Generalizing \textit{Def} and \textit{Pos} to Type Analysis

PATRICIA M. HILL, \textit{School of Computing, University of Leeds, Leeds, LS2 9JT, UK.} 
\textit{E-mail: hill@comp.leeds.ac.uk}

FAUSTO SPOTO, \textit{Dipartimento Scientifico e Tecnologico, Strada Le Grazie, 15, 37134 Verona, Italy.} 
\textit{E-mail: spoto@sci.univr.it}

Abstract

This paper is concerned with the type analysis of logic programs where, by \textit{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 generalization of the traditional Boolean domains \textit{Def} and \textit{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.

\textit{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 \textit{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 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 optimization 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 specialized code for the unification of that variable.

A well-known and useful type, which distinguishes whether a term contains variables or not, is groundness \cite{2, 10, 11}. The usual domains for groundness analysis, \textit{Def} and \textit{Pos}, feature some desirable properties: simplicity, effectivity, usefulness. Moreover, it has been
shown [34, 44] that Pos is condensing, i.e. it propagates the property of 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 generalization 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 generalization
of groundness to generic types is given in [8], but it is assumed, without any proof of cor-
rectness, 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 related work in more detail in Subsection 1.1.

In this paper we generalize 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 $\psi(H)$ and $\psi(Sol)$, 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 down-
ward (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
generalized 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. More-
over, 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 generalization 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 synthesizes the various domains considered in this paper, and their re-
lationships as abstraction (represented by horizontal arrows) or lifting to the powerset (repre-
sented by vertical arrows).
Generalizing Def and Pos to Type Analysis

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 generalizing 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 formalized 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 generalized to types. First, as is the case in [45], the property of groundness itself can be generalized. 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 generalize 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 generalization of a domain like $\text{Pos}$, the domain for propagating groundness. There, it is assumed that the usual properties that hold for $\text{Pos}$ in the case of groundness still hold for its generalization 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 $\text{Pos}$ is defined, it is not a generalization of $\text{Pos}$. For instance, it is not made of logical formulas. Finally, in [42], $\text{Pos}$ is combined with type information. However, their construction is not the result of any automatic, methodological construction which starts from the properties of interest and leads to the abstract domain.

2 Preliminaries

The powerset of a set $S$ is $\mathcal{P}(S) = \{ S' \mid S' \subseteq S \}$. We denote by $\mathcal{P}_1(S)$ 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 $\downarrow \mathcal{P}(S) = \{ S' \subseteq S \mid S' = \downarrow S \}$ 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 $\text{Seq}(S)$. If $\vec{x}$ is a sequence we will silently assume that $\vec{x} = (x_1, \ldots, x_l)$, 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.

¹Although, 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.
**Definition 2.1**

A constraint system over a set of variables $V$ is a family of sets $D = \{D_V\}_{V \in \mathcal{P}(V)}$ together with three operations: for $V \in \mathcal{P}(V)$ we have a (partial) infix conjunction operation $\star^{D_V} : D_V \times D_V \rightarrow D_V$, a (partial) renaming operation $\text{rename}^{D_V} : \text{Seq}(V) \times \text{Seq}(V) \times D_V \rightarrow D_V$ and a (total) cylindrification operation $\exists^{D_V} : V \times D_V \rightarrow D_V$. We write $\text{rename}^{D_V}_{\tilde{x} \rightarrow \tilde{y}} c$ for $\text{rename}^{D_V}(\tilde{x}, \tilde{y}, c)$ and $\exists^{D_V}_c c$ 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 2.2**

For every $V \in \mathcal{P}(V)$, let $D_V = \wp(V)$. This means that the constraints over the variables $V$ are all the subsets of $V$. We define the operations $\star^{D_V}(V_1)(V_2) = V_1 \cup V_2$, $\text{rename}^{D_V}_{\tilde{x} \rightarrow \tilde{y}}(V') = V'[\tilde{y} / \tilde{x}]$ and $\exists^{D_V}(V') = V' \setminus \{x\}$, 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 2.1.

In Subsection 2.3 we will see a more complex and useful constraint system. It turns out that the constraints of Example 2.2 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 2.3**

Let $D = \{D_V\}_{V \in \mathcal{P}(V)}$ be a constraint system and $\Pi$ a finite set of predicate symbols with associated arity. We denote by $\pi$ the set of distinct variables $\{i_1, \ldots, i_m\}$ where $m$ is the maximum arity of the predicates in $\Pi$.

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

$$G^{D_V} ::= c \mid G^{D_V} \land G^{D_V} \mid G^{D_V} \lor G^{D_V} \mid p(x_1, \ldots, x_l),$$

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

By $P^{D_V}$ 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 $p^l$ has the form

$$p(y_1, \ldots, y_k) \leftarrow G,$$

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

We write $p$ for $p()$ if $p$ 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 2.3, 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 2.4**

An interpretation over the constraint system $D_V$ is a function $I : \Pi \rightarrow \wp(D_V)$. The set of interpretations over $D_V$ is denoted by $I^{D_V}$. The set $I^{D_V}$ is a complete lattice w.r.t. the $\subseteq$
Given a program \( P \in \mathcal{P}^{D_V} \), the immediate consequence operator \( T_P^{D_V} : \mathcal{I}^{D_V} \rightarrow \mathcal{I}^{D_V} \) is defined as

\[
T_P^{D_V}(I)(p^i) = \begin{cases} 
\exists y_i \mapsto \text{rename}^{D_V}_{\bar{y} \rightarrow \bar{v}}, \mathcal{C}A^{D_V}[G] & \text{if } (p(y_1, \ldots, y_l) \leftarrow G) \in P \\
\emptyset & \text{otherwise},
\end{cases}
\]

for every predicate symbol \( p^i \in \Pi \). The exponent \( D_V \) will be omitted when it is clear from the context.

**Proposition 2.7**

Given a program \( P \in \mathcal{P}^{D_V} \), \( T_P \) is additive.

Proposition 2.7 allows us to give the following definition.
Given a program $P \in \mathbf{P}^{D_V}$, its computed answer semantics is
\[
S^{D_V}_{P} = \bigcup_{n \geq 0} T^{D_V}_P \uparrow n(\bot),
\]
where $T^{D_V}_P \uparrow o(\bot) = \bot$ and $T^{D_V}_P \uparrow t+1(\bot) = T_P(T^{D_V}_P(\bot))$. 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 $\Sigma$ with associated arity and $k \in \mathbb{N}$, we define
\[
\begin{align*}
\text{terms}^0(\Sigma, V) &= V \\
\text{terms}^{k+1}(\Sigma, V) &= \text{terms}^k(\Sigma, V) \\
 &\quad \cup \left\{ f(t_1, \ldots, t_n) \mid f^n \in \Sigma \text{ and } \{t_1, \ldots, t_n\} \subseteq \text{terms}^k(\Sigma, V) \right\}
\end{align*}
\]
\[
\text{terms}(\Sigma, V) = \bigcup_{d \geq 0} \text{terms}^d(\Sigma, V).
\]

We will always assume that $\Sigma$ contains at least one function symbol of arity 0. We denote by $\text{vars}(t)$ the set of variables which occur in a term $t$. When $\text{vars}(t) = \emptyset$ 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 $\vec{x} = \langle x_1, \ldots, x_t \rangle$ and $\vec{y} = \langle y_1, \ldots, y_t \rangle$ are two sequences of variables without repetitions then $t[\vec{y}/\vec{x}]$ stands for $t[y_1/x_1, \ldots, y_t/x_t]$, i.e., parallel substitution of variables. If $f : \varphi(\text{terms}(\Sigma, V)) \to \varphi(\text{terms}(\Sigma, V))$ then $\mu f$ 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 $\theta$ such that $\text{dom}(\theta) \subseteq V$, $\theta(x) \in \text{terms}(\Sigma, W)$ for every $x \in V$ and $\text{dom}(\theta) \cap \text{mg}(\theta) \subseteq Z$. If $Z = \emptyset$ we omit the superscript. The elements of $\Theta_{V,W}$ are called idempotent substitutions [38]. We write $\Theta_V$ for $\Theta_{V,V}$ and $\Theta^Z_{V,V}$ for $\Theta^Z_{V,V}$. We denote by $\varepsilon$ the empty substitution. Given $\theta$ 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 $t \in \text{terms}(\Sigma, V)$ and $\theta \in \Theta^Z_{V,W}$, $t[\theta] \in \text{terms}(\Sigma, W)$ 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 $(\sigma \theta)(x) = \theta(x) \sigma$ 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 $\sigma[n/x](x) = x$, $\sigma[n/x](n) = \sigma(n)$ and $\sigma[n/x](y) = \sigma(y)$ 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 $\Theta_V$ as $\theta \preceq \theta'$ if there exists a substitution $\sigma \in \Theta^V_V$ such that $\sigma \theta = \theta'$. When $V$ is clear from the context, we write $\preceq$ instead of $\preceq_V$. A preorder, called subsumption, is defined on $\text{terms}(\Sigma, V)$ as $t_1 \preceq t_2$ (t_1 \preceq t_2$.

\footnote{Note that $\sigma$ is not required to be idempotent, even when $\theta$ and $\theta'$ are idempotent [38].}
Given \( t_1 = t_2 \theta \) for a suitable \( \theta \in \Theta_Y \). 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 \( \Sigma \) 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

\[
C_V = \varphi_f(\{t^1 = t^2 \mid t^1, t^2 \in \text{terms}(\Sigma, V)\}).
\]

Every substitution can be seen as a set of Herbrand equations. The embedding map is \( \text{Eq}(\theta) = \{v = \theta(v) \mid v \in \text{dom}(\theta)\} \). 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 \( \text{mgu}(c) \in \Theta_V \) which is such that \( c \theta \) is true if and only if \( \text{mgu}(c) \theta \) is true. If no \( \theta \in \Theta_V \) exists such that \( c \theta \) is true, then \( \text{mgu}(c) \) is undefined.

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

\[
H_V = \left\{ \exists_w c \mid W \in \varphi_f(W), c \in C_{V \cup W}, \text{there exists } \theta \in \Theta_{V \cup W, V} \text{ such that } c \theta \text{ is true} \right\}.
\]

Here, \( \mathcal{V} \) are called the program variables and \( \mathcal{W} \) the existential variables. The requirement about the existence of \( \theta \) 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

\[
\text{sol}_V(\exists_w c) = \{\theta \mid \theta \in \Theta_{V \cup W, V} \text{ and } c \theta \text{ is true}\}.
\]

Hence \( \text{sol}_V(\exists_w c) = \text{sol}_V(\exists_w \text{mgu}(c)) \). Note that \( \text{sol}_V(\exists_w c) \) 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 2.9**

Given \( \theta \in \Theta_V \), we have \( \text{sol}_V(\text{Eq}(\theta)) = \downarrow\{\theta\} \).

A preorder is defined on \( H_V \) as \( h_1 \leq h_2 \) if and only if \( \text{sol}_V(h_1) \subseteq \text{sol}_V(h_2) \). 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 \( \text{sol}_V(h_1) = \text{sol}_V(h_2) \). 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 2.1.
Given \( V \in \varphi_I(V), x \in V \) and \( \hat{x}, \hat{y} \in Seq(V) \), we define \(^3\)

\[ (\exists_{v,c_1}) x^H_v (\exists_{w_c} o_2) = \begin{cases} \exists_{w,\cup w_2} mgu(c_1 \cup c_2) & \text{if } mgu(c_1 \cup c_2) \text{ exists} \\ \text{undefined} & \text{otherwise} \end{cases} \]

\[ \exists^H_x (\exists_{w_c} N) = \exists_{w,\cup N} c[N/x] \quad \text{with } N \in W \setminus W \]

\[ \text{rename}^H_{\hat{x} \rightarrow \hat{y}} (\exists_{w_c}) = \begin{cases} \exists_{w,c}[\hat{y}/\hat{x}] & \text{if } \exists_{w_c} \in H_{v \setminus \hat{y}} \\ \text{undefined} & \text{otherwise}. \end{cases} \]

### 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 \( \lor \)) and greatest lower bound (or meet, denoted by \( \land \)) 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 \( f(\cup_{i \in I} \{l_i\}) = \cup_{i \in I} f(l_i) \). It is co-additive when, in the same condition, we have \( f(\cap_{i \in I} \{l_i\}) = \cap_{i \in I} f(l_i) \).

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 \( \mu(A) \).

**Definition 2.11**

Let \((C, \leq)\) and \((A, \preceq)\) be two complete lattices (the concrete and the abstract domain). A *Galois connection* \((\alpha, \gamma) : (C, \leq) \rightleftarrows (A, \preceq)\) is a pair of monotonic maps \( \alpha : C \rightarrow A \) (abstraction) and \( \gamma : A \rightarrow C \) (concretization) 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 \((A, \preceq)\)) \((\alpha, \gamma) : (C, \leq) \rightarrow (A, \preceq)\) 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 insertion if and only if \( \gamma \) is one-to-one or, equivalently, if and only if \( \alpha \) is onto. If \( \alpha : C \rightarrow A \) is additive or \( \gamma : A \rightarrow C \) is co-additive, then \((\alpha, \gamma)\) is a Galois connection from \( C \) to \( A \). In a Galois insertion, the abstraction map uniquely identifies the concretization 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 Moore families of \( C \).

Let \( f : C^n \rightarrow C \) be a concrete operator and let \( \bar{f} : A^n \rightarrow A \). Then \( \bar{f} \) is *correct* with respect to \( f \) if and only if for all \( y_1, \ldots, y_n \in A \) we have \( \alpha(f(\gamma(y_1), \ldots, \gamma(y_n))) \preceq \bar{f}(y_1, \ldots, y_n) \).

For each operator \( f \), there exists an *optimal (most precise) correct abstract operator* \( \bar{f} \) defined as \( \bar{f}(y_1, \ldots, y_n) = \alpha(f(\gamma(y_1), \ldots, \gamma(y_n))) \). We say that \( \bar{f} \) is *precise w.r.t.* \( f \) if and only if for every \( x_1, \ldots, x_n \in C \) we have \( \bar{f}(\alpha(x_1), \ldots, \alpha(x_n)) = \alpha(f(x_1, \ldots, x_n)) \). Every precise operator is optimal. The following properties are well-known: the composition of correct

---

\(^3\)In another context, 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 \( \exists^H_{w\setminus\hat{y}} \) is irrelevant.
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 2.1 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 instantiation 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 \( A \cap B \) 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 \( a_1 \cap b_1 = a_2 \cap b_2 \). This means that pairs having the same meaning are identified. Linear refinement [20] is a slight generalization 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 \( a, b \in \wp(\Theta_V) \) we define

\[
a \rightarrow b = \bigcup \{ d \in \wp(\Theta_V) \mid a \cap d \subseteq b \} = \{ \theta \in \Theta_V \mid \text{for all } \sigma \leq \theta \text{ if } \sigma \in a \text{ then } \sigma \in b \}. \tag{2.1}
\]

The set \( a \rightarrow b \) 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 \( a \rightarrow b \) as the set of substitutions which transform the property \( a \) into the property \( b \) upon unification.

**Example 2.12**

Given \( v \in V \), let \( \mathbf{v} = \{ \theta \in \Theta_V \mid \text{vars}(\theta(v)) = \emptyset \} \). The set \( \mathbf{v} \) is the set of substitutions where the variable \( v \) is bound to a ground term. We have \( \mathbf{v} \in \wp(\Theta_V) \). Given \( x, y \in V \), Equation (2.1) says that

\[
x \rightarrow y = \{ \theta \in \Theta_V \mid \text{for all } \sigma \leq \theta \text{ if } \text{vars}(\sigma(x)) = \emptyset \text{ then } \text{vars}(\sigma(y)) = \emptyset \}.
\]

This means that every \( \theta \in x \rightarrow y \) is such that in all its instantiations if \( x \) is ground then \( y \) is ground. Equivalently, you can say that when we unify \( \theta \) with \( \theta' \in \Theta_V \), the fact that \( x \) is ground in \( \theta' \) entails that \( y \) is ground in \( \theta \theta' \). That is, \( \theta \) transforms the groundness of \( x \) in \( \theta' \) into the groundness of \( y \) in \( \theta \theta'. \) For instance, we have \( \{ x \mapsto \text{f(y)} \} \in x \rightarrow y \) and \( \{ y \mapsto \text{f(a)} \} \in x \rightarrow y \). But \( \{ x \mapsto \text{f(a)} \} \not\in x \rightarrow y \). Note that \( x \rightarrow x = \Theta_V \), 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 \( L \subseteq \wp(\Theta_V) \) we define

\[
L \triangleright L = \bigcup \{ a \rightarrow b \mid a, b \in L \}. \tag{2.2}
\]

The linear refinement of \( L \) is the domain

\[
L \rightarrow L = L \cap (L \triangleright L), \tag{2.3}
\]

Generalizing Def and Pos to Type Analysis
which can be simplified into

\[ L \rightarrow L = L \triangleright L \tag{2.4} \]

if \( L \subseteq L \triangleright L \), as will always be the case in this paper. This case is relevant since it allows a simpler representation and simpler operations for \( L \rightarrow L \). Note that we overload \( \rightarrow \). It is defined, in (2.1), as \( a \rightarrow b \) where \( a, b \in \mathcal{P}(\Theta_V) \) and, in (2.3), as \( L \rightarrow L \) over the single domain \( L \subseteq \mathcal{P}(\Theta_V) \).

