# Abstraction of Syntax

Vijay D'Silva* and Daniel Kroening

Department of Computer Science,
Oxford University
firstname.surname@comlab.ox.ac.uk

**Abstract.** The theory of abstract interpretation is a conceptual framework for reasoning about approximation of semantics. We ask if the creative process of designing an approximation can be studied mathematically. Semantic approximations, whether studied in a purely mathematical setting, or implemented in a static analyser, must have a representation. We apply abstract interpretation to syntactic representations and study abstraction of syntax. We show that semantic abstractions and syntactic abstractions are different, and identify criteria for deriving semantic abstractions by purely syntactic means. As a case study, we show that descriptions of numeric abstract domains can be derived by abstraction of syntax.

## 1 Where Do Abstractions Come from?

An important aspect of reasoning about the behaviour of dynamic systems such as programs, transition systems or process calculi is to focus on a few properties of interest and ignore irrelevant details. Abstract interpretation crystallises this intuition into a mathematical framework [4,5]. The first step is to characterise the behaviour of a system by fixed point in a lattice of semantic objects. The second step is to approximate this fixed point using a lattice and transformers, together called an abstract domain. The third step is to compute this fixed point approximation using iteration algorithms and operators to ensure termination and improve precision. The literature contains numerous domain-agnostic techniques for designing and computing fixed point approximations. The process of designing of an abstract domain has not been formalised in mathematical terms.

One component of designing an abstract interpreter is designing an abstract domain. The designer of an abstract domain usually has to answer three questions. What are the elements of the domain? How are abstract transformers implemented? Does analysis with the domain produce information for solving the problem? The first is a specification question, the second, an algorithmic question, and the third, an empirical evaluation question. Algorithmic details depend on the properties in the lattice, and the evaluation depends on the programs considered, so we expect that finding a generic, formal framework for answering the second and third questions will be difficult.

* Supported by Microsoft Research's European PhD Scholarship Programme.

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 396–413, 2013.

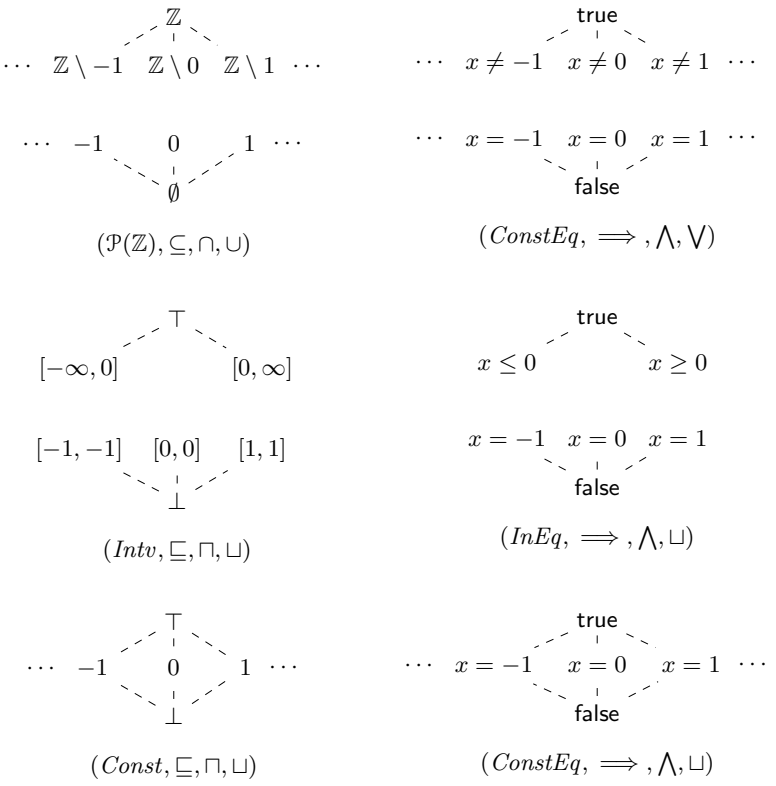



**Fig. 1.** The left column shows the powerset lattice of integers $\mathcal{P}(\mathbb{Z})$, the lattice of integer intervals *Intv*, and the lattice of integer constants *Const*. The right column contains logical representations of these domains. Each domain is a set of formulae ordered by implication. All sets of formulae are closed under infinitary conjunction, but not all are closed under disjunction.

We consider the problem of discovering *syntactic* specifications of abstract domains. We show that thinking of domains as logics enables a uniform approach to deriving specifications of several abstract domains used in practice. Our work applies to domains for numeric data [20], functional programs [16], and temporal logics [19]. We do not claim to formalise the process by which static analysis experts design abstractions. Instead, our work presents an alternative approach to deriving abstractions that are typically discovered manually. We believe this alternative can eventually be automated.

For an illustration of the ideas in this paper, imagine a program with a single variable $x$ that ranges over the mathematical integers $\mathbb{Z}$. The concrete domain containing possible values of $x$ is the lattice $\mathcal{P}(\mathbb{Z})$ shown in Figure 1. Operating on this lattice is intractable, so analysis tools use abstract domains such intervals. In the figure, each element on the left (a semantic object) is represented by a

formula on the right (a syntactic object). For instance, $x \leq 1$ represents the interval $[-\infty, 1]$. The lattice of intervals can be viewed as a logic containing formulae of the form $x \leq k$ and $x \geq k$, and closed under conjunction but not under disjunction. Implication defines the lattice order. The logics in the right-hand column of Figure 1 can be viewed as specifications of abstract domains. We study the problem of systematically discovering logical descriptions of abstract domains given a logical description of a concrete domain.

There is a loose connection between the approach we take and proof theory. Proof theory provides a mathematical basis for studying the structure and properties of formal proofs, and procedures for deriving proofs. Though proof theory does not formalise, or even automate, the exact process by which mathematicians construct proofs, it has led to new insights about the structure of proofs, and to automated deduction techniques. Analogously, this paper is a first step towards a mathematical description of how abstract domains can be derived.

One may ask if the syntactic, logical approach we take has advantages over the semantic approach usually adopted in the literature. In the Galois insertion setting, a result of Cousot and Cousot [5] shows that the space of domains is a complete lattice. Domains in this lattice can be combined using reduced sum, reduced product, reduced power, and various other operations on domains [5,12,13]. This lattice of domains is a rich mathematical object that cannot be manually searched.

In contrast, the syntax of a logic representing an abstract domain can usually be described by a short BNF grammar. Committing to a syntactic representation may restrict the kinds of domains that can be expressed, but also makes it easier to enumerate syntactic descriptions of domains. There are many parameters such as the number of operators, variables and nesting depth, that can be manipulated to generate logics from a grammar. We show later that a wide range domain specifications can be derived with our approach, and that the syntactic descriptions are succinct.

**Contribution.** This paper addresses the problem of deriving syntactic specifications of abstract domains. We present a rigorous, grammar-based solution to this problem. The main idea is to exploit a non-trivial connection between abstract domains and logical theories and reduce the problem of specifying domains to that of underapproximating a grammar. We make the following contributions.

- A characterisation of BNF grammars in terms of a domain of syntax trees and syntax transformers. By underapproximating the fixed point defined by a grammar, we systematically derive sub-languages.
- The derivation of overapproximating semantic domains by underapproximating the semantics of grammars.
- A case study showing that several existing domains can be derived using the framework developed in this paper.

