Boolean Functions for Finite-Tree Dependencies*

Roberto Bagnara[1], Enea Zaffanella[1], Roberta Gori[2], and Patricia M. Hill[3]

[1] Department of Mathematics, University of Parma, Italy.
{bagnara,zaffanella}@cs.unipr.it
[2] Department of Computer Science, University of Pisa, Italy.
gori@di.unipi.it
[3] School of Computing, University of Leeds, United Kingdom.
hill@comp.leeds.ac.uk

Abstract. Several logic-based languages, such as Prolog II and its successors, SICStus Prolog and Oz, offer a computation domain including *rational trees* that allow for increased expressivity and faster unification. Unfortunately, the use of infinite rational trees has problems. For instance, many of the built-in and library predicates are ill-defined for such trees and need to be supplemented by run-time checks whose cost may be significant. In a recent paper [3], we have proposed a data-flow analysis called *finite-tree analysis* aimed at identifying those program variables (the *finite variables*) that are not currently bound to infinite terms. Here we present a domain of Boolean functions, called *finite-tree dependencies* that precisely captures how the finiteness of some variables influences the finiteness of other variables. We also summarize our experimental results showing how finite-tree analysis, enhanced with finite-tree dependencies is a practical means of obtaining precise finiteness information.

1 Introduction

Many logic-based languages refer to a computation domain of *rational trees*. While rational trees allow for increased expressivity, they also have a surprising number of problems. (See [4] for a survey of known applications of rational trees and a detailed account of many of the problems caused by their use.) Some of these problems are so serious that rational trees must be used in a very controlled way, disallowing infinite trees in any context where they are “dangerous”. This, in turn, causes a secondary problem: in order to disallow infinite trees in selected contexts, one must first detect them, an operation that may be expensive.

In [4], we have introduced a composite abstract domain, $H \times P$, for finite-tree analysis. The H domain, written with the initial of *Herbrand* and called the *finiteness* component, is the direct representation of the property of interest: a set of variables guaranteed to be bound to finite terms. The generic domain P (the *parameter* of the construction) provides sharing information that can include,

* The MURST project, “Certificazione automatica di programmi mediante interpretazione astratta”, partly supported the work of the first two authors and EPSRC grant M05645 partly supported the work of the second and fourth authors.

apart from variable aliasing, groundness, linearity, freeness and any other kind of information that can improve the precision on these components, such as explicit structural information. Sharing information is exploited in $H \times P$ for two purposes: detecting when new infinite terms are possibly created (this is done along the lines of [22]) and confining the propagation of those terms as much as possible. As shown in [3, 4], by giving a generic specification for this parameter component in terms of the *abstract queries* it supports (in the style of the *open product* construct [12]), it is possible to define and establish the correctness of the abstract operators on the finite-tree domain independently from any particular domain for sharing analysis.

The domain $H \times P$ captures the negative aspect of term-finiteness, that is, the circumstances under which finiteness can be lost. However, term-finiteness has also a positive aspect: there are cases where a variable is granted to be bound to a finite term and this knowledge can be propagated to other variables. Guarantees of finiteness are provided by several built-ins like `unify_with_occurs_check/2`, `var/1`, `name/2`, all the arithmetic predicates, besides those explicitly provided to test for term-finiteness such as the `acyclic_term/1` predicate of SICStus Prolog. The information encoded by H is *attribute independent* [14], which means that each variable is considered in isolation. What is missing is information concerning how finiteness of one variable affects the finiteness of other variables. This kind of information, usually called *relational information*, is not captured at all by H and is only partially captured by the composite domain $H \times P$ of [4].

Here we present a domain of Boolean functions that precisely captures how the finiteness of some variables influences the finiteness of other variables. This domain of *finite-tree dependencies* provides relational information that is important for the precision of the overall finite-tree analysis. It also combines obvious similarities, interesting differences and somewhat unexpected connections with classical domains for *groundness dependencies*.

Finite-tree and groundness dependencies are similar in that they both track *covering* information (a term s covers t if all the variables in t also occur in s) and share several abstract operations. However, they are different because covering does not tell the whole story. Suppose x and y are free variables before either the unification $x = f(y)$ or the unification $x = f(x, y)$ are executed. In both cases, x will be ground if and only if y will be so. However, when $x = f(y)$ is the performed unification, this equivalence will also carry over to finiteness. In contrast, when the unification is $x = f(x, y)$, x will never be finite and will be totally independent, as far as finiteness is concerned, from y. Among the unexpected connections is the fact that finite-tree dependencies can improve the groundness information obtained by the usual approaches to groundness analysis.

The paper is structured as follows: the required notations and preliminary concepts are given in Section 2; the concrete domain for the analysis is presented in Section 3; Section 4 introduces the use of Boolean functions for tracking finite-tree dependencies, whereas Section 5 illustrates the interaction between groundness and finite-tree dependencies. Our experimental results are presented in Section 6. The paper concludes in Section 7.

2 Preliminaries

2.1 Infinite Terms and Substitutions

For a set S, $\wp(S)$ is the powerset of S, $\wp_{\mathrm{f}}(S)$ is the set of all the *finite* subsets of S, whereas $\# S$ denotes the cardinality of S. Let *Sig* denote a possibly infinite set of function symbols, ranked over the set of natural numbers and *Vars* a denumerable set of variable symbols, disjoint from *Sig*. Then *Terms* denotes the free algebra of all (possibly infinite) terms in the signature *Sig* having variables in *Vars*. It is assumed that *Sig* contains at least two distinct function symbols, one having rank 0 and one with rank greater than 0 (so that there exist finite and infinite terms both with and without variables). If $t \in \mathit{Terms}$ then $\mathrm{vars}(t)$ denotes the set of variables occurring in t. If $\mathrm{vars}(t) = \varnothing$ then t is said to be *ground*; t is a *finite term* (or *Herbrand term*) if it contains a finite number of occurrences of function symbols. The sets of all ground and finite terms are denoted by *GTerms* and *HTerms*, respectively.

A *substitution* is a total function $\sigma \colon \mathit{Vars} \to \mathit{HTerms}$ that is the identity almost everywhere; in other words, the *domain of* σ, which is defined as $\mathrm{dom}(\sigma) \stackrel{\mathrm{def}}{=} \bigl\{\, x \in \mathit{Vars} \bigm| \sigma(x) \neq x \,\bigr\}$, is a finite set of variables. If $x \in \mathit{Vars}$ and $t \in \mathit{HTerms} \setminus \{x\}$, then $x \mapsto t$ is called a *binding*. The set of all bindings is denoted by *Bind*. Substitutions are conveniently denoted by the set of their bindings. Accordingly, a substitution σ is identified with the (finite) set $\bigl\{\, x \mapsto \sigma(x) \bigm| x \in \mathrm{dom}(\sigma) \,\bigr\}$. We denote by $\mathrm{vars}(\sigma)$ the set of all variables occurring in the bindings of σ.

