Abstract. This thesis presents the application of the linear refinement technique to the development of abstract domains for static analysis of logic programs. We consider the typical cases of abstract properties: types (hence, groundness), sharing and freeness. Since types are downward closed properties, specific representations and specific algorithms for the abstract operations can be easily devised. Then we define a generic representation and generic abstract algorithms for any abstract property, and we show that the linear refinement technique makes clear when the reduced product of two analyses is very precise, in the sense that the two analyses help each other. A typical case is sharing/freeness interaction. As a consequence, we instantiate our generic construction to the case of non pair-sharing and freeness. Finally, we show that sometimes linear refinement does not lead to precise domains. This happens, for instance, if we want to approximate freeness without the help of any auxiliary property. We show how to overcome this problem through a non standard use of linear refinement.
Paesi di contadini e solfaraj, poveri analfabeti. Quattromila, soltanto a Casteltermini.

L. Pirandello,
I vecchi e i giovani, 1913

Credo che l’uomo sia maturo per altro.

E. Vittorini,
Conversazione in Sicilia, 1941

Come tutti i siciliani “buoni”, come tutti i siciliani migliori, Majorana non era portato a far gruppo, a stabilire solidarietà e a stabilirvisi.

L. Sciascia,
La scomparsa di Majorana, 1975
To my family
Acknowledgments

_Cui dono lepidum novum libellum?_

Catullus,  
_Nugae, 50 bC_

First of all, I wish to thank Prof Giorgio Levi, for trusting in me during these four years. He taught me what scientific quality and originality is. This thesis is the result of our long term collaboration.

My special thanks go to Dr Patricia Hill. She taught me precision and fairness. We have been working together for six months in Leeds, and she made that time useful and productive.

I thank Thomas Jensen for providing me with a quiet environment at Rennes during the final revision of this thesis.

I thank Maurice Bruynooghe and Michael Codish for their precise and useful refereeing. They helped me to improve the overall aspect of this thesis.

My family has always supported me during these years. My dearest thanks go to my parents and my sisters for their understanding as well as their misunderstanding.

I thank my friends with whom I shared the love for theoretical computer science while I was still an undergraduate. Among them, I want to remember Giacomo Piccinelli and Gabriele Zugliani.

I thank my PhD colleagues of the University of Pisa. Namely, I want to thank Gianluca Amato, Marco Comini, Vincenzo Gervasi, Roberta Gori, Ernesto Lastres, René Moreno, Francesca Scozzari and Paolo Volpe. Their presence has been stimulating from the scientific side, and important from the human side.

I am greatly indebted to Gianluca Amato for proofreading a preliminary draft of this thesis. His criticisms have greatly helped me to correct and improve even small details of this work.

I thank all my friends not already mentioned. Namely, I thank Alberto Gelmini, Alberto Pambira, Angelita Russo, Annamaria Spagnolo, Antonio Fernandez, Antonio Savona, Carmen Caserta, Caterina Cadoni, Christian Ponnelli, Cinzia Conficoni, Costanzo Ferraro, Erika Soriani, Guido Spoto, Jan Smaus, Karim Djemame (who
had the burden of sharing his house with me), Klaus Thiel, Laura Crosilla, Leda Giuffrida, Marilena Albanese, Mauro Arcese, Mauro Tantillo and Sarah Monaci.

My final thanks go to Francesca, for being with me.
# Contents

## Introduction

1. Static program analysis .............................................. v
2. Linear refinement ....................................................... vii
3. Compiling program analyses .......................................... vii
4. Plan of the thesis ...................................................... viii

## Preliminaries

1.1 Notation on sets and sequences .................................... 1
1.2 Relations and functions .............................................. 1
  1.2.1 Lambda notation ............................................... 2
  1.2.2 Composing relations and functions ............................ 2
  1.2.3 Direct and inverse image of a relation ....................... 3
  1.2.4 Equivalence relations and congruences ....................... 3
1.3 Complete partial orders and closure operators ......... 3
1.4 Fixpoint theory .................................................... 5
1.5 Terms and substitutions ............................................ 6
1.6 Abstract interpretation ............................................ 7
  1.6.1 Galois insertions and abstract interpretation ............. 7
  1.6.2 From Galois insertions to closure operators ............. 8
  1.6.3 Correctness, optimality and precision of abstract functions .. 9
  1.6.4 Domain refinement .......................................... 9

## Semantics

2.1 Introduction ....................................................... 11
2.2 Constraint logic programming .................................. 12
2.3 Classical semantics ............................................... 13
  2.3.1 Classical call pattern semantics ......................... 15
2.4 Existential Herbrand constraints ............................... 16
2.5 Elastic semantics .................................................. 18
  2.5.1 Elastic semantics for call patterns ..................... 21
  2.5.2 Elastic semantics over Herbrand ....................... 23
2.6 An equivalence result .......................................... 24
  2.6.1 Classical vs elastic .................................. 25
5.6.1 Information extraction ............................... 86
5.7 The complexity issue .................................. 87
5.7.1 Reduction rules .................................. 88
5.8 Conclusions .................................. 89
5.9 Proofs ........................................ 91

6 Non pair-sharing and freeness ............................ 95
6.1 Introduction ....................................... 95
  6.1.1 Related works .................................. 96
6.2 Non pair-sharing analysis .............................. 96
  6.2.1 The domain $NSh_V \Rightarrow^{*_{\text{NSh}}} NSh_V$ ............... 98
6.3 Freeness analysis .................................. 99
  6.3.1 The domain $Free_V \Rightarrow^{*_{\text{Free}}} Free_V$ ............... 99
6.4 A pragmatic look at linear refinement ................. 100
6.5 Non pair-sharing and freeness analysis ............... 101
6.6 Representation and algorithms ......................... 102
6.7 Abstraction and information extraction ............... 103
  6.7.1 Abstraction .................................... 103
  6.7.2 Information extraction ......................... 104
6.8 Reduction rules .................................. 105
6.9 Example ........................................ 110
6.10 Conclusions .................................... 110
6.11 Proofs ........................................ 112

7 Freeness in isolation .................................. 117
7.1 Introduction ...................................... 117
7.2 Freeness analysis .................................. 119
7.3 Internal freeness dependencies ........................ 119
7.4 A representation for $Fr_V$ .......................... 121
  7.4.1 The abstraction function ......................... 122
  7.4.2 Information extraction ......................... 122
7.5 The abstract operators ................................ 123
7.6 An example ...................................... 124
7.7 $Fr_V$ vs $Sharing_V \cap Free_V$ .................... 126
7.8 Conclusions .................................... 127
7.9 Proofs ........................................ 128

Conclusions ........................................ 135

Bibliography ........................................ 137

Index ........................................ 145
Introduction

One day Alice came to a fork in the road and saw a Cheshire cat in a tree.
“Which road do I take?” she asked.
“Where do you want to go?” was his response.
“I don’t know”, Alice answered.
“Then”, said the cat, “it doesn’t matter”.

Lewis Carroll,
Alice in Wonderland, 1946

1 Static program analysis

Static program analysis infers, at compile time, information about the run time behaviour of a program. This information can be useful for the programmer, who may need it in order to understand or debug or improve his programs. It can be useful to the compiler itself, which may use it in order to optimise the compiled code or make it run on special hardware like parallel machines. For a survey of the applications of program analysis to the logic programming paradigm, see [28]. Applications of program analysis to the imperative paradigm can be found in [1].

A clean and formal definition of program analysis is obtained through the use of abstract interpretation [25, 26]. Once a formal semantics of a programming language and a property of interest is given, abstract interpretation allows one to define the most precise abstract semantics which models that property. The idea is to execute the program over abstract data (domains) by using abstract semantical operators. Though abstract interpretation defines the most precise domain for a given property, this definition is not effective, meaning that this domain cannot be derived automatically through an algorithmic definition. Therefore, in practice, for any property of interest, plenty of domains have been developed, together with correct but not necessarily optimal semantical operators, given as algorithms over the domain.

Static program analysis, based or not on abstract interpretation, has been used successfully in practice. These successes justify the growing interest shown by the scientific community. Meanwhile, more and more applications of program analysis have been devised, as a consequence of the use of critical applications in hostile
environments like the web, where security and secrecy are strongly desired properties of applets.

An abstract analyser for a given property can be depicted as in Figure 1. It receives a program as its input, and outputs the result of the analysis, which we call the (abstract) denotation of the program. A controversial issue is whether a description of possible inputs of the program should be provided to the analyser or to the denotation. In the first case, the analysis is called goal-dependent or input-driven. In the second case it is called goal-independent.

Both approaches have their advantages and disadvantages. For our purposes, we prefer goal-independent analyses since they can be used for modular program analysis. In this case, indeed, it is not possible to know in advance the abstract properties of the input of the program. Moreover, making conservative hypotheses about these properties would allow a conservative analysis only, while we want to use as much information as possible for every possible instantiation of the input. Note, however, that this advantage of goal-independent analyses can be exploited only if the abstract domain is so expressive as to represent the behaviour of the program for every possible input. The ideal case is when the specialisation of the denotation w.r.t. the input leads to the same result as the input-driven analysis of the program which, instead, is in general more precise. When this happens, we say that the abstract domain used for the denotations is condensing.

