Finite-Tree Analysis for Constraint Logic-Based Languages[⋆]

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

[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, U. K.
hill@comp.leeds.ac.uk

Abstract. Logic languages based on the theory of rational, possibly infinite, trees have much appeal in that rational trees allow for faster unification (due to the omission of the occurs-check) and increased expressivity. Note that cyclic terms can provide a very efficient representation of grammars and other useful objects. 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. Moreover, some widely-used program analysis and manipulation techniques are only correct for those parts of programs working over finite trees. It is thus important to obtain, automatically, a knowledge of those program variables (the *finite variables*) that, at the program points of interest, will always be bound to finite terms. For these reasons, we propose here a new data-flow analysis that captures such information. We present a parametric domain where a simple component for recording finite variables is coupled with a generic domain (the parameter of the construction) providing sharing information. The sharing domain is abstractly specified so as to guarantee the correctness of the combined domain and the generality of the approach.

1 Introduction

The intended computation domain of most logic-based languages[1] includes the algebra (or structure) of *finite trees.* Other (constraint) logic-based languages, such as Prolog II and its successors [10, 12], SICStus Prolog [36], and Oz [34], refer to a computation domain of *rational trees.* A rational tree is a possibly infinite tree with a finite number of distinct subtrees and, as is the case for finite trees,

[⋆] This work has been partly supported by MURST project “Certificazione automatica di programmi mediante interpretazione astratta.” Some of this work was done during visits of the fourth author to Leeds, funded by EPSRC under grant M05645.

[1] That is, ordinary logic languages, (concurrent) constraint logic languages, functional logic languages and variations of the above.

where each node has a finite number of immediate descendants. These properties will ensure that rational trees, even though infinite in the sense that they admit paths of infinite length, can be finitely represented. One possible representation makes use of connected, rooted, directed and possibly cyclic graphs where nodes are labeled with variable and function symbols as is the case of finite trees.

Applications of rational trees in logic programming include graphics [18], parser generation and grammar manipulation [10, 21], and computing with finite-state automata [10]. Other applications are described in [20] and [23]. Going from Prolog to CLP, [31] combines constraints on rational trees and record structures, while the logic-based language *Oz* allows constraints over rational and feature trees [34]. The expressive power of rational trees is put to use, for instance, in several areas of natural language processing. Rational trees are used in implementations of the HPSG formalism (Head-driven Phrase Structure Grammar) [32], in the ALE system (Attribute Logic Engine) [8], and in the ProFIT system (Prolog with Features, Inheritance and Templates) [19].

While rational trees allow for increased expressivity, they also come equipped with a surprising number of problems. As we will see, some of these problems are so serious that rational trees must be used in a very controlled way, disallowing them in any context where they are “dangerous”. This, in turn, causes a secondary problem: in order to disallow rational trees in selected contexts one must first detect them, an operation that may be expensive.

The first thing to be aware of is that almost any semantics-based program manipulation technique developed in the field of logic programming —whether it be an analysis, a transformation, or an optimization— assumes a computation domain of *finite trees*. Some of these techniques might work with the rational trees but their correctness has only been proved in the case of finite trees. Others are clearly inapplicable. Let us consider a very simple Prolog program:

```
list([]).
list([_|T]) :- list(T).
```

Most automatic and semi-automatic tools for proving program termination and for complexity analysis agree on the fact that `list/1` will terminate when invoked with a ground argument. Consider now the query

```
?- X = [a|X], list(X).
```

and note that, after the execution of the first rational unification, the variable `X` will be bound to a rational term containing no variables, i.e., the predicate `list/1` will be invoked with `X` ground. However, if such a query is given to, say, SICStus Prolog, then the only way to get the prompt back is by pressing `^C`. The problem stems from the fact that the analysis techniques employed by these tools are only sound for finite trees: as soon as they are applied to a system where the creation of cyclic terms is possible, their results are inapplicable. The situation can be improved by combining these termination and/or complexity analyses by a finiteness analysis providing the precondition for the applicability of the other techniques.

The implementation of built-in predicates is another problematic issue. Indeed, it is widely acknowledged that, for the implementation of a system that provides real support for the rational trees, the biggest effort concerns proper handling of built-ins. Of course, the meaning of 'proper' depends on the actual built-in. Built-ins such as `copy_term/2` and `==/2` maintain a clear semantics when passing from finite to rational trees. For others, like `sort/2`, the extension can be questionable:[2] both raising an exception and answering `Y = [a]` can be argued to be "the right reaction" to the query

```
?- X = [a|X], sort(X, Y).
```

Other built-ins do not tolerate infinite trees in some argument positions. A good implementation should check for finiteness of the corresponding arguments and make sure "the right thing" —failing or raising an appropriate exception— always happens. However, such behavior appears to be uncommon. A small experiment we conducted on six Prolog implementations with queries like

```
?- X = 1+X, Y is X.
?- X = [97|X], name(Y, X).
?- X = [X|X], Y =.. [f|X].
```

resulted in infinite loops, memory exhaustion and/or system thrashing, segmentation faults or other fatal errors. One of the implementations tested, SICStus Prolog, is a professional one and implements run-time checks to avoid most cases where built-ins can have catastrophic effects.[3] The remaining systems are a bit more than research prototypes, but will clearly have to do the same if they evolve to the stage of production tools. Again, a data-flow analysis aimed at the detection of those variables that are definitely bound to finite terms would allow to avoid a (possibly significant) fraction of the useless run-time checks. Note that what has been said for built-in predicates applies to libraries as well. Even though it may be argued that it is enough for programmers to know that they should not use a particular library predicate with infinite terms, it is clear that the use of a "safe" library, including automatic checks which ensure that such predicates are never called with an illegal argument, will result in more robust systems. With the appropriate data-flow analyses, safe libraries do not have to be inefficient libraries.

Another serious problem is the following: the ISO Prolog standard term ordering cannot be extended to rational trees [M. Carlsson, Personal communication, October 2000]. Consider the rational trees defined by `A = f(B, a)` and `B = f(A, b)`. Clearly, `A == B` does not hold. Since the standard term ordering is total, we must have either `A @< B` or `B @< A`. Assume `A @< B`. Then `f(A, b) @< f(B, a)`, since the ordering of terms having the same principal functor is inherited by the ordering of subterms considered in a left-to-right fashion. Thus `B @< A` must hold, which is a contradiction. A dual contradiction

[2] Even though `sort/2` is not required to be a built-in by the standard, it is offered as such by several implementations.

[3] SICStus 3.8.5 still loops on `?- X = [97|X], name(Y, X).`

is obtained by assuming `B @< A`. As a consequence, applying one of the Prolog term-ordering predicates to one or two infinite terms may cause inconsistent results, giving rise to bugs that are exceptionally difficult to diagnose. For this reason, any system that extends ISO Prolog with rational trees ought to detect such situations and make sure they are not ignored (e.g., by throwing an exception or aborting execution with a meaningful message). However, predicates such as the term-ordering ones are likely to be called a significant number of times, since they are often used to maintain structures implementing ordered collections of terms. This is another instance of the efficiency issue mentioned above.

