Forward versus Backward Verification of Logic Programs

Andy King and Lunjin Lu

[1] Computing Laboratory, University of Kent, Canterbury, CT2 7NF, UK
[2] Department of Computer Science, Oakland University, Rochester, MI 48309, USA

Abstract. One recent development in logic programming has been the application of abstract interpretation to verify the partial correctness of a logic program with respect to a given set of assertions. One approach to verification is to apply forward analysis that starts with an initial goal and traces the execution in the direction of the control-flow to approximate the program state at each program point. This is often enough to verify that the assertions hold. The dual approach is to apply backward analysis to propagate properties of the allowable states against the control-flow to infer queries for which the program will not violate any assertion. This paper is a systematic comparison of these two approaches to verification. The paper reports some equivalence results that relate the relative power of various forward and backward analysis frameworks.

1 Introduction

Recently there has been growing awareness that abstract interpretation has an important rôle in both the verification and debugging of logic programs [6, 20, 26]. In this context, the programmer is typically equipped with an annotation language in which she/he can encode properties of the program state at various program points [26]. One approach to verification is to trace the program state in the direction of control-flow from an initial goal, using abstract interpretation to finitely represent and track the state. The program is deemed to be correct if all the assertions are satisfied whenever they are encountered; otherwise the program is potentially buggy. This is how forward analysis can be applied in logic program verification. The dual approach is to trace execution against the control-flow to infer those queries which ensure that the assertions are satisfied should they be encountered during execution [17, 23]. If the class of initial queries does not conform to those expected by the programmer, then the program is potentially buggy. This is how backward analysis can be applied in verification.

This paper is an examination and comparison of these two opposing approaches to verification. Specifically the paper compares forward analysis [3, 10, 14, 21, 27] to the backward analysis of [17] which uses the relative pseudo-complement operator [13] to trace information (weakest pre-conditions) against the control-flow. Every condensing domain possesses a pseudo-complement operator and it is always possible to synthesise a condensing domain from an arbitrary (downward-closed) domain by applying Heyting completion [13]. Examples of condensing domains include the class of positive Boolean functions, the relational type domain of [4], directional types [1, 13] and the two variable per inequality power domain presented in section 6 of this paper. This paper arose from a study of how the domain operations that arise in backward analysis effect precision. In fact, the equivalence results reported in this paper flow from the following practical, albeit technical, questions (annotated Q1, Q2 and Q3).

The backward analysis framework of [17] is parameterised by an abstract domain that is required to be condensing. Fixing the domain $\langle D, \unlhd \rangle$, fixes the join $\oplus$ and meet $\otimes$ operations that are used to model the merging of computation paths and the conjunction of constraints. Fixing D also fixes the relative pseudo-complement. The pseudo-complement of d_1 relative to d_2, denoted $d_1 \Rightarrow d_2$, delivers the weakest element of D whose conjunction with d_1 implies d_2, or more exactly, $d_1 \Rightarrow d_2 = \oplus\{d \in D \mid d \otimes d_1 \unlhd d_2\}$. The rôle of pseudo-complement is that if d_2 expresses a set of requirements that must hold after a constraint is added to the store, and d_1 models the constraint itself, then $d_1 \Rightarrow d_2$ expresses the requirements that must hold on the store before the constraint. In addition to these operations that are fixed by D, backward analysis employs an unusual operator $\forall_x : D \to D$, dubbed universal projection, that complements standard projection $\exists_x : D \to D$, hereafter named existential projection, in that $\forall_x(d) \unlhd d \unlhd \exists_x(d)$. Both projections are monotonic, both eliminate a variable x from a given abstraction d and both are used to restrain the size of abstractions; the fundamental difference is in the direction of approximation. The correctness of backward analysis relies on the property $\forall_x(d) \unlhd d$. Defining $\forall_x(d) = \perp$ (the strongest abstraction) for all $d \in D$ is sufficient for correctness but the resulting analysis is useless in that the class of queries inferred by the analysis is empty.

> Q1. This leads to the question of how to define $\forall_x$ so the precision of the resulting analysis compares favourably with that of forward analysis?

This paper reports that if $\langle D, \forall_x, D, \exists_x \rangle$ is a Galois connection, then a surprising equivalence is established between forward and backward analysis so that the power of backward analysis exactly matches that of forward analysis for verification. On the practical side, it means that backward analysis need not be applied, if forward analysis cannot verify that a given query satisfies the assertions. Conversely, if an initial query is not inferred by backward analysis, then it follows that forward analysis cannot infer that the query satisfies the assertions.

Another issue relates to the fixpoint engines that are used in forward analysis. Since these engines vary in complexity, another question relates to which engine in the precision and tractability continuum is best suited to verification.

> Q2. This begs the question of whether a polyvariant analysis (that maintains a set of call and answer patterns for each predicate) has any precision advantage over a monovariant analysis (that records a single call and answer pattern pair for each predicate) for the task of verification?

The answer to this question is negative for condensing domains provided that $\exists_x$ is additive, that is, $\exists_x(\oplus E) = \oplus\{\exists_x(d) \mid d \in E\}$ whenever $E \subseteq D$.

Since one way of constructing a condensing domain, is to lift a base domain $\langle D, \trianglelefteq \rangle$ to its power domain $\langle D', \trianglelefteq' \rangle$, then the answer to Q1 raises the question of what properties are required of D for $\langle D', \forall'_x, D', \exists'_x \rangle$ to be Galois connection to thereby guarantee equivalence between the frameworks.

Q3. Specifically if $\forall'_x$ and $\exists'_x$ are synthesised from $\forall_x$ and $\exists_x$ (in the natural way) then what does this require of D?

This paper shows that the only way to satisfy the Galois connection on D' is to engineer $\forall_x$ and $\exists_x$ so that $\langle D, \forall_x, D, \exists_x \rangle$ is also a Galois connection.

The paper is structured as follows. Section 2 presents an operational semantics for verification. Sections 3, 4 and 5 compare top-down, condensing and backward framework for the task of verification. Section 6 discuses the rôle of power domains and section 7 the related work. Finally section 8 concludes.

2 Operational semantics

To precisely spell out the relationship between various forward and backward analysis frameworks, a formal language is required to specify both the operational semantics and the frameworks themselves. This section, and the proceeding section, introduces this necessary formalism.

Let Con be a set of constraints that is pre-ordered by entailment $\models$ and includes equational constraints of the form $\boldsymbol{x} = \boldsymbol{y}$ where $\boldsymbol{x}$ and $\boldsymbol{y}$ are tuples of variables. The set of order-ideals of Con is defined $Ord = \{C \subseteq Con \mid C = \downarrow(C)\}$ where $\downarrow(C) = \{c \in Con \mid \exists c' \in C.c \models c'\}$. The basic semantic domain is the complete lattice $\langle Ord, \subseteq, \cup, \cap, Con, \emptyset \rangle$. To observe violations of assertions, Ord is augmented with a new top element $\top$ to obtain $\widehat{Ord} = Ord \cup \{\top\}$. The ordering $\subseteq$ extends to $\widehat{Ord}$ by $C \subseteq \top$ for all $C \in \widehat{Ord}$ and the operators $\cup$ and $\cap$ extend analogously. It is useful, however, to define a variant of $\cap$, denoted $\widehat{\cap}$, that ensures that $\top$ can never be annulled. This is defined $C \widehat{\cap} \top = \top \widehat{\cap} C = \top$ for all $C \in \widehat{Ord}$ and $C_1 \widehat{\cap} C_2 = C_1 \cap C_2$ for all $C_1, C_2 \in Ord$.

