) \in \mathscr{F}$ which we wanted to prove.

References

[1] M. Alt, F. Martin, Generation of efficient interprocedural analyzers with PAG, in: Proceedings of the Seconnd Static Analysis Symposium (SAS), LNCS 983, Springer, Berlin, 1995, pp. 33–50.

[2] F. 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), LNCS 456, Springer, Berlin, 1990, pp. 307–323.

[3] F. Bourdoncle, Abstract interpretation by dynamic partioning, Journal of Functional Programming 2 (4) (1992) 407–435.

[4] F. Bourdoncle, Sémantiques des Langages Impératifs d'Ordre Supérieur et Interprétation Abstraite Ph.D. thesis, École Polytechnique, Paris, 1992.

[5] B. Le Charlier, A. Cortesi, P. Van Hentenryck, Evaluation of the domain Prop, Journal of Logic Programming 23 (3) (1995) 237–278.

[6] B. Le Charlier, P. Van Hentenryck, A universal top-down fixpoint algorithm, Technical Report CS-92-25, Brown University, Providence, RI 02912, 1992.

[7] B. Le Charlier, P. VanHentenryck, A generic fixpoint semantics for Prolog and its application for abstract interpretation, Technical Report, Institute of Computer Science, University of Namur, Belgium, (also Brown University), January 1993. Presented at the LOPSTR'93–COMPUNET Program Development Joint Workshop, Louvain la Neuve, Belgium, July 1993, Published in Proc. of JFLP'95, Dijon, France, June 1995.

[8] B. Le Charlier, P. Van Hentenryck, Experimental evaluation of a generic abstract interpretation algorithm for Prolog, ACM Transactions of Programming Languages and Systems (TOPLAS) 16 (1) (1994) 35–101.

[9] B.Le Charlier, S. Rossi, P. VanHentenryck, An abstract interpretation framework which accurately handles Prolog search-rule and the cut, in: M. Bruynooghe (Ed.), Proceedings of the International Logic Programming Symposium (ILPS'94), Ithaca NY, USA, November 1994, MIT Press, Cambridge, MA.

[10] A. Cortesi, G. Filé, W.H. Winsborough, Optimal groundness analysis using propositional logic, Journal of Logic Programming 27 (2) (1996) 137–167.

[11] P. Cousot, Semantic foundations of program analysis, in: S.S. Muchnick, N.D. Jones (Eds.), Program Flow Analysis: Theory and Applications, ch. 10, Prentice-Hall, Englewood Cliffs, NJ, 1981, pp. 303–342.

[12] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proceedings of the Fourth ACM Symposium on Principles of Programming Languages (POPL), ACM Press, New York, 1977, pp. 238–252.

[13] P. Cousot, R. Cousot, Static determination of dynamic properties of recursive programs, in: E.J. Neuhold (Ed.), Formal Descriptions of Programming Concepts, North-Holland, Amesterdam, 1977, pp. 237–277.

[14] P. Cousot, R. Cousot, Abstract interpretation and application to logic programs, Journal of Logic Programming 13 (2/3) 1992.

[15] P. Cousot, R. Cousot, Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), in: Proceedings of the International IEEE Conference on Computer Languages (ICCL), 1994, pp. 95–112.

[16] Eric Villemonte de la Clergerie, Automates à Piles et Programmation Dynamique: DyALog, Application à la Programmation en Logique, Ph.D. thesis, Université Paris VII, Paris, France, June 1993.

[17] E. Villemonte de la Clergerie, F. Barthélemy, Information flow in tabular interpretations for generalized pushdown automata, Theoretical Computer Science 199 (1/2) (1998) 167–198.

[18] S.K. Debray, Todd Proebsting, Inter-procedural control flow analysis of first order programs with tail call optimization, ACM Transactions on Programming Languages and Systems (TOPLAS) 19(4) (1997) 568–585.

[19] A. Deutsch, On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications, in: Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL), ACM Press, New York, 1990, pp. 157–168.

[20] V. Englebert, B. Le Charlier, D. Roland, P. Van Hentenryck, Generic abstract interpretation algorithms for Prolog: two optimization techniques and their experimental evaluation, Software – Practice and Experience 23 (4) (1993) 419–459.

[21] C. Fecht, Abstrakte Interpretation logischer Programme: Theorie, Implementierung, Generierung, Ph.D. thesis, Universität des Saarlandes, Saarbrücken, 1997.

[22] C. Fecht, H. Seidl, An even faster solver for general systems of equations, in: Proceedings of the Third Static Analysis Symposium (SAS), LNCS 1145, Springer, Berlin, 1996, pp. 189–204. Long version to appear in Science of Computer Programming (SCP).

[23] C. Fecht, H. Seidl, Propagating differences: an efficient new fixpoint algorithm for distributive constraint systems, in: European Symposium on Programming (ESOP), LNCS 1381, Springer, Berlin, 1998, pp. 90–104 Long version in Northern Journal of Computing 5 (1998) 304–329.

[24] G. Filé, F. Ranzato, Improving abstract interpretations by systematic lifting to the powerset, in: Proceedings of the International Symposium on Logic Programming (SLP), MIT Press, Cambridge, MA, 1994, pp. 655–669.

[25] M.S. Hecht, Flow Analysis of Computer Programs, Elsevier, Amsterdam, 1977.