A substitution of the form $\{x_1 \mapsto x_2, \ldots, x_{n-1} \mapsto x_n, x_n \mapsto x_1\}$ is *circular* if and only if $n > 1$ and $x_1, \ldots, x_n$ are distinct variables. A substitution is in *rational solved form* if it has no circular subset. The set of all substitutions in rational solved form is denoted by *RSubst*.

Given a substitution $\sigma \colon \mathit{Vars} \to \mathit{HTerms}$, the symbol '$\sigma$' also denotes the function $\sigma \colon \mathit{HTerms} \to \mathit{HTerms}$ defined as usual. That is, for each $t \in \mathit{HTerms}$, $\sigma(t)$ is the term obtained by replacing each occurrence of each variable x in t by the term $\sigma(x)$. If $t \in \mathit{HTerms}$, we write $t\sigma$ to denote $\sigma(t)$. Let $s \in \mathit{HTerms}$ and $\sigma \in \mathit{RSubst}$. Then $\sigma^0(s) \stackrel{\mathrm{def}}{=} s$ and $\sigma^i(s) \stackrel{\mathrm{def}}{=} \sigma\bigl(\sigma^{i-1}(s)\bigr)$ for all $i \in \mathbb{N}$, $i > 0$. Thus the sequence of finite terms $\sigma^0(s), \sigma^1(s), \ldots$ converges to a (possibly infinite) term, denoted by $\sigma^\infty(s)$ [17, 18].

2.2 Equations

An *equation* has the form $s = t$ where $s, t \in \mathit{HTerms}$. *Eqs* denotes the set of all equations. A substitution σ may be regarded as a finite set of equations, that is, as the set $\{\, x = t \mid x \mapsto t \in \sigma \,\}$. We say that a set of equations e is in *rational solved form* if $\bigl\{\, s \mapsto t \bigm| (s = t) \in e \,\bigr\} \in \mathit{RSubst}$. In the rest of the paper, we will often write a substitution $\sigma \in \mathit{RSubst}$ to denote a set of equations in rational solved form (and vice versa).

Some logic-based languages, such as Prolog II, SICStus and Oz, are based on $\mathcal{RT}$, the theory of rational trees [9, 10]. This is a syntactic equality theory (i.e., a theory where the function symbols are uninterpreted), augmented with a *uniqueness axiom* for each substitution in rational solved form. It is worth noting that any set of equations in rational solved form is, by definition, satisfiable in $\mathcal{RT}$.

Given a set of equations $e \in \wp_\mathrm{f}(\mathit{Eqs})$ that is satisfiable in $\mathcal{RT}$, a substitution $\sigma \in \mathit{RSubst}$ is called a *solution for e in $\mathcal{RT}$* if $\mathcal{RT} \vdash \forall(\sigma \rightarrow e)$, i.e., if every model of the theory $\mathcal{RT}$ is also a model of the first order formula $\forall(\sigma \rightarrow e)$. If in addition $\mathrm{vars}(\sigma) \subseteq \mathrm{vars}(e)$, then σ is said to be a *relevant* solution for e. If $\mathcal{RT} \vdash \forall(\sigma \leftrightarrow e)$, then σ is a *most general solution for e in $\mathcal{RT}$*. The set of all the relevant most general solution for e in $\mathcal{RT}$ will be denoted by $\mathrm{mgs}(e)$.

The function $\downarrow(\cdot) \colon \mathit{RSubst} \rightarrow \wp(\mathit{RSubst})$ is defined, for each $\sigma \in \mathit{RSubst}$, by $\downarrow \sigma \stackrel{\mathrm{def}}{=} \bigl\{ \tau \in \mathit{RSubst} \bigm| \exists \sigma' \in \mathit{RSubst} \,.\, \tau \in \mathrm{mgs}(\sigma \cup \sigma') \bigr\}$. The next result shows that $\downarrow(\cdot)$ corresponds to the closure by entailment in $\mathcal{RT}$.

Proposition 1. *Let $\sigma \in \mathit{RSubst}$. Then $\downarrow \sigma = \bigl\{ \tau \in \mathit{RSubst} \bigm| \mathcal{RT} \vdash \forall(\tau \rightarrow \sigma) \bigr\}$.*

2.3 Boolean Functions

Boolean functions have already been extensively used for data-flow analysis of logic-based languages. An important class of these functions used for tracking groundness dependencies is *Pos* [1]. This domain was introduced in [19] under the name *Prop* and further refined and studied in [11, 20].

Boolean functions are based on the notion of Boolean valuation.

Definition 2. (Boolean valuations.) *Let $\mathit{VI} \in \wp_\mathrm{f}(\mathit{Vars})$ and $\mathbb{B} \stackrel{\mathrm{def}}{=} \{0, 1\}$. The set of* Boolean valuations over VI *is $\mathit{Bval} \stackrel{\mathrm{def}}{=} \mathit{VI} \rightarrow \mathbb{B}$. For each $a \in \mathit{Bval}$, each $x \in \mathit{VI}$, and each $c \in \mathbb{B}$ the valuation $a[c/x] \in \mathit{Bval}$ is given, for each $y \in \mathit{VI}$, by*

$$a[c/x](y) \stackrel{\mathrm{def}}{=} \begin{cases} c, & \text{if } x = y; \\ a(y), & \text{otherwise.} \end{cases}$$

If $X = \{x_1, \ldots, x_k\} \subseteq \mathit{VI}$, then $a[c/X]$ denotes $a[c/x_1] \cdots [c/x_k]$.

Bval contains the distinguished elements $\mathbf{0} \stackrel{\mathrm{def}}{=} \lambda x \in \mathit{VI} \,.\, 0$ and $\mathbf{1} \stackrel{\mathrm{def}}{=} \lambda x \in \mathit{VI} \,.\, 1$.

