Generalising *Def* and *Pos* to Type Analysis

Patricia M. Hill
School of Computing, University of Leeds
LS2 9JT, Leeds, UK
hill@scs.leeds.ac.uk
Ph.: +44 (0)1132336807 Fax: +44 (0)1132335468

Fausto Spoto*
IRISA
Campus Universitaire de Beaulieu, Avenue du Général Leclerc
35042 Rennes Cedex, France
spoto@irisa.fr

Abstract

This paper is concerned with the type analysis of logic programs where, by *type*, we mean a property closed under instantiation. We define a chain of abstractions from Herbrand constraints to logical formulas via the set of their solutions. Every step of the chain is an instance of abstract interpretation. The use of logical formulas for type analysis is a generalisation of the traditional Boolean domains *Def* and *Pos* for groundness analysis. In this context, implication is the logical counterpart of the use of linear refinement. While logical formulas can sometime be used for an actual implementation of our domains, in the general case they are infinite objects. Therefore, we apply a final abstraction from possibly infinite logical formulas to (finite) logic programs. Thus, logic programs are themselves used for the type analysis of logic programs.

The advantage of our technique with respect to the many frameworks for type analysis present in the literature is that we have developed our domains by using the formal techniques of abstract interpretation and linear refinement. Therefore, their construction is guided by the underlying theory, from which their properties are derived.

Keywords: *Abstract interpretation, domain theory, linear refinement, type theory, type analysis, logic programming.*

1 Introduction

This paper is concerned with the type analysis of logic programs where, by *type*, we mean a downward closed property, that is, a property closed under instantiation. For instance, the set of integers and the set of lists are types, since

*Part of this work was done while Fausto Spoto was visiting the School of Computing of the University of Leeds, UK. His visit has been supported by EPSRC grant GR/M05645.

once a variable is bound to an integer or a list it will maintain this property throughout the computation. Similarly, the set of all difference lists is a type. On the other hand, the set of all free variables is not a type since freeness can be lost by computation. Type analysis is the upward approximation of the success set of a program through types.

Type analysis of logic programs is important for optimisation of unification as well as for verification. For instance, the programmer can use type analysis to check that the arguments of all procedure calls that can arise at run-time actually belong to some types. Note that we do not consider how to check whether they are input or output parameters. Similarly, if a compiler knows that a given variable is bound to an integer in a given program point then it can generate a specialised code for the unification of that variable.

A well-known and useful type, which distinguishes whether a term contains variables or not, is groundness [2, 10, 11]. The usual domains for groundness analysis, *Def* and *Pos*, feature some desirable properties: simplicity, effectivity, usefulness. Moreover, it has been shown [34, 44] that *Pos* is condensing, i.e., it propagates the properties of being groundness in the best possible way. Finally, it can be used for abstract compilation [8, 23]. All these good properties of the *Pos* domain should have encouraged a generalisation of *Pos* to a general type domain. Instead, type domains have been developed, up to now, in a way which is almost always totally independent from the domain *Pos* for groundness analysis. A generalisation of groundness to generic types is given in [8], but it is assumed, without any proof of correctness, that the usual properties that hold for *Def* and *Pos* in the case of groundness still hold for all the type domains. In [42] the domain *Pos* is combined with type information. However, the resulting abstract domain is not induced by any underlying theory, and it is not possible to speak of any form of optimality for it. We do not know of any approach where the abstract domains for inferring a generic type system are developed in an automatic way, and choices about the representation and the algorithms are *implied* by the same theory of abstract interpretation. This is exactly the distinguishing feature of our construction. Note that we discuss the related work in more detail in Subsection 1.1.

In this paper we generalise the construction of groundness to a generic type domain. In Section 3 we show that the domain H of existential Herbrand constraints is isomorphic to the domain Sol of their solutions. Therefore, the powersets $\wp(H)$ and $\wp(Sol)$, the domains of the collecting semantics, can express any properties of the existential Herbrand constraints. Since we are interested in types, we abstract these powersets into a domain *Down* of downward (instantiation) closed sets of substitutions. The optimal counterparts over *Down* of the operations over existential Herbrand constraints are explicitly defined. In Section 4 we consider a type dependent abstraction from *Down* to a set T of transfinite formulas. If, in this abstraction, we use groundness and the set of positive transfinite formulas, we obtain the domain *Pos* for groundness analysis. We obtain *Def* by considering definite transfinite formulas. However, our construction is much more general, and can be applied to every type and set of transfinite formulas. Moreover, if a very weak condition holds for the type and the set of formulas, then the good properties of *Def* and *Pos* in the case of groundness can be generalised to the new abstract domain of transfinite formulas. Namely, a Galois insertion exists between *Down* and that domain so that it does not contain useless elements. Moreover, the traditional abstract operations over transfinite formulas (conjunction and Schröder elimination) are the optimal counterparts of the operations over *Down*. Therefore, the use of transfinite formulas as a representation of type domains does not introduce any loss of precision for the computation of the abstract operators. We show that some sets of transfinite formulas are *optimal*, in the sense of being closed w.r.t. a linear refinement operation.

The generalisation of the set *Def* of transfinite formulas to generic types can be used for actual type analysis only in the very special case when its formulas are finite, like in the case of groundness or non-freeness. In Section 5 we show that, when this is not the case, the transfinite formulas of *Def* can be abstracted into a finite domain *Prog* of logic programs. We provide correct and sometimes optimal counterparts over logic programs of the operations over transfinite formulas. Therefore, we justify, by a formal construction through abstract interpretation and linear refinement, the use of logic programs for the analysis of logic programs themselves [16]. In Section 6 we show some examples of type analysis of logic programs through our domain *Prog*.

The picture below synthesises the various domains considered in this paper, and their relationships as abstraction (represented by horizontal arrows) or lifting to the powerset (represented by vertical arrows).

$$
\begin{array}{lccccc|ccccc}
\text{collecting} & \wp(H) & \leftrightarrow & \wp(Sol) & \to & Down & \to & T & \to & Prog \\
 & \uparrow & & \uparrow & & & & & & \\
\hline
\text{non-collecting} & H & \leftrightarrow & Sol & & & & & \text{type} & \\
 & & & & & & & & \text{dependent} & \\
 & & \multicolumn{3}{c}{\text{type independent abstractions}} & & & & \text{abstractions.} &
\end{array}
$$

Partial and preliminary versions of this paper appeared in [25] and [32].

1.1 Related work

It is common to divide approaches to types (in logic programming) into those that require the types to be declared by the user and those that expect them to be inferred by the system [43]. Type systems designed for the first approach are often said to be *prescriptive* whereas those intended for the second are called *descriptive* [41]. However, this division is rather artificial and instead, we prefer to see a continuous spectrum between completely type specified programs and untyped ones. At the top end of the spectrum, when the types are completely specified, the type checking is then a matter of exploiting any redundant information to check that the program and type declarations are consistent. When the rules for the type system together with a partial type specification in the program are sufficient to uniquely specify the program types, then type reconstruction is used to determine any missing type information [31, 36]. The majority of typed logic programming languages such as Gödel [24] and Mercury [46] use a combination of type checking and type reconstruction, the latter often being used to determine the types of the variables. If, however, insufficient or no type information is provided, then it is the job of a type inference tool to type the program so that the program is well-typed and any results that may be computed are also well-typed. In all cases, it is assumed that no type errors can occur at run-time. Although, in this paper, we are concerned with type inference and hence towards the lower end of this spectrum, we do assume that the types themselves are already defined. Moreover, as we are interested in generalising the groundness analysis techniques to types, we require that these types enjoy the same downward closed property that the groundness domains possess. Such a condition on types is common in work on type analysis [8, 28]. In particular, this means that, if the analyser infers a typing of the program so that each clause is well-typed, then every instance of the clause will be well-typed.

Regarding the actual approach to type analysis, some techniques are similar to those developed for (higher order) functional languages (see, for example, [3, 30, 40, 41, 49]) while others are inspired by program verification methods [1]. Others use type graphs [28, 47]. We use here the abstract interpretation framework of [13] which is the basis for the type analysis techniques of many proposals [4, 8, 9, 28, 29, 33, 45, 48, 49].

The first step in designing a type inference system based on abstract interpretation is to decide on the abstract domain. For type inference, it is important that the abstract domain can express generic dependencies between the types. As shown in [7], if the types are ground (i.e., *monomorphic*), one cannot handle generic type dependencies. This is illustrated in [4] which describes an inference system which uses only ground types. As a result their abstract domains are usually infinite and hence impractical without widening. *Polymorphic* types using type variables (often called *parameters*) in the type language were first proposed for logic programming in [37] and then formalised in [22, 26][1], although these were intended for use with type checking rather than type inference. These types have since been adapted in a number of ways for use with type inference systems, such as in [29, 33, 48]. The use of parametric polymorphism to express type dependencies between a procedure's arguments is a standard solution, used for instance in [4, 9, 28, 49]. The same solution is used in the framework of regular approximation of the success set in [18, 49]. However, the use of type variables does not allow one to express all type dependencies between argument positions. Only in [4, 8, 42] are there examples of domains which *explicitly* allow one to express type dependencies between polymorphic types.

There are two ways in which groundness analysis may be generalised to types. First, as is the case in [45], the property of groundness itself can be generalised. There, it is assumed that the language is already completely typed and the authors provide a means of constructing mode domains for representing different degrees of instantiation of well-typed expressions occurring in the execution of a program. Thus the typed modes generalise the property of whether or not a variable is bound to a ground term and are intended for use with abstract compilation. Secondly, as described in [8], polymorphic types may be obtained through a generalisation of a domain like *Pos*, the domain for propagating groundness. There, it is assumed that the usual properties that hold for *Pos* in the case of groundness still hold for its generalisation to types. For instance, logical conjunction between formulas is used as conjunction operator and Schröder elimination as cylindrification operator. However, it is not obvious at all that these operators, which are optimal in the case of groundness analysis, as shown in [11], are even correct in the general case of type analysis and no proof is given. Although in [9] a domain with properties similar to those of *Pos* is defined, it is not a generalisation of *Pos*. For instance, it is not made of logi-

[1] Although, as indicated in [5] there is an error in the development of the type system in [26], this does not affect the results for pure parametric polymorphism when subtypes are ignored.

cal formulas. Finally, in [42], *Pos* is combined with type information. However, their contruction is not the result of any automatic, methodological construction which starts from the properties of interest and leads to the abstract domain.

Acknowledgements. We are grateful to Giorgio Levi for his contribution to the technical ideas described here and also to the anonymous referee for useful suggestions for improvement.

2 Preliminaries

The *powerset* of a set S is $\wp(S) = \{S' \mid S' \subseteq S\}$. We denote by $\wp_f(S)$ the set of all finite subsets of S. If S is partially ordered w.r.t. $\leq$ and $s \in S$, we denote by $\downarrow s = \{s' \in S \mid s' \leq s\}$ the *downward closure* of s and by $\wp\!\downarrow\!(S) = \{S' \subseteq S \mid S' = \downarrow S'\}$ the set of all the downward closed sets of S. A sequence is an ordered collection of elements. The set of sequences over S is denoted by $\mathrm{Seq}(S)$. If $\tilde{x}$ is a sequence we will silently assume that $\tilde{x} = \langle x_1, \ldots, x_l \rangle$, where l is the length of the sequence.

2.1 S-semantics

We define the semantics of logic programs by the use of the s-semantics approach [6]. The s-semantics is a bottom-up, fixpoint definition of the set of computed answers of a program, though it can be rephrased for call pattern or resultant analysis [17].

We assume there is an infinite set of program variables $\mathcal{V}$. For our purposes, we give a very abstract definition of constraint system over $\mathcal{V}$ as a data structure together with three operations.

Definition 1. A *constraint system* over a set of variables $\mathcal{V}$ is a family of sets $D = \{D_V\}_{V \in \wp_f(\mathcal{V})}$ together with three operations: for $V \in \wp_f(\mathcal{V})$ we have a (partial) infix conjunction operation $\star^{D_V} : D_V \times D_V \to D_V$, a (partial) renaming operation $\mathsf{rename}^{D_V} : \mathrm{Seq}(V) \times \mathrm{Seq}(V) \times D_V \to D_V$ and a (total) cylindrification operation $\exists^{D_V} : V \times D_V \to D_V$. We write $\mathsf{rename}^{D_V}_{\tilde{x} \to \tilde{y}}\, c$ for $\mathsf{rename}^{D_V}(\tilde{x}, \tilde{y}, c)$ and $\exists^{D_V}_x c$ for $\exists^{D_V}(x, c)$. In the following, when speaking of renaming, we will silently assume that $\tilde{x} = \langle x_1, \ldots, x_l \rangle$ and $\tilde{y} = \langle y_1, \ldots, y_l \rangle$ are two disjoint sequences in V without repetitions.

Note that the definition above is very abstract since it does not make any assumption about the behaviour of conjunction, renaming and cylindrification except for their signatures.

Example 2. For every $V \in \wp_f(\mathcal{V})$, let $D_V = \wp(V)$. This means that the constraints over the variables V are all the subsets of V. We define the operations $\star^{D_V}(V_1)(V_2) = V_1 \cup V_2$, $\mathsf{rename}^{D_V}_{\tilde{x} \to \tilde{y}}(V') = V'[\tilde{y}/\tilde{x}]$ and $\exists^{D_V}_x(V') = V' \setminus \{x\}$, where $[\tilde{y}/\tilde{x}]$ means substitution of the variables $\tilde{x}$ with the variables $\tilde{y}$ (see Subsection 2.2). These operations respect the signatures of Definition 1.

In Subsection 2.3 we will see a more complex and useful constraint system. It turns out that the constraints of Example 2 track the set of variables *used* by the constraints of Subsection 2.3. Moreover, in the following sections we will present every abstract domain as a constraint system.

Given a constraint system, we can define the set of goals and programs.

Definition 3. Let $D = \{D_V\}_{V \in \wp_f(\mathcal{V})}$ be a constraint system and Π a finite set of predicate symbols with associated arity. We denote by π the set of distinct variables $\{\iota_1, \ldots, \iota_m\}$ where m is the maximum arity of the predicates in Π.

Assume $\pi \subseteq V$. By $\mathbb{G}^{D_V}$ we refer to the set of *goals* over D_V, as defined by the grammar

$$G^{D_V} ::= c \mid G^{D_V} \texttt{ and } G^{D_V} \mid G^{D_V} \texttt{ or } G^{D_V} \mid \mathtt{p}(x_1, \ldots, x_l)$$

where $c \in D_V$, $\mathtt{p}^l \in \Pi$ with $l \geq 0$ and $\{x_1, \ldots, x_l\} \subseteq V$ are distinct variables not in π.

By $\mathbb{P}^{D_V}$ we refer to the set of *programs* over D_V, i.e., to the set of sets of clauses, at most one for every predicate symbol, where the clause for $\mathtt{p}^l$ has the form

$$\mathtt{p}(y_1, \ldots, y_l) \leftarrow G$$

where $G \in \mathbb{G}^{D_V}$ and $\{y_1, \ldots, y_l\} \subseteq V$ are distinct variables not in π.

We write $\mathtt{p}$ for $\mathtt{p}()$ if $\mathtt{p}$ has arity 0.

Note that this abstract syntax will be used only for the programs we want to analyse. When we consider a Prolog program before its transformation into the syntax of Definition 3, instead, we will use its standard syntax [27].

The meaning of a program is an interpretation, i.e., a map from predicate symbols to sets of constraints.

Definition 4. An *interpretation* over the constraint system D_V is a function $I : \Pi \to \wp(D_V)$. The set of interpretations over D_V is denoted by $\mathbb{I}^{D_V}$. The set $\mathbb{I}^{D_V}$ is a complete lattice w.r.t. the $\leq$ ordering defined as $I_1 \leq I_2$ if and only if $I_1(\mathtt{p}) \subseteq I_2(\mathtt{p})$ for every $\mathtt{p} \in \Pi$. The least upper bound and greatest lower bound operations are $\cup$ and $\cap$ defined as

$$(\cup_{j \in J}\{I_j\})(\mathtt{p}) = \cup_{j \in J}(I_j(\mathtt{p})) \ , \qquad (\cap_{j \in J}\{I_j\})(\mathtt{p}) = \cap_{j \in J}(I_j(\mathtt{p})) \ ,$$

respectively, with $\{I_j\}_{j \in J} \subseteq \mathbb{I}^{D_V}$ and $J \subseteq \mathbb{N}$. The bottom interpretation $\bot$ is such that $\bot(\mathtt{p}) = \emptyset$ for every $\mathtt{p} \in \Pi$.

If $\mathtt{p}^l \in \Pi$ then $I(\mathtt{p})$ is the *meaning* that I gives to $\mathtt{p}$. Every constraint in $I(\mathtt{p})$ refers to the arguments of $\mathtt{p}$ through the variables in π. For instance, the first argument is referred to as ι_1, the second as ι_2 and so on. This means that we can obtain the meaning of a procedure call $\mathtt{p}(y_1, \ldots, y_l)$ simply by substituting y_i by ι_i for every $i = 1, \ldots, l$.

Given an interpretation, we define the evaluation of a goal (query), i.e., the set of computed answers obtained by executing the goal in a context where procedure calls are denoted by the interpretation.

Definition 5. Given $\{S, S_1, S_2\} \subseteq \wp(D_V)$ and $\tilde{x} \in \mathrm{Seq}(V)$, let

$$\begin{aligned}
S_1 \otimes^{D_V} S_2 &= \{c_1 \star^{D_V} c_2 \mid c_1 \in S_1,\ c_2 \in S_2 \text{ and } c_1 \star^{D_V} c_2 \text{ is defined}\} \ , \\
S_1 \oplus^{D_V} S_2 &= S_1 \cup S_2 \ , \\
\mathsf{rename}^{D_V}_{\tilde{x} \to \tilde{y}} S &= \{\mathsf{rename}^{D_V}_{\tilde{x} \to \tilde{y}} c \mid c \in S \text{ and } \mathsf{rename}^{D_V}_{\tilde{x} \to \tilde{y}} c \text{ is defined}\} \ , \\
\exists\!\exists^{D_V}_{\tilde{x}} S &= \{\exists^{D_V}_{x_1} \cdots \exists^{D_V}_{x_l} c \mid c \in S\} \ .
\end{aligned}$$

We define $\mathcal{CA}^{D_V}[\![]\!] : \mathbb{G}^{D_V} \times \mathbb{I}^{D_V} \to \wp(D_V)$ as

$$\begin{aligned}
\mathcal{CA}^{D_V}[\![c]\!]I &= \{c\}\\
\mathcal{CA}^{D_V}[\![G_1 \texttt{ and } G_2]\!]I &= \mathcal{CA}^{D_V}[\![G_1]\!]I \otimes^{D_V} \mathcal{CA}^{D_V}[\![G_2]\!]I\\
\mathcal{CA}^{D_V}[\![G_1 \texttt{ or } G_2]\!]I &= \mathcal{CA}^{D_V}[\![G_1]\!]I \oplus^{D_V} \mathcal{CA}^{D_V}[\![G_2]\!]I\\
\mathcal{CA}^{D_V}[\![\mathtt{p}(x_1,\ldots,x_l)]\!]I &= \mathsf{rename}^{D_V}_{\tilde{\iota}\to\tilde{x}}(I(\mathtt{p}))\ .
\end{aligned}$$

The immediate consequence operator improves an interpretation for a program by using its clauses.

Definition 6. Given a program $P \in \mathbb{P}^{D_V}$, the *immediate consequence operator* $T_P^{D_V} : \mathbb{I}^{D_V} \to \mathbb{I}^{D_V}$ is defined as

$$T_P^{D_V}(I)(\mathtt{p}^l) = \begin{cases} \exists^{D_V}_{V\setminus\tilde{\iota}}\, \mathsf{rename}^{D_V}_{\tilde{y}\to\tilde{\iota}}\, \mathcal{CA}^{D_V}[\![G]\!]I & \text{if } (\mathtt{p}(y_1,\ldots,y_l) \leftarrow G) \in P\\ \emptyset & \text{otherwise,}\end{cases}$$

for every predicate symbol $\mathtt{p}^l \in \Pi$. The exponent D_V will be omitted when it is clear from the context.

Proposition 7. *Given a program $P \in \mathbb{P}^{D_V}$, T_P is additive.*

Proposition 7 allows us to give the following definition.

Definition 8. Given a program $P \in \mathbb{P}^{D_V}$, its *computed answer semantics* is

$$\mathcal{S}_P^{D_V} = \bigcup_{n\geq 0} T_P^{D_V}{\uparrow}n(\bot)\ ,$$

where $T_P{\uparrow}0(\bot) = \bot$ and $T_P{\uparrow}i{+}1(\bot) = T_P(T_P{\uparrow}i(\bot))$. When clear from the context, we will drop the exponent D_V.

In the following, we will instantiate the definitions of this subsection with various constraint systems, starting from that of existential Herbrand constraints.

2.2 Terms and substitutions

Given a set of variables V, a set of function symbols Σ with associated arity and $k \in \mathbb{N}$, we define

$$\begin{aligned}
\mathsf{terms}^0(\Sigma,V) &= V\\
\mathsf{terms}^{k+1}(\Sigma,V) &= \mathsf{terms}^k(\Sigma,V)\\
&\quad\cup \left\{ \mathtt{f}(t_1,\ldots,t_n) \;\middle|\; \begin{array}{l} \mathtt{f}^n \in \Sigma \text{ and}\\ \{t_1,\ldots,t_n\} \subseteq \mathsf{terms}^k(\Sigma,V)\end{array} \right\}\\
\mathsf{terms}(\Sigma,V) &= \bigcup_{d\geq 0} \mathsf{terms}^d(\Sigma,V)\ .
\end{aligned}$$

We will always assume that Σ contains at least one function symbol of arity 0. We denote by $\mathsf{vars}(t)$ the set of variables which occur in a term t. When $\mathsf{vars}(t) = \emptyset$ we say that the term t is *ground.* Given a set of variables V and a variable x, $V \cup x$ means $V \cup \{x\}$ and $V \setminus x$ means $V \setminus \{x\}$. Variable substitution in a term t is denoted by $t[y/x]$ (x is substituted by y). If $\tilde{x} = \langle x_1,\ldots,x_l\rangle$

and $\tilde{y} = \langle y_1, \ldots, y_l \rangle$ are two sequences of variables without repetitions then $t[\tilde{y}/\tilde{x}]$ stands for $t[y_1/x_1, \ldots, y_l/x_l]$, i.e., parallel substitution of variables. If $f : \wp(\mathsf{terms}(\Sigma, V)) \to \wp(\mathsf{terms}(\Sigma, V))$ then $\mu t.f(t)$ denotes the least fixpoint of f, when it exists.

We define $\Theta^Z_{V,W}$, with $Z \subseteq V \cap W$, as the set of substitutions θ such that $\mathsf{dom}(\theta) \subseteq V$, $\theta(x) \in \mathsf{terms}(\Sigma, W)$ for every $x \in V$ and $\mathsf{dom}(\theta) \cap \mathsf{rng}(\theta) \subseteq Z$. If $Z = \emptyset$ we omit the superscript. The elements of $\Theta_{V,W}$ are called *idempotent* substitutions [38]. We write Θ_V for $\Theta_{V,V}$ and Θ^Z_V for $\Theta^Z_{V,V}$. We denote by ε the empty substitution. Given θ and a set of variables R, we define $\theta|_R(x) = \theta(x)$ if $x \in R$ and $\theta|_R(x) = x$ otherwise. Given a term $t \in \mathsf{terms}(\Sigma, V)$ and $\theta \in \Theta^Z_{V,W}$, $t\theta \in \mathsf{terms}(\Sigma, W)$ is the term obtained with parallel substitution of every variable x in t with $\theta(x)$. *Composition* of substitutions $\theta \in \Theta_{V,W}$ and $\sigma \in \Theta_{W,Z}$ is defined as $(\theta\sigma)(x) = \theta(x)\sigma$ for every $x \in V \cup W$. We sometimes write $\theta \circ \sigma$ for $\theta\sigma$. If $n \in V$, $x \in V \setminus n$ and $\sigma \in \Theta_{V \setminus n}$ then $\sigma[n/x] \in \Theta_V$ is defined as $\sigma[n/x](x) = x$, $\sigma[n/x](n) = \sigma(x)[n/x]$ and $\sigma[n/x](y) = \sigma(y)[n/x]$ if $y \neq x$ and $y \neq n$. This notation is extended to sets of substitutions.

For every set of variables V, a preorder is defined on Θ_V as $\theta' \leq_V \theta$ if there exists a substitution $\sigma \in \Theta^V_V$ such that[2] $\theta' = \theta\sigma$. When V is clear from the context, we write $\leq$ instead of $\leq_V$. A preorder, called *subsumption*, is defined on $\mathsf{terms}(\Sigma, V)$ as $t_1 \leq_V t_2$ (t_1 is an instance of t_2) if $t_1 = t_2\theta$ for a suitable $\theta \in \Theta^V_V$. As usual, the subscript is omitted in $\leq_V$ when it is clear from the context. By $\equiv$ we denote the associated equivalence relation (*variance*).