By decoupling the problem of domain specification from that of algorithm design and evaluation, we believe intellectual effort can be directed towards the parts of the problem that are difficult to systematise. To revisit the analogy to proof

theory, we believe that a syntactic approach to abstract domain design, may lead to new automated procedures for domain construction.

The paper is organised as follows. We introduce *meta-syntax*, a variant of BNF in Section 2, and present a collecting semantics for meta-syntax grammars. We apply standard abstract interpretation to underapproximate the collecting semantics of meta-syntax grammars in Section 3. A sub-language is an underapproximation of a language. A sub-language derived by abstract interpretation is an inductively defined sub-language of an inductively defined language. We present our case study in Section 4 and review related work in Section 5.

## 2 Meta-syntax

In this section, we introduce a variant of BNF called *meta-syntax*, and define the interpretation of meta-syntax grammars using domains and transformers. The difference between BNF and meta-syntax is clarified below.

*Example 1.* Let *Prop* be a set of propositions and $p$ range over *Prop*. BNF definitions for propositional logic and a sub-logic is given below.

$$\varphi ::= p \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \qquad \text{Propositional logic}$$
$$\varphi ::= p \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \qquad \text{Monotone propositional logic}$$

The first line above can be read as follows: a propositional formula $\varphi$ is a proposition, or is the composition of formulae using the Boolean connectives $\wedge$, $\vee$ and $\neg$. The grammars above are similar, but not identical. The meta-syntax grammar below specifies a language containing constants and operators.

$$\mathsf{f} ::= \mathsf{prop} \mid \mathsf{op}(\widehat{\mathsf{f}}) \qquad \mathsf{Gram}$$

The two logics above can be obtained by instantiating the macros $\mathsf{prop}$ and $\mathsf{op}$ as shown below.

$$\mathsf{inst}_1 \mathrel{\hat{=}} \{\mathsf{prop} \mapsto \mathit{Prop}, \mathsf{op} \mapsto \{\neg, \vee, \wedge\}\} \quad \mathsf{inst}_2 \mathrel{\hat{=}} \{\mathsf{prop} \mapsto \mathit{Prop}, \mathsf{op} \mapsto \{\vee, \wedge\}\}$$

We prefer to use meta-syntax over BNF because we can instantiate *the same grammar* in different ways to derive different logics. More precisely, a grammar is a structure analogous to a program. The instantiations above can be viewed as specifying two different abstract transformers for the same syntactic macros. We show later that meta-syntax allows us to describe the syntactic relationship between the formulae in monotone propositional logic and propositional logic using abstract interpretation. ◁

The meta-syntax system below syntactically resembles BNF, but is based on Istrail's [15] fixed point characterisation of recursively enumerable languages. Though meta-syntax is *less general* than Cousot and Cousot's bi-inductive semantics [10], we have chosen to work with meta-syntax because it suffices for this paper, and because it is similar to BNF.

**Definition 1.** *Let* $\mathsf{Var}$ *be a set of meta-variables and* $\mathsf{Sym}$*, a set of meta-symbols. A syntax term is one of the following.*

1. *A meta-variable* $\mathsf{x}$ *or a meta-symbol* $\mathsf{s}$*.*
2. *The composition* $\mathsf{s}(t_0, \ldots, t_n)$ *of a meta-symbol with terms.*
3. *The uniform composition* $\mathsf{s}(\widehat{t})$ *of a meta-symbol* $\mathsf{s}$ *with a term* $t$*.*
4. *The substitution* $\mathsf{t}_1[\mathsf{x}/\mathsf{t}_2]$ *of a meta-symbol* $\mathsf{x}$ *in a term* $\mathsf{t}_1$ *with a term* $\mathsf{t}_2$*.*

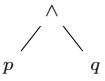
*A grammar rule is of the form* $\mathsf{x} ::= t_0 \mid \cdots \mid t_n$*, where* $t_i$ *is a syntax term. A grammar is a finite set of grammar rules.*

A meta-variable is *free* in a grammar if it does not occur on the left hand side of a grammar rule. Uniform composition, not present in BNF, is required to collate symbols like $\wedge$ and $\neg$ independent of their arity.

The language of a grammar is analogous to the semantics of a program. The operational semantics of a program statement is given by state transitions, the collecting semantics is defined using state transformers, and properties of programs are characterised by fixed points. Similarly, the meaning of a meta-syntax term is given by a set of syntax trees, the collecting semantics is defined using syntax transformers on a powerset lattice of syntax trees, and languages are defined by fixed points. A formalisation of this idea follows. An alternative formalisation is to use partial-expressions instead of syntax trees.

**Signatures, Syntax Trees and Languages.** A *signature* $(Sig, ar)$ is a set of symbols $Sig$ with an arity function $ar : Sig \to \mathbb{O}$, associating an ordinal with each symbol. The arity function is left implicit. Ordinals are required for infinitary conjunction and disjunction. A nullary symbol has arity 0. Fix a signature $Sig$.

A *syntax tree* $\tau = (V, E, sym)$ is a finite-height, ordered tree $(V, E)$ with a function $sym : V \to Sig$ labelling vertices with symbols. The children of a vertex form a possibly infinite sequence $\bar{v} = v_0, v_1, \ldots$. The root of a tree is $root(\tau)$. The set of syntax trees over $Sig$ is $\mathsf{Syn}(Sig)$, written $\mathsf{Syn}$ if $Sig$ is clear. A *well formed tree* satisfies that every vertex $v$ has $ar(sym(v))$ children. If we assume the symbol $\wedge$ to have arity 2, the tree on the left is not well-formed but the tree on the right is. An *expression* is a well-formed syntax tree and a *language* is a set of expressions.


An *instantiation* $\mathsf{In} = (\mathsf{Syn}, \mathsf{inst})$ is a set of syntax trees $\mathsf{Syn}$ with an instantiation function $\mathsf{inst} : \mathsf{Sym} \to \mathcal{P}(\mathsf{Syn})$ satisfying that, for each $\mathsf{s}$ in $\mathsf{Sym}$, $\mathsf{inst}(\mathsf{s})$ contains single vertex trees. (Strictly speaking, vertices are not symbols because operations on symbols and trees differ.) A *syntax environment* (or just environment) $\mathsf{env} : \mathsf{Var} \to \mathcal{P}(\mathsf{Syn})$ maps meta-variables to sets of syntax trees.