Definition 3. (Boolean functions.) *The set of* Boolean functions over VI *is $\mathit{Bfun} \stackrel{\mathrm{def}}{=} \mathit{Bval} \rightarrow \mathbb{B}$. Bfun is partially ordered by the relation $\models$ where, for each $\phi, \psi \in \mathit{Bfun}$,*

$$\phi \models \psi \quad \stackrel{\mathrm{def}}{\iff} \quad \bigl(\forall a \in \mathit{Bval} : \phi(a) = 1 \implies \psi(a) = 1\bigr).$$

The distinguished elements $\top, \bot \in \mathit{Bfun}$ are defined by $\bot \stackrel{\mathrm{def}}{=} \lambda a \in \mathit{Bval} \,.\, 0$ and $\top \stackrel{\mathrm{def}}{=} \lambda a \in \mathit{Bval} \,.\, 1$. respectively. For each $\phi \in \mathit{Bfun}$, $x \in \mathit{VI}$, and $c \in \mathbb{B}$, the function $\phi[c/x] \in \mathit{Bfun}$ is given, for each $a \in \mathit{Bval}$, by $\phi[c/x](a) \stackrel{\mathrm{def}}{=} \phi\bigl(a[c/x]\bigr)$. When $X \subseteq \mathit{VI}$, $\phi[c/X]$ is defined in the expected way. If $\phi \in \mathit{Bfun}$ and $x, y \in \mathit{VI}$ the function $\phi[y/x] \in \mathit{Bfun}$ is given by $\phi[y/x](a) \stackrel{\mathrm{def}}{=} \phi\bigl(a\bigl[a(y)/x\bigr]\bigr)$, for each

$a \in \mathit{Bval}$. Boolean functions are constructed from the elementary functions corresponding to variables and by means of the usual logical connectives. Thus x denotes the Boolean function ϕ such that, for each $a \in \mathit{Bval}$, $\phi(a) = 1$ if and only if $a(x) = 1$. For $\phi_1, \phi_2 \in \mathit{Bfun}$, we write $\phi_1 \wedge \phi_2$ to denote the function ϕ such that, for each $a \in \mathit{Bval}$, $\phi(a) = 1$ if and only if both $\phi_1(a) = 1$ and $\phi_2(a) = 1$. A variable is restricted away using Schröder's elimination principle [21]: $\exists x \mathrel{.} \phi \stackrel{\text{def}}{=} \phi[1/x] \vee \phi[0/x]$. Note that existential quantification is both monotonic and extensive on *Bfun*. The other Boolean connectives and quantifiers are handled similarly.

Pos $\subset$ *Bfun* consists precisely of those functions assuming the true value under the *everything-is-true* assignment, i.e., $\mathit{Pos} \stackrel{\text{def}}{=} \bigl\{\, \phi \in \mathit{Bfun} \bigm| \phi(\mathbf{1}) = 1 \,\bigr\}$. For each $\phi \in \mathit{Bfun}$, the *positive part of* ϕ, denoted $\operatorname{pos}(\phi)$, is the strongest *Pos* formula that is entailed by ϕ. Formally, $\operatorname{pos}(\phi) \stackrel{\text{def}}{=} \phi \vee \bigwedge \mathit{VI}$.

For each $\phi \in \mathit{Bfun}$, the set of *variables necessarily true for* ϕ and the set of *variables necessarily false for* ϕ are given, respectively, by

$$\begin{aligned}
\operatorname{true}(\phi) &\stackrel{\text{def}}{=} \bigl\{\, x \in \mathit{VI} \bigm| \forall a \in \mathit{Bval} : \phi(a) = 1 \implies a(x) = 1 \,\bigr\},\\
\operatorname{false}(\phi) &\stackrel{\text{def}}{=} \bigl\{\, x \in \mathit{VI} \bigm| \forall a \in \mathit{Bval} : \phi(a) = 1 \implies a(x) = 0 \,\bigr\}.
\end{aligned}$$

3 The Concrete Domain

A knowledge of the basic concepts of abstract interpretation theory [13, 15] is assumed. In this paper, the concrete domain consists of pairs of the form (Σ, V), where V is a finite set of *variables of interest* and Σ is a (possibly infinite) set of substitutions in rational solved form.

Definition 4. (The concrete domain.) *Let* $\mathcal{D}^\flat \stackrel{\text{def}}{=} \wp(\mathit{RSubst}) \times \wp_{\mathrm{f}}(\mathit{Vars})$. *If* $(\Sigma, V) \in \mathcal{D}^\flat$, *then* (Σ, V) *represents the (possibly infinite) set of first-order formulas* $\bigl\{\, \exists \Delta \mathrel{.} \sigma \bigm| \sigma \in \Sigma, \Delta = \operatorname{vars}(\sigma) \setminus V \,\bigr\}$ *where* σ *is interpreted as the logical conjunction of the equations corresponding to its bindings. The operation of projecting* $x \in \mathit{Vars}$ *away from* $(\Sigma, V) \in \mathcal{D}^\flat$ *is defined as follows:*

$$\exists\, x \mathrel{.} (\Sigma, V) \stackrel{\text{def}}{=} \left\{ \sigma' \in \mathit{RSubst} \;\middle|\; \begin{array}{l} \sigma \in \Sigma, \overline{V} = \mathit{Vars} \setminus V, \\ \mathcal{RT} \vdash \forall\bigl(\exists \overline{V} \mathrel{.} (\sigma' \leftrightarrow \exists x \mathrel{.} \sigma)\bigr) \end{array} \right\}.$$

The concrete element $\bigl(\bigl\{\{x \mapsto f(y)\}\bigr\}, \{x, y\}\bigr)$ expresses a dependency between x and y. In contrast, $\bigl(\bigl\{\{x \mapsto f(y)\}\bigr\}, \{x\}\bigr)$ only constrains x. The same concept can be expressed by saying that the variable name 'y' matters in the first case but not in the second. Thus the set of variables of interest is crucial for defining the meaning of the concrete and abstract descriptions. Despite this, always specifying the set of variables of interest would significantly clutter the presentation. Moreover, most of the needed functions on concrete and abstract descriptions preserve the set of variables of interest. For these reasons, we assume there exists a set $\mathit{VI} \in \wp_{\mathrm{f}}(\mathit{Vars})$ containing, at each stage of the analysis, the current variables of interest. As a consequence, when the context makes it clear, we will write $\Sigma \in \mathcal{D}^\flat$ as a shorthand for $(\Sigma, \mathit{VI}) \in \mathcal{D}^\flat$.

3.1 Operators on Substitutions in Rational Solved Form

There are cases when an analysis tries to capture properties of the particular substitutions computed by a specific rational unification algorithm. This is the case, for example, when the analysis needs to track structure sharing for the purpose of compile-time garbage collection, or provide upper bounds to the amount of memory needed to perform a given computation. More often the interest is on properties of the rational trees themselves. In these cases it is possible to define abstraction and concretization functions that are independent from the finite representations actually considered. Moreover, it is important that these functions precisely capture the properties under investigation so as to avoid any unnecessary precision loss.