Another advantage of goal-independent analyses is that, once a procedure has been analysed, its code is not required anymore, since every call to that procedure can be analysed by using its denotation, instead of running again its analysis. This aspect is very important when the source code of a procedure is not available for copyright reasons. In such a case, the producer can provide its abstract denotation together with the compiled code. This means that precise program analysis can be done even if we do not know the source code of the procedure.
2 Linear refinement

Linear refinement [39, 73] is an automatic technique for improving the precision of an abstract domain by adding new abstract elements which represent the input/output behaviour of a concrete operator w.r.t. the property of interest. These new elements are usually represented by arrows. An arrow $i \rightarrow o$ means that if the abstract property $i$ holds for the input of the operator, then the abstract property $o$ holds for its output. In the logic programming case, the operator is usually unification. We think that assignment should be used in the imperative paradigm.

The availability of arrows between abstract properties means that a linearly refined domain allows one to express the behaviour of a program for every possible input. With an abuse of notation, we can say that linear refinement allows us to refine an abstract domain into a “more condensing” one. Therefore, a linearly refined domain is a good choice for a goal-independent analysis.

3 Compiling program analyses

Since the implementation of the most precise analyser for a given property cannot be made automatic, being a non effective procedure, we accept to deal with suboptimal analysers, which are just correct w.r.t. some analysis. However, even in this case, the domains which have been developed up to now for every given property are enormously different from each other. The same can be said for their abstract operators. Moreover, few of them can be used for precise goal-independent analysis. It might seem that the definition of a precise abstract domain and operators for a given property is a skillful task, meaning that human intervention is required. We do
not agree with this belief. Though human intervention is needed in general, we claim that its role can be made much smaller than it used to be. Our opinion is based on the observation that linear refinement allows one to define abstract domains and abstract operators independently from the property of interest. This is because, though the meaning of arrows is property-dependent, their high level meaning, i.e., the propagation of information, is not. Therefore, correct abstract operators can be defined independently from the property of interest.

This way of looking at program analysis pushes to its extreme consequences the idea of abstraction as compilation [44]. Note that this idea has been applied up to now at the semantics level only. Instead, we extend this idea to the same domain and abstract operators. By using this approach, an analyser can be seen as a black box (Figure 2) receiving as input a program and the abstraction map (compiler) which formalises the property of interest. As before, it outputs the denotation of the program.

4 Plan of the thesis

In this thesis we use linear refinement for the development of abstract domains for the analysis of logic programs. Moreover, we show how linear refinement can be used to provide a property-independent analyser.

The thesis is organised as follows. Chapter 1 introduces notations and preliminaries needed in subsequent chapters. Chapter 2 defines the concrete semantics of logic programs in a bottom-up fashion, both for computed answers and call patterns. Chapter 3 defines generic domains for downward closed properties of logic programs, in the same way as the domains for groundness [22, 3, 23, 72] has been defined in the past. Moreover, it shows that good domains enjoy the same properties of groundness, and can be defined through the linear refinement technique. This chapter is used as the basis, in Chapter 4, for the development of abstract domains for type analysis of logic programs. These domains are made of pure logic programs, in the spirit of [33]. Downward closed properties and types are very special properties, since they enjoy simple and efficient implementations. Instead, this is not the case anymore for non downward closed properties of logic programs, like sharing [56, 49, 12, 54, 55, 66, 75], freeness [29, 66, 12, 15, 21, 65, 79, 55, 46] and linearity [12, 54]. Therefore, Chapter 5 provides a framework for the analysis of generic properties through the use of linear refinement. Namely, it defines a generic abstract analyser which can be used as soon as the domain designer specifies the abstraction map from the concrete domain to sets of arrows. We consider the issues of the precision and of the computational complexity of the analysis. Chapter 6 applies the framework of Chapter 5 to the analysis of non pair-sharing and freeness. The resulting domain is more precise than traditional ones, and can be used for abstract compilation [44]. The framework of Chapter 5 cannot be used for modelling freeness in isolation, without the help of any auxiliary property. This is because the
resulting abstract analyser would be very imprecise. Therefore, Chapter 7 shows how a precise domain for freeness analysis can be obtained by a clever use of the linear refinement technique. Since we do not use the framework of Chapter 5 this time, we must explicitly provide a representation and operators for this linearly refined domain.

The proofs are kept apart at the end of the corresponding chapter. Slanted boldface introduces the proof of a result contained in the chapter, while standard boldface introduces a definition or result which is local to the proofs themselves. The index at the end of the thesis relates notations and concepts to the place where they have been defined.
Chapter 1 Preliminaries

*Karma police, arrest this man, he talks in maths.*

Radiohead,
*Karma police, 1997*

We introduce here most of the notations and results which will be used throughout this thesis. These results are not new. Our contribution is to present them in a uniform and consistent way. Some more specific notions will be introduced in the chapters where they are needed. The index at the end of the thesis can be used to find where some notation has been defined.

1.1 Notation on sets and sequences

A set is an unordered collection of elements. Given two sets $S_1$ and $S_2$, their Cartesian product is the set of pairs $S_1 \times S_2 = \{(s_1, s_2) \mid s_1 \in S_1 \text{ and } s_2 \in S_2\}$. The powerset of a set $S$ is $\wp(S) = \{S' \mid S' \subseteq S\}$. The cardinality of a set $S$ is denoted by $\text{card}(S)$. We denote by $\wp_f(S)$ the set of all finite subsets of $S$.

A sequence is an ordered collection of elements. If $\bar{x}$ is a sequence of elements of $S$, we write $\bar{x} \subseteq S$.

1.2 Relations and functions

A binary relation between $S$ and $S'$ ($R : S \times S'$) is an element of $\wp(S \times S')$. We write $x R y$ for $(x, y) \in R$. A partial function from $S$ to $S'$ is a relation $f : S \times S'$ such that for any $x \in S$ and $\{y, y'\} \subseteq S'$ we have that $(x, y) \in f$ and $(x, y') \in f$ entails $y = y'$. By $f : S \rightarrow S'$ we denote a partial function of the set $S$ (the domain) into the set $S'$ (the range). We use the notation $f(x) = y$ when there exists $y$ such that $(x, y) \in f$. In such a case, we say that $f(x)$ is defined. Otherwise, we say that $f(x)$ is undefined.

A (total) function $f$ from $S$ to $S'$ is a partial function from $S$ to $S'$ such that, for all $x \in S$, there is some $y \in S'$ such that $f(x) = y$. Although total functions are a
special kind of partial function, it is traditional to understand something described as simply a function to be a total function. So we will always say explicitly when a function is partial. To indicate that a function \( f \) from \( S \) to \( S' \) is total, we write \( f : S \to S' \).

We denote by \( f = g \) the extensional equality, i.e., for each \( x \in S \), \( f(x) = g(x) \). Furthermore, \( g = f[v/x] \) denotes the function \( g \) which differs from \( f \) only for the assignment of \( v \) to \( x \). Namely, \( g(x) = v \) and, for each \( y \in S \), \( y \neq x \), \( g(y) = f(y) \).

### 1.2.1 Lambda notation

It is sometimes useful to use the lambda notation to describe functions. It provides a way of referring to functions without having to name them. Suppose \( f : S \to S' \) is a function which, for any element \( x \in S \), gives a value \( f(x) \) which is exactly described by the expression \( E \), possibly involving \( x \). Then we can write \( \lambda x \in S . E \) for the function \( f \). Thus, \( (x \in S . E) = \{ (x, E[x]) \mid x \in S \} \) and so \( \lambda x \in S . E \) is just an abbreviation for the set of input-output values determined by the expression \( E[x] \). We use the lambda notation also to denote partial functions by allowing expressions in lambda-terms that are not always defined. Hence, a lambda expression \( \lambda x \in S . E \) denotes a partial function \( S \to S' \) which, on input \( x \in S \), assumes the value \( E[x] \in S' \), if the expression \( E[x] \) is defined, and otherwise it is undefined.

### 1.2.2 Composing relations and functions

We compose relations, and so partial and total functions, \( R : S \times S' \) and \( Q : S' \times S'' \) by defining their composition \( (a \text{ relation between } S \text{ and } S'' \) \) as \( Q \circ R = \{ (x, z) \in S \times S'' \mid \text{there exists } y \in S' \text{ such that } (x, y) \in R \text{ and } (y, z) \in Q \} \). We denote by \( R^n \) the relation

\[
R \circ \cdots \circ R,
\]

i.e., \( R^1 = R \) and \( R^{n+1} = R \circ R^n \). Each set \( S \) is associated with an identity function \( \text{Id}_S = \{ (x, x) \mid x \in S \} \), which is the neutral element of \( \circ \). We define \( R^0 = \text{Id}_S \).