The verification problem is formulated in terms of an additional (abstract) domain $\langle D, \trianglelefteq, \oplus, \otimes, \top, \bot \rangle$ equipped with an abstraction map $\alpha : Ord \to D$ and a concretisation map $\gamma : D \to Ord$ that interpret elements of D. A constraint logic program is annotated with assertions over D and a program is deemed to be correct if the assertions are satisfied whenever they are reached. The problem is to decide whether a given program is correct for some input. For brevity, programs are expressed (concurrent constraint style) using $\mathsf{ask}(d)$ where $d \in D$ to distinguish an assertion from a conventional store write that is denoted $\mathsf{tell}(c)$ where $c \in Con$. In what follows, P denotes a program and A an agent, that is, $P ::= \epsilon \mid p(\boldsymbol{x}) \leftarrow A \mid P.P'$ and $A ::= \mathsf{ask}(d) \mid \mathsf{tell}(c) \mid A_1, A_2 \mid \sum_{i=1}^{n} A_i \mid p(\boldsymbol{x})$. Due to the presence of an explicit choice operator predicates can be assumed to be defined with exactly one definition of the form $p(\boldsymbol{x}) \leftarrow A$ without any loss of generality. Let Age_P denote the set of agents in a program P that is closed under renaming and let $Poly_P$ denote the function space $Age_P \to \widehat{Ord} \to \widehat{Ord}$. The ordering $\subseteq$ over $\widehat{Ord}$ lifts $Poly_P$ point-wise by $f_1 \sqsubseteq f_2$ iff $f_1[\![A]\!](C) \subseteq f_2[\![A]\!](C)$ for all $A \in Age_P$ and $C \in \widehat{Ord}$. In fact $\langle Poly_P, \sqsubseteq, \sqcup, \sqcap\rangle$ is a complete lattice where $f_1 \sqcup f_2 = \lambda A.\lambda d.(f_1[\![A]\!](d) \cup f_2[\![A]\!](d))$ and $\sqcap$ is defined analogously.

Definition 1 (operational semantics). The operator $\mathcal{F}_P : Poly_P \to Poly_P$ is defined $\mathcal{F}_P(f) = f'$ where $f, f' \in Poly_P$ and:

$$\begin{aligned}
f'[\![\mathsf{ask}(d)]\!] &= \lambda C.\text{if } \alpha(C) \trianglelefteq d \text{ then } C \text{ else } \top \\
f'[\![\mathsf{tell}(c)]\!] &= \lambda C.{\downarrow}(\{c\})\widehat{\cap} C \\
f'[\![A_1, A_2]\!] &= \lambda C.f[\![A_2]\!](f[\![A_1]\!](C)) \\
f'[\![\textstyle\sum_{i=1}^{n} A_i]\!] &= \lambda C.\cup_{i=1}^{n} f[\![A_i]\!](C) \\
f'[\![p(\boldsymbol{x})]\!] &= \lambda C.\cup\{f[\![A]\!](\Downarrow(\{\boldsymbol{x} = \boldsymbol{y}\})\widehat{\cap} C) \mid p(\boldsymbol{y}) \leftarrow A \ll_{p(\boldsymbol{x}),C} P\}
\end{aligned}$$

Note that the composition operator , is sequential thus control is left-to-right. Note too that choice occurs both explicitly within the construct $\sum_{i=1}^{n} A_i$ and implicitly within renaming. The notation $p(\boldsymbol{y}) \leftarrow A \ll_{p(\boldsymbol{x}),C} P$ indicates that $p(\boldsymbol{y}) \leftarrow A$ is a renaming of a definition in P such that $var(p(\boldsymbol{y}) \leftarrow A) \cap (var(p(\boldsymbol{x})) \cup var(C)) = \emptyset$ where $var(o)$ is the set of variables in the object o. Since $\mathcal{F}_P$ is monotonic and $Poly_P$ is a complete lattice, $\mathsf{lfp}(\mathcal{F}_P)$ exists and the verification problem can be formally stated as the problem of characterising the following set of atomic queries:

Definition 2. $\mathcal{F}[\![P]\!] = \{\langle p(\boldsymbol{x}), C\rangle \mid \mathsf{lfp}(\mathcal{F}_P)[\![p(\boldsymbol{x})]\!](C) \neq \top\}$

3 Verification with a top-down framework

An analysis for approximating $\mathcal{F}[\![P]\!]$ can be constructed by mimicking concrete operations over Ord with abstract operations over D and applying the projection operator $\exists_x : D \to D$ to finitely bound the number of variables. The operator $\exists_x$ is assumed to comply with the rules $d \trianglelefteq \exists_x(d)$, $\exists_x(d_1) \trianglelefteq \exists_x(d_2)$ if $d_1 \trianglelefteq d_2$, $\exists_x(\exists_y(d)) = \exists_y(\exists_x(d))$ and $\exists_x(d_1) \otimes \exists_x(d_2) = \exists_x(d_1 \otimes \exists_x(d_2))$. The last rule is useful within itself as well as implying that $\exists_x$ is idempotent, that is, $\exists_x(\exists_x(d)) = \exists_x(d)$. Finally, $\exists_x$ is also required to eliminate a variable, hence $x \notin var(\exists_x(d))$. For brevity, let $\exists_o(d) = (\exists_{x_1} \ldots \exists_{x_n})(d)$ where $var(o) = \{x_1, \ldots, x_n\}$ and let $\overline{\exists}_o(d) = \exists_{var(d)\setminus var(o)}(d)$. D is assumed to contain elements of the form $\boldsymbol{x} = \boldsymbol{y}$ to model argument passing. To express renaming, let $\rho_{\boldsymbol{x},\boldsymbol{y}}(d) = \exists_{\boldsymbol{x}}((\boldsymbol{x} = \boldsymbol{y}) \otimes d)$. Suppose that $\langle \widehat{D}, \trianglelefteq\rangle$ is augmented with $\top$ in a similar fashion to $\langle \widehat{Ord}, \subseteq\rangle$. To trace violations of assertions, a variant of $\otimes$, denoted $\widehat{\otimes}$, is defined such that $d\widehat{\otimes}\top = \top\widehat{\otimes}d = \top$ for all $d \in \widehat{D}$ and $d_1\widehat{\otimes}d_2 = d_1 \otimes d_2$ for all $d_1, d_2 \in D$.

The semantic equations for a polyvariant, top-down framework are given overleaf. The map $\mathcal{F}_P^D$ operates over $Poly_P^D \to Poly_P^D$ where $Poly_P^D = Age_P \to \widehat{D} \to \widehat{D}$. Thus if $f \in Poly_P^D$, f associates each agent A with a map between input (the call pattern) and output (the answer pattern). The ordering $\trianglelefteq$ over $\widehat{D}$ induces an ordering on $Poly_P^D$ by $f_1 \trianglelefteq f_2$ iff $f_1[\![A]\!](d) \trianglelefteq f_2[\![A]\!](d)$ for all $d \in \widehat{D}$ and $A \in Age_P$. Moreover $\langle Poly_P^D, \trianglelefteq, \oplus, \otimes\rangle$ is a complete lattice where $f_1 \circledast f_2 = \lambda A.\lambda d.(f_1[\![A]\!](d) \circledast f_2[\![A]\!](d))$ and $\circledast \in \{\oplus, \otimes\}$.