Pursuing this goal requires the ability to observe properties of (infinite) rational trees while just dealing with one of their finite representations. This is not always an easy task since even simple properties can be "hidden" when using non-idempotent substitutions. For instance, when $\sigma^\infty(x) \in \mathit{GTerms} \setminus \mathit{HTerms}$ is an infinite and ground rational tree, all of its finite representations in RSubst will map the variable x into a finite term that is not ground.

These are the motivations behind the introduction of two computable operators on substitutions that will be used later to define the concretization functions for the considered abstract domains. First, the groundness operator 'gvars' captures the set of variables that are mapped to ground rational trees by 'σ^∞'. We define it by means of the *occurrence operator* 'occ' introduced in [16].

Definition 5. (Occurrence and groundness operators.) *For each $n \in \mathbb{N}$, the* occurrence function $\mathrm{occ}_n \colon \mathit{RSubst} \times \mathit{Vars} \to \wp_\mathrm{f}(\mathit{Vars})$ *is defined, for each $\sigma \in \mathit{RSubst}$ and each $v \in \mathit{Vars}$, by*

$$\mathrm{occ}_n(\sigma, v) \stackrel{\mathrm{def}}{=} \begin{cases} \{v\} \setminus \mathrm{dom}(\sigma), & \text{if } n = 0; \\ \bigl\{\, y \in \mathit{Vars} \bigm| \mathrm{vars}(y\sigma) \cap \mathrm{occ}_{n-1}(\sigma, v) \neq \varnothing \,\bigr\}, & \text{if } n > 0. \end{cases}$$

The occurrence operator $\mathrm{occ} \colon \mathit{RSubst} \times \mathit{Vars} \to \wp_\mathrm{f}(\mathit{Vars})$ *is given, for each substitution $\sigma \in \mathit{RSubst}$ and $v \in \mathit{Vars}$, by $\mathrm{occ}(\sigma, v) \stackrel{\mathrm{def}}{=} \mathrm{occ}_\ell(\sigma, v)$, where $\ell = \#\,\sigma$.*

The groundness operator $\mathrm{gvars} \colon \mathit{RSubst} \to \wp_\mathrm{f}(\mathit{Vars})$ *is given, for each substitution $\sigma \in \mathit{RSubst}$, by*

$$\mathrm{gvars}(\sigma) \stackrel{\mathrm{def}}{=} \bigl\{\, y \in \mathrm{dom}(\sigma) \bigm| \forall v \in \mathrm{vars}(\sigma) : y \notin \mathrm{occ}(\sigma, v) \,\bigr\}.$$

The finiteness operator 'hvars', introduced in [4], captures the set of variables that 'σ^∞' maps to finite terms.

Definition 6. (Finiteness operator.) *For each $n \in \mathbb{N}$, the* finiteness function $\mathrm{hvars}_n \colon \mathit{RSubst} \to \wp(\mathit{Vars})$ *is defined, for each $\sigma \in \mathit{RSubst}$, by*

$$\mathrm{hvars}_n(\sigma) \stackrel{\mathrm{def}}{=} \begin{cases} \mathit{Vars} \setminus \mathrm{dom}(\sigma), & \text{if } n = 0; \\ \mathrm{hvars}_{n-1}(\sigma) \cup \bigl\{\, y \in \mathrm{dom}(\sigma) \bigm| \mathrm{vars}(y\sigma) \subseteq \mathrm{hvars}_{n-1}(\sigma) \,\bigr\}, & \text{if } n > 0. \end{cases}$$

The finiteness operator $\mathrm{hvars}\colon \mathit{RSubst} \to \wp(\mathit{Vars})$ *is given, for each substitution* $\sigma \in \mathit{RSubst}$, *by* $\mathrm{hvars}(\sigma) \stackrel{\mathrm{def}}{=} \mathrm{hvars}_\ell(\sigma)$, *where* $\ell \stackrel{\mathrm{def}}{=} \# \sigma$.

Example 7. Let

$$\begin{aligned}\sigma &= \bigl\{ x \mapsto f(y,z), y \mapsto g(z,x), z \mapsto f(a) \bigr\}, \\ \tau &= \bigl\{ v \mapsto g(z,w), x \mapsto f(y), y \mapsto g(w), z \mapsto f(v) \bigr\},\end{aligned}$$

where $\mathrm{vars}(\sigma) \cup \mathrm{vars}(\tau) = \{v,w,x,y,z\}$. Then $\mathrm{gvars}(\sigma) \cap \mathrm{vars}(\sigma) = \{x,y,z\}$ and $\mathrm{hvars}(\tau) \cap \mathrm{vars}(\tau) = \{w,x,y\}$.

The following proposition states how 'gvars' and 'hvars' behave with respect to the further instantiation of variables.

Proposition 8. *Let* $\sigma, \tau \in \mathit{RSubst}$, *where* $\tau \in {\downarrow} \sigma$. *Then*

$$\mathrm{hvars}(\sigma) \supseteq \mathrm{hvars}(\tau), \tag{8a}$$

$$\mathrm{gvars}(\sigma) \cap \mathrm{hvars}(\sigma) \subseteq \mathrm{gvars}(\tau) \cap \mathrm{hvars}(\tau). \tag{8b}$$

4 Finite-Tree Dependencies

Any finite-tree domain must keep track of those variables that are definitely bound to finite terms, since this is the final information delivered by the analysis. In [4] we have introduced the composite abstract domain $H \times P$, where the set of such variables is explicitly represented in the *finiteness component* H.

Definition 9. (The finiteness component H**.)** *The set* $H \stackrel{\mathrm{def}}{=} \wp(\mathit{VI})$, *partially ordered by reverse subset inclusion, is called* finiteness component. *The concretization function* $\gamma_H \colon H \to \wp(\mathit{RSubst})$ *is given, for each* $h \in H$, *by*

$$\gamma_H(h) \stackrel{\mathrm{def}}{=} \bigl\{ \sigma \in \mathit{RSubst} \bigm| \mathrm{hvars}(\sigma) \supseteq h \bigr\}.$$

As proven in [3], equivalent substitutions in rational solved form have the same finiteness abstraction.

Proposition 10. *Let* $\sigma, \tau \in \mathit{RSubst}$, *where* $\sigma \in \gamma_H(h)$ *and* $\mathcal{RT} \vdash \forall(\sigma \leftrightarrow \tau)$. *Then* $\tau \in \gamma_H(h)$.