2.3 Existential Herbrand constraints

Let Σ be a set of function symbols with associated arity and V a finite set of variables. We define the set of finite sets of *Herbrand equations* C_V as

$$C_V = \wp_f(\{t^1 = t^2 \mid t^1, t^2 \in \mathsf{terms}(\Sigma, V)\}) \ .$$

Every substitution can be seen as a set of Herbrand equations. The embedding map is $\mathsf{Eq}(\theta) = \{v = \theta(v) \mid v \in \mathsf{dom}(\theta)\}$. Therefore, we assume that $\Theta^Z_V \subseteq C_V$ for every $Z \subseteq V$.

Suppose $c \in C_V$. We say that $c\theta$ *is true* if $t^1\theta$ is syntactically equal to $t^2\theta$ for every $(t^1 = t^2) \in c$. We know [35] that if there exists $\theta \in \Theta_V$ such that $c\theta$ is true, then c can be put in the *normal form* $\mathsf{mgu}(c) \in \Theta_V$ which is such that $c\theta$ is true if and only if $\mathsf{mgu}(c)\theta$ is true. If no $\theta \in \Theta_V$ exists such that $c\theta$ is true, then $\mathsf{mgu}(c)$ is undefined.

Let $\mathcal{V}$ and $\mathcal{W}$ be disjoint infinite sets of variables. For each $V \in \wp_f(\mathcal{V})$, we have a set of constraints, called *existential Herbrand constraints*, given by

$$H_V = \left\{ \exists_W c \;\middle|\; \begin{array}{l} W \in \wp_f(\mathcal{W}),\ c \in C_{V \cup W}, \\ \text{there exists } \theta \in \Theta_{V \cup W, V} \text{ such that } c\theta \text{ is true} \end{array} \right\} .$$

Here, $\mathcal{V}$ are called the *program variables* and $\mathcal{W}$ the *existential variables.* The requirement about the existence of θ in the definition of H_V means that we consider only satisfiable constraints. A constraint $\exists_W c$ is said to be in *normal form* if c is in normal form.

The set of *solutions* of an existential Herbrand constraint is

$$\mathsf{sol}_V(\exists_W c) = \{\theta|_V \mid \theta \in \Theta_{V \cup W, V} \text{ and } c\theta \text{ is true}\} \ .$$

[2] Note that σ is not required to be idempotent, even when θ and θ' are idempotent [38].

Hence $\mathsf{sol}_V(\exists_W c) = \mathsf{sol}_V(\exists_W \mathsf{mgu}(c))$. Note that $\mathsf{sol}_V(\exists_W c)$ is downward closed. In the special case when $\exists_W c$ is a substitution, i.e., when $W = \emptyset$, we have the following result.

Proposition 9. *Given $\theta \in \Theta_V$ we have* $\mathsf{sol}_V(\mathsf{Eq}(\theta)) = \downarrow\{\theta\}$.

A preorder is defined on H_V as $h_1 \leq h_2$ if and only if $\mathsf{sol}_V(h_1) \subseteq \mathsf{sol}_V(h_2)$. This preorder becomes a partial order if we consider equivalence classes of constraints, where $h_1, h_2 \in H_V$ are called *equivalent* if and only if $\mathsf{sol}_V(h_1) = \mathsf{sol}_V(h_2)$. In the following, a constraint will stand for its equivalence class. Since, as shown above, every constraint can be put in an equivalent normal form, in the following we will consider only normal constraints. Therefore, existential Herbrand constraints can be seen as existential substitutions.

The set of existential Herbrand constraints can be seen as a constraint system, once it is endowed with the operations of Definition 1.

Definition 10. Given $V \in \wp_f(\mathcal{V})$, $x \in V$ and $\tilde{x}, \tilde{y} \in \mathrm{Seq}(V)$, we define [3]

$$(\exists_{W_1} c_1) \star^{H_V} (\exists_{W_2} c_2) = \begin{cases} \exists_{W_1 \cup W_2} \mathsf{mgu}(c_1 \cup c_2) & \text{if } \mathsf{mgu}(c_1 \cup c_2) \text{ exists} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\exists_x^{H_V}(\exists_W c) = \exists_{W \cup N} c[N/x] \qquad \text{with } N \in \mathcal{W} \setminus W$$

$$\mathsf{rename}_{\tilde{x} \to \tilde{y}}^{H_V}(\exists_W c) = \begin{cases} \exists_W c[\tilde{y}/\tilde{x}] & \text{if } \exists_W c \in H_{V \setminus \tilde{y}} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

2.4 Abstract interpretation

Abstract interpretation [13] is a theory developed to reason about the abstraction relation between two different domains (the *concrete* and the *abstract* domain).

We recall that a complete lattice L is a partially ordered set where least upper bound (or *join*, denoted by $\sqcup$) and greatest lower bound (or *meet*, denoted by $\sqcap$) exist for every subset of L. A function f on L is additive when for every $I \subseteq \mathbb{N}$ and $\{l_i\}_{i \in I} \subseteq L$ we have $f(\sqcup_{i \in I}\{l_i\}) = \sqcup_{i \in I} f(l_i)$. It is co-additive when, in the same condition, we have $f(\sqcap_{i \in I}\{l_i\}) = \sqcap_{i \in I} f(l_i)$.

A *Moore family* M of C is a topped completely meet-closed subset of C, i.e., M contains the top element of C and is closed w.r.t. arbitrary meets. The *Moore (intersection) closure* of a set A is denoted by $\curlywedge(A)$.

Definition 11. Let $(C, \leq)$ and $(A, \preceq)$ be two complete lattices (the concrete and the abstract domain). A *Galois connection* $\langle \alpha, \gamma \rangle : (C, \leq) \rightleftharpoons (A, \preceq)$ is a pair of monotonic maps $\alpha : C \to A$ (*abstraction*) and $\gamma : A \to C$ (*concretisation*) such that for each $x \in C$ we have $x \leq (\gamma \circ \alpha)(x)$, and for each $y \in A$ we have $(\alpha \circ \gamma)(y) \preceq y$. Moreover, a *Galois insertion* (of $(C, \leq)$ into $(A, \preceq)$) $\langle \alpha, \gamma \rangle : (C, \leq) \rightleftharpoons (A, \preceq)$ is a Galois connection where $\alpha \circ \gamma$ is the identity map on A.

The composition of Galois connections is a Galois connection. The composition of Galois insertions is a Galois insertion. A Galois connection is a Galois

[3] In $\star^{H_V}$, we can assume $W_1 \cap W_2 = \emptyset$ since the names of existential variables are irrelevant: given an existential Herbrand constraint $\exists_W c$, the constraint $\exists_{W'} c[W'/W]$ is equivalent to it. Hence we can always assume existential Herbrand constraints to be renamed apart with respect to existential variables. Similarly, the choice of N in $\exists_x^{H_V}$ is irrelevant.

insertion if and only if γ is one-to-one or, equivalently, if and only if α is onto. If $\alpha : C \to A$ is additive or $\gamma : A \to C$ is co-additive, then $\langle \alpha, \gamma \rangle$ is a Galois connection from C to A. In a Galois insertion, the abstraction map uniquely identifies the concretisation map and vice versa. It is well-known [12] that the set of Galois insertions from C to A is isomorphic to the set of the Moore families of C.

Let $f : C^n \to C$ be a concrete operator and assume that $\tilde{f} : A^n \to A$. Then $\tilde{f}$ is *correct* with respect to f if and only if for all $y_1, \ldots, y_n \in A$ we have $\alpha(f(\gamma(y_1), \ldots, \gamma(y_n))) \preceq \tilde{f}(y_1, \ldots, y_n)$. For each operator f, there exists an *optimal* (*most precise*) correct abstract operator $\tilde{f}$ defined as $\tilde{f}(y_1, \ldots, y_n) = \alpha(f(\gamma(y_1), \ldots, \gamma(y_n)))$. We say that $\tilde{f}$ is *precise* w.r.t. f if and only if for every $x_1, \ldots, x_n \in C$ we have $\tilde{f}(\alpha(x_1), \ldots, \alpha(x_n)) = \alpha(f(x_1, \ldots, x_n))$. Every precise operator is optimal. The following properties are well-known: The composition of correct operators is correct; the composition of precise operators is precise; but the composition of optimal operators is not necessarily optimal.

Consider our denotational semantics of Subsection 2.1 over a constraint system C. It is traditional to define an equivalent operational semantics, in the form of a transition system. Every abstraction A of C forms a constraint system with the optimal (most precise) abstract counterparts of the operations of Definition 1 over C. However, there is no guarantee that the precision of the instantiation with A of the denotational semantics is the same as that of the instantation with A of the operational one. When this is the case, we say that the abstract domain A is *condensing* [21].

Given an abstract domain A, a domain refinement operator R yields an abstract domain $R(A)$ which is more precise than A, i.e., which contains more points than A [15, 19]. A classical domain refinement operator is the *reduced product* $A \sqcap B$ of two domains A and B, both contained in another domain C. It is isomorphic to the Cartesian product of A and B, modulo the equivalence relation $\langle a_1, b_1 \rangle \equiv \langle a_2, b_2 \rangle$ if and only if $a_1 \sqcap b_1 = a_2 \sqcap b_2$. This means that pairs having the same *meaning* are identified. *Linear refinement* [20] is a slight generalisation of Cousot's reduced power operation [12]. For our purposes, we consider only its instantiation to the case of downward closed sets of substitutions. Given $a, b \in \wp{\downarrow}(\Theta_V)$ we define

$$\begin{aligned} a \multimap b &= \bigcup \{ d \in \wp{\downarrow}(\Theta_V) \mid a \cap d \subseteq b \} \\ &= \{ \theta \in \Theta_V \mid \text{for all } \sigma \leq \theta \text{ if } \sigma \in a \text{ then } \sigma \in b \} . \end{aligned} \tag{1}$$

The set $a \multimap b$ contains exactly those substitutions which, when unified with a substitution in a, become a substitution in b. If a and b are sets of substitutions satisfying some type property, you can view $a \multimap b$ as the set of substitutions which transform the property a into the property b upon unification.

Example 12. Given $v \in V$, let $\mathbf{v} = \{ \theta \in \Theta_V \mid \mathsf{vars}(\theta(v)) = \emptyset \}$. The set $\mathbf{v}$ is the set of substitutions where the variable v is bound to a ground term. We have $\mathbf{v} \in \wp{\downarrow}(\Theta_V)$. Given $x, y \in V$, Equation (1) says that

$$\mathbf{x} \multimap \mathbf{y} = \{ \theta \in \Theta_V \mid \text{for all } \sigma \leq \theta \text{ if } \mathsf{vars}(\sigma(x)) = \emptyset \text{ then } \mathsf{vars}(\sigma(y)) = \emptyset \} .$$

This means that every $\theta \in \mathbf{x} \multimap \mathbf{y}$ is such that in all its instantiations if x is ground then y is ground. Equivalently, you can say that when we unify θ with $\theta' \in \Theta_V$, the fact that x is ground in θ' entails that y is ground in $\theta\theta'$.

That is, θ transforms the groundness of x in θ' into the groundness of y in $\theta\theta'$. For instance, we have $\{x \mapsto \mathtt{f}(y)\} \in \mathbf{x} \multimap \mathbf{y}$ and $\{y \mapsto \mathtt{f}(\mathtt{a})\} \in \mathbf{x} \multimap \mathbf{y}$. But $\{x \mapsto \mathtt{f}(\mathtt{a})\} \notin \mathbf{x} \multimap \mathbf{y}$. Note that $\mathbf{x} \multimap \mathbf{x} = \Theta_V$, since groundness cannot be lost. The same holds for every downward closed property and, therefore, can be considered as a general result as far as we are concerned in this paper.

Given a Moore family $L \subseteq \wp{\downarrow}(\Theta_V)$ we define

$$L \triangleright L = \bigwedge \{a \multimap b \mid a, b \in L\} \ , \tag{2}$$

The linear refinement of L is the domain

$$L \multimap L = L \sqcap (L \triangleright L) \ , \tag{3}$$

which can be simplified into

$$L \multimap L = L \triangleright L \tag{4}$$

if $L \subseteq L \triangleright L$, as it will always be the case in this paper. This case is relevant since it allows a simpler representation and simpler operations for $L \multimap L$. Note that we overload $\multimap$. It is defined, in (1), as $a \multimap b$ where $a, b \in \wp{\downarrow}(\Theta_V)$ and, in (3), as $L \multimap L$ over the single domain $L \subseteq \wp{\downarrow}(\Theta_V)$.

The set $L \triangleright L$ is then the collection of all possible intersections of arrows which can be built in the *language* which uses the properties in L. The condition $L \subseteq L \triangleright L$ means then that the properties in L are (degenerate) cases of intersections of arrows.

Example 13. Using the notation of Example 12, the set $\{\mathbf{v} \mid v \in V\}$ is able to express basic facts about the groundness of single variables. The domain $G = \bigwedge\{\mathbf{v} \mid v \in V\}$ is able to express basic facts about the groundness of sets of variables. For instance, $\mathbf{x} \cap \mathbf{y}$ (traditionally written as $\mathbf{xy}$) is the set of substitutions where both x and y are ground, for every $x, y \in V$. It has been proved [44] that the traditional domains *Def* and *Pos* for groundness analysis [2, 10, 11] can be derived from G through linear refinement. Namely, we have $\mathit{Def} = G \multimap G$ and $\mathit{Pos} = \mathit{Def} \multimap \mathit{Def}$. Moreover, *Pos* cannot be further linearly refined, i.e., $\mathit{Pos} \multimap \mathit{Pos} = \mathit{Pos}$. We have $G \subseteq G \triangleright G$. In particular, given $\{x_1, \ldots, x_n\} \subseteq V$, it can be shown that $\mathbf{x_1} \cdots \mathbf{x_n} = \Theta_V \multimap (\mathbf{x_1} \cdots \mathbf{x_n})$, where $\Theta_V = \cap\{\} \in G$. This means that the variables $x_1, \ldots, x_n$ are ground in $\theta \in \Theta_V$ if and only if for every $\theta' \in \Theta_V$ they are ground in $\theta\theta'$.

2.5 Analysis

In this subsection we show that the problem of the analysis of a program can be reduced to that of the definition of an abstract interpretation of the concrete constraint system used by the program. Namely, if the concrete constraint system is abstracted by an abstract constraint system, the concrete semantics of the program is approximated by its abstract semantics [13].

Formally, let $C = \{C_V\}_{V \in \wp_f(\mathcal{V})}$ and $A = \{A_V\}_{V \in \wp_f(\mathcal{V})}$ be two constraint systems such that, for every $V \in \wp_f(\mathcal{V})$, C_V is partially ordered w.r.t. $\leq$, A_V is partially ordered w.r.t. $\preceq$ and there is a Galois connection $\langle \alpha_V, \gamma_V \rangle$ between C_V and A_V such that all the operators of Definition 1 in A are correct w.r.t. their counterpart in C. The Galois connection $\langle \alpha_V, \gamma_V \rangle$ can be extended into a

Galois connection between the collecting domains $(\wp(C_V), \subseteq)$ and $(\wp(A_V), \sqsubseteq)$ by defining $\alpha_V(P) = \{\alpha_V(c) \mid c \in P\}$ for all $P \in \wp(C_V)$. The $\sqsubseteq$ partial ordering is defined as $Q_1 \sqsubseteq Q_2$ if and only if for every $a_1 \in Q_1$ there exists $a_2 \in Q_2$ such that $a_1 \preceq a_2$ and vice versa, for all $Q_1, Q_2 \in \wp_f(A_V)$. Suppose that $\mathcal{S}_P^{A_V}$ is defined as the abstract counterpart to the computed answer semantics $\mathcal{S}_P^{D_V}$ of Definition 8 where $\mathcal{CA}^{A_V}[\![]\!]$ is defined as in Definition 5 but where D_V is replaced by A_V and

$$\mathcal{CA}^{A_V}[\![c]\!]I = \{\alpha_V(c)\} \ . \tag{5}$$

Since the immediate consequence operator of Definition 6 works on set-theoretic complete lattices and is additive (Proposition 7), by the general theory of fixpoint and abstract interpretation [13] we conclude that, for every $P \in \mathbb{P}^{C_V}$, we have

$$\alpha_V(\mathcal{S}_P^{C_V}) \sqsubseteq \mathcal{S}_P^{A_V} \ ,$$

that is, the abstract semantics is a correct approximation of the concrete semantics. Moreover, whenever $T_P^{A_V}$ is precise w.r.t. $T_P^{C_V}$, we have the stronger result

$$\alpha_V(\mathcal{S}_P^{C_V}) = \mathcal{S}_P^{A_V} \ ,$$

which says that the abstract semantics is *exactly* the abstraction of the concrete semantics.

Note that the precision of $T_P^{A_V}$ w.r.t. $T_P^{C_V}$ is entailed by the precision of the sub-operations used in its definition (Definitions 6 and 5).

2.6 Abstract compilation

When, as in Definition 8, the fixpoint computation is based on a compositional definition, like that of Definition 5, the $(i+1)$-th iteration can re-use any intermediate results already computed during the i-th iteration that are known not to change. Such results are usually the denotations of some program parts which do not contain recursive procedure calls. Therefore, these parts can be replaced by their denotation and the fixpoint computed on this modified (partially *compiled*) program. This technique is traditionally known as *abstract compilation* [8, 23], since it is an application of abstract interpretation where a program is iteratively compiled to its abstract denotation. This leads, in general, to a more efficient computation of the abstract fixpoint.

As a simple example of abstract compilation, we define below a partially compiled program, where only its constraints have been *compiled.*

Definition 14. Let C and A be as in Subsection 2.5. Given a program $P \in \mathbb{P}^{C_V}$, we define $\alpha_V(P) \in \mathbb{P}^{A_V}$ as the program obtained from P by substituting the constraints in the clauses with their abstraction through α_V.

The semantics of $\alpha_V(P)$ is then computed as the least fixpoint of $T_{\alpha_V(P)}^{A_V}$, where Definition 5 is used, without the change introduced with Equation (5) since the constraints considered by that equation have already been compiled in $\alpha_V(P)$. It can be proved by induction that

$$\mathcal{S}_P^{A_V} = \mathcal{S}_{\alpha_V(P)}^{A_V} \ .$$

On the left hand side, Equation (5) is used. On the right hand side, instead, the original Definition 5 is used, since the constraints considered by Equation (5) have been compiled in $\alpha_V(P)$.

More aggressive abstract compilation techniques than that of Definition 14 can be implemented. In particular, by using the call graph of a program it is possible to compile the calls to procedures which have been already analysed.

Note that the hypothesis about the compositionality of the semantics is actually a hypothesis about the (concrete and abstract) domains. Indeed, a compositional definition like that of Definition 5 is meaningful if the operations of Definition 1 are defined on the domains, i.e., if the domains are constraint systems.

3 The domains *Down* and *Sol*

In this section we define a domain of downward closed sets of substitutions. Given a substitution θ, its downward closure represents the set of substitutions which are *consistent* with θ, i.e., which can be derived from θ with computation. For instance, if in a program point we have $y = \mathtt{f}(x)$, then it is possible that, as computation proceeds, we have $y = \mathtt{f}(\mathtt{g}(w))$ and $x = \mathtt{g}(w)$. It is not possible, however, that $y = \mathtt{g}(w)$. With this interpretation in mind, we can say that a downward closed set of substitutions contains exactly all substitutions which are consistent with the prosecution of computation. For instance, if S_1 is the (downward closed) set of substitutions which are consistent with a procedure call p_1 and S_2 is the (downward closed) set of substitutions which are consistent with a procedure call p_2, then $S_1 \cap S_2$ is the (downward closed) set of substitutions which are consistent with the calls p_1, p_2 and p_2, p_1.

Definition 15. We define the family of sets $Down = \{Down_V\}_{V \in \wp_f(\mathcal{V})}$ where $Down_V = \wp\!\downarrow\!(\Theta_V)$ is a complete lattice. Let $S, S_1, S_2 \in Down_V$, $x \in V$ and $\tilde{x}, \tilde{y} \in \mathrm{Seq}(V)$. We define

$$
\begin{aligned}
S_1 \star^{Down_V} S_2 &= S_1 \cap S_2 \\
\exists_x^{Down_V} S &= \left\{ \theta' \in \Theta_V \;\middle|\; \begin{array}{l} \exists n \in \mathcal{V} \setminus V,\ \sigma \in S[n/x],\ \theta \in \Theta_{V \cup n, V} \\ \text{such that } \theta \leq_{V \cup n} \sigma \text{ and } \theta' = \theta|_V \end{array} \right\} \\
\mathsf{rename}_{\tilde{x} \to \tilde{y}}^{Down_V} S &= \{ s[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}] \mid s \in S \} \ .
\end{aligned}
$$

While the definition of conjunction is the classical one for the case of downward closed sets of substitution and is justified by the above considerations, it turns out that an *explicit* definition of cylindrification on downward closed sets of substitutions was never given.

Example 16. Let $V = \{v, x, y, z\}$ and

$$\theta_1 = \{x \mapsto \mathtt{f}(y)\}, \qquad \theta_2 = \{x \mapsto \mathtt{f}(z)\}.$$

Suppose

$$
\begin{aligned}
&\sigma_1 = \{x \mapsto \mathtt{f}(\mathtt{h}(v)), y \mapsto \mathtt{h}(v)\}, \quad \sigma_2 = \{x \mapsto \mathtt{f}(\mathtt{h}(v)), z \mapsto \mathtt{h}(v)\}, \\
&\sigma_3 = \{x \mapsto \mathtt{f}(\mathtt{h}(v)), y \mapsto \mathtt{h}(v), z \mapsto \mathtt{h}(v)\}, \quad \sigma_4 = \{x \mapsto \mathtt{f}(\mathtt{h}(v))\}.
\end{aligned}
$$

Then

$$\sigma_1, \sigma_3 \in \downarrow\theta_1, \quad \sigma_2, \sigma_4 \notin \downarrow\theta_1, \quad \sigma_2, \sigma_3 \in \downarrow\theta_2, \quad \sigma_1, \sigma_4 \notin \downarrow\theta_2 \ .$$

Therefore $\sigma_1, \sigma_2, \sigma_4 \notin \downarrow\theta_1 \star^{Down_V} \downarrow\theta_2$, and $\sigma_3 \in \downarrow\theta_1 \star^{Down_V} \downarrow\theta_2$.

It follows from Definition 15 that $\theta_1 \in \exists_x^{Down_V}\downarrow\theta_1$. To see this choose any $n \in \mathcal{V} \setminus V$ and let $\sigma = \{n \mapsto \mathtt{f}(y)\} = \theta_1[n/x]$. Then $\sigma \in (\downarrow\theta_1)[n/x]$. Therefore, if $\theta = \{n \mapsto \mathtt{f}(y), x \mapsto \mathtt{f}(y)\}$, we have $\theta = \sigma \circ \{x \mapsto \mathtt{f}(y)\}$ and $\theta|_V = \theta_1$. Note that we have $\varepsilon \in \exists_x^{Down_V}\downarrow\theta_1$ since, in Definition 15 we can choose σ as above and $\theta = \sigma$. We have $\theta \leq_{V \cup n} \sigma$ and $\theta|_V = \varepsilon$.

It follows from Definition 15 that $\{v \mapsto \mathtt{f}(y)\} \in \mathsf{rename}_{\langle x\rangle \to \langle v\rangle}^{Down_V}(\downarrow\theta_1)$. Moreover, as $\{x \mapsto \mathtt{f}(y), v \mapsto \mathtt{a}\} \in \downarrow\theta_1$, and $\{x \mapsto \mathtt{f}(y), v \mapsto \mathtt{a}\}[v/x, x/v] = \{v \mapsto \mathtt{f}(y), x \mapsto \mathtt{a}\}$, we have $\{v \mapsto \mathtt{f}(y), x \mapsto \mathtt{a}\} \in \mathsf{rename}_{\langle x\rangle \to \langle v\rangle}^{Down_V}(\downarrow\theta_1)$.

The operations of Definition 15 are closed on $Down_V$. Moreover, they satisfy some interesting properties.