Definition 3 (top-down framework). The operator $\mathcal{F}_P^D : Poly_P^D \to Poly_P^D$ is defined $\mathcal{F}_P^D(f) = f'$ such that:

$$\begin{aligned}
f'[\![\mathsf{ask}(d')]\!] &= \lambda d.\text{if } d \trianglelefteq d' \text{ then } d \text{ else } \top \\
f'[\![\mathsf{tell}(c)]\!] &= \lambda d.\alpha(\downarrow(\{c\}))\widehat{\otimes} d \\
f'[\![A_1, A_2]\!] &= \lambda d.f[\![A_2]\!](f[\![A_1]\!](d)) \\
f'[\![\textstyle\sum_{i=1}^n A_i]\!] &= \lambda d.\oplus_{i=1}^n f[\![A_i]\!](d) \\
f'[\![p(\boldsymbol{x})]\!] &= \lambda d.\rho_{\boldsymbol{y},\boldsymbol{x}}(\overline{\exists}_{\boldsymbol{y}}(f[\![A]\!](\rho_{\boldsymbol{x},\boldsymbol{y}}(\overline{\exists}_{\boldsymbol{x}}(d)))))\widehat{\otimes} d \text{ where } p(\boldsymbol{y}) \leftarrow A \ll_{p(\boldsymbol{x})} P
\end{aligned}$$

Note how the use of projection eliminates the requirement for considering each definition renaming separately. The functional defined in the semantics can be interpreted as a formalisation of the top-down framework of Bruynooghe [3] that is widely used in analysis and specialisation because of its precision and polyvariance (different calls to the same predicate are analysed separately). Since $\mathcal{F}_P^D$ is monotonic and $Poly_P^D$ is a complete lattice, $\mathsf{lfp}(\mathcal{F}_P^D)$ exists. However, efficient implementations of the Bruynooghe framework, such as GENA [10], PLAI [14] and GAIA [21], only compute $\mathsf{lfp}(\mathcal{F}_P^D)$ in a partial query-directed fashion. The verification problem can be tackled in the abstract setting by characterising the following set of atomic queries:

Definition 4. $\mathcal{F}^D[\![P]\!] = \{\langle p(\boldsymbol{x}), C\rangle \mid \mathsf{lfp}(\mathcal{F}_P^D)[\![p(\boldsymbol{x})]\!](\alpha(C)) \neq \top\}$

The following proposition states this is a safe, albeit possibly imprecise, strategy for solving the verification problem. The proof is straightforward.

Proposition 1. $\mathcal{F}^D[\![P]\!] \subseteq \mathcal{F}[\![P]\!]$

Observe that Age_P is finite (modulo renaming) since P is finite. Therefore a computable analysis can be constructed by appropriately factoring out renaming provided that D is finite. The proof for the following lemma is straightforward.

Lemma 1. If $d_1 \trianglelefteq d_2$ then $\mathsf{lfp}(\mathcal{F}_P^D)[\![A]\!](d_1) \trianglelefteq \mathsf{lfp}(\mathcal{F}_P^D)[\![A]\!](d_2)$.

4 Verification with condensing domain

It has long been realised that if an abstract domain is condensing [16, 19], then a goal-dependent analysis can be performed in a goal-independent way without incurring a loss in precision. Langen observed that if a compound goal (the body of a clause) returns an answer pattern of d when invoked with a call pattern of $\top$, then the compound goal will return an answer pattern of $d \wedge d'$ when invoked with a call pattern of d' [19][Lemma 9]. Jacobs and Langen exploited this to factor out repeated computation in a polyvariant, top-down framework [16, 19]. Other condensing frameworks were monovariant [27]; first computing one success pattern for each predicate with a goal-independent analysis then, second, deriving one call pattern for each predicate in a goal-dependent fashion as directed by an initial query. Quite apart from their efficiency, monovariant condensing frameworks are attractive because of their simplicity and modularity [27] and therefore it is interesting to compare a monovariant, condensing framework against a polyvariant, top-down framework for the purposes of verification.

Semantic equations for a monovariant condensing framework are given below. Success and call patterns are calculated by $\mathcal{S}_P^D$ and $\mathcal{C}_P^D$ respectively. Both maps operate over the function space $Mono_P^D = Age_P \to D$ where each $f \in Mono_P^D$ assigns a domain element to each agent of P. The ordering $\trianglelefteq$ over D induces a point-wise ordering on $Mono_P^D$ by $f_1 \trianglelefteq f_2$ iff $f_1[\![A]\!] \trianglelefteq f_2[\![A]\!]$ for all $A \in Age_P$. Moreover $f_1 \circledast f_2 = \lambda A.(f_1[\![A]\!] \circledast f_2[\![A]\!])$ where $\circledast \in \{\oplus, \otimes\}$. In fact $\langle Mono_P^D, \trianglelefteq, \oplus, \otimes\rangle$ is a complete lattice with $\lambda A.\bot$ and $\lambda A.\top$ for bottom and top.

Definition 5 (condensing framework). The operators $\mathcal{S}_P^D : Mono_P^D \to Mono_P^D$ and $\mathcal{C}_P^D : Mono_P^D \to Mono_P^D$ are defined $\mathcal{S}_P^D(f) = f'$ and $\mathcal{C}_P^D(g) = g'$ such that:

$$\begin{aligned} f'[\![\mathsf{ask}(d)]\!] &= \top \\ f'[\![\mathsf{tell}(c)]\!] &= \alpha(\downarrow(\{c\})) \\ f'[\![A_1, A_2]\!] &= f[\![A_1]\!] \otimes f[\![A_2]\!] \\ f'[\![\textstyle\sum_{i=1}^n A_i]\!] &= \oplus_{i=1}^n f[\![A_i]\!] \\ f'[\![p(\boldsymbol{x})]\!] &= \rho_{\boldsymbol{y},\boldsymbol{x}}(\overline{\exists}_{\boldsymbol{y}}(f[\![A]\!])) \end{aligned} \qquad \begin{aligned} g'[\![A_1]\!] &= g[\![A_1, A_2]\!] \\ g'[\![A_2]\!] &= g[\![A_1, A_2]\!] \otimes \mathsf{lfp}(\mathcal{S}_P^D)[\![A_1]\!] \\ g'[\![A_i]\!] &= g[\![\textstyle\sum_{i=1}^n A_i]\!] \\ g'[\![A]\!] &= \rho_{\boldsymbol{x},\boldsymbol{y}}(\overline{\exists}_{\boldsymbol{x}}(g[\![p(\boldsymbol{x})]\!])) \end{aligned}$$

where $p(\boldsymbol{y}) \leftarrow A \ll_{p(\boldsymbol{x})} P$

The equations of $\mathcal{C}_P^D$ detail how to propagate the call pattern for an agent to its sub-agents; equations are not required for $\mathsf{ask}(d)$ and $\mathsf{tell}(c)$ since they do not invoke sub-agents. The verification problem can be formalised in this setting as the problem of computing the class of atomic queries that lead to call patterns which do not violate the $\mathsf{ask}(d)$ assertions.

Definition 6. $\mathcal{C}^D[\![P]\!] = \left\{ \langle p(\boldsymbol{x}), C\rangle \;\middle|\; \begin{array}{l} f \in \mathsf{fp}(\mathcal{C}_P^D) \ \wedge \ \alpha(C) \trianglelefteq f[\![p(\boldsymbol{x})]\!] \ \wedge \\ \forall\, \mathsf{ask}(d) \in Age_P. f[\![\mathsf{ask}(d)]\!] \trianglelefteq d \end{array} \right\}$

More exactly, to verify the correctness of a concrete atomic query $\langle p(\boldsymbol{x}), C\rangle$, it is sufficient to find a fixpoint $f \in \mathsf{fp}(\mathcal{C}_P^D)$ (any fixpoint) with a call to $p(\boldsymbol{x})$ that is not stronger than $\alpha(C)$ yet with a call for each $\mathsf{ask}(d)$ agent that is not weaker than d. Since $\mathcal{C}_P^D$ is continuous such a fixpoint, if one exists, can be computed by assigning $g_0 = \lambda A.(\text{if } A = p(\boldsymbol{x}) \text{ then } \alpha(C) \text{ else } \bot)$ and calculating $g_{i+1} = \mathcal{C}_P^D(g_i)$. Then $g = \oplus_i g_i$ is the least fixpoint of $\mathcal{C}_P^D$ such that $\alpha(C) \trianglelefteq g[\![p(\boldsymbol{x})]\!]$. Computation can be aborted if $g_i[\![\mathsf{ask}(d)]\!] \not\trianglelefteq d$ for some $\mathsf{ask}(d) \in Age_P$ since then no fixpoint of $\mathcal{C}_P^D$ can satisfy both the $p(\boldsymbol{x})$ and $\mathsf{ask}(d)$ requirements and hence verify $\langle p(\boldsymbol{x}), C\rangle$. The following lemmas explain how $\mathsf{lfp}(\mathcal{S}_P^D)$ can characterise $\mathsf{lfp}(\mathcal{F}_P^D)$.