The precision of the finite-tree analysis of [4] is highly dependent on the precision of the generic component P. As explained before, the information provided by P on groundness, freeness, linearity, and sharing of variables is exploited, in the combination $H \times P$, to circumscribe as much as possible the creation and propagation of cyclic terms. However, finite-tree analysis can also benefit from other kinds of relational information. In particular, we now show how *finite-tree dependencies* allow to obtain a positive propagation of finiteness information.

Let us consider the finite terms $t_1 = f(x)$, $t_2 = g(y)$, and $t_3 = h(x,y)$: it is clear that, for each assignment of rational terms to x and y, t_3 is finite if and only if t_1 and t_2 are so. We can capture this by the Boolean formula $t_3 \leftrightarrow (t_1 \wedge t_2)$. The important point to notice is that this dependency will keep holding for any further simultaneous instantiation of t_1, t_2, and t_3. In other words, such dependencies are preserved by forward computations (which proceed by consistently instantiating program variables).

Consider $x \mapsto t \in \mathit{Bind}$ where $t \in \mathit{HTerms}$ and $\mathrm{vars}(t) = \{y_1, \dots, y_n\}$. After this binding has been successfully applied, the destinies of x and t concerning term-finiteness are tied together: forever. This tie can be described by the dependency formula

$$x \leftrightarrow (y_1 \wedge \cdots \wedge y_n), \tag{2}$$

meaning that x will be bound to a finite term if and only if y_i is bound to a finite term, for each $i = 1, \dots, n$. While the dependency expressed by (2) is a correct description of any computation state following the application of the binding $x \mapsto t$, it is not as precise as it could be. Suppose that x and y_k are indeed the same variable. Then (2) is logically equivalent to

$$x \rightarrow (y_1 \wedge \cdots \wedge y_{k-1} \wedge y_{k+1} \wedge \cdots \wedge y_n). \tag{3}$$

Correct: whenever x is bound to a finite term, all the other variables will be bound to finite terms. The point is that x has just been bound to a non-finite term, irrevocably: no forward computation can change this. Thus, the implication (3) holds vacuously. A more precise and correct description for the state of affairs caused by the cyclic binding is, instead, the negated atom $\neg x$, whose intuitive reading is "x is not (and never will be) finite."

We are building an abstract domain for finite-tree dependencies where we are making the deliberate choice of including only information that cannot be withdrawn by forward computations. The reason for this choice is that we want the concrete constraint accumulation process to be paralleled, at the abstract level, by another constraint accumulation process: logical conjunction of Boolean formulas. For this reason, it is important to distinguish between *permanent* and *contingent* information. Permanent information, once established for a program point p, maintains its validity in all points that follow p in any forward computation. Contingent information, instead, does not carry its validity beyond the point where it is established. An example of contingent information is given by the h component of $H \times P$: having $x \in h$ in the description of some program point means that x is definitely bound to a finite term *at that point*; nothing is claimed about the finiteness of x at later program points and, in fact, unless x is ground, x can still be bound to a non-finite term. However, if at some program point x is finite and ground, then x will remain finite. In this case we will ensure our Boolean dependency formula entails the positive atom x.

At this stage, we already know something about the abstract domain we are designing. In particular, we have positive and negated atoms, the requirement of describing program predicates of any arity implies that arbitrary conjunctions of these atomic formulas must be allowed and, finally, it is not difficult to observe that the merge-over-all-paths operations [13] will be logical disjunction, so that the domain will have to be closed under this operation. This means that the carrier of our domain must be able to express any Boolean function: *Bfun* is the carrier.

Definition 11. **($\gamma_F \colon \mathit{Bfun} \to \wp(\mathit{RSubst})$.)** *The function* $\mathrm{hval} \colon \mathit{RSubst} \to \mathit{Bval}$ *is defined, for each $\sigma \in \mathit{RSubst}$ and each $x \in \mathit{VI}$, by*

$$\mathrm{hval}(\sigma)(x) = 1 \quad \overset{\text{def}}{\iff} \quad x \in \mathrm{hvars}(\sigma).$$

The concretization function $\gamma_F \colon \mathit{Bfun} \to \wp(\mathit{RSubst})$ is defined, for $\phi \in \mathit{Bfun}$, by

$$\gamma_F(\phi) \overset{\text{def}}{=} \bigl\{\, \sigma \in \mathit{RSubst} \bigm| \forall \tau \in {\downarrow}\sigma : \phi\bigl(\mathrm{hval}(\tau)\bigr) = 1 \,\bigr\}.$$

The following theorem shows how most of the operators needed to compute the concrete semantics of a logic program can be correctly approximated on the abstract domain *Bfun*.

Theorem 12. *Let $\Sigma, \Sigma_1, \Sigma_2 \in \wp(\mathit{RSubst})$ and $\phi, \phi_1, \phi_2 \in \mathit{Bfun}$ be such that $\gamma_F(\phi) \supseteq \Sigma$, $\gamma_F(\phi_1) \supseteq \Sigma_1$, and $\gamma_F(\phi_2) \supseteq \Sigma_2$. Let also $(x \mapsto t) \in \mathit{Bind}$, where $\{x\} \cup \mathrm{vars}(t) \subseteq \mathit{VI}$. Then the following hold:*

$$\gamma_F\Bigl(x \leftrightarrow \bigwedge \mathrm{vars}(t)\Bigr) \supseteq \bigl\{\{x \mapsto t\}\bigr\}; \tag{12a}$$

$$\gamma_F(\neg x) \supseteq \bigl\{\{x \mapsto t\}\bigr\}, \text{ if } x \in \mathrm{vars}(t); \tag{12b}$$

$$\gamma_F(x) \supseteq \bigl\{\, \sigma \in \mathit{RSubst} \bigm| x \in \mathrm{gvars}(\sigma) \cap \mathrm{hvars}(\sigma) \,\bigr\}; \tag{12c}$$

$$\gamma_F(\phi_1 \wedge \phi_2) \supseteq \bigl\{\, \mathrm{mgs}(\sigma_1 \cup \sigma_2) \bigm| \sigma_1 \in \Sigma_1, \sigma_2 \in \Sigma_2 \,\bigr\}; \tag{12d}$$

$$\gamma_F(\phi_1 \vee \phi_2) \supseteq \Sigma_1 \cup \Sigma_2; \tag{12e}$$

$$\gamma_F(\exists x \mathrel{.} \phi) \supseteq \exists\!\!\exists\, x \mathrel{.} \Sigma. \tag{12f}$$

Cases (12a), (12b), and (12d) of Theorem 12 ensure that the following definition of amgu_F provides a correct approximation on *Bfun* of the concrete unification of rational trees.