The *meaning* of a term $t$ given an instantiation $\mathsf{In} = (\mathsf{Syn}, \mathsf{inst})$ and environment $\mathsf{env}$, is denoted $\|t\|_{\mathsf{In},\mathsf{env}}$ and is defined below. We abbreviate the symbol for the arity of the root symbol of a tree $\tau$ to $ars(\tau)$.

$$
\begin{aligned}
\|\mathsf{x}\|_{\mathsf{In},\mathsf{env}} &\hat{=} \mathsf{env}(\mathsf{x}) \\
\|\mathsf{s}(t_0,\ldots,t_{n-1})\|_{\mathsf{In},\mathsf{env}} &\hat{=} \{\tau \mid \mathit{root}(\tau) \text{ is in } \mathsf{inst}(\mathsf{s}), ars(\tau) = n \\
&\qquad \text{and child } \tau_i \text{ of } \mathit{root}(\tau) \text{ is in } \|t_i\|_{\mathsf{In},\mathsf{env}}\} \\
\|\mathsf{s}(\widehat{t})\|_{\mathsf{In},\mathsf{env}} &\hat{=} \{\tau \mid \mathit{root}(\tau) \text{ is in } \mathsf{inst}(\mathsf{s}) \text{ and for } i < ars(\tau) \\
&\qquad \tau \text{ has children } \tau_i \in \|t\|_{\mathsf{In},\mathsf{env}}\} \\
\|t_1[\mathsf{x}/t_2]\|_{\mathsf{In},\mathsf{env}} &\hat{=} \{\tau \mid \tau \text{ is in } \|t_1\|_{\mathsf{In},\mathsf{env}'}, \text{ where} \\
&\qquad \mathsf{env}' \text{ maps } \mathsf{x} \text{ to } \|t_2\|_{\mathsf{In},\mathsf{env}}, \text{ and } \mathsf{y} \neq \mathsf{x} \text{ to } \|\mathsf{y}\|_{\mathsf{In},\mathsf{env}}\} \\
\|t_1 \mid t_2\|_{\mathsf{In},\mathsf{env}} &\hat{=} \|t_1\|_{\mathsf{In},\mathsf{env}} \cup \|t_2\|_{\mathsf{In},\mathsf{env}}
\end{aligned}
$$

The terms above may contain free variables. The *language of a grammar* $\mathsf{x}_1 ::= t_1, \ldots, \mathsf{x}_n ::= t_n$, with no free variables is an environment $\mathsf{env}$, satisfying that $\mathsf{env}(\mathsf{x}_i)$ equals $\|t_i\|_{\mathsf{In},\mathsf{env}}$, for every $i$.

**Domains and Transformers.** We now define the semantics of a grammar in terms of domains and transformers. The *domain of syntax trees* is the powerset lattice $(\mathcal{P}(\mathsf{Syn}), \subseteq, \cap, \cup)$ containing sets of syntax trees ordered by inclusion, and closed under union, intersection and set complement. The *domain of instantiation functions* is $(\mathsf{Sym} \to \mathcal{P}(\mathsf{Syn}), \sqsubseteq, \sqcap, \sqcup)$, where the order is defined *pointwise*:

$\mathsf{inst}_1 \sqsubseteq \mathsf{inst}_2$ if $\mathsf{inst}_1(\mathsf{s}) \subseteq \mathsf{inst}_2(\mathsf{s})$ holds for every meta-symbol $\mathsf{s}$.

The *pointwise meet* $\mathsf{inst}_1 \sqcap \mathsf{inst}_2$ of two instantiation functions maps each meta-symbol $\mathsf{s}$ to $\mathsf{inst}_1(\mathsf{s}) \cap \mathsf{inst}_2(\mathsf{s})$. The pointwise join is similarly defined. The *domain of syntax-environments* is $(\mathsf{Env}, \sqsubseteq, \sqcap, \sqcup)$, where $\mathsf{Env}$ is $\mathsf{Var} \to \mathcal{P}(\mathsf{Syn})$ and the operations are defined pointwise. All these domains are complete lattices. The least element of $\mathsf{Env}$, denoted $\mathsf{env}_\emptyset$, maps all meta-variables to the emptyset.

Fix an instantiation $\mathsf{In}$. We define the transformers for each entity in meta-syntax. The definitions below can be read like the definitions of transformers for statements in a programming language.

A *syntax transformer* $syn_t : \mathsf{Env} \to \mathcal{P}(\mathsf{Syn})$ for a term $t$ maps an environment $\mathsf{env}$ to the set of syntax trees $\|t\|_{\mathsf{In},\mathsf{env}}$. For brevity, we do not enumerate the definition of $syn_{\mathsf{c}}$, $syn_{\mathsf{f}(\widehat{c})}$, and $syn_{t_1[\mathsf{x}/t_2]}$, but we emphasise that $syn_t$ is inductively defined. Note that the transformer defined by the separator $\mid$ is union, so $\mid$ the equivalent of a join point in a flow graph. A grammar rule $\mathsf{x} ::= r$ defines a transformer $syn_{\mathsf{x}::=r} : \mathsf{Env} \to \mathsf{Env}$ that maps an environment $\mathsf{env}$ to $\mathsf{env}[\mathsf{x}/\, syn_r(\mathsf{env})]$. To define the transformer for a grammar rule, we require a function $\mathsf{null}$ to filter out nullary symbols.

$$
\begin{aligned}
\mathsf{null}(\mathsf{s}) &\hat{=} \{\tau \in \mathsf{inst}(\mathsf{s}) \mid ars(\tau) = 0\} & \mathsf{null}(\mathsf{x}) &\hat{=} \emptyset \\
\mathsf{null}(\mathsf{s}(\widehat{t})) &\hat{=} \mathsf{null}(\mathsf{s}) & \mathsf{null}(t_1 \mid t_2) &\hat{=} \mathsf{null}(t_1) \cup \mathsf{null}(t_1)
\end{aligned}
$$

A rule $\mathsf{x}_i ::= r_i$ generates an equation

$$\mathsf{x}_i = \mathsf{null}(r_i) \sqcup \mathit{syn}_{r_i}(\mathsf{env})$$

and a grammar generates a system of equations. Recall that the reachable states of a transition system are the fixed point of an equation $X = \mathit{Init} \cup \mathit{post}(X)$, where *Init* is a set of initial states and *post* is the successor transformer. Observe that the equations above have the same form. The *language of a grammar* is the environment representing the least solution to these equations.

*Example 2.* Let us derive a fixed point for the grammar and instantiation $\mathsf{inst}_1$ in Example 1. Iterating through the grammar produces the environments below.

$$\mathsf{env}_0 = \{\mathsf{f} \mapsto \emptyset\} \qquad \mathsf{env}_1 = \{\mathsf{f} \mapsto \mathit{Prop}\} \quad \cdots \qquad \mathsf{env}_n = \{\mathsf{f} \mapsto T_n\}$$

The initial environment is empty. After one iteration, we have the set of propositions. After $n$ iterations, $T_n$ contains expressions with at most $n-1$ operators. The fixed point of this sequence maps $\mathsf{f}$ to all propositional formulae. ◁

**Proposition 1.** *The language of a grammar with respect to an instantiation is the least fixed point of the system of equations the grammar generates.*

**Semantic Structures.** Grammars specify syntax. The semantics associated with the syntax is usually given by an interpretation function, which can be viewed as the operational semantics of the language. The collecting semantics is often derived by lifting the operational semantics to a powerset lattice. We directly assign a "collecting interpretation" to a language. For example, the standard interpretation of $+$ is a binary function in $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$. The lattice interpretation of $+$ is interpreted as a unary function $\mathcal{P}(\mathbb{N} \times \mathbb{N}) \to \mathcal{P}(\mathbb{N})$ that maps every pair $(m, n)$ in a set $X$ to $m + n$.