[26] P. Van Hentenryck, O. Degimbe, B. Le Charlier, L. Michel, Abstract Interpretation of Prolog Based on OLDT Resolution, Technical Report CS-93-05, Brown University, Providence, RI 02912, 1993.

[27] P. Van Hentenryck, O. Degimbe, B. Le Charlier, L. Michel, The impact of granularity in abstract interpretation of Prolog, in: Proceedings of the Static Analysis, Third International Workshop (WSA), LNCS 724, Springer, Berlin, 1993, pp. 1–14.

[28] S. Horwitz, T.W. Reps, M. Sagiv, Precise interprocedural dataflow analysis via graph reachability, in: Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL), ACM Press, New York, 1995, pp. 49–61.

[29] S. Horwitz, T.W. Reps, M. Sagiv, Precise interprocedural dataflow analysis with applications to constant propagation, in: Proceedings of the Sixth International Conference on Theory and Practice of Software Development (TAPSOFT), LNCS 915, Springer, Berlin, 1995, pp. 651–665.

[30] S. Horwitz, T.W. Reps, M. Sagiv, Precise interprocedural dataflow analysis with applications to constant propagation, Theoretical Computer Science 167 (1/2) (1996) 131–170.

[31] J. Hughes, J. Launchbury, Reversing abstract interpretations, Science of Computer Programming (SCP) 22 (1994) 307–326.

[32] D. Jacobs, A. Langen, Static analysis of logic programs for independent AND parallelism, Journal of Logic Programming 13 (1992) 291–314.

[33] N.D. Jones, S.S. Muchnick, A flexible approach to interprocedural data flow analysis and programs with recursive data structures, in: Proceedings of the Ninth ACM Symposium on Principles of Programming Languages (POPL), ACM Press, New York, 1982, pp. 66–74.

[34] N.D. Jones, A. Mycroft, Data flow analysis of applicative programs using minimal function graphs, in: Proceedings of the 13th ACM Symposium on Principles of Programming Languages (POPL), ACM Press, New York, 1986, pp. 296–306.

[35] T. Kanamori, T. Kawamura, Abstract interpretation based on OLDT resolution, Journal of Logic Programming 15(1/2) (1993) 1–30.

[36] J. Knoop, O. Rüthing, B. Steffen, Towards a tool kit for the automatic generation of interprocedural data flow analyses, Journal of Programming Languages 4 (1996) 211–246.

[37] J. Knoop, B. Steffen, The interprocedural coincidence theorem, in: Proceedings of the Fourth International Conference on Compiler Construction, LNCS 641, Springer, Berlin, 1992, pp. 125–140.

[38] A. Krall, T. Berger, Incremental global compilation of Prolog with the Vienna abstract machine, in: International Conference on Logic Programming (ICLP), MIT Press, Cambridge, MA, 1995, pp. 333–347.

[39] B. Lang, Complete Evaluation of Horn Clauses: an Automata Theoretic Approach, Technical Report 913, INRIA, 1988.

[40] K. Marriott, H. Søndergaard, Precise and efficient groundness analysis for logic programs, ACM Letters on Programming Languages and Systems (LOPLAS 2 (1993) 181–196.

[41] K. Marriott, H. Søndergaard, N.D. Jones, Denotational abstract interpretation of logic programs, ACM Transactions of Programming Languages and Systems (TOPLAS) 16 (3) (1994) 607–648.

[42] F. Martin, M. Alt, R. Wilhelm, Analysis of loops, in: Proceedings of the Compiler Construction, Seventh International Conference, LNCS 1383, Springer, Berlin, 1998, pp. 80–94.

[43] U. Nilsson, Systematic semantic approximations of logic programs, in: International Workshop on Programming Language Implementation and Logic Programming (PLILP), LNCS 456, Springer, Berlin, 1990, pp. 293–306.

[44] T.W. Reps, Personal Communication, February 1997.

[45] T. Sato, H. Tamaki, OLD resolution with tabulation, in: E.Y. Shapiro (Ed.), Proceedings of the Third International Conference on Logic Programming (ICLP), LNCS 225, Springer, Berlin, 1986, pp. 84–98.

[46] D.A. Schmidt, Abstract interpretation of small-step semantics, in: M. Dam (Ed.) Proceedings of the Fifth LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages, LNCS 1192, Springer, Berlin, 1996.
[47] H. Seidl, C. Fecht, Disjunctive completion is not "optimal", in: International Logic Programming Symposium (ILPS), MIT Press, Cambridge, MA, 1997, p. 408.
[48] H. Seidl, C. Fecht, Interprocedural analysis based on PDAs, in: Verification, Model Checking and Abstract Interpretation, A Workshop in Association with ILPS'97, 1997.
[49] M. Sharir, A. Pnueli, Two approaches to interprocedural data flow analysis, in: S.S. Muchnick, N.D. Jones (Eds), Program Flow Analysis: Theory and Applications, ch. 7, Prentice-Hall, Englewood Cliffs, NJ, 1981, pp. 189–234.
[50] H. Søndergaard, An application of abstract interpretation of logic programs: occur check reduction, in: Proceedings of the First European Symposium on Programming (ESOP), LNCS 213, Springer, Berlin, 1986, pp. 327–338.