The function composition of \( g : S \to S' \) and \( f : S' \to S'' \) is the partial function \( f \circ g : S \to S'' \), where \( (f \circ g)(x) = f(g(x)) \), if \( g(x) \) (first) and \( f(g(x)) \) (then) are defined, and is undefined otherwise. When it is clear from the context, \( \circ \) will be omitted. A function \( f : S \to S' \) is one-to-one if and only if for each \( \{ x, y \} \subseteq S \) if \( f(x) = f(y) \) then \( x = y \). \( f \) is onto if and only if for each \( x' \in S' \) there exists \( x \in S \) such that \( f(x) = x' \). \( f \) is bijective if it has an inverse \( g : S' \to S \), i.e., if and only if there exists a function \( g \) such that \( g \circ f = \text{Id}_S \) and \( f \circ g = \text{Id}_{S'} \). Then the sets \( S \) and \( S' \) are said to be in one-to-one correspondence. Any set in one-to-one correspondence with a subset of natural numbers is said to be countable. Note that a function \( f \) is bijective if and only if it is one-to-one and onto.
1.2.3 Direct and inverse image of a relation

We extend relations, and thus partial and total functions, $R: S \times S'$ to functions on subsets by taking $R(X) = \{ y \in S' \mid \text{there exists } x \in X. (x, y) \in R \}$ for $X \subseteq S$. The set $R(X)$ is called the direct image of $X$ under $R$. Therefore, if $f : S \to S'$ is a partial function and $X \subseteq S$, we denote by $f(X)$ the image of $X$ under $f$, i.e., the set $f(X) = \{ f(x) \mid x \in X \}$.

1.2.4 Equivalence relations and congruences

An equivalence relation $\approx$ on a set $S$ is a binary relation on $S (\approx : S \times S)$ such that, for each $\{x, y, z\} \subseteq S$, we have

\[
\begin{align*}
  x &\approx x \quad &\text{(reflexivity)} \\
  x \approx y &\text{ entails } y \approx x \quad &\text{(symmetry)} \\
  x \approx y &\text{ and } y \approx z \text{ entails } x \approx z \quad &\text{(transitivity)}.
\end{align*}
\]

The equivalence class of an element $x \in S$ w.r.t. $\approx$ is the subset $[x]_\approx = \{ y \mid x \approx y \}$. When clear from the context, we abbreviate $[x]_\approx$ by $[x]$ and often abuse notation by letting the elements of a set denote their correspondent equivalence classes. The quotient set $S /_\approx$ of $S$ modulo $\approx$ is the set of equivalence classes of elements in $S$ w.r.t. $\approx$.

An equivalence relation $\approx$ on $S$ is a congruence w.r.t. a partial function $f : S^n \to S$ if and only if, given $\{a_i, b_i\} \subseteq S$ with $i = 1, \ldots, n$, if $f(a_1, \ldots, a_n)$ is defined then also $f(b_1, \ldots, b_n)$ is defined and $f(a_1, \ldots, a_n) \approx f(b_1, \ldots, b_n)$. Then, we can define the partial function $f_\approx : (S /_\approx)^n \to S /_\approx$ as

\[
f_\approx([a_1]_\approx, \ldots, [a_n]_\approx) = [f(a_1, \ldots, a_n)]_\approx
\]

since, given $[a_1]_\approx, \ldots, [a_n]_\approx$, the class $[f(a_1, \ldots, a_n)]_\approx$ is uniquely determined independently from the choice of the representatives $a_1, \ldots, a_n$.

1.3 Complete partial orders and closure operators

A binary relation $\leq$ on $S (\leq : S \times S)$ is a partial order if, for each $\{x, y\} \subseteq S$,

\[
\begin{align*}
  x &\leq x \quad &\text{(reflexivity)} \\
  x \leq y &\text{ and } y \leq x \text{ entails } x = y \quad &\text{(antisymmetry)} \\
  x \leq y &\text{ and } y \leq z \text{ entails } x \leq z \quad &\text{(transitivity)}.
\end{align*}
\]

A partially ordered set (poset) $(S, \leq)$ is a set $S$ equipped with a partial order $\leq$. A set $S$ is totally ordered if it is partially ordered and, for each $\{x, y\} \subseteq S$, we have $x \leq y$ or $y \leq x$. A chain is a (possibly empty) totally ordered subset of $S$. 
A preorder is a binary relation which is reflexive and transitive. A preorder \(\leq\) on a set \(S\) induces on \(S\) an equivalence relation \(\approx\) defined as follows: for each \(\{x, y\} \subseteq S\), \(x \approx y\) if and only if \(x \leq y\) and \(y \leq x\). Moreover, \(\leq\) induces on \(S/\approx\) the partial order \(\leq_{\approx}\) such that, for each \(\{[x]_{\approx}, [y]_{\approx}\} \subseteq S/\approx\), we have \([x]_{\approx} \leq_{\approx} [y]_{\approx}\) if and only if \(x \leq y\).

If \((S, \leq)\) is a preorder and \(S' \subseteq S\), \(S'\) is downward closed if and only if from \(s_1 \in S'\) and \(s_2 \leq s_1\) it follows that \(s_2 \in S'\). We denote the set of downward closed subsets of \((S, \leq)\) as \(\downarrow(S)\).

A binary relation \(<\) on \(S\) is strict if and only if it is anti-reflexive (i.e., \(x < x\) does not hold for every \(x \in S\)) and transitive.

Given a poset \((S, \leq)\) and \(X \subseteq S\), \(y \in S\) is an upper bound for \(X\) if and only if for each \(x \in X\) we have \(x \leq y\). Moreover, \(y \in S\) is the least upper bound (called also join) of \(X\), if \(y\) is an upper bound of \(X\) and for every upper bound \(y'\) of \(X\), \(y \leq y'\). A least upper bound of \(X\) is often denoted by \(\text{lub}_S X\) or by \(\bigcup_S X\). We also write \(\bigcup_S \{d_1, \ldots, d_n\}\) as \(d_1 \sqcup_S \cdots \sqcup_S d_n\). Dually, an element \(y \in S\) is a lower bound for \(X\) if and only if for each \(x \in X\) we have \(x \geq y\). Moreover, \(y \in S\) is the greatest lower bound (called also meet) of \(X\), if \(y\) is a lower bound of \(X\) and for every lower bound \(y'\) of \(X\) we have \(y' \leq y\). A greatest lower bound of \(X\) is often denoted by \(\text{glb}_S X\) or by \(\bigcap_S X\). We also write \(\bigcap_S \{d_1, \ldots, d_n\}\) as \(d_1 \sqcap_S \cdots \sqcap_S d_n\). When it is clear from the context, the subscript \(S\) will be omitted. It is easy to check that if \(\text{lub}\) and \(\text{glb}\) exist, then they are unique.

A direct set is a poset in which any subset of two elements (and hence any finite subset) has an upper bound in the set. A complete partial order (CPO) \(S\) is a poset such that every chain \(D\) has a least upper bound (i.e., \(\bigcup_S D\) exists).

A complete lattice is a poset \((S, \leq)\) such that for every subset \(X\) of \(S\) there exists \(\bigcup X\) and \(\bigcap X\). We let \(\top\) denote the top element \(\bigcup S = \bigcap \emptyset\) and \(\bot\) denote the bottom element \(\bigcap S = \bigcup \emptyset\) of \(S\).

Let \((L, \leq)\) and \((M, \subseteq)\) be two posets. A function \(f : L \rightarrow M\) is monotonic if and only if for every \(\{x, y\} \subseteq L\) such that \(x \leq y\) we have \(f(x) \subseteq f(y)\). Moreover, \(f\) is continuous if and only if for each non empty chain \(D \subseteq L\) we have \(f(\bigcup_L D) = \bigcup_M f(D)\). Every continuous function is also monotonic. A function \(f : S \rightarrow S'\) is additive if and only if the previous continuity condition is satisfied for each non empty set. Hence, every additive function is also continuous. Co-continuity and co-additivity are defined dually.

It can be proved that composition of monotonic, continuous or additive functions is monotonic, continuous or additive, respectively.

The mathematical way of expressing that structures are “essentially the same” is through the concept of isomorphism. A continuous function \(f : D \rightarrow E\) between two CPOs \(D\) and \(E\) is said to be an isomorphism if there is a continuous function \(g : E \rightarrow D\) such that \(g \circ f = 1_D\) and \(f \circ g = 1_E\). Thus \(f\) and \(g\) are mutual inverses. It follows from the definition that isomorphic CPOs are essentially the same but for a renaming of elements. A function \(f : D \rightarrow E\) is an isomorphism if and only if \(f\)
is bijective and continuous.

Let $L$ be a CPO. A function $\rho : L \to L$ is an upper closure operator if:

\begin{align*}
x \leq y & \text{ entails } \rho(x) \leq \rho(y) & \text{(monotonicity)} \\
\rho(\rho(x)) & = \rho(x) & \text{(idempotency)} \\
x & \leq \rho(x) & \text{(extensivity)}.
\end{align*}

### 1.4 Fixpoint theory

Given a poset $(S, \leq)$ and a function $f : S \to S$, a fixpoint of $f$ is an element $x \in S$ such that $f(x) = x$. A pre-fixpoint of $f$ is an element $x \in S$ such that $f(x) \leq x$ and dually a post-fixpoint of $f$ is an element $x \in S$ such that $x \leq f(x)$. Moreover, we say that $x \in S$ is the least fixpoint of $f$ (denoted by $\text{lfp } f$ or $\mu f(t)$) if and only if $x$ is a fixpoint of $f$ and for all fixpoints $y$ of $f$ we have $x \leq y$. Dually, we define the greatest fixpoint (denoted by $\text{gfp } f$).

A fundamental theorem by Knaster-Tarski states that the set $\text{fp}(f)$ of fixpoints of a monotonic function $f$ is a complete lattice.