A *semantic structure* $\mathcal{S} = (\mathcal{A}, \mathcal{F})$ for a signature *Sig* is a family of lattices $\mathcal{A}$ called *domains* and a family of functions $\mathcal{F}$ called *interpretations*. Each symbol $f$ is interpreted as a function $f^{\mathcal{S}} : A_{f,i} \to A_{f,o}$ with input and output domains $A_{f,i}$ and $A_{f,o}$. If all the domains are identical, we write $A$ for $\mathcal{A}$.

A *structure* $(\mathsf{Gram}, \mathsf{In}, \mathcal{S})$ consists of a grammar, an instantiation, and a semantic structure. The *meaning* of an expression in $\mathcal{S}$ is *partially* defined below.

$$\begin{aligned}
[\![c]\!]_{\mathcal{S}} &\mathrel{\hat{=}} c^{\mathcal{S}} \text{ where } c, \text{ is a symbol with arity } 0\\
[\![f(\bar{a})]\!]_{\mathcal{S}} &\mathrel{\hat{=}} f^{\mathcal{S}}([\![a_0]\!]_{\mathcal{S}}, \ldots, [\![a_{n-1}]\!]_{\mathcal{S}}), \text{ where } f(\bar{a}) \in \|\mathsf{s}(t_0, \ldots, t_{n-1})\|_{\mathsf{In},\mathsf{env}}\\
[\![f(\bar{a})]\!]_{\mathcal{S}} &\mathrel{\hat{=}} f^{\mathcal{S}}([\![a_0]\!]_{\mathcal{S}}, \ldots, [\![a_i]\!]_{\mathcal{S}}, \ldots), \text{ where } f(\bar{a}) \in \|\mathsf{s}(\widehat{t})\|_{\mathsf{In},\mathsf{env}}
\end{aligned}$$

The definitions above suffice because every syntax tree arises from a meta-syntax term. The definition is partial because the arguments of a function symbol must have the same domain as their sub-expressions. It is routine to make this notion precise using types.

## 3 Abstraction of Syntax

This section contains two ideas. First, we use abstract interpretation to derive underapproximations of the fixed point of a grammar and obtain sub-languages. A grammar defines a language, and every subset of the language is a sub-language. An arbitrary subset may, however, not have an inductive definition related to the grammar. By applying abstract interpretation, we can underapproximate fixed points by fixed points, thereby deriving inductively defined sub-languages of an inductively defined language.

The second idea is to underapproximate syntax to derive overapproximations of semantics. The idea is straightforward, but relies on non-trivial conditions relating abstract domains and logics.

**Abstract Interpretation.** Let $(L, \sqsubseteq)$ and $(M, \preccurlyeq)$ be posets. Two functions $\alpha : L \to M$ and $\gamma : M \to L$ form a *Galois connection* if,

$$\text{for all } x \in L \text{ and } y \in M,\ \alpha(x) \preccurlyeq y \text{ if and only if } x \sqsubseteq \gamma(y).$$

If $L$ is a powerset lattice, abstractions with respect to $\subseteq$ are *overapproximating* and those with respect to $\supseteq$ are *underapproximating*. Several abstract elements may have the same concretisation. For instance, all intervals $[a, b]$ with $a$ greater than $b$ concretise to the empty set. Such redundancies do not occur in a Galois insertion. A *Galois insertion* is a Galois connection in which $\gamma$ is injective.

*Example 3.* This example shows that the standard task of defining a sub-language, such as the negation-free fragment of propositional logic, can be completely formalised and understood as deriving and applying best abstract transformers in abstract interpretation. Recall the grammar and instantiation $\mathsf{inst}$ from Example 2. The signature of propositional logic is given by $Sig_1$ below and of the negation-free fragment by $Sig_2$ below.

$$Sig_1 = Prop \cup \{\wedge, \vee, \neg\} \qquad Sig_2 = Prop \cup \{\wedge, \vee\}$$

These signatures define domains which form an underapproximating Galois connection with abstraction and concretisation given below.

$$\mathcal{P}(\mathsf{Syn}(Sig_1)) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{P}(\mathsf{Syn}(Sig_2)) \qquad \alpha(T) = T \cap \mathsf{Syn}(Sig_2) \qquad \gamma(S) = S$$

The abstraction function maps a set of syntax trees to the subset that is negation-free, and the concretisation function is the identity function. The Galois connection lifts pointwise to the instantiation and environment domains.

If we have an instantiation function $\mathsf{inst}$ as shown below, we can *derive* the *abstract instantiation* function $\alpha \circ \mathsf{inst} \circ \gamma$ also shown below.

$$\mathsf{inst} = \{\mathsf{prop} \mapsto Prop, \mathsf{op} \mapsto \{\wedge, \vee, \neg\}\}$$
$$\alpha \circ \mathsf{inst} \circ \gamma = \{\mathsf{prop} \mapsto Prop, \mathsf{op} \mapsto \{\wedge, \vee\}\}$$

The abstract instantiation function is the best abstract transformer approximating $\mathsf{inst}$. The fixed point of the grammar for propositional logic with the abstract instantiation defines the set of negation-free formulae. $\lhd$

*Example 4.* This example illustrates a more complex abstract domain. Let $\mathsf{ATree}$ be the set of syntax trees (not necessarily well-formed) with at most one negation symbol. Observe that $\mathsf{ATree}$ cannot be of the form $\mathsf{Syn}(Sig)$, because $Sig$ would contain negation and the resulting trees, multiple negations. Nonetheless, $\mathcal{P}(\mathsf{ATree})$ is a subset of $\mathcal{P}(\mathsf{Syn}(Sig_1))$ from Example 3. Intuitively, the *abstract syntax transformers* operate on the lattice $\mathsf{Var} \to \mathcal{P}(\mathsf{ATree})$, and the fixed point of the grammar contains formulae with at most one negation. $\lhd$

**Sub-languages by Abstract Interpretation.** A *sub-signature* is a subset of a signature. A *sub-language* is a subset of a language. Every underapproximation of the fixed point of a grammar is trivially a sub-language. We are interested in inductively defined sub-languages, because they finitely represent infinitely many formulae. We derive such languages using abstract interpretation to find underapproximations that are also fixed points.

**Definition 2.** *An instantiation* $\mathsf{AIn} = (\mathsf{ATree}, \mathsf{ainst})$ *is a syntactic abstraction of* $\mathsf{In} = (\mathsf{Syn}, \mathsf{inst})$ *if there is an underapproximating Galois connection* $\mathcal{P}(\mathsf{Syn}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{P}(\mathsf{ATree})$ *and the inclusion* $\gamma(\mathsf{ainst}(\mathsf{s})) \subseteq \mathsf{inst}(\mathsf{s})$ *holds for all symbols* $\mathsf{s}$.

Fix a syntactic abstraction $\mathsf{AIn} = (\mathsf{ATree}, \mathsf{ainst})$. We call $\mathsf{AIn}$ a *signature abstraction* if $\mathsf{ATree}$ contains all trees over a signature. The set $\mathsf{ATree}$ in Example 4 does not contain all trees over a signature.

The set of *abstract environments* is $\mathsf{AEnv} = \mathsf{Var} \to \mathcal{P}(\mathsf{ATree})$. The *abstract syntactic transformer* $asyn_t : \mathsf{AEnv} \to \mathcal{P}(\mathsf{ATree})$ for a term $t$ is $\alpha \circ syn_t \circ \gamma$, where $\alpha$ and $\gamma$ are defined by pointwise lifting. The abstract transformer for a grammar rule is similarly defined, and the abstract equations are obtained by replacing syntax transformers by their abstract counterparts. The *abstract language* defined by a grammar is the least abstract environment that represents a solution to the abstract equations. From standard abstract interpretation theory, we have that the abstract language underapproximates the language of a grammar.