Lemma 2. If $\mathsf{lfp}(\mathcal{F}_P^D)[\![A]\!](d) \neq \top$ then $\mathsf{lfp}(\mathcal{F}_P^D)[\![A]\!](d) \trianglelefteq d \otimes \mathsf{lfp}(\mathcal{S}_P^D)[\![A]\!]$.

Lemma 3. If D is a complete Heyting algebra (cHa) then $d \otimes \mathsf{lfp}(\mathcal{S}_P^D)[\![A]\!] \trianglelefteq \mathsf{lfp}(\mathcal{F}_P^D)[\![A]\!](d)$.

A domain D is a cHa if it is (completely) meet-distributive, that is, $d \otimes (\oplus_{i\in I} d_i) = \oplus_{i\in I}(d \otimes d_i)$ whenever $d \in D$ and $\{d_i \mid i \in I\} \subseteq D$ where I is some index set [2][Chapter IX, Theorem 15]. Lemmas 2 and 3 are closely related to Theorem 8.2

of [13] which can be interpreted as stating $\mathsf{lfp}(\mathcal{F}_P^D)[\![A]\!](d) = d \otimes \mathsf{lfp}(\mathcal{F}_P^D)[\![A]\!](\top)$ for a program P without $\mathsf{ask}(d)$ agents. Theorem 8.2 does not stipulate any requirements on existential projection since the semantics of [13] does not apply this operator. Theorem 1 and theorem 2 (with its corollary) flow from the lemmas and state conditions on the domain operations for a monovariant, condensing framework to match the verification power of a polyvariant, top-down framework.

Theorem 1 (precision). If D is a cHa and $\exists_x$ is additive then $\mathcal{F}^D[\![P]\!] \subseteq \mathcal{C}^D[\![P]\!]$.

Theorem 2. $\mathcal{C}^D[\![P]\!] \subseteq \mathcal{F}^D[\![P]\!]$.

Corollary 1 (correctness). $\mathcal{C}^D[\![P]\!] \subseteq \mathcal{F}[\![P]\!]$.

Proof. By theorem 2, $\mathcal{C}^D[\![P]\!] \subseteq \mathcal{F}^D[\![P]\!]$ and by proposition 1, $\mathcal{F}^D[\![P]\!] \subseteq \mathcal{F}[\![P]\!]$.

The correctness of the program depends only on those calls that actually arise in a derivation from the initial query. The functional $\mathcal{F}_P^D$ that defines the top-down framework, on the other hand, maps an arbitrary call pattern to its answer pattern. Thus the domain is augmented with $\top$ to record whether any of the call patterns that occur in a derivation violate any of the assertions. This domain element is not required in the condensing approach because all the calls that arise in a derivation are merged and recorded. This approach to verification, however, is only guaranteed to be as precise as the top-down scheme if D is a cHa and $\exists_x$ is additive. The following example illustrates one domain which satisfies these properties that has been widely applied in verification.

Example 1. Let $Term$ denote the set of terms and $\wp^{\downarrow}(Term)$ denote the set of term sets that are closed under instantiation. Let $\Pi \subseteq \wp^{\downarrow}(Term)$ denote a finite set of primitive types and suppose $Term \in \Pi$. To enrich Π with dependencies, let $Type_X = \{x \subseteq \pi \mid x \in X \wedge \pi \in \Pi\} \cup \{\tau_1 \circledast \tau_2 \mid \circledast \in \{\wedge, \vee, \rightarrow\} \wedge \tau_1, \tau_2 \in Type_X\}$. The construction is completed by augmenting $Type_X$ with a bottom element $\perp$. A mapping $\theta : X \rightarrow \Pi$ assigns a truth value to each $\tau \in Type_X$ as follows: $\theta(\perp)$ is always false, $\theta(x \subseteq \pi) \iff \theta(x) \subseteq \pi$ and $\theta(\tau_1 \circledast \tau_2) \iff \theta(\tau_1) \circledast \theta(\tau_2)$. Then $\langle Type_X, \models, \vee, \wedge \rangle$ is a complete lattice where $\tau_1 \models \tau_2$ iff $\theta(\tau_1) \rightarrow \theta(\tau_2)$ holds for all $\theta : X \rightarrow \Pi$. In fact it can be shown that this lattice is a cHa. The concretisation map $\gamma_{Type} : Type_X \rightarrow \wp(Eqn)$ is defined $\gamma_{Type}(f) = \{E \in Eqn \mid \alpha(\sigma) \models f \wedge \sigma \in mgu(E)\}$ where

$$\alpha(\sigma) = \bigwedge \left\{ y \subseteq \pi \leftrightarrow \wedge_{i=1}^{n} y_i \subseteq \pi_i \;\middle|\; \begin{array}{ll} y \in dom(\sigma) \ \wedge \ var(\sigma(y)) = \{y_i\}_{i=1}^{n} & \wedge \\ \kappa \in var(\sigma(y)) \rightarrow Term & \wedge \\ \kappa(\sigma(y)) \in \pi \iff \wedge_{i=1}^{n} \kappa(y_i) \in \pi_i & \end{array} \right\}$$

Moreover, the abstraction map $\alpha_{Type} : \wp(Eqn) \rightarrow Type_X$ is defined as $\alpha_{Type}(S) = \wedge\{f \in Type_X \mid S \subseteq \gamma_{Type}(f)\}$. $Type_X$ includes types of the form $(\wedge_{i=1}^{n} x_i \subseteq \pi_i) \rightarrow (\wedge_{i=1}^{n} x_i \subseteq \pi_i')$ where $\pi_i, \pi_i' \in \Pi$ that can capture the input and output types of an n-ary predicate [13] in a similar fashion to directional types [1] that are used in type checking and verification. Finally define $\exists_x(\tau) = \vee\{\{x \mapsto \pi\}(\tau) \mid \pi \in \Pi\}$ and let $\pi \in \Pi$, $\{\tau_i \mid i \in I\} \subseteq Type_X$ where I is some index set. Observe that $\{x \mapsto \pi\}(\vee_{i \in I} \tau_i) = \vee_{i \in I}\{x \mapsto \pi\}\tau_i$. It follows that $\exists_x$ is additive, hence the condensing framework is applicable.

5 Verification with a backward framework

The semantic equations for (a reformulation of) the backward analysis framework of [17] are given below. The key idea in $\mathcal{B}_P^D$ is embedded in the semantic equation for sequenced agents A_1, A_2. The problem is to find the weakest $d \in D$ (which describes the largest set of states) which guarantees that the agent A_1, A_2 will not violate an assertion. The problem, in fact, is to compute such a d given $d_1, d_2 \in D$ which ensures that A_1 and A_2 will not violate their assertions. One observation that underpins backward analysis is that $d \otimes \mathsf{lfp}(\mathcal{S}_P^D)[\![A_1]\!]$ describes that state immediately before the execution of A_2, hence A_2 will not violate an assertion if $d \otimes \mathsf{lfp}(\mathcal{S}_P^D)[\![A_1]\!] \trianglelefteq d_2$. This will hold if $d = d_1 \otimes (\mathsf{lfp}(\mathcal{S}_P^D)[\![A_1]\!] \Rightarrow d_2)$ where $\Rightarrow$ denotes the pseudo-complement in D (which exists whenever D is a cHa). Because $d \trianglelefteq d_1$, it follows that A_1 must satisfy its assertions. Since $d \models (\mathsf{lfp}(\mathcal{S}_P^D)[\![A_1]\!] \Rightarrow d_2)$ it follows from the axioms of a cHa [29], that $d \otimes \mathsf{lfp}(\mathcal{S}_P^D)[\![A_1]\!] \trianglelefteq d_2$, hence A_2 cannot violate an assertion either. The insight behind the $\Rightarrow$ application comes by using the axioms of a cHa to rewrite $d = d_1 \otimes (\mathsf{lfp}(\mathcal{S}_P^D)[\![A_1]\!] \Rightarrow d_2)$ as $d = d_1 \otimes ((d_1 \otimes \mathsf{lfp}(\mathcal{S}_P^D)[\![A_1]\!]) \Rightarrow d_2)$. Then $(d_1 \otimes \mathsf{lfp}(\mathcal{S}_P^D)[\![A_1]\!]) \Rightarrow d_2$ is the *weakest* element of D whose meet with $d_1 \otimes \mathsf{lfp}(\mathcal{S}_P^D)[\![A_1]\!]$ implies d_2. Thus d is the *weakest* element of D which ensures that A_1 and A_2 satisfy their assertions. The question is, of course, whether this tactic for propagating requirements leads to a useful approach to verification.