In this paper, we present a parametric abstract domain for finite-tree analysis, denoted by $H \times P$. This domain combines a simple component H (the *finiteness* component), recording the set of definitely finite variables, with a generic domain P (the parameter of the construction), providing sharing information. The term "sharing information" is to be understood in its broader meaning, which includes variable aliasing, groundness, linearity, freeness and any other kind of information that can improve the precision on these components, such as explicit structural information. Several domain combinations and abstract operators, characterized by different precision/complexity trade-offs, have been proposed to capture these properties (see [5] for an account of some of them). By giving a generic specification for this parameter component, in the style of the *open product* construct proposed in [14], 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 paper is structured as follows. The required notations and preliminary concepts are given in Section 2. The finite-tree domain is then introduced in Section 3: Section 3.1 provides the specification of the parameter domain P; Section 3.2 defines the abstraction function for the finiteness component H; Section 3.3 defines the abstract unification operator for $H \times P$. A description of some ongoing work on the subject is given in Section 4 where a possible instance of the parameter P is also specified. We conclude in Section 5.

A longer version of this paper with proofs of the results presented here is available as a technical report [1].

2 Preliminaries

2.1 Infinite Terms and Substitutions

For a set S, $\wp(S)$ is the powerset of S, whereas $\wp_{\mathrm{f}}(S)$ is the set of all the *finite* subsets of S. Let *Sig* denote a possibly infinite set of function symbols, ranked over the set of natural numbers. It is assumed that *Sig* contains at least one function symbol having rank 0 and one having rank greater than 0. Let *Vars* denote a denumerable set of variables, disjoint from *Sig*. Then *Terms* denotes the free algebra of all (possibly infinite) terms in the signature *Sig* having variables in *Vars*. Thus a term can be seen as an ordered labeled tree, possibly having some infinite paths and possibly containing variables: every inner node is labeled with a function symbol in *Sig* with a rank matching the number of the node's immediate descendants, whereas every leaf is labeled by either a variable in *Vars* or a function symbol in *Sig* having rank 0 (a constant).

If $t \in \mathit{Terms}$ then $\mathrm{vars}(t)$ and $\mathrm{mvars}(t)$ denote the set and the multiset of variables occurring in t, respectively. We will also write $\mathrm{vars}(o)$ to denote the set of variables occurring in an arbitrary syntactic object o. If a occurs more than once in a multiset M we write $a \Subset M$.

Suppose $s, t \in \mathit{Terms}$: s and t are *independent* if $\mathrm{vars}(s) \cap \mathrm{vars}(t) = \varnothing$; if $y \in \mathrm{vars}(t)$ and $\neg\bigl(y \Subset \mathrm{mvars}(t)\bigr)$ we say that variable *y occurs linearly in t*, more briefly written using the predication $\mathrm{occ\_lin}(y, t)$; t is said to be *ground* if $\mathrm{vars}(t) = \varnothing$; t is *free* if $t \in \mathit{Vars}$; t is *linear* if, for all $y \in \mathrm{vars}(t)$, we have $\mathrm{occ\_lin}(y, t)$; finally, t is a *finite term* (or *Herbrand term*) if it contains a finite number of occurrences of function symbols. The sets of all ground, linear and finite terms are denoted by *GTerms*, *LTerms* and *HTerms*, respectively. As we have specified that *Sig* contains function symbols of rank 0 and rank greater than 0, $\mathit{GTerms} \cap \mathit{HTerms} \neq \varnothing$ and $\mathit{GTerms} \setminus \mathit{HTerms} \neq \varnothing$.

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,

$$\mathrm{dom}(\sigma) \stackrel{\mathrm{def}}{=} \bigl\{\, x \in \mathit{Vars} \bigm| \sigma(x) \neq x \,\bigr\},$$

is finite. Given a substitution $\sigma\colon \mathit{Vars} \to \mathit{HTerms}$, we overload the symbol 'σ' so as to denote also the function $\sigma\colon \mathit{HTerms} \to \mathit{HTerms}$ defined as follows, for each term $t \in \mathit{HTerms}$:

$$\sigma(t) \stackrel{\mathrm{def}}{=} \begin{cases} t, & \text{if } t \text{ is a constant symbol;} \\ \sigma(t), & \text{if } t \in \mathit{Vars}; \\ f\bigl(\sigma(t_1), \ldots, \sigma(t_n)\bigr), & \text{if } t = f(t_1, \ldots, t_n). \end{cases}$$

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 denoted by the set of their bindings, thus 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 variables occurring in the bindings of σ.

A substitution is said to be *circular* if, for $n > 1$, it has the form

$$\{x_1 \mapsto x_2, \ldots, x_{n-1} \mapsto x_n, x_n \mapsto x_1\},$$

where x_1, ..., 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*.

If $t \in \mathit{HTerms}$, we write $t\sigma$ to denote $\sigma(t)$ and $t[x/s]$ to denote $t\{x \mapsto s\}$.

The composition of substitutions is defined in the usual way. Thus $\tau \circ \sigma$ is the substitution such that, for all terms $t \in \mathit{HTerms}$,

$$(\tau \circ \sigma)(t) = \tau\bigl(\sigma(t)\bigr)$$

and has the formulation

$$\tau \circ \sigma = \bigl\{\, x \mapsto x\sigma\tau \bigm| x \in \operatorname{dom}(\sigma), x \neq x\sigma\tau \,\bigr\} \cup \bigl\{\, x \mapsto x\tau \bigm| x \in \operatorname{dom}(\tau) \setminus \operatorname{dom}(\sigma) \,\bigr\}.$$

As usual, σ^0 denotes the identity function (i.e., the empty substitution) and, when $i > 0$, σ^i denotes the substitution $(\sigma \circ \sigma^{i-1})$.

For each $\sigma \in \mathit{RSubst}$, $s \in \mathit{HTerms}$, the sequence of finite terms

$$\sigma^0(s), \sigma^1(s), \sigma^2(s), \ldots$$

converges to a (possibly infinite) term, denoted $\sigma^\infty(s)$ [25, 29]. Therefore, the function $\mathrm{rt}\colon \mathit{HTerms} \times \mathit{RSubst} \to \mathit{Terms}$ such that

$$\mathrm{rt}(s, \sigma) \stackrel{\mathrm{def}}{=} \sigma^\infty(s)$$

is well defined. Note that, in general, this function is not a substitution: while having a finite domain, its "bindings" $x \mapsto t$ can map a domain variable x into a term $t \in \mathit{Terms} \setminus \mathit{HTerms}$.

2.2 Equations

An *equation* is of 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).

Languages such as Prolog II, SICStus and Oz are based on $\mathcal{RT}$, the theory of rational trees [10, 11]. 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. Informally speaking these axioms state that, after assigning a ground rational tree to each non-domain variable, the substitution uniquely defines a ground rational tree for each of its domain variables. Thus, any set of equations in rational solved form is, by definition, satisfiable in $\mathcal{RT}$. Note that being in rational solved form is a very weak property. Indeed, unification algorithms returning a set of equations in rational solved form are allowed to be much more "lazy" than one would usually expect. We refer the interested reader to [27, 28, 30] for details on the subject.