**Proposition 2.** *The abstract language of a grammar underapproximates the language of a grammar.*

**Languages to Domains.** We now relate sub-languages to abstract domains over lattices representing semantic objects. A semantic abstraction $(\mathcal{A}, \mathcal{F})$ of a semantic structure $(\mathcal{C}, \mathcal{G})$ satisfies two conditions for every $f$ in $Sig$.

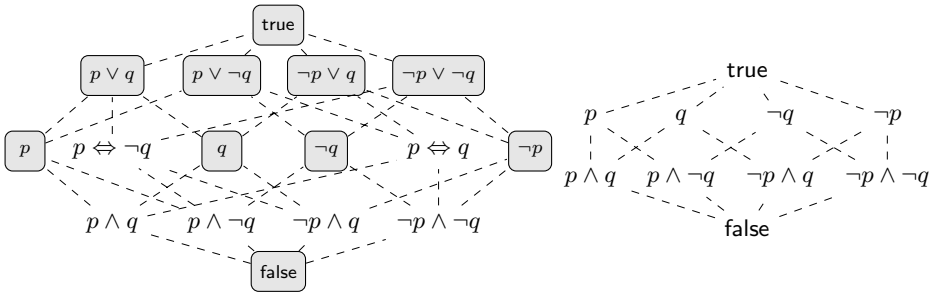
1. There is Galois connection $(C_{f,i}, \alpha_{f,i}, \gamma_{f,i}, A_{f,i})$, and a similar one for $f, o$.
2. The interpretations satisfy the soundness condition $\alpha_{f,o} \circ f^{\mathcal{C}} \sqsubseteq f^{\mathcal{A}} \circ \alpha_{f,i}$.

Fix a structure $(\mathsf{Gram}, \mathsf{In}, \mathcal{S})$ defining a language $\mathsf{Lang}$. Since we interpret languages over lattices, sub-languages define posets. The *structure defined by* $\mathsf{Lang}$,

denoted *struct*(Lang), is the family of posets that together contain the elements $\{[\![e]\!]_{\mathcal{S}} \in \ \mid e \in \mathsf{Lang}\}$, and are ordered as before. The next example illustrates sub-languages *struct*(Lang) and how they define lattices.

*Example 5.* All languages in this example can be derived by abstraction of syntax, but are presented informally to save space. Let $\mathsf{Lang}_1$ be propositional logic over two variables $p$ and $q$. The semantic domain is the set of truth assignments $\mathcal{P}(Prop \to \mathbb{B})$. The lattice *struct*($\mathsf{Lang}_1$) is shown on the left below.

Consider a language $\mathsf{Lang}_2$ closed under conjunction, with negation only of propositions. The lattice *struct*($\mathsf{Lang}_2$) is on the right. There is a Galois insertion from *struct*($\mathsf{Lang}_1$) to *struct*($\mathsf{Lang}_2$).




A third language $\mathsf{Lang}_3$ contains propositions, negation of propositions, and is closed under disjunction. The elements of *struct*($\mathsf{Lang}_3$) are shaded in the left-lattice. There is no Galois connection from *struct*($\mathsf{Lang}_1$) to *struct*($\mathsf{Lang}_3$) because *struct*($\mathsf{Lang}_3$) contains $p$ and $q$, but not $p \wedge q$. However, $\mathsf{Lang}_3$ is a sub-language of $\mathsf{Lang}_1$ and can be derived by abstraction of syntax. ◁

The poset defined by $\mathsf{Lang}_3$ above does not admit a Galois connection because $p \wedge q$ do not have a unique, minimal overapproximation. We resolve this problem with the following result from lattice theory. Consult the textbook [11, Theorem 7.3] or Cousot and Cousot's result on Moore families [5] for details.

**Theorem 1.** *An abstract domain $A$ is in a Galois insertion with $C$ exactly if $A$ is a subset of $C$ closed under meets.*

This characterisation allows us to relate languages with abstract domains in the setting of Galois insertions. A language that defines a structure closed under arbitrary meets also defines an abstract domain. Let Lang be a language interpreted over a semantic structure $\mathcal{S} = (\mathcal{A}, \mathcal{F})$. For simplicity, assume that $\mathcal{A}$ is a single domain. The general case is similar to what follows but notationally cumbersome. The language Lang is *completely conjunctive* if for every set of expressions $E \subseteq \mathsf{Lang}$, there exists an expression $C_E$ in Lang satisfying the equality $\bigsqcap \{[\![e]\!]_{\mathcal{S}} \mid e \in E\} = [\![C_E]\!]_{\mathcal{S}}$. The theorem below is new, and states that completely conjunctive languages define abstract domains.

**Theorem 2.** *Let* Lang *be a language interpreted over a domain* $(C, \sqsubseteq, \sqcap, \sqcup)$*. If* Lang *is completely conjunctive, the poset struct*(Lang) *is a complete lattice that is in a Galois insertion with* $C$*.*

The theorem can be proved by showing that for a completely conjunctive Lang, *struct*(Lang) is meet-closed, and then by invoking Theorem 1.

*Discussion.* We have considered the Galois insertion and complete lattice setting of abstract interpretation for two reasons. We require concretisation functions to be injective because every element of the abstract domain will have only one representation. Abstract domains used in practice may contain redundant representations of the same semantic element. Such domains are not definable in the approach we give above. We consider complete lattices because when arbitrary meet operations are defined, every concrete element has a unique over-approximation. Our completeness assumption excludes important domains such as polyhedra, or regular languages, which do not form complete lattices.

The framework in this paper can be extended to logics whose interpretation defines posets, by interpreting formulae in the logic and using the implication order. In this weaker setting, abstraction functions have to be manually defined, so the abstract element that overapproximates arbitrary concrete elements will not be defined by the framework. Extend this framework to derive abstract domains in which the concretisation function is not injective requires significantly more effort. This is because we map expressions to their meaning, and to obtain redundant representations, we must weaken the meaning function $[\![\cdot]\!]_{\mathcal{S}}$ to distinguish between semantically equivalent formulae.

## 4 Syntactic Derivation of Semantic Domains

We now follow the approach below to generate specifications of abstract domains.

1. Define a structure $(\mathsf{Gram}, \mathsf{In}, \mathcal{S})$, for a logic expressing properties of a program and it's datatypes, and let $\mathcal{S}$ be the semantic structure for this logic.
2. Choose parameters defining subsets of syntax trees.
3. Fix the parameters and compute the sub-language to obtain a sub-logic $\mathsf{L}$.
4. If $\mathsf{L}$ is closed under infinitary conjunction, *struct*($\mathsf{L}$) is an abstract domain.

In this section, we apply this flow to a specific logic and derive many abstract domains existing in practice.