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 2.13**

Using the notation of Example 2.12, the set \( \{ v \mid v \in V \} \) is able to express basic facts about the groundness of single variables. The domain \( G = \bigwedge \{ v \mid v \in V \} \) is able to express basic facts about the groundness of sets of variables. For instance, \( x \cap y \) (traditionally written as \( xy \)) 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 \( \text{Def} \) and \( \text{Pos} \) for groundness analysis [2, 10, 11] can be derived from \( G \) through linear refinement. Namely, we have \( \text{Def} = G \rightarrow G \) and \( \text{Pos} = \text{Def} \rightarrow \text{Def} \). Moreover, \( \text{Pos} \) cannot be further linearly refined, i.e. \( \text{Pos} \rightarrow \text{Pos} = \text{Pos} \). We have \( G \subseteq G \triangleright G \). In particular, given \( \{ x_1, \ldots , x_n \} \subseteq V \), it can be shown that \( x_1 \cdots x_n = \Theta_V \rightarrow (x_1 \cdots x_n) \), 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 \( C = \{ C_V \}_{V \in \mathcal{P}(\gamma)} \) and \( A = \{ A_V \}_{V \in \mathcal{P}(\gamma)} \) be two constraint systems such that, for every \( V \in \mathcal{P}(\gamma) \), \( C_V \) 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 2.1 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 \( (\mathcal{P}(C_V), \subseteq) \) and \( (\mathcal{P}(A_V), \subseteq) \) by defining \( \alpha_V(P) = \{ \alpha_V(c) \mid c \in P \} \) for all \( P \in \mathcal{P}(C_V) \). The 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 \( Q_1, Q_2 \in \mathcal{P}(A_V) \). Suppose that \( S_{p^\downarrow V} \) is defined as the abstract counterpart to the computed answer semantics \( S_{p^V} \) of Definition 2.8 where \( C.A_{\downarrow V} \) is defined as in Definition 2.5 but where \( D_V \) is replaced by \( A_V \) and

\[ C.A_{\downarrow V}[c]I = \{ \alpha_V(c) \} \tag{2.5} \]

Since the immediate consequence operator of Definition 2.6 works on set-theoretic complete lattices and is additive (Proposition 2.7), by the general theory of fixpoint and abstract interpretation [13] we conclude that, for every \( P \in \mathcal{P}(A_V) \), we have

\[ \alpha_V(S_{p^V}) \subseteq S_{p^\downarrow V}, \]
that is, the abstract semantics is a correct approximation of the concrete semantics. Moreover, whenever \( T^{AV}_P \) is precise w.r.t. \( T^{CV}_P \), we have the stronger result
\[
\alpha_V(S^{CV}_P) = S^{AV}_P,
\]
which says that the abstract semantics is exactly the abstraction of the concrete semantics.

Note that the precision of \( T^{AV}_P \) w.r.t. \( T^{CV}_P \) is entailed by the precision of the sub-operations used in its definition (Definitions 2.6 and 2.5).

### 2.6 Abstract compilation

When, as in Definition 2.8, the fixpoint computation is based on a compositional definition, like that of Definition 2.5, 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 2.14**

Let \( C \) and \( A \) be as in Subsection 2.5. Given a program \( P \in P^{CV} \), we define \( \alpha_V(P) \in P^{AV} \) as the program obtained from \( P \) by substituting the constraints in the clauses with their abstraction through \( \alpha_V \).

The semantics of \( \alpha_V(P) \) is then computed as the least fixpoint of \( T^{AV}_{\alpha_V(P)} \), where Definition 2.5 is used, without the change introduced with Equation (2.5) since the constraints considered by that equation have already been compiled in \( \alpha_V(P) \). It can be proved by induction that
\[
S_P^{AV} = S^{AV}_{\alpha_V(P)},
\]

On the left-hand side, Equation (2.5) is used. On the right-hand side, instead, the original Definition 2.5 is used, since the constraints considered by Equation (2.5) have been compiled in \( \alpha_V(P) \).

More aggressive abstract compilation techniques than that of Definition 2.14 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 2.5 is meaningful if the operations of Definition 2.1 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 \( \theta \), its downward closure represents the set of substitutions which are consistent with \( \theta \).
We define the family of sets \( \text{Down} = \{\text{Down}_V\} \subseteq \Theta_V \) where \( \text{Down}_V = \downarrow (\Theta_V) \) is a complete lattice. Let \( S, S_1, S_2 \in \text{Down}_V, x \in V \) and \( \hat{x}, \hat{y} \in \text{Seq}(V) \). We define

\[
S_1 \star \text{Down}_V S_2 = S_1 \cap S_2
\]

\[
\exists_{\text{Down}_V} S = \{\theta \in \Theta_V \mid \exists n \in V \setminus V, \sigma \in S[n/x], \theta \in \Theta_{V \cup n \setminus V} \text{ such that } \theta \leq_{V \cup n \setminus V} \sigma \text{ and } \theta \upharpoonright V = \sigma \}
\]

\[
\text{rename}_{\text{Down}_V} S = \{s[\hat{y} / \hat{x}; \hat{x} / \hat{y}] \mid s \in S \}.
\]

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 3.2**

Let \( V = \{v, x, y, z\} \) and

\[
\theta_1 = \{x \mapsto f(y)\}, \quad \theta_2 = \{x \mapsto f(z)\}.
\]

Suppose

\[
\sigma_1 = \{x \mapsto f(h(v)), y \mapsto h(v)\}, \quad \sigma_2 = \{x \mapsto f(h(v)), z \mapsto h(v)\},
\]

\[
\sigma_3 = \{x \mapsto f(h(v)), y \mapsto h(v), z \mapsto h(v)\}, \quad \sigma_4 = \{x \mapsto f(h(v))\}.
\]

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 \( \sigma_1, \sigma_2, \sigma_4 \notin \downarrow \theta_1 \star \text{Down}_V \downarrow \theta_2 \), and \( \sigma_3 \in \downarrow \theta_1 \star \text{Down}_V \downarrow \theta_2 \).

It follows from Definition 3.1 that \( \theta_1 \in \exists_{\text{Down}_V} \downarrow \theta_1 \). To see this choose any \( n \in V \setminus V \) and let \( \sigma = \{n \mapsto f(y)\} \). Then \( \sigma \in (\downarrow \theta_1)[n / x] \). Therefore, if \( \theta = \{n \mapsto f(y), x \mapsto f(y)\} \), we have \( \theta = \sigma \circ \{x \mapsto f(y)\} \) and \( \theta \upharpoonright V = \theta_1 \). Note that we have \( \varepsilon \in \exists_{\text{Down}_V} \downarrow \theta_1 \) since, in Definition 3.1 we can choose \( \sigma \) as above and \( \theta = \sigma \). We have \( \theta \leq_{V \cup n \setminus V} \sigma \) and \( \theta \upharpoonright V = \varepsilon \).

It follows from Definition 3.1 that \( \{v \mapsto f(y)\} \in \text{rename}_{\text{Down}_V} (\downarrow \theta_1) \). Moreover, as \( \{x \mapsto f(y), v \mapsto a\} \notin \downarrow \theta_1 \), and \( \{x \mapsto f(y), v \mapsto a\}[v / x, x / v] = \{v \mapsto f(y), x \mapsto a\} \), we have \( \{v \mapsto f(y), x \mapsto a\} \in \text{rename}_{\text{Down}_V} (\downarrow \theta_1) \).

The operations of Definition 3.1 are closed on \( \text{Down}_V \). Moreover, they satisfy some interesting properties.

**Proposition 3.3**

Let \( V \in \varphi_f(V) \) and \( S, S_1, S_2 \subseteq \Theta_V \).
510 Generalizing Def and Pos to Type Analysis

(i) If $S_1 = \downarrow S_1$ and $S_2 = \downarrow S_2$ then $S_1 \star_{\text{Down}^V} S_2 = \downarrow (S_1 \star_{\text{Down}^V} S_2)$.
(ii) If $S = \downarrow S$ then $\exists_{\text{Down}^V} \downarrow S = \downarrow (\exists_{\text{Down}^V} S)$.
(iii) If $S_1 \subseteq S_2$, then $\exists_{\text{Down}^V} S_1 \subseteq \exists_{\text{Down}^V} S_2$ (cylindrification is monotonic).
(iv) $S \subseteq \exists_{\text{Down}^V} (S)$ (cylindrification is extensive).
(v) If $S = \downarrow S$ then rename$^V_{\text{Down}^V} (S) = \downarrow (\text{rename}^V_{\text{Down}^V} (S))$.

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

**Proposition 3.4**

If $\Sigma$ contains at least a constant and a functor symbol, then

$$\{ \text{sol}_V(h) \mid h \in H_V \} \subset \wp(\Theta_V).$$

In spite of this result, we can show that $\star_{\text{Down}^V}$, rename$^V_{\text{Down}^V}$ and $\exists_{\text{Down}^V}$ are closed on the set $\text{sol}_V(H_V)$.

**Proposition 3.5**

Let $V \in \wp_f(V)$ and $h, h_1, h_2 \in H_V$.

(i) $\text{sol}_V(h_1) \star_{\text{Down}^V} \text{sol}_V(h_2) = \text{sol}_V(h_1 \star_{H_V} h_2)$,
(ii) $\exists_{\text{Down}^V} \text{sol}_V(h) = \text{sol}_V(\exists_{H_V} h)$ for every $x \in V$,
(iii) if $h \in H_V \backslash \gamma$, then rename$^V_{\text{Down}^V} \text{sol}_V(h) = \text{sol}_V(\text{rename}_{\text{Down}^V}^{H_V} h)$ for every $\bar{x}, \bar{y} \in \text{Seq}(V)$.

**Example 3.6**

Let $V$ and $\theta_1$ be as in Example 3.2. By Proposition 2.9 we have that $\exists_{\text{Down}^V} \downarrow \theta_1 = \exists_{\text{Down}^V} \text{sol}_V(\theta_1)$. Thus, by Proposition 3.5(ii), we have $\exists_{\text{Down}^V} \downarrow \theta_1 = \text{sol}_V(\exists_{H_V} \theta_1) = \text{sol}_V(\varepsilon) = \Theta_V$ which entails the result $\{\theta_1, \varepsilon\} \subseteq \exists_{\text{Down}^V} \downarrow \theta_1$ shown in Example 3.2. Similarly, by Proposition 3.5(iii) we have

$$\text{rename}_{(x) \mapsto (y)}^V \downarrow \theta_1 = \text{rename}_{(x) \mapsto (y)}^V \text{sol}_V(\theta_1) = \text{sol}_V(\text{rename}_{(x) \mapsto (y)}^{H_V} \theta_1)$$

This entails the result $\{\{v \mapsto f(y)\}, \{v \mapsto f(y), x \mapsto a\}\} \subseteq \text{rename}_{(x) \mapsto (y)}^V \downarrow \theta_1$, also shown in Example 3.2.

Thanks to Proposition 3.5, we can introduce the following definition.

**Definition 3.7**

We define $\text{Sol} = \{\text{sol}_V\}_{V \in \wp_f(V)}$, where $\text{Sol}_V = \{\text{sol}_V(h) \mid h \in H_V\}$. The $\star_{\text{Sol}^V}$, rename$^{\text{Sol}^V}$ and $\exists_{\text{Sol}^V}$ operators on $\text{Sol}_V$ are the restriction on $\text{Sol}_V$ of the corresponding operators of $\text{Down}^V$.

Since $\text{sol}_V$ is one-to-one and onto from $H_V$ into $\text{Sol}_V$, and for every $\{h_1, h_2\} \subseteq H_V$ we have $h_1 \leq h_2$ if and only if $\text{sol}_V(h_1) \subseteq \text{sol}_V(h_2)$, we conclude that $\text{sol}_V$, endowed with the $\subseteq$ partial ordering, is isomorphic to $H_V$, and we know that the corresponding operations coincide (Proposition 3.5). The usefulness of $\text{Sol}$ 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 \( \phi(H) \) and \( \phi(Sol) \), the domains used for the collecting semantics. The domain \( \phi(Sol) \) 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 \( Down \) to represent our abstract properties. We prove here that there is a Galois insertion from \( \phi(Sol) \) into \( Down \) for every \( V \in \phi_f(V) \).

We use \( \cup \) as abstraction map from \( \phi(Sol) \) into \( Down \). The map \( \cup \) is onto from \( \phi(Sol) \) into \( Down \). This is because every \( d \in Down \) can be written as the infinite union \( d = \bigcup \{ \theta \mid \theta \in d \} \) and every single \( \downarrow \theta \) belongs to \( Sol \), as shown by Proposition 2.9. It can be shown that \( \cup \) is additive from \( \phi(Sol) \) into \( Down \).

**Proposition 3.8**

Given \( \{ p_i \}_{i \in I} \subseteq \phi(Sol) \) with \( I \subseteq \mathbb{N} \), we have

\[
\bigcup \left( \bigcup_{i \in I} \phi(Sol) p_i \bigcup \right) = \bigcup_{i \in I} Down (\bigcup p_i)
\]

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

**Proposition 3.9**

Let \( \{ p_1, p_2, p \} \subseteq \phi(Sol) \). We have

(i) \( \bigcup \{ p_1 \} \bigcup (Sol) p_2 = (\bigcup p_1) \bigcup Down (\bigcup p_2) \)

(ii) \( \bigcup \{ \exists x \} (Sol) p = \exists x Down (\bigcup p) \) for every \( x \in V \).

(iii) \( \bigcup \{ \text{rename} \} (Sol) p = \text{rename} Down (\bigcup p) \) for every \( \bar{x}, \bar{y} \in Seq(V) \).

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 4.1**

A type system is \( \langle \Delta, \Sigma, I \rangle \) where \( \Delta \) and \( \Sigma \), called type signature and term signature, respectively, are sets of function symbols with associated arities and \( I(\tau^n) \) is a total map from \( \phi_l(\text{terms}(\Sigma, V)) \) to \( \phi_l(\text{terms}(\Sigma, V)) \) for every \( \tau^n \in \Delta \). We define

\[
[f^0]I = I(\tau^0) \quad \text{and} \quad [\tau^n(t_1, \ldots, t_n)]I = I(\tau^n)([t_1]I, \ldots, [t_n]I).
\]
512 Generalizing Def and Pos to Type Analysis

Example 4.2
Let \( G = \langle g^0, \Sigma, I \rangle \) and let \( I(g) = \text{terms}(\Sigma, \emptyset) \). The functor symbol \( g \) is interpreted as the set of terms which do not contain variables. Therefore, \( G \) is the type system for groundness.