**Theorem 1.1 (Fixpoint theorem \cite{80}).** A monotonic function $f$ on a complete lattice $(L, \leq)$ has a least and a greatest fixpoint. Moreover, we have

\begin{align*}
\text{lfp}(f) &= \bigsqcap \{x \mid f(x) \leq x\} = \bigsqcap \{x \mid x = f(x)\} \\
\text{gfp}(f) &= \bigsqcup \{x \mid x \leq f(x)\} = \bigsqcup \{x \mid x = f(x)\}.
\end{align*}

The Knaster-Tarski Theorem is important because it applies to any monotone function on a complete lattice. However, most of the time we will be concerned with least fixpoints of continuous functions. Therefore, it is useful to state some more notations and results on fixpoints of functions defined on (complete) lattices.

Given a CPO $S$, a function $T : S \to S$ and $x \in S$, we define $T^0(x) = x$ and $T^{n+1}(x) = T(T^n(x))$ for $n \geq 1$. The result below is usually attributed to Kleene and gives an explicit construction of the least fixpoint of a continuous function $f$ on a CPO $D$.

**Theorem 1.2 (Fixpoint Theorem).** Let $f : D \to D$ be a continuous function on a CPO $D$ and $d \in D$ be a post-fixpoint of $f$. Then $\bigsqcup_{n \geq 0} f^\uparrow n(d)$ is the least fixpoint of $f$ greater than $d$. In particular, $\bigsqcup_{n \geq 0} f^\uparrow n(\perp D)$ is the least pre-fixpoint and least fixpoint of $f$. 
1.5 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

$$\text{terms}^0(\Sigma, V) = V$$

$$\text{terms}^{k+1}(\Sigma, V) = \text{terms}^k(\Sigma, V) \cup \left\{ f(t_1, \ldots, t_n) \mid f^i \in \Sigma \text{ and } \{t_1, \ldots, t_n\} \subseteq \text{terms}^k(\Sigma, V) \right\}$$

$$\text{terms}(\Sigma, V) = \bigcup_{d \geq 0} \text{terms}^d(\Sigma, V).$$

We assume that $\Sigma$ contains at least a 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 $V \setminus \{x\}$.

A substitution $\theta$ is a map from variables into terms. We define the sets $\text{dom}(\theta) = \{x \mid \theta(x) \neq x\}$ and $\text{rng}(\theta) = \cup_{x \in \text{dom}(\theta)} \text{vars}(\theta(x))$. We require $\text{dom}(\theta)$ to be finite. This allows us to represent a substitution $\theta$ in extensional way as $\theta = \{v_1 \mapsto t_1, \ldots, v_n \mapsto t_n\}$, meaning that $\theta(v_i) = t_i$ for all $i = 1, \ldots, n$ and $\theta(v) = v$ for every $v \in V \setminus \{v_i \mid i = 1, \ldots, n\}$.

We define $\Theta_{V,W}^Z$, 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 $\text{dom}(\theta) \cap \text{rng}(\theta) \subseteq Z$. If $Z = \emptyset$ we omit the superscript. The elements of $\Theta_{V,W}$ are called idempotent substitutions. We write $\Theta_V$ for $\Theta_{V,V}$ and $\Theta_{V}^Z$ for $\Theta_{V,V}^Z$. A substitution $\theta$ is called grounding for a set of variables $G$ if $\theta(x)$ is ground for every $x \in G$. 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 $\theta \in \Theta_{V,W}^Z$, $t\theta \in \text{terms}(\Sigma, W)$ is the term obtained with parallel substitution of every variable $x$ in $t$ with $\theta(x)$. We often write $t[t'/x]$ for $t\{x \mapsto t'\}$ for any $t' \in \text{terms}(\Sigma, V)$. Given a substitution $\sigma$ and $\{x, n\} \subseteq V$, we define the substitution $\sigma[n/x]$ as $\sigma[n/x](x) = x$, $\sigma[n/x](n) = \sigma(x)[n/x]$ and $\sigma[n/x](y) = \sigma(y)[n/x]$ if $y \neq x$ and $y \neq n$. Composition of substitutions $\theta \in \Theta_{V,W}$ and $\sigma \in \Theta_{W,Z}$ is defined as $(\theta \sigma)(x) = \theta(x)\sigma$ for every $x \in V \cup W$. We recall that composition of substitutions is associative, the empty substitution $\epsilon$ is the neutral element and, for each term $t$, $t(\theta \sigma) = (t\theta)\sigma$. A renaming is a substitution $\rho$ for which there exists $\rho^{-1}$, such that $\rho\rho^{-1} = \rho^{-1}\rho = \epsilon$.

For every set of variables $V$, a preorder is defined on $\Theta_V$ as $\theta' \leq_V \theta$ if there exists a substitution $\sigma \in \Theta_V^V$ such that $\theta' = \theta\sigma$. When $V$ is clear from the context, we write $\leq$ instead of $\leq_V$. A preorder, called subsumption, is defined on $\text{terms}(\Sigma, V)$ as $t_1 \leq t_2$ ($t_1$ is an instance of $t_2$) if $t_1 = t_2\theta$ for a suitable $\theta \in \Theta_V^V$. As usual, the subscript is omitted in $\leq_V$ when it is clear from the context. By $\equiv$ we denote the associated equivalence relation (variance). Namely, two terms $t$ and $t'$ are variants if and only if $t$ is an instance of $t'$ and vice versa. This definition is equivalent to saying that $t$ and $t'$ are variants if and only if there exists a renaming $\rho$ such that
Abstract interpretation

1.6 Abstract interpretation

1.6.1 Galois insertions and abstract interpretation

Abstract interpretation is a theory developed to reason about the abstraction relation between two different semantics (the concrete and the abstract semantics). The idea of approximating program properties by evaluating a program on a simpler domain of descriptions of concrete program states goes back to the early 70's. The inspiration was that of approximating properties from the exact (concrete) semantics into an approximate (abstract) semantics, that explicitly exhibits a structure (e.g. ordering) which is somehow present in the richer concrete structure associated to program execution.

The guiding idea is to relate the concrete and the abstract interpretation of the calculus by a pair of functions, abstraction $\alpha$ and concretisation $\gamma$, which form a Galois connection. Galois connections are used to formalise this relation between abstract and concrete meaning of a computation.

Let $(C, \leq)$ (concrete domain) be the domain of the concrete semantics, while $(A, \preceq)$ (abstract domain) be the domain of the abstract semantics. The partial order relations reflect an approximation relation. Since in approximation theory a partial order specifies the precision degree of any element in a poset, it is obvious to assume that if $\alpha$ maps every concrete element in $(C, \leq)$ into an abstract element in $(A, \preceq)$ then the following holds: if $\alpha(x) \leq y$, then $y$ is also a correct, although less precise, approximation of $x$. The same argument holds if $x \leq \gamma(y)$. Then $y$ is also a correct approximation of $x$, although $x$ provides more accurate information than $\gamma(y)$. This is formally defined below.

**Definition 1.3 (Galois insertion).** Let $(C, \leq)$ and $(A, \preceq)$ be two posets (the concrete and the abstract domain). A Galois connection $(\alpha, \gamma) : (C, \leq) \Rightarrow (A, \preceq)$ is a pair of maps $\alpha : C \to A$ and $\gamma : A \to C$ such that

1. $\alpha$ and $\gamma$ are monotonic,

2. For each $x \in C$ we have $x \leq (\gamma \circ \alpha)(x)$, and

3. 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 $\alpha \circ \gamma = Id_A$.

Property 2 is called extensivity of $\gamma \alpha$. The map $\alpha$ ($\gamma$) is called the abstraction (concretisation) function in the context of abstract interpretation.
In view of the compositional design of abstract interpretations we have that the composition of Galois insertions is a Galois insertion.

The following basic properties are satisfied by any Galois connection.

1. \( \gamma \) is one-to-one if and only if \( \alpha \) is onto if and only if \( \alpha \circ \gamma = \text{Id}_A \).

2. \( \alpha \) is additive and \( \gamma \) is co-additive.

3. The abstraction map uniquely determines the concretisation map and vice versa. Namely,

\[
\gamma = \lambda y. \bigsqcup_C \{ x \in C \mid \alpha(x) \leq y \}, \quad \alpha = \lambda x. \bigsqcap_A \{ y \in A \mid x \leq \gamma(y) \}.
\]

Conversely, if \( C \) and \( A \) are complete lattices and \( \alpha : C \to A \) is additive or \( \gamma : A \to C \) is co-additive, then \( \langle \alpha, \gamma \rangle \) is a Galois connection from \( C \) to \( A \).