**A Logic of Program Properties.** The logic we consider is an extension of Presburger arithmetic with infinitary conjunction and disjunction, and with the next-state modality. This logic can encode properties (such as reachability or termination) of programs written in Turing complete languages. The grammar below defines a language with terms and formulae.

$$\mathsf{t} ::= \mathsf{var} \mid \mathsf{fun}(\widehat{\mathsf{t}}), \quad \mathsf{f} ::= \mathsf{pred}(\widehat{\mathsf{t}}) \mid \mathsf{bool}(\widehat{\mathsf{f}}) \mid \mathsf{mod}(\widehat{\mathsf{f}})$$

The signature *Sig* contains variables *Var* and the sets below ($k$ is a positive integer). The binary predicate $\equiv_k$ denotes congruence modulo $k$. Other symbols have their standard arity.

$$Fun \mathrel{\hat{=}} \{+\} \cup \mathbb{N} \quad Pred \mathrel{\hat{=}} \{<, \leq, =, >, \geq, \equiv_k\} \quad Bool \mathrel{\hat{=}} \left\{\bigvee, \bigwedge, \wedge, \vee, \neg\right\}$$

The instantiation of meta-symbols is below.

$$\mathsf{var} \mapsto Var \quad \mathsf{fun} \mapsto Fun \quad \mathsf{pred} \mapsto Pred \quad \mathsf{bool} \mapsto Bool \quad \mathsf{mod} \mapsto \{\mathsf{EX}\}$$

An example formula is $\mathsf{EX}(x = y + y)$, stating that there is a successor state in which $x$ equals $2y$. Familiar modalities can be recovered using infinitary operators: $\mathsf{AX}\varphi$ is the formula $\neg\mathsf{EX}\neg\varphi$, which is satisfied by a state if all its successors satisfy $\varphi$. Let $\mathsf{AX}^i\varphi$ denote the formula $\mathsf{AX}\cdots\mathsf{AX}\varphi$, in which $\mathsf{AX}$ occurs $i$-times in sequence. The formula $\mathsf{AG}\varphi$, defined as $\bigwedge_i \mathsf{AX}^i\varphi$, asserts that $\varphi$ is true on every path. Infinitary operators can also be used to express multiplication.

We interpret this logic over transition systems. Let *Val* be a set of values and $Env \mathrel{\hat{=}} Var \to Val$ be the set of states (program environments). A *transition system* $M = (Env, E)$ contains a relation $E \subseteq Env \times Env$ . Recall that $E$ defines a *predecessor transformer* $pre : \mathcal{P}(Env) \to \mathcal{P}(Env)$ as below.

$$pre(X) \mathrel{\hat{=}} \{s \in Env \mid \text{ there exists } t \text{ in } X \text{ and } (s,t) \text{ is in } E\}$$

The semantic structure we need contains the lattices $\mathcal{P}(\mathbb{N})$ and $\mathcal{P}(Env)$. The semantics of a term $t$ is a set of values $[\![t]\!]_{\mathcal{S}} : \mathcal{P}(Env) \to \mathcal{P}(\mathbb{N})$, and of a formula $\varphi$ is a set of environments $[\![\varphi]\!]_{\mathcal{S}} : \mathcal{P}(Env) \to \mathcal{P}(Env)$, both defined below.

$$[\![x]\!]_{\mathcal{S}} \mathrel{\hat{=}} X \mapsto \{\varepsilon(x) \mid \varepsilon \in X\}$$
$$[\![t_1 + t_2]\!]_{\mathcal{S}} \mathrel{\hat{=}} X \mapsto \{m + n \mid m \in [\![t_1]\!]_{\mathcal{S}}, n \in [\![t_2]\!]_{\mathcal{S}}\}$$

$$[\![\mathsf{EX}\varphi]\!]_{\mathcal{S}} \mathrel{\hat{=}} pre([\![\varphi]\!]_{\mathcal{S}}) \qquad [\![P(\bar{t})]\!]_{\mathcal{S}} \mathrel{\hat{=}} \{\varepsilon \in Env \mid \varepsilon \text{ satisfies } P(\bar{t})\}$$
$$[\![\varphi \wedge \psi]\!]_{\mathcal{S}} \mathrel{\hat{=}} [\![\varphi]\!]_{\mathcal{S}} \cap [\![\psi]\!]_{\mathcal{S}} \qquad [\![\varphi \vee \psi]\!]_{\mathcal{S}} \mathrel{\hat{=}} [\![\varphi]\!]_{\mathcal{S}} \cup [\![\psi]\!]_{\mathcal{S}}$$

Negation is interpreted as set-complement and infinitary conjunction and disjunction are arbitrary union and intersection, respectively. We now derive abstractions of $\mathcal{P}(Env)$ using abstraction of syntax.

**Sub-language Parameters.** In the previous section, we showed how one can generate a sub-language by starting with a set of syntax trees. We use a notion of parameters to generate syntax trees.

A *$k$-variable predicate* $P(\bar{t})$ is one that contains at most $k$ variables. A *syntax parameter* is a tuple $(S, k)$ consisting of a signature $S \subseteq Sig$ and a value $k$ from $\mathbb{N} \cup \{\infty\}$. A *syntax parameter* $(S, k)$ defines a set of syntax trees $\mathsf{ATree}$ over the symbols $S \cup Var \cup \{\bigwedge, \wedge\}$ and containing only $k$-variable predicates. Note that we always include variables, finite and infinitary conjunction in every syntax abstraction. Define an order $(Q, m) \sqsubseteq (S, n)$ that holds if $Q \subseteq S$ and $m \leq n$

both hold. The parameter order implies subset inclusion of the syntax trees they represent.

The results of the previous section imply that evaluating the grammar Gram over ATree defines a sub-language Lang for every parameter value, and that *struct*(Lang) defines an abstract domain over $\mathcal{P}(Env)$. Let *struct*$(P)$ denote the substructure for the language generated by a parameter $P$. We will illustrate the passage from parameters, via sub-languages to abstract domains. To reduce clutter, we write the parameter $(\{\leq, \equiv_2\}, 3)$ as $(\leq, \equiv_2; 3)$ in figures.

*Example 6.* Consider the transition system $M$, on the left below. States represent the values of a variable $x$. A standard method of abstraction is to partition states using predicates. The result $N$ of partitioning states of $M$ using the predicates $x > 0$ and $x = 0$ is on the right below.


We will see how this abstraction, and others, can be derived syntactically. Three parameters are given below and satisfy the order $P_3 \sqsubseteq P_2 \sqsubseteq P_1$.

$$P_1 \hat{=} (\{=0, >0, \vee, \neg, \mathsf{EX}\}, 1)$$
$$P_2 \hat{=} (\{=0, >0, \vee, \mathsf{EX}\}, 1)$$
$$P_3 \hat{=} (\{=0, >0, \mathsf{EX}\}, 1)$$

The language generated by $P_1$ contains one place predicates for equality with 0 and being strictly greater than 0, is closed under Boolean operations, and has predicates with at most one variable. The predicate $x < 0$ is expressible in this language but $x < y$ is not. The language generated by $P_2$ is only closed under conjunction and disjunction but not negation, while the language generated by $P_3$ is only closed under conjunction. All the languages are also closed under the modal operator EX.