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

Let \( \mathbf{NL} = \langle \{ \mathbf{nat}^0, \mathbf{top}^0, \mathbf{list}^1 \}, \Sigma, I \rangle \), where \( \{d^0, s^1, [^0, [^2] \} \subseteq \Sigma \) and

\[
I(\mathbf{nat}) = \mu t. \{0\} \cup \{s(n) | n \in i\} \\
I(\mathbf{top}) = \text{terms}(\Sigma, \emptyset) \\
I(\mathbf{list}) = \lambda \tau. \mu t. \{[\ ]\} \cup \{[h[t]] | h \in I(\mathbf{nat}), t \in l\}.
\]

Since \( \mathbf{list} \) is unary, we can write types like \( \mathbf{list}(\mathbf{nat}) \) or even \( \mathbf{list}(\mathbf{list}(\mathbf{nat})) \). For instance, it follows from Definition 4.1 that the meaning of \( \mathbf{list}(\mathbf{nat}) \) in the interpretation \( I \) defined above is

\[
[[\mathbf{list}(\mathbf{nat})]]I = I(\mathbf{list})([[\mathbf{nat}]]I) = \mu t. \{[\ ]\} \cup \{[h[t]] | h \in I(\mathbf{nat}), t \in l\} \\
= \{[\ ], [0], [s(0)], [0, 0], [0, s(0)], [s(0), s(0)], \ldots \}.
\]

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 \( \mathbf{NLT} = \langle \{\mathbf{nat}^0, \mathbf{top}^0, \mathbf{list}^1, \mathbf{tree}^1 \}, \Sigma, I \rangle \), where \( \{d^0, s^1, [^0, [^2], \mathbf{void}^0, \mathbf{tree}^2 \} \subseteq \Sigma, I(\mathbf{nat}), I(\mathbf{top}) \) and \( I(\mathbf{list}) \) are defined as above and

\[
I(\mathbf{tree}) = \lambda \tau. \mu t. \{\mathbf{void}\} \cup \{\mathbf{tree}(n, l, r) | n \in \tau, l \in t \text{ and } r \in t\}.
\]

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

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

Definition 4.3
Given a type signature \( \Delta \) and a set of variables \( V \), the set \( \Phi_{\Delta, V} \) of transfinite formulas over \( \Delta \) and \( V \) is defined as the least set such that (\( x \in t \)) \( \in \Phi_{\Delta, V} \), where \( x \in V \) and \( t \in \text{terms}(\Delta, \emptyset) \), if \( S \in \varphi(\Phi_{\Delta, V}) \) then \( (\land S) \in \Phi_{\Delta, V} \) and \( (\lor S) \in \Phi_{\Delta, V} \). 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 \( \phi_1 \land \phi_2 \) for \( \land \{\phi_1, \phi_2\} \), \( \phi_1 \lor \phi_2 \) for \( \lor \{\phi_1, \phi_2\} \), \( \text{true} \) for \( \land \emptyset \) and \( \text{false} \) for \( \lor \emptyset \).

Example 4.4
Consider \( \Delta = \{\mathbf{nat}^0, \mathbf{list}^1 \} \) and \( V = \{x, y, z\} \). Examples of transfinite formulas in \( \Phi_{\Delta, V} \) are \( x \in \mathbf{nat} \land y \in \mathbf{list}(\mathbf{nat}) \) and \( z \in \mathbf{nat} \land z \in \mathbf{list}(\mathbf{nat}) \). 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 this in 4.7. Moreover, note that all those transfinite formulas are actually finite formulas. As an example of a strictly transfinite formula, consider

\[
x \in \mathbf{list}(\mathbf{nat}) \land x \in \mathbf{list}(\mathbf{list}(\mathbf{nat})) \land x \in \mathbf{list}(\mathbf{list}(\mathbf{list}(\mathbf{nat}))) \land \cdots.
\]

Once the set of formulas \( \Phi_{\Delta, V} \) of Definition 4.3 is endowed with the operations of Definition 2.1, it becomes a constraint system.
DEFINITION 4.5  
Let $T = \langle \Delta, \Sigma, I \rangle$ be a type system. A type constraint system for $T$ is $T = \{ T_V \}_{V \in \mathcal{V}(V)}$, where $T_V \subseteq \Phi_{\Delta, V}$ is closed w.r.t. the operations

$$
\phi_1 \land T_V \phi_2 = \phi_1 \land \phi_2
$$

rename$_{\overline{x} \rightarrow \phi(y)}^T V = \begin{cases} 
\phi[\overline{y}/\overline{x}] & \text{if } \phi \in \Phi_{\Delta, V} \backslash \overline{y} \\
\text{undefined} & \text{otherwise}
\end{cases}
$$

$$
\exists T_V \phi = \bigvee \left\{ \phi[P/x] \Big| \text{there exists } t \in \text{terms}(\Sigma, V) \text{ such that } P = \{ d \in \text{terms}(\Delta, \emptyset) \mid t \in \{d\} \} \right\}
$$

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

$$(x \in t)[y/x] = y \in t \quad (z \in t)[y/x] = (z \in t) \quad \text{if } z \neq x$$

$$(\phi_1 \land \phi_2)[y/x] = (\phi_1[y/x]) \land (\phi_2[y/x])$$

$$(\phi_1 \lor \phi_2)[y/x] = (\phi_1[y/x]) \lor (\phi_2[y/x])$$

$$(\phi_1 \Rightarrow \phi_2)[y/x] = (\phi_1[y/x]) \Rightarrow (\phi_2[y/x])$$

and, for all $P \in \varphi(\text{terms}(\Delta, \emptyset))$,

$$(x \in t)[P/x] = \begin{cases} 
\text{true} & \text{if } \cap y \in P[I] \subseteq [I]I \\
\text{false} & \text{otherwise}
\end{cases}$$

$$(\phi_1 \land \phi_2)[P/x] = (\phi_1[P/x]) \land (\phi_2[P/x])$$

$$(\phi_1 \lor \phi_2)[P/x] = (\phi_1[P/x]) \lor (\phi_2[P/x])$$

$$(\phi_1 \Rightarrow \phi_2)[P/x] = (\phi_1[P/x]) \Rightarrow (\phi_2[P/x])$$

Note that the simplest way, in Definition 4.5, 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 4.12).

Note that if 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 4.2, cylindrification becomes the classical Schröder elimination [2], since terms$(\Delta, \emptyset)$ contains only a constant symbol whose evaluation is not empty.\(^4\)

EXAMPLE 4.6  
Consider the type system NL = $\{\text{nat, top, list}, \Sigma, I\}$ of Example 4.2 and let $NL_V = \Phi_{\{\text{nat, top, list}\}, V}$ for every $V \in \mathcal{V}(\mathcal{V})$. As examples of operations on that type constraint system, let $V = \{x, y, z\}$. Then, by Definition 4.5,

$$(x \in \text{nat})^{NL_V} (y \in \text{list(nat)}) = x \in \text{nat} \land y \in \text{list(nat)}$$

rename$_{\overline{x} \rightarrow \phi(y)}^{NL_V} (x \in \text{list(nat)}) = y \in \text{list(nat)}$.

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

\(^4\)We recall that the Schröder elimination of $x$ from a propositional formula $\phi$ is $\phi[\text{false}/x] \lor \phi[\text{true}/x]$.
to \text{top} and there are terms which belongs only to \text{top}, for instance, \text{S}([\emptyset]). A term can be just a natural number, for instance 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 [[], 0]. A term can be a list of natural numbers, and then even a list of \text{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 \[0\]. We can continue this way indefinitely. In conclusion, we have infinite possibilities for \(P\), collected in the set \(S\) below:

\[S = \{\text{top}, \text{top, nat}, \text{top, list(top)}, \text{top, list(top), list(nat)}, \text{top, list(top), list(list(top)), list(list(nat))}, \ldots \}.
\]

As a consequence, we have

\[
\exists^N_{x^V} (x \in \text{list(nat)} \land y \in \text{nat}) = \bigvee_{P \in S} (x \in \text{list(nat)} \land y \in \text{nat})[P/x]
\]

\[
= \bigvee_{P \in S} (x \in \text{list(nat})[P/x] \land y \in \text{nat})
\]

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

\[
\left(\bigvee_{P \in S, P \neq P'} (false \land y \in \text{nat})\right) \lor (true \land y \in \text{nat}). \tag{4.1}
\]

We are obviously tempted to reduce Equation (4.1) to \(y \in \text{nat}\). In order to do so, we first need to define a notion of equivalence over transfinite formulas.

\textbf{Definition 4.7}

Given a type system \(T = \langle \Delta, \Sigma, I \rangle\), we define the map \([\phantom{\sum}]_T : \Phi_{\Delta, V} \to (\Theta_V \to \{0, 1\})\) as

\[
[x \in t]_T = \begin{cases} 
1 & \text{if } \sigma(x) \in [t]I \\
0 & \text{otherwise}
\end{cases}
\]

\[
\land S = \begin{cases} 
1 & \text{if } [\phi]_T = 1 \\
0 & \text{otherwise}
\end{cases}
\]

\[
\lor S = \begin{cases} 
1 & \text{if } [\phi]_T = 1 \\
0 & \text{otherwise}
\end{cases}
\]

\[
\neg \phi = 1 - [\phi]_T
\]

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

\textbf{Example 4.8}

In the hypotheses of Example 4.6, Equation (4.1) can be shown to be \((NL, V)\)-equivalent to \(y \in \text{nat}\). Indeed, for every transfinite formula \(\phi\) we have \(false \land \phi \equiv false, false \lor false = false, true \land \phi \equiv \phi\) and \(false \lor \phi = \phi\).
In the rest of the paper, if \( T = \{ T_V \}_{V \in \varphi_f(V)} \) is a type constraint system and \( V \in \varphi_f(V) \), 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. \( \leq_T, V \), since by Definition 4.5 it is completely \( \wedge \)-closed and topped (since \( \text{true} = \wedge(\emptyset) \)). We require that the operations \( \wedge^T, \exists^T \) and rename\( ^T \) of Definition 4.5 are independent from the representatives chosen for the \((T, V)\)-equivalence classes. This follows directly from Definition 4.5, for conjunction. We can also show that it is true for renaming.

**Proposition 4.9**
Let \( T = \{ T_V \}_{V \in \varphi_f(V)} \) be a type constraint system for the type system \( (\Delta, \Sigma, I) \). For every \( V \in \varphi_f(V), x, y \in V \) 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.

**Example 4.10**
Let \( x \in V \). Consider the type constraint system defined as \( F = \{ \Phi_{f, g, x} \}_{V \in \varphi_f(V)} \) for the type system \( F = \{ (g, gx), \Sigma, I \} \), where \( \{ a_0, f^1 \} \subseteq \Sigma \) and

\[
I(g) = \text{terms}(\Sigma, \emptyset), \quad I(gx) = \text{terms}(\Sigma, \{x\}) \setminus \{x\}.
\]

This type system has two types, i.e. the type \( g \) of ground terms and the type \( gx \) of terms different from \( x \) and whose only allowed variable is \( x \). Note that \( I(gx) \) is downward closed if we assume \( V = \{x\} \). Consider

\[
\phi_1 = (x \in gx \Rightarrow x \in g), \quad \phi_2 = \text{true}.
\]

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

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

\[
\phi_1[P/x] = (\text{true} \Rightarrow \text{false}) \equiv \text{false}, \quad \phi_2[P/x] = \text{true},
\]

which are not equivalent.

### 4.2 The semantics of a type constraint system

For all \( V \in \varphi_f(V) \), a Moore family of \( \text{Down}_V \) (i.e. an abstract interpretation of \( \text{Down}_V \)) can be defined once a type constraint system is given.

**Definition 4.11**
Given a type constraint system \( T = \{ T_V \}_{V \in \varphi_f(V)} \) for the type system \( T \) and \( \phi \in T_V \), we define

\[
\gamma_{T_V}(\phi) = \{ \theta \in \Theta_V \mid \text{for all } \sigma \in \Theta_V \text{ s.t. } \sigma \leq \theta \text{ we have } [\phi]_{\{x\}} = 1 \}.
\]
A type constraint system

Given a type system $\Gamma$.

In Example 4.12, we constructed a type constraint system in which there were formulas $\phi_1, \phi_2 \in T_V$. When it is clear from the context, we drop the subscript $T_V$ from $\gamma_{T_V}$.

It follows from Definition 4.11 that $\equiv$ entails $\equiv_\gamma$, though the converse does not hold.

**Example 4.12**

Consider the type system $G$ of Example 4.2 and the type constraint system $\{ \Phi_{\Delta, V} \}_{V \in \Gamma(V)}$ for $G$. Consider $V$ such that $\{ x, y \} \subseteq V$. Let $\phi_1 = \text{false}$ and $\phi_2 = x \in g \Rightarrow \text{false}$. Note that there is no $\theta$ such that $[\phi_1]_\theta = 1$. But any substitution $\theta$ such that $\theta(x)$ is not ground satisfies $[\phi_2]_\theta = 1$. 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 $\phi_1$. For $\phi_2$, let $\theta \in \gamma(\phi_2)$. Any instance of $\theta$ 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 $[\text{false}]_\theta = 1$. Note that we can always find such a $\theta'$ because $\Sigma$ contains at least one function symbol of arity 0 (Subsection 2.2).

**Proposition 4.13**

Given a type constraint system $T = \{ T_V \}_{V \in \Gamma(V)}$, $\gamma_{T_V}$ is co-additive and $\gamma_{T_V}(T_V)$ is a Moore family of $\text{Down}_V$ for all $V \in \Gamma(V)$.

Since $\gamma$ is co-additive and $\text{Down}_V$ and $T_V$ are complete lattices, we conclude that $\gamma$ is the concretization map of a Galois connection from $\text{Down}_V$ into $T_V$. Let $\alpha$ (i.e. $\alpha_{T_V}$) be the corresponding abstraction map. Since, in general, $\gamma$ is not one-to-one (Example 4.12), we are not guaranteed to have a Galois insertion instead of just a Galois connection. This is what the property below requires.

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

**Example 4.14**

In Example 4.12, we constructed a type constraint system in which there were formulas $\phi_1, \phi_2$ 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 4.15**

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

**Definition 4.16**

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

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

Positive type systems generalize the traditional type system for groundness, as defined in Example 4.2. Every type constraint system with just one type symbol is positive, provided we restrict the set of transfinite formulas to the positive ones.
EXAMPLE 4.17
Consider the type system $G$ of Example 4.2 and the type constraint system $Pos_{[g]} = \{Pos_{[g], V}\}_{V \in \text{Var}(V)}$ where $Pos_{[g], V}$ is formed by the set of positive transfinite formulas in $\Phi_{[g], V}$. Note that this set is finite. In [2] it is shown that $Pos_{[g], V}$ 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 $Def_{[g]} = \{Def_{[g], V}\}_{V \in \text{Var}(V)}$ where $Def_{[g], V}$ is the set of (positive) definite formulas in $\Phi_{[g], V}$. 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 4.18
The construction of Example 4.17 can be used with non-freeness (type system NF of Example 4.2). The resulting type constraint systems will be called $Pos_{\text{nf}}$ and $Def_{\text{nf}}$. 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 $\theta$ there exists a grounding substitution $\theta'$ with the same type properties as $\theta$. 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 4.16). Therefore, we provide a sufficient condition which entails that a type constraint system is structural, and shows how large that class is.

PROPOSITION 4.19
Let $T = \{T_V\}_{V \in \text{Var}(V)}$ be a type constraint system for the type system $\langle \Delta, \Sigma, I \rangle$. Assume there exists a ground term $t \in \text{terms}(\Sigma, \emptyset)$ such that for every $d \in \text{terms}(\Delta, \emptyset)$ and for all $t' \in \llbracket d \rrbracket I$, every term obtained from $t'$ by substituting some occurrences of $t$ with variables of $V$ is still in $\llbracket d \rrbracket I$. Then $T$ is structural.

EXAMPLE 4.20
Consider the type system $NL$ of Example 4.2 and the type constraint system $NL = \{NL_V\}_{V \in \text{Var}(V)}$ where $NL_V = \Phi_{\Delta, V}$. Since Proposition 4.19 holds with $t = s([])$, the type constraint 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 $\text{NLT}$ of Example 4.2.

EXAMPLE 4.21
Consider the type constraint system $Pos_{[g]}$ for the type system $G$ of Example 4.17. 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 maintaining the same type properties of $t$. This shows that a type constraint system may be positive but not structural.

EXAMPLE 4.22
Consider the type constraint system $N = \{N_V\}_{V \in \text{Var}(V)}$ for the type system $\langle \text{nat}^0, \Sigma, I \rangle$ where $\Sigma$ contains $0^0$, $s^1$ and $d^0$ and $I(\text{nat}) = \mu V. \{0\} \cup \{s(n) \mid n \in V\}$. If $N_V$ is the set of positive formulas over $\{\text{nat}\}$ and $V$, we have that $N$ is positive. Moreover, it is structural, as it can be proved by choosing $t = s$ in Proposition 4.19. This proves that there are type constraint systems which are both positive and structural.
The following propositions show the importance of positive or structural type constraint systems.

**Proposition 4.24**
Every positive or structural type constraint system satisfies both properties P1 and P2.

**Proposition 4.25**
Let $T = \{ T V \}_{V \in \{ y \}}$ be a type constraint system for the type system $\langle \Delta, \Sigma, I \rangle$, satisfying property P1.

(i) The operator $\#_{TV}$ is always correct w.r.t. $\#^{DownV}$, and it is its best possible approximation if property P2 holds for $T$.