Definition 13. *The function* $\mathrm{amgu}_F \colon \mathit{Bfun} \times \mathit{Bind} \to \mathit{Bfun}$ *captures the effects of a binding on a finite-tree dependency formula. Let $\phi \in \mathit{Bfun}$ and $(x \mapsto t) \in \mathit{Bind}$ be such that $\{x\} \cup \mathrm{vars}(t) \subseteq \mathit{VI}$. Then*

$$\mathrm{amgu}_F(\phi, x \mapsto t) \overset{\text{def}}{=} \begin{cases} \phi \wedge \bigl(x \leftrightarrow \bigwedge \mathrm{vars}(t)\bigr), & \text{if } x \notin \mathrm{vars}(t); \\ \phi \wedge \neg x, & \text{otherwise.} \end{cases}$$

Other semantic operators, such as the consistent renaming of variables, are very simple and, as usual, their approximation does not pose any problem.

The next result shows how finite-tree dependencies may improve the finiteness information encoded in the h component of the domain $H \times P$.

Theorem 14. *Let $h \in H$ and $\phi \in \mathit{Bfun}$. Let also $h' \overset{\text{def}}{=} \mathrm{true}\Bigl(\phi \wedge \bigwedge h\Bigr)$. Then*

$$\gamma_H(h) \cap \gamma_F(\phi) = \gamma_H(h') \cap \gamma_F(\phi).$$

Example 15. Consider the following program, where it is assumed that the only "external" query is '`?- r(X, Y)`':

```
p(X, Y) :- X = f(Y, _).
q(X, Y) :- X = f(_, Y).
r(X, Y) :- p(X, Y), q(X, Y), acyclic_term(X).
```

Then the predicate `p/2` in the clause defining `r/2` will called with `X` and `Y` both unbound. Computing on the abstract domain $H \times P$ gives us the finiteness description $h_p = \{x, y\}$, expressing the fact that both `X` and `Y` are bound to finite terms. Computing on the finite-tree dependencies domain *Bfun*, gives us the Boolean formula $\phi_p = x \rightarrow y$ (`Y` is finite if `X` is so).

Considering now the call to the predicate `q/2`, we note that, since variable `X` is already bound to a non-variable term sharing with `Y`, all the finiteness information encoded by H will be lost (i.e., $h_q = \varnothing$). So, both `X` and `Y` are detected as possibly cyclic. However, the finite-tree dependency information is preserved, because $\phi_q = (x \rightarrow y) \wedge (x \rightarrow y) = x \rightarrow y$.

Finally, consider the effect of the abstract evaluation of `acyclic_term(X)`. On the $H \times P$ domain we can only infer that variable `X` cannot be bound to an infinite term, while `Y` will be still considered as possibly cyclic, so that $h_r = \{x\}$. On the domain *Bfun* we can just confirm that the finite-tree dependency computed so far still holds, so that $\phi_r = x \rightarrow y$ (no stronger finite-tree dependency can be inferred, since the finiteness of `X` is only contingent). Thus, by applying the result of Theorem 14, we can recover the finiteness of `Y`:

$$h'_r = \mathrm{true}\Bigl(\phi_r \wedge \bigwedge h_r\Bigr) = \mathrm{true}\bigl((x \rightarrow y) \wedge x\bigr) = \{x, y\}.$$

Information encoded in $H \times P$ and *Bfun* is not completely orthogonal and the following result provides a kind of consistency check.

Theorem 16. *Let $h \in H$ and $\phi \in$ Bfun. Then*

$$\gamma_H(h) \cap \gamma_F(\phi) \neq \varnothing \quad \Longrightarrow \quad h \cap \mathrm{false}(\phi) = \varnothing.$$

Note however that, provided the abstract operators are correct, the computed descriptions will always be mutually consistent, unless $\phi = \bot$.

5 Groundness Dependencies

Since information about the groundness of variables is crucial for many applications, it is natural to consider a static analysis domain including both a finite-tree and a groundness component. In fact, any reasonably precise implementation of the parameter component P of the abstract domain specified in [4] will include some kind of groundness information. We highlight similarities, differences and connections relating the domain *Bfun* for finite-tree dependencies to the abstract domain *Pos* for groundness dependencies. Note that these results also hold when considering a combination of *Bfun* with the groundness domain *Def* [1].

Definition 17. **($\gamma_G \colon Pos \to \wp(RSubst)$).** *The function* gval: $RSubst \to Bval$ *is defined as follows, for each $\sigma \in RSubst$ and each $x \in VI$:*

$$\mathrm{gval}(\sigma)(x) = 1 \quad \overset{\mathrm{def}}{\iff} \quad x \in \mathrm{gvars}(\sigma).$$

The concretization function $\gamma_G \colon Pos \to \wp(RSubst)$ is defined, for each $\psi \in Pos$,

$$\gamma_G(\psi) \overset{\mathrm{def}}{=} \bigl\{\, \sigma \in RSubst \bigm| \forall \tau \in {\downarrow}\,\sigma : \psi\bigl(\mathrm{gval}(\tau)\bigr) = 1 \,\bigr\}.$$

Definition 18. *The function* $\mathrm{amgu}_G \colon Pos \times Bind \to Pos$ *captures the effects of a binding on a groundness dependency formula. Let $\psi \in Pos$ and $(x \mapsto t) \in Bind$ be such that $\{x\} \cup \mathrm{vars}(t) \subseteq VI$. Then*

$$\mathrm{amgu}_G(\psi, x \mapsto t) \overset{\mathrm{def}}{=} \psi \land \Bigl(x \leftrightarrow \bigwedge\bigl(\mathrm{vars}(t) \setminus \{x\}\bigr)\Bigr).$$

Note that this is a simple variant of the standard abstract unification operator for groundness analysis over finite-tree domains: the only difference concerns the case of cyclic bindings [2].

The next result shows how, by exploiting the finiteness component H, the finite-tree dependencies (*Bfun*) component and the groundness dependencies (*Pos*) component can improve each other.

Theorem 19. *Let $h \in H$, $\phi \in Bfun$ and $\psi \in Pos$. Let also $\phi' \in Bfun$ and $\psi' \in Pos$ be defined as $\phi' = \exists\, VI \setminus h \,.\, \psi$ and $\psi' = \exists\, VI \setminus h \,.\, \mathrm{pos}(\phi)$. Then*

$$\gamma_H(h) \cap \gamma_F(\phi) \cap \gamma_G(\psi) = \gamma_H(h) \cap \gamma_F(\phi) \cap \gamma_G(\psi \land \psi'); \tag{19a}$$

$$\gamma_H(h) \cap \gamma_F(\phi) \cap \gamma_G(\psi) = \gamma_H(h) \cap \gamma_F(\phi \land \phi') \cap \gamma_G(\psi). \tag{19b}$$

Moreover, even without any knowledge of the H component, combining Theorem 14 and Eq. (19a), the groundness dependencies component can be improved.