Proposition 17. *Let $V \in \wp_f(\mathcal{V})$ and $S, S_1, S_2 \subseteq \Theta_V$.*

i) If $S_1 = \downarrow S_1$ and $S_2 = \downarrow S_2$ then $S_1 \star^{Down_V} S_2 = \downarrow(S_1 \star^{Down_V} S_2)$.

ii) If $S = \downarrow S$ then $\exists_x^{Down_V} S = \downarrow(\exists_x^{Down_V} S)$.

iii) If $S_1 \subseteq S_2$, then $\exists_x^{Down_V} S_1 \subseteq \exists_x^{Down_V} S_2$ (cylindrification is monotonic).

iv) $S \subseteq \exists_x^{Down_V}(S)$ (cylindrification is extensive).

v) If $S = \downarrow S$ then $\mathsf{rename}_{\tilde{x}\to\tilde{y}}^{Down_V}(S) = \downarrow(\mathsf{rename}_{\tilde{x}\to\tilde{y}}^{Down_V}(S))$.

An existential Herbrand constraint $h \in H_V$ can be mapped into a downward closed set of substitutions through the map sol_V which yields the set of its solutions. However, this map is not onto.

Proposition 18. *If Σ contains at least a constant and a functor symbol, then $\{\mathsf{sol}_V(h) \mid h \in H_V\} \subset \wp\downarrow(\Theta_V)$.*

In spite of this result, we can show that $\star^{Down_V}$, rename^{Down_V} and $\exists^{Down_V}$ are closed on the set $\mathsf{sol}_V(H_V)$.

Proposition 19. *Let $V \in \wp_f(\mathcal{V})$ and $h, h_1, h_2 \in H_V$.*

i) $\mathsf{sol}_V(h_1) \star^{Down_V} \mathsf{sol}_V(h_2) = \mathsf{sol}_V(h_1 \star^{H_V} h_2)$,

ii) $\exists_x^{Down_V} \mathsf{sol}_V(h) = \mathsf{sol}_V\left(\exists_x^{H_V} h\right)$ for every $x \in V$,

iii) if $h \in H_{V\setminus\tilde{y}}$, then $\mathsf{rename}_{\tilde{x}\to\tilde{y}}^{Down_V} \mathsf{sol}_V(h) = \mathsf{sol}_V\left(\mathsf{rename}_{\tilde{x}\to\tilde{y}}^{H_V} h\right)$ for every $\tilde{x}, \tilde{y} \in \mathrm{Seq}(V)$.

Example 20. Let V and θ_1 be as in Example 16. By Proposition 9 we have $\exists_x^{Down_V}\downarrow\theta_1 = \exists_x^{Down_V} \mathsf{sol}_V(\theta_1)$. Thus, by Proposition 19.ii, we have $\exists_x^{Down_V}\downarrow\theta_1 = \mathsf{sol}_V(\exists_x^{H_V}\theta_1) = \mathsf{sol}_V(\varepsilon) = \Theta_V$ which entails the result $\{\theta_1, \varepsilon\} \subseteq \exists_x^{Down_V}\downarrow\theta_1$ shown in Example 16. Similarly, by Proposition 19.iii we have

$$\begin{aligned}\mathsf{rename}_{\langle x\rangle\to\langle v\rangle}^{Down_V} \downarrow\theta_1 &= \mathsf{rename}_{\langle x\rangle\to\langle v\rangle}^{Down_V} \mathsf{sol}_V(\theta_1) = \mathsf{sol}_V(\mathsf{rename}_{\langle x\rangle\to\langle v\rangle}^{H_V} \theta_1) \\ &= \mathsf{sol}_V(\{v \mapsto \mathtt{f}(y)\}) = \downarrow\{v \mapsto \mathtt{f}(y)\} \ .\end{aligned}$$

This entails the result $\{\{v \mapsto \mathtt{f}(y)\}, \{v \mapsto \mathtt{f}(y), x \mapsto \mathtt{a}\}\} \subseteq \mathsf{rename}_{\langle x\rangle\to\langle v\rangle}^{Down_V} \downarrow\theta_1$, also shown in Example 16.

Thanks to Proposition 19, we can introduce the following definition.

Definition 21. We define $Sol = \{Sol_V\}_{V \in \wp_f(\mathcal{V})}$, where $Sol_V = \{\mathsf{sol}_V(h) \mid h \in H_V\}$. The $\star^{Sol_V}$, rename^{Sol_V} and $\exists^{Sol_V}$ operators on Sol_V are the restriction on Sol_V of the corresponding operators of $Down_V$.

Since sol_V is one-to-one and onto from H_V into Sol_V, and for every $\{h_1, h_2\} \subseteq H_V$ we have $h_1 \le h_2$ if and only if $\mathsf{sol}_V(h_1) \subseteq \mathsf{sol}_V(h_2)$, we conclude that sol_V, endowed with the $\subseteq$ partial ordering, is isomorphic to H_V, and we know that the corresponding operations coincide (Proposition 19). The usefulness of Sol is that of presenting an existential Herbrand constraint as the set of its solutions. This will be very important in the next section.

3.1 The collecting semantics

Now we know that existential Herbrand constraints are essentially the same as their set of solutions. This isomorphism can be lifted to an isomorphism between $\wp(H)$ and $\wp(Sol)$, the domains used for the collecting semantics. The domain $\wp(Sol)$ is not exactly what we are looking for. Indeed, we want to be able to represent every downward closed set of substitutions rather than every set of sets of solutions. This means that we want to use $Down_V$ to represent our abstract properties. We prove here that there is a Galois insertion from $\wp(Sol_V)$ into $Down_V$ for every $V \in \wp_f(\mathcal{V})$.

We use $\cup$ as abstraction map from $\wp(Sol_V)$ into $Down_V$. The map $\cup$ is onto from $\wp(Sol_V)$ into $Down_V$. This is because every $d \in Down_V$ can be written as the infinite union $d = \cup\{\downarrow\{\theta\} \mid \theta \in d\}$ and every single $\downarrow\{\theta\}$ belongs to Sol_V, as shown by Proposition 9. It can be shown that $\cup$ is additive from $\wp(Sol_V)$ into $Down_V$.

Proposition 22. *Given $\{p_i\}_{i \in I} \subseteq \wp(Sol_V)$ with $I \subseteq \mathbb{N}$, we have*

$$\cup\left(\sqcup_{i \in I}^{\wp(Sol_V)} p_i\right) = \sqcup_{i \in I}^{Down_V}(\cup p_i) \ .$$

The above proposition and the fact that $\cup$ is onto entail that $\cup$ induces a Galois insertion from $\wp(Sol_V)$ into $Down_V$. Moreover, the operators on $Down_V$ are the best abstraction through $\cup$ of the corresponding operators on $\wp(Sol_V)$.

Proposition 23. *Let $\{p_1, p_2, p\} \subseteq \wp(Sol_V)$. We have*

i) $\cup\left(p_1 \star^{\wp(Sol_V)} p_2\right) = (\cup p_1) \star^{Down_V} (\cup p_2)$.

ii) $\cup\left(\exists_x^{\wp(Sol_V)} p\right) = \exists_x^{Down_V}(\cup p)$ *for every* $x \in V$.

iii) $\cup\left(\mathsf{rename}_{\tilde{x} \to \tilde{y}}^{\wp(Sol_V)}\, p\right) = \mathsf{rename}_{\tilde{x} \to \tilde{y}}^{Down_V}(\cup p)$ *for every* $\tilde{x}, \tilde{y} \in \mathrm{Seq}(V)$.

4 Type constraint systems

We have shown that the collecting semantics over existential Herbrand constraints can be abstracted into a semantics over the domain $Down$. This domain allows the representation of *every* downward closed property of logic programs. In general, the full power of $Down$ is not needed. For instance, we might be interested in *some* downward closed property, like a set of types and their dependencies. This means that we want to abstract *Down* into a domain for a specific *type system*, where a type system specifies which downward closed properties we are interested in. ¿From now on the abstraction becomes type-dependent.

4.1 Type systems and type constraint systems

A type system is a specification of the types to be used in the analysis.

Definition 24. A *type system* is $\langle \Delta, \Sigma, I \rangle$ where Δ and Σ, called *type signature* and *term signature*, respectively, are sets of function symbols with associated arities and $I(\mathtt{f}^n)$ is a total map from $(\wp{\downarrow}(\mathsf{terms}(\Sigma, \mathcal{V})))^n$ to $\wp{\downarrow}(\mathsf{terms}(\Sigma, \mathcal{V}))$ for every $\mathtt{f}^n \in \Delta$. We define

$$[\![\mathtt{f}^0]\!]I = I(\mathtt{f}^0) \qquad \text{and} \qquad [\![\mathtt{f}^n(t_1, \ldots, t_n)]\!]I = I(\mathtt{f}^n)([\![t_1]\!]I, \ldots, [\![t_n]\!]I) \ .$$

Example 25. Let $\mathsf{G} = \langle \{\mathtt{g}^0\}, \Sigma, I \rangle$ and let $I(\mathtt{g}) = \mathsf{terms}(\Sigma, \emptyset)$. The functor symbol $\mathtt{g}$ is interpreted as the set of terms which do not contain variables. Therefore, G is the type system for groundness.

Let $\mathsf{NF} = \langle \{\mathtt{nf}^0\}, \Sigma, I \rangle$ where $I(\mathtt{nf}) = \mathsf{terms}(\Sigma, \mathcal{V}) \setminus \mathcal{V}$. The functor symbol $\mathtt{nf}$ is interpreted as the set of terms which are not in $\mathcal{V}$, i.e., the set of non-free terms. Thus, NF is the type system for non-freeness.

Let $\mathsf{NL} = \langle \{\mathtt{nat}^0, \mathtt{top}^0, \mathtt{list}^1\}, \Sigma, I \rangle$, where $\{\mathsf{0}^0, \mathsf{s}^1, []^0, [|]^2\} \subseteq \Sigma$ and

$$\begin{aligned} I(\mathtt{nat}) &= \mu i.\{\mathsf{0}\} \cup \{\mathsf{s}(n) \mid n \in i\} \\ I(\mathtt{top}) &= \mathsf{terms}(\Sigma, V) \\ I(\mathtt{list}) &= \lambda\tau.\mu l.\{[]\} \cup \{[h|t] \mid h \in \tau,\ t \in l\} \ . \end{aligned}$$

Since $\mathtt{list}$ is a unary predicate, we can write types like $\mathtt{list(nat)}$ or even $\mathtt{list(list(nat))}$. For instance, it follows from Definition 24 that the meaning of $\mathtt{list(nat)}$ in the interpretation I defined above is

$$\begin{aligned} [\![\mathtt{list(nat)}]\!]I = I(\mathtt{list})([\![\mathtt{nat}]\!]I) &= \mu l.\{[]\} \cup \{[h|t] \mid h \in I(\mathtt{nat}),\ t \in l\} \\ &= \{[], [\mathsf{0}], [\mathsf{s}(\mathsf{0})], [\mathsf{0}, \mathsf{0}], [\mathsf{0}, \mathsf{s}(\mathsf{0})], [\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0})], \ldots\} \ . \end{aligned}$$

This type system can be used to express type properties involving natural numbers and polymorphic lists. If we want to consider also a type of trees, we define $\mathsf{NLT} = \langle \{\mathtt{nat}^0, \mathtt{top}^0, \mathtt{list}^1, \mathtt{tree}^1\}, \Sigma, I \rangle$, where $\{\mathsf{0}^0, \mathsf{s}^1, []^0, [|]^2, \mathtt{void}^0, \mathtt{tree}^3\} \subseteq \Sigma$, $I(\mathtt{nat})$, $I(\mathtt{top})$ and $I(\mathtt{list})$ are defined as above and

$$I(\mathtt{tree}) = \lambda\tau.\mu t.\{\mathtt{void}\} \cup \{\mathtt{tree}(n, l, r) \mid n \in \tau,\ l \in t \text{ and } r \in t\} \ .$$

Since $\mathtt{tree}$ is a unary predicate, we can write a type like $\mathtt{tree(list(nat))}$.

Given a type system, we can define a type constraint system. It is formed by transfinite propositional formulas, defined below.

Definition 26. Given a type signature Δ and a set of variables V, the set $\Phi_{\Delta,V}$ of *transfinite formulas* over Δ and V is defined as the least set such that $(x \in t) \in \Phi_{\Delta,V}$, where $x \in V$ and $t \in \mathsf{terms}(\Delta, \emptyset)$, if $S \in \wp(\Phi_{\Delta,V})$ then $(\wedge S) \in \Phi_{\Delta,V}$ and $(\vee S) \in \Phi_{\Delta,V}$, if $\phi_1, \phi_2 \in \Phi_{\Delta,V}$ then $(\phi_1 \Rightarrow \phi_2) \in \Phi_{\Delta,V}$ and if $\phi \in \Phi_{\Delta,V}$ then $(\neg\phi) \in \Phi_{\Delta,V}$. We write $\phi_1 \wedge \phi_2$ for $\wedge\{\phi_1, \phi_2\}$, $\phi_1 \vee \phi_2$ for $\vee\{\phi_1, \phi_2\}$, *true* for $\wedge\emptyset$ and *false* for $\vee\emptyset$.

Example 27. Consider $\Delta = \{\mathtt{nat}^0, \mathtt{list}^1\}$ and $V = \{x, y, z\}$. Examples of transfinite formulas in $\Phi_{\Delta,V}$ are $x \in \mathtt{nat}$, $x \in \mathtt{nat} \wedge y \in \mathtt{list(nat)}$ and $z \in \mathtt{nat} \wedge z \in \mathtt{list(nat)}$. Note that the last one has the intuitive meaning of an unsatisfiable formula. However, we have not defined any notion of *semantics* for transfinite formulas yet. We will do it in Definition 30. Moreover, note that all those transfinite formulas are actually finite formulas. As an example of a strictly transfinite formula, consider

$$x \in \mathtt{list(nat)} \wedge x \in \mathtt{list(list(nat))} \wedge x \in \mathtt{list(list(list(nat)))} \wedge \cdots$$

Once the set of formulas $\Phi_{\Delta,V}$ of Definition 26 is endowed with the operations of Definition 1, it becomes a constraint system.

Definition 28. Let $\mathsf{T} = \langle \Delta, \Sigma, I \rangle$ be a type system. A *type constraint system* for T is $T = \{T_V\}_{V \in \wp_f(\mathcal{V})}$, where $T_V \subseteq \Phi_{\Delta,V}$ is closed w.r.t. the operations

$$\phi_1 \star^{T_V} \phi_2 = \phi_1 \wedge \phi_2$$

$$\mathsf{rename}^{T_V}_{\tilde{x}\to\tilde{y}}\, \phi = \begin{cases} \phi[\tilde{y}/\tilde{x}] & \text{if } \phi \in \Phi_{\Delta,V\setminus\tilde{y}} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\exists^{T_V}_x \phi = \vee \left\{ \phi[P/x] \;\middle|\; \begin{array}{l} \text{there exists } t \in \mathsf{terms}(\Sigma, V) \text{ such that} \\ P = \{d \in \mathsf{terms}(\Delta, \emptyset) \mid t \in [\![d]\!] I\} \end{array} \right\},$$

where, for all $x, y \in V$,

$$\begin{aligned} (x \in t)[y/x] &= y \in t & (z \in t)[y/x] &= (z \in t) \quad \text{if } z \neq x \\ (\phi_1 \wedge \phi_2)[y/x] &= (\phi_1[y/x]) \wedge (\phi_2[y/x]) \\ (\phi_1 \vee \phi_2)[y/x] &= (\phi_1[y/x]) \vee (\phi_2[y/x]) & (\neg\phi)[y/x] &= \neg(\phi[y/x]) \\ (\phi_1 \Rightarrow \phi_2)[y/x] &= (\phi_1[y/x]) \Rightarrow (\phi_2[y/x]) \end{aligned}$$

and, for all $P \in \wp(\mathsf{terms}(\Delta, \emptyset))$,

$$\begin{aligned} (x \in t)[P/x] &= \begin{cases} \mathit{true} & \text{if } \cap_{p \in P} [\![p]\!] I \subseteq [\![t]\!] I \\ \mathit{false} & \text{otherwise} \end{cases} & (y \in t)[P/x] &= (y \in t) \\ & & &\quad \text{if } y \neq x \\ (\phi_1 \wedge \phi_2)[P/x] &= (\phi_1[P/x]) \wedge (\phi_2[P/x]) \\ (\phi_1 \vee \phi_2)[P/x] &= (\phi_1[P/x]) \vee (\phi_2[P/x]) & (\neg\phi)[P/x] &= \neg(\phi[P/x]) \\ (\phi_1 \Rightarrow \phi_2)[P/x] &= (\phi_1[P/x]) \Rightarrow (\phi_2[P/x]) . \end{aligned}$$

Note that the simplest way, in Definition 28, for choosing $T_V \subseteq \Phi_{\Delta,V}$ in such a way that it is closed w.r.t. the three operations of that definition, is by letting $T_V = \Phi_{\Delta,V}$. However, we will see that this choice sometimes leads to redundancy in the type constraint system (Example 35).

Note that if $\mathsf{terms}(\Delta, \emptyset)$ is finite then the set $\Phi_{\Delta,V}$ is finite too, and the resulting type constraint systems are finite. Moreover, the conjunction and the cylindrification operations become effective, while they are not computable in the general case. Finally, note that, for the type systems G and NF of Example 25, cylindrification becomes the classical Schröder elimination [2], since $\mathsf{terms}(\Delta, \emptyset)$ contains only a constant symbol whose evaluation is not empty[4].

[4] We recall that the Schröder elimination of x from a propositional formula ϕ is $\phi[\mathit{false}/x] \vee \phi[\mathit{true}/x]$.

Example 29. Consider the type system $\mathsf{NL} = \langle\{\mathtt{nat}, \mathtt{top}, \mathtt{list}\}, \Sigma, I\rangle$ of Example 25 and let $NL_V = \Phi_{\{\mathtt{nat},\mathtt{top},\mathtt{list}\},V}$ for every $V \in \wp_f(\mathcal{V})$. As examples of operations on that type constraint system, let $V = \{x, y, z\}$. Then, by Definition 28,

$$(x \in \mathtt{nat}) \star^{NL_V} (y \in \mathtt{list}(\mathtt{nat})) = x \in \mathtt{nat} \wedge y \in \mathtt{list}(\mathtt{nat})$$
$$\mathsf{rename}^{NL_V}_{\langle x\rangle \to \langle y\rangle}(x \in \mathtt{list}(\mathtt{nat})) = y \in \mathtt{list}(\mathtt{nat}) \ .$$

In the cylindrification operation, we need first to compute which are the possible sets P of types such that there exists a term t which belongs *exactly* to those types. Every term belongs to $\mathtt{top}$ and there are terms which belongs only to $\mathtt{top}$, for instance, $\mathtt{s}([])$. A term can be just a natural number, for instance $\mathtt{0}$ is a natural number and is not a list. A term can be a list but not a natural number, nor a list of natural numbers, nor a list of lists of natural numbers, and so on. As an example, consider $[[], \mathtt{0}]$. A term can be a list of natural numbers, and *then* even a list of $\mathtt{top}$, but not a list of lists of natural numbers, nor a list of lists of lists of natural numbers and so on. As an example, we have $[\mathtt{0}]$. We can continue this way indefinitely. In conclusion, we have infinite possibilities for P, collected in the set S below:

$$\begin{aligned} S = \{&\{\mathtt{top}\}, \{\mathtt{top}, \mathtt{nat}\}, \{\mathtt{top}, \mathtt{list}(\mathtt{top})\}, \{\mathtt{top}, \mathtt{list}(\mathtt{top}), \mathtt{list}(\mathtt{nat})\},\\ &\{\mathtt{top}, \mathtt{list}(\mathtt{top}), \mathtt{list}(\mathtt{list}(\mathtt{top})), \mathtt{list}(\mathtt{list}(\mathtt{nat}))\}, \ldots\} \ . \end{aligned}$$

As a consequence, we have

$$\begin{aligned} \exists_x^{NL_V}(x \in \mathtt{list}(\mathtt{nat}) \wedge y \in \mathtt{nat}) &= \bigvee_{P \in S} (x \in \mathtt{list}(\mathtt{nat}) \wedge y \in \mathtt{nat})[P/x] \\ &= \bigvee_{P \in S} (x \in \mathtt{list}(\mathtt{nat})[P/x] \wedge y \in \mathtt{nat}) \ . \end{aligned}$$

Since only the set $P' = \{\mathtt{top}, \mathtt{list}(\mathtt{top}), \mathtt{list}(\mathtt{nat})\}$ listed in the set S above is such that $\cap_{p \in P'} [\![p]\!] \subseteq [\![\mathtt{list}(\mathtt{nat})]\!]$, we can rewrite the equation above into

$$\left(\bigvee_{P \in S,\ P \neq P'} (\mathit{false} \wedge y \in \mathtt{nat}) \right) \vee (\mathit{true} \wedge y \in \mathtt{nat}) \ . \tag{6}$$

We are obviously tempted to reduce Equation (6) to $y \in \mathtt{nat}$. In order to do so, we first need to define a notion of equivalence over transfinite formulas.

Definition 30. Given a type system $\mathsf{T} = \langle\Delta, \Sigma, I\rangle$, we define the map $[\![]\!]_{\mathsf{T}} : \Phi_{\Delta,V} \to (\Theta_V \to \{0, 1\})$ as

$$[\![x \in t]\!]_{\mathsf{T}}\sigma = \begin{cases} 1 & \text{if } \sigma(x) \in [\![t]\!]I \\ 0 & \text{otherwise} \end{cases} \qquad [\![\wedge S]\!]_{\mathsf{T}}\sigma = \begin{cases} 1 & \begin{array}{l}\text{if } [\![\phi]\!]_{\mathsf{T}}\sigma = 1 \\ \text{for all } \phi \in S\end{array} \\ 0 & \text{otherwise} \end{cases}$$

$$[\![\vee S]\!]_{\mathsf{T}}\sigma = \begin{cases} 1 & \begin{array}{l}\text{if } [\![\phi]\!]_{\mathsf{T}}\sigma = 1 \\ \text{for some } \phi \in S\end{array} \\ 0 & \text{otherwise} \end{cases} \qquad [\![\phi_1 \Rightarrow \phi_2]\!]_{\mathsf{T}}\sigma = \begin{cases} 1 & \begin{array}{l}\text{if } [\![\phi_1]\!]_{\mathsf{T}}\sigma = 0 \\ \text{or } [\![\phi_2]\!]_{\mathsf{T}}\sigma = 1\end{array} \\ 0 & \text{otherwise} \end{cases}$$

$$[\![\neg\phi]\!]_{\mathsf{T}}\sigma = 1 - [\![\phi]\!]_{\mathsf{T}}\sigma \ .$$

When it is clear from the context, we write $[\![]\!]$ for $[\![]\!]_{\mathsf{T}}$. Given $V \in \wp_f(\mathcal{V})$ and $\phi_1, \phi_2 \in \Phi_{\Delta,V}$, we define $\phi_1 \leq_{\mathsf{T},V} \phi_2$ if and only if for every $\theta \in \Theta_V$ we have that $[\![\phi_1]\!]_{\mathsf{T}}\theta = 1$ entails $[\![\phi_2]\!]_{\mathsf{T}}\theta = 1$. When it is clear from the context, we write $\leq$ for $\leq_{\mathsf{T},V}$. We define $\phi_1 \equiv_{\mathsf{T},V} \phi_2$ if and only if $\phi_1 \leq_{\mathsf{T},V} \phi_2$ and $\phi_2 \leq_{\mathsf{T},V} \phi_1$. This equivalence relation is called (T, V)-*equivalence*. Again, we drop the subscripts T and V when it does not lead to confusion.

Example 31. In the hypotheses of Example 29, Equation (6) can be shown to be (NL, V)-equivalent to $y \in \mathtt{nat}$. Indeed, for every transfinite formula ϕ we have $false \wedge \phi \equiv false$, $false \vee false = false$, $true \wedge \phi \equiv \phi$ and $false \vee \phi = \phi$.

In the rest of the paper, if $T = \{T_V\}_{V \in \wp_f(\mathcal{V})}$ is a type constraint system and $V \in \wp_f(\mathcal{V})$, every transfinite formula in T_V will stand for its (T, V)-equivalence class. Note that T_V is a complete lattice w.r.t. $\leq_{\mathsf{T},V}$, since by Definition 28 it is completely $\wedge$-closed and topped (since $true = \wedge\emptyset$). We require that the operations $\star^{T_V}$, $\exists^{T_V}$ and rename^{T_V} of Definition 28 are independent from the representatives chosen for the (T, V)-equivalence classes. This follows directly from Definition 28, for conjunction. We can also show that it is true for renaming.

Proposition 32. *Let $T = \{T_V\}_{V \in \wp_f(\mathcal{V})}$ be a type constraint system for the type system $\langle \Delta, \Sigma, I \rangle$. For every $V \in \wp_f(\mathcal{V})$, $x, y \in V$ and $\{\phi_1, \phi_2\} \subseteq T_{V \setminus y}$, if $\phi_1 \equiv \phi_2$ then $\phi_1[y/x] \equiv \phi_2[y/x]$.*

For cylindrification, it is true if the following property holds.