Definition 7 (backward framework). The operator $\mathcal{B}_P^D : Mono_P^D \to Mono_P^D$ is defined $\mathcal{B}_P^D(f) = f'$ such that:

$$\begin{aligned}
f'[\![\mathsf{ask}(d)]\!] &= d \\
f'[\![\mathsf{tell}(c)]\!] &= \top \\
f'[\![A_1, A_2]\!] &= f[\![A_1]\!] \otimes (\mathsf{lfp}(\mathcal{S}_P^D)[\![A_1]\!] \Rightarrow f[\![A_2]\!]) \\
f'[\![\textstyle\sum_{i=1}^n A_i]\!] &= \otimes_{i=1}^n f[\![A_i]\!] \\
f'[\![p(\boldsymbol{x})]\!] &= \rho_{\boldsymbol{y},\boldsymbol{x}}(\overline{\forall}_{\boldsymbol{y}}(f[\![A]\!])) \text{ where } p(\boldsymbol{y}) \leftarrow A \ll_{p(\boldsymbol{x})} P
\end{aligned}$$

Universal projection $\forall_x(d) \trianglelefteq d$ is required to satisfy $\forall_x(d) \trianglelefteq d$ for all $d \in D$ for reasons of correctness [17]. This is because of the way it is used to propagate requirements over procedure boundaries; if d describes a set of states for which an agent A does not violate an assertion, then so does $\forall_x(d)$ since it represents a subset of those states. The $\overline{\forall}$ operator is defined in an analogous fashion to $\overline{\exists}$. Like $\mathcal{C}_P^D$, $\mathcal{B}_P^D$ requires $\mathsf{lfp}(\mathcal{S}_P^D)$ to be pre-computed. Like $\mathcal{C}_P^D$, the map $\mathcal{B}_P^D$ operates over $Mono_P^D$ and hence is monovariant. Unlike $\mathcal{C}_P^D$, repeated application yields a decreasing sequence. In fact $\mathcal{B}_P^D$ is co-continuous, thus the sequence $f_0 = \top$, $f_{i+1} = \mathcal{B}_P^D(f_i)$ converges onto the greatest fixpoint of $\mathcal{B}_P^D$, that is, $\mathsf{gfp}(\mathcal{B}_P^D) = \otimes_i f_i$. The following definition states how $\mathsf{gfp}(\mathcal{B}_P^D)$ can be interpreted for the purposes of verification.

Definition 8. $\mathcal{B}^D[\![P]\!] = \{\langle p(\boldsymbol{x}), C\rangle \mid \alpha(C) \trianglelefteq \mathsf{gfp}(\mathcal{B}_P^D)[\![p(\boldsymbol{x})]\!]\}$

The following theorems and corollary state conditions under which this backward approach to verification coincides with forward verification. These equivalence results rest crucially, and perhaps surprisingly, on the relationship between the projection operators used within forward and backward analysis.

Theorem 3 (precision). If D is a cHa and $\langle D, \forall_x, D, \exists_x \rangle$ is a Galois connection, then $\mathcal{F}^D[\![P]\!] \subseteq \mathcal{B}^D[\![P]\!]$

Theorem 4. If D is a cHa, $\exists_X(\bot) = \bot$ and $\langle D, \forall_x, D, \exists_x \rangle$ is a Galois connection, then $\mathcal{B}^D[\![P]\!] \subseteq \mathcal{F}^D[\![P]\!]$.

Corollary 2 (correctness). If D is a cHa, $\exists_X(\bot) = \bot$ and $\langle D, \forall_x, D, \exists_x \rangle$ is a Galois connection, then $\mathcal{B}^D[\![P]\!] \subseteq \mathcal{F}[\![P]\!]$

The proofs of these theorems (see the technical report version of this paper [18]) rely on properties that flow from the Galois connection. The proof of theorem 3 relies on two properties of universal quantification – the monotonicity of $\forall_x$ and the property that $d = \forall_x(d)$ whenever $d = \exists_x(d)$. Since $\exists_x(d) = \exists_x(\exists_x(d))$ the latter property ensures that $\exists_x(d) = \forall_x(\exists_x(d))$ and since $d \trianglelefteq \exists_x(d)$ it follows that $d \trianglelefteq \forall_x(\exists_x(d))$, that is, that $\forall_x \circ \exists_x$ is extensive. On the other hand, the proof of theorem 4 relies on the property that $\forall_x(d) \trianglelefteq d$. From this it follows that $\exists_x(\forall_x(d)) = \forall_x(d) \trianglelefteq d$, that is, that $\exists_x \circ \forall_x$ is reductive. The monotonicity of $\forall_x$ combined with the monotonicity of $\exists_x$ and the extensive and reductive properties of $\forall_x \circ \exists_x$ and $\exists_x \circ \forall_x$, implies that $\langle D, \forall_x, D, \exists_x \rangle$ is a Galois connection [7]. Thus the Galois connection requirement cannot be relaxed. Interestingly, the direction of approximation in $\forall_x$ and $\exists_x$ suggests the existance of a Galois connection: the adjoint of an upper closure operator ($\exists_x$) is a lower closure operator ($\forall_x$). Curiously, $\exists_X(\bot) = \bot$ is required to guarantee that $\forall_x$ eliminates the variable x for each $x \in X$; specifically $\exists_X(\bot) = \bot$ ensures $x \notin var(\forall_x(d))$.

A Galois connection gives a systematic way of synthesising $\forall_x$ from $\exists_x$, that is, $\forall_x(d) = \oplus\{d' \in D \mid \exists_x(d') \trianglelefteq d\}$ [7, 17]. It also ensures that $\exists_x$ is additive [7], thereby satisfying the condensing requirement. The equivalence it induces, also provides a simple tactic to establishing safety which avoids arguments that involve both state abstraction and reversed information flow [17]. In fact Hughes and Launchbury [15] argue that ideally the direction of an analysis should be reversed without reference to the concrete semantics. Indeed, the equivalence between backward and forward analysis, means that the correctness of backward analysis follows immediately from that of forward analysis. The following examples illustrate some domains for which $\langle D, \forall_x, D, \exists_x \rangle$ is a Galois connection.