Corollary 20. *Let $\phi \in Bfun$ and $\psi \in Pos$. Then*

$$\gamma_F(\phi) \cap \gamma_G(\psi) = \gamma_F(\phi) \cap \gamma_G\bigl(\psi \land \mathrm{true}(\phi)\bigr).$$

The following example shows that, when computing on rational trees, finite-tree dependencies may provide groundness information that is not captured by the usual approaches.

Example 21. Consider the program:

```
p(a, Y).
p(X, a).
q(X, Y) :- p(X, Y), X = f(X, Z).
```

The abstract semantics of `p/2`, for both finite-tree and groundness dependencies, is $\phi_p = \psi_p = x \lor y$. The finite-tree dependency for `q/2` is $\phi_q = (x \lor y) \land \neg x = \neg x \land y$. Using Definition 18, the groundness dependency for `q/2` is

$$\psi_q = \exists z \,.\, \bigl((x \lor y) \land (x \leftrightarrow z)\bigr) = x \lor y.$$

This can be improved, using Corollary 20, to

$$\psi'_q = \psi_q \wedge \bigwedge \mathrm{true}(\phi_q) = y.$$

Since better groundness information, besides being useful in itself, may also improve the precision of many other analyses such as sharing [7, 8], the reduction steps given by Theorem 19 and Corollary 20 can trigger improvements to the precision of other components. Theorem 19 can also be exploited to recover precision after the application of a widening operator on either the groundness dependencies or the finite-tree dependencies component.

6 Experimental Results

The work described here and in [4] has been experimentally evaluated in the framework provided by the CHINA analyzer [2]. We implemented and compared the three domains Pattern(P), Pattern($H \times P$) and Pattern($\mathit{Bfun} \times H \times P$),[1] where the parameter component P has been instantiated to the domain $\mathit{Pos} \times \mathit{SFL}$ [7] for tracking groundness, freeness, linearity and (non-redundant) set-sharing information. The Pattern($\cdot$) operator [5] further upgrades the precision of its argument by adding explicit structural information.

Concerning the *Bfun* component, the implementation was straightforward, since all the techniques described in [6] (and almost all the code, including the widenings) has been reused unchanged, obtaining comparable efficiency. As a consequence, most of the implementation effort was in the coding of the abstract operators on the H component and of the reduction processes between the different components. A key choice, in this sense, is 'when' the reduction steps given in Theorems 14 and 19 should be applied. When striving for maximum precision, a trivial strategy is to immediately perform reductions after any application of any abstract operator. For instance, this is how predicates like `acyclic_term/1` should be handled: after adding the variables of the argument to the H component, the reduction process is applied to propagate the new information to all domain components. However, such an approach turns out to be unnecessarily inefficient. In fact, the next result shows that Theorems 14 and 19 cannot lead to a precision improvement if applied just after the abstract evaluation of the merge-over-all-paths or the existential quantification operations (provided the initial descriptions are already reduced).

Theorem 22. *Let $x \in \mathit{VI}$, $h, h' \in H$ $\phi, \phi' \in \mathit{Bfun}$ and $\psi, \psi' \in \mathit{Pos}$. Let*

$$h_1 \overset{\mathrm{def}}{=} h \cap h', \qquad \phi_1 \overset{\mathrm{def}}{=} \phi \vee \phi', \qquad \psi_1 \overset{\mathrm{def}}{=} \psi \vee \psi',$$
$$h_2 \overset{\mathrm{def}}{=} h \cup \{x\}, \qquad \phi_2 \overset{\mathrm{def}}{=} \exists x \,.\, \phi, \qquad \psi_2 \overset{\mathrm{def}}{=} \exists x \,.\, \psi.$$

Let also

$$h \supseteq \mathrm{true}\bigl(\phi \wedge \bigwedge h\bigr), \qquad \phi \models (\exists\, \mathit{VI} \setminus h \,.\, \psi), \qquad \psi \models \bigl(\exists\, \mathit{VI} \setminus h \,.\, \mathrm{pos}(\phi)\bigr),$$
$$h' \supseteq \mathrm{true}\bigl(\phi' \wedge \bigwedge h'\bigr), \qquad \phi' \models (\exists\, \mathit{VI} \setminus h' \,.\, \psi'), \qquad \psi' \models \bigl(\exists\, \mathit{VI} \setminus h' \,.\, \mathrm{pos}(\phi')\bigr).$$

[1] For ease of notation, the domain names are shortened to P, H and Bfun, respectively.

Prec. class	P	H	Bfun
$p = 100$	2	84	86
$80 \le p < 100$	1	31	36
$60 \le p < 80$	7	26	23
$40 \le p < 60$	6	41	40
$20 \le p < 40$	47	47	46
$0 \le p < 20$	185	19	17

Prec. improvement	P $\rightarrow$ H	H $\rightarrow$ Bfun
$i > 20$	185	4
$10 < i \le 20$	31	3
$5 < i \le 10$	11	6
$2 < i \le 5$	4	10
$0 < i \le 2$	2	24
no improvement	15	201

Table 1. The precision on finite variables when using P, H and Bfun.

Then, for $i = 1, 2$,

$$h_i \supseteq \mathrm{true}\bigl(\phi_i \wedge \bigwedge h_i\bigr), \qquad \phi_i \models (\exists VI \setminus h_i \mathrel{.} \psi_i), \qquad \psi_i \models \bigl(\exists VI \setminus h_i \mathrel{.} \mathrm{pos}(\phi_i)\bigr).$$

We conjecture that Theorem 22 can be strengthened: the reduction process affecting the *Bfun* component, corresponding to Eq. (19b) of Theorem 19, seems to be useless also after the application of an abstract unification. In any case, this reduction process can be usefully exploited to recover precision after the application of a widening operator on the *Bfun* component.

A goal-dependent analysis was run for all the programs in our benchmark suite and the results (with respect to the precision) are summarized in Table 1. Here, the precision is measured as the percentage of the total number of variables that the analyser can show to be Herbrand. Two alternative views are provided.