When, in a Galois connection \( \langle \alpha, \gamma \rangle \), \( \gamma \) is not one-to-one, several distinct elements of the abstract domain \( (A, \preceq) \) have the same meaning (by \( \gamma \)). Therefore, the abstract domain contains redundancy [25]. Therefore, a Galois insertion can always be forced by collapsing abstract elements denoting the same concrete element into a single element, and the result is an abstract domain containing no redundant elements. This process is known as reduction of the abstract domain. It ensures that any abstract element is the image of some concrete element or, equivalently, that the abstraction function is onto. A reduction of a Galois connection \( \langle \alpha, \gamma \rangle : (C, \preceq) \vdash (A, \preceq) \) is \( \langle \alpha, \gamma \rangle : (C, \preceq) \vdash (\alpha(C), \preceq) \). If \( \langle \alpha, \gamma \rangle : (C, \preceq) \vdash (\alpha(C), \preceq) \) is a Galois insertion then \( \alpha(C) \) is isomorphic to \( \gamma(\alpha(C)) \) and \( \langle \gamma \alpha, \text{id} \rangle : (C, \preceq) \vdash (\gamma(\alpha(C)), \preceq) \) is also a Galois insertion. This allows us to consider Galois insertions of type \( \langle \rho, \text{id} \rangle : (C, \preceq) \vdash (\rho(C), \preceq) \) only, with \( \rho : C \to C \). The abstraction function and the abstract domain uniquely determine each other. Hence, to specify a Galois insertion \( \langle \rho, \text{id} \rangle : (C, \preceq) \vdash (A, \preceq) \), it suffices to give either the abstraction map \( \rho \) (and the abstract domain will be \( \rho(C) \)) or an abstract domain \( A \subseteq C \) (and the abstraction function will be the unique function \( \rho : C \to C \) such that \( \langle \rho, \text{id} \rangle : (C, \preceq) \vdash (\rho(C), \preceq) \) is a Galois insertion and \( \rho(C) = A \)).

### 1.6.2 From Galois insertions to closure operators

It has been observed in [25] that an often more practical way of reasoning about abstract domains in abstract interpretations is by closure operators. This allows us to be independent from specific representations of the objects of the abstract domain.

The equivalence between Galois insertions and closure operators is well-known [8]. In view of this equivalence, abstract domains can be studied independently from the representation chosen for their abstract objects. Assume \( L \) be a complete lattice playing the role of the concrete domain. The result below allows us to move from one representation to another.
1.6. Abstract interpretation

**Theorem 1.4 ([25]).** Let \( A \subseteq L \) be a complete lattice and \( \rho : L \to L \). We have that \( (\rho, \text{id}) : (L, \leq_L) \Rightarrow (A, \leq_A) \) is a Galois insertion if and only if \( \rho \) is an upper closure operator such that \( \rho(L) = A \).

Any closure operator \( \rho : L \to L \) is identified by the set of its fixpoints, which is its image \( \rho(L) \). Recall that \( \rho(L) \) is a Moore family of \( L \), i.e., a complete meet-sublattice of \( L \) (i.e., \( \top_L \in \rho(L) \)) and for any non-empty \( Y \subseteq \rho(L) \) we have \( \cap_L Y \in \rho(L) \). Moreover, any Moore family of \( L \) is the image of some upper closure operator on \( L \). Note that \( \rho(L) \) is not, in general, a complete sublattice of \( L \), since the join \( \sqcup_{\rho} \) in \( \rho(L) \), which is \( \rho \circ \sqcup_L \), might be different from \( \sqcup_L \). For any \( X \subseteq L \), we denote by \( \bigwedge(X) = \{ \cap_L I \mid I \subseteq X \} \) the Moore closure of \( X \), i.e., the least Moore family of \( L \) containing \( X \).

Upper closure operators and Moore families form complete lattices.

### 1.6.3 Correctness, optimality and precision of abstract functions

Let \( (\alpha, \gamma) \) be a Galois connection between \((C, \leq)\) and \((A, \leq)_\). Let \( f : C^n \to C \) be a concrete operator and assume that \( f : A^n \to A \). Then \( f \) is correct with respect to \( f \) if and only if for all \( y_1, \ldots, y_n \in A \) we have \( \alpha(f(\gamma(y_1), \ldots, \gamma(y_n))) \leq \gamma(f(y_1, \ldots, y_n)) \).

For each operator \( f \), there exists an optimal (most precise) correct abstract operator \( f \) defined as \( \bar{f}(y_1, \ldots, y_n) = \alpha(f(\gamma(y_1), \ldots, \gamma(y_n))) \), where \( \alpha \) is extended to sets \( S \subseteq C \) by defining \( \alpha(S) = \bigcap_{s \in S} \alpha(s) \). The operator \( \bar{f} \) is precise w.r.t. \( f \) if and only if \( \bar{f}(\alpha(x_1), \ldots, \alpha(x_n)) = \alpha(f(x_1, \ldots, x_n)) \) for every \( \{x_1, \ldots, x_n\} \subseteq C \). The composition of correct operators is correct. The composition of optimal operators is not necessarily optimal. The composition of precise operators is precise.

If \( (\alpha, \gamma) \) is a Galois connection between the complete lattices \((C, \leq)\) and \((A, \leq)_\), \( f : C \to C \) is a monotonic operator and \( \bar{f} : A \to A \) is monotonic and correct w.r.t. \( f \), we have \( \alpha(\text{lfp}(f)) \leq \text{lfp}(\bar{f}) \). If \( \bar{f} \) is precise, we have \( \alpha(\text{lfp}(f)) = \text{lfp}(\bar{f}) \).

### 1.6.4 Domain refinement

A systematic approach to the development of abstract domains is based on the use of domain refinement operators. Given an abstract domain \( A \), a domain refinement operator \( \delta \) yields an abstract domain \( \delta(A) \) which is more precise than \( A \), i.e., contains more points than \( A \). Classical domain refinement operators are reduced product and disjunctive completion [32]. The reduced product \( A \sqcap B \) of two domains \( A \subseteq C \) and \( B \subseteq C \) is isomorphic to the Cartesian product of \( A \) and \( B \), modulo the equivalence relation \( \langle a_1, b_1 \rangle \equiv \langle a_2, b_2 \rangle \) if and only if \( a_1 \sqcap b_1 = a_2 \sqcap b_2 \). This means that pairs having the same meaning are identified. It corresponds to the greatest lower bound operation over the lattice of Moore families.

Linear refinement was proposed in [39] as a powerful domain refinement operator. It allows one to include in a domain the information relative to the propagation of the
abstract property of interest before and after the application of a concrete operator \( \boxtimes \).

A **quantale** is a pair \( \langle C, \boxtimes \rangle \) such that

1. \( C \) is a complete lattice;
2. \( \boxtimes : C \times C \to C \) is associative;
3. for any \( a \in C \) and \( \{b_i\}_{i \in I} \subseteq C \) with \( I \subseteq \mathbb{N} \) we have
   \[
   a \boxtimes (\bigcup_{i \in I} b_i) = \bigcup_{i \in I} \{a \boxtimes b_i\}, \\
   (\bigcup_{i \in I} b_i) \boxtimes a = \bigcup_{i \in I} \{b_i \boxtimes a\}.
   \]

Let \( \langle C, \boxtimes \rangle \) be a quantale. The functions \( \to^\boxtimes \) and \( \leftarrow^\boxtimes \) are defined as

\[
\begin{align*}
\to^\boxtimes a b &= \bigsqcup_c \{c \in C \mid \text{if } a \boxtimes c \text{ is defined then } a \boxtimes c \leq_C b\}, \\
\leftarrow^\boxtimes a b &= \bigsqcup_c \{c \in C \mid \text{if } c \boxtimes a \text{ is defined then } c \boxtimes a \leq_C b\},
\end{align*}
\]

for any \( a, b \in C \). Note that if \( \boxtimes \) is commutative then \( a \leftarrow^\boxtimes b = a \to^\boxtimes b \) for every \( a, b \in C \).

Given a quantale \( \langle C, \boxtimes \rangle \), for any \( a \in C \) and \( \{b_i\}_{i \in I} \subseteq C \) with \( I \subseteq \mathbb{N} \), we have

\[
\to^\boxtimes (\bigcap_{i \in I} b_i) = \bigcap_{i \in I} (a \to^\boxtimes b_i), \\
\leftarrow^\boxtimes a = \bigcap_{i \in I} (b_i \leftarrow^\boxtimes a).
\]

If \( L_1 \subseteq C \) and \( L_2 \subseteq C \), we define

\[
L_1 \to^\boxtimes L_2 = \bigsqcup \{a \to^\boxtimes b, b \leftarrow^\boxtimes a \mid a \in L_1 \text{ and } b \in L_2\}.
\]

The linear refined domain \( L \to^\boxtimes L \) is defined as

\[
L \to^\boxtimes L = L \cap (L \to^\boxtimes L),
\]

which can be simplified into

\[
L \to^\boxtimes L = L \to^\boxtimes L
\]

if \( L \subseteq L \to^\boxtimes L \), i.e., if the elements of \( L \) can be obtained as greatest lower bounds of arrows.