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

(iii) The operator $\text{rename}_{TV}$ is always correct w.r.t. $\text{rename}^{DownV}$ and it is its best possible approximation if property P2 holds for $T$.

The correctness and optimality results for $\text{rename}_{TV}$ cannot be immediately combined with the similar results of Propositions 3.9(iii) and 3.5(iii) since Proposition 3.5(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 $\phi$, then $\gamma(\phi)$ is the union of sets of solutions of existential Herbrand constraints where $y$ does not occur.

**Proposition 4.26**
Let $T = \{ T V \}_{V \in \{ y \}}$ be a type constraint system. Let $V \in \wp(V)$, $\hat{y} \in \text{Seq}(V)$ and $\phi \in T_{V \setminus \hat{y}}$. There exists $I \subseteq H_{V \setminus \hat{y}}$ such that $\gamma_{TV}(\phi) = \cup \{ \text{sol}_{TV}(h) \mid h \in I \}$.

Thus, a positive or structural type constraint system $\text{Type}_{TV} = \{ \text{Type}_{TV} \}_{V \in \{ y \}}$ enjoys interesting properties. Namely, cylindrification is well-defined (property P1), a Galois insertion can be established between $\text{Down}_V$ and $\text{Type}_{TV}$ (property P2), the operators over $\text{Type}_{TV}$ are the best possible approximations of the corresponding operators over $\text{Down}_V$ (Propositions 4.24 and 4.25).

Since $\text{Pos}_{\{ y \}}$ and $\text{Def}_{\{ y \}}$ of Example 4.17 and $\text{Pos}_{\{ y \}}$ and $\text{Def}_{\{ y \}}$ of Example 4.18 are positive, we conclude that they are related to $\text{Down}$ through a Galois insertion and that the operations of Definition 4.5 are the best possible approximations of the corresponding operations of $\text{Down}$, by using Proposition 4.25. Since that proposition considers every positive type constraint system, we have generalized 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 4.5 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 4.17 and 4.18), while we considered the whole set of formulas for NL and NLT (Example 4.20).
Indeed, in the case of groundness or non-freeness, a variable can always eventually belong to the type, and a formula like \( \neg (x \in g) \) has an empty concretization. Therefore, it is useless. Instead, the formula \( \neg (x \in \text{nat}) \) 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 \textit{Down} (Proposition 4.13). In this subsection, we want to show how a hierarchy of abstract interpretations of \textit{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 generalizes an analogous result about groundness analysis shown in [44].

\textbf{Definition 4.27}

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

\[
\mathbf{v}_t = \{ \theta \in \Theta_V \mid \theta(v) \in \{t\} \} \quad \text{with } v \in V \text{ and } t \in \text{terms}(\Delta, \emptyset) \\
\mathbf{Basic}^0_{\mathcal{T}, V} = \bigwedge \{ \mathbf{v}_t \mid v \in V \text{ and } t \in \text{terms}(\Delta, \emptyset) \} ,
\]

\[
\mathbf{Basic}^{i+1}_{\mathcal{T}, V} = \mathbf{Basic}^i_{\mathcal{T}, V} \Theta \mathbf{Basic}^i_{\mathcal{T}, V} \quad \text{for } i \geq 0.
\]

We show now that \( \mathbf{Basic}^i_{\mathcal{T}, V} \subseteq \mathbf{Basic}^i_{\mathcal{T}, V} \Theta \mathbf{Basic}^i_{\mathcal{T}, V} \), which allows us to use Equation (2.4) in Definition 4.27.

\textbf{Proposition 4.28}

Let \( V \in \wp_f(V) \) and \( \mathcal{T} \) be a type system. We have \( \mathbf{Basic}^i_{\mathcal{T}, V} \subseteq \mathbf{Basic}^i_{\mathcal{T}, V} \Theta \mathbf{Basic}^i_{\mathcal{T}, V} \) for all \( i \geq 0 \).

The domains \( \mathbf{Basic}^0_{\mathcal{T}, V} \), \( \mathbf{Basic}^1_{\mathcal{T}, V} \), and \( \mathbf{Basic}^2_{\mathcal{T}, V} \) can be easily represented.

\textbf{Definition 4.29}

Let \( \langle \Delta, \Sigma, I \rangle \) be a type system. Let \( V \in \wp_f(V) \). We define

\[
\text{And}_{\Delta, V} = \{ \land S \mid S \subseteq \{ v \in t \mid v \in V \text{ and } t \in \text{terms}(\Delta, \emptyset) \} \} ,
\]

\[
\text{Or}_{\Delta, V} = \{ \lor S \mid S \subseteq \{ v \in t \mid v \in V \text{ and } t \in \text{terms}(\Delta, \emptyset) \} \text{ and } S \neq \emptyset \} ,
\]

\[
\text{Def}_{\Delta, V} = \{ A \Rightarrow A_2 \mid \{ A_1, A_2 \} \subseteq \text{And}_{\Delta, V} \} ,
\]

\[
\text{Pos}_{\Delta, V} = \{ A \Rightarrow O \mid A \in \text{And}_{\Delta, V} \text{ and } O \in \text{Or}_{\Delta, V} \} .
\]

\textbf{Proposition 4.30}

Let \( \mathcal{T} = \langle \Delta, \Sigma, I \rangle \) be a type system and \( V \in \wp_f(V) \). Then \( \text{Def}_{\Delta, V} \) is isomorphic to \( \mathbf{Basic}^1_{\mathcal{T}, V} \). If \( \mathcal{T} \) is structural then \( \text{Pos}_{\Delta, V} \) is isomorphic to \( \mathbf{Basic}^2_{\mathcal{T}, V} \). \( \mathbf{Basic}^2_{\mathcal{T}, V} \) is condensing, \( \mathbf{Basic}^{i+1}_{\mathcal{T}, V} = \mathbf{Basic}^i_{\mathcal{T}, V} \) for \( i \geq 2 \) and \( \mathbf{Basic}^{i+1}_{\mathcal{T}, V} = \mathbf{Basic}^i_{\mathcal{T}, V} \Theta \mathbf{Basic}^i_{\mathcal{T}, V} \Theta \mathbf{Basic}^i_{\mathcal{T}, V} \).

The condensing property means that, for structural type constraint systems, \( \text{Pos}_{\Delta, V} \) induces a conjunction operation which fully propagates the properties in \( \mathbf{Basic}^0_{\mathcal{T}, V} \). 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 4.30 holds for the type system of a positive type constraint system (Definition 4.16). However, in [44] it is shown based on an equivalent operational semantics.

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. Proposition 4.30 holds for the type system of a positive type constraint system (Definition 4.16). However, in [44] it is shown based on an equivalent operational semantics.

In this section we show how the Basic\(^{\Delta, V}_{\Lambda, V}\) domain of the previous section can be implemented. We have shown in Proposition 4.30 that we can use Def\(^{\Delta, V}_{\Lambda, V}\) 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 4.5 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 \(\text{terms}(\Sigma, \emptyset)\) 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 \(\text{list}\), the infinite conjunction

\[\land_{d \in \text{term}(\Delta, \emptyset)} (x \in \text{list}(d) \iff (h \in d \land l \in \text{list}(d)))\]

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 \(x \in \text{list}(T) \iff (h \in T \land l \in \text{list}(T))\). Note that this formula can be written as the logic program

\[
\begin{align*}
  x(\text{list}(T))&: -h(T), 1(\text{list}(T)). \\
  h(T)&: -x(\text{list}(T)). \\
  1(\text{list}(T))&: -x(\text{list}(T)).
\end{align*}
\]

In this program, the variables of interest \((x, h \text{ and } l\) 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 \(T\) of type variables, denoted by uppercase letters. Moreover, we will consider a fact \(f\) equivalent to a clause \(f\) : -.

### 5.1 The domain \(\text{Prog}^k\)

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 5.1**

Given a type system \(T = \langle \Delta, \Sigma, I \rangle\), a type model for \(T\) is a map \(M : V \mapsto \{\{d \in \text{term}(\Delta, \emptyset) \mid t \in [d]I \mid t \in \text{term}(\Sigma, V)\}\}.

Programs in \(\text{Prog}^k\) use type terms of depth \(k\) in order to have a finite constraint system.

**Definition 5.2**

Let \(T = \langle \Delta, \Sigma, I \rangle\) be a type system, \(V \in \wp(V)\) and \(k \geq 1\). A \(k\)-atom for \(V\) and \(\Delta\) is \(v(t)\), where \(v \in V\) and \(t \in \text{term}^k(\Delta, T)\). An element of \(\text{Prog}_V^k\) is a (possibly empty or infinite)
Figure 1 shows a program together with its restrictions w.r.t.

\[
x(\text{list}(T)): - y(T), z(\text{list}(T)).
\]

\[
y(T): - x(\text{list}(T)).
\]

\[
z(\text{list}(T)).
\]

A program \( P \in \text{Prog}_V^1 \) and two of its cylindrifications

\[
\exists_x^{\text{Prog}_V^1} P
\]

\[
\exists_x^{\text{Prog}_V^1} P
\]

\[
\text{Figure 1. A program and two of its cylindrifications}
\]

set of clauses \( H : - B \), where \( H \) is a \( k \)-atom for \( V \) and \( \Delta \) and \( B \) is a (possibly empty) sequence of \( k \)-atoms for \( V \) and \( \Delta \). We define the constraint system \( \text{Prog}^k = \{ \text{Prog}_{V}^k \}_{V \in \wp_T(V)} \) with the operations

\[
P_1 \star^{\text{Prog}^k}_{V} P_2 = P_1 \cup P_2
\]

\[
\text{rename}^{\text{Prog}^k}_{V} P = \begin{cases} P'[\hat{y}/\hat{x}] & \text{if } P \in \text{Prog}_{V \setminus \hat{y}}^k \\ \text{undefined} & \text{otherwise} \end{cases}
\]

\[
\exists_x^{\text{Prog}^k}_{V} P = (P \cup P') \cap \text{Prog}_{V \setminus x}^k
\]

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

Given a program \( P \in \text{Prog}_{V}^k \) and a type model \( M \) for \( T \), we define

\[
M \models v(t) \text{ iff } t \in M(v), \text{ with } t \text{ ground}
\]

\[
M \models A_1, \ldots, A_n \text{ iff } M \models A_i, \text{ for all } i = 1, \ldots, n (n \geq 0), \text{ with } A_1, \ldots, A_n \text{ ground}
\]

\[
M \models H : - B. \text{ iff } M \models B \text{ entails } M \models H, \text{ when } H : - B \text{ is ground}
\]

\[
M \models H : - B. \text{ iff } M \models H' : - B', \text{ for every ground instance } H' : - B'. \text{ of } H : - B, \text{ when } H : - B \text{ is not ground}
\]

\[
M \models P \text{ iff } M \models c \text{ for all } c \in P.
\]

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

For all \( V \in \wp_T(V) \), the programs in \( \text{Prog}_{V}^k \) are partially ordered as \( P_1 \preceq_T P_2 \) if and only if \( M_T(P_1) \subseteq M_T(P_2) \). We define \( P_1 \equiv_T P_2 \) if and only if \( P_1 \preceq_T P_2 \) and \( P_2 \preceq_T P_1 \).

The quotient of the set \( \text{Prog}_{V}^k \), w.r.t. \( \equiv_T \) is a complete lattice. The top element is the empty program and the greatest lower bound operation is the composition of programs (i.e. \( \star^{\text{Prog}_{V}^k} \)). From now on, every program will stand for its equivalence class.

Example 5.3

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

Since \( \text{Prog}_{V}^k \) and \( T_V \) are complete lattices, by defining a co-additive function \( \gamma^k : \text{Prog}_{V}^k \rightarrow T_V \) we conclude that \( \text{Prog}_{V}^k \) is an abstract interpretation of \( T_V \).

\(^5\)This folding could be applied more than once, to achieve a better precision. However, practical experiments have shown that one application leads to a sufficiently precise result.
DEFINITION 5.4
Let \( \langle \Delta, \Sigma, I \rangle \) be a type system, \( k \geq 1 \) and \( V \in \wp_f(V) \). We define
\[
\gamma(t') = v \in t, \quad B' = \bigwedge_{b \in B} b \quad \text{where } B \text{ is a sequence of } k\text{-atoms.}
\]

Let \( V_1, \ldots, V_n \) be the type variables in the clause \( H : \neg B \) and let \( P \in \text{Prog}_V^k \). We define
\[
\gamma^k_T(H : \neg B) = \bigwedge_{t_1, \ldots, t_n \in \text{terms}(\Delta, \emptyset)} H'[t_1/V_1] \cdots [t_n/V_n] : \neg B'[t_1/V_1] \cdots [t_n/V_n].
\]
\[
\gamma^k_T(P) = \bigwedge_{c \in P} \gamma^k_T(c).
\]

When it is clear from the context, we write \( \gamma^k \) for \( \gamma^k_T \). The abstraction map induced by \( \gamma^k \) will be called \( \alpha^k \).

PROPOSITION 5.5
Given a type system \( T, k \geq 1 \) and \( V \in \wp_f(V) \), \( \gamma^k_T : \text{Prog}_V^k \rightarrow TV \) is co-additive.

The operations of Definition 5.2 are correct or even optimal w.r.t. the corresponding operations of \( \text{Def}_{\Delta,V} \) (Definition 4.5).

PROPOSITION 5.6
Let \( T = \langle \Delta, \Sigma, I \rangle \) be a type system, \( k \geq 1 \) and \( V \in \wp_f(V) \).

(i) The operator \( \star_{\text{Prog}_V^k} \) is the best possible approximation of \( \star_{\text{Def}_{\Delta,V}} \).

(ii) The operator \( \exists_{\text{Prog}_V^k} \) is correct w.r.t. \( \exists_{\text{Def}_{\Delta,V}} \).

(iii) The operator \( \text{rename}_{\text{Prog}_V^k} \) is the best possible approximation of the operator \( \text{rename}_{\text{Def}_{\Delta,V}} \).

When \( k \) is finite, \( \text{Prog}_V^k \) is a finite set for every \( V \in \wp_f(V) \). Since the operations introduced in Definition 5.2 are algorithms, we conclude that \( \text{Prog}_V^k \) 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 \( \exists_{\text{Prog}_V^k} \) uses concrete unification between type terms. Since types are partially ordered with respect to subtyping (for instance, \( \text{nat} \leq \text{top} \)), the unification procedure used for folding might be too coarse. For instance, if we have a clause whose head is \( x(\text{list(nat)}) \) and we try to fold it in the body of a clause containing \( x(\text{list(top)}) \), 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 \( x(T) \), we can remove this \( k \)-atom from the body and instantiate the resulting clause with the substitution \( \{ T \mapsto \text{top} \} \), provided we have a top type \( \text{top} \). This is correct because every term is always in \( \text{top} \). In conclusion, the precision of the cylindrification operator can be improved by using a unification procedure which is aware of subtyping information.

To make \( \text{Prog}_V^k \) useful for program analysis, we need two algorithms: one that abstracts a concrete Herbrand constraint into an element of \( \text{Prog}_V^k \), and another algorithm that extracts from an element of \( \text{Prog}_V^k \) 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 composition of the abstraction maps seen so far, i.e. of \( \lambda h. \alpha^\beta(\alpha_{\text{def}} \cup \text{sol}_V(h)) \), 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 \( \text{Prog}^b \). Since, for the first time in this abstraction chain, \( \text{Prog}^b \) 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 \( \text{type}(\text{Term}, \text{Type}) \) 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 5.7**

Let \( T = (\Delta, \Sigma, I) \) be a type system, \( k \geq 1 \) and \( V \in \psi_T(V) \). We define the predicate \( \text{type}(\text{Term}, \text{Type}) \) so that a query of the form \( \text{type}(\text{Term}, \text{Type}) \), with \( \text{Term} \in \text{terms}(\Sigma, V) \) such that \( \text{vars}(\text{Term}) = X \) and \( \text{Type} \in \text{terms}(\Delta, \mathcal{T}) \), has computed answer \( \theta \) where \( \theta \) satisfies the following condition: for each \( \sigma \in \Theta_V \) and \( \mu : \mathcal{T} \rightarrow \text{terms}(\Delta, \emptyset) \) we have \( (\text{Term})\sigma \in [(\text{Type})\theta\mu]I \) if and only if \( \sigma(x) \in [\theta(x)\mu]I \) for every \( x \in X \).