Example 2. Let $Bool_X$ denote the Boolean functions over a set of variables X. The domain Pos_X is defined by $Pos_X = \{\bot\} \cup \{f \in Bool_X \mid (\wedge X) \models f\}$. The lattice $\langle Pos_X, \models, \vee, \wedge, 1, \bot \rangle$ is finite. Each element of Pos_X is interpreted as a set of equation sets by the concretisation map $\gamma_{Pos} : Pos_X \rightarrow \wp(Eqn)$ where $\gamma_{Pos}(f) = \{E \in Eqn \mid \alpha(\theta) \models f \wedge \theta \in mgu(E)\}$ and $\alpha(\theta) = \wedge\{y \leftrightarrow \wedge var(\theta(y)) \mid y \in dom(\theta)\}$. The abstraction map $\alpha_{Pos} : \wp(Eqn) \rightarrow Pos_X$ is defined as $\alpha_{Pos}(S) = \wedge\{f \in Pos_X \mid S \subseteq \gamma_{Pos}(f)\}$. In forward analysis, existential projection is conventionally defined by Schröder elimination as $\exists_x(f) = f[x \mapsto 1] \vee f[x \mapsto 0]$. To obtain a Galois connection, define universal projection by $\forall_x(f) = f'$ if $f' \in Pos$ otherwise $\forall_x(f) = \bot$ where $f' = f[x \mapsto 0] \wedge f[x \mapsto 1]$. Although $f[x \mapsto 0] \vee f[x \mapsto 1] \in Pos_X$ for any $f \in Pos_X$, $f[x \mapsto 0] \wedge f[x \mapsto 1] \notin Pos_X$ for some $f \in Pos$. Consider, for instance,

$f = (x \leftarrow y)$. Note that $f \models \exists_x(f) = \forall_x(\exists_x(f))$, hence $\forall_x \circ \exists_x$ is extensive. Moreover, if $\forall_x(f) = \bot$ then $\exists_x(\forall_x(f)) = \bot \models f$. Otherwise $f \models \forall_x(f) = \exists_x(\forall_x(f))$. Thus $\exists_x \circ \forall_x$ is reductive. Since $\exists_x$ and $\forall_x$ are monotonic, $\langle Pos_X, \forall_x, Pos_X, \exists_x \rangle$ is a Galois connection. The pseudo-complement $\Rightarrow$ is $\rightarrow$ for Pos_X.

Example 3. The Galois connection property does not uniquely define the existential and projection operators for a given domain. For example, for Pos_X consider $\exists_x(f) = 1$ and $\forall_x(f) = \bot$. Then $f \models 1 = \exists_x(\forall_x(f))$ and $\forall_x(\exists_x(f)) = \bot \models f$, and $\langle Pos_X, \forall_x, Pos_X, \exists_x \rangle$ is again a Galois connection.

Example 4. An intriguing non-example for Pos_X is obtained by defining:

$$\exists_x(f) = \begin{cases} f \text{ if } x \notin var(f) \\ 1 \text{ otherwise} \end{cases} \qquad \forall_x(f) = \begin{cases} f \text{ if } x \notin var(f) \\ \bot \text{ otherwise} \end{cases}$$

Now compare a forward analysis that uses $\exists_x$ against a backward analysis that applies both $\exists_x$ and $\forall_x$ for a program P that consists of two definitions $p(x) \leftarrow \mathsf{ask}(x \vee y)$ and $q(x) \leftarrow \mathsf{ask}(x)$. Then $\mathsf{lfp}(\mathcal{F}_P^D)[\![p(\boldsymbol{x})]\!](x) = x \neq \top$ however $\mathsf{gfp}(\mathcal{B}_P^D)[\![p(x)]\!] = \bot$. Dually $\mathsf{lfp}(\mathcal{F}_P^D)[\![q(\boldsymbol{x})]\!](x \wedge y) = \top$ whereas $\mathsf{gfp}(\mathcal{B}_P^D)[\![q(x)]\!] = x$. Since Pos_X is a cHa, from theorems 3 and 4 it follows that $\langle Pos_X, \forall_x, Pos_X, \exists_x \rangle$ is not a Galois connection although $\forall_x \circ \exists_x$ is extensive and $\exists_x \circ \forall_x$ is reductive. In fact equivalence is lost because neither $\exists_x$ nor $\forall_x$ are monotonic as is witnessed by $\exists_x(x \wedge y) = 1 \not\models y = \exists_x(y)$ and $\forall_x(y) = y \not\models \bot = \exists_x(x \vee y)$.

6 Verification with a power domain

One classic way [7] of enriching an abstract domain is to apply a power domain construction in which the elements of the new domain correspond to sets of elements in the old domain. The rational for this construction is usually to improve the precision of join that is required to merge abstractions arising along different computational paths. However, as originally pointed out in [19], it also provides a mechanism for synthesising a domain that is condensing. This approach is useful if the Heyting completion [13] of a domain is unknown. Thus consider a power domain constructed from an abstract domain $\langle D, \trianglelefteq, \oplus, \otimes \rangle$ that is a complete lattice. The ordering $\trianglelefteq$ over D lifts to sets $S_1, S_2 \subseteq D$ by $S_1 \trianglelefteq S_2$ if and only if for all $d_1 \in S_1$ there exists $d_2 \in S_2$ such that $d_1 \trianglelefteq d_2$.

Proposition 2. Let $\langle D, \trianglelefteq \rangle$ be a poset that satisfies the ascending chain condition. Let $S_1, S_2 \subseteq S \subseteq D$ such that $S \trianglelefteq S_1$ and $S \trianglelefteq S_2$. Then $S \trianglelefteq S_1 \cap S_2$.

The force of proposition 2, is that it ensures that the following operator is well-defined (at least for domains that satisfy the ascending chain condition):

Definition 9. The map $\varrho : \wp(D) \rightarrow \wp(D)$ is defined $\varrho(S) = \cap\{S' \subseteq S \mid S \trianglelefteq S'\}$.

For domains that satisfy the ascending chain condition – the focus of our study – this operator ϱ computes the most compact representation of a set of abstractions

S. This provides a normal form that enables a power domain to be constructed without recourse to equivalence class manipulation. The power domain is then the complete lattice $\langle \varrho(\wp(D)), \trianglelefteq, \oplus, \otimes \rangle$ where $\oplus$ and $\otimes$ are defined as $S_1 \oplus S_2 = \varrho(S_1 \cup S_2)$ and $S_1 \otimes S_2 = \varrho(\{d_1 \otimes d_2 \mid d_1 \in S_1 \wedge d_2 \in S_2\})$. To observe that $\langle \varrho(\wp(D)), \trianglelefteq, \oplus, \otimes \rangle$ is a cHa, let $S \in \varrho(\wp(D))$ and $\{S_i \mid i \in I\} \subseteq \varrho(\wp(D))$ for some index set I. It follows from the definitions of $\otimes$ and $\oplus$ that $S \otimes (\oplus_{i \in I} S_i) = \varrho(\{d \otimes d_i \mid d \in S \wedge d_i \in S_i \wedge i \in I\}) = \oplus_{i \in I}(S \otimes S_i)$ and this equivalence is enough to verify that the power domain a cHa [2][Chapter IX, Theorem 15].

The projection operators lift to the power domain in a natural way by $\exists_x(S) = \varrho(\{\exists_x(d) \mid d \in S\})$ and similarly $\forall_x(S) = \varrho(\{\forall_x(d) \mid d \in S\})$. For equivalence between the three semantics to hold, $\langle \varrho(\wp(D)), \forall_x, \varrho(\wp(D)), \exists_x \rangle$ is required to be a Galois connection. The following proposition asserts that the only way to ensure this property, is to engineer $\forall_x : D \to D$ and $\exists_x : D \to D$ so that $\langle D, \forall_x, D, \exists_x \rangle$ is a Galois connection.

Theorem 5. $\langle \varrho(\wp(D)), \forall_x, \varrho(\wp(D)), \exists_x \rangle$ is a Galois connection if and only if $\langle D, \forall_x, D, \exists_x \rangle$ is a Galois connection.