Assume \( L = \bigwedge M \). Given \( \{b_i\}_{i \in I} \subseteq M \) with \( I \subseteq \mathbb{N} \) and \( l \in L \), by using Equation (1.2) we have \( l \to^\boxtimes (\bigcap_L \{b_i \mid i \in I\}) = \bigcap_L \{l \to^\boxtimes b_i \mid i \in I\} \) and \( (\bigcap_L \{b_i \mid i \in I\}) \leftarrow^\boxtimes l = \bigcap_L \{b_i \leftarrow^\boxtimes l \mid i \in I\} \) (see [73]). We conclude that

\[
L \to^\boxtimes L = L \cap (L \to^\boxtimes M),
\]

which can be simplified into

\[
L \to^\boxtimes L = L \to^\boxtimes M
\]

if \( L \subseteq L \to^\boxtimes M \) (or, equivalently, if \( L \subseteq L \to^\boxtimes L \), i.e., if the elements of \( L \) can be obtained as greatest lower bounds of arrows.
Chapter 2 Semantics

Even things that are true can be proved.
O. Wilde, The picture of Dorian Gray, 1890

In this chapter we define a generic denotational semantics modelling computed answers and a generic denotational semantics modelling call patterns of constraint logic programs and two practically more interesting versions which we call elastic semantics. We instantiate both semantics over the set of existential Herbrand constraints, thus providing a constraint logic programming version of classical logic programming. We show that the first denotational semantics and the elastic semantics over existential Herbrand constraints are equivalent. Finally, we lift these semantics over existential Herbrand constraints to their powerset versions, thus providing a collecting semantics over a collecting domain which will be used as our concrete semantics and domain in the following chapters.

2.1 Introduction

A semantics is a map which provides a meaning to every program of a given programming language. In this chapter we define a generic semantics for constraint logic programs [50] using the technique of cylindrical algebras [42, 38]. We choose a denotational semantics in order to obtain a goal-independent semantics. A goal-independent semantics provides a denotation for a program and then allows one to compute the behaviour of a query executed in the program as the evaluation of the query in the semantics of the program. This feature is very important for global program analysis, i.e., when we need information about the program independently from its input. Indeed, a goal-independent semantics provides information which is correct for every possible query.

In Section 2.2 we give the basic definitions of constraint logic programming, i.e., logic programming over a generic constraint system. In Section 2.3 we define the classical denotational semantics for computed answers of constraint logic programs as defined in [9], and its version for call patterns. These two semantics deal
with data structures (constraints) which admit a conjunction and a cylindrification operation, together with so-called diagonal elements. They show that two operations are enough for the definition of a denotational semantics for constraint logic programs. In Section 2.4 we instantiate these semantics to the set of existential Herbrand constraints, in order to see logic programming as an instance of constraint logic programming. The resulting semantics are practically cumbersome, since their computation uses a very large number of program variables. Instead, a more efficient version of the same semantics is defined in Section 2.5 using the exact set of variables which is needed at any given program point. These new versions will be called elastic semantics. Their drawback is that more operations are needed on the set of constraints. Namely, we need conjunction, restriction, renaming and extension. However, we show that an elastic semantics is definitively better than its classical version, for many practical reasons. In Section 2.6 we show that the elastic semantics over existential Herbrand constraints are equivalent to the classical semantics over the same set of constraints. Section 2.7 defines the collecting versions of the classical and elastic semantics over existential Herbrand constraints. These semantics use the powerset extension of the data structure of existential Herbrand constraints and its operations. This powerset extension will be used in the following chapters as our concrete domain, i.e., as the starting point of every abstract interpretation for program analysis. In Section 2.8 we show how an abstract interpretation of the constraints induces an abstract interpretation of a semantics. In Section 2.9 we discuss various extensions of the semantics of this chapter.

Most of the ideas contained in this chapter are derived from similar results contained in [20, 19, 34].

### 2.2 Constraint logic programming

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 two operations and distinguished elements called diagonal elements.

**Definition 2.1.** A **constraint system** over a set of variables $\mathcal{V}$ is a family of sets $D = \{D_V\}_{V \in \wp(\mathcal{V})}$ together with two operations: for $V \in \wp(\mathcal{V})$ we have a (partial) infix conjunction operation $\Diamond_V : D_V \times D_V \rightharpoonup D_V$ and a (total) cylindrification operation $\exists_V : V \times D_V \rightarrow D_V$. We write $\exists^V_C$ for $\exists_V(x, c)$. Moreover, for every two sequences $\bar{x}$ and $\bar{y}$ in $V$ of the same length, without repetitions and whose intersection is empty, we assume there is an element $\delta^V_{\bar{x}, \bar{y}} \in D_V$. In the following, when speaking of diagonal elements, we assume that the conditions on $\bar{x}$ and $\bar{y}$ hold.

Note that the definition above is very abstract since it does not require anything about the behaviour of conjunction, cylindrification and diagonal elements except their signatures. In Section 2.4 we provide an instantiation of this definition with
existential Herbrand constraints, while in the following chapters we will provide instantiations of this definition with abstract constraint systems for program analysis.

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

**Definition 2.2.** Let $D = \{D_V\}_{V \in \wp(f(V))}$ be a constraint system and $\Pi$ a finite set of predicate symbols with associated arities. We denote by $\pi$ the set of distinct variables $\{\iota_1, \ldots, \iota_n\}$ where $n$ is the maximum arity of the predicates in $\Pi$.

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

$$G^D := c \mid G^D \land G^D \mid G^D \lor G^D \mid p(x_1, \ldots, x_n)$$

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

By $\mathcal{P}^D$ 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 $p^n$ has the form

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

where $G \in \mathcal{G}^D$ and $\{y_1, \ldots, y_n\} \subseteq V$ 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 2.2, instead, we will use its standard syntax [48].

### 2.3 Classical semantics

In this section we define a generic denotational semantics for observing the set of computed answers of a program. In the following, we assume $D = \{D_V\}_{V \in \wp(f(V))}$ to be a constraint system and $\Pi$ to be a finite set of predicate symbols with associated arity. Moreover, we assume $V \in \wp(f(V))$ and $\pi \subseteq V$.

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

**Definition 2.3.** An *interpretation* over $D_V$ is a function $I : \Pi \to \wp(D_V)$. The set of interpretations over $D_V$ is denoted by $\mathcal{I}^D$. The set $\mathcal{I}^D$ is a complete lattice w.r.t. the $\subseteq$ ordering defined by $I_1 \subseteq I_2$ if and only if $I_1(p) \subseteq I_2(p)$ for every $p \in \Pi$. The least upper bound and greatest lower bound operations are $\cup$ and $\cap$ defined as

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

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

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

Given an interpretation, i.e., the meaning of a program, we define the evaluation of a goal (query) in the program.

**Definition 2.4.** Given \( \{S, S_1, S_2\} \subseteq \wp(D_V) \) and \( \{x_1, \ldots, x_n\} \subseteq V \) we define\(^1\)

\[
\begin{align*}
S_1 \otimes^{D_V} S_2 &= \{c_1 \times^{D_V} c_2 \mid c_1 \in S_1, c_2 \in S_2 \text{ and } c_1 \times^{D_V} c_2 \text{ is defined}\}, \\
S_1 \oplus^{D_V} S_2 &= S_1 \cup S_2, \\
\exists^{D_V}_{\{x_1, \ldots, x_n\}} S &= \{\exists^{D_V}_{x_1} \cdots \exists^{D_V}_{x_n} c \mid c \in S\}.
\end{align*}
\]

We define \( \mathcal{CA}^{D_V} \) : \( \mathbb{G}^{D_V} \times \mathbb{I}^{D_V} \rightarrow \wp(D_V) \) as follows:

\[
\begin{align*}
\mathcal{CA}^{D_V} [\{\} | I] &= \{c\} \\
\mathcal{CA}^{D_V} [G_1 \text{ and } G_2] I &= \mathcal{CA}^{D_V} [G_1] I \otimes^{D_V} \mathcal{CA}^{D_V} [G_2] I \\
\mathcal{CA}^{D_V} [G_1 \text{ or } G_2] I &= \mathcal{CA}^{D_V} [G_1] I \oplus^{D_V} \mathcal{CA}^{D_V} [G_2] I \\
\mathcal{CA}^{D_V} [p(x_1, \ldots, x_n)] I &= \exists^{D_V}_{\{x_1, \ldots, x_n\}} \left(\{\delta^{D_V}_{x_1, \ldots, x_n}, y_1, \ldots, y_n\} \otimes^{D_V} I(p)\right).
\end{align*}
\]

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

**Definition 2.5.** Given a program \( P \in \mathbb{P}^{D_V} \), the **immediate consequence operator** \( T_P : \mathbb{I}^{D_V} \rightarrow \mathbb{I}^{D_V} \) is defined as

\[
T_P(I)(p^n) = \begin{cases} \\
\exists^{D_V}_{\{x_1, \ldots, x_n\}} \left(\{\delta^{D_V}_{x_1, \ldots, x_n}, y_1, \ldots, y_n\} \otimes^{D_V} \mathcal{CA}^{D_V} [G] I\right) \\
&\text{if } p(y_1, \ldots, y_n) \leftarrow G \in P \\
\emptyset &\text{otherwise,}
\end{cases}
\]

for every predicate symbol \( p^n \in \mathbb{I} \).

**Proposition 2.6.** Given a program \( P \in \mathbb{P}^{D_V} \), \( T_P \) is a continuous map.

By using Theorem 1.2, Proposition 2.6 allows us to give the following definition.

**Definition 2.7.** Given a program \( P \in \mathbb{P}^{D_V} \), we define its computed answer semantics as

\[
\mathcal{S}_P = \bigcup_{n \geq 0} T_P^\uparrow(n(\bot)).
\]

---

\(^1\)We can assume there is a total ordering on \( V \), so that the definition of \( \exists \) is not ambiguous.
2.3. Classical call pattern semantics

A call pattern is every procedure call that is selected during resolution of a query in a program, instantiated with the current partial computed answer. Call patterns are very important for program analysis since they describe the set of all possible instantiations of the arguments of a procedure at call time \[.\] Call patterns for a logic program \(P\) are often computed by using a computed answer semantics like that described before, applied to a transformation of the program \(P\) (\textit{magic-set transformation}). We do not like this approach, since it is a special mechanism, designed for call patterns only, and cannot be easily extended to deal with meta-logical built-in’s. Moreover, a very simple semantics which deals directly with call patterns can be defined, as we are going to show in this section. A similar semantics has been defined in [34].