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 $\mathcal{RT} \vdash \forall(\sigma \to e)$, i.e., if every model of the theory $\mathcal{RT}$ is also a model of the first order formula $\forall(\sigma \to e)$. If in addition $\mathrm{vars}(\sigma) \subseteq \mathrm{vars}(e)$, then σ is said to be a *relevant* solution for e. Finally, σ is a *most general solution for e in $\mathcal{RT}$* if $\mathcal{RT} \vdash \forall(\sigma \leftrightarrow e)$. In this paper, the set of all the relevant most general solution for e in $\mathcal{RT}$ will be denoted by $\mathrm{mgs}(e)$.

2.3 The Concrete Domain

Throughout the paper, we assume a knowledge of the basic concepts of abstract interpretation theory [15, 16].

For the purpose of this paper, we assume a concrete domain constituted by 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 1. (The concrete domain.) *Let* $\mathcal{D}^\flat \stackrel{\mathrm{def}}{=} \wp(\mathit{RSubst}) \times \wp_{\mathrm{f}}(\mathit{Vars})$. *If* $(\Sigma, V) \in \mathcal{D}^\flat$*, then* (Σ, V) *represents the (possibly infinite) set of first-order formulas* $\bigl\{\, \exists \Delta \mathrel{.} \sigma \bigm| \sigma \in \Sigma, \Delta = \mathrm{vars}(\sigma) \setminus V \,\bigr\}$ *where* σ *is interpreted as the logical conjunction of the equations corresponding to its bindings.*

Concrete domains for constraint languages would be similar. If the analyzed language allows the use of constraints on various domains to restrict the values of the variable leaves of rational trees, the corresponding concrete domain would have one or more extra components to account for the constraints (see [2] for an example).

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 in the first case the variable name 'y' matters, but it does not in the second case. 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 the existence of a set $\mathit{VI} \in \wp_{\mathrm{f}}(\mathit{Vars})$ that contains, at each stage of the analysis, the current variables of interest.[4] As a consequence, when the context makes it clear that $\Sigma \in \wp(\mathit{RSubst})$, we will write $\Sigma \in \mathcal{D}^\flat$ as a shorthand for $(\Sigma, \mathit{VI}) \in \mathcal{D}^\flat$.

3 An Abstract Domain for Finiteness Analysis

Finite-tree analysis applies to logic-based languages computing over a domain of rational trees where cyclic structures are allowed. In contrast, analyses aimed at occurs-check reduction [17, 35] apply to programs that are meant to compute on a domain of finite trees only, but have to be executed over systems that are either designed for rational trees or intended just for the finite trees but omit the occurs-check for efficiency reasons. Despite their different objectives, finite-tree and occurs-check analyses have much in common: in both cases, it is important to detect all program points where cyclic structures can be generated.

[4] This parallels what happens in the efficient implementation of data-flow analyzers. In fact, almost all the abstract domains currently in use do not need to represent explicitly the set of variables of interest. In contrast, this set is maintained externally and in a unique copy, typically by the fixpoint computation engine.

Note however that, when performing occurs-check reduction, one can take advantage of the following invariant: all data structures generated so far are finite. This property is maintained by transforming the program so as to force finiteness whenever it is possible that a cyclic structure could have been built.[5] In contrast, a finite-tree analysis has to deal with the more general case when some of the data structures computed so far may be cyclic. It is therefore natural to consider an abstract domain made up of two components. The first one simply represents the set of variables that are guaranteed not to be bound to infinite terms. We will denote this *finiteness component* by H (from *Herbrand*).

Definition 2. (The finiteness component.) *The* finiteness component *is the set* $H \stackrel{\mathrm{def}}{=} \wp(\mathit{VI})$ *partially ordered by reverse subset inclusion.*

The second component of the finite-tree domain should maintain any kind of information that may be useful for computing finiteness information.

It is well-known that sharing information as a whole, therefore including possible variable aliasing, definite linearity, and definite freeness, has a crucial role in occurs-check reduction so that, as observed before, it can be exploited for finite-tree analysis too. Thus, a first choice for the second component of the finite-tree domain would be to consider one of the standard combinations of sharing, freeness and linearity as defined, e.g., in [5, 6, 22]. However, this would tie our specification to a particular sharing analysis domain, whereas the overall approach seems to be inherently more general. For this reason, we will define a finite-tree analysis based on the abstract domain schema $H \times P$, where the generic *sharing component* P is a parameter of the abstract domain construction. This approach can be formalized as an application of the *open product* operator [14].

3.1 The parameter component P

Elements of P can encode any kind of information. We only require that substitutions that are equivalent in the theory $\mathcal{RT}$ are identified in P.

Definition 3. (The parameter component.) *The parameter component P is an abstract domain related to the concrete domain $\mathcal{D}^\flat$ by means of the* concretization function *$\gamma_P \colon P \to \wp(\mathit{RSubst})$ such that, for all $p \in P$,*

$$\Bigl(\sigma \in \gamma_P(p) \land \bigl(\mathcal{RT} \vdash \forall(\sigma \leftrightarrow \tau)\bigr)\Bigr) \implies \tau \in \gamma_P(p).$$

The interface between H and P is provided by a set of predicates and functions that satisfy suitable correctness criteria. Note that, for space limitations, we will only specify those abstract operations that are useful to define abstract unification on the combined domain $H \times P$. The other operations needed for a full description of the analysis, such as renamings, upper bound operators and projections, are very simple and, as usual, do not pose any problems.

[5] Such a requirement is typically obtained by replacing the unification with a call to `unify_with_occurs_check/2`. As an alternative, in some systems based on rational trees it is possible to insert, after each problematic unification, a finiteness test for the generated term.

Definition 4. **(Abstract operators on P.)** *Let $s, t \in \mathit{HTerms}$ be finite terms. For each $p \in P$, we define the following predicates:*
s and t are independent in p *if and only if* $\mathrm{ind}_p \colon \mathit{HTerms}^2 \to \mathit{Bool}$ *holds for (s, t), where*

$$\mathrm{ind}_p(s, t) \implies \forall \sigma \in \gamma_P(p) : \mathrm{vars}\bigl(\mathrm{rt}(s, \sigma)\bigr) \cap \mathrm{vars}\bigl(\mathrm{rt}(t, \sigma)\bigr) = \varnothing;$$

s and t share linearly in p *if and only if* $\mathrm{share\_lin}_p \colon \mathit{HTerms}^2 \to \mathit{Bool}$ *holds for (s, t), where*

$$\begin{aligned}\mathrm{share\_lin}_p(s, t) \implies \forall \sigma \in \gamma_P(p) :&\\ &\forall y \in \mathrm{vars}\bigl(\mathrm{rt}(s, \sigma)\bigr) \cap \mathrm{vars}\bigl(\mathrm{rt}(t, \sigma)\bigr) :\\ &\qquad \mathrm{occ\_lin}\bigl(y, \mathrm{rt}(s, \sigma)\bigr) \land \mathrm{occ\_lin}\bigl(y, \mathrm{rt}(t, \sigma)\bigr);\end{aligned}$$

t is ground in p *if and only if* $\mathrm{ground}_p \colon \mathit{HTerms} \to \mathit{Bool}$ *holds for t, where*

$$\mathrm{ground}_p(t) \implies \forall \sigma \in \gamma_P(p) : \mathrm{rt}(t, \sigma) \in \mathit{GTerms};$$

t is ground-or-free in p *if and only if* $\mathrm{gfree}_p \colon \mathit{HTerms} \to \mathit{Bool}$ *holds for t, where*