Example 5. Consider the construction of a power domain for capturing numeric relationships between variables such that $\langle \varrho(\wp(D)), \forall_x, \varrho(\wp(D)), \exists_x \rangle$ is a Galois connection. Specifically consider Lin_X, the set of finite sets of equations of the form $ax + by < 0$ and $ax + by \leq 0$ where $a, b \in \{-1, 0, 1\}$ and $x, y \in X$ – a domain that arises in termination verification [22]. A mapping $\theta : X \to \mathbb{R}$ assigns a truth value to each $E \in Lin_X$ by $\theta(E) = \wedge_{e \in E} \theta(e)$ and $\theta(ax + by \circledast 0) \iff a\theta(x) + b\theta(y) \circledast 0$ where $\circledast \in \{<, \leq\}$. Then $E_1 \models E_2$ iff $\theta(E_1) \to \theta(E_2)$ holds for all $\theta : X \to \mathbb{R}$. Let $\bot = \{0 < 0\}$ and observe that $\theta(\bot)$ is false for all $\theta : X \to \mathbb{R}$. To construct $\oplus$ and $\otimes$, an operator $cl : Lin_X \to Lin_X$ is introduced to compute the entire set of equations entailed by a given equation set E (unless $E \models \bot$). Specifically $cl(E) = \cup\{E' \in Lin_X \mid E \models E'\}$ if $E \not\models \bot$ otherwise $cl(E) = \bot$. Then $\langle cl(Lin_X), \models, \oplus, \otimes \rangle$ is a finite lattice with a bottom element $\bot$ where $E_1 \oplus E_2 = E_1 \cap E_2$ if $E_1 \neq \bot$ and $E_2 \neq \bot$ whereas $E_1 \oplus E_2 = E_1$ if $E_2 = \bot$ and $E_1 \oplus E_2 = E_2$ if $E_1 = \bot$. Moreover $E_1 \otimes E_2 = cl(E_1 \cup E_2)$. By applying Floyd-Warshall shortest-path algorithms $cl(E)$ can be computed in $O(X^3)$ time [25]. To specify $\exists_x$ and $\forall_x$, the concept of a free variable is formalised as $FV(E) = \cup_{e \in E} FV(e)$ where $FV(ax + by \circledast 0) = \{x \mid a \neq 0\} \cup \{y \mid b \neq 0\}$. Then $\exists_x(E) = cl(\{e \in E \mid x \notin FV(e)\})$ and $\forall_x(E) = E$ if $x \notin FV(E)$ otherwise $\forall_x(E) = \bot$. Since $\exists_x \circ \forall_x(E) \models E \models \forall_x \circ \exists_x(E)$ and $\exists_x$ and $\forall_x$ are both monotonic, it follows that $\langle cl(Lin_X), \forall_x, cl(Lin_X), \exists_x \rangle$ is a Galois insertion. Moreover, when $\exists_x$ and $\forall_x$ are lifted to $\varrho(\wp(cl(Lin_X))$, as specified above, then theorem 5 ensures that $\langle \varrho(\wp(cl(Lin_X)), \forall_x, \varrho(\wp(cl(Lin_X)), \exists_x \rangle$ is also a Galois connection. This guarantees that the semantics have equal power for verification.

If a cHa is constructed via a power domain, although the pseudo-complement is guaranteed to exist, it may not be clear how to compute $S_1 \Rightarrow S_1$ so that the backward framework can be applied in verification. However, for a given cHa $\langle L, \sqcup, \sqcap \rangle$, from the axioms of Heyting algebras it follows that $(\sqcup_{i \in I} a_i) \Rightarrow b = \sqcap_{i \in I}(a_i \Rightarrow b)$ where $\{a_i \mid i \in I\} \subseteq L$ for an index set I and $b \in L$. Moreover, it can be shown that $b \Rightarrow (\sqcup_{i \in I} a_i) = \sqcup_{i \in I}(b \Rightarrow a_i)$. These properties enable strength reduction to be applied in the calculation of $S_1 \Rightarrow S_2$ for $S_1, S_2 \in \varrho(\wp(D))$. Specifically $S_1 \Rightarrow S_2 = \otimes\{\{d_1\} \Rightarrow S_2 \mid d_1 \in S_1\}$ and $S_1 \Rightarrow S_2 = \oplus\{S_1 \Rightarrow \{d_2\} \mid d_2 \in S_2\}$. Thus, in a similar fashion to $\forall_x$ and $\exists_x$, it is enough to define an procedure for computing $\{d_1\} \Rightarrow \{d_2\}$ over $d_1, d_2 \in D$, and then lift the operator to full $\varrho(\wp(D))$. This construction scheme is illustrated below.

Example 6. Returning to example 5, it is thus sufficient to construct an operation $\rightrightarrows: Lin_X^2 \to \wp(Lin_X)$ such that $E_1 \rightrightarrows E_2 = \{E_1\} \Rightarrow \{E_2\}$ if $E_1, E_2 \in cl(Lin_X)$. To aid the construction, define $\neg(ax + by < 0) = (-a)x + (-b)y \leq 0$ and $\neg(ax + by \leq 0) = (-a)x + (-b)y < 0$. Suppose $E_2 = \{e_1, \ldots, e_n\}$. Then $E_1 \rightrightarrows E_2 = \varrho(\{cl(\cup_{i=1}^n \{\neg e_i'\}) \mid e_i' \in cl(E_1 \cup \{\neg e_i\})\})$. The following proposition asserts the correctness of this construction.

Proposition 3. Let $E_1, E_2 \in cl(Lin_X)$. Then $E_1 \rightrightarrows E_2 = \{E_1\} \Rightarrow \{E_2\}$.

Example 7. To illustrate an application of $E_1 \rightrightarrows E_2$ consider

$$E_1 = \{x - y \leq 0, -x + y \leq 0\} \qquad \text{and} \qquad E_2 = \{y - z \leq 0, y < 0\}$$

so that $E_1, E_2 \in cl(Lin_X)$ as required. Let $e_1 = y - z \leq 0$ and $e_2 = y < 0$, hence $\neg e_1 = -y + z < 0$ and $\neg e_2 = -y \leq 0$. Then

$$\begin{aligned} cl(E_1 \cup \{\neg e_1\}) &= \{x - y \leq 0, -x + y \leq 0, -y + z < 0, -x + z < 0\} \\ cl(E_1 \cup \{\neg e_2\}) &= \{x - y \leq 0, -x + y \leq 0, -y \leq 0, -x \leq 0\} \end{aligned}$$

and therefore $E_1 \rightrightarrows E_2 = \{\{-x + y < 0\}, \{x - y < 0\}, \{y - z \leq 0, y < 0\}, \{y - z \leq 0, x < 0\}, \{x - z \leq 0, y < 0\}, \{x - z \leq 0, x < 0\}\}$. Observe that $\{\neg e_1\} \in E_1 \rightrightarrows E_2$ for all $e_1 \in E_1$ and that $E_2 \in E_1 \rightrightarrows E_2$. By lifting $\{E_1\} \Rightarrow \{E_2\}$ to arbitrary $S_1 \Rightarrow S_2$, the power domain construction is complete, thereby enabling any of the verification frameworks to be applied.

7 Related work

This paper compares various fixpoint frameworks for the task of verification. However, if assertions are given for each predicate, for instance, to specify properties of computed answers as in [8], then the verification problem reduces to checking a pre-fixpoint [6] and iteration can be avoided altogether. This check merely requires the assertion language to possess a decidable entailment test and therefore these languages can be particularly expressive [30]. If the assertion language coincides with an abstract domain, then properties can be automatically inferred relaxing the requirement to systematically annotate each predicate.

Schachte compares the precision of a goal-independent analysis for abstract success patterns against the concrete success patterns [28] and likewise compares a goal-dependent analysis for abstract call patterns (derived using a condensing framework) relative to the concrete call patterns [27]. Optimality theorems for the goal-independent [28][Theorem 15] and goal-dependent analysis [27][Theorem 3.13] state that these analyses derive abstractions that exactly match those obtained by applying the abstraction map to the concrete patterns. These results hold for condensing domains equipped with an abstraction map α that satisfies the relation $\alpha(C_1) \mathbin{\triangle\!\!\!\!\wedge} \alpha(C_2) = \alpha(\{c_1 \wedge c_2 \mid c_1 \in C_1 \wedge c_2 \in C_2\} \setminus \{false\})$ where $\mathbin{\triangle\!\!\!\!\wedge}$ is the abstract conjunction operator. Interestingly, whether these results are applicable critically depends on how α handles sets of unsolvable constraints. For instance, for the domain Pos_X consider $C_1 = \{E_1\}$ and $C_2 = \{E_2\}$ with the equation sets $E_1 = \{x = a\}$ and $E_1 = \{x = b\}$. Then $\alpha_{Pos}(\{E_1\}) \mathbin{\triangle\!\!\!\!\wedge} \alpha_{Pos}(\{E_2\}) = x \wedge x = x$, however $\alpha_{Pos}(\{E_1 \cup E_2\} \setminus \{false\}) \models \alpha_{Pos}(\{E_1 \cup E_2\}) = false$ since $\{E_1 \cup E_2\} \setminus \{false\} \subseteq \{E_1 \cup E_2\}$ and $E_1 \cup E_2$ is unsolvable. Although comparing abstract with concrete is a laudable goal, our work merely compares one abstract framework against another and thereby relaxing the requirement on α.