**Example 5.8**

Figure 2 shows an example of the procedure \( \text{type}(\text{Term}, \text{Type}) \) for the type system \( \text{NL} \) of Example 4.2. That type system contains the types \( \text{top, nat} \) and polymorphic \( \text{lists} \). The query \( \text{type}([\text{H}, \text{L}], \text{Type}) \) yields a computed answer substitution \( \{\text{Type} \mapsto \text{list}(\text{S}), \text{H} \mapsto \text{S}, \text{L} \mapsto \text{list}(\text{S})\} \), meaning that the term \( [\text{H}, \text{L}] \) can be instantiated to a term of type \( \text{list}(\text{S}) \) if and only if \( \text{H} \) is instantiated to a term of type \( \text{S} \) and \( \text{L} \) to a term of type \( \text{list}(\text{S}) \), for every instantiation of \( \text{S} \).

A definition of the \( \text{type}(\text{Term}, \text{Type}) \) 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 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 5.9
Given a type system $T = \langle \Delta, \Sigma, I \rangle$, a procedure 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 $\text{vars}(c(x)) = \{x_1, \ldots, x_n\}$, we define

$$\alpha^{\text{def}}(c) = \{ \gamma^{\text{def}}(c) \} \cap \{ \gamma^{\text{def}}(c) \}$$

where $\gamma^{\text{def}}(c)$ is the set of solutions of the 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.

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

Proposition 5.11
Let $T = \langle \Delta, \Sigma, I \rangle$ be a type system, $k \geq 1$ and $V \in \wp_f(\mathcal{V})$. For all $\{P_1, P_2\} \subseteq \text{Prog}_V^t$, we have that

$$P_1 \leq P_2 \text{ entails } \gamma^h(P_1) \leq \gamma^h(P_2).$$

Since in general we have an infinite set of type models, Proposition 5.11 does not provide an algorithm for checking if $\gamma^h(P) \leq \gamma^h(\{v(d)\})$. This can be achieved, instead, by using the following result.

Proposition 5.12
Let $T = \langle \Delta, \Sigma, I \rangle$ be a type system, $k \geq 1$, $V \in \wp_f(\mathcal{V})$, $P \in \text{Prog}_V^t$, $v \in V$ and $d \in \text{terms}(\Delta, T)$. If $\{v(d)\}$ can be derived from $P$ by resolution (i.e. there is a refutation of $v(d)$ in $P$ with $\varepsilon$ as computed answer substitution), then $P \leq \{v(d)\}$. 

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 $P = \{v(\text{nat}),\} \text{ and } d = \text{top}$. We have $P \not\leq \{v(\text{top}),\}$ but $v(\text{top})$ 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 $Prog^b$ 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.

\[
\begin{align*}
\text{append}([], L, L). \\
\text{append}([H|T], L, [H|A]) : -\text{append}(T, L, A).
\end{align*}
\]

We want to compute its s-semantics by using the $Prog^2_\lambda$ constraint system.

The first step of the analysis consists in the transformation of the program in the abstract syntax of Definition 2.3. The result is

\[
\begin{align*}
\text{append}(x, y, z) & \leftarrow \{x = [], y = z\} \text{ or } (\{x = [H|t], z = [H|a]\} \text{ and } \text{append}(t, y, a))
\end{align*}
\]

The second step is in the abstraction of the program (Definition 2.14) through the $\alpha_{alg}^{\text{t}}$ map of Definition 5.9. We use the type constraint system $NL$ from Examples 4.2 and 4.20 and the type procedure of Example 5.8, given in Figure 2. The result of the abstraction of the program is the following.

\[
\begin{align*}
\text{append}(x, y, z) & \leftarrow \begin{cases}
  x(\text{list}(T)). \\
y(T) : -z(T). \\
z(T) : -y(T).
\end{cases} \\
\text{or } & \begin{cases}
x(\text{list}(T)) : -h(T), t(\text{list}(T)). \\
h(T) : -x(\text{list}(T)). \\
t(\text{list}(T)) : -x(\text{list}(T)). \\
z(\text{list}(T)) : -h(T), a(\text{list}(T)). \\
h(T) : -z(\text{list}(T)). \\
a(\text{list}(T)) : -z(\text{list}(T)).
\end{cases}
\end{align*}
\]

By definition we have

\[
T_P \cdot o(\bot)(\text{append}) = \emptyset.
\]
526 Generalizing Def and Pos to Type Analysis

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

\[
TP \uparrow_1(\perp)(\text{append}) = \begin{cases} 
\ell_1(\text{list}(T)), \\
\ell_2(T) : = \ell_3(T), \\
\ell_3(T) : = \ell_2(T).
\end{cases}
\]

Note that variables have been renamed into \( \ell \) variables, following Definition 2.6.

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 \( CA_{\text{Prog}_{\ell}^T}[\text{append}(t, y, a)] TP \uparrow_1(\perp) \). Following definition 2.5, we have

\[
CA_{\text{Prog}_{\ell}^T}[\text{append}(t, y, a)] TP \uparrow_1(\perp)
= \text{rename}_{\ell_1, \ell_2, \ell_3} \rightarrow (t, y, a) \begin{cases} 
\ell_1(\text{list}(T)), \\
\ell_2(T) : = \ell_3(T), \\
\ell_3(T) : = \ell_2(T).
\end{cases}
\]

\[
= \begin{cases} 
t(\text{list}(T)), \\
y(T) : = a(T), \\
a(T) : = y(T).
\end{cases}
\]

The second step is the computation of \( CA_{\text{Prog}_{\ell}^T}[Q \text{ and append}(t, y, a)] TP \uparrow_1(\perp) \). By Definition 2.5 we have

\[
CA_{\text{Prog}_{\ell}^T}[Q \text{ and append}(t, y, a)] TP \uparrow_1(\perp)
= CA_{\text{Prog}_{\ell}^T}[Q] TP \uparrow_1(\perp) \odot CA_{\text{Prog}_{\ell}^T}[\text{append}(t, y, a)] TP \uparrow_1(\perp)
= \{Q\} \odot_{\text{Prog}_{\ell}^T} CA_{\text{Prog}_{\ell}^T}[\text{append}(t, y, a)] TP \uparrow_1(\perp)
= \{Q\} \odot_{\text{Prog}_{\ell}^T} \begin{cases} 
t(\text{list}(T)), \\
y(T) : = a(T), \\
a(T) : = y(T).
\end{cases}
\]

\[
= \begin{cases} 
t(\text{list}(T)), \\
y(T) : = a(T), \\
a(T) : = y(T), \\
x(\text{list}(T)) : = h(T), t(\text{list}(T)). \\
h(T) : = x(\text{list}(T)), \\
t(\text{list}(T)) : = x(\text{list}(T)), \\
z(\text{list}(T)) : = h(T), a(\text{list}(T)), \\
h(T) : = z(\text{list}(T)), \\
a(\text{list}(T)) : = z(\text{list}(T)).
\end{cases}
\]

Following Definition 2.6, we must rename the program \( R \) defined in Equation (6.2) and
cylindrify w.r.t. the variables not in \( \{ t_1, t_2, t_3 \} \). The renaming operation yields

\[
\text{rename}_{\text{Prop}^2_v}^{\{l_1, l_2, l_3\}}(\{R\}) = \left\{ \begin{array}{l}
t_1(\text{list}(T)) : - \text{h}(T), t(\text{list}(T)). \\
t_2(\text{list}(T)) : - \text{a}(T), a(\text{list}(T)). \\
t_3(\text{list}(T)) : - \text{h}(T), a(\text{list}(T)). \\
\text{h}(T) : - t_1(\text{list}(T)). \\
\text{h}(T) : - t_2(\text{list}(T)). \\
\text{h}(T) : - t_3(\text{list}(T)). \\
\end{array} \right\}_{R^1}.
\]

We perform the cylindrification operation one variable at a time.

\[
\exists_{\text{Prop}^2_v}^{\{l\}}(\{R^1\}) = \left\{ \begin{array}{l}
t_2(\text{list}(T)) : - \text{a}(T), a(\text{list}(T)). \\
t_1(\text{list}(T)) : - \text{h}(T). \\
\text{h}(T) : - t_1(\text{list}(T)). \\
t_3(\text{list}(T)) : - \text{h}(T), a(\text{list}(T)). \\
\text{h}(T) : - t_2(\text{list}(T)). \\
a(\text{list}(T)) : - t_3(\text{list}(T)). \\
\end{array} \right\}_{R^2},
\]

\[
\exists_{\text{Prop}^2_v}^{\{a\}}(\{R^2\}) = \left\{ \begin{array}{l}
t_2(\text{list}(T)) : - t_3(\text{list}(T)). \\
t_1(\text{list}(T)) : - \text{h}(T). \\
t_3(\text{list}(T)) : - \text{h}(T), t_2(\text{list}(T)). \\
\text{h}(T) : - t_1(\text{list}(T)). \\
\text{h}(T) : - t_3(\text{list}(T)). \\
\end{array} \right\}_{R^3}.
\]

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 \( \lor \) construct. This yields the set of two programs

\[
T_P \uparrow_2(\bot)(\text{append}) = \left\{ \begin{array}{l}
t_1(\text{list}(T)). \\
t_2(\text{list}(T)) : - t_3(\text{list}(T)). \\
t_3(\text{list}(T)) : - t_3(\text{list}(T)). \\
\end{array} \right\} \cup \exists_{\text{Prop}^2_v}^{\{h\}}(\{R^3\})
\]

\[
= \left\{ \begin{array}{l}
t_1(\text{list}(T)). \\
t_2(\text{list}(T)) : - t_3(\text{list}(T)). \\
t_3(\text{list}(T)) : - t_3(\text{list}(T)). \\
\end{array} \right\}.
\]

Since it can be shown that \( T_P \uparrow_3(\bot) \) is equivalent to \( T_P \uparrow_2(\bot) \), we have \( S_P = T_P \uparrow_2(\bot) \).
in this fixpoint. We have used 
gram into the abstract syntax of Definition 2.3, then abstracts the program by using a generic 
We have implemented a small analyser for pure logic programs. It transforms a logic pro-
6.2 Type analysis of 

Since from both programs it is possible to derive the fact \( z(x\text{(top)}) \) by resolution, we 
conclude that \( z \) is bound to a list after the call of \texttt{append} with its first two arguments bound 
to lists (Proposition 5.12).

6.2 Type analysis of \texttt{derivative/2}

We have implemented a small analyser for pure logic programs. It transforms a logic pro-
gram into the abstract syntax of Definition 2.3, 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 \( Prog^\omega \) as constraint system. It can be specialized w.r.t. a given 
set of types, through the specification of the \texttt{type/2} procedure of Definition 5.7. 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 \texttt{top}, representing the whole set of terms; \texttt{int}, representing 
integers; \texttt{expr}, representing generic expressions on \( x \); and \texttt{algebraic}, representing expressions on \( x \) which do not involve exponentiation or trigonometric functions. We evaluate the query 

\( (x(\text{algebraic}).) \) and \texttt{derivative}(x,y)
More interestingly, the same constraints allow us to derive the fact in Figure 4. If the predicate \text{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 \( y \text{(expr)} \). This means that the second argument is bound to an expression. More interestingly, the same constraints allow us to derive the fact \( y \text{(algebraic)} \), 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.

<table>
<thead>
<tr>
<th>Constraint 1</th>
<th>Constraint 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>false : = x(int).</td>
<td>x(algebraic).</td>
</tr>
<tr>
<td>x[algebraic].</td>
<td>x[expr].</td>
</tr>
<tr>
<td>x[expr].</td>
<td>x[int].</td>
</tr>
<tr>
<td>y[algebraic].</td>
<td>y[algebraic].</td>
</tr>
<tr>
<td>y[expr].</td>
<td>y[expr].</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constraint 3</th>
<th>Constraint 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>false : = x[algebraic].</td>
<td>x[algebraic].</td>
</tr>
<tr>
<td>false : = x[int].</td>
<td>x[expr].</td>
</tr>
<tr>
<td>false : = y[algebraic].</td>
<td>x[algebraic].</td>
</tr>
<tr>
<td>false : = y[int].</td>
<td>y[expr].</td>
</tr>
<tr>
<td>x[algebraic].</td>
<td>y[algebraic].</td>
</tr>
<tr>
<td>x[expr].</td>
<td>y[expr].</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constraint 5</th>
<th>Constraint 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>false : = x[int].</td>
<td>x[algebraic].</td>
</tr>
<tr>
<td>false : = y[int].</td>
<td>x[expr].</td>
</tr>
<tr>
<td>x[algebraic].</td>
<td>false : = y[algebraic].</td>
</tr>
<tr>
<td>x[expr].</td>
<td>false : = y[int].</td>
</tr>
<tr>
<td>y[algebraic] : = x[algebraic].</td>
<td>y[expr] : = x[expr].</td>
</tr>
<tr>
<td>y[expr] : = x[expr].</td>
<td>y[expr] : = x[expr].</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constraint 7</th>
<th>Constraint 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>false : = x[algebraic].</td>
<td>x[algebraic].</td>
</tr>
<tr>
<td>false : = x[int].</td>
<td>false : = x[int].</td>
</tr>
<tr>
<td>false : = y[algebraic].</td>
<td>false : = y[algebraic].</td>
</tr>
<tr>
<td>false : = y[int].</td>
<td>false : = y[int].</td>
</tr>
<tr>
<td>x[algebraic].</td>
<td>x[algebraic].</td>
</tr>
<tr>
<td>x[expr] : = y[expr].</td>
<td>x[expr].</td>
</tr>
<tr>
<td>y[expr] : = x[expr].</td>
<td>y[expr].</td>
</tr>
</tbody>
</table>

**Figure 4.** The set of constraints computed for our query
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 \cite{2, 10, 11}. This leads to the use of transfinite formulas and operators on these 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 effectivity of the analysis has been considered only in the final abstraction to $P_{mg}^k$, 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 $Pos_\Delta$ type constraint system (Definition 4.29) should be investigated. We think that $Pos_\Delta$ should be represented by disjunctive logic programs instead of traditional logic programs.
- The operation $\exists_{Prog^\Delta}$ is correct (Proposition 5.6). 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 5.10). 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 \cite{24}.
- Subtyping information should be extracted from a type specification and used for improving the precision of the analysis.

Acknowledgements

We are grateful to Giorgio Levi for the original idea and contributions to its technical development. We would also like to thank the anonymous referee who made many useful suggestions for improving the submitted version of this paper.

Part of this work was done while Fausto Spoto was studying at the School of Computing of the University of Leeds: some of the costs of the visit being funded by EPSRC grant GR/M05645. The research for this paper was completed while Fausto Spoto was a Postdoctoral fellow at IRISA at Rennes under the supervision of Thomas Jensen and we are grateful for his support for this work.
References


Appendix

A Proofs

This appendix contains the proofs of the propositional statements in the main text.

Proofs for Section 2

Proof. [Proposition 2.7] This is a corollary of Definition 2.6, since it follows by structural induction on a goal \( G \) that \( CA_{Dv} [\emptyset] \) is additive:

\[
CA_{Dv} [G] \cup_{j \in J} \{ f_j \} = \bigcup_{j \in J} \left( CA_{Dv} [G] f_j \right)
\]

for every \( \{ f_j \}_{j \in J} \subseteq D_{Dv} \) with \( J \subseteq N \). Since, by Definition 2.5, \( \bigcup_{v \in V} \text{terms} \) and \( \text{terms} \cup_{v \in V} \) are additive, we conclude that \( T_{Dv} \) is additive.

Proof. [Proposition 2.9] Since \( \theta \in \text{sol}_{V} \{ \text{Eq}(\theta) \} \) and \( \text{sol}_{V} \{ \text{Eq}(\theta) \} \) is downward closed, we conclude that \( \text{sol}_{V} \{ \text{Eq}(\theta) \} \subseteq \downarrow \{ \theta \} \). To show that \( \text{sol}_{V} \{ \text{Eq}(\theta) \} \subseteq \downarrow \{ \theta \} \), suppose \( \sigma \in \text{sol}_{V} \{ \text{Eq}(\theta) \} \). This means that \( \text{Eq}(\theta) \sigma \) is true, i.e., that \( \sigma(x) = \theta(x) \sigma \) for every \( x \in \text{dom}(\theta) \). This entails that \( \sigma = \theta \circ \sigma \), i.e., \( \sigma \leq \theta \). Therefore, \( \sigma \in \downarrow \{ \theta \} \).