The structures *struct*$(P_i)$ for each parameter contain a lattice representing the semantics of predicates and a transformer for the semantics of EX. These substructures are shown in Figure 2. We only present the calculation of formulae in *struct*$(P_3)$. The elements generated by formulae in $P_1$ and $P_2$ can be derived by Boolean combinations of elements in *struct*$(P_3)$. We show a formula on the left and its interpretation in Figure 2 on the right. Assume that formulae are interpreted over the transition system $M$, with EX interpreted as the predecessor operator.

$$[\![x = 0]\!] = (x = 0) \qquad [\![x > 0]\!] = (x > 0) \qquad [\![\neg x > 0]\!] = (x \leq 0)$$
$$[\![\mathsf{EX}\ x = 0]\!] = (x \leq 0) \qquad [\![\mathsf{EX}\ x > 0]\!] = (x \leq 0) \qquad [\![\neg x = 0]\!] = (x \neq 0)$$

Every predicate in the first lattice represents a set of states of $N$ above and *pre* is the predecessor function for $N$. Note that this abstraction *was not* derived

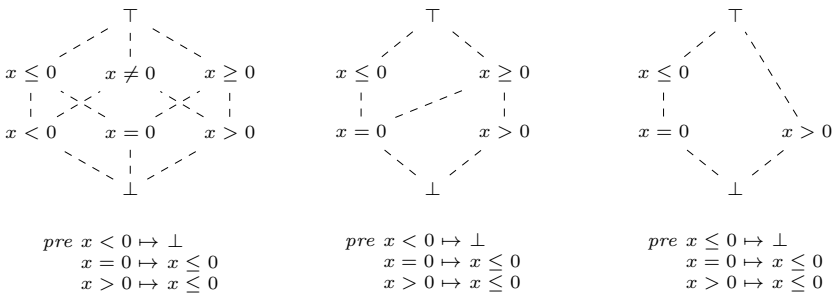



**Fig. 2.** Abstract domains generated signature abstraction

from $N$ but by evaluating formulae. The other two lattices are not closed under Boolean operations, hence cannot be represented as transition systems. ◁

The set of syntax parameters with the order $\sqsubseteq$ forms a lattice, and each element of this lattice defines an abstract domain. The lattice of syntax parameters can be viewed as a lattice of abstract domains in which $(Q, m) \sqsubseteq (S, n)$ implies that the abstract domain defined by $(S, n)$ refines the domain defined by $(Q, m)$. In fact, the lattice of syntax parameters is an abstraction of the lattice of abstract domains [5].

**Generating Abstract Domains.** We now show that appropriate parameters generate abstract domains that are used in practice. We emphasise again that this approach does not solve all the issues that arise in designing an abstraction. Nonetheless, it provides an alternative, syntax-based approach, which we believe is more amenable to automation. We now show that a small range of parameter values generates the specifications of many abstract domains used in practice.

*Relational and Non-relational Abstractions.* A standard form of abstraction is to decouple the relationship between variables. Non-relational domains cannot express facts like $x = y$, while relational abstractions can. The terms weakly-relational refers to domains that express a limited amount of relational information. The octagon domain is weakly relational because it can express $x + y \leq c$, but not $x + y + z \leq c$, because at most two variables can occur in a constraint. A parameter satisfying $(S, m) \sqsubseteq (Sig, 1)$ contains at most one variable per predicate and cannot express relational information. A parameter satisfying $(S, m) \sqsubseteq (Sig, n)$ for a finite, positive integer $n$ generates a weakly relational domain, because some, but not all relational information can be expressed. A parameter $(S, \infty)$ contains predicates with no bound on the number of variables. Such parameters generate domains encoding relational information.

*Temporal Abstractions.* We use the term temporal abstractions for those based in bisimulation, simulation and similar preorders defined on transition systems.

The facts we use about such equivalences are summarised in [2]. Recall that a relation $R \subseteq Env \times Env$ is a simulation if whenever $(s,t)$ is in $R$ and $(s,s')$ is a transition, there must be a transition $(t,t')$ such that $(s',t')$ is in $R$. It is known that bisimulation can be characterised by a logic that is closed under Boolean operations, infinitary conjunction, and $\mathsf{EX}$, and that simulation is characterised by a logic closed under infinitary conjunction, disjunction and $\mathsf{EX}$. We can use these results to specify abstract domains based on bisimulation and simulation.

Every parameter greater than $(\{\bigvee, \neg, \mathsf{EX}\}, 0)$ represents a bisimulation quotient with respect to a given set of predicates. The first abstraction in Figure 2 represents the bisimulation quotient of $M$ using the predicates $x < 0$, $x = 0$ and $x > 0$. Every parameter greater than $(\{\bigvee, \mathsf{EX}\}, 0)$, containing infinitary disjunction but not necessarily negation, represents a simulation preserving domain. Such domains are studied in [19].

*Numeric Abstractions.* We now discuss parameter settings that generate standard abstract domains. For complete rigour, each claim that follows should be accompanied with a proof that the generated domain represents the claimed, existing domain.

- *Affine Equalities* $(\{+, \mathbb{N}, =\}, \infty)$. This abstraction contains conjunctions of constraints of the form $a_1x_1 + \cdots a_nx_n = k$.
- *Affine Congruences* $(\{+, \mathbb{N}, \equiv_k\}, \infty)$. This abstraction contains conjunctions of constraints of the form $a_1x_1 + \cdots a_nx_n \equiv_k m$, stating that the constraint on the left is congruent to $m$, modulo $k$.
- *Intervals* $(\{\mathbb{N}, \leq\}, 1)$. If at most one variable per predicate is allowed when using inequalities, we have constraints of the form $0 \leq x \wedge x \leq 3$, which encode that $x$ is in the interval $[0, 3]$.
- *Constants* $(\{\mathbb{N}, =\}, 1)$. If at most one variable per predicate is allowed when using equalities, we have constraints of the form $x = 3$. Since no two distinct equalities are satisfiable, all conjunctions of formulae become $\mathsf{false}$. This domain is used in constant propagation.
- *Parity* $(\{0, 1, \equiv_2\}, 1)$. If the domain is non-relational and the only predicate is $\equiv_2$, we obtain the parity domain.
- *Signs* $(\{0, <, =, >\}, 1)$. A non-relational domain in which every variable can only be compared with 0 is the signs domain. The parameter above only generates the 5 element signs domain containing the predicates $x < 0$, $x = 0$ and $x > 0$. If extended with disjunction, we obtain the 8 element signs domain closed under Boolean operations.

We emphasise that a signature can contain infinitely many symbols, and the resulting domain, infinitely many elements. Several numeric domains mentioned above have infinitely many elements. Note also that the domains of constants, affine equalities and congruences, all contain infinitely many inequalities but have finite height. Logically, such domains correspond to logics in which there are no infinite sequences of strict implications. For these reasons, generating domains by abstraction of syntax is difficult to automate. Predicate abstraction [1] applies only to finite sets of predicates, but is automatic. The parameters for some domains we discussed are shown in Figure 3.