P1: Let $T = \{T_V\}_{V \in \wp_f(\mathcal{V})}$ be a type constraint system for $\langle \Delta, \Sigma, I \rangle$. For every $V \in \wp_f(\mathcal{V})$, $x \in V$ and $\{\phi_1, \phi_2\} \subseteq T_V$, if $\phi_1 \equiv \phi_2$ then $\phi_1[P/x] \equiv \phi_2[P/x]$ for all $P = \{d \in \mathsf{terms}(\Delta, \emptyset) \mid t \in [\![d]\!]I\}$ with $t \in \mathsf{terms}(\Sigma, V)$.

However, property P1 does not always hold.

Example 33. Let $x \in \mathcal{V}$. Consider the type constraint system defined as $F = \{\Phi_{\{\mathsf{g},\mathsf{gx}\},V}\}_{V \in \wp_f(\mathcal{V})}$ for the type system $\mathsf{F} = \langle \{\mathtt{g}, \mathtt{gx}\}, \Sigma, I \rangle$, where $\{\mathtt{a}^0, \mathtt{f}^1\} \subseteq \Sigma$ and

$$I(\mathtt{g}) = \mathsf{terms}(\Sigma, \emptyset) \ , \qquad I(\mathtt{gx}) = \mathsf{terms}(\Sigma, \{x\}) \setminus \{x\} \ .$$

This type system has two types, i.e., the type $\mathtt{g}$ of ground terms and the type $\mathtt{gx}$ of terms different from x and whose only allowed variable is x. Note that $I(\mathtt{gx})$ is downward closed. Assume $V \supseteq \{x\}$ and consider

$$\phi_1 = (x \in \mathtt{gx} \Rightarrow x \in \mathtt{g}) \ , \qquad \phi_2 = true \ .$$

Those formulas are equivalent. Indeed, given $\sigma \in \Theta_V$, assume $[\![x \in \mathtt{gx}]\!]_{\mathsf{F}}\sigma = 1$. Then $\sigma(x) \in [\![\mathtt{gx}]\!]I$, i.e., $\sigma(x) \in I(\mathtt{gx})$, i.e., $\sigma(x) \in \mathsf{terms}(\Sigma, \{x\}) \setminus \{x\}$. But since σ is idempotent, this is true if and only if $\sigma(x) \in \mathsf{terms}(\Sigma, \emptyset)$, i.e., $\sigma(x) \in I(\mathtt{g})$. Finally, this holds if and only if $[\![x \in \mathtt{g}]\!]_{\mathsf{F}}\sigma = 1$. Since σ is arbitrary, we conclude that $\phi_1 \equiv true$.

Consider now $P = \{\mathtt{gx}\} = \{d \in \mathsf{terms}(\Delta, \emptyset) \mid \mathtt{f}(x) \in [\![d]\!]I\}$. Since $[\![\mathtt{gx}]\!]I \not\subseteq [\![\mathtt{g}]\!]I$ (for the choice of V) we have

$$\phi_1[P/x] = (true \Rightarrow false) \equiv false \ , \qquad \phi_2[P/x] = true \ ,$$

which are not equivalent.

4.2 The semantics of a type constraint system

For all $V \in \wp_f(\mathcal{V})$, a Moore family of $Down_V$ (i.e., an abstract interpretation of $Down_V$) can be defined once a type constraint system is given.

Definition 34. Given a type constraint system $T = \{T_V\}_{V \in \wp_f(\mathcal{V})}$ for the type system T and $\phi \in T_V$, we define

$$\gamma_{T_V}(\phi) = \{\theta \in \Theta_V \mid \text{for all } \sigma \in \Theta_V \text{ s.t. } \sigma \leq \theta \text{ we have } [\![\phi]\!]_{\mathsf{T}}\sigma = 1\} .$$

Note that $\gamma_{T_V}(\phi) \in Down_V$ for all $\phi \in T_V$. Given $\phi_1, \phi_2 \in T_V$, we define $\phi_1 \equiv_{\gamma_{T_V}} \phi_2$ if and only if $\gamma_{T_V}(\phi_1) = \gamma_{T_V}(\phi_2)$. When it is clear from the context, we drop the subscript T_V from γ_{T_V}.

It follows from Definition 34 that $\equiv$ entails $\equiv_\gamma$, though the converse does not hold.

Example 35. Consider the type system G of Example 25 and the type constraint system $\{\Phi_{\Delta,V}\}_{V \in \wp_f(\mathcal{V})}$ for G. Consider V such that $\{x, y\} \subseteq V$. Let $\phi_1 = \mathit{false}$ and $\phi_2 = x \in \mathsf{g} \Rightarrow \mathit{false}$. Note that there is no θ such that $[\![\phi_1]\!]_{\mathsf{G}}\theta = 1$. But any substitution θ such that $\theta(x)$ is not ground satisfies $[\![\phi_2]\!]_{\mathsf{G}}\theta = 1$. Hence $\phi_1 \not\equiv \phi_2$. However, $\gamma(\phi_1) = \emptyset$ and $\gamma(\phi_2) = \emptyset$ so that $\phi_1 \equiv_\gamma \phi_2$. This is obvious for ϕ_1. For ϕ_2, let $\theta \in \gamma(\phi_2)$. Any instance of θ must belong to $\gamma(\phi_2)$. But this is a contradiction as soon as we consider $\theta' \leq \theta$ which makes x ground, because in such a case it would be $[\![\mathit{false}]\!]_{\mathsf{G}}\theta' = 1$. Note that we can always find such a θ' because Σ contains at least one function symbol of arity 0 (Subsection 2.2).

Proposition 36. *Given a type constraint system $T = \{T_V\}_{V \in \wp_f(\mathcal{V})}$, γ_{T_V} is co-additive and $\gamma_{T_V}(T_V)$ is a Moore family of $Down_V$ for all $V \in \wp_f(\mathcal{V})$.*

Since γ is co-additive and $Down_V$ and T_V are complete lattices, we conclude that γ is the concretisation map of a Galois connection from $Down_V$ into T_V. Let α (i.e., α_{T_V}) be the corresponding abstraction map. Since, in general, γ is not one-to-one (Example 35), we are not guaranteed to have a Galois insertion instead of just a Galois connection. This is what the property below requires.

P2: Let $T = \{T_V\}_{V \in \wp_f(\mathcal{V})}$ be a type constraint system. For all $V \in \wp_f(\mathcal{V})$ and $\phi_1, \phi_2 \in T_V$, if $\phi_1 \equiv_\gamma \phi_2$ then $\phi_1 \equiv \phi_2$.

Example 37. In Example 35, we constructed a type constraint system in which there were formulas ϕ_1, ϕ_2 where $\phi_1 \not\equiv \phi_2$ although $\phi_1 \equiv_\gamma \phi_2$. Thus this type constraint system does not satisfy property P2 above and contains redundant elements.

4.3 Positive and structural type constraint systems

We look now for a class of type constraint systems which satisfy both properties P1 and P2.

Definition 38. Given a type system $\mathsf{T} = \langle\{\mathsf{c}^0\}, \Sigma, I\rangle$ and $V = \{x_1, \ldots, x_n\} \in \wp_f(\mathcal{V})$, $\phi \in \Phi_{\Delta,V}$ is called *positive* if, for all ground terms t such that $t \in [\![\mathsf{c}]\!]I$, we have $[\![\phi]\!]\{x_1 \mapsto t, \ldots, x_n \mapsto t\} = 1$.

Definition 39. A type constraint system $T = \{T_V\}_{V \in \wp_f(\mathcal{V})}$ for the type system $\langle \Delta, \Sigma, I \rangle$ is *positive* if Δ is formed by just one constant c^0 and for every $V \in \wp_f(\mathcal{V})$ and every $\phi \in T_V$, ϕ is positive.

A type system $\mathsf{T} = \langle \Delta, \Sigma, I \rangle$ is *structural* if, for every $V \in \wp_f(\mathcal{V})$ and finite set $\tilde{t} \subseteq \mathsf{terms}(\Sigma, V)$, there exists $\sigma \in \Theta_V^V$ such that, for every $t \in \tilde{t}$, $t\sigma$ is ground. Moreover, we require that, for every $d \in \mathsf{terms}(\Delta, \emptyset)$, we have $t \in [\![d]\!]I$ if and only if $t\sigma \in [\![d]\!]I$. A type constraint system $T = \{T_V\}_{V \in \wp_f(\mathcal{V})}$ for T is structural if T is structural.

Positive type systems generalise the traditional type system for groundness, as defined in Example 25. Every type constraint system with just one type symbol is positive, provided we restrict the set of transfinite formulas to the positive ones.

Example 40. Consider the type system G of Example 25 and the type constraint system $Pos_{\{\mathtt{g}\}} = \{Pos_{\{\mathtt{g}\},V}\}_{V \in \wp_f(\mathcal{V})}$ where $Pos_{\{\mathtt{g}\},V}$ is formed by the set of positive transfinite formulas in $\Phi_{\{\mathtt{g}\},V}$. Note that this set is finite. In [2] it is shown that $Pos_{\{\mathtt{g}\},V}$ is closed under conjunction and cylindrification. Moreover, it is obviously closed under renaming. This type constraint system is well-known [2, 10, 11].

A formula is definite if the set of its propositional models is closed under instantiation. Consider the type constraint system $Def_{\{\mathtt{g}\}} = \{Def_{\{\mathtt{g}\},V}\}_{V \in \wp_f(\mathcal{V})}$ where $Def_{\{\mathtt{g}\},V}$ is the set of (positive) definite formulas in $\Phi_{\{\mathtt{g}\},V}$. Even this type constraint system is well-known [2]. In [2] it is shown that the set of definite formulas is closed under conjunction and cylindrification. It is obviously closed under renaming.

Example 41. The construction of Example 40 can be used with non-freeness (type system NF of Example 25). The resulting type constraint systems will be called $Pos_{\{\mathtt{nf}\}}$ and $Def_{\{\mathtt{nf}\}}$. Note that they are positive type constraint systems.

In a structural type constraint system, every finite set of terms can be instantiated into a finite set of ground terms with the same type properties. This is useful in the proofs since it guarantees that for every substitution θ there exists a grounding substitution θ' with the same type properties as θ. Namely, the types of $\theta(x)$ are exactly the same as the types of $\theta'(x)$. This allows to work with grounding substitutions only.

It is not easy to apply the definition of structural type constraint system (Definition 39). Therefore, we provide a sufficient condition which entails that a type constraint system is structural, and shows how large that class is.

Proposition 42. *Let $T = \{T_V\}_{V \in \wp_f(\mathcal{V})}$ be a type constraint system for the type system $\langle \Delta, \Sigma, I \rangle$. Assume there exists a ground term $t \in \mathsf{terms}(\Sigma, \emptyset)$ such that for every $d \in \mathsf{terms}(\Delta, \emptyset)$ and for all $t' \in [\![d]\!]I$, every term obtained from t' by substituting some occurrences of t with variables of $\mathcal{V}$ is still in $[\![d]\!]I$. Then T is structural.*

Example 43. Consider the type system NL of Example 25 and the type constraint system $NL = \{NL_V\}_{V \in \wp_f(\mathcal{V})}$ where $NL_V = \Phi_{\Delta,V}$. Since Proposition 42 holds with $t = \mathsf{s}([])$, the type system NL is structural. As it contains more than one type, NL is not positive. This shows that a structural type constraint system may not be positive. The same construction and results can be applied to the type constraint system NLT derived in the same way as NL from the type system NLT of Example 25.

Example 44. Consider the type constraint system $Pos_{\{g\}}$ for the type system G of Example 40. A non-ground term t does not belong to any type, although, every ground instance of t belongs to g. Therefore, there is no way of grounding t and maintain the same type properties of t. This shows that a type constraint system may be positive but *not* structural.

Example 45. Consider the type constraint system $N = \{N_V\}_{V \in \wp_f(\mathcal{V})}$ for the type system $\langle \{\mathtt{nat}^0\}, \Sigma, I\rangle$ where Σ contains $\mathsf{0}^0$, s^1 and a^0 and $I(\mathtt{nat}) = \mu i.\{\mathsf{0}\} \cup \{\mathsf{s}(n) \mid n \in i\}$. If N_V is the set of positive formulas over $\{\mathtt{nat}\}$ and V, we have that N is positive. Moreover, it is structural, as it can be proved by choosing $t = \mathsf{a}$ in Proposition 42. This proves that there are type constraint systems which are both positive and structural.

Example 46. Consider the type constraint system $NG = \{\Phi_{\{\mathtt{nat}^0, \mathtt{g}^0\}, V}\}_{V \in \wp_f(\mathcal{V})}$ for the type system $\langle \{\mathtt{nat}, \mathtt{g}\}, \Sigma, I\rangle$ where Σ contains $\mathsf{0}^0$ and s^1, $I(\mathtt{nat}) = \mu i.\{\mathsf{0}\} \cup \{\mathsf{s}(n) \mid n \in i\}$ and $I(\mathtt{g}) = \mathsf{terms}(\Sigma, \emptyset)$. This type constraint system is not positive since it contains more than one type. Moreover, it is not structural since every non-ground term has no ground instance with its same type properties. This proves that there are type constraint systems which are neither positive nor structural.

The following propositions show the importance of positive or structural type constraint systems.

Proposition 47. *Every positive or structural type constraint system satisfies both properties P1 and P2.*

Proposition 48. *Let* $T = \{T_V\}_{V \in \wp_f(\mathcal{V})}$ *be a type constraint system for the type system* $\langle \Delta, \Sigma, I\rangle$*, satisfying property P1.*

i) The operator $\star^{T_V}$ *is always correct w.r.t.* $\star^{Down_V}$*, and it is its best possible approximation if property P2 holds for* T*.*

ii) The operator $\exists^{T_V}$ *is always correct w.r.t.* $\exists^{Down_V}$*, and it is its best possible approximation if* T *is positive or structural.*

iii) The operator rename^{T_V} *is always correct w.r.t.* rename^{Down_V} *and it is its best possible approximation if property P2 holds for* T*.*

The correctness and optimality results for rename^{T_V} cannot be immediately combined with the similar results of Propositions 23.iii and 19.iii since Proposition 19.iii is a restricted optimality result, for the hypothesis on h. Therefore, we must check that, if a variable y does not occur in a transfinite formula ϕ, then $\gamma(\phi)$ is the union of sets of solutions of existential Herbrand constraints where y does not occur.

Proposition 49. *Let* $T = \{T_V\}_{V \in \wp_f(\mathcal{V})}$ *be a type constraint system. Let* $V \in \wp_f(\mathcal{V})$*,* $\tilde{y} \in \mathrm{Seq}(V)$ *and* $\phi \in T_{V \setminus \tilde{y}}$*. There exists* $I \subseteq H_{V \setminus \tilde{y}}$ *such that* $\gamma_{T_V}(\phi) = \cup\{\mathsf{sol}_V(h) \mid h \in I\}$*.*

In conclusion, a positive or structural type constraint system $\mathit{Type}_V = \{\mathit{Type}_V\}_{V\in\wp_f(\mathcal{V})}$ enjoys interesting properties. Namely, cylindrification is well-defined (property P1), a Galois insertion can be established between Down_V and Type_V (property P2), the operators over Type_V are the best possible approximations of the corresponding operators over Down_V (Propositions 47 and 48).

Since $\mathit{Pos}_{\{\mathtt{g}\}}$ and $\mathit{Def}_{\{\mathtt{g}\}}$ of Example 40 and $\mathit{Pos}_{\{\mathtt{nf}\}}$ and $\mathit{Def}_{\{\mathtt{nf}\}}$ of Example 41 are positive, we conclude that they are related to *Down* through a Galois insertion and that the operations of Definition 28 are the best possible approximations of the corresponding operations of *Down*, by using Proposition 48. Since that proposition considers every positive type constraint system, we have generalised the result proved in [11] for the case of groundness. Therefore, we can use many *incarnations* of positive type analyses, combining them with a reduced product operation and using the operations of Definition 28 as done in [8], where, however, the authors did not provide any justification of the correctness of this approach.

For groundness and non-freeness we considered only positive formulas (Examples 40 and 41), while we considered the whole set of formulas for *NL* and *NLT* (Example 43). Indeed, in the case of groundness or non-freeness, a variable can always eventually belong to the type, and a formula like $\neg(x \in \mathtt{g})$ has an empty concretisation. Therefore, it is useless. Instead, the formula $\neg(x \in \mathtt{nat})$ has a clear meaning. Namely, it says that x is not and can never become a natural number. For instance, x might be bound to a list. Those formulas are said to represent *negative information.*

4.4 Type systems and linear refinement

We have shown that every type constraint system induces an abstract interpretation of *Down* (Proposition 36). In this subsection, we want to show how a hierarchy of abstract interpretations of *Down* can be defined starting from one that just models the type properties of interest. Every domain in this hierarchy will be the linear refinement of the previous one. This means that our hierarchy will be a chain of domains which induce an approximation of concrete conjunction with increasing precision. This generalises an analogous result about groundness analysis shown in [44].

Definition 50. Let $\mathsf{T} = \langle \Delta, \Sigma, I\rangle$ be a type system and $V \in \wp_f(\mathcal{V})$. We define

$$\mathbf{v}_t = \{\theta \in \Theta_V \mid \theta(v) \in [\![t]\!]I\} \qquad \text{with } v \in V \text{ and } t \in \mathsf{terms}(\Delta, \emptyset)$$

$$\mathit{Basic}^0_{\mathsf{T},V} = \bigwedge\{\mathbf{v}_t \mid v \in V \text{ and } t \in \mathsf{terms}(\Delta,\emptyset)\},$$

$$\mathit{Basic}^{i+1}_{\mathsf{T},V} = \mathit{Basic}^i_{\mathsf{T},V} \twoheadrightarrow \mathit{Basic}^i_{\mathsf{T},V} \qquad \text{for } i \geq 0.$$

We show now that $\mathit{Basic}^i_{\mathsf{T},V} \subseteq \mathit{Basic}^i_{\mathsf{T},V} \triangleright \mathit{Basic}^i_{\mathsf{T},V}$, which allows us to use Equation (4) in Definition 50.

Proposition 51. *Let* $V \in \wp_f(\mathcal{V})$ *and* T *be a type system. We have* $\mathit{Basic}^i_{\mathsf{T},V} \subseteq \mathit{Basic}^i_{\mathsf{T},V} \triangleright \mathit{Basic}^i_{\mathsf{T},V}$ *for all* $i \geq 0$.

The domains $\mathit{Basic}^0_{\mathsf{T},V}$, $\mathit{Basic}^1_{\mathsf{T},V}$ and $\mathit{Basic}^2_{\mathsf{T},V}$ can be easily represented.

Definition 52. Let $\langle \Delta, \Sigma, I \rangle$ be a type system. Let $V \in \wp_f(\mathcal{V})$. We define

$$\begin{aligned}
And_{\Delta,V} &= \{\wedge S \mid S \subseteq \{v \in t \mid v \in V \text{ and } t \in \mathsf{terms}(\Delta, \emptyset)\}\} , \\
Or_{\Delta,V} &= \{\vee S \mid S \subseteq \{v \in t \mid v \in V \text{ and } t \in \mathsf{terms}(\Delta, \emptyset)\} \text{ and } S \neq \emptyset\} , \\
Def_{\Delta,V} &= \{A_1 \Rightarrow A_2 \mid \{A_1, A_2\} \subseteq And_{\Delta,V}\} , \\
Pos_{\Delta,V} &= \{A \Rightarrow O \mid A \in And_{\Delta,V} \text{ and } O \in Or_{\Delta,V}\} .
\end{aligned}$$

Proposition 53. *Let* $\mathsf{T} = \langle \Delta, \Sigma, I \rangle$ *be a type system and* $V \in \wp_f(\mathcal{V})$. *Then* $Def_{\Delta,V}$ *is isomorphic to* $Basic^1_{\Delta,V}$. *If* T *is structural then* $Pos_{\Delta,V}$ *is isomorphic to* $Basic^2_{\Delta,V}$, $Basic^2_{\mathsf{T},V}$ *is condensing,* $Basic^2_{\mathsf{T},V} = Basic^i_{\mathsf{T},V}$ *for* $i \geq 2$ *and* $Basic^2_{\mathsf{T},V} = (Basic^0_{\mathsf{T},V} \twoheadrightarrow Basic^0_{\mathsf{T},V}) \twoheadrightarrow Basic^0_{\mathsf{T},V}$.

The condensing property means that, for structural type constraint systems, $Pos_{\Delta,V}$ induces a conjunction operation which fully propagates the properties in $Basic^0_{\Delta,V}$. Therefore, our denotational semantics (Subsection 2.1) leads to an analysis which is as precise as an analysis based on an equivalent operational semantics.

Note that we do not know if an analogous result of Proposition 53 holds for the type system of a positive type constraint system (Definition 39). However, in [44] it is shown that Proposition 53 holds for the type system G of Example 25. The proofs of [44] hold for the type system NF for non-freeness too (Example 25). However, they cannot be generalised to every type system of a positive type constraint system.

5 Logic programs as finite type domains

In this section we show how the $Basic^1_{\Delta,V}$ domain of the previous section can be implemented. We have shown in Proposition 53 that we can use $Def_{\Delta,V}$ although, in general, this solution is not effective since transfinite formulas are finite objects only if the set of types is finite. In such a case, the formulas are finite and the abstract operators of Definition 28 can be seen as algorithms. Even the equivalence test between two formulas (needed to check termination of the fixpoint calculation) becomes effective, though very expensive, being an *NP*-complete problem [39]. However, a finite set of types is useful for mode analysis (groundness, non-freeness) but does not allow us to make interesting type analyses involving polymorphic types. When $\mathsf{terms}(\Sigma, \emptyset)$ is infinite, transfinite formulas are not finite objects. However, in many cases, a finite formula with *type variables* can be used instead. For instance, assuming that we have a polymorphic type `list`, the infinite conjunction

$$\wedge_{d \in \mathsf{terms}(\Delta,\emptyset)} (x \in \mathtt{list}(d) \iff (h \in d \wedge l \in \mathtt{list}(d)))$$

expresses the relationship between the variables of the constraint $\{x = [h|l]\}$. By using type variables, the same information can be expressed by the *finite* formula $x \in \mathtt{list}(T) \iff (h \in T \wedge l \in \mathtt{list}(T))$. Note that this formula can be written as the logic program

```
x(list(T)) :- h(T), l(list(T)).
h(T) :- x(list(T)).
l(list(T)) :- x(list(T)).
```

In this program, the variables of interest (x, h and l in the example) are constants in the language of the program, while the type variables are the real variables of the program. In the following, we assume there is an infinite set $\mathcal{T}$ of type variables, denoted by uppercase letters. Moreover, we will consider a fact `f.` equivalent to a clause `f :-` .

5.1 The domain $Prog^k$

A type model provides a set of types for every program variable. This set must be exactly the set of types a given term belongs to.

Definition 54. Given a type system $\mathsf{T} = \langle \Delta, \Sigma, I \rangle$, a *type model* for T is a map $M : \mathcal{V} \mapsto \{\{d \in \mathsf{terms}(\Delta, \emptyset) \mid t \in [\![d]\!]I\} \mid t \in \mathsf{terms}(\Sigma, \mathcal{V})\}$.

Programs in $Prog^k$ use type terms of depth k in order to have a finite constraint system.

Definition 55. Let $\mathsf{T} = \langle \Delta, \Sigma, I \rangle$ be a type system, $V \in \wp_f(\mathcal{V})$ and $k \geq 1$. A k-atom for V and Δ is $\mathtt{v}(t)$, where $v \in V$ and $t \in \mathsf{terms}^k(\Delta, \mathcal{T})$. An element of $Prog^k_V$ is a (possibly empty or infinite) set of clauses $H :- B.$ where H is a k-atom for V and Δ and B is a (possibly empty) sequence of k-atoms for V and Δ. We define the constraint system $Prog^k = \{Prog^k_V\}_{V \in \wp_f(\mathcal{V})}$ with the operations

$$P_1 \star^{Prog^k_V} P_2 = P_1 \cup P_2$$

$$\mathsf{rename}^{Prog^k_V}_{\tilde{x} \to \tilde{y}} P = \begin{cases} P[\tilde{y}/\tilde{x}] & \text{if } P \in Prog^k_{V \setminus \tilde{y}} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\exists^{Prog^k_V}_x P = (P \cup P') \cap Prog^k_{V \setminus x} \ ,$$

where P' is the set of clauses which can be obtained by folding and removing the clauses in P of the form $\mathtt{x}(t) :- B.$ in the body of the other clauses[5].