$$\mathrm{gfree}_p(t) \implies \forall \sigma \in \gamma_P(p) : \mathrm{rt}(t, \sigma) \in \mathit{GTerms} \lor \mathrm{rt}(t, \sigma) \in \mathit{Vars};$$

s and t are or-linear in p *if and only if* $\mathrm{or\_lin}_p \colon \mathit{HTerms}^2 \to \mathit{Bool}$ *holds for (s, t), where*

$$\mathrm{or\_lin}_p(s, t) \implies \forall \sigma \in \gamma_P(p) : \mathrm{rt}(s, \sigma) \in \mathit{LTerms} \lor \mathrm{rt}(t, \sigma) \in \mathit{LTerms};$$

s is linear in p *if and only if* $\mathrm{lin}_p \colon \mathit{HTerms} \to \mathit{Bool}$ *holds for s, where*

$$\mathrm{lin}_p(s) \overset{\mathrm{def}}{\iff} \mathrm{or\_lin}_p(s, s).$$

For each $p \in P$, the following functions compute subsets of the set of variables of interest:
the function $\mathrm{share\_same\_var}_p \colon \mathit{HTerms} \times \mathit{HTerms} \to \wp(\mathit{VI})$ *returns a set of variables that may share with the given terms via the same variable. For each $s, t \in \mathit{HTerms}$,*

$$\mathrm{share\_same\_var}_p(s, t) \supseteq \left\{\, y \in \mathit{VI} \;\middle|\; \begin{array}{l} \exists \sigma \in \gamma_P(p) \,.\\ \quad \exists z \in \mathrm{vars}\bigl(\mathrm{rt}(y, \sigma)\bigr) \,.\\ \qquad z \in \mathrm{vars}\bigl(\mathrm{rt}(s, \sigma)\bigr) \cap \mathrm{vars}\bigl(\mathrm{rt}(t, \sigma)\bigr) \end{array} \right\};$$

the function $\mathrm{share\_with}_p \colon \mathit{HTerms} \to \wp(\mathit{VI})$ *yields a set of variables that may share with the given term. For each $t \in \mathit{HTerms}$,*

$$\mathrm{share\_with}_p(t) \overset{\mathrm{def}}{=} \bigl\{\, y \in \mathit{VI} \bigm| y \in \mathrm{share\_same\_var}_p(y, t) \,\bigr\}.$$

The function $\mathrm{amgu}_P \colon P \times \mathit{Bind} \to P$ *correctly captures the effects of a binding on an element of* P*. For each* $(x \mapsto t) \in \mathit{Bind}$ *and* $p \in P$*, let*

$$p' \stackrel{\mathrm{def}}{=} \mathrm{amgu}_P\bigl(p, x \mapsto t\bigr).$$

For all $\sigma \in \gamma_P(p)$*, if* $\tau \in \mathrm{mgs}\bigl(\sigma \cup \{x = t\}\bigr)$*, then* $\tau \in \gamma_P(p')$*.*

As it will be shown in Section 4.1, some of these generic operators can be directly mapped into the corresponding abstract operators defined for well-known sharing analysis domains. However, the specification given in Definition 4, besides being more general than a particular implementation, also allows for a modular approach when proving correctness results.

3.2 The abstraction function for *H*

When the concrete domain is based on the theory of finite trees, idempotent substitutions provide a finitely computable *strong normal form* for domain elements, meaning that different substitutions describe different sets of finite trees.[6] In contrast, when working on a concrete domain based on the theory of rational trees, substitutions in rational solved form, while being finitely computable, no longer satisfy this property: there can be an infinite set of substitutions in rational solved form all describing the same set of rational trees (i.e., the same element in the “intended” semantics). For instance, the substitutions

$$\sigma_n = \{x \mapsto \overbrace{f(\cdots f(}^{n}x)\cdots)\}$$

for $n = 1, 2, \ldots,$ all map the variable x into the same rational tree (which is usually denoted by f^ω).

Ideally, a strong normal form for the set of rational trees described by a substitution $\sigma \in \mathit{RSubst}$ can be obtained by computing the limit σ^∞. The problem is that we may end up with $\sigma^\infty \notin \mathit{RSubst}$, as σ^∞ can map domain variables to infinite rational terms.

This poses a non-trivial problem when trying to define a “good” abstraction function, since it would be really desirable for this function to map any two equivalent concrete elements to the same abstract element. As shown in [24], the classical abstraction function for set-sharing analysis [13, 26], which was defined for idempotent substitutions only, does not enjoy this property when applied, as it is, to arbitrary substitutions in rational solved form. A possibility is to look for a more general abstraction function that allows to obtain the desired property. For example, in [24] the sharing-group operator sg of [26] is replaced by an occurrence operator, occ, defined by means of a fixpoint computation. We now provide a similar fixpoint construction defining the finiteness operator.

[6] As usual, this is modulo the possible renaming of variables.

Definition 5. (Finiteness functions.) *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*

$$\mathrm{hvars}_0(\sigma) \stackrel{\mathrm{def}}{=} \mathit{Vars} \setminus \mathrm{dom}(\sigma)$$

and, for $n > 0$, by

$$\mathrm{hvars}_n(\sigma) \stackrel{\mathrm{def}}{=} \mathrm{hvars}_{n-1}(\sigma) \cup \bigl\{\, y \in \mathrm{dom}(\sigma) \bigm| \mathrm{vars}(y\sigma) \subseteq \mathrm{hvars}_{n-1}(\sigma) \,\bigr\}.$$

For each $\sigma \in \mathit{RSubst}$ and each $i \geq 0$, we have $\mathrm{hvars}_i(\sigma) \subseteq \mathrm{hvars}_{i+1}(\sigma)$ and also that $\mathit{Vars} \setminus \mathrm{hvars}_i(\sigma) \subseteq \mathrm{dom}(\sigma)$ is a finite set. By these two properties, the following fixpoint computation is well defined and finitely computable.

Definition 6. (Finiteness operator.) *For each $\sigma \in \mathit{RSubst}$, the* finiteness operator $\mathrm{hvars} \colon \mathit{RSubst} \to \wp(\mathit{Vars})$ *is given by* $\mathrm{hvars}(\sigma) \stackrel{\mathrm{def}}{=} \mathrm{hvars}_\ell(\sigma)$ *where* $\ell \stackrel{\mathrm{def}}{=} \ell(\sigma) \in \mathbb{N}$ *is such that* $\mathrm{hvars}_\ell(\sigma) = \mathrm{hvars}_n(\sigma)$ *for all $n \geq \ell$.*

The following proposition shows that the hvars operator precisely captures the intended property.

Proposition 1. *If $\sigma \in \mathit{RSubst}$ and $x \in \mathit{Vars}$ then*

$$x \in \mathrm{hvars}(\sigma) \iff \mathrm{rt}(x, \sigma) \in \mathit{HTerms}.$$

Example 1. Consider $\sigma \in \mathit{RSubst}$, where

$$\sigma = \bigl\{x_1 \mapsto f(x_2), x_2 \mapsto g(x_5), x_3 \mapsto f(x_4), x_4 \mapsto g(x_3)\bigr\}.$$

Then,

$$\begin{aligned}
\mathrm{hvars}_0(\sigma) &= \mathit{Vars} \setminus \{x_1, x_2, x_3, x_4\},\\
\mathrm{hvars}_1(\sigma) &= \mathit{Vars} \setminus \{x_1, x_3, x_4\},\\
\mathrm{hvars}_2(\sigma) &= \mathit{Vars} \setminus \{x_3, x_4\}\\
&= \mathrm{hvars}(\sigma).
\end{aligned}$$

Thus, $x_1 \in \mathrm{hvars}(\sigma)$, although $\mathrm{vars}(x_1\sigma) \subseteq \mathrm{dom}(\sigma)$.