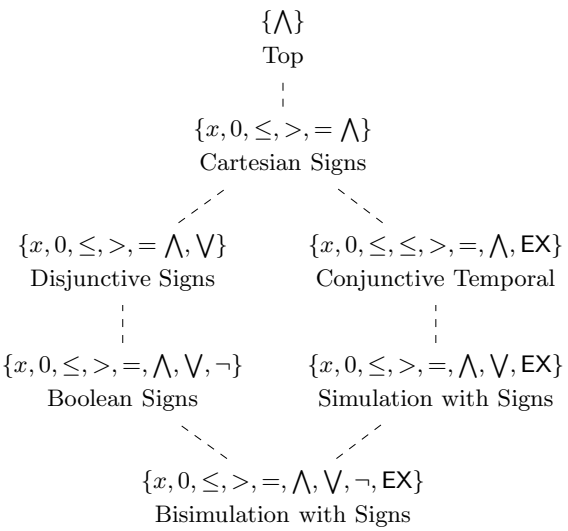



**Fig. 3.** A lattice of signatures that generates the abstract domains shown

## 5 Related Work

The ideas that domains can be viewed as logics, and manipulation of grammars can be understood as underapproximation is folklore in the abstract interpretation community. This paper has attempted to formalise this view, and to combine two folklore ideas to obtain a new approach to deriving abstract domains.

The technical background for this paper was drawn from language theory, abstract interpretation, and lattice theory. The language-theoretic basis of our work is the Chomsky-Schützenberger theorem, which characterises context free languages as fixed points. Ginsburg and Rice [14] extended this characterisation to recursively enumerable languages. Istrail [15] showed that concatenation, union, intersection and substitution on languages suffice for the fixed point characterisation. It follows that the language of a BNF grammar has a fixed point characterisation. See Cousot and Cousot [6,7] or Paulson [18], for related discussions of inductive definitions.

We use abstract interpretation to approximate the fixed point defined by a BNF grammar. Cousot and Cousot's bi-inductive semantics [10] has similar motivations with important differences. If we only consider the languages that can be derived, bi-inductive semantics uses posets and possibly non-monotone functions, hence is a strictly more general framework than ours. However, we use BNF grammars, which is how logics are specified in practice. There is much work on grammar abstractions [3,8,17,22], with a focus on program analysis. Our focus is language generation from BNF, akin to invariant generation from a program, while the cited methods above are similar to analysis of a transition system.

Abstract interpretation has also been combined with parsing algorithms [9,21]. Parsing is a language recognition, not language generation task.

## 6 Conclusion

Approximation of semantics, as formalised by abstract interpretation, is fundamental to tractable reasoning about programs. Abstract reasoning depends on manually defined artefacts such as abstract domains. We have studied the problem of developing a systematic approach to the narrow task of specifying the syntax of an abstract domain. The solution proposed in this paper was to use meta-syntax, a variant of BNF-style grammars to specify abstract domains. Our contribution is to show that abstraction of grammars provides a formal framework for generating specifications of abstract domains. We demonstrated this framework with a simple case study that covered a broad range of abstractions.

The method in this paper is a first step to deriving a systematic framework for thinking about and manipulating abstract domains. We have presented no algorithms for generically deriving implementations of abstract domains. Most numeric abstractions we discussed are sub-logics of Presburger arithmetic, which is decidable. One question is whether implementations of infinite domains over sub-logics of Presburger arithmetic can be derived automatically. Specifically, given a decision procedure for Presburger arithmetic, and a syntactic restriction of formulae can we automatically synthesise abstract transformers for domains such as intervals or constants.

Another problem in dire need of a rigorous framework is that of designing operators to ensure convergence of fixed point iterations. There are results showing that convergence operators cannot be monotone in general. We have shown that abstraction of syntax, in certain situations yields overapproximating abstractions. Another interpretation of this result, is that there are situations in which a syntactically defined subset of a lattice only admits a non-monotone abstraction function. A question arising from our work is whether abstraction of syntax can be used to define convergence acceleration operations. We observe that several standard interval widening operators can be defined in this manner. Investigating such questions further is future work.

**Acknowledgements.** We thank the reviewers for their careful reading and positive feedback.

## References

1. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian Abstraction for Model Checking C Programs. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 268–283. Springer, Heidelberg (2001)
2. Blackburn, P., de Rijke, M., Venema, Y.: Modal Logic. Cambridge University Press, Cambridge (2001)

3. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise Analysis of String Expressions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 1–18. Springer, Heidelberg (2003)
4. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages, pp. 238–252. ACM Press (1977)
5. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Principles of Programming Languages, pp. 269–282. ACM Press (1979)
6. Cousot, P., Cousot, R.: Inductive definitions, semantics and abstract interpretations. In: Principles of Programming Languages, pp. 83–94. ACM Press (1992)
7. Cousot, P., Cousot, R.: Compositional and Inductive Semantic Definitions in Fixpoint, Equational, Constraint, Closure-condition, Rule-based and Game-Theoretic Form. In: Wolper, P. (ed.) CAV 1995. LNCS, vol. 939, pp. 293–308. Springer, Heidelberg (1995)
8. Cousot, P., Cousot, R.: Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In: Functional Programming Languages and Computer Architecture, June 25–28, pp. 170–181. ACM Press (1995)
9. Cousot, P., Cousot, R.: Grammar Analysis and Parsing by Abstract Interpretation. In: Reps, T., Sagiv, M., Bauer, J. (eds.) Wilhelm Festschrift. LNCS, vol. 4444, pp. 175–200. Springer, Heidelberg (2007)
10. Cousot, P., Cousot, R.: Bi-inductive structural semantics. Information and Computation 207, 258–283 (2009)
11. Davey, B.A., Priestley, H.A.: Introduction to lattices and order. Cambridge University Press, Cambridge (1990)
12. Filé, G., Giacobazzi, R., Ranzato, F.: A unifying view of abstract domain design. ACM Computing Surveys 28(2), 333–336 (1996)
13. Giacobazzi, R., Ranzato, F., Scozzari, F.: Making abstract interpretations complete. Journal of the ACM 47(2), 361–416 (2000)
14. Ginsburg, S., Rice, H.G.: Two families of languages related to Algol. Journal of the ACM 9, 350–371 (1962)
15. Istrail, S.: Generalization of the Ginsburg-Rice Schützenberger fixed-point theorem for context-sensitive and recursive-enumerable languages. Theoretical Computer Science 18, 333–341 (1982)
16. Jensen, T.P.: Strictness Analysis in Logical Form. In: Hughes, J. (ed.) FPCA 1991. LNCS, vol. 523, pp. 352–366. Springer, Heidelberg (1991)
17. Okhotin, A., Reitwießner, C.: Conjunctive grammars with restricted disjunction. Theoretical Computer Science 411, 2559–2571 (2010)
18. Paulson, L.C.: A Fixedpoint Approach to Implementing (Co)Inductive Definitions. In: Bundy, A. (ed.) CADE 1994. LNCS, vol. 814, pp. 148–161. Springer, Heidelberg (1994)
19. Ranzato, F., Tapparo, F.: Generalized strong preservation by abstract interpretation. J. of Logic and Computation 17(1), 157–197 (2007)
20. Schmidt, D.A.: Internal and External Logics of Abstract Interpretations. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 263–278. Springer, Heidelberg (2008)
21. Schmitz, S.: Approximating Context-Free Grammars for Parsing and Verification. Thèse de doctorat, Université de Nice-Sophia Antipolis, France (2007)
22. Wassermann, G., Gould, C., Su, Z., Devanbu, P.: Static checking of dynamically generated queries in database applications. ACM Transactions on Software Engineering Methodology 16(4), 14 (2007)