Given a program $P \in Prog^k_V$ and a type model M for T, we define

$$\begin{aligned}
M \models \mathtt{v}(t) &\text{ iff } t \in M(v), \quad \text{with } t \text{ ground} \\
M \models A_1, \ldots, A_n &\text{ iff } M \models A_i \text{ , for all } i = 1, \ldots, n \ (n \geq 0) \ , \\
&\qquad \text{with } A_1, \ldots, A_n \text{ ground} \\
M \models H :- B. &\text{ iff } M \models B \text{ entails } M \models H, \text{ when } H :- B. \text{ is ground} \\
M \models H :- B. &\text{ iff } M \models H' :- B'. \text{ for every ground instance } H' :- B'. \\
&\qquad \text{of } H :- B., \text{ when } H :- B. \text{ is not ground} \\
M \models P &\text{ iff } M \models c \text{ for all } c \in P \ .
\end{aligned}$$

We define $M_{\mathsf{T}}(P) = \{M \mid M \text{ is a type model for } \mathsf{T} \text{ and } M \models P\}$.

For all $V \in \wp_f(\mathcal{V})$, the programs in $Prog^k_V$ are partially ordered as $P_1 \leq_{\mathsf{T}} P_2$ if and only if $M_{\mathsf{T}}(P_1) \subseteq M_{\mathsf{T}}(P_2)$. We define $P_1 \equiv_{\mathsf{T}} P_2$ if and only if $P_1 \leq_{\mathsf{T}} P_2$ and $P_2 \leq_{\mathsf{T}} P_1$. The quotient of the set $Prog^k_V$ w.r.t. $\equiv_{\mathsf{T}}$ is a complete lattice. The top element is the empty program and the greatest lower bound operation is the composition of programs (i.e., $\star^{Prog^k_V}$). From now on, every program will stand for its equivalence class.

[5] This folding could be applied more than once, for achieving a better precision. However, practical experiments have shown that one application leads to a sufficiently precise result.

`x(list(T)) :- y(T), z(list(T)).`
`y(T) :- x(list(T)).`
`z(list(T)).`
	`x(list(T)) :- y(T).`
	`y(T) :- x(list(T)).`
		`z(list(T)).`

A program $P \in Prog_V^1$	$\exists_z^{Prog_V^1} P$	$\exists_x^{Prog_V^1} P$

Figure 1: A program and two of its cylindrifications.

Example 56. Figure 1 shows a program together with its restrictions w.r.t. z and x. Note that in $\exists_x^{Prog_V^1} P$ we have dropped a tautological clause for $\mathtt{y(T)}$, since we deal with equivalence classes of constraints.

Since $Prog_V^k$ and T_V are complete lattices, by defining a co-additive function $\gamma^k : Prog_V^k \to T_V$ we conclude that $Prog_V^k$ is an abstract interpretation of T_V.

Definition 57. Let $\langle \Delta, \Sigma, I \rangle$ be a type system, $k \geq 1$ and $V \in \wp_f(\mathcal{V})$. We define

$$\mathtt{v}(t)' = v \in t \ , \qquad B' = \bigwedge_{b \in B} b' \quad \text{where } B \text{ is a sequence of } k\text{-atoms.}$$

Let $V_1, \ldots, V_n$ be the type variables in the clause $H :- B.$ and let $P \in Prog_V^k$. We define

$$\gamma_{\mathsf{T}}^k(H :- B.) = \bigwedge_{t_1,\ldots,t_n \in \mathsf{terms}(\Delta,\emptyset)} H'[t_1/V_1]\cdots[t_n/V_n] :- B'[t_1/V_1]\cdots[t_n/V_n].$$

$$\gamma_{\mathsf{T}}^k(P) = \bigwedge_{c \in P} \gamma_{\mathsf{T}}^k(c) \ .$$

When it is clear from the context, we write γ^k for γ_{T}^k. The abstraction map induced by γ^k will be called α^k.

Proposition 58. *Given a type system* T*,* $k \geq 1$ *and* $V \in \wp_f(\mathcal{V})$*,* $\gamma_{\mathsf{T}}^k : Prog_V^k \to T_V$ *is co-additive.*

The operations of Definition 55 are correct or even optimal w.r.t. the corresponding operations of $Def_{\Delta,V}$ (Definition 28).

Proposition 59. *Let* $\mathsf{T} = \langle \Delta, \Sigma, I \rangle$ *be a type system,* $k \geq 1$ *and* $V \in \wp_f(\mathcal{V})$*.*

i) The operator $\star^{Prog_V^k}$ *is the best possible approximation of* $\star^{Def_{\Delta,V}}$*.*

ii) The operator $\exists^{Prog_V^k}$ *is correct w.r.t.* $\exists^{Def_{\Delta,V}}$*.*

iii) The operator $\mathsf{rename}^{Prog_V^k}$ *is the best possible approximation of the operator* $\mathsf{rename}^{Def_{\Delta,V}}$*.*

When k is finite, $Prog_V^k$ is a finite set for every $V \in \wp_f(\mathcal{V})$. Since the operations introduced in Definition 55 are algorithms, we conclude that $Prog_V^k$ can be used for type analysis. Note that considering only programs with bounded term depth does not reduce our analysis to the case of a finite set of types. Indeed, type variables can be bound to terms of arbitrary depth. Therefore, as the examples in Section 6 will show, the restriction on the depth of the terms does not introduce a significant loss in precision.

The definition of $\exists_x^{Prog_V^k}$ uses concrete unification between type terms. Since types are partially ordered with respect to subtyping (for instance, $\mathtt{nat} \leq \mathtt{top}$), the unification procedure used for folding might be too coarse. For instance, if we have a clause whose head is $\mathtt{x(list(nat))}$ and we try to fold it in the body of a clause containing $\mathtt{x(list(top))}$, the unification procedure fails. Actually, folding should be allowed because if x is a list of integers then it is a list of generic terms. Similarly, if we have a clause whose body contains $\mathtt{x(T)}$, we can remove this k-atom from the body and instantiate the resulting clause with the substitution $\{T \mapsto \mathtt{top}\}$, provided we have a top type `top`. This is correct because every term is always in `top`. In conclusion, the precision of the cylindrification operator can be improved by using a unification procedure which is aware of subtyping information.

To make $Prog_V^k$ useful for program analysis, we need two algorithms: one that abstracts a concrete Herbrand constraint into an element of $Prog_V^k$, and another algorithm that extracts from an element of $Prog_V^k$ the set of types a variable is bound to. They are described in the following two subsections.

5.2 Abstraction

We first define an algorithm which approximates the restriction of the map $\lambda h.\alpha^k(\alpha_{Def_{\Delta,V}}(\cup(\mathsf{sol}_V(h))))$ to singleton sets of existential Herbrand constraints without existential variables (i.e., to substitutions). This map is the concatenation of the previously defined abstraction maps that link the existential Herbrand constraints to $Prog^k$. Since, for the first time in this abstractions chain, $Prog^k$ will be used to implement a type analyser, we need an algorithmic definition for this compound abstraction map.

We assume there exists a Prolog procedure `type`/2 which determines if some instance of a term can belong to a type, and provides necessary and sufficient conditions on the instantiation of the variables of the term such that this happens.

Definition 60. Let $\mathsf{T} = \langle \Delta, \Sigma, I \rangle$ be a type system, $k \geq 1$ and $V \in \wp_f(\mathcal{V})$. We define the predicate `type`/2 so that a query of the form $\mathtt{type}(Term, Type)$, with $Term \in \mathsf{terms}(\Sigma, V)$ such that $\mathsf{vars}(Term) = X$ and $Type \in \mathsf{terms}(\Delta, \mathcal{T})$, has computed answer θ where θ satisfies the following condition: for each $\sigma \in \Theta_V$ and $\mu : \mathcal{T} \to \mathsf{terms}(\Delta, \emptyset)$ we have $(Term)\sigma \in [\![(Type)\theta\mu]\!]I$ if and only if $\sigma(x) \in [\![\theta(x)\mu]\!]I$ for every $\in X$.

Example 61. Figure 2 shows an example of the procedure `type`/2 for the type system NL of Example 25. That type system contains the types `top`, `nat` and polymorphic `lists`. The query `type([H|L],Type)` yields a computed answer substitution $\{\mathtt{Type} \mapsto \mathtt{list(S)}, \mathtt{H} \mapsto \mathtt{S}, \mathtt{L} \mapsto \mathtt{list(S)}\}$, meaning that the term `[H|L]` can be instantiated to a term of type `list(S)` if and only if `H` is instantiated to a term of type `S` and `L` to a term of type `list(S)`, for every instantiation of `S`.

A definition of the `type`/2 procedure can be derived automatically from the definition of types and can be made compositional with respect to addition of new types to the type system. We do not address this problem in detail. This

meta-clause `type(X,S):-var(X),!,X=S.`
the whole universe $\mathsf{terms}(\Sigma, V)$ `type(X,top).`

natural numbers: $\mu n.n = \{\mathtt{0}\} \cup \{\mathtt{s}(l) \mid l \in n\}$
`type(X,nat):-X=0.`
`type(X,nat):-X=s(N),type(N,nat).`

polymorphic lists: $\lambda s.\mu l.l = \{[]\} \cup \{[h\|t] \mid h \in s \text{ and } t \in l\}$
`type(X,list(S)):-X=[].`
`type(X,list(S)):-X=[H|T],type(H,S),type(T,list(S)).`

Figure 2: An example of the procedure `type`.

would require the description of a type specification language. Note, however, that the problem is not new, since it is very similar to the problem of the definition of an abstraction map given a type specification, described in [45].

Definition 62. Given a type system $\mathsf{T} = \langle \Delta, \Sigma, I \rangle$, a procedure $\mathtt{type}/2$ for T, $k \geq 1$, $V \in \wp_f(\mathcal{V})$, $x \in V$ and $c \in C_V$ in normal form such that $\mathsf{vars}(c(x)) = \{x_1, \ldots, x_n\}$, we define

$$\begin{aligned} \alpha_x^{alg}(c) = & \{\mathtt{x}(t^i) :- \mathtt{x}_1(t^i_{x_1}), \ldots, \mathtt{x}_\mathtt{n}(t^i_{x_n}). \mid i = 1, \ldots, m\} \cup \\ & \{\mathtt{x}_\mathtt{j}(t^i_{x_j}) :- \mathtt{x}(t^i). \mid j = 1, \ldots, n, \ i = 1, \ldots, m\} \ , \end{aligned}$$

where $\mathtt{type}(c(x), Type)$ yields m computed answers, and the i-th computed answer, for $i = 1, \ldots, m$, is

$$\{Type \mapsto t^i, x_1 \mapsto t^i_{x_1}, \ldots, x_n \mapsto t^i_{x_n}\}.$$

We define

$$\alpha_V^{alg}(c) = \bigcup_{x \in V} \alpha_x^{alg}(c) \ .$$

The following proposition shows that the set of the solutions of an existential Herbrand constraint of the form $\exists_\emptyset c$ is correctly approximated by $\alpha_V^{alg}(c)$.

Proposition 63. *Given a type system $\mathsf{T} = \langle \Delta, \Sigma, I \rangle$ and a procedure $\mathtt{type}/2$ for T, then for each $k \geq 1$, $V \in \wp_f(\mathcal{V})$ and $c \in C_V$ in normal form, we have*

$$\downarrow c \subseteq \gamma^{Def_{\Delta,V}} \gamma^k \alpha_V^{alg}(c) \ .$$

The definition of the abstraction map can be improved by using the negative information contained in an existential Herbrand constraint. That information says that some variables *cannot* belong to some types. We do not consider this improved version here, though it has been implemented in the prototypical analyser used in Subsection 6.2.

5.3 Information extraction

We consider now the problem of how information can be extracted from an abstract constraint $P \in Prog_V^k$. Namely, we provide an algorithm which is able to determine if a variable $v \in V$ belongs to a type $d \in \mathsf{terms}(\Delta, \emptyset)$ when P is satisfied, i.e., if $\gamma^k(P) \leq (v \in d)$. Since $(v \in d) = \gamma^k(\{\mathtt{v(d)}.\})$, the following result allows us to compare the models of P and $\{\mathtt{v(d)}.\}$ instead of their concretisation through γ^k.

Proposition 64. *Let $\mathsf{T} = \langle \Delta, \Sigma, I \rangle$ be a type system, $k \geq 1$ and $V \in \wp_f(\mathcal{V})$. For all $\{P_1, P_2\} \subseteq Prog_V^k$ we have that*

$$P_1 \leq P_2 \textit{ entails } \gamma_{\mathsf{T}}^k(P_1) \leq \gamma_{\mathsf{T}}^k(P_2) .$$

Since in general we have an infinite set of type models, Proposition 64 does not provide an algorithm for checking if $\gamma^k(P) \leq \gamma^k(\{\mathtt{v(d)}.\})$. This can be achieved, instead, by using the following result.

Proposition 65. *Let $\mathsf{T} = \langle \Delta, \Sigma, I \rangle$ be a type system, $k \geq 1$, $V \in \wp_f(\mathcal{V})$, $P \in Prog_V^k$, $v \in V$ and $d \in \mathsf{terms}(\Delta, \mathcal{T})$. If $\mathtt{v(d)}$ can be derived from P by resolution (i.e., there is a refutation of $\mathtt{v(d)}$. in P with ε as computed answer substitution), then $P \leq \{\mathtt{v(d)}.\}$.*

In general, the resolution process is not finite. Therefore, we must halt after a finite fixed number of steps of resolution. The greater the number of steps is, the more complete will be the algorithmic entailment check. Moreover, note that another source of incompleteness is related to subtyping. Consider for instance the case $P = \{\mathtt{v(nat)}.\}$ and $d = \mathtt{top}$. We have $P \leq \{\mathtt{v(top)}.\}$ but $\mathtt{v(top)}$ cannot be derived by resolution from P. This is because resolution embeds a unification mechanism which does not consider any subtyping information. Using such information would improve the precision of the entailment test.

6 Examples

We illustrate the application of the domain $Prog^k$ to the type analysis of two procedures. The first one is the classical `append/3` procedure which appends two lists. We compute its denotation by hand, step by step. The second example is the type analysis of a complex procedure. This time, a prototypical analyser is used, embedding negative and some subtyping information.

6.1 Type analysis of append/3

Consider the well-known procedure `append/3` which appends two lists.

```
append([],L,L).
append([H|T],L,[H|A]):-append(T,L,A).
```

We want to compute its s-semantics by using the $Prog_V^2$ constraint system.

The first step of the analysis consists in the transformation of the program in the abstract syntax of Definition 3. The result is

$$\mathtt{append}(x, y, z) \leftarrow \{x = [], y = z\} \texttt{ or } (\{x = [h|t], z = [h|a]\} \texttt{ and } \mathtt{append}(t, y, a))$$

The second step is in the abstraction of the program (Definition 14) through the α_V^{alg} map of Definition 62. We use the type constraint system *NL* from Examples 25 and 43 and the `type` procedure of Example 61, given in Figure 2. The result of the abstraction of the program is the following.

$$
\begin{array}{l}
\texttt{append}(x,y,z) \leftarrow \left(\begin{array}{l} \texttt{x(list(T)).} \\ \texttt{y(T)} :- \texttt{z(T).} \\ \texttt{z(T)} :- \texttt{y(T).} \end{array} \right) \texttt{ or} \\
\quad \texttt{or} \left(\underbrace{\begin{array}{l} \texttt{x(list(T))} :- \texttt{h(T),t(list(T)).} \\ \texttt{h(T)} :- \texttt{x(list(T)).} \\ \texttt{t(list(T))} :- \texttt{x(list(T)).} \\ \texttt{z(list(T))} :- \texttt{h(T),a(list(T)).} \\ \texttt{h(T)} :- \texttt{z(list(T)).} \\ \texttt{a(list(T))} :- \texttt{z(list(T)).} \end{array}}_{Q} \texttt{ and append}(t,y,a) \right)
\end{array} \tag{7}
$$

By definition we have

$$T_P{\uparrow}0(\perp)(\texttt{append}) = \emptyset \ .$$

The first iteration uses only the first branch of the `or`. This is because the second branch relies on the denotation of `append`, which is still empty. Therefore, we have

$$T_P{\uparrow}1(\perp)(\texttt{append}) = \left\{ \begin{array}{l} \iota_1\texttt{(list(T)).} \\ \iota_2\texttt{(T)} :- \iota_3\texttt{(T).} \\ \iota_3\texttt{(T)} :- \iota_2\texttt{(T).} \end{array} \right\} \ .$$

Note that variables have been renamed into ι variables, following Definition 6.

The second iteration yields the same result through the left branch of `or`, and a new constraint through its right branch. We compute this second constraint step by step.

The first step is the computation of $\mathcal{CA}^{Prog_V^2}[\![\texttt{append}(t,y,a)]\!]\, T_P{\uparrow}1(\perp)$. Following definition 5, we have

$$
\begin{aligned}
&\mathcal{CA}^{Prog_V^2}[\![\texttt{append}(t,y,a)]\!]\, T_P{\uparrow}1(\perp) \\
&\qquad = \mathsf{rename}^{Prog_V^2}_{\langle \iota_1,\iota_2,\iota_3\rangle \to \langle t,y,a\rangle} \left\{ \begin{array}{l} \iota_1\texttt{(list(T)).} \\ \iota_2\texttt{(T)} :- \iota_3\texttt{(T).} \\ \iota_3\texttt{(T)} :- \iota_2\texttt{(T).} \end{array} \right\} \\
&\qquad = \left\{ \begin{array}{l} \texttt{t(list(T)).} \\ \texttt{y(T)} :- \texttt{a(T).} \\ \texttt{a(T)} :- \texttt{y(T).} \end{array} \right\} \ .
\end{aligned}
$$

The second step is the computation of $\mathcal{CA}^{Prog_V^2}[\![Q \texttt{ and append}(t,y,a)]\!]\, T_P{\uparrow}1(\perp)$, where Q is the program defined in Equation (7). By Definition 5 we have

$$\begin{aligned}
&\mathcal{CA}^{Prog^2_V}[\![Q \texttt{ and append}(t,y,a)]\!]\, T_P{\uparrow}1(\bot)\\
&= \mathcal{CA}^{Prog^2_V}[\![Q]\!]\, T_P{\uparrow}1(\bot) \otimes^{Prog^2_V} \mathcal{CA}^{Prog^2_V}[\![\texttt{append}(t,y,a)]\!]\, T_P{\uparrow}1(\bot)\\
&= \{Q\} \otimes^{Prog^2_V} \mathcal{CA}^{Prog^2_V}[\![\texttt{append}(t,y,a)]\!]\, T_P{\uparrow}1(\bot)\\
&= \{Q\} \otimes^{Prog^2_V} \left\{\begin{array}{l}\texttt{t(list(T)).}\\ \texttt{y(T) :- a(T).}\\ \texttt{a(T) :- y(T).}\end{array}\right\}
\end{aligned}$$

$$= \left\{\underbrace{\begin{array}{l}\texttt{t(list(T)).}\\ \texttt{y(T) :- a(T).}\\ \texttt{a(T) :- y(T).}\\ \texttt{x(list(T)) :- h(T), t(list(T)).}\\ \texttt{h(T) :- x(list(T)).}\\ \texttt{t(list(T)) :- x(list(T)).}\\ \texttt{z(list(T)) :- h(T), a(list(T)).}\\ \texttt{h(T) :- z(list(T)).}\\ \texttt{a(list(T)) :- z(list(T)).}\end{array}}_{R}\right\}. \tag{8}$$

Following Definition 6, we must rename the program R defined in Equation (8) and cylindrify w.r.t. the variables not in $\{\iota_1, \iota_2, \iota_3\}$. The renaming operation yields

$$\mathsf{rename}^{Prog^2_V}_{\langle x,y,z\rangle \to \langle \iota_1,\iota_2,\iota_3\rangle}(\{R\}) = \left\{\underbrace{\begin{array}{l}\texttt{t(list(T)).}\\ \iota_2\texttt{(T) :- a(T).}\\ \texttt{a(T) :- }\iota_2\texttt{(T).}\\ \iota_1\texttt{(list(T)) :- h(T), t(list(T)).}\\ \texttt{h(T) :- }\iota_1\texttt{(list(T)).}\\ \texttt{t(list(T)) :- }\iota_1\texttt{(list(T)).}\\ \iota_3\texttt{(list(T)) :- h(T), a(list(T)).}\\ \texttt{h(T) :- }\iota_3\texttt{(list(T)).}\\ \texttt{a(list(T)) :- }\iota_3\texttt{(list(T)).}\end{array}}_{R^1}\right\}.$$

We perform the cylindrification operation one variable at a time.

$$\exists^{Prog^2_V}_{\{t\}}(\{R^1\}) = \left\{\underbrace{\begin{array}{l}\iota_2\texttt{(T) :- a(T).}\\ \texttt{a(T) :- }\iota_2\texttt{(T).}\\ \iota_1\texttt{(list(T)) :- h(T).}\\ \texttt{h(T) :- }\iota_1\texttt{(list(T)).}\\ \iota_3\texttt{(list(T)) :- h(T), a(list(T)).}\\ \texttt{h(T) :- }\iota_3\texttt{(list(T)).}\\ \texttt{a(list(T)) :- }\iota_3\texttt{(list(T)).}\end{array}}_{R^2}\right\},$$

$$\exists_{\{a\}}^{Prog_V^2}(\{R^2\}) = \left\{ \underbrace{\begin{array}{l} \iota_2(\texttt{list(T)}) :- \iota_3(\texttt{list(T)}). \\ \iota_1(\texttt{list(T)}) :- \texttt{h(T)}. \\ \iota_3(\texttt{list(T)}) :- \texttt{h(T)}, \iota_2(\texttt{list(T)}). \\ \texttt{h(T)} :- \iota_1(\texttt{list(T)}). \\ \texttt{h(T)} :- \iota_3(\texttt{list(T)}). \end{array}}_{R^3} \right\}.$$

The final cylindrification w.r.t. h is done together with the addition of the constraint which arises from the left hand branch of the `or` construct. This yields the set of two programs

$$\begin{aligned} T_P{\uparrow}2(\bot)(\texttt{append}) &= \left\{ \begin{array}{l} \iota_1(\texttt{list(T)}). \\ \iota_2(\texttt{T}) :- \iota_3(\texttt{T}). \\ \iota_3(\texttt{T}) :- \iota_2(\texttt{T}). \end{array} \right\} \cup \exists_{\{h\}}^{Prog_V^2}(\{R^3\}) \\ &= \left\{ \begin{array}{l} \iota_1(\texttt{list(T)}). \\ \iota_2(\texttt{T}) :- \iota_3(\texttt{T}). \\ \iota_3(\texttt{T}) :- \iota_2(\texttt{T}). \end{array} , \begin{array}{l} \iota_1(\texttt{list(T)}) :- \iota_3(\texttt{list(T)}). \\ \iota_2(\texttt{list(T)}) :- \iota_3(\texttt{list(T)}). \\ \iota_3(\texttt{list(T)}) :- \iota_1(\texttt{list(T)}), \iota_2(\texttt{list(T)}). \end{array} \right\}. \end{aligned}$$

Since it can be shown that $T_P{\uparrow}3(\bot) = T_P{\uparrow}2(\bot)$, we have $\mathcal{S}_P = T_P{\uparrow}2(\bot)$.

Assume we are interested in the abstract behaviour of **append**/3 when it is called with its first and second argument bound to lists. This means that we want to compute

$$\mathcal{CA}^{Prog_V^2} \left[\!\!\left[\begin{array}{l} \texttt{x(list(top))}. \\ \texttt{y(list(top))}. \end{array} \ \textbf{and}\ \texttt{append}(x, y, z) \right]\!\!\right] \mathcal{S}_P$$

which, after renaming and conjunctions (Definition 5), is the set of two programs

$$\left\{ \begin{array}{l} \texttt{x(list(top))}. \\ \texttt{y(list(top))}. \\ \texttt{x(list(T))}. \\ \texttt{y(T)} :- \texttt{z(T)}. \\ \texttt{z(T)} :- \texttt{y(T)}. \end{array} , \begin{array}{l} \texttt{x(list(top))}. \\ \texttt{y(list(top))}. \\ \texttt{x(list(T))} :- \texttt{z(list(T))}. \\ \texttt{y(list(T))} :- \texttt{z(list(T))}. \\ \texttt{z(list(T))} :- \texttt{x(list(T))}, \texttt{y(list(T))}. \end{array} \right\}.$$

Since from both programs it is possible to derive the fact `z(list(top))` by resolution, we conclude that z is bound to a list after the call of **append** with its first two arguments bound to lists (Proposition 65).

6.2 Type analysis of `derivative/2`

We have implemented a small analyser for pure logic programs. It transforms a logic program into the abstract syntax of Definition 3, then abstracts the program by using a generic constraint system. Finally, it computes the abstract fixpoint and allows us to evaluate queries in this fixpoint. We have used $Prog_V^k$ as constraint system. It can be specialised w.r.t. a given set of types, through the specification of the `type`/2 procedure of Definition 60. We have implemented negative information.