The abstraction function for H can then be defined in the obvious way.

Definition 7. (The abstraction function for H.) *The abstraction function $\alpha_H \colon \mathit{RSubst} \to H$ is defined, for each $\sigma \in \mathit{RSubst}$, by*

$$\alpha_H(\sigma) \stackrel{\mathrm{def}}{=} \mathit{VI} \cap \mathrm{hvars}(\sigma).$$

The concrete domain $\mathcal{D}^\flat$ is related to H by means of the abstraction function *$\alpha_H \colon \mathcal{D}^\flat \to H$ such that, for each $\Sigma \in \wp(\mathit{RSubst})$,*

$$\alpha_H(\Sigma) \stackrel{\mathrm{def}}{=} \bigcap \bigl\{\, \alpha_H(\sigma) \bigm| \sigma \in \Sigma \,\bigr\}.$$

Since the abstraction function α_H is additive, the concretization function is given by its adjoint [15]*:*

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

With these definitions, we have the desired result: equivalent substitutions in rational solved form have the same finiteness abstraction.

Theorem 1. *If $\sigma, \tau \in \mathit{RSubst}$ and $\mathcal{RT} \vdash \forall(\sigma \leftrightarrow \tau)$, then $\alpha_H(\sigma) = \alpha_H(\tau)$.*

3.3 Abstract unification on $H \times P$

The abstract unification for the combined domain $H \times P$ is defined by using the abstract predicates and functions as specified for P as well as a new finiteness predicate for the domain H.

Definition 8. (Abstract unification on $H \times P$.) *A term $t \in \mathit{HTerms}$ is a* finite tree in h *if and only if the predicate* $\mathrm{hterm}_h \colon \mathit{HTerms} \to \mathit{Bool}$ *holds for t, where* $\mathrm{hterm}_h(t) \stackrel{\text{def}}{=} \mathrm{vars}(t) \subseteq h$.

The function $\mathrm{amgu}_H \colon (H \times P) \times \mathit{Bind} \to H$ *captures the effects of a binding on an H element. Let $\langle h, p\rangle \in H \times P$ and $(x \mapsto t) \in \mathit{Bind}$. Then*

$$\mathrm{amgu}_H\bigl(\langle h, p\rangle, x \mapsto t\bigr) \stackrel{\text{def}}{=} h',$$

where

$$h' \stackrel{\text{def}}{=} \begin{cases} h \cup \mathrm{vars}(t), & \text{if } \mathrm{hterm}_h(x) \wedge \mathrm{ground}_p(x); \\ h \cup \{x\}, & \text{if } \mathrm{hterm}_h(t) \wedge \mathrm{ground}_p(t); \\ h, & \text{if } \mathrm{hterm}_h(x) \wedge \mathrm{hterm}_h(t) \\ & \quad \wedge \mathrm{ind}_p(x,t) \wedge \mathrm{or\_lin}_p(x,t); \\ h, & \text{if } \mathrm{hterm}_h(x) \wedge \mathrm{hterm}_h(t) \\ & \quad \wedge \mathrm{gfree}_p(x) \wedge \mathrm{gfree}_p(t); \\ h \setminus \mathrm{share\_same\_var}_p(x,t), & \text{if } \mathrm{hterm}_h(x) \wedge \mathrm{hterm}_h(t) \\ & \quad \wedge \mathrm{share\_lin}_p(x,t) \\ & \quad \wedge \mathrm{or\_lin}_p(x,t); \\ h \setminus \mathrm{share\_with}_p(x), & \text{if } \mathrm{hterm}_h(x) \wedge \mathrm{lin}_p(x); \\ h \setminus \mathrm{share\_with}_p(t), & \text{if } \mathrm{hterm}_h(t) \wedge \mathrm{lin}_p(t); \\ h \setminus \bigl(\mathrm{share\_with}_p(x) \cup \mathrm{share\_with}_p(t)\bigr), & \text{otherwise.} \end{cases}$$

The abstract unification function $\mathrm{amgu} \colon (H \times P) \times \mathit{Bind} \to H \times P$*, for any $\langle h, p\rangle \in H \times P$ and $(x \mapsto t) \in \mathit{Bind}$, is given by*

$$\mathrm{amgu}\bigl(\langle h, p\rangle, x \mapsto t\bigr) \stackrel{\text{def}}{=} \Bigl\langle \mathrm{amgu}_H\bigl(\langle h, p\rangle, x \mapsto t\bigr), \mathrm{amgu}_P(p, x \mapsto t)\Bigr\rangle.$$

In the computation of h' (the new finiteness component resulting from the abstract evaluation of a binding) there are eight cases based on properties holding for the concrete terms described by x and t.

1. In the first case, the concrete term described by x is both finite and ground. Thus, after a successful execution of the binding, any concrete term described by t will be finite. Note that t could have contained variables which may be possibly bound to cyclic terms just before the execution of the binding.
2. The second case is symmetric to the first one. Note that these are the only cases when a "positive" propagation of finiteness information is correct. In contrast, in all the remaining cases, the goal is to limit as much as possible the propagation of "negative" information, i.e., the possible cyclicity of terms.
3. The third case exploits the classical results proved in research work on occurs-check reduction [17, 35]. Accordingly, it is required that both x and t describe finite terms that do not share. The use of the implicitly disjunctive predicate $\mathrm{or\_lin}_p$ allows for the application of this case even when neither x nor t are known to be definitely linear. For instance, as observed in [17], this may happen when the component P embeds the domain *Pos* for groundness analysis.[7]
4. The fourth case exploits the observation that cyclic terms cannot be created when unifying two finite terms that are either ground or free. Ground-or-freeness [5] is a safe, more precise and inexpensive replacement for the classical freeness property when combining sharing analysis domains.
5. The fifth case applies when unifying a linear and finite term with another finite term possibly sharing with it, provided they can only share linearly (namely, all the shared variables occur linearly in the considered terms). In such a context, only the shared variables can introduce cycles.
6. In the sixth case, we drop the assumption about the finiteness of the term described by t. As a consequence, all variables sharing with x become possibly cyclic. However, provided x describes a finite and linear term, all finite variables independent from x preserve their finiteness.
7. The seventh case is symmetric to the sixth one.
8. The last case states that term finiteness is preserved for all variables that are independent from both x and t. Note that this case is only used when none of the other cases apply.

The following result, together with the assumption on amgu_P as specified in Definition 4, ensures that abstract unification on the combined domain $H \times P$ is correct.