Proofs for Section 3

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

Proposition A.1

Given \( S \in \Theta_{V} \) and \( x \in V \), we have

\[
\exists x \ Do_{Dv} \text{un} V S = \downarrow \{ S' |_{V \setminus x} \}
\]

where

\[
S' = \{ \theta \circ \{ x \mapsto a \} \in \Theta_{V} |_{x} \bigcup_{u \in \text{terms}(\Sigma, V)} \} \text{ and } \theta \in S \}
\]

Proof. Let \( \theta' \in \exists x \ Do_{Dv} \text{un} V S \). By definition, we have \( \theta' = \theta |_{V \setminus x} \subseteq_{V \setminus x} \sigma, \sigma = \sigma'[n/x] \) and \( \sigma' \in S \) for suitable \( \theta, \sigma \) and \( \sigma' \). Hence \( \theta = \sigma \circ \sigma'' \) for a suitable \( \sigma'' \). Since \( \sigma \circ \sigma'' \) is idempotent, \( \sigma' \circ \sigma'' \in \Theta_{V} \) is also idempotent. Moreover, it belongs to \( S \) by downward closure and we have:

\[
\theta |_{V \setminus x} = (\sigma' \circ \sigma'' |_{V \setminus x}) \circ \{ x \mapsto \sigma''(n) \} |_{V \setminus x} \in S' |_{V \setminus x}
\]

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

\[
\theta |_{V \setminus x} (y) = (\sigma \circ \sigma'') |_{V \setminus x} (y) = (\sigma'[n/x] \circ \sigma'') |_{V \setminus x} (y)
\]

\[
= \sigma'[n/x](y) \sigma'' = \sigma'(y) |_{n/x} \sigma''
\]

\[
= (\sigma'(y) |_{\sigma'' |_{V \setminus x}}) \{ x \mapsto \sigma''(n) \}
\]

\[
= (\sigma' |_{\sigma'' |_{V \setminus x}} \circ \{ x \mapsto \sigma''(n) \}) (y)
\]

\[
= (\sigma' |_{\sigma'' |_{V \setminus x}} \circ \{ x \mapsto \sigma''(n) \}) |_{V \setminus x} (y)
\]
Generalizing Def and Pos to Type Analysis

Therefore,

\[ \theta = \theta \mid_{V \setminus \sigma} \circ \{ x \mapsto \theta(x) \} \in \downarrow(S' \mid_{V \setminus \sigma}). \]

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

\[ \theta'' = \theta' \circ \sigma = [\theta[n/x] \circ \{ n \mapsto u \}] \mid_{V \setminus \sigma} \circ \sigma = [\theta[n/x] \circ \{ n \mapsto u \} \circ \sigma] \mid_{V \setminus \sigma}. \]

Therefore, \( \theta' \in \Xi_{\Downarrow V}' \).

**Lemma A.2**

Let \( V \in \mathcal{P}^l(V) \), \( \bar{x}, \bar{y} \in \text{Seq}(V) \) disjoint, of the same length and without repetitions and \( \sigma_1, \sigma_2 \in \Theta_V \). Then

\[ \sigma_1 \leq \sigma_2 \text{ if and only if } \sigma_1[\bar{y}/\bar{x}, \bar{x}/\bar{y}] \leq \sigma_2[\bar{y}/\bar{x}, \bar{x}/\bar{y}] \).

**Proof.** Assume \( \sigma_1 \leq \sigma_2 \), i.e. \( \sigma_1 = \sigma_2 \sigma \) for a suitable \( \sigma \). We show that \( \sigma_1[\bar{y}/\bar{x}, \bar{x}/\bar{y}] = [\sigma_2[\bar{y}/\bar{x}, \bar{x}/\bar{y}]](\sigma[\bar{y}/\bar{x}, \bar{x}/\bar{y}]) \).

Assume \( v \notin \bar{x} \cup \bar{y} \). We have

\[ \sigma_1[\bar{y}/\bar{x}, \bar{x}/\bar{y}][v] = \sigma_2[\bar{y}/\bar{x}, \bar{x}/\bar{y}][v] \]

(i) By using the notion of Proposition A.1, we have \( \sigma_1[\bar{y}/\bar{x}, \bar{x}/\bar{y}][v] = \sigma_2[\bar{y}/\bar{x}, \bar{x}/\bar{y}][v] \).

(ii) Proposition A.1 shows that it is the downward closure of a set of substitutions.

(iii) By the intersection is closed on the set of downward closed sets of substitutions.

(iv) Proposition A.1 shows that it is the downward closure of a set of substitutions.

(v) Let \( \sigma_2 \in \text{rename}_{\downarrow S'}(S) \) and \( \sigma_1 \leq \sigma_2 \). Since \( \sigma_2 = \sigma_2'[\bar{y}/\bar{x}, \bar{x}/\bar{y}] \) for some \( \sigma_2' \in S \), by Lemma A.2 we have \( \sigma_1[\bar{y}/\bar{x}, \bar{x}/\bar{y}] \leq \sigma_2[\bar{y}/\bar{x}, \bar{x}/\bar{y}] = \sigma_2' \). From \( S = \downarrow S \) we conclude that \( \sigma_1[\bar{y}/\bar{x}, \bar{x}/\bar{y}] \in S \). Then \( \sigma_1 \in \text{rename}_{\downarrow S}^\text{L}(S) \).

**Proof.** [Proposition 3.3]

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

(ii) Proposition A.1 shows that it is the downward closure of a set of substitutions.

(iii) Proposition A.1 shows that it is the downward closure of a set of substitutions.

(iv) Let \( S' = S \). Choosing \( u = x \), we have \( \theta \in S' \). If \( x \notin \text{dom}(\theta) \) then \( \theta[n/x] = \theta \).

(v) Let \( \sigma_2 \in \text{rename}_{\downarrow S}^\text{L}(S) \) and \( \sigma_1 \leq \sigma_2 \). Since \( \sigma_2 = \sigma_2'[\bar{y}/\bar{x}, \bar{x}/\bar{y}] \) for some \( \sigma_2' \in S \), by Lemma A.2 we have \( \sigma_1[\bar{y}/\bar{x}, \bar{x}/\bar{y}] \leq \sigma_2[\bar{y}/\bar{x}, \bar{x}/\bar{y}] = \sigma_2' \). From \( S = \downarrow S \) we conclude that \( \sigma_1[\bar{y}/\bar{x}, \bar{x}/\bar{y}] \in S \). Then \( \sigma_1 \in \text{rename}_{\downarrow S}^\text{L}(S) \).

**Proof.** [Proposition 3.4]

We assume the constant in \( \Sigma \) to be \( a \) and the functor to be the unary functor \( f \), though the proof can be easily generalized to greater arities.

The non-strict inclusion is a consequence of the fact that \( \text{solve } \exists_W c \) 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 $M : N \to N$, 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:

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

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

**Proof.** [Proposition 3.5(i)] Assume $h_1 = \exists W, c_1$ and $h_2 = \exists W, c_2$ with $W_1 \cap W_2 = \emptyset$.

Let $\theta' \in \text{sol}_V(h_1 \ast h_2)$. Then $\theta' = \emptyset[V, \theta \in \Theta_{V \cup W_1, U \cup W_2, V} \cap c_1 \theta$ and $c_2 \theta$ are true. Hence, we have $\theta[V, U, h_1 \ast h_2, V] = \text{sol}_V(h_1)$ for $i = 1, 2$. This means that $\theta' \in \text{sol}_V(h_1) \cap \text{sol}_V(h_2) = \text{sol}_V(h_1 \ast h_2)$. Conversely, let $\theta' \in \text{sol}_V(h_1 \ast h_2)$. Then $\theta' = \emptyset[V, \theta \in \Theta_{V \cup W_1, U \cup W_2, V} \cap c_1 \theta$ and $c_2 \theta$ are true. Hence there exist $\theta_1 \in \text{sol}_V(h_1 \ast h_2)$ and $\theta_2 \in \text{sol}_V(h_1 \ast h_2)$ such that $\theta'_1[V, \theta = \emptyset \text{ and } c_1 \theta$ is true for $i = 1, 2$. Since $\theta_1$ and $\theta_2$ 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 $\theta \in \Theta_{V \cup W_1, U \cup W_2, V}$ and $c \theta$ is true for $i = 1, 2$. Therefore, $\theta[V, \emptyset = \theta \in \text{sol}_V(h_1 \ast h_2)$.

**Proof.** [Proposition 3.5(ii)] We have to prove that

$$\exists D \ast V, \text{sol}_V(\exists W \subset c) = \text{sol}_V(\exists W \cup U \cup \{N \mapsto x\})$$

for every $x \in V$.

Let $\theta' \in \exists D \ast V, \text{sol}_V(\exists W \subset c)$. We have, by definition, $\theta' = \theta' \in V \cup W, \sigma$ and $\sigma \in \{\text{sol}_V(\exists W \subset c)n[x]\} \mapsto [\emptyset[V, \theta \in \Theta_{V \cup W_1, U \cup W_2, V} \cap c_1 \theta$ and $c_2 \theta$ are true. The substitution $\sigma'' = \sigma' \circ \{\emptyset[V, \emptyset = \theta \in \text{sol}_V(h_1 \ast h_2)$] is such that $e(N[x])\sigma''$ 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 $e(N[x])$ contains neither $x$ nor $n$, we have $t_1[N/x]\sigma'' = t_2[N/x]\sigma''$ and $e(N[x])\sigma'' \in V, U \cup W$. This allows us to conclude that $e(N[x])\sigma'' \in V, U \cup W$. We have:

$$\emptyset[V, \emptyset = \theta \in \text{sol}_V(h_1 \ast h_2) \subseteq \text{sol}_V(h_1 \ast h_2) \subseteq \emptyset[V, \emptyset = \theta \in \text{sol}_V(h_1 \ast h_2) \subseteq \text{sol}_V(h_1 \ast h_2).$$

Finally, we have:

$$\emptyset[V, \emptyset = \theta \in \text{sol}_V(h_1 \ast h_2 \subseteq \text{sol}_V(h_1 \ast h_2).$$

Assume, conversely, that $\emptyset[V, \emptyset = \theta \in \text{sol}_V(h_1 \ast h_2 \subseteq \text{sol}_V(h_1 \ast h_2).$ Therefore $\emptyset[V, \emptyset = \theta \in \text{sol}_V(h_1 \ast h_2 \subseteq \text{sol}_V(h_1 \ast h_2).$ Therefore $\emptyset[V, \emptyset = \theta \in \text{sol}_V(h_1 \ast h_2 \subseteq \text{sol}_V(h_1 \ast h_2).$ Therefore $\emptyset[V, \emptyset = \theta \in \text{sol}_V(h_1 \ast h_2 \subseteq \text{sol}_V(h_1 \ast h_2).$ Therefore $\emptyset[V, \emptyset = \theta \in \text{sol}_V(h_1 \ast h_2 \subseteq \text{sol}_V(h_1 \ast h_2).$ Therefore $\emptyset[V, \emptyset = \theta \in \text{sol}_V(h_1 \ast h_2 \subseteq \text{sol}_V(h_1 \ast h_2).$

The thesis follows by Proposition A.1.

**Lemma A.3**

Let $V \subseteq \mathcal{P}(V), W \subseteq \mathcal{P}(V)$, $\bar{x}, \bar{y} \in \text{Seq}(V)$ disjoint, of the same length and without repetitions, $t_1, t_2 \in \text{term}_1(\mathcal{V}, \{V \setminus \bar{y} \cup \bar{W} \})$ and $\theta \in \Theta_{V \cup W, V}$. Then $t_1 \theta = t_2 \theta$ if and only if $t_1[\bar{y}/\bar{x}]\theta[\bar{y}/\bar{x}, \bar{y}/\bar{y}] = t_2[\bar{y}/\bar{x}]\theta[\bar{y}/\bar{x}, \bar{y}/\bar{y}]$.

**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 $t_1[\bar{y}/\bar{x}]\theta[\bar{y}/\bar{x}, \bar{y}/\bar{y}] = t_2[\bar{y}/\bar{x}]\theta[\bar{y}/\bar{x}, \bar{y}/\bar{y}]$ by induction on the depth of $t_1$. If $t_1 = v \not\in \bar{x} \cup \bar{y}$ we have

$$t_1[\bar{y}/\bar{x}]\theta[\bar{y}/\bar{x}, \bar{y}/\bar{y}] = v[\theta[\bar{y}/\bar{x}, \bar{y}/\bar{y}] = \emptyset(v)[\bar{y}/\bar{x}, \bar{y}/\bar{y}] = (\emptyset(v)[\bar{y}/\bar{x}, \bar{y}/\bar{y}] = (\emptyset(v)[\bar{y}/\bar{x}, \bar{y}/\bar{y}].$$
536 Generalizing Def and Pos to Type Analysis

If \( t_1 = x_i \) we have
\[
  t_1[y/x][\theta[y/x, \bar{x}/\bar{y}]] = y_i(\theta[y/x, \bar{x}/\bar{y}]) = \theta(x_i)[y/x, \bar{x}/\bar{y}]
\]  
\[
  = (t_2 \theta)[y/x, \bar{x}/\bar{y}] = (t_2[y/x][\theta[y/x, \bar{x}/\bar{y}]]).
\]

Note that the case \( t_1 = y_i \) is impossible since \( t_1 \in \mathrm{terms}(\Sigma, (V \setminus \bar{y}) \cup W) \). If \( t_1 = \mathbf{\not}{t_1, \ldots, t'_1} \) then \( t_2 = \mathbf{t_2, \ldots, t'_2} \) and \( t''_2 = t''_2 \theta \) for every \( j = 1, \ldots, t \). Thus we have
\[
  t_1[y/x][\theta[y/x, \bar{x}/\bar{y}]] = \mathbf{t_2, \ldots, t'_2} \cdot \mathbf{t_2, \ldots, t'_2} \theta[y/x, \bar{x}/\bar{y}]] = \mathbf{t_2, \ldots, t'_2} \theta[y/x, \bar{x}/\bar{y}]] = \mathbf{t_2}[y/x][\theta[y/x, \bar{x}/\bar{y}]]
\]
(by ind. hyp.)
\[
  \mathbf{t_2}[y/x][\theta[y/x, \bar{x}/\bar{y}]] = \mathbf{t_2}[y/x][\theta[y/x, \bar{x}/\bar{y}]] = \mathbf{t_2}[y/x][\theta[y/x, \bar{x}/\bar{y}]]
\]
and
\[
  \mathbf{t_2}[y/x][\theta[y/x, \bar{x}/\bar{y}]] = \mathbf{t_2}[y/x][\theta[y/x, \bar{x}/\bar{y}]] = \mathbf{t_2}[y/x][\theta[y/x, \bar{x}/\bar{y}]]
\]

The converse holds since \( \theta[y/x][\bar{x}/\bar{y}] = \theta \) for every \( \theta \in \Theta_{V \cup W \cup V} \).

PROOF. [Proposition 3.5(iii)] Let \( h = \exists \bar{v} c \). Let \( \theta \in \text{rename}_{\bar{x} \to \bar{y}}^{\text{Down}_V \left( \text{Sol}_V \right)} (h) \). We have \( \theta = \theta' \left[ y/x, \bar{x}/\bar{y} \right] \) with \( \theta' = \theta'' \left[ y/x, \bar{x}/\bar{y} \right] \) and \( \theta'' \) is true. By Lemma A.3 we have that \( c(y/x)[\theta[y/x, \bar{x}/\bar{y}]] \) is true. Since \( \theta'(y/x, \bar{x}/\bar{y}) = \theta''(y/x, \bar{x}/\bar{y}) \), we have \( \theta'(y/x, \bar{x}/\bar{y}) \in \text{sol}_V \{ \text{rename}_{\bar{x} \to \bar{y}}^{\text{Down}_V \left( \text{Sol}_V \right)} (h) \} \). Conversely, let \( \theta \in \text{sol}_V \{ \text{rename}_{\bar{x} \to \bar{y}}^{\text{Down}_V \left( \text{Sol}_V \right)} (h) \} \). Then \( \theta = \theta'' \left[ y/x, \bar{x}/\bar{y} \right] \) with \( \theta'' \in \Theta_{V \cup W \cup V} \) and \( c(y/x)[\theta''] \) is true. By Lemma A.3 we have that \( c(y/x)[\theta''] \) is true. Then \( \theta''(y/x, \bar{x}/\bar{y}) \) is true. Then \( \theta''(y/x, \bar{x}/\bar{y}) \in \text{sol}_V (h) \) and \( \theta \in \text{rename}_{\bar{x} \to \bar{y}}^{\text{Down}_V \left( \text{Sol}_V \right)} (h) \).