In the first view, each column is labeled by an analysis domain and each row is labeled by a precision interval. For instance, the value '31' at the intersection of column 'H' and row '$80 \le p < 100$' is to be read as "*for 31 benchmarks, the percentage p of the total number of variables that the analyser can show to be Herbrand using the domain* H *is between 80% and 100%.*"

The second view provides a better picture of the precision *improvements* obtained when moving from P to H (in the column 'P $\rightarrow$ H') and from H to Bfun (in the column 'H $\rightarrow$ Bfun'). For instance, the value '10' at the intersection of column 'H $\rightarrow$ Bfun' and row '$2 < i \le 5$' is to be read as "*when moving from* H *to* Bfun*, for 10 benchmarks the improvement i in the percentage of the total number of variables shown to be Herbrand was between 2% and 5%.*"

It can be seen from Table 1 that, even though the H domain is remarkably precise, the inclusion of the *Bfun* component allows for a further, and sometimes significant, precision improvement for a number of benchmarks. It is worth noting that the current implementation of CHINA does not yet fully exploit the finite-tree dependencies arising when evaluating many of the built-in predicates, therefore incurring an avoidable precision loss. We are working on this issue and we expect that the specialised implementation of the abstract evaluation of some built-ins will result in more and better precision improvements. The experimentation has also shown that, in practice, the Bfun domain does not improve the groundness information.

7 Conclusion

Several modern logic-based languages offer a computation domain based on rational trees. On the one hand, the use of such trees is encouraged by the possibility of using efficient and correct unification algorithms and by an increase in expressivity. On the other hand, these gains are countered by the extra problems rational trees bring with themselves As a consequence, those applications that exploit rational trees tend to do so in a very controlled way, that is, most program variables can only be bound to finite terms. By detecting the program variables that may be bound to infinite terms with a good degree of accuracy, we can significantly reduce the disadvantages of using rational trees.

In [4], an initial solution to the problem was proposed where the composite abstract domain $H \times P$ allows to track the creation and propagation of infinite terms. Even though this information is crucial to any finite-tree analysis, propagating the guarantees of finiteness that come from several built-ins (including those that are explicitly provided to test term-finiteness) is also important. Therefore, in this paper we have introduced a domain of Boolean functions *Bfun* for finite-tree dependencies which, when coupled to the domain $H \times P$, can enhance its expressive power. Since *Bfun* has many similarities with the domain *Pos* used for groundness analysis, we have investigated how these two domains relate to each other and, in particular, the synergy arising from their combination in the "global" domain of analysis.

References

1. T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two classes of Boolean functions for dependency analysis. *Science of Computer Programming*, 31(1):3–45, 1998.
2. R. Bagnara. *Data-Flow Analysis for Constraint Logic-Based Languages.* PhD thesis, Dipartimento di Informatica, Università di Pisa, Pisa, Italy, 1997. Printed as Report TD-1/97.
3. R. Bagnara, R. Gori, P. M. Hill, and E. Zaffanella. Finite-tree analysis for constraint logic-based languages. Quaderno 251, Dipartimento di Matematica, Università di Parma, 2001. Available at http://www.cs.unipr.it/ bagnara/.
4. R. Bagnara, R. Gori, P. M. Hill, and E. Zaffanella. Finite-tree analysis for constraint logic-based languages. In P. Cousot, editor, *Static Analysis: 8th International Symposium, SAS 2001*, volume 2126 of *Lecture Notes in Computer Science*, pages 165–184, Paris, France, 2001. Springer-Verlag, Berlin.
5. R. Bagnara, P. M. Hill, and E. Zaffanella. Efficient structural information analysis for real CLP languages. In M. Parigot and A. Voronkov, editors, *Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning (LPAR 2000)*, volume 1955 of *Lecture Notes in Computer Science*, pages 189–206, Réunion Island, France, 2000. Springer-Verlag, Berlin.
6. R. Bagnara and P. Schachte. Factorizing equivalent variable pairs in ROBDD-based implementations of *Pos.* In A. M. Haeberer, editor, *Proceedings of the "Seventh International Conference on Algebraic Methodology and Software Technology (AMAST'98)"*, volume 1548 of *Lecture Notes in Computer Science*, pages 471–485, Amazonia, Brazil, 1999. Springer-Verlag, Berlin.

7. R. Bagnara, E. Zaffanella, and P. M. Hill. Enhanced sharing analysis techniques: A comprehensive evaluation. In M. Gabbrielli and F. Pfenning, editors, *Proceedings of the 2nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 103–114, Montreal, Canada, 2000. Association for Computing Machinery.
8. M. Codish, H. Søndergaard, and P. J. Stuckey. Sharing and groundness dependencies in logic programs. *ACM Transactions on Programming Languages and Systems*, 21(5):948–976, 1999.
9. A. Colmerauer. Prolog and infinite trees. In K. L. Clark and S. Å. Tärnlund, editors, *Logic Programming, APIC Studies in Data Processing*, volume 16, pages 231–251. Academic Press, New York, 1982.
10. A. Colmerauer. Equations and inequations on finite and infinite trees. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'84)*, pages 85–99, Tokyo, Japan, 1984. ICOT.
11. A. Cortesi, G. Filé, and W. Winsborough. *Prop* revisited: Propositional formula as abstract domain for groundness analysis. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 322–327, Amsterdam, The Netherlands, 1991. IEEE Computer Society Press.
12. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combinations of abstract domains for logic programming: Open product and generic pattern construction. *Science of Computer Programming*, 38(1–3), 2000.
13. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
14. P. Cousot and R. Cousot. Abstract interpretation and applications to logic programs. *Journal of Logic Programming*, 13(2&3):103–179, 1992.
15. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.
16. P. M. Hill, R. Bagnara, and E. Zaffanella. Soundness, idempotence and commutativity of set-sharing. *Theory and Practice of Logic Programming*, 2001. To appear. Available at http://arXiv.org/abs/cs.PL/0102030.
17. B. Intrigila and M. Venturini Zilli. A remark on infinite matching vs infinite unification. *Journal of Symbolic Computation*, 21(3):2289–2292, 1996.
18. A. King. Pair-sharing over rational trees. *Journal of Logic Programming*, 46(1–2):139–155, 2000.
19. K. Marriott and H. Søndergaard. Notes for a tutorial on abstract interpretation of logic programs. North American Conference on Logic Programming, Cleveland, Ohio, USA, 1989.
20. K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems*, 2(1–4):181–196, 1993.
21. E. Schröder. *Der Operationskreis des Logikkalkuls.* B. G. Teubner, Leibzig, 1877.
22. H. Søndergaard. An application of abstract interpretation of logic programs: Occur check reduction. In B. Robinet and R. Wilhelm, editors, *Proceedings of the 1986 European Symposium on Programming*, volume 213 of *Lecture Notes in Computer Science*, pages 327–338. Springer-Verlag, Berlin, 1986.