Theorem 2. *Let $\langle h, p \rangle \in H \times P$ and $(x \mapsto t) \in \mathit{Bind}$, where $\{x\} \cup \mathrm{vars}(t) \subseteq \mathit{VI}$. Let also $\sigma \in \gamma_H(h) \cap \gamma_P(p)$ and $h' = \mathrm{amgu}_H\bigl(\langle h, p \rangle, x \mapsto t\bigr)$. Then*

$$\tau \in \mathrm{mgs}\bigl(\sigma \cup \{x = t\}\bigr) \implies \tau \in \gamma_H(h').$$

[7] Let t be y. Let also P be *Pos*. Then, given the *Pos* formula $\phi \stackrel{\mathrm{def}}{=} (x \vee y)$, both $\mathrm{ind}_\phi(x, y)$ and $\mathrm{or\_lin}_\phi(x, y)$ satsify the conditions in Definition 4. Note that from ϕ we cannot infer that x is definitely linear and neither that y is definitely linear.

4 Ongoing and Further Work

4.1 An instance of the parameter domain P

As discussed in Section 3, several abstract domains for sharing analysis can be used to implement the parameter component P. One could consider the well-known set-sharing domain of Jacobs and Langen [26]. In such a case, all the non-trivial correctness results have already been established in [24]: in particular, the abstraction function provided in [24] satisfies the requirement of Definition 3 and the abstract unification operator has been proven correct with respect to rational-tree unification. Note however that, since no freeness and linearity information is recorded in the plain set-sharing domain, some of the predicates of Definition 4 need to be grossly approximated.

Therefore, a better choice would be to consider the abstract domain *SFL* [5] (see also [6]) that represents possible sharing. This domain incorporates the set-sharing domain of Jacobs and Langen with definite freeness and linearity information; the information being encoded by two sets of variables, one satisfying the property of freeness and the other, the property of linearity.

Definition 9. (The set-sharing domain *SH*.) *The set SH is defined by* $SH \stackrel{\mathrm{def}}{=} \wp(SG)$*, where* $SG \stackrel{\mathrm{def}}{=} \wp(VI) \setminus \{\varnothing\}$ *is the set of* sharing groups. *SH is ordered by subset inclusion.*

Definition 10. (The domain *SFL*.) *Let* $F \stackrel{\mathrm{def}}{=} \wp(VI)$ *and* $L \stackrel{\mathrm{def}}{=} \wp(VI)$ *be partially ordered by reverse subset inclusion. The domain SFL is defined by the Cartesian product* $SFL \stackrel{\mathrm{def}}{=} SH \times F \times L$ *ordered by '*$\leq_S$*', the component-wise extension of the orderings defined on the sub-domains.*

Note that a complete definition, besides explicitly dealing with the set of relevant variables VI, would require the addition of a bottom element $\bot$ representing the semantics of those program fragments that have no successful computations.

In the next definition we introduce a few well-known operations on the set-sharing domain *SH*. These will be used to define the operations on the domain *SFL*.

Definition 11. (Abstract operators on *SH*.) *For each* $sh \in SH$ *and each* $V \subseteq VI$*, the extraction of the* relevant component of sh with respect to V *is given by the function* $\mathrm{rel}\colon \wp(VI) \times SH \to SH$ *defined as*

$$\mathrm{rel}(V, sh) \stackrel{\mathrm{def}}{=} \{\, S \in sh \mid S \cap V \neq \varnothing \,\}.$$

For each $sh \in SH$ *and each* $V \subseteq VI$*, the function* $\overline{\mathrm{rel}}\colon \wp(VI) \times SH \to SH$ *gives the* irrelevant component of sh with respect to V. *It is defined as*

$$\overline{\mathrm{rel}}(V, sh) \stackrel{\mathrm{def}}{=} sh \setminus \mathrm{rel}(V, sh).$$

The function $(\cdot)^\star \colon \mathit{SH} \to \mathit{SH}$*, called* star-union*, is given, for each* $sh \in \mathit{SH}$*, by*

$$sh^\star \stackrel{\mathrm{def}}{=} \left\{ S \in \mathit{SG} \;\middle|\; \exists n \geq 1 \,.\, \exists T_1, \dots, T_n \in sh \,.\, S = \bigcup_{i=1}^{n} T_i \right\}.$$

For each $sh_1, sh_2 \in \mathit{SH}$*, the function* $\mathrm{bin} \colon \mathit{SH} \times \mathit{SH} \to \mathit{SH}$*, called* binary union*, is given by*

$$\mathrm{bin}(sh_1, sh_2) \stackrel{\mathrm{def}}{=} \{\, S_1 \cup S_2 \mid S_1 \in sh_1, S_2 \in sh_2 \,\}.$$

It is now possible to define the implementation, on the domain *SFL*, of all the predicates and functions specified in Definition 4.

Definition 12. (Abstract operators on *SFL*.) *For each* $d = \langle sh, f, l \rangle \in \mathit{SFL}$*, for each* $s, t \in \mathit{HTerms}$*, where* $\mathrm{vars}(s) \cup \mathrm{vars}(t) \subseteq \mathit{VI}$*, let* $R_s = \mathrm{rel}\bigl(\mathrm{vars}(s), sh\bigr)$ *and* $R_t = \mathrm{rel}\bigl(\mathrm{vars}(t), sh\bigr)$*. Then*

$$\begin{aligned}
\mathrm{ind}_d(s,t) &\stackrel{\mathrm{def}}{=} \bigl(R_s \cap R_t = \varnothing\bigr); \\
\mathrm{ground}_d(t) &\stackrel{\mathrm{def}}{=} \bigl(\mathrm{vars}(t) \subseteq \mathit{VI} \setminus \mathrm{vars}(sh)\bigr); \\
\mathrm{occ\_lin}_d(y,t) &\stackrel{\mathrm{def}}{=} \mathrm{ground}_d(y) \lor \Bigl(\mathrm{occ\_lin}(y,t) \land (y \in l) \\
&\qquad\qquad \land \forall z \in \mathrm{vars}(t) : \bigl(y \neq z \implies \mathrm{ind}_d(y,z)\bigr)\Bigr); \\
\mathrm{share\_lin}_d(s,t) &\stackrel{\mathrm{def}}{=} \forall y \in \mathrm{vars}(R_s \cap R_t) : \\
&\qquad y \in \mathrm{vars}(s) \implies \mathrm{occ\_lin}_d(y,s) \\
&\qquad \land\, y \in \mathrm{vars}(t) \implies \mathrm{occ\_lin}_d(y,t); \\
\mathrm{free}_d(t) &\stackrel{\mathrm{def}}{=} \exists y \in \mathit{VI} \,.\, (y = t) \land (y \in f); \\
\mathrm{gfree}_d(t) &\stackrel{\mathrm{def}}{=} \mathrm{ground}_d(t) \lor \mathrm{free}_d(t); \\
\mathrm{lin}_d(t) &\stackrel{\mathrm{def}}{=} \forall y \in \mathrm{vars}(t) : \mathrm{occ\_lin}_d(y,t); \\
\mathrm{or\_lin}_d(s,t) &\stackrel{\mathrm{def}}{=} \mathrm{lin}_d(s) \lor \mathrm{lin}_d(t); \\
\mathrm{share\_same\_var}_d(s,t) &\stackrel{\mathrm{def}}{=} \mathrm{vars}(R_s \cap R_t); \\
\mathrm{share\_with}_d(t) &\stackrel{\mathrm{def}}{=} \mathrm{vars}(R_t).
\end{aligned}$$