Consider the program shown in Figure 3. It computes the derivative of an expression w.r.t. the variable x. We use the types `top`, representing the whole set of terms; `int`, representing integers; `expr`, representing generic expressions

```
int(0).
int(s(I)):-int(I).

derivative(x,s(0)).
derivative(X,0):-int(X).
derivative(X*Y,(DX*Y)+(X*DY)):-derivative(X,DX),derivative(Y,DY).
derivative(X+Y,DX+DY):-derivative(X,DX),derivative(Y,DY).
derivative(-(X),-(DX)):-derivative(X,DX).
derivative(X-Y,DX-DY):-derivative(X,DX),derivative(Y,DY).
derivative(X^K,DK*K*(X^(K-s(0)))):-derivative(K,DK).
derivative(exp(X),DX*exp(X)):-derivative(X,DX).
derivative(sin(X),DX*cos(X)):-derivative(X,DX).
derivative(cos(X),-(DX*sin(X))):-derivative(X,DX).
```

Figure 3: The `derivative`/2 procedure.

on x; and `algebraic`, representing expressions on x which do not involve exponentiation or trigonometric functions. We evaluate the query

$$(\texttt{x(algebraic)}.) \texttt{ and derivative}(x, y)$$

in the abstract fixpoint computed by the analyser. The result is the set of constraints shown in Figure 4. If the predicate `false` is derivable by resolution from a constraint, then that constraint can be dropped. This is a consequence of the use of negative information. In our case, constraints 3, 6, 7 and 8 can be dropped. ¿From the remaining four constraints, we derive the fact `y(expr)`. This means that the second argument is bound to an expression. More interestingly, the same constraints allow us to derive the fact `y(algebraic)`, i.e., the second argument is bound to an algebraic expression. Thus the analyser allows us to conclude that the derivative of an algebraic expression is an algebraic expression. Note that this result has been possible only by using negative information.

7 Conclusions

We have defined a large class of type domains that enjoy the same desirable properties of the well-known domains for groundness analysis [2, 10, 11]. This leads to the use of transfinite formulas and operators on transfinite formulas for the type analysis of logic programs. The analysis can be made finite by using *type variables*, which allow one to represent infinite conjunctions by using a finite object. The resulting domains are logic programs, whose variables can be interpreted as type variables. The abstract operations are operations over logic programs. We conjecture that the use of logic programs as abstract domains is not restricted to the particular case of the analysis of logic programs, but is a general result which can be applied to other programming paradigms. Indeed, a logic program expresses dependency information about the abstract property.

Since our framework is based on abstract interpretation and linear refinement, its design has been largely *guided* by the theory, rather than being the consequence of a particular problem or desire. Therefore, the problem of the

```
constraint 1
false :- x(int).
x(algebraic).
x(expr).
y(algebraic).
y(expr).
y(int).
```

```
constraint 2
x(algebraic).
x(expr).
x(int).
y(algebraic).
y(expr).
y(int).
```

```
constraint 3
false :- x(algebraic).
false :- x(int).
false :- y(algebraic).
false :- y(int).
x(algebraic).
x(expr).
y(expr).
```

```
constraint 4
false :- x(int).
false :- y(int).
x(algebraic).
x(expr).
y(algebraic).
y(expr).
```

```
constraint 5
false :- x(int).
false :- y(int).
x(algebraic).
x(expr).
x(expr) :- y(expr).
y(algebraic) :- x(algebraic).
y(expr) :- x(expr).
```

```
constraint 6
false :- x(algebraic).
false :- x(int).
false :- y(algebraic).
false :- y(int).
x(algebraic).
x(expr) :- y(expr).
y(expr) :- x(expr).
```

```
constraint 7
false :- x(algebraic).
false :- x(int).
false :- y(algebraic).
false :- y(int).
x(algebraic).
x(expr) :- y(expr).
y(expr) :- x(expr).
```

```
constraint 8
false :- x(algebraic).
false :- x(int).
false :- y(algebraic).
false :- y(int).
x(algebraic).
x(expr).
y(expr).
```

Figure 4: The set of constraints computed for our query.

effectivity of the analysis has been considered only in the final abstraction to $Prog^k$, where we fixed a finite k in order to have a finite domain. This distinguishes our approach from the many others contained in the literature.

We are left with several open problems.

- It would be interesting to know if the condition of being positive or structural, which entails all the desirable properties of a type domain, can be weakened.

- The use of programs for representing the Pos_Δ type constraint system (Definition 52) should be investigated. We think that Pos_Δ should be represented by disjunctive logic programs instead of traditional logic programs.

- The operation $\exists^{Prog^k_V}$ is correct (Proposition 59). However, we are also interested in an algorithm for the optimal cylindrification operation.

- We know that the algorithm for approximating the abstraction map is correct (Proposition 63). However, an optimal version would be desirable, even for a restricted class of types.

- A type specification language should be provided, similar to that of the Gödel programming language [24].

- Subtyping information should be extracted from a type specification and used for improving the precision of the analysis.

References

[1] K. R. Apt and E. Marchiori. Reasoning about Prolog Programs: from Modes through Types to Assertions. *Formal Aspects of Computing*, 6(6A):743–765, 1994.

[2] 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.

[3] H. Azzoune. Type Inference in Prolog. In E. Lusk and R. Overbeek, editors, *Proc. of the Ninth International Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, pages 258–277, Berlin, 1988. Springer-Verlag.

[4] R. Barbuti and R. Giacobazzi. A Bottom-up Polymorphic Type Inference in Logic Programming. *Science of Computer Programming*, 19(3):281–313, 1992.

[5] C. Beierle. Type Inferencing for Polymorphic Order-Sorted Logic Programs. In L. Sterling, editor, *12th International Conference on Logic Programming*, pages 765–779. MIT Press, 1995.

[6] A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics Approach: Theory and Applications. *Journal of Logic Programming*, 19-20:149–197, 1994.

[7] J. Boye. *Directional Types in Logic Programming.* PhD thesis, Linköping University (Sweden), 1996.

[8] M. Codish and B. Demoen. Deriving Polymorphic Type Dependencies for Logic Programs Using Multiple Incarnations of Prop. In *Proc. of the first International Symposium on Static Analysis*, volume 864 of *Lecture Notes in Computer Science*, pages 281–296. Springer-Verlag, 1994.

[9] M. Codish and V. Lagoon. Type Dependencies for Logic Programs Using ACI-Unification. *Theoretical Computer Science*, 238:131–159, 2000.

[10] A. Cortesi, G. Filè, and W. Winsborough. *Prop* Revisited: Propositional Formula as Abstract Domain for Groundness Analysis. In *Proc. Sixth IEEE Symp. on Logic In Computer Science*, pages 322–327. IEEE Computer Society Press, 1991.

[11] A. Cortesi, G. Filè, and W. Winsborough. Optimal Groundness Analysis Using Propositional Logic. *Journal of Logic Programming*, 27(2):137–167, 1996.

[12] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. 6th ACM Symp. on Principles of Programming Languages*, pages 269–282, 1979.

[13] P. Cousot and R. Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2 & 3):103–179, 1992.

[14] N. J. Cutland. *Computability: An Introduction to Recursive Function Theory.* Cambridge University Press, 1980.

[15] G. Filé, R. Giacobazzi, and F. Ranzato. A Unifying View on Abstract Domain Design. *ACM Computing Surveys*, 28(2):333–336, 1996.

[16] T. Frühwirth, E. Shapiro, M. Y. Varai, and E. Yardeni. Logic Programs as Types for Logic Programs. In A. R. Meyer, editor, *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science*, pages 300–309, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[17] M. Gabbrielli and M. C. Meo. Fixpoint Semantics for Partial Computed Answer Substitutions and Call Patterns. In H. Kirchner and G. Levi, editors, *Algebraic and Logic Programming, Proceedings of the 3rd International Conference*, volume 632 of *Lecture Notes in Computer Science*, pages 84–99. Springer-Verlag, 1992.

[18] J. Gallagher and D. A. de Waal. Fast and Precise Regular Approximation of Logic Programs. In P. Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613, Santa Margherita Ligure, Italy, 1994. The MIT Press.

[19] R. Giacobazzi and F. Ranzato. Refining and Compressing Abstract Domains. In *Proc. of the ICALP'97 Conf.*, volume 1256 of *LNCS*, pages 771–781. Springer-Verlag, 1997.

[20] R. Giacobazzi and R. Ranzato. The Reduced Relative Power Operation on Abstract Domains. *Theoretical Computer Science*, 216:159–211, 1999.

[21] R. Giacobazzi and F. Scozzari. A Logical Model for Relational Abstract Domains. *ACM Transactions on Programming Languages and Systems*, 20(5):1067–1109, 1998.

[22] M. Hanus. Logic Programming with Type Specification. In F. Pfenning, editor, *Types in Logic Programming*. MIT Press, 1992.

[23] M. Hermenegildo, W. Warren, and S. K. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programing*, 13(2 & 3):349–366, 1992.

[24] P. M. Hill and J. W. Lloyd. *The Gödel Programming Language*. The MIT Press, 1994.

[25] P. M. Hill and F. Spoto. Analysis of Downward Closed Properties of Logic Programs. In T. Rus, editor, *Proc. of the International Conference on Algebraic Methodology and Software Technology, AMAST 2000*, volume 1816 of *Lecture Notes in Computer Science*, pages 181–196, Iowa City, Iowa, USA, May 2000. Springer-Verlag.

[26] P. M. Hill and R. W. Topor. A Semantics for Typed Logic Programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 1–62. MIT Press, 1992.

[27] ISO/IEC. *ISO/IEC 13211-1: 1995 Information Technology — Programming Languages — Prolog — Part 1: General Core*. International Standard Organization, 1995.

[28] G. Janssens and M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by means of Abstract Interpretation. *Journal of Logic Programming*, 13(2 & 3):205–258, 1992.

[29] T. Kanamori and K. Horiuchi. Polymorphic Type Inference in Prolog by Abstract Interpretation. In *Logic Programming 87 - Tokyo*, volume 315 of *Lecture Notes in Computer Science*, pages 195–214. Springer-Verlag, 1988.

[30] M. Kifer and J. Wu. A First–Order Theory of Types and Polymorphism in Logic Programming. In *IEEE Symposium on Logic in Computer Science*, 1991.

[31] T. K. Lakshman and U. S. Reddy. Typed Prolog: A Semantic Reconstruction of the Mycroft-O'Keefe Type System. Technical report, Dept. of Computer Science, University of Illinois, 1991.

[32] G. Levi and F. Spoto. An Experiment in Domain Refinement: Type Domains and Type Representations for Logic Programs. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Principles of Declarative Programming*, volume 1490 of *Lecture Notes in Computer Science*, pages 152–169, Pisa, Italy, September 1998. ©Springer-Verlag.

[33] L. Lu. A Polymorphic Type Analysis in Logic Programs by Abstract Interpretation. *Journal of Logic Programming*, 36(1):1–54, 1998.

[34] 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.

[35] A. Martelli and U. Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

[36] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[37] A. Mycroft and R. A. O'Keefe. A Polymorphic Type System for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[38] C. Palamidessi. Algebraic Properties of Idempotent Substitutions. In M. S. Paterson, editor, *Proc. of the 17th International Colloquium on Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 386–399. Springer-Verlag, 1990.

[39] C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[40] C. Pyo and U. S. Reddy. Inference of Polymorphic Types for Logic Programming. In E. Lusk and R. Overbeek, editors, *Proc. North American Conf. on Logic Programming'89*, pages 1115–1132. The MIT Press, 1989.

[41] U. S. Reddy. Notions of Polymorphism for Predicate Logic Programs. In R. A. Kowalski and K. A. Bowen, editors, *Fifth International Conference on Logic Programming,* Seattle. MIT Press, 1988. Printed in separate booklet to main proceedings.

[42] O. Ridoux, P. Boizumault, and F. Malésieux. Typed Static Analysis Application to Groundness Analysis of Prolog and λProlog. In A. Middeldorp and T. Sato, editors, *Proc. of the 4th Fuji Int. Symp. on Functional and Logic Programming, FLOPS'99*, volume 1722 of *Lecture Notes in Computer Science*, pages 267–283, Tsukuba, Japan, November 1999. Springer-Verlag.

[43] Y. Rouzard and L. Nguyen-Phong. Integrating Modes and Subtypes into a Prolog Type-Checker. In K. Apt, editor, *Proceedings of the Joint International Conference on Logic Programming,* Washington, USA, pages 87–97. MIT Press, 1992.

[44] F. Scozzari. Logical Optimality of Groundness Analysis. In P. Van Hentenryck, editor, *Proceedings of the 4th International Static Analysis Symposium SAS'97*, volume 1302 of *Lecture Notes in Computer Science*, pages 83–97. Springer-Verlag, 1997.

[45] J.-G. Smaus, P. M. Hill, and A. King. Mode Analysis Domains for Typed Logic Programs. In A. Bossi, editor, *Logic-Based Program Synthesis and Transformation: Proceedings of the 9th International Workshop, LOPSTR-99,* volume 1817 of *Lecture Notes in Computer Science*, pages 82–101, Venice, Italy, 1999. Springer-Verlag.

[46] Z. Somogyi, F. Henderson, and T. Conway. Mercury: An Efficient Purely Declarative Logic Programming Language. In *Proceedings of the Australian Computer Science Conference*, pages 499–512, Glenelg, Australia,

1995. This paper and up-to-date information on Mercury is available from `http://www.cs.mu.oz.au/research/mercury/`.

[47] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type Analysis of Prolog Using Type Graphs. *Journal of Logic Programming*, 22(3):179–209, 1995.

[48] J. Xu and D. S. Warren. A Type Inference System for Prolog. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth International Conf. on Logic Programming*, pages 604–619. The MIT Press, 1988.

[49] E. Yardeni and E. Shapiro. A Type System for Logic Programs. *Journal of Logic Programming*, 10:125–135, 1991.

8 Proofs

This appendix contains the proofs of the propositional statements in the main text, introduced by the corresponding environment in slanted font. Non-slanted environments are local definitions and results.

Proofs of Section 2

Proposition 7. This is a corollary of Definition 6, since it follows by structural induction on a goal G that $\mathcal{CA}^{D_V}[\![]\!]$ is additive:

$$\mathcal{CA}^{D_V}[\![G]\!]\left(\cup_{j\in J}\{I_j\}\right) = \cup_{j\in J}\left(\mathcal{CA}^{D_V}[\![G]\!]I_j\right)$$

for every $\{I_j\}_{j\in J} \subseteq \mathbb{I}^{D_V}$ with $J \subseteq \mathbb{N}$. Since, by Definition 5, $\exists^{D_V}_{V\setminus\tilde{\iota}}$ and $\mathsf{rename}^{D_V}_{\tilde{\iota}\to\tilde{x}}$ are additive, we conclude that T_P is additive. □

Proposition 9. Since $\theta \in \mathsf{sol}_V(\mathsf{Eq}(\theta))$ and $\mathsf{sol}_V(\mathsf{Eq}(\theta))$ is downward closed, we conclude that $\mathsf{sol}_V(\mathsf{Eq}(\theta)) \supseteq {\downarrow}\{\theta\}$. To show that $\mathsf{sol}_V(\mathsf{Eq}(\theta)) \subseteq {\downarrow}\{\theta\}$, suppose $\sigma \in \mathsf{sol}_V(\mathsf{Eq}(\theta))$. This means that $\mathsf{Eq}(\theta)\sigma$ is true, i.e., that $\sigma(x) = \theta(x)\sigma$ for every $x \in \mathsf{dom}(\theta)$. This entails that $\sigma = \theta \circ \sigma$, i.e., $\sigma \leq \theta$. Therefore, $\sigma \in {\downarrow}\{\theta\}$. □

Proofs of Section 3

We recall that a substitution θ is called *grounding* for a set of variables G if and only if $\theta(x)$ is ground for every $x \in G$.

Proposition 66. *Given $S \in \wp{\downarrow}(\Theta_V)$ and $x \in V$, we have*

$$\exists^{Down_V}_x S = {\downarrow}(S'|_{V\setminus x}) \ ,$$

where

$$S' = \left\{ \theta \circ \{x \mapsto u\} \in \Theta_V^{\{x\}} \,\middle|\, u \in \mathsf{terms}(\Sigma, V) \text{ and } \theta \in S \right\} \ .$$

Proof. Let $\theta' \in \exists^{Down_V}_x S$. By definition, we have $\theta' = \theta|_V$, $\theta \leq_{V\cup n} \sigma$, $\sigma = \sigma'[n/x]$ and $\sigma' \in S$ for suitable θ, σ and σ'. Hence $\theta = \sigma \circ \sigma''$ for a suitable σ''. Since $\sigma \circ \sigma''$ is idempotent, $\sigma' \circ (\sigma''|_{V\setminus x}) \in \Theta_V$ is also idempotent. Moreover, it belongs to S by downward closure and we have:

$$\theta|_{V\setminus x} = \left((\sigma' \circ (\sigma''|_{V\setminus x})) \circ \{x \mapsto \sigma''(n)\}\right)|_{V\setminus x} \in S'|_{V\setminus x} \ .$$

This is because, given $y \in V \setminus x$, we have

$$\begin{aligned}
\theta|_{V\setminus x}(y) &= (\sigma \circ \sigma'')|_{V\setminus x}(y) = (\sigma'[n/x] \circ \sigma'')|_{V\setminus x}(y) \\
&= \sigma'[n/x](y)\sigma'' = \sigma'(y)[n/x]\sigma'' \\
&= (\sigma'(y)(\sigma''|_{V\setminus x}))\{x \mapsto \sigma''(n)\} \\
&= \left(((\sigma' \circ \sigma'')|_{V\setminus x}) \circ \{x \mapsto \sigma''(n)\}\right)(y) \\
&= \left((\sigma' \circ (\sigma''|_{V\setminus x})) \circ \{x \mapsto \sigma''(n)\}\right)|_{V\setminus x}(y) \ .
\end{aligned}$$

Therefore,

$$\theta = \theta|_{V \setminus x} \circ \{x \mapsto \theta(x)\} \in \downarrow(S'|_{V \setminus x}) \ .$$

Assume now $\theta' \in \downarrow(S'|_{V\setminus x})$. We have $\theta' \leq \theta''$ for a suitable $\theta'' \in S'|_{V\setminus x}$, i.e., $\theta' = \theta'' \circ \sigma$ for a suitable $\sigma \in \Theta_V^V$ and $\theta'' = \theta'''|_{V\setminus x}$ for a suitable $\theta''' \in S'$. This means that $\theta''' = \theta \circ \{x \mapsto u\}$ for a suitable $\theta \in S$ and $u \in \mathsf{terms}(\Sigma, V)$. We have $\theta'' = \theta'''|_{V\setminus x} = (\theta[n/x] \circ \{n \mapsto u\})|_V$ and

$$\theta' = \theta'' \circ \sigma = (\theta[n/x] \circ \{n \mapsto u\})|_V \circ \sigma = (\theta[n/x] \circ (\{n \mapsto u\} \circ \sigma))|_V \ .$$

Therefore, $\theta' \in \exists_x^{Down_V} S$. $\square$

Lemma 67. *Let $V \in \wp_f(\mathcal{V})$, $\tilde{x}, \tilde{y} \in \mathrm{Seq}(V)$ disjoint, of the same length and without repetitions and $\sigma_1, \sigma_2 \in \Theta_V$. Then*

$$\sigma_1 \leq \sigma_2 \textit{ if and only if } \sigma_1[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}] \leq \sigma_2[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}] \ .$$

Proof. Assume $\sigma_1 \leq \sigma_2$, i.e., $\sigma_1 = \sigma_2\sigma$ for a suitable σ. We show that $\sigma_1[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}] = (\sigma_2[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}])(\sigma[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}])$. Assume $v \notin \tilde{x} \cup \tilde{y}$. We have

$$\begin{aligned}
\sigma_1[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}](v) &= (\sigma_2\sigma)[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}](v) \\
(v \notin \tilde{x} \cup \tilde{y}) &= (\sigma_2(v)\sigma)[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}] = \sigma_2(v)[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}](\sigma[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]) \\
(v \notin \tilde{x} \cup \tilde{y}) &= \sigma_2[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}](v)(\sigma[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]) \\
&= ((\sigma_2[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}])(\sigma[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]))(v) \ .
\end{aligned}$$

Moreover, we have

$$\begin{aligned}
\sigma_1[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}](x_i) &= (\sigma_2\sigma)[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}](x_i) = ((\sigma_2\sigma)(y_i))[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}] \\
&= (\sigma_2(y_i)\sigma)[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}] = \sigma_2(y_i)[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}](\sigma[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]) \\
&= (\sigma_2[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}](x_i))(\sigma[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]) \\
&= ((\sigma_2[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}])(\sigma[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]))(x_i) \ .
\end{aligned}$$

The case for y_i is similar.

The converse holds since $\sigma[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}][\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}] = \sigma$ for every $\sigma \in \Theta_V$. $\square$

Proposition 17.

i) Intersection is closed on the set of downward closed sets of substitutions.

ii) Proposition 66 shows that it is the downward closure of a set of substitutions.

iii) By using the notation of Proposition 66, we have $S_1' = \{\theta \circ \{x \mapsto u\} \in \Theta_V^{\{x\}} \mid u \in \mathsf{terms}(\Sigma, V) \text{ and } \theta \in S_1\} \subseteq \{\theta \circ \{x \mapsto u\} \in \Theta_V^{\{x\}} \mid u \in \mathsf{terms}(\Sigma, V) \text{ and } \theta \in S_2\} = S_2'$. Therefore, we have $\downarrow(S_1'|_{V\setminus x}) \subseteq \downarrow(S_2'|_{V\setminus x})$ and by Proposition 66 we have the thesis.

iv) Let S' be as in Proposition 66. Let $\theta \in S$. Choosing $u = x$, we have $\theta \in S'$. If $x \notin \mathsf{dom}(\theta)$ then $\theta|_{V\setminus x} = \theta$ and $\theta \in \downarrow(S'|_{V\setminus x})$. If $x \in \mathsf{dom}(\theta)$ then $\theta = \theta|_{V\setminus x} \circ \{x \mapsto \theta(x)\}$. Therefore, even in this case we have $\theta \in \downarrow(S'|_{V\setminus x})$. By Proposition 66 we have the thesis.

v) Let $\sigma_2 \in \mathsf{rename}^{Down_V}_{\tilde{x}\to\tilde{y}}(S)$ and $\sigma_1 \leq \sigma_2$. Since $\sigma_2 = \sigma_2'[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]$ for some $\sigma_2' \in S$, by Lemma 67 we have $\sigma_1[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}] \leq \sigma_2[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}] = \sigma_2'$. From $S = \downarrow S$ we conclude that $\sigma_1[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}] \in S$. Then $\sigma_1 \in \mathsf{rename}^{Down_V}_{\tilde{x}\to\tilde{y}}(S)$.

□

Proposition 18. We assume the constant in Σ to be $\mathtt{a}$ and the functor to be the unary functor $\mathtt{f}$, though the proof can be easily generalised to greater arities.

The non-strict inclusion is a consequence of the fact that $\mathsf{sol}_V(\exists_W c)$ is a downward closed set of substitutions. The strict inclusion follows from the fact that every set on the left is recursive [14], while some sets on the right are not. Indeed, to check if a substitution $\theta \in \Theta_V$ is a solution of a given existential Herbrand constraint $\exists_W c$, it suffices to check whether $\exists_W(c\theta)$ admits a solution or not. This, in turn, can be checked with the Martelli and Montanari unification algorithm [35] applied to $c\theta$. On the contrary, there are downward closed sets of substitutions which are not recursive. Indeed, given a Turing machine M, seen as a partial map $M : \mathbb{N} \to \mathbb{N}$, such that $M(i)$ is defined if and only if the machine M terminates on input i, yielding the result $M(i)$, we can define the downward closed set of substitutions:

$$S = \{\theta \in \Theta_V \mid \theta(x) = \mathtt{f}^i(\mathtt{a}) \text{ and } M(i) \text{ is defined}\} ,$$

for a given variable $x \in V$. Given $i \in \mathbb{N}$, $\{x \mapsto \mathtt{f}^i(\mathtt{a})\} \in S$ if and only if $M(i)$ terminates. Since the halting problem for Turing machines is undecidable [14], we conclude that S is not recursive. □

Proposition 19.i. Assume $h_1 = \exists_{W_1} c_1$ and $h_2 = \exists_{W_2} c_2$ with $W_1 \cap W_2 = \emptyset$.