We assume \(\Pi\) to be a finite set of predicate symbols with associated arity. Moreover, we assume \(V \in \wp(V)\) to contain the variables \(\pi\) and the distinct variables \(K = \{\kappa_1, \ldots, \kappa_n\}\), where \(n\) is the largest arity of the predicates in \(\Pi\). We assume \(\pi \cap K = \emptyset\). While the variables in \(\pi\) are used to represent the arguments of a procedure \(p\) in \(I(p)\), where \(I\) is an interpretation, the variables in \(K\) are used to represent the arguments of every call pattern selected during the resolution of \(p\). These can be call patterns for \(p\) or for any other predicate contained in \(\Pi\) (since \(p\) can call itself and any other predicate as well, as dictated by its definition in the program).

Compare the definition below with Definition 2.3.

Definition 2.8. A call pattern interpretation over \(D_V\) is a function \(I : \Pi \to \wp(D_V \cup (D_V \times \Pi))\). The set of call pattern interpretations over \(D_V\) is denoted by \(\mathbb{P}_{cp}^{D_V}\). The set \(\mathbb{P}_{cp}^{D_V}\) is a complete lattice w.r.t. the \(\leq\) ordering defined by \(I_1 \leq I_2\) if and only if \(I_1(p) \subseteq I_2(p)\) for every \(p \in \Pi\). The least upper bound and greatest lower bound operations are \(\cup\) and \(\cap\) defined as

\[
(\cup_{j \in J} \{I_j\})(p) = \cup_{j \in J} (I_j(p)) , \quad (\cap_{j \in J} \{I_j\})(p) = \cap_{j \in J} (I_j(p)) ,
\]

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

Definition 2.9. Given \(\{S, S_1, S_2\} \subseteq \wp(D_V \cup (D_V \times \Pi))\) and \(\{x_1, \ldots, x_n\} \subseteq V\), we define

\[
S_1 \otimes_x^{D_V} S_2 = \{c_1 \ast^{D_V} c_2 \mid c_1 \in S_1 \cap D_V, c_2 \in S_2 \cap D_V, c_1 \ast^{D_V} c_2 \text{ is defined}\} \cup

\cup \{\langle c_1 \ast^{D_V} c_2, q \rangle \mid c_1 \in S_1 \cap D_V, \langle c_2, q \rangle \in S_2, c_1 \ast^{D_V} c_2 \text{ is defined}\} \cup

\cup \{\langle c_1, q \rangle \in S_1\},
\]

\[
\psi_{cp}^{D_V} S_2 = S_1 \cup S_2 ,
\]

\[
\psi_{cp}^{D_V} \langle c, q \rangle = \langle \psi_{cp}^{D_V} c, q \rangle
\]

\[
\psi_{cp}^{D_V} \{x_1, \ldots, x_n\} S = \{\psi_{cp}^{D_V} \cdots \psi_{cp}^{D_V} s \mid s \in S\} .
\]
We define $\mathcal{CP}^{D_V}[c] : \mathbb{P}^{D_V} \times \mathbb{P}_c^{D_V} \to \varphi(D_V \cup (D_V \times \Pi))$ as

$$\mathcal{CP}^{D_V}[c]I = \{c\}$$

$$\mathcal{CP}^{D_V}[G_1 \text{ and } G_2]I = \mathcal{CP}^{D_V}[G_1]I \otimes_{\mathbb{P}_c^{D_V}} \mathcal{CP}^{D_V}[G_2]I$$

$$\mathcal{CP}^{D_V}[G_1 \text{ or } G_2]I = \mathcal{CP}^{D_V}[G_1]I \oplus_{\mathbb{P}_c^{D_V}} \mathcal{CP}^{D_V}[G_2]I$$

$$\mathcal{CP}^{D_V}[p(x_1, \ldots, x_n)]I = \{(\delta^{D_V}_{(\bar{x}_1, \ldots, \bar{x}_n)}, p)\} \cup$$

$$\cup \exists_{\bar{x}_1, \ldots, \bar{x}_n} \left(\{\delta^{D_V}_{(\bar{x}_1, \ldots, \bar{x}_n)}, g\} \otimes^{D_V} I(p)\right).$$

**Definition 2.10.** Given a program $P \in \mathbb{P}^{D_V}$, the call pattern immediate consequence operator $T_P^{cp} : \mathbb{P}_c^{D_V} \to \mathbb{P}_c^{D_V}$ is defined as

$$T_P^{cp}(I)(p^n) = \begin{cases} \exists_{\mathbb{P}_c^{D_V} \setminus \{K \cup \{1, \ldots, n\}\}} \left(\{\delta^{D_V}_{(\bar{x}_1, \ldots, \bar{x}_n)}, g\} \otimes^{D_V} \mathcal{CP}^{D_V}[G]I\right) & \text{if } p(y_1, \ldots, y_n) \leftarrow G \in P \\ \emptyset & \text{otherwise,} \end{cases}$$

for every predicate symbol $p^n \in \Pi$.

**Proposition 2.11.** Given a program $P \in \mathbb{P}^{D_V}$, $T_P^{cp}$ is a continuous map.

By using Theorem 1.2, Proposition 2.11 allows us to give the following definition.

**Definition 2.12.** Given a program $P \in \mathbb{P}^{D_V}$, its call pattern semantics is

$$\mathcal{S}^{cp}_P = \bigcup_{n \geq 0} T_P^{cp} \uparrow_n(\bot).$$

### 2.4 Existential Herbrand constraints

In this section, we define the constraint system of existential Herbrand constraints, through which logic programming can be seen as an instance of constraint logic programming. 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

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

Suppose $c \in C_V$. Then 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 [63] that if there exists $\theta \in \Theta_V$ such that $c\theta$ is true, then $c$ can be put in normal form $\text{mgu}(c) = \{t^1_j = t^2_j \mid j \in J\}$, where $t^1_j \in V$ are distinct variables which do not occur in $t^2_j$ for every $j, k \in J$ and the variables in $\text{mgu}(c)$ are all contained in the variables of $c$. Moreover, we have $c\theta$ is true if and only if $\text{mgu}(c)\theta$ is true. If no $\theta$ exists such that $c\theta$ is true, then $\text{mgu}(c)$ and
hence the normal form of \( c \) are undefined. If \( c \) is in normal form, \( c \) can be seen as a substitution, and we use the notation \( c(v) \) meaning the term \( t \) on the right of an equality \( v = t \in c \) if such an equality exists, \( v \) itself otherwise. Moreover, every substitution can be seen as an existential Herbrand constraint without existential variable. Namely, we define \( \mathbb{E}(\theta) = \exists \theta \{ v = \theta(v) \mid v \in \text{dom}(\theta) \} \).

Let \( \mathcal{V} \) and \( \mathcal{W} \) be disjoint infinite sets of variables. For each \( V \in \wp_1(\mathcal{V}) \), we have a set of constraints, called existential Herbrand constraints, given by

\[
H_V = \left\{ \exists_W c \left| \begin{array}{l}
W \in \wp_1(\mathcal{W}),
\text{there exists } \theta \in \Theta_{V \cup \mathcal{W}} \text{ such that } \text{rng}(\theta) \subseteq V \text{ and } c\theta \text{ is true}
\end{array} \right. \right\}.
\]

Here, \( \mathcal{V} \) are called the \textit{program variables} and \( \mathcal{W} \) the \textit{existential variables}. The set of solutions of an existential Herbrand constraint is defined as

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

Hence \( \text{sol}_V(\exists_W c) = \text{sol}_V(\exists_W \text{mgu}(c)) \). A constraint \( \exists_W c \) is said to be in \textit{normal form} if \( c \) is in normal form.

**Proposition 2.13.** Given \( V \in \wp_1(\mathcal{V}) \) and \( h \in H_V \), the set \( \text{sol}_V(h) \) is downward closed.

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 \textit{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 existential Herbrand constraint can be put in an equivalent normal form, in the following we will consider only normal constraints.

We provide now the operations and the diagonal elements over \( H_V \) such that \( H = \{ H_V \}_{V \in \wp_1(\mathcal{V})} \) becomes a constraint system (Definition 2.1).

**Definition 2.14 (Conjunction on \( H_V \)).** Assuming \( W_1 \cap W_2 = \emptyset \), we have

\[
(\exists_W c_1) \land H_V^c_2 = \begin{cases}
\exists_{W_1 \cup W_2} \text{mgu}(c_1 \cup c_2) & \text{if } \text{mgu}(c_1 \cup c_2) \text{ exists} \\
\text{undefined} & \text{otherwise}.
\end{cases}
\]

**Definition 2.15 (Cylindrification on \( H_V \)).** We define the cylindrification of an existential Herbrand constraint \( \exists_W c \) with respect to a program variable \( x \) as

\[
\exists_x H_V^c = \exists_{W \cup N} c[N/x] \quad \text{where } N \in \mathcal{W} \text{ is fresh}.
\]

That is, we consider the program variable \( x \) as a new existential variable \( N \).