The function $\mathrm{amgu}_S \colon \mathit{SFL} \times \mathit{Bind} \to \mathit{SFL}$ *captures the effects of a binding on an element of SFL. Let* $d = \langle sh, f, l \rangle \in \mathit{SFL}$ *and* $(x \mapsto t) \in \mathit{Bind}$*, where* $V_{xt} = \{x\} \cup \mathrm{vars}(t) \subseteq \mathit{VI}$*. Let* $R_x = \mathrm{rel}\bigl(\{x\}, sh\bigr)$ *and* $R_t = \mathrm{rel}\bigl(\mathrm{vars}(t), sh\bigr)$*. Let also*

$$\begin{aligned}
sh' &\stackrel{\mathrm{def}}{=} \overline{\mathrm{rel}}(V_{xt}, sh) \cup \mathrm{bin}\bigl(S_x, S_t\bigr), \\
S_x &\stackrel{\mathrm{def}}{=} \begin{cases} R_x, & \text{if } \mathrm{free}_d(x) \lor \mathrm{free}_d(t) \lor \bigl(\mathrm{lin}_d(t) \land \mathrm{ind}_d(x,t)\bigr); \\ R_x^\star, & \text{otherwise;} \end{cases}
\end{aligned}$$

$$S_t \stackrel{\text{def}}{=} \begin{cases} R_t, & \textit{if } \mathrm{free}_d(x) \lor \mathrm{free}_d(t) \lor \bigl(\mathrm{lin}_d(x) \land \mathrm{ind}_d(x,t)\bigr); \\ R_t^\star, & \textit{otherwise;} \end{cases}$$

$$f' \stackrel{\text{def}}{=} \begin{cases} f, & \textit{if } \mathrm{free}_d(x) \land \mathrm{free}_d(t); \\ f \setminus \mathrm{vars}(R_x), & \textit{if } \mathrm{free}_d(x); \\ f \setminus \mathrm{vars}(R_t), & \textit{if } \mathrm{free}_d(t); \\ f \setminus \mathrm{vars}(R_x \cup R_t), & \textit{otherwise;} \end{cases}$$

$$l' \stackrel{\text{def}}{=} \bigl(\mathit{VI} \setminus \mathrm{vars}(\mathit{sh}')\bigr) \cup f' \cup l'';$$

$$l'' \stackrel{\text{def}}{=} \begin{cases} l \setminus \bigl(\mathrm{vars}(R_x) \cap \mathrm{vars}(R_t)\bigr), & \textit{if } \mathrm{lin}_d(x) \land \mathrm{lin}_d(t); \\ l \setminus \mathrm{vars}(R_x), & \textit{if } \mathrm{lin}_d(x); \\ l \setminus \mathrm{vars}(R_t), & \textit{if } \mathrm{lin}_d(t); \\ l \setminus \mathrm{vars}(R_x \cup R_t), & \textit{otherwise.} \end{cases}$$

Then $\mathrm{amgu}_S\bigl(d, x \mapsto t\bigr) \stackrel{\text{def}}{=} \langle \mathit{sh}', f', l' \rangle$.

It is worth noting that, when observing the term finiteness property, set-sharing is strictly more precise than pair-sharing, since a set-sharing domain is strictly more precise when computing the functions $\mathrm{share\_same\_var}_p$ and $\mathrm{share\_lin}_p$.[8] This observation holds regardless of the pair-sharing variant considered, including ASub [9, 35], *PSD* [3] and $\mathsf{Sh}^{\mathsf{PSh}}$ [33].

It remains for us to establish that the relations and functions given in Definition 12 satisfy all the requirements of Definitions 3 and 4. This will require a proof of the correctness, with respect to rational unification, of the abstract operators defined on the domain *SFL*, thereby generalizing and extending the results proved in [24] for the set-sharing domain of Jacobs and Langen.

Note that the domain *SFL* is not the target of the generic specification given in Definition 4; more powerful sharing domains can also satisfy this schema, including all the enhanced combinations considered in [5]. For instance, as the predicate gfree_d defined on *SFL* does not fully exploit the disjunctive nature of its generic specification gfree_p, the precision of the analysis may be improved by adding a domain component explicitly tracking ground-or-freeness, as proposed in [5]. The same argument applies to the predicate $\mathrm{or\_lin}_d$, with respect to $\mathrm{or\_lin}_p$, when considering the combination with the groundness domain *Pos*.

In order to provide an experimental evaluation of the proposed finiteness analysis, we are implementing $H \times P$ where the P component is the *SFL* domain extended with some of the enhancements described in [5]. One of these

[8] For the expert: consider the abstract evaluation of the binding $x \mapsto y$ and the description $\langle h, d \rangle \in H \times \mathit{SFL}$, where $h = \{x, y, z\}$ and $d = \langle \mathit{sh}, f, l \rangle$ is such that $\mathit{sh} = \bigl\{\{x, y\}, \{x, z\}, \{y, z\}\bigr\}$, $f = \varnothing$ and $l = \{x, y, z\}$. Then $z \notin \mathrm{share\_same\_var}_d(x, y)$ so that we have $h' = \{z\}$. In contrast, when using a pair-sharing domain such as *PSD*, the element d is equivalent to $d' = \langle \mathit{sh}', f, l \rangle$, where $\mathit{sh}' = \mathit{sh} \cup \bigl\{\{x, y, z\}\bigr\}$. Hence we have $z \in \mathrm{share\_same\_var}_{d'}(x, y)$ and $h' = \varnothing$. Thus, in sh the information provided by the lack of the sharing group $\{x, y, z\}$ is redundant when observing pair-sharing and groundness, but it is not redundant when observing term finiteness.

enhancements uses information about the actual structure of terms. It has been shown in [2] that this structural information, provided by integrating the generic Pattern$(\cdot)$ construction with *SFL*, can have a key role in improving the precision of sharing analysis and, in particular, allowing better identification where cyclic structures may appear. Thus, it is expected that structural information captured using Pattern$(H \times P)$ can improve the precision of finite-tree analysis; both with respect to the parametric component P and the finiteness component H itself.

4.2 Term-Finiteness Dependencies

The parametric domain $H \times P$ captures the negative aspect of term-finiteness, that is, the circumstances under which finiteness can be lost. When a binding has the potential for creating one or more rational terms, the operator amgu_H removes from h all the variables that may be bound to non-finite terms. However, term-finiteness has also a positive aspect: there are cases where a variable is guaranteed 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, and so forth. SICStus Prolog also provides an explicit `acyclic_term/1` predicate.

The term-finiteness information provided by the h component of $H \times P$ does not capture the information concerning how finiteness of one variable affects the finiteness of other variables. This kind of information, usually termed *relational information*, is very important as it allows the propagation of positive finiteness information. An important source of relational information comes from *dependencies.* Consider the terms $t_1 \stackrel{\mathrm{def}}{=} f(x)$, $t_2 \stackrel{\mathrm{def}}{=} g(y)$, and $t_3 \stackrel{\mathrm{def}}{=} h(x, y)$: 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 reasoning is based on the following facts:

1. t_1, t_2, and t_3 are finite terms, so that the finiteness of their instances depends only on the finiteness of the terms that take the place of x and y.
2. t_3 *covers* both t_1 and t_2, that is, $\mathrm{vars}(t_3) \supseteq \mathrm{vars}(t_1) \cup \mathrm{vars}(t_2)$; this means that, if an assignment to the variables of t_3 produces a finite instance of t_3, that very same assignment will necessarily result in finite instances of t_1 and t_2. Conversely, an assignment producing non-finite instances of t_1 or t_2 will forcibly result in a non-finite instance of t_3.
3. Similarly, t_1 and t_2, taken together, cover t_3.