Let $\theta' \in \mathsf{sol}_V(h_1 \star^{H_V} h_2)$. Then $\theta' = \theta|_V$, $\theta \in \Theta_{V\cup W_1 \cup W_2, V}$ and $c_1\theta$ and $c_2\theta$ are true. Hence, we have $\theta|_{V\cup W_i} \in \Theta_{V\cup W_i, V}$ and $c_i\theta|_{V\cup W_i}$ is true for $i = 1, 2$. Therefore, $\theta' = (\theta|_{V\cup W_i})|_V$ belongs to $\mathsf{sol}_V(h_i)$ for $i = 1, 2$. This means that $\theta' \in \mathsf{sol}_V(h_1) \cap \mathsf{sol}_V(h_2) = \mathsf{sol}_V(h_1) \star^{Down_V} \mathsf{sol}_V(h_2)$.

Conversely, let $\theta' \in \mathsf{sol}_V(h_1) \star^{Down_V} \mathsf{sol}_V(h_2) = \mathsf{sol}_V(h_1) \cap \mathsf{sol}_V(h_2)$. Hence there exist $\theta_1 \in \Theta_{V\cup W_1, V}$ and $\theta_2 \in \Theta_{V\cup W_2, V}$ such that $\theta_i|_V = \theta'$ and $c_i\theta_i$ is true for $i = 1, 2$. Since θ_1 and θ_2 coincide on the variables in V and existential variables are standardised apart, we can define $\theta = \theta' \circ \theta_1 \circ \theta_2$ which is such that $\theta \in \Theta_{V\cup W_1\cup W_2, V}$ and $c_i\theta$ is true for $i = 1, 2$. Therefore, $\theta' = \theta|_V \in \mathsf{sol}_V(h_1 \star^{H_V} h_2)$. □

Proposition 19.ii. We have to prove that

$$\exists_x^{Down_V} \mathsf{sol}_V(\exists_W c) = \mathsf{sol}_V(\exists_{W\cup N} c[N/x])$$

for every $x \in V$.

Let $\theta' \in \exists_x^{Down_V}(\mathsf{sol}_V(\exists_W c))$. We have, by definition, $\theta' = \theta''|_V$ with $\theta'' \leq_{V\cup n} \sigma$ and $\sigma \in (\mathsf{sol}_V(\exists_W c))[n/x]$. Let $\theta'' = \sigma \circ \rho$ with $\rho \in \Theta^{V\cup n}_{V\cup n}$. We have $\sigma = \sigma'[n/x]$ with $\sigma' \in \mathsf{sol}_V(\exists_W c)$ and $\sigma' = \sigma''|_V$ with $c\sigma''$ true and $\sigma'' \in \Theta_{V\cup W, V}$. The substitution $\sigma''' = \sigma'' \circ \{N \mapsto \sigma''(x)\}$ is such that $c[N/x]\sigma'''$ is true, i.e., for all $t_1 = t_2 \in c$, we have $t_1[N/x]\sigma''' = t_2[N/x]\sigma'''$. Since $c[N/x]$ contains neither x nor n, we have $t_1[N/x](\sigma'''[n/x]) = t_2[N/x](\sigma'''[n/x])$ and $(\sigma'''[n/x])|_{V\cup n} = \sigma$. This allows us to conclude that $\sigma'''[n/x] \circ \rho$ is idempotent, since $\sigma \circ \rho$ is. Moreover, $t_1[N/x](\sigma'''[n/x] \circ \rho) = t_2[N/x](\sigma'''[n/x] \circ \rho)$. Finally, we have:

$$\theta' = \theta''|_V = (\sigma \circ \rho)|_V = (\sigma'''[n/x] \circ \rho)|_V \in \mathsf{sol}(\exists_{W \cup N} c[N/x]) \ .$$

Assume, conversely, that $\theta \in \mathsf{sol}_V(\exists_{W \cup N} c[N/x])$. Then there exists $\theta' \in \Theta_{V \cup W \cup N, V}$ such that $\theta = \theta'|_V$ and $c[N/x]\theta'$ is true. Thus $\mathsf{dom}(c) \setminus x \subseteq \mathsf{dom}(\theta')$ and $\mathsf{rng}(\theta') \cap \mathsf{dom}(c) \subseteq \mathsf{rng}(\theta') \cap (\mathsf{dom}(\theta) \cup x) \subseteq \{x\}$. We have

$$\begin{aligned}\theta = (c[N/x]\theta'|_{W \cup N})|_V \circ \theta'|_V &= ((c(\theta'|_W))|_V \circ \{x \mapsto \theta'(N)\})\,|_{V \setminus x}\theta'|_V \\ &\leq ((c(\theta'|_W))|_V \circ \{x \mapsto \theta'(N)\})\,|_{V \setminus x} \ .\end{aligned}$$

Since $c(c(\theta'|_W))$ is true and $c(\theta'|_W) \in \Theta_{V \cup W, V}$, we conclude that $(c(\theta'|_W))|_V \in \mathsf{sol}_V(h)$. Moreover, letting $\theta'' = (c(\theta'|_W))|_V \circ \{x \mapsto \theta'(N)\}$, we have

$$\begin{aligned}\mathsf{dom}(\theta'') \cap \mathsf{rng}(\theta'') &\subseteq ((\mathsf{dom}(c) \cap V) \cup \{x\}) \cap (\mathsf{rng}(c) \cup \mathsf{rng}(\theta')) \\ &\subseteq (\mathsf{dom}(c) \cap V \cap \mathsf{rng}(c)) \cup (\mathsf{dom}(c) \cap V \cap \mathsf{rng}(\theta')) \cup \\ &\quad \cup (\{x\} \cap (\mathsf{rng}(x) \cup \mathsf{rng}(\theta'))) \subseteq \{x\} \ .\end{aligned}$$

The thesis follows by Proposition 66. $\square$

Lemma 68. *Let $V \in \wp_f(\mathcal{V})$, $W \in \wp_f(\mathcal{W})$, $\tilde{x}, \tilde{y} \in \mathrm{Seq}(V)$ disjoint, of the same length and without repetitions, $t_1, t_2 \in \mathsf{terms}(\Sigma, (V \setminus \tilde{y}) \cup W)$ and $\theta \in \theta_{V \cup W, V}$. Then $t_1\theta = t_2\theta$ if and only if $t_1[\tilde{y}/\tilde{x}](\theta[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]) = t_2[\tilde{y}/\tilde{x}](\theta[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}])$.*

Proof. Let $t_1\theta = t_2\theta$. Assume, without any loss of generality, that t_1 has depth no greater than the depth of t_2. We prove that $t_1[\tilde{y}/\tilde{x}](\theta[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]) = t_2[\tilde{y}/\tilde{x}](\theta[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}])$ by induction on the depth of t_1. If $t_1 = v \notin \tilde{x} \cup \tilde{y}$ we have

$$\begin{aligned}t_1[\tilde{y}/\tilde{x}](\theta[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]) &= v(\theta[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]) = \theta(v)[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}] \\ &= (t_2\theta)[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}] = (t_2[\tilde{y}/\tilde{x}])(\theta[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]) \ .\end{aligned}$$

If $t_1 = x_i$ we have

$$\begin{aligned}t_1[\tilde{y}/\tilde{x}](\theta[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]) &= y_i(\theta[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]) = \theta(x_i)[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}] \\ &= (t_2\theta)[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}] = (t_2[\tilde{y}/\tilde{x}])(\theta[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]) \ .\end{aligned}$$

Note that the case $t_1 = y_i$ is impossible since $t_1 \in \mathsf{terms}(\Sigma, (V \setminus \tilde{y}) \cup W)$. If $t_1 = \mathtt{f}(t_1^1, \ldots, t_1^l)$ then $t_2 = \mathtt{f}(t_2^1, \ldots, t_2^l)$ and $t_1^j\theta = t_2^j\theta$ for every $j = 1, \ldots, l$. Thus we have

$$\begin{aligned}t_1[\tilde{y}/\tilde{x}](\theta[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]) &= \mathtt{f}(t_1^1[\tilde{y}/\tilde{x}](\theta[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]), \ldots, t_1^l[\tilde{y}/\tilde{x}](\theta[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}])) \\ \text{(by ind. hyp.)} &= \mathtt{f}(t_2^1[\tilde{y}/\tilde{x}](\theta[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]), \ldots, t_2^l[\tilde{y}/\tilde{x}](\theta[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}])) \\ &= t_2[\tilde{y}/\tilde{x}](\theta[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]) \ .\end{aligned}$$

The converse holds since $t[\tilde{y}/\tilde{x}][\tilde{x}/\tilde{y}] = t$ for every $t \in \mathsf{terms}(\Sigma, (V \setminus \tilde{y}) \cup W)$ and $\theta[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}][\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}] = \theta$ for every $\theta \in \Theta_{V \cup W, V}$. $\square$

Proposition 19.iii. Let $h = \exists_W c$. Let $\theta \in \mathsf{rename}_{\tilde{x} \to \tilde{y}}^{Down_V} \mathsf{sol}_V(h)$. We have $\theta = \theta'[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]$ with $\theta' = \theta''|_V$, $\theta'' \in \Theta_{V \cup W, V}$ and $c\theta''$ is true. By Lemma 68 we have that $c[\tilde{y}/\tilde{x}](\theta''[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}])$ is true. Since $\theta'[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}] = \theta''[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]|_V$, we have $\theta'[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}] \in \mathsf{sol}_V(\mathsf{rename}_{\tilde{x} \to \tilde{y}}^{H_V}(h))$, i.e., $\theta \in \mathsf{sol}_V(\mathsf{rename}_{\tilde{x} \to \tilde{y}}^{H_V}(h))$.

Conversely, let $\theta \in \mathsf{sol}_V(\mathsf{rename}_{\tilde{x} \to \tilde{y}}^{H_V}(h))$. Then $\theta = \theta''|_V$ with $\theta'' \in \Theta_{V \cup W, V}$ and $c[\tilde{y}/\tilde{x}]\theta''$ is true. By Lemma 68 we have that $c(\theta''[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}])$ is true. Then $\theta''[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]|_V = \theta[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}] \in \mathsf{sol}_V(h)$ and $\theta \in \mathsf{rename}_{\tilde{x} \to \tilde{y}}^{Down_V} \mathsf{sol}_V(h)$. $\square$

Proposition 22. Since $\wp(Sol_V)$ and $Down_V$ are set-theoretic lattices, union is their least upper bound operator. Therefore,

$$\cup\left(\sqcup^{\wp(Sol_V)}\{p_i \mid i \in I\}\right) = \cup\left(\cup p_i \mid i \in I\right) = \sqcup_{i\in I}^{Down_V}\left(\cup p_i\right) \ .$$

□

Proposition 23.

i)

$$\begin{aligned}\cup(p_1 \star^{\wp(Sol_V)} p_2) &= \cup\{s_1 \star^{Sol_V} s_2 \mid s_1 \in p_1 \text{ and } s_2 \in p_2\}\\ &= \cup\{s_1 \cap s_2 \mid s_1 \in p_1 \text{ and } s_2 \in p_2\}\\ &= (\cup p_1) \cap (\cup p_2) = (\cup p_1) \star^{Down_V} (\cup p_2) \ .\end{aligned}$$

ii)

$$\begin{aligned}\cup\left(\exists_x^{\wp(Sol_V)} p\right) &= \cup\left\{\exists_x^{Sol_V} s \,\middle|\, s \in p\right\} = \cup\left\{\exists_x^{Down_V} s \,\middle|\, s \in p\right\}\\ &= \exists_x^{Down_V}(\cup\{s \mid s \in p\}) = \exists_x^{Down_V}(\cup p) \ .\end{aligned}$$

iii)

$$\begin{aligned}\cup\left(\mathsf{rename}_{\tilde{x}\to\tilde{y}}^{\wp(Sol_V)} p\right) &= \cup\left\{\mathsf{rename}_{\tilde{x}\to\tilde{y}}^{Sol_V} s \,\middle|\, s \in p\right\}\\ &= \cup\left\{\mathsf{rename}_{\tilde{x}\to\tilde{y}}^{Down_V} s \,\middle|\, s \in p\right\}\\ &= \mathsf{rename}_{\tilde{x}\to\tilde{y}}^{Down_V}(\cup\{s \mid s \in p\}) = \mathsf{rename}_{\tilde{x}\to\tilde{y}}^{Down_V}(\cup p) \ .\end{aligned}$$

□

Proofs of Section 4

Definition 69. Given a type constraint system $T = \{T_V\}_{V\in\wp_f(\mathcal{V})}$ for the type system $\langle\Delta, \Sigma, I\rangle$ and $\{\theta_1, \theta_2\} \subseteq \Theta_V$, θ_1 and θ_2 are *type-equivalent* if and only if for every $v \in V$ and every $d \in \mathsf{terms}(\Delta, \emptyset)$ we have $\theta_1(v) \in [\![d]\!]I$ if and only if $\theta_2(v) \in [\![d]\!]I$.

The importance of type-equivalent substitutions is that they are indistinguishable by the evaluation of any transfinite formula.

Proposition 70. *Given a type constraint system $T = \{T_V\}_{V\in\wp_f(\mathcal{V})}$ for the type system $\langle\Delta, \Sigma, I\rangle$, two type-equivalent substitutions θ_1 and θ_2 in Θ_V and $\phi \in \Phi_{\Delta,V}$, we have $[\![\phi]\!]\theta_1 = [\![\phi]\!]\theta_2$.*

Proof. By induction on the structure of transfinite formulas. □

Proposition 32. Let $\theta \in \Theta_V$. We define

$$t' = \begin{cases}\theta(y) & \text{if } x \notin \mathsf{vars}(\theta(y))\\ \theta(y)\{x \mapsto y\} & \text{otherwise.}\end{cases}$$

Let $\theta' = \{x \mapsto t'\} \circ (\theta|_{V \setminus \{x,y\}})$. Then θ' is idempotent and coincides with θ on $V \setminus \{x, y\}$. Moreover, $\theta'(x)$ is type-equivalent to $\theta(y)$. Indeed, $\theta'(x) = \theta(y)$ if $x \notin \mathsf{vars}(\theta(y))$, and $\theta'(x) = \theta(y)\{x \mapsto y\}$ otherwise. In this latter case, since $x \in \mathsf{vars}(\theta(y))$, we have $y \notin \mathsf{vars}(\theta(y))$. Thus, for every $d \in \mathsf{terms}(\Delta, \emptyset)$ we have that if $\theta'(x) \in [\![d]\!]I$ then $\theta(y) = (\theta'(x))\{y \mapsto x\} \in [\![d]\!]I$. Conversely, if $\theta(y) \in [\![d]\!]I$ then $\theta'(x) = \theta(y)\{x \mapsto y\} \in [\![d]\!]I$. Thus we have $[\![\phi_1[y/x]]\!]\theta = [\![\phi_1]\!]\theta' = [\![\phi_2]\!]\theta' = [\![\phi_2[y/x]]\!]\theta$. Since θ was arbitrary, we have the thesis. $\square$

We rewrite the cylindrification operator of Definition 28 in a way which simplifies the following proofs.

Definition 71. Given $t \in \mathsf{terms}(\Sigma, V)$, we define

$$(x \in d)[t/x] = \begin{cases} true & \text{if } t \in [\![d]\!]I \\ false & \text{otherwise} \end{cases} \qquad (y \in d)[t/x] = (y \in d) \quad \text{if } x \neq y$$

$$(\wedge S)[t/x] = \wedge\{s[t/x] \mid s \in S\} \qquad (\vee S)[t/x] = \vee\{s[t/x] \mid s \in S\}$$

$$(\phi_1 \Rightarrow \phi_2)[t/x] = (\phi_1[t/x]) \Rightarrow (\phi_2[t/x]) \qquad (\neg\phi)[t/x] = \neg(\phi[t/x]) \,.$$

Proposition 72. *Given a type constraint system $T = \{T_V\}_{V \in \wp_f(\mathcal{V})}$ for a type system $\langle \Delta, \Sigma, I \rangle$, $V \in \wp_f(\mathcal{V})$ and $x \in V$,*

$$\exists_x^{T_V} \phi = \vee\{\phi[t/x] \mid t \in \mathsf{terms}(\Sigma, V)\}$$

for all $\phi \in T_V$.

Proof. For all $t \in \mathsf{terms}(\Sigma, V)$, let $P_t = \{d \in \mathsf{terms}(\Delta, \emptyset) \mid t \in [\![d]\!]I\}$. We have $\phi[t/x] \equiv \phi[P_t/x]$, which entails the thesis (the definition of $\phi[P_t/x]$ is given in Definition 28). $\square$

Proposition 36. For the co-additivity of γ, let $S \subseteq T_V$. We have

$$\begin{aligned} \gamma(\wedge S) =& \{\theta \in \Theta_V \mid \text{for all } \sigma \leq \theta \text{ we have } [\![\phi]\!]\sigma = 1 \text{ for all } \phi \in S\} \\ =& \bigcap_{\phi \in S} \{\theta \in \Theta_V \mid \text{for all } \sigma \leq \theta \text{ we have } [\![\phi]\!]\sigma = 1\} = \bigcap_{\phi \in S} \gamma(\phi) \,. \end{aligned}$$

Moreover, since $\wedge\emptyset \in T_V$ and $\gamma(\wedge\emptyset) = \Theta_V$, which is the top of $Down_V$, we conclude that $\gamma(T_V)$ is topped. Finally, since for all $S \subseteq T_V$ we know that $\wedge S \in T_V$, the above result about co-additivity entails that the set $\gamma(T_V)$ is completely $\cap$-closed. $\square$

Proposition 42. Given $V \in \wp_f(\mathcal{V})$ and $\{t_1, \ldots, t_n\} \subseteq \mathsf{terms}(\Sigma, V)$, consider $\sigma \in \Theta_V^V$ such that $\sigma(v) = t$ for any $v \in V$. For all $i = 1, \ldots, n$ and $d \in \mathsf{terms}(\Delta, \emptyset)$, if $t_i \in [\![d]\!]I$ then, by the downward closure of types, $t_i\sigma \in [\![d]\!]I$. Conversely, if $t_i\sigma \notin [\![d]\!]I$ we conclude that $t_i \notin [\![d]\!]I$ by the choice of t, since t_i is obtained from $t_i\sigma$ by substituting some instances of t with variables in $V \subset \mathcal{V}$. Therefore, T is structural. $\square$

Proposition 47. Let $T = \{T_V\}_{V \in \wp_f(\mathcal{V})}$ be a type constraint system for the type system $\langle \Delta, \Sigma, I \rangle$. Consider property P1 first. Let $\phi_1, \phi_2 \in T_V$ be such that $\phi_1 \equiv \phi_2$. Let $\theta \in \Theta_V$ and $t \in \mathsf{terms}(\Sigma, V)$.

If T is positive, let $\Delta = \{\mathsf{c}\}$ be its only type. If $[\![\mathsf{c}]\!]I \neq \emptyset$, let t' be a ground term in $[\![\mathsf{c}]\!]I$ (otherwise, we do not need such a term). Define θ' such that

$$\theta'(x) = \begin{cases} t' & \text{if } t \in [\![\mathsf{c}]\!]I \\ x & \text{otherwise} \end{cases} \qquad \theta'(v) = \begin{cases} t' & \text{if } \theta(v) \in [\![\mathsf{c}]\!]I \\ v & \text{otherwise} \end{cases}$$

for every $v \in V \setminus x$. By construction, θ' is idempotent and $\theta|_{V\setminus x}$ and $\theta'|_{V\setminus x}$ are type-equivalent. We have

$$\begin{aligned} [\![\phi_1[t/x]]\!]\theta &= [\![\phi_1[t/x]]\!]\theta' \\ \text{(by the choice of } t') &= [\![\phi_1]\!]\theta' \\ \text{(since } \phi_1 \equiv \phi_2) &= [\![\phi_2]\!]\theta' \\ \text{(as above)} &= [\![\phi_2[t/x]]\!]\theta \ . \end{aligned}$$

Therefore, $\phi_1[t/x] \equiv \phi_2[t/x]$.

If T is structural, we know that there exists a substitution θ'' grounding for V which is type-equivalent to θ. Consider $\theta' = \theta''|_{V\setminus x} \circ \{x \mapsto t'\}$, where t' is a ground term with the same type properties as t (we can find such a term since T is structural). The thesis follows as in the case above.

Consider now property P2. Given $\phi_1, \phi_2 \in T_V$ such that $\phi_1 \not\equiv \phi_2$, we want to show that $\gamma(\phi_1) \neq \gamma(\phi_2)$. Let θ be such that $[\![\phi_1]\!]\theta = 1$ and $[\![\phi_2]\!]\theta = 0$ (this is possible since $\phi_1 \not\equiv \phi_2$, and does not introduce any loss of generality). We will show that there exists a substitution θ' which is type-equivalent to θ and belongs to $\gamma(\phi_1)$. Since $[\![\phi_2]\!]\theta' = [\![\phi_2]\!]\theta = 0$ entails $\theta' \notin \gamma(\phi_2)$, this will entail the thesis.

If T is positive, let $\Delta = \{\mathsf{c}\}$. If $[\![\mathsf{c}]\!]I \neq \emptyset$, let t be a ground term in $[\![\mathsf{c}]\!]I$ (otherwise we do not need such a term). Let $z \in V$ arbitrary. We define

$$\theta'(v) = \begin{cases} t & \text{if } \theta(v) \in [\![\mathsf{c}]\!]I \\ z & \text{otherwise.} \end{cases}$$

By construction, θ' is idempotent and type-equivalent to θ. Moreover, every instance of θ' is type-equivalent to θ' or it binds every variable to a term in $[\![\mathsf{c}]\!]I$. Since ϕ_1 is positive, we have $\theta' \in \gamma(\phi_1)$, as required.

If T is structural, we know that there exists a substitution θ' grounding for V and type-equivalent to θ. Then $[\![\phi_1]\!]\theta' = 1$, and every instance of θ' is θ' itself. This entails that $\theta' \in \gamma(\phi_1)$, as required. $\square$

Proposition 48.i. We have to show that

$$\alpha\left(\gamma(\phi_1) \star^{Down_V} \gamma(\phi_2)\right) \leq \phi_1 \star^{T_V} \phi_2$$

for all $\{\phi_1, \phi_2\} \subseteq T_V$. Indeed

$$\begin{aligned} \alpha\left(\gamma(\phi_1) \star^{Down_V} \gamma(\phi_2)\right) &= \alpha\left(\gamma(\phi_1) \cap \gamma(\phi_2)\right) \\ \text{(Proposition 36)} &= \alpha(\gamma(\phi_1 \wedge \phi_2)) \\ (\alpha\gamma \text{ is reductive}) &\leq \phi_1 \wedge \phi_2 = \phi_1 \star^{T_V} \phi_2 \ . \end{aligned}$$

If property P2 holds then $\alpha\gamma$ is the identity map and the result holds with $=$ instead of $\leq$. $\square$

Proposition 48.ii. For the result about correctness, it suffices to show that

$$\exists_x^{Down_V}(\gamma(\phi)) \subseteq \gamma\left(\exists_x^{T_V}(\phi)\right) \ ,$$

since we can apply α to both sides of the equation above obtaining the thesis as a consequence of the monotonicity of α and the reductivity of $\alpha\gamma$.

Let $\theta \in \exists_x^{Down_V}(\gamma(\phi))$. Then there exists $\theta' \in \gamma(\phi)$ such that $\theta = (\theta'|_{V\setminus x} \circ \{x \mapsto u\}) \circ \sigma$, for suitable $\sigma \in \Theta_V^V$ and $u \in \mathsf{terms}(\Sigma, V)$ (Proposition 66). Then

$$\theta|_{V\setminus x} = \left((\theta'|_{V\setminus x} \circ \{x \mapsto u\}) \circ \sigma\right)|_{V\setminus x} = ((\theta' \circ \{x \mapsto u\}) \circ \sigma)\,|_{V\setminus x} \ .$$

Let $\theta'' = (\theta' \circ \{x \mapsto u\})\circ\sigma$. We have $\theta'' \leq \theta'$. Then $[\![\phi]\!]\theta'' = 1$, which entails that $[\![\phi[\theta''(x)/x]]\!](\theta''|_{V\setminus x}) = 1$. This means that $[\![\exists_x^{T_V}\phi]\!]\theta = 1$, because $\theta|_{V\setminus x} = \theta''_{V\setminus x}$. Since this is true for every $\theta \in \exists_x^{Down_V}(\gamma(\phi))$ and $\exists_x^{Down_V}(\gamma(\phi))$ is downward closed (Proposition 17.ii), we have the thesis.

Let T be positive or structural. Let θ be such that $[\![\exists_x^{T_V}\phi]\!]\theta = 1$. We will show that, if T is positive or structural, there exists a substitution θ' type-equivalent to θ such that $\theta' \in \exists_x^{Down_V}\gamma(\phi)$. By extensivity, it follows that $\theta' \in \gamma\alpha(\exists_x^{Down_V}\gamma(\phi))$ and hence $[\![\alpha(\exists_x^{Down_V}\gamma(\phi))]\!]\theta' = 1$. However, θ and θ' are type-equivalent so that $[\![\alpha(\exists_x^{Down_V}\gamma(\phi))]\!]\theta = 1$. Therefore, we conclude that $\exists_x^{T_V}\phi \leq \alpha(\exists_x^{Down_V}\gamma(\phi))$.