PROOF. [Proposition 3.8] Since \( \psi(\text{Sol}_V) \) and \( \text{Down}_V \) are set-theoretic lattices, their least upper bound operator is \( \cup \). Therefore,
\[
  \cup \left( \text{rename}_{\bar{x} \to \bar{y}}^{\text{Down}_V \left( \text{Sol}_V \right))} (p_i \mid i \in I) \right) = \cup \left( \cup p_i \mid i \in I \right) = \text{rename}_{\bar{x} \to \bar{y}}^{\text{Down}_V \left( \text{Sol}_V \right)} (\cup p_i).
\]

PROOF. [Proposition 3.9]

(i)
\[
  \cup (p_1 \ast \psi(\text{Sol}_V) \mid p_2) = \{ s_1 \ast \psi(\text{Sol}_V) s_2 \mid s_1 \in p_1 \text{ and } s_2 \in p_2 \}
\]
\[
  = \{ s_1 \cap s_2 \mid s_1 \in p_1 \text{ and } s_2 \in p_2 \}
\]
\[
  = (\cup p_1) \cap (\cup p_2) = (\cup p_1) \ast \text{rename}_{\bar{x} \to \bar{y}}^{\text{Down}_V \left( \text{Sol}_V \right)} (\cup p_2).
\]

(ii)
\[
  \cup \{ \exists \bar{x} \psi(\text{Sol}_V) \mid p \} = \{ \exists \bar{x} \psi(\text{Sol}_V) s \mid s \in p \}
\]
\[
  = \text{rename}_{\bar{x} \to \bar{y}}^{\text{Down}_V \left( \text{Sol}_V \right)} (\cup \{ s \mid s \in p \}) = \text{rename}_{\bar{x} \to \bar{y}}^{\text{Down}_V \left( \text{Sol}_V \right)} (\cup p).
\]

(iii)
\[
  \cup \{ \text{rename}_{\bar{x} \to \bar{y}}^{\psi(\text{Sol}_V)} \mid p \} = \{ \text{rename}_{\bar{x} \to \bar{y}}^{\psi(\text{Sol}_V)} s \mid s \in p \}
\]
\[
  = \{ \text{rename}_{\bar{x} \to \bar{y}}^{\text{Down}_V \left( \text{Sol}_V \right)} s \mid s \in p \}
\]
\[
  = \text{rename}_{\bar{x} \to \bar{y}}^{\text{Down}_V \left( \text{Sol}_V \right)} (\cup \{ s \mid s \in p \}) = \text{rename}_{\bar{x} \to \bar{y}}^{\text{Down}_V \left( \text{Sol}_V \right)} (\cup p).
\]
**Proofs for Section 4**

**Definition A.4**
Given a type constraint system \( T = \{ \{ T_v \} \} \) for the type system \( \{ \Sigma, I \} \) and \( \{ \theta, \phi \} \subseteq \Theta \), \( \theta_1 \) and \( \theta_2 \) are type-equivalent if and only if for every \( v \in V \) and every \( d \in \text{terms}(\Delta, \emptyset) \) we have \( \theta_1(v) = [d]I \) and only if \( \theta_2(v) = [d]I \).

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

**Proposition A.5**
Given a type constraint system \( T = \{ \{ T_v \} \} \) for the type system \( \{ \Delta, \Sigma, I \} \), two type-equivalent substitutions \( \theta_1 \) and \( \theta_2 \) in \( \Theta \) and \( \phi \in \Phi \), we have \( \phi = [\theta_1] = [\theta_2] \).

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

**Proof.** [Proposition 4.9] Let \( \theta \in \Theta \). We define

\[
\theta' = \begin{cases} \theta(y) & \text{if } x \not\in \text{vars}(\theta(y)) \\
\theta(y)[x \mapsto y] & \text{otherwise.}
\end{cases}
\]

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

We rewrite the type-cylindrification operator of Definition 4.5 in a way which simplifies the following proofs.

**Definition A.6**
Given \( t \in \text{terms}(\Sigma, V) \), we define

\[
x \in d \ [t/x] = \begin{cases} \text{true} & \text{if } t \in [d]I \\
false & \text{otherwise}
\end{cases}
\]

\[
(x \not\in \text{vars}(\theta(y)) = \begin{cases} \text{true} & \text{if } x \not\in \text{vars}(\theta(y)) \\
false & \text{otherwise}
\end{cases}
\]

\[
\emptyset \Rightarrow \phi \ [t/x] = (\phi_1 [t/x] \Rightarrow (\phi_2 [t/x])
\]

**Proposition A.7**
Given a type constraint system \( T = \{ \{ T_v \} \} \) for the type system \( \{ \Delta, \Sigma, I \} \), \( V \in \mu J \) and \( x \in V \),

\[
\exists \theta_\phi \ [t/x] = \text{terms}(\Sigma, V) \}
\]

for all \( \phi \in T \).

**Proof.** For all \( t \in \text{terms}(\Sigma, V) \), let \( P_t \subseteq [d]I \). 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 4.5).

**Proposition 4.13** [For the co-additivity of \( \gamma \), let \( S \subseteq T \). We have

\[
\gamma(\land S) = \{ \theta \in \Theta \} \ \text{for all } \theta \in \text{terms}(\Delta, \emptyset) \ 	ext{and all } \phi \in S \}
\]

Moreover, since \( \land 0 \in T \) and \( \gamma(\land 0) = \Theta \), which is the top of \( \text{Down}_V \), we conclude that \( \gamma(T) \) is topped.

Finally, since for all \( S \subseteq T \) we know that \( \land S \in T \), the above result about co-additivity entails that the set \( \gamma(T) \) is completely \( \cup \)-closed.

**Proposition 4.19** [Given \( V \in \mu J \) and \( \{ t_1, \ldots, t_n \} \subseteq \text{terms}(\Sigma, V) \), consider \( \sigma \in T \) such that \( \sigma[v] = t \) for any \( v \in V \). For all \( i = 1, \ldots, n \) and \( d \in \text{terms}(\Delta, \emptyset) \), if \( t_i \in [d]I \) then, by the downward closure of types, \( t_i \sigma \in [d]I \). Conversely, if \( t_i \sigma \not\in [d]I \) we conclude that \( t_i \not\in [d]I \) by the choice of \( t_i \), since \( t_i \) is obtained from \( t_i \sigma \) by substituting some instances of \( t \) with variables in \( V \subseteq V \). Therefore, \( T \) is structural.
538 Generalizing Def and Pos to Type Analysis

PROOF. [Proposition 4.24] Let $T = \{ T_V : V \in P_f(V) \}$ be a type constraint system for the type system $(\Delta, \Sigma, I)$. 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 $t \in \text{terms}(\Sigma, V)$.

If $T$ is positive, let $\Delta = \{ e \}$ be its only type. If $\llbracket e \rrbracket I \neq \emptyset$, let $t'$ be a ground term in $\llbracket e \rrbracket I$ (otherwise, we do not need such a term). Define $\theta'$ such that

$$\theta'(x) = \begin{cases} t' & \text{if } t \in \llbracket e \rrbracket I \\ x & \text{otherwise} \end{cases}$$

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

$$\llbracket \phi_1[t/x] \rrbracket \theta = \llbracket \phi_1[t/x] \rrbracket \theta'$$

(by the choice of $t'$) = $\llbracket \phi_1 \rrbracket \theta'$

(since $\phi_1 \equiv \phi_2$) = $\llbracket \phi_2 \rrbracket \theta'$

(as above) = $\llbracket \phi_2[t/x] \rrbracket \theta$.

Therefore, $\phi_1[t/x] \equiv \phi_2[t/x]$. If $T$ is structural, we know that there exists a substitution $\theta'$ grounding for $V$ which is type-equivalent to $\theta$. Consider $\theta'' = \theta'' |_{V \setminus x} \circ \{ x \to t' \}$, 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 $\gamma(\phi_1) \not\equiv \gamma(\phi_2)$. Let $\theta$ be such that $\llbracket \phi_1 \rrbracket \theta = 1$ and $\llbracket \phi_2 \rrbracket \theta = 0$ (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 $\theta''$ which is type-equivalent to $\theta$ and belongs to $\gamma(\phi_1)$. Since $\llbracket \phi_2 \rrbracket \theta'' = [\phi_2] \theta = 0$ entails $\theta'' \not\equiv \gamma(\phi_2)$, this will entail the thesis.

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

$$\theta'(v) = \begin{cases} t & \text{if } \theta(v) \in \llbracket e \rrbracket I \\ z & \text{otherwise.} \end{cases}$$

By construction, $\theta'$ is idempotent and type-equivalent to $\theta$. Moreover, every instance of $\theta'$ is type-equivalent to $\theta'$ or it binds every variable to a term in $\llbracket e \rrbracket I$. Since $\phi_1$ is positive, we have $\theta' \in \gamma(\phi_1)$, as required.

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

PROOF. [Proposition 4.25(i)] We have to show that

$$\alpha \left( \gamma(\phi_1) \ast D_{\text{new}} \gamma(\phi_2) \right) \leq \phi_1 \ast T_V \phi_2$$

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

$$\alpha \left( \gamma(\phi_1) \ast D_{\text{new}} \gamma(\phi_2) \right) = \alpha \left( \gamma(\phi_1) \cap \gamma(\phi_2) \right)$$

(Proposition 4.13) = $\alpha(\gamma(\phi_1 \land \phi_2))$

($\alpha \gamma$ is reductive) $\leq \phi_1 \land \phi_2 = \phi_1 \ast T_V \phi_2$.

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

PROOF. [Proposition 4.25(ii)] For the result about correctness, it suffices to show that

$$\exists_{\text{new}} \gamma(\phi) \leq \gamma \left( \exists_{\text{new}} \gamma(\phi) \right)$$

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

Let $\theta \in \exists_{\text{new}} \gamma(\phi)$. Then there exists $\theta' \in \gamma(\phi)$ such that $\theta = (\theta' |_{V \setminus \nu} \circ \{ x \to u \}) \circ \sigma$, for suitable $\sigma \in \Theta_V$ and $\nu \in \text{terms}(\Sigma, V)$ (Proposition A.1). Then

$$\theta |_{V \setminus \nu} = ((\theta' |_{V \setminus \nu} \circ \{ x \to u \}) \circ \sigma |_{V \setminus \nu} = \left( \{ x \to u \} \circ \{ x \to u \} \circ \sigma \right) |_{V \setminus \nu}.$$
Let $\theta'' = (\theta' \circ \{x \mapsto u\}) \circ \sigma$. We have $\theta'' \leq \theta'$. Then $[\theta][\theta''] = 1$, which entails that $[\theta][\theta'] = 1$ if and only if $[\theta'][\nu \upsilon] = 1$. This means that $[\Theta^2 \nu \upsilon][\theta] = 1$, because $\theta'[\nu \upsilon] = \theta''[\nu \upsilon]$. Since this is true for every $\theta \in \Xi_{\nu \upsilon}^{\nu \upsilon}[\gamma(\phi)]$ and $\Xi_{\nu \upsilon}^{\nu \upsilon}[\gamma(\phi)]$ is downward closed (Proposition 3.3(ii)), we have the thesis.

Let $T$ be positive or structural. Let $\theta$ be such that $[\Theta^2 \nu \upsilon][\theta] = 1$. We will show that, if $T$ is positive or structural, there exists a substitution $\theta'$ type-equivalent to $\theta$ such that $\theta' \in \Xi_{\nu \upsilon}^{\nu \upsilon}[\gamma(\phi)]$. By extensivity, it follows that $\theta' \in \gamma(\alpha_x \Xi_{\nu \upsilon}^{\nu \upsilon}[\gamma(\phi)])$ and hence $\alpha_x \Xi_{\nu \upsilon}^{\nu \upsilon}[\gamma(\phi)] \leq 1$. However, $\theta$ and $\theta'$ are type-equivalent so that $\alpha_x \Xi_{\nu \upsilon}^{\nu \upsilon}[\gamma(\phi)] \leq 1$. Therefore, we conclude that $\Xi_{\nu \upsilon}^{\nu \upsilon}[\gamma(\phi)] \leq 1$. If $T$ is positive, then let $e^\alpha$ be its only type. Let $t$ be a ground term in $\mathcal{C}_I$, whenever $\mathcal{C}_I$ is not empty (otherwise, we do not need such a $t$). Let $z \in V$ be arbitrary. Let us define

$$
\theta'(v) = \begin{cases} 
t & \text{if } \theta(v) \in \mathcal{C}_I 
z & \text{otherwise}
\end{cases}
$$

for every $v \in V$. By construction, $\theta'$ is idempotent and type-equivalent to $\theta$. Then $[\Theta^2 \nu \upsilon][\theta'] = 1$. Thus there exists a term $t'$ such that $[\theta[t'/x]][\theta'] = 1$, i.e., $[\theta][\theta'] = [x \mapsto t'] = 1$, where

$$
t' = \begin{cases} 
t & \text{if } t' \in \mathcal{C}_I 
z & \text{otherwise}.
\end{cases}
$$

Let $\theta'' = \theta'|_{\nu \upsilon} \circ \{x \mapsto t'\}$. Every instance of $\theta''$ is type-equivalent to $\theta'$ or binds every variable to a term in $\mathcal{C}_I$. Since $\phi$ is positive, we have $\theta'' \in \gamma(\phi)$. Then $\theta'' \in \Xi_{\nu \upsilon}^{\nu \upsilon}[\gamma(\phi)]$ (Proposition A.1).

If $T$ is structural, we know that there exists a grounding substitution $\theta''$ which is type-equivalent to $\theta$. Then $[\Theta^2 \nu \upsilon][\theta''] = 1$. Then there exists a term $t''$ such that $[\theta[t''/x]][\theta] = 1$, i.e., $[\theta][\theta'] = [x \mapsto t''] = 1$, 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 $\theta'' = \theta'|_{\nu \upsilon} \circ \{x \mapsto t''\}$. It is guarding for $V$. Therefore, every instance of $\theta''$ is $\theta''$ itself and $\theta'' \in \gamma(\phi)$. Then $\theta'' \in \Xi_{\nu \upsilon}^{\nu \upsilon}[\gamma(\phi)]$ (Proposition A.1).

**Lemma A.8**

Let $V \in pB(V), \bar{x}, \bar{y} \in Seq(V)$ disjoint, of the same length and whose intersection is empty, $T = (\Delta, \Sigma, I)$ be a type system, $\phi \in \Phi_{\Delta, V}$ and $\theta \in \Theta_{\nu \upsilon}$. We have

$$
[\theta][\theta'] = [\theta[y/x][\bar{x}, \bar{y}]]_{T}[\theta[y/x][\bar{x}, \bar{y}]]_{T}.
$$

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

**Proof.** We prove that, given $\bar{x}, \bar{y} \in Seq(V)$ and $\phi \in \Phi_{\Delta, V \setminus \bar{y}}$, we have

$$
\text{rename}_{\Xi_{\nu \upsilon}^{\nu \upsilon}}[\gamma(\phi)] = \gamma(\text{rename}_{\Xi_{\nu \upsilon}^{\nu \upsilon}}[\gamma(\phi)]).
$$

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

Let $\theta \in \text{rename}_{\Xi_{\nu \upsilon}^{\nu \upsilon}}[\gamma(\phi)]$. Then $\theta = \theta'[y/x, \bar{x}, \bar{y}]$ with $\theta' \in \gamma(\phi)$. We have

$$
\text{rename}_{\Xi_{\nu \upsilon}^{\nu \upsilon}}[\gamma(\phi)][\theta'] = [\theta[y/x][\bar{x}, \bar{y}]]_{T}.
$$

(since $\phi \in \Phi_{\Delta, V \setminus \bar{y}}$)

$$
[\theta[y/x][\bar{x}, \bar{y}]][\theta'] = [\theta[y/x][\bar{x}, \bar{y}]][\theta'[y/x, \bar{x}, \bar{y}]].
$$

(Lemma A.8)

$$
[\theta][\theta'] = 1.
$$

Since $\theta \leq \theta'$ and $\text{rename}_{\Xi_{\nu \upsilon}^{\nu \upsilon}}[\gamma(\phi)]$ is downward closed (Proposition 3.3(ii)), we have $\theta \in \gamma(\text{rename}_{\Xi_{\nu \upsilon}^{\nu \upsilon}}[\gamma(\phi)]).