The important point to notice is that the indicated dependency will continue to hold for any further simultaneous instantiation of t_1, t_2, and t_3. In other words, such dependencies are preserved by forward computations (since they proceed by consistently instantiating program variables).

Consider the abstract binding $x \mapsto t$ where t is a finite term such that $\mathrm{vars}(t) = \{y_1, \ldots, y_n\}$. After this binding has been successfully performed, the destinies of x and t concerning term-finiteness are tied together forever. This tie can be described by the dependency formula

$$x \leftrightarrow (y_1 \wedge \cdots \wedge y_n), \tag{1}$$

meaning that x will be bound to a finite term if and only if, for each $i = 1$, $\ldots$, n, y_i is bound to a finite term. While the dependency expressed by (1) 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 (1) is logically equivalent to

$$x \to (y_1 \wedge \cdots \wedge y_{k-1} \wedge y_{k+1} \wedge \cdots \wedge y_n). \tag{2}$$

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 (2) holds vacuously. The 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."

Following the intuition outlined above, in [4] we have studied a domain, whose carrier is the set of all Boolean functions, for representing and propagating finiteness dependencies. We believe that coupling this new domain with $H \times P$ can greatly improve the precision of the analysis.

5 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 and that can be summarized as follows: several built-ins, library predicates, program analysis and manipulation techniques are only well-defined for program fragments working with finite trees.

In this paper we propose an abstract-interpretation based solution to the problem of detecting program variables that can only be bound to finite terms. The rationale behind this is that applications exploiting rational trees tend to do so in a very controlled way. If the analysis we propose proves to be precise enough, then we will have a practical way of taking advantage of rational trees while minimizing the impact of their disadvantages.

References

1. 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/`.
2. 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, Reunion Island, France, 2000. Springer-Verlag, Berlin.

3. R. Bagnara, P. M. Hill, and E. Zaffanella. Set-sharing is redundant for pair-sharing. *Theoretical Computer Science*, 2001. To appear.
4. R. Bagnara, E. Zaffanella, R. Gori, and P. M. Hill. Boolean functions for finite-tree dependencies. Quaderno 252, Dipartimento di Matematica, Università di Parma, 2001. Available at `http://www.cs.unipr.it/~bagnara/`.
5. 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.
6. M. Bruynooghe, M. Codish, and A. Mulkers. A composite domain for freeness, sharing, and compoundness analysis of logic programs. Technical Report CW 196, Department of Computer Science, K.U. Leuven, Belgium, July 1994.
7. J. A. Campbell, editor. *Implementations of Prolog.* Ellis Horwood/Halsted Press/Wiley, 1984.
8. B. Carpenter. *The Logic of Typed Feature Structures with Applications to Unification-based Grammars, Logic Programming and Constraint Resolution*, volume 32 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, New York, 1992.
9. M. Codish, D. Dams, and E. Yardeni. Derivation and safety of an abstract unification algorithm for groundness and aliasing analysis. In K. Furukawa, editor, *Logic Programming: Proceedings of the Eighth International Conference on Logic Programming*, MIT Press Series in Logic Programming, pages 79–93, Paris, France, 1991. The MIT Press.
10. 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.
11. 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.
12. A. Colmerauer. An introduction to Prolog-III. *Communications of the ACM*, 33(7):69–90, 1990.
13. A. Cortesi and G. Filé. Sharing is optimal. *Journal of Logic Programming*, 38(3):371–386, 1999.
14. 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.
15. 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.
16. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.
17. L. Crnogorac, A. D. Kelly, and H. Søndergaard. A comparison of three occur-check analysers. In R. Cousot and D. A. Schmidt, editors, *Static Analysis: Proceedings of the 3rd International Symposium*, volume 1145 of *Lecture Notes in Computer Science*, pages 159–173, Aachen, Germany, 1996. Springer-Verlag, Berlin.
18. P. R. Eggert and K. P. Chow. Logic programming, graphics and infinite terms. Technical Report UCSB DoCS TR 83-02, Department of Computer Science, University of California at Santa Barbara, 1983.

19. G. Erbach. ProFIT: Prolog with Features, Inheritance and Templates. In *Proceedings of the 7th Conference of the European Chapter of the Association for Computational Linguistics*, pages 180–187, Dublin, Ireland, 1995.
20. M. Filgueiras. A Prolog interpreter working with infinite terms. In Campbell [7], pages 250–258.
21. F. Giannesini and J. Cohen. Parser generation and grammar manipulation using Prolog's infinite trees. *Journal of Logic Programming*, 3:253–265, 1984.
22. W. Hans and S. Winkler. Aliasing and groundness analysis of logic programs through abstract interpretation and its safety. Technical Report 92–27, Technical University of Aachen (RWTH Aachen), 1992.
23. S. Haridi and D. Sahlin. Efficient implementation of unification of cyclic structures. In Campbell [7], pages 234–249.
24. 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`.
25. B. Intrigila and M. Venturini Zilli. A remark on infinite matching vs infinite unification. *Journal of Symbolic Computation*, 21(3):2289–2292, 1996.
26. D. Jacobs and A. Langen. Accurate and efficient approximation of variable aliasing in logic programs. In E. L. Lusk and R. A. Overbeek, editors, *Logic Programming: Proceedings of the North American Conference*, MIT Press Series in Logic Programming, pages 154–165, Cleveland, Ohio, USA, 1989. The MIT Press.
27. J. Jaffar, J-L. Lassez, and M. J. Maher. Prolog-II as an instance of the logic programming scheme. In M. Wirsing, editor, *Formal Descriptions of Programming Concepts III*, pages 275–299. North-Holland, 1987.
28. T. Keisu. *Tree Constraints*. PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, May 1994.
29. A. King. Pair-sharing over rational trees. *Journal of Logic Programming*, 46(1–2):139–155, 2000.
30. M. J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proceedings, Third Annual Symposium on Logic in Computer Science*, pages 348–357, Edinburgh, Scotland, 1988. IEEE Computer Society.
31. K. Mukai. *Constraint Logic Programming and the Unification of Information*. PhD thesis, Department of Computer Science, Faculty of Engineering, Tokio Institute of Technology, 1991.
32. C. Pollard and I. A. Sag. *Head-Driven Phrase Structure Grammar*. University of Chicago Press, Chicago, 1994.
33. F. Scozzari. Abstract domains for sharing analysis by optimal semantics. In J. Palsberg, editor, *Static Analysis: 7th International Symposium, SAS 2000*, volume 1824 of *Lecture Notes in Computer Science*, pages 397–412, Santa Barbara, CA, USA, 2000. Springer-Verlag, Berlin.
34. Gert Smolka and Ralf Treinen. Records for logic programming. *Journal of Logic Programming*, 18(3):229–258, 1994.
35. 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.
36. Swedish Institute of Computer Science, Programming Systems Group. *SICStus Prolog User's Manual*, release 3 #0 edition, 1995.