If T is positive, then let c^0 be its only type. Let t be a ground term in $[\![\mathsf{c}]\!]I$, whenever $[\![\mathsf{c}]\!]I$ is not empty (otherwise, we do not need such a t). Let $z \in V$ be arbitrary. Let us define

$$\theta'(v) = \begin{cases} t & \text{if } \theta(v) \in [\![\mathsf{c}]\!]I \\ z & \text{otherwise} \end{cases}$$

for every $v \in V$. By construction, θ' is idempotent and type-equivalent to θ. Then $[\![\exists_x^{T_V}\phi]\!]\theta' = 1$. Thus there exists a term t' such that $[\![\phi[t'/x]]\!]\theta' = 1$, i.e., $[\![\phi]\!](\theta'|_{V\setminus x} \circ \{x \mapsto t''\}) = 1$, where

$$t'' = \begin{cases} t & \text{if } t' \in [\![\mathsf{c}]\!]I \\ z & \text{otherwise.} \end{cases}$$

Let $\theta'' = \theta'|_{V\setminus x} \circ \{x \mapsto t''\}$. Every instance of θ'' is type-equivalent to θ'' or binds every variable to a term in $[\![\mathsf{c}]\!]I$. Since ϕ is positive, we have $\theta'' \in \gamma(\phi)$. Then $\theta' \in \exists_x^{Down_V}\gamma(\phi)$ (Proposition 66).

If T is structural, we know that there exists a grounding substitution θ' which is type-equivalent to θ. Then $[\![\exists_x^{T_V}\phi]\!]\theta' = 1$. Then there exists a term t' such that $[\![\phi[t'/x]]\!]\theta' = 1$, i.e., $[\![\phi]\!](\theta'|_{V\setminus x} \circ \{x \mapsto t''\}) = 1$, where t'' is a ground instance of t' with the same type properties as t' (we can find such a t'' since T is structural). Let $\theta'' = \theta'|_{V\setminus x} \circ \{x \mapsto t''\}$. It is grounding for V. Therefore, every instance of θ'' is θ'' itself and $\theta'' \in \gamma(\phi)$. Then $\theta' \in \exists_x^{Down_V}\gamma(\phi)$ (Proposition 66). □

Lemma 73. *Let $V \in \wp_f(V)$, $\tilde{x}, \tilde{y} \in \mathrm{Seq}(V)$ disjoint, of the same length and whose intersection is empty, $\mathsf{T} = \langle \Delta, \Sigma, I\rangle$ be a type system, $\phi \in \Phi_{\Delta,V}$ and $\theta \in \Theta_V$. We have*

$$[\![\phi]\!]_{\mathsf{T}}\theta = [\![\phi[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]]\!]_{\mathsf{T}}(\theta[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]) \ .$$

Proof. Let $\sigma = \{x_1 \mapsto y_1, \ldots, x_l \mapsto y_l, y_1 \mapsto x_1, \ldots, y_l \mapsto x_l\}$. Note that, for every $t \in \mathsf{terms}(\Sigma, V)$ and $d \in \mathsf{terms}(\Delta, \emptyset)$, we have $t \in [\![d]\!]I$ if and only if $t\sigma \in [\![d]\!]I$, since $t\sigma\sigma = t$. Let $v \notin \tilde{x} \cup \tilde{y}$. We have $\theta[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}](v) = \theta(v)[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}] = \theta(v)\sigma$. Moreover, we have $\theta[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}](x_i) = \theta(y_i)[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}] = \theta(y_i)\sigma$ and, similarly, $\theta[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}](y_i) = \theta(x_i)\sigma$. By the above mentioned property of σ, we have the thesis. □

Proposition 48.iii. We prove that, given $\tilde{x}, \tilde{y} \in \mathrm{Seq}(V)$ and $\phi \in \Phi_{\Delta, V \setminus \tilde{y}}$, we have

$$\mathsf{rename}^{Down_V}_{\tilde{x} \to \tilde{y}}\ \gamma(\phi) = \gamma(\mathsf{rename}^{T_V}_{\tilde{x} \to \tilde{y}}\ \phi)\ .$$

The thesis will follow by applying α to both sides of the equation above and from the reductivity of $\alpha\gamma$. Moreover, if property P2 holds for T_V, then $\alpha\gamma$ is the identity map.

Let $\theta \in \mathsf{rename}^{Down_V}_{\tilde{x} \to \tilde{y}}\ \gamma(\phi)$. Then $\theta = \theta'[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]$ with $\theta' \in \gamma(\phi)$. We have

$$\begin{aligned}
[\![\mathsf{rename}^{T_V}_{\tilde{x} \to \tilde{y}}\ \phi]\!]\theta &= [\![\phi[\tilde{y}/\tilde{x}]]\!]\theta \\
(\text{since } \phi \in \Phi_{\Delta, V \setminus \tilde{y}}) &= [\![\phi[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]]\!]\theta = [\![\phi[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]]\!](\theta'[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]) \\
(\text{Lemma 73}) &= [\![\phi]\!]\theta' = 1\ .
\end{aligned}$$

Since $\theta \le \theta'$ and $\mathsf{rename}^{Down_V}_{\tilde{x} \to \tilde{y}}\ \gamma(\phi)$ is downward closed (Proposition 17.v), we have $\theta \in \gamma(\mathsf{rename}^{T_V}_{\tilde{x} \to \tilde{y}}\ \phi)$.

Conversely, assume $\theta \in \gamma(\mathsf{rename}^{T_V}_{\tilde{x} \to \tilde{y}}\ \phi)$. Consider $\theta'' \le \theta[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]$, i.e., $\theta'' = \theta[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]\sigma$ for a suitable σ. We have $\theta'' = (\theta(\sigma[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]))[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]$ and, by Lemma 73, we have $[\![\phi]\!]\theta'' = [\![\phi[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]]\!](\theta(\sigma[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}])) = 1$ since $\theta(\sigma[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}]) \le \theta$ and $\theta \in \gamma(\phi[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}])$. We conclude that $\theta[\tilde{y}/\tilde{x}, \tilde{x}/\tilde{y}] \in \gamma(\phi)$, i.e., $\theta \in \mathsf{rename}^{Down_V}_{\tilde{x} \to \tilde{y}}\ \gamma(\phi)$. □

Proposition 49. Note that, given $h = \exists_W c \in H_V$ and $\theta \in \mathsf{sol}_V(h)$, then there exists $\theta' \in \Theta_{V \cup W, V}$ such that $\theta = \theta'|_V$ and $c\theta'$ is true. Then $c|_V\theta'$ is true, as well as $c|_V\theta'\theta'$, which means that $\theta \in \mathsf{sol}_V(\sigma)$, with $\sigma = c|_V\theta' \in \Theta_V$. This entails that, for every $h \in H_V$, we can find a suitable $\sigma \in \Theta_V$ such that $\mathsf{sol}_V(h) \subseteq \mathsf{sol}_V(\sigma)$. We use this result below. Namely, we have

$$\begin{aligned}
\gamma(\phi) &= \{\theta \in \Theta_V \mid \text{for all } \sigma \le \theta \text{ we have } [\![\phi]\!]\sigma = 1\} \\
&= \bigcup\{\mathsf{sol}_V(\theta) \mid \theta \in \Theta_V \text{ and } \mathsf{sol}_V(\theta) \subseteq \gamma(\phi)\} \\
&= \bigcup\{\mathsf{sol}_V(h) \mid h \in H_V \text{ and } \mathsf{sol}_V(h) \subseteq \gamma(\phi)\} \\
&= \bigcup\{\mathsf{sol}_V(h) \mid h \in I\}
\end{aligned}$$

where

$$I = \left\{ h \in H_V \;\middle|\; \begin{array}{l} \mathsf{sol}_V(h) \subseteq \gamma(\phi) \text{ and there does not exist } h' > h \\ \text{such that } \mathsf{sol}_V(h') \subseteq \gamma(\phi) \end{array} \right\}\ .$$

We want to prove that $I \subseteq H_{V \setminus \tilde{y}}$. Let $h = \exists_W c \in I$, and assume, by contradiction, that $\overline{y} \in \mathsf{dom}(c) \cup \mathsf{rng}(c)$, for some $\overline{y} \in \tilde{y}$. Let $h' = \exists_{W \cup N} c[N/\overline{y}]$. Note that $h' > h$. Indeed, by construction we have $\mathsf{sol}_V(h) \subseteq \mathsf{sol}_V(h')$. To show that that inclusion is strict, let $\sigma \in \Theta_{W \cup N}$ be such that $\sigma(v) = \mathsf{a}$ for every $v \in W \cup N$. We have $(c[N/\overline{y}]\sigma)|_V \in \mathsf{sol}_V(h')$, but $(c[N/\overline{y}]\sigma)|_V \notin \mathsf{sol}_V(h)$. This is because $\overline{y}$ does not occur in $(c[N/\overline{y}]\sigma)|_V$ which, therefore, cannot be a solution of $h = \exists_W c$, where $\overline{y}$ occurs.

By definition, we have $h' = \exists_{\overline{y}}^{H_V} h$. Therefore,

$$\begin{aligned} &\mathsf{sol}_V(h') \\ (\text{Prop. 19.ii}) &= \exists_{\overline{y}}^{Down_V} \mathsf{sol}_V(h) \\ (\text{since } \mathsf{sol}_V(h) \subseteq \gamma(\phi), \text{ by using Prop. 17.iii}) &\subseteq \exists_{\overline{y}}^{Down_V} \gamma(\phi) \\ (\text{Prop. 48.ii}) &\subseteq \gamma(\exists_{\overline{y}}^{T_V} \phi) \\ (\overline{y} \text{ does not occur in } \phi) &= \gamma(\phi) \ . \end{aligned}$$

In conclusion, $h' > h$ and $\mathsf{sol}_V(h') \subseteq \gamma(\phi)$, which contradicts the choice of h. $\square$

Proposition 51. For every $j \geq 0$ we have $\Theta_V \in Basic^j_{\mathsf{T},V}$ since $Basic^j_{\mathsf{T},V}$ is a Moore family of $Down_V$. Given $i \geq 0$ and $d \in Basic^i_{\mathsf{T},V}$, since d is downward closed, by Equation (1) of page 10 we have $\Theta_V \multimap d = d$, i.e., $d \in Basic^{i+1}_{\mathsf{T},V}$. $\square$

Proposition 74. *Let $\langle \Delta, \Sigma, I \rangle$ be a type system and $V \in \wp_f(\mathcal{V})$. Letting γ denote $\gamma_{\Phi_{\Delta,V}}$, we have*

i) $\gamma(v \in t) = \mathbf{v}_t$ *for all* $v \in V$ *and* $t \in \mathsf{terms}(\Delta, \emptyset)$.

ii) $\gamma(\vee(S)) = \cup_{\phi \in S} \gamma(\phi)$ *if* $\vee(S) \in Or_{\Delta,V}$.

iii) $\gamma(A \Rightarrow O) = \gamma(A) \multimap \gamma(O)$ *if* $A \in And_{\Delta,V}$ *and* $O \in Or_{\Delta,V}$.

Proof.

i)

$$\begin{aligned} \gamma(v \in t) &= \{\theta \in \Theta_V \mid \text{for all } \sigma \leq \theta \text{ we have } \sigma(x) \in [\![t]\!]I\} \\ ([\![t]\!]I \text{ is downward closed}) &= \{\theta \in \Theta_V \mid \theta(x) \in [\![t]\!]I\} = \mathbf{v}_t \ . \end{aligned}$$

ii) Note that, since $\vee(S) \in Or_{\Delta,V}$, every $\phi \in S$ has the form $x \in t$ for suitable $x \in V$ and $t \in \mathsf{terms}(\Delta, \emptyset)$. Therefore,

$$\begin{aligned} \gamma(\vee(S)) &= \left\{ \theta \in \Theta_V \;\middle|\; \begin{array}{l} \text{for all } \sigma \leq \theta \text{ there exists } \phi \in S \\ \text{such that } [\![\phi]\!]\sigma = 1 \end{array} \right\} \\ (\text{since } \phi = (x \in t)) &= \{\theta \in \Theta_V \mid \text{there exists } \phi \in S \text{ such that } [\![\phi]\!]\theta = 1\} \\ &= \bigcup_{\phi \in S} \{\theta \in \Theta_V \mid [\![\phi]\!]\theta = 1\} \\ (\text{since } \phi = (x \in t)) &= \bigcup_{\phi \in S} \{\theta \in \Theta_V \mid \text{for all } \sigma \leq \theta \text{ we have } [\![\phi]\!]\sigma = 1\} \\ &= \bigcup_{\phi \in S} \gamma(\phi) \ . \end{aligned}$$

iii)

$$\begin{aligned}
\gamma(A \Rightarrow O) &= \{\theta \in \Theta_V \mid \text{for all } \sigma \leq \theta \text{ if } [\![A]\!]\sigma = 1 \text{ then } [\![O]\!]\sigma = 1\} \\
\text{(Proposition 51)} &= \{\theta \in \Theta_V \mid [\![A]\!]\theta = 1\} \multimap \{\theta \in \Theta_V \mid [\![O]\!]\theta = 1\} \\
\left(\begin{array}{c} A \in And_{\Delta,V} \\ \text{and } O \in Or_{\Delta,V} \end{array}\right) &= \begin{array}{l} \{\theta \in \Theta_V \mid \text{for all } \sigma \leq \theta \text{ we have } [\![A]\!]\sigma = 1\} \multimap \\ \quad \multimap \{\theta \in \Theta_V \mid \text{for all } \sigma \leq \theta \text{ we have } [\![O]\!]\sigma = 1\} \end{array} \\
&= \gamma(A) \multimap \gamma(O) \ .
\end{aligned}$$

□

Proposition 53. By Proposition 74.i and iii and by Proposition 36, we conclude immediately that $Basic^1_{\Delta,V} = \gamma(Def_{\Delta,V})$.

In [44] it is shown that, provided Proposition 74 holds, the results which we state to be entailed by the structural nature of T hold if, letting $\{b_i\}_{i\in I}$, $\{c_j\}_{j\in J}$ and $\{d_k\}_{k\in K}$ contained in $\{\mathbf{v}_t \mid v \in V \text{ and } t \in \mathsf{terms}(\Delta, \emptyset)\}$, with $I, J, K \subset \mathbb{N}$, letting $B = \cap\{b_i\}_{i\in I}$, $C = \cup\{c_j\}_{j\in J}$, $D = \cup\{d_k\}_{k\in K}$ and letting $\theta \in \Theta_V$ be such that $\theta \notin B \cup C \cup D$, we have $\theta \notin (B \multimap C) \multimap D$. But this is true since, by Definition 39, we know that there exists $\sigma \in \Theta_V^V$ such that $\theta\sigma$ is grounding for V and θ is type-equivalent to $\theta\sigma$ (Definition 69). Therefore, every $\sigma' \leq \theta\sigma$ is such that $\sigma' = \theta\sigma$ and $\sigma' \notin B \cup D$. We conclude that $\theta\sigma \in (B \multimap C)$ and $\theta\sigma \notin D$. This means that $\theta\sigma \notin (B \multimap C) \multimap D$, i.e., $\theta \notin (B \multimap C) \multimap D$ since $(B \multimap C) \multimap D$ is downward closed. □

Proofs of Section 5

Proposition 58. Given $\{P_1, P_2\} \subseteq Prog^k_V$, we have

$$\gamma^k(P_1 \cup P_2) = \bigwedge_{c \in P_1 \cup P_2} \gamma^k(c) = \bigwedge_{c \in P_1} \gamma^k(c) \wedge \bigwedge_{c \in P_2} \gamma_k(c) = \gamma^k(P_1) \wedge \gamma^k(P_2) \ .$$

□

Proposition 59.

i) This is a direct consequence of Proposition 58.

ii) Let $P \in Prog^k_V$ and let P' be as in Definition 55. First we show that $\gamma^k(P) \leq \gamma^k(P' \cap Prog^k_{V\setminus x})$. Indeed, consider a clause $c \in P' \cap Prog^k_{V\setminus x}$. It is the folding of some clauses $c_1, \ldots, c_n$ of P in a clause c_{n+1} of P. Every ground instance c^g of c is the folding of suitable ground instances $c^g_1, \ldots, c^g_n$ of $c_1, \ldots, c_n$ in a suitable ground instance c^g_{n+1} of c_{n+1}. Therefore, if $[\![\gamma^k(P)]\!]\theta = 1$ then $[\![c^g_i]\!]\theta = 1$ for $i = 1, \ldots, n+1$ and $[\![c^g]\!]\theta = 1$. Since this is true for every $c \in P' \cap Prog^k_{V\setminus x}$, we have that $[\![\gamma^k(P' \cap Prog^k_{V\setminus x})]\!]\theta = 1$.

To show that $\exists_x^{Def_{\Delta,V}}\gamma^k(P) \leq \gamma^k(P \cap Prog^k_{V\setminus x})$, let $[\![\exists_x^{Def_{\Delta,V}}\gamma^k(P)]\!]\theta = 1$. Then $[\![\gamma^k(P)[S/x]]\!]\theta = 1$ for a suitable $S \in \wp(\mathsf{terms}(\Delta, \emptyset))$. Consider a clause $H :- B. \in P \cap Prog^k_{V\setminus x}$. Since $H :- B. \in P$ and x does not occur in $H :- B.$, we have $[\![H'[t_1/V_1]\cdots[t_n/V_n] :- B'[t_1/V_1]\cdots[t_n/V_n].]\!]\theta = 1$, for every $\{t_1, \ldots, t_n\} \subseteq \mathsf{terms}(\Delta, \emptyset)$, where $V_1, \ldots, V_n$ are the type variables of $H :- B.$. This means that $[\![\gamma^k(P \cap Prog^k_{V\setminus x})]\!]\theta = 1$.

By using the above facts we conclude that

$$
\begin{aligned}
\exists_x^{Def_{\Delta,V}} \gamma^k(P) &= \exists_x^{Def_{\Delta,V}} \left(\gamma^k(P) \wedge \gamma^k\left(P' \cap Prog^k_{V\setminus x}\right)\right) \\
&= \left(\exists_x^{Def_{\Delta,V}} \gamma^k(P)\right) \wedge \gamma^k\left(P' \cap Prog^k_{V\setminus x}\right) \\
&\le \gamma^k\left(P \cap Prog^k_{V\setminus x}\right) \wedge \gamma^k\left(P' \cap Prog^k_{V\setminus x}\right) \\
&= \gamma^k\left(\left(P \cap Prog^k_{V\setminus x}\right) \cup \left(P' \cap Prog^k_{V\setminus x}\right)\right) \\
&= \gamma^k\left(\exists_x^{Prog^k_V}(P)\right) .
\end{aligned}
$$

iii) Let $P \in Prog^k_V$. If some variable in $\tilde{y}$ occurs in P, then both sides are undefined. Otherwise

$$
\begin{aligned}
\gamma^k(\mathsf{rename}^{Prog^k}_{\tilde{x}\to\tilde{y}}\ P) = \gamma^k(P[\tilde{y}/\tilde{x}]) &= \bigwedge_{(H:-B.)\in P} \gamma^k(H[\tilde{y}/\tilde{x}] :-\ B[\tilde{y}/\tilde{x}].) \\
&= \bigwedge_{(H:-B.)\in P} ((\gamma^k(H :-\ B.))[\tilde{y}/\tilde{x}]) = \gamma^k(P)[\tilde{y}/\tilde{x}] \\
&= \mathsf{rename}^{T_V}_{\tilde{x}\to\tilde{y}}\ \gamma^k(P) .
\end{aligned}
$$

□

Proposition 63. Since c can be seen as a substitution, consider $\sigma \le c$. We show that $[\![\gamma^k \alpha^{alg}_V(c)]\!]\sigma = 1$. Let $x \in V$, $\{x_1, \ldots, x_n\} = \mathsf{vars}(c(x))$ and $\theta = \{Type \mapsto t^i, x_1 \mapsto t^i_{x_1}, \ldots, x_n \mapsto t^i_{x_n}\}$ be a computed answer substitution for $\mathtt{type}(c(x), Type)$, for $1 \le i \le m$. Then

$$
\begin{aligned}
& [\![(\mathtt{x_1}(t^i_{x_1}), \ldots, \mathtt{x_n}(t^i_{x_n}))'\mu]\!]\sigma = 1 \\
& \iff [\![x_1 \in t^i_{x_1}\mu \wedge \ldots \wedge x_n \in t^i_{x_n}\mu]\!]\sigma = 1 \\
& \iff [\![x_j \in t^i_{x_j}\mu]\!]\sigma = 1 && \text{for all } j = 1, \ldots, n \\
& \iff \sigma(x_j) \in [\![t^i_{x_j}\mu]\!]I && \text{for all } j = 1, \ldots, n \\
& \iff \sigma(x_j) \in [\![\theta(x_j)\mu]\!]I && \text{for all } j = 1, \ldots, n \\
\text{(Def. 60)}\ & \iff c(x)\sigma \in [\![Type\theta\mu]\!]I \\
& \iff c(x)\sigma \in [\![t^i\mu]\!]I \\
(\sigma \le c)\ & \iff \sigma(x) \in [\![t^i\mu]\!]I \\
& \iff [\![(\mathtt{x} \in t_i)'\mu]\!]\sigma = 1 .
\end{aligned}
$$

Therefore, for all $i = 1, \ldots, m$ we have $[\![(\mathtt{x}(t^i) :-\ \mathtt{x_1}(t^i_{x_1}), \ldots, \mathtt{x_n}(t^i_{x_n}).)'\mu]\!]\sigma = 1$ and $[\![(\mathtt{x_j}(t^i_{x_j}) :-\ \mathtt{x}(t^i).)'\mu]\!]\sigma = 1$ for all $j = 1, \ldots, n$. Thus, by Definition 62, $[\![\gamma^k \alpha^{alg}_x(c)]\!]\sigma = 1$. As $x \in V$ was arbitrary, we have the thesis. □

Proposition 64. Assume $[\![\gamma^k(P_1)]\!]\theta = 1$. Let $M(v) = \{d \in \mathsf{terms}(\Delta, \emptyset) \mid \theta(v) \in [\![d]\!]I\}$. For every clause $H :-\ B. \in P_1$ and every $\mu : \mathcal{T} \to \mathsf{terms}(\Delta, \emptyset)$ we have $[\![H'\mu :-\ B'\mu.]\!]\theta = 1$. This means that $M \models H\mu :-\ B\mu.$. Therefore, $M \models P_1$. Since $P_1 \le P_2$ we have $M \models P_2$. This means that for every clause $H :-\ B. \in P_2$ and every $\mu : \mathcal{T} \to \mathsf{terms}(\Delta, \emptyset)$ we have $M \models H\mu :-\ B\mu.$, which entails that $[\![H'\mu :-\ B'\mu.]\!]\theta = 1$. We conclude that $[\![\gamma^k(P_2)]\!]\theta = 1$. □

Proposition 65. Consider $\mu : \mathcal{T} \to \mathsf{terms}(\Delta, \emptyset)$. Since $\mathtt{v(d)}$ is derivable from P, then $\mathtt{v(d)}\mu$ is derivable from P. We show that if $M \models P$ then $M \models \mathtt{v(d)}\mu.$. Since this is done for every μ, we have the thesis. We proceed by induction on n, the number of resolution steps. If $n = 1$ then there exists a clause $\mathtt{v(t)}. \in P$ such that $t\mu' = d\mu$ for a suitable μ'. Since $M \models \mathtt{v(t)}.$, we have $M \models \mathtt{v(t)}\mu'.$, i.e., $M \models \mathtt{v(d)}\mu.$. Assume the result is true for $n \geq 1$ and that $\mathtt{v(d)}\mu$ is derivable from P by $n + 1$ steps of resolution. Then there exists a clause $\mathtt{v(t)} :- B_1, \ldots, B_m. \in P$ such that $t\mu' = d\mu$ for a suitable μ' and $B_i\mu'$ is derivable from P in n steps for all $i = 1, \ldots, m$. By inductive hypothesis, we conclude that $P \leq B_i\mu'$ for all $i = 1, \ldots, m$. Since $M \models P$, we have $M \models B_i\mu$ for all $i = 1, \ldots, m$, and since $M \models \mathtt{v(t)}\mu' :- B_1\mu', \ldots, B_m\mu'.$ we have $M \models \mathtt{v(t)}\mu'.$, i.e., $M \models \mathtt{v(d)}\mu.$. $\square$