One alternative approach to analysis that is more in tune with the needs of verification is to structure the analysis around the assertions themselves and only perform the computation necessary for verifying the assertions, thereby analysing the program on demand. A method for constructing such an demand-driven analysis is presented in [9] for dataflow analyses with distributive flow functions. These demand-driven algorithms propagate assertion requirements backward against the control-flow until they are satisfied. Interestingly, reversing the binding mechanism between actual and formal arguments is analogous to calculating universal projection. However, the reverse binding operator of [9] is incorrect (for copy constant propagation) – the direction of approximation in parameter passing needs to be revised to return the strongest abstraction and thereby simulate universal projection. In fact, incredibly, the same Galois connection requirement for correctness and precision appears also to be necessary in the demand-driven analysis of imperative programs.

Termination checking is the problem of verifying that a logic program left-terminates for a given query whereas termination inference is the problem of inferring initial queries under which a logic program left-terminates [24]. It has been observed [12] that the "missing link" between termination inference and termination checking is the backward analysis of [17]. Indeed, Genaim and Codish [12] reconstruct the method of [24] in terms of existing black-box components that, according to [12], simplifies the formal justification and implementation of a termination inference analyser. First, the termination engine of [5] is used to compute a set of binary clauses which describe possible loops in the program with size relations. Second, Boolean functions are inferred for each predicate that describes moding conditions sufficient for each loop to only be executed a finite number of times. Third, the backward analysis of [17] is applied to infer initial modes that guarantee termination. The technical report version of [12] addresses the intriguing question of whether termination checking can verify all queries that can be inferred by termination inference and dually whether termination inference can infer all the queries that be verified with termination checking. The technical report presents a theorem that basically says that a termination checker reports that a program terminates for a mode if and only if the mode is deduced by a termination inference engine. The proof makes two assumptions about backward analysis named BA_1 and BA_2, and focuses on comparing the CHK and INF procedures that arise in termination analysis [12]. BA_1 is a precision assumption on backward analysis relating backward to forward analysis driven from input mode for a predicate q. Specifically, if backward analysis is applied to a program which is annotated with the call modes derived by the forward analysis, then the input mode inferred for q by backward analysis is not stronger than the mode of q used to initial the forward analysis. Note that this assumption relies, among other things, on the precision of universal projection.

Future work will examine the relation precision of differential methods [11].

8 Conclusions

This paper has provided a systematic comparison of the relative power of three different abstract interpretation frameworks for the problem of logic program verification. Conditions on the abstract domain operations have been derived which detail when these frameworks possess equivalent power. The paper also explains how power domains can satisfy the requirements for equivalence.

Acknowledgments We thank Roberto Giacobazzi for his insightful comments on the relationship between existential and universal projection that motivated this work. We also thank John Gallagher, Samir Genaim, Peter Schachte for their comments and NSF grant CCR-0131862 for partly funding this work.

References

1. A. Aiken and T. K. Lakshman. Directional Type Checking of Logic Programs. In *Static Analysis Symposium*, volume 864 of *LNCS*, pages 43–60. Springer, 1994.
2. G. Birkhoff. *Lattice Theory.* AMS Press, 1967.
3. M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *The Journal of Logic Programming*, 10(1/2/3&4):91–124, 1991.
4. M. Codish and V. Lagoon. Type Dependencies for Logic Programs using ACI-unification. *Theoretical Computer Science*, 238:131–159, 2000.
5. M. Codish and C. Taboch. A Semantic Basis for the Termination Analysis of Logic Programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.
6. M. Comini, R. Gori, G. Levi, and P. Volpe. Abstract Interpretation based Verification of Logic Programs. *Electronic Notes of Theoretical Computer Science*, 30(1), 1999.
7. P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *The Journal of Logic Programming*, 13(2–3):103–179, 1992.
8. W. Drabent and J. Małuszyński. Inductive Assertion Method for Logic Programs. *Theoretical Computer Science*, 59(1):133–155, 1988.
9. E. Duesterwald, R. Gupta, and M. L. Soffa. Demand-driven Computation of Interprocedural Data Flow. In *Principles of Programming Languages*, pages 37–48. ACM, 1995.

10. C. Fecht and H. Seidl. A Faster Solver for General Systems of Equations. *Science of Computer Programming*, 35(2–3):137–162, 1999.
11. M. García de la Banda, K. Marriott, P. J. Stuckey, and H. Søndergaard. Differential Methods in Logic Program Analysis. *The Journal of Logic Programming*, 35(1):1–37, 1998.
12. S. Genaim and M. Codish. Inferring Termination Conditions for Logic Programs using Backwards Analysis. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 2250 of *LNAI*, pages 681–690. Springer, 2001. Technical report version available at http://www.cs.bgu.ac.il/~mcodish/Papers/Pages/lpar01.html.
13. R. Giacobazzi and F. Scozzari. A Logical Model for Relational Abstract Domains. *ACM Transactions on Programming Languages and Systems*, 20(5):1067–1109, 1998.
14. M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental analysis of constraint logic programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, 2000.
15. R. J. M. Hughes and J. Launchbury. Reversing Abstract Interpretations. *Science of Computer Programming*, 22(3):307–326, 1994.
16. D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent And-Parallelism. *The Journal of Logic Programming*, 13(1/2/3&4):291–314, 1992.
17. A. King and L. Lu. A Backward Analysis for Constraint Logic Programs. *Theory and Practice of Logic Programming*, 2(4–5):517–547, 2002.
18. A. King and L. Lu. Forward versus Backward Verification of Logic Programs. Technical Report 5-03, Computing Laboratory, University of Kent, April 2003.
19. A. Langen. *Advanced Techniques for Approximating Variable Aliasing in Logic Programs*. PhD thesis, Computer Science Department, University of Southern California, Los Angeles, 1991.
20. B. Le Charlier, C. Leclére, S. Rossi, and A. Cortesi. Automatic Verification of Prolog Programs. *The Journal of Logic Programming*, 39(1–3):3–42, 1999.
21. B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, 1994.
22. N. Lindenstrauss and Y. Sagiv. Automatic Termination Analysis of Logic Programs. In *International Conference on Logic Programming*, pages 63–77. MIT Press, 1997.
23. L. Lu and A. King. Backward Type Inference Generalises Type Checking. In *Static Analysis Symposium*, volume 2477 of *LNCS*, pages 85–101. Springer, 2002.
24. F. Mesnard and S. Ruggieri. On Proving Left Termination of Constraint Logic Programs. *ACM Transactions on Computational Logic*, 4(2):207–259, 2003.
25. A. Miné. The Octagon Abstract Domain. In *Eighth Working Conference on Reverse Engineering*, pages 310–319. IEEE Computer Society, 2001.
26. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming*, volume 1870 of *LNCS*, pages 23–61. Springer, 2000.
27. P. Schachte. *Precise and Efficient Static Analysis of Logic Programs*. PhD thesis, Department of Computer Science, University of Melbourne, 1999.
28. P. Schachte. Precise Goal-independent Abstract Interpretation of Constraint Logic Programs. *Theoretical Computer Science*, 293(3):557–577, 2003.
29. D. van Dalen. *Logic and Structure*. Springer, 1997.
30. P. Volpe. A First-Order Language for Expressing Aliasing and Type Properties of Logic Programs. *Science of Computer Programming*, 39(1):125–148, 2001.