Conversely, assume $\theta \in \gamma(\text{rename}_{\Xi_{\nu \upsilon}^{\nu \upsilon}}[\gamma(\phi)]).$ Consider $\theta' \leq \theta[y/x, \bar{x}, \bar{y}]$, i.e., $\theta' = \theta[y/x, \bar{x}, \bar{y}] \sigma$ for a suitable $\sigma$. We have $\theta'' = (\theta[\sigma[y/x, \bar{x}, \bar{y}][x], \bar{y}])$, and, by Lemma A.8, we have $[\theta'[\theta''] = [\theta[y/x, \bar{x}, \bar{y}][x], \bar{y}]]_{T}$ and, by Lemma A.8, we have $[\theta][\theta'' = [\theta[y/x, \bar{x}, \bar{y}][x], \bar{y}]_{T} \leq \theta$ and $\theta \in \gamma(\text{rename}_{\Xi_{\nu \upsilon}^{\nu \upsilon}}[\gamma(\phi)]$. We conclude that $\theta[y/x, \bar{x}, \bar{y}]$ is $\gamma(\phi), i.e., \theta \in \text{rename}_{\Xi_{\nu \upsilon}^{\nu \upsilon}}[\gamma(\phi)].$
We want to prove that $\gamma(\phi) = \{ \theta \in \Theta_V \mid \text{for all } \sigma \leq \theta \text{ we have } [\theta]^{\sigma} = 1 \}$

$$= \bigcup \{ [\text{sol}_V(\theta)] \mid \theta \in \Theta_V \text{ and } [\text{sol}_V(\theta)] \subseteq \gamma(\phi) \}$$

$$= \bigcup \{ [\text{sol}_V(h)] \mid h \in H_V \text{ and } [\text{sol}_V(h)] \subseteq \gamma(\phi) \}$$

$$= \bigcup \{ [\text{sol}_V(h)] \mid h \in I \}$$

where

$$I = \left\{ h \in H_V \mid \text{sol}_V(h) \subseteq \gamma(\phi) \text{ and there does not exist } h' > h \text{ such that } \text{sol}_V(h') \subseteq \gamma(\phi) \right\}.$$ 

We want to prove that $I \subseteq H_V \setminus \overline{\gamma}$. Let $h = \exists_w c \in I$, and assume, by contradiction, that $\overline{\gamma} \subseteq \text{dom}(c) \cup \text{rng}(c)$, for some $\overline{\gamma} \in \gamma$. Let $h' = \exists_w c \in N[\overline{\gamma}]$. Note that $h' > h$. Indeed, by construction we have $\text{sol}_V(h) \subseteq \text{sol}_V(h')$.

To show that that inclusion is strict, let $\sigma \in \Theta_{W \cup N}$ be such that $\sigma[v] = a$ for every $v \in W \cup N$. We have $[c[N[\overline{\gamma}]]]_V \in \text{sol}_V(h')$, but $[c[N[\overline{\gamma}]]]_V \notin \text{sol}_V(h)$. This is because $\overline{\gamma}$ does not occur in $[c[N[\overline{\gamma}]]]_V$ which, therefore, cannot be a solution of $h = \exists_w c$, where $\overline{\gamma}$ occurs.

By definition, we have $h' = \exists_w c$. Therefore,

\[
\text{sol}_V(h') = \exists_w c \text{ (Prop. 3.5(ii))} = \exists_w c \text{ (Prop. 4.25(iii))} = \gamma(\phi).
\]

(since $\text{sol}_V(h) \subseteq \gamma(\phi)$, by using Prop. 3.3(iii))

$\gamma(\phi)$ does not occur in $\phi$ = $\gamma(\phi)$.

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

PROOF. [Proposition 4.28] For every $j \geq 0$ we have $\Theta_V \in \text{Basic}^j_{T_V}$ since $\text{Basic}^j_{T_V}$ is a Moore family of $\text{Down}_V$. Given $i \geq 0$ and $d \in \text{Basic}^i_{T_V}$, since $d$ is downward closed, by Equation (2.1) we have $\Theta_V \rightarrow d = d$, i.e. $d \in \text{Basic}^{i+1}_{T_V}$.

PROPOSITION A.9

Let $(\Delta, \Sigma, I)$ be a type system and $V \subseteq \psi_f(V)$. Letting $\gamma$ denote $\gamma_{\Sigma, \Delta, V}$, we have

(i) $\gamma(v \in t) = \psi_t$ for all $v \in V$ and $t \in \text{terms}(\Delta, \emptyset)$.

(ii) $\gamma(\forall S) = \bigcup_{\phi \in S} \sigma(\phi)$ if $\forall S \in \mathcal{O} \Delta, V$.

(iii) $\gamma(A \Rightarrow O) = \gamma(A) \rightarrow \gamma(O)$ if $A \in \text{And}_{\Delta, V}$ and $O \in \mathcal{O} \Delta, V$.

PROOF

(i)

$$\gamma(v \in t) = \{ \theta \in \Theta_V \mid \text{for all } \sigma \leq \theta \text{ we have } \sigma(x) \in [t]^{\sigma} \}$$

\([t]^{\sigma} \text{ is downward closed} \} = \{ \theta \in \Theta_V \mid \theta(x) \in [t]^{\sigma} \} = \psi_t.$$

(ii) Note that, since $\forall S \in \mathcal{O} \Delta, V$, every $\phi \in S$ has the form $x \in t$ for suitable $x \in V$ and $t \in \text{terms}(\Delta, \emptyset)$.
Therefore,
\[ \gamma(\forall \phi \in S) = \left\{ \theta \in \Theta_V \mid \text{for all } \sigma \leq \theta \text{ there exists } \phi \in S \quad \text{such that } \llbracket \phi \rrbracket_\sigma = 1 \right\} \]
(since \( \phi = \{x \in t\} = \{\theta \in \Theta_V \mid \text{there exists } \phi \in S \text{ such that } \llbracket \phi \rrbracket_\theta = 1 \})

\[ = \bigcup_{\phi \in S} \{\theta \in \Theta_V \mid \llbracket \phi \rrbracket_\theta = 1 \} \]
(since \( \phi = \{x \in t\} = \bigcup_{\phi \in S} \{\theta \in \Theta_V \mid \text{for all } \sigma \leq \theta \text{ we have } \llbracket \phi \rrbracket_\sigma = 1 \}

\[ = \bigcup_{\phi \in S} \gamma(\phi). \]

(iii)

\[ \gamma(A \Rightarrow O) = \{\theta \in \Theta_V \mid \text{for all } \sigma \leq \theta \text{ if } [A]_\sigma = 1 \text{ then } [O]_\sigma = 1 \}
\]

(Proposition 4.28) \[ = \{\theta \in \Theta_V \mid [A]_\theta = 1 \Rightarrow [O]_\theta = 1 \}
\]

\[ = \{\theta \in \Theta_V \mid \text{for all } \sigma \leq \theta \text{ we have } [A]_\sigma = 1 \Rightarrow [O]_\sigma = 1 \}
\]

\[ = \gamma(A) \Rightarrow \gamma(O). \]

PROOF. [Proposition 4.30] By Proposition A.9(i) and (iii) and by Proposition 4.13, we conclude immediately that

\[ \text{Basic}_V = \gamma(\text{Def}_V) \]

In [44] it is shown that, provided Proposition A.9 holds, the results which we state to be entailed by the structural nature of \( T \) hold if, letting \( \{ t_i \}_{i \in I}, \{ c_j \}_{j \in J} \) and \( \{ d_k \}_{k \in K} \) contained in \( \{ v_r \mid v \in V \text{ and } t \in \text{terms}(\Delta, \emptyset) \} \), with \( I, J, K \subseteq \mathbb{N} \), letting \( B = \cap \{ t_i \}_{i \in I}, C = \cup \{ c_j \}_{j \in J}, D = \cup \{ d_k \}_{k \in K} \) and letting \( \theta \in \Theta_V \) be such that \( \theta \not\in B \cup C \cup D \), we have \( \theta \not\in \{ B \Rightarrow C \} \Rightarrow D \). But this is true since, by Definition 4.16, we know that there exists \( \sigma \in \Theta_V \) such that \( \theta \sigma \) is grounding for \( V \) and \( \theta \sigma \) is type-equivalent to \( \theta \sigma \) (Definition A.4). Therefore, every \( \theta \sigma \not\in \{ B \Rightarrow C \} \Rightarrow D \), i.e. \( \theta \not\in \{ B \Rightarrow C \} \Rightarrow D \) since \( \{ B 
Rightarrow C \} \Rightarrow D \) is downward closed.

**Proofs for Section 5**

PROOF. [Proposition 5.5] Given \( \{ P_1, P_2 \} \subseteq \text{Prog}_{V}^{h} \), we have

\[ \gamma^h(P_1 \cup P_2) = \bigwedge_{c \in P_1 \cup P_2} \gamma^h(c) = \bigwedge_{c \in P_1} \gamma^h(c) \land \bigwedge_{c \in P_2} \gamma^h(c) = \gamma^h(P_1) \land \gamma^h(P_2). \]

PROOF. [Proposition 5.6]

(i) This is a direct consequence of Proposition 5.5.

(ii) Let \( P \in \text{Prog}_{V}^{h} \) and let \( P' \) be as in Definition 5.2. First we show that \( \gamma^h(P) \leq \gamma^h(P' \cap \text{Prog}_{V}^{h}) \). Indeed, consider a clause \( c \in P' \cap \text{Prog}_{V}^{h} \), which is the folding of some clauses \( c_1, \ldots, c_n \) of \( P \) in a clause \( c_{n+1} \) of \( P \). Every ground instance \( \sigma \) of \( c \) is the folding of suitable ground instances \( c_1^\sigma, \ldots, c_n^\sigma \) of \( c_1, \ldots, c_n \) in a suitable ground instance \( c_{n+1}^\sigma \) of \( c_{n+1} \). Therefore, if \( \llbracket \gamma^h(P) \rrbracket \sigma = 1 \) then \( \llbracket c_i^\sigma \rrbracket \sigma = 1 \) for \( i = 1, \ldots, n + 1 \) and \( \llbracket c^\sigma \rrbracket \sigma = 1 \). Since this is true for every \( c \in P' \cap \text{Prog}_{V}^{h} \), we have that \( \llbracket \gamma^h(P') \rrbracket \sigma = 1 \).

To show that \( \exists x \in \text{terms} \Delta \Delta. \gamma^h(P) \leq \gamma^h(P \cap \text{Prog}_{V}^{h}) \), let \( \llbracket \exists x \in \text{terms} \Delta \Delta. \gamma^h(P) \rrbracket \sigma = 1 \). Then \( \llbracket \gamma^h(P') \rrbracket \sigma = 1 \) for a suitable \( S \in \{ \text{terms} \Delta \Delta \} \). Consider a clause \( H : = B \in P \cap \text{Prog}_{V}^{h} \). Since \( H : = B \in P \) and \( x \) does not occur in \( H \), we have \( H'[t_1/V_1] \cdots [t_n/V_n] : = B'[t_1/V_1] \cdots [t_n/V_n] \llbracket \theta = 1 \) for every set of terms \( \{ t_1, \ldots, t_n \} \subseteq \text{terms} \Delta \Delta \), where \( V_1, \ldots, V_n \) are the type variables of \( H : = B \). This means that \( \llbracket \gamma^h(P' \cap \text{Prog}_{V}^{h}) \rrbracket \sigma = 1 \).
By using the above facts we conclude that
\[
\gamma^h(P) = \gamma^h\left( P' \cap \text{Prog}^h \right)
\]
which implies that
\[
\gamma^h(P) = \gamma^h\left( P' \cap \text{Prog}^h \right)
\]
Therefore, for all
\[
\gamma^h(P) = \gamma^h\left( P' \cap \text{Prog}^h \right)
\]
We proceed by induction on
\[
\gamma^h(P) = \gamma^h\left( P' \cap \text{Prog}^h \right)
\]
result is true for
\[
\gamma^h(P) = \gamma^h\left( P' \cap \text{Prog}^h \right)
\]
Therefore, we have the thesis.

Proof: [Proposition 5.10] Since \( c \) can be seen as a substitution, consider \( \sigma \leq c \). We show that \( \gamma^h\alpha^c|\gamma^h(c)\sigma = 1 \). Let \( x \in V \), \( \{x_1, \ldots, x_n\} = \text{vars}(c(x)) \) and \( \theta = \{\text{type} \rightarrow t', x_1 \rightarrow t_{x_1}', \ldots, x_n \rightarrow t_{x_n}'\} \) be a computed answer substitution for \( \text{type}(c(x)), \text{Type} \), for \( 1 \leq i \leq m \). Then
\[
[[\{x_1(t_{x_1}'), \ldots, x_n(t_{x_n}')\}]\sigma = 1
\]
\[
[[x_1 \in t_{x_1}', \mu \wedge \ldots \wedge x_n \in t_{x_n}', \mu]\sigma = 1
\]
\[
[[x_j \in t_{x_j}', \mu]\sigma = 1
\]
for all \( j = 1, \ldots, n \)
\[
[[\sigma(x_j) \in \{t_{x_j}', \mu\}]\sigma = 1
\]
for all \( j = 1, \ldots, n \)
\[
[[\sigma(x_j) \in \{\theta(x_j), \mu\}]\sigma = 1
\]
for all \( j = 1, \ldots, n \)
\[
\text{(Def. 5.7)} \quad c(x)\sigma \in \{\text{Type} \theta \mu\}
\]
\[
[[c(x)\sigma \in \{\text{Type} \theta \mu\}
\]
\[
([\sigma \leq c \} \quad c(x)\sigma \in \{\text{Type} \theta \mu\}
\]
\[
\text{Therefore, for all } i = 1, \ldots, m \text{ we have } [[\{x(t_i') \rightarrow x_i(t_{x_i}'), \ldots, x_n(t_{x_n}'), \mu\}]\sigma = 1 \text{ and }[[x_j(t_{x_j}') \rightarrow x_i(t_i')]\mu]\sigma = 1 \text{ for } j = 1, \ldots, n \text{. Thus, by Definition 5.9, } \gamma^h\alpha^c|\gamma^h(c)\sigma = 1 \text{. As } x \in V \text{ was arbitrary, we have the thesis.}
\]

Proof: [Proposition 5.11] Assume \( [[\gamma^h(P)]\theta = 1 \). Let \( M[v] = \{d \in \text{terms}(\Delta, 0) \mid \theta(v) \in \{d\}\}
\). For every clause \( H : B \in P \) and every \( \mu : T \rightarrow \text{terms}(\Delta, 0) \) we have \( [H' \mu : - B' \mu.]\theta = 1 \). This means that \( M \models H' \mu : - B' \mu \cdot \). Therefore, \( M \models P_1 \). Since \( P_1 \leq P_2 \) we have \( M \models P_2 \). This means that for every clause \( H : B \in P_2 \) and every \( \mu : T \rightarrow \text{terms}(\Delta, 0) \) we have \( M \models H' \mu : - B' \mu \cdot \). Since \( M \models P \cdot \), we have \( M \models P_2 \). This means that for every \( \mu : T \rightarrow \text{terms}(\Delta, 0) \) we have \( M \models H' \mu : - B' \mu \cdot \). We conclude that \( [[\gamma^h(P)]\theta = 1 \).

Proof: [Proposition 5.12] Consider \( \mu : T \rightarrow \text{terms}(\Delta, 0) \). Since \( v(\alpha) \) is derivable from \( P \), then \( v(\alpha)\mu \) is derivable from \( P \). We show that if \( M \models P \) and \( M \models v(\alpha)\mu \cdot \), then \( M \models v(\alpha)\mu \cdot \). Since this is done for every \( \mu \), we have the thesis. We proceed by induction on \( n \), the number of resolution steps. If \( n = 1 \) then there exists a clause \( v(\tau) \in P \) such that \( v(\tau) = d \mu \) for a suitable \( \mu' \). Since \( M \models v(\tau) \cdot \), we have \( M \models v(\tau)\mu' \cdot \). Assume that \( v(\alpha)\mu \) is derivable from \( P \) by \( n + 1 \) steps of resolution. Then there exists a clause \( v(\tau) : - B_1', \ldots, B_m \cdot \in P \) such that \( v(\tau) = d \mu \) for a suitable \( \mu' \) and \( B_1', \ldots, B_m \cdot \) is derivable from \( P \) in \( n \) steps for all \( i = 1, \ldots, m \). By inductive hypothesis, we conclude that \( P \models B_i \mu' \cdot \) for all \( i = 1, \ldots, m \). Since \( M \models P \cdot \), we have \( M \models B_i \mu' \cdot \) for all \( i = 1, \ldots, m \), and since \( M \models v(\tau)\mu' : - B_1', \ldots, B_m, \mu' \cdot \), we have \( M \models v(\tau)\mu' \cdot \). Therefore, we have the thesis.

Received 23 March 2000