---

\(^2\)Note that this is not restrictive because the names of existential variables are irrelevant: given an existential Herbrand constraint \( \exists_W c \), the constraint \( \exists_{W'[\mathcal{W}]} c[W'/W] \) is equivalent to it. Hence we can always assume existential Herbrand constraints to be renamed apart with respect to existential variables.
Definition 2.16 (Diagonal elements on $H_V$). Given two sequences of distinct variables $\langle x_1, \ldots, x_n \rangle$ and $\langle y_1, \ldots, y_n \rangle$ in $V$ whose intersection is empty, we define

$$\delta^H_{\langle x_1, \ldots, x_n \rangle, \langle y_1, \ldots, y_n \rangle} = \{ x_i = y_i \mid i = 1, \ldots, n \}.$$ 

We can instantiate Definition 2.2 with the constraint system $H$. By requiring that the constraints in goals do not contain the special variables in $\pi$ or the existential variables$^3$ in $W$, we obtain the definition of traditional goals and logic programs$^4$. Definitions 2.3, 2.4, 2.5 and 2.7 leads to the definition of the traditional computed answer semantics for logic programs [9]. Similarly, we can instantiate Definitions 2.8, 2.9, 2.10 and 2.12. The resulting semantics is the call pattern semantics for logic programs.

2.5 Elastic semantics

The semantics of a program (Definitions 2.7 and 2.12) is not finitely computable, in general. Instead, its abstract versions, as they will be presented in next chapters, must be finitely computable, since they are designed to be used by an automatic analysis program. In general, the bigger $V$ is, the more expensive is the computation of the abstract counterparts of the $\star^H_V$ and $\exists^H_V$ operations. This means that it is very important to keep $V$ as small as possible. Note that, given $h \in H_V$, we have $h \in H_{V'}$ for every $V' \supseteq V$. Therefore, for any program $P \in \mathbb{P}^H_V$, we have $P \in \mathbb{P}^H_{V'}$. This means that there is a minimal $V$ such that $P \in \mathbb{P}^H_V$. The dimension of this $V$ is given by the number of different variables used in the program plus the maximum arity of the predicates in $P$ (since we must consider the variables in $\pi$, too). We can assume that different clauses of $P$ use the same variables, in order to keep that number small. However, for real programs, even a single clause can use a very large number of variables, and predicates with up to ten arguments are usual.

The solution of this problem is a logical consequence of the observation that even if the number of different variables contained in a clause can be large, the number of different variables used in any of its constraints is in general small. Therefore, we can use the smallest set of variables which are needed at any particular point of the definition of the immediate consequence operator. This leads to the definition of a new semantics, which uses for its computation the exact number of variables which are needed at any point of the program. We provide here a denotational definition of this semantics, which we call elastic semantics. This technique is not new, since many implemented program analysers are defined in such a way [6, 11]. However, we want a formal definition of this technique. This is because an elastic semantics requires two operations on the abstract constraint system, in order to let grow and

$^3$This means that the existential variables are only generated at run time.

$^4$Note that we use a more abstract syntax for logic programs than the one traditionally used in real Prolog compilers. However, a direct translation of pure Prolog programs into our syntax is straightforward.
2.5. Elastic semantics

shrink the set of variables of a constraint. The definition of such operations, in turn, often requires special care in the definition of the abstract constraint system itself. If we know precisely which operations on the abstract constraint system are needed by the abstract semantics, and we implement them, we can claim that this constraint system can be actually used for static analysis, since nothing has been left unspecified (see, for instance, Chapter 5, and, in particular, Subsections 5.5.3 and 5.5.4).

In order for a constraint system to be used for the computation of an elastic semantics, we need three operations that allow to restrict, rename and enlarge the set of variables of a constraint.

**Definition 2.17.** An elastic constraint system over a set of variables \( V \) is a family of sets \( D = \{ D_V \}_{V \in \wp(V)} \) together with four operations: given \( V \in \wp(V) \), we have

1. \( \star^D_V : D_V \times D_V \to D_V \) (infix);
2. \( \text{restrict}^n_{D_V} : D_V \to D_{V \setminus n} \), with \( n \in V \);
3. \( \text{expand}^x_{D_V} : D_V \to D_{V \cup x} \), with \( x \in V \setminus V \);
4. \( \text{rename}^D_{x \to n} : D_V \to D_{(V \setminus x) \cup n} \), with \( x \in V \) and \( n \in V \setminus V \).

As usual, we do not require anything of these operations but their signatures. Later, we will instantiate this scheme with operations over \( H \) and some abstract domains.

The operations above are extended as follows:

\[
\begin{align*}
\text{restrict}_{\{x_1, \ldots, x_n\}}^D c &= \text{restrict}_{x_1}^{D_{V \setminus \{x_2, \ldots, x_n\}}} \cdots \text{restrict}_{x_n}^{D_V} c, \\
\text{expand}_{\{x_1, \ldots, x_n\}}^D c &= \text{expand}_{x_1}^{D_{V \cup \{x_2, \ldots, x_n\}}} \cdots \text{expand}_{x_n}^{D_V} c, \\
\text{rename}_{(x_1, \ldots, x_n) \to (y_1, \ldots, y_n)}^D c &= \text{rename}_{x_1 \to y_1}^{D_{(V \cup \{y_2, \ldots, y_n\}) \setminus \{x_2, \ldots, x_n\}}} \cdots \text{rename}_{x_n \to y_n}^{D_V} c.
\end{align*}
\]

Definition 2.2 gets slightly modified, since we require now that every constraint \( c \) occurring in a goal brings explicitly the set of variables \( V \) such that \( c \in D_V \). In the following, we assume \( D = \{ D_V \}_{V \in \wp(V)} \) to be an elastic constraint system and \( \Pi \) to be a finite set of predicate symbols with associated arity. We denote by \( \pi \) the set of distinct variables \( \{ \pi_1, \ldots, \pi_n \} \) where \( n \) is the maximum arity of the predicates in \( \Pi \).

**Definition 2.18.** By \( G^D \) we refer to the set of (elastic) goals over \( D \), as defined by the grammar

\[
G^D ::= \langle c, V \rangle \mid G^D \text{ and } G^D \mid G^D \text{ or } G^D \mid p(x_1, \ldots, x_n)
\]

where \( c \in D_V, \ p^n \in \Pi \) with \( n \geq 0 \) and \( \{x_1, \ldots, x_n\} \subseteq V \) are distinct variables not in \( \pi \).
By $\mathbb{P}^D$ we refer to the set of (elastic) programs over $D$, i.e., to the set of sets of clauses, at most one for every predicate symbol, where the clause for $p^n$ has the form

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

where $G \in \mathbb{G}^D$ and \{y_1, \ldots, y_n\} $\subseteq \mathcal{V}$ are distinct variables not in $\pi$.

We write $p$ for $p()$ if $p$ has arity $0$.

An elastic interpretation yields a set of constraints for every predicate symbol. However, the set of variables of the constraints is fixed by the arity of the predicate (compare this with Definition 2.3).

**Definition 2.19.** An elastic interpretation over $D$ is a map $I : \Pi \rightarrow \bigcup_{V \in \wp(\mathcal{V})} \wp(D_V)$ such that $I(p^n) \in \wp(D_{t_1, \ldots, t_n})$ for every $p^n \in \Pi$. The set of elastic interpretations over $D$ is denoted by $\mathbb{I}^D$. Elastic interpretations are ordered by set inclusion: $I_1 \leq I_2$ if and only if $I_1(p) \subseteq I_2(p)$ for every $p \in \Pi$. The set $\mathbb{I}^D$ is a complete lattice w.r.t. the $\leq$ ordering. The least upper bound and greatest lower bound operations are $\bigcup$ and $\bigcap$ defined as

$$\bigcup_{j \in J} \{I_j\}(p) = \bigcup_{j \in J}(I_j(p)),$$

$$\bigcap_{j \in J} \{I_j\}(p) = \bigcap_{j \in J}(I_j(p)),$$

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

We define the evaluation of a goal (query, input) in an interpretation in a way similar to Definition 2.4. However, we use a smaller set of variables at any step of the following definition.

**Definition 2.20.** Given $\{V_1, V_2\} \subseteq \wp_f(\mathcal{V})$, $S_1 \subseteq D_{V_1}$ and $S_2 \subseteq D_{V_2}$, we define

$$\langle S_1, V_1 \rangle \otimes^D \langle S_2, V_2 \rangle = \left\{ \left( c_1 \ast_{D_{V_1 \cup V_2}} c_2', \begin{bmatrix} c_1' & \text{c_1' \in S_1, c_2' \in S_2,} \\ c_2' = \text{expand}_{V_1 \setminus V_2} c_1, \\ c_2' = \text{expand}_{V_2 \setminus V_1} c_2, \\ c_1' \ast_{D_{V_1 \cup V_2}} c_2' \text{ is defined} \end{bmatrix} \right), V_1 \cup V_2 \right\},$$

$$\langle S_1, V_1 \rangle \otimes^D \langle S_1, V_2 \rangle = \{ \{\text{expand}_{V_2 \setminus V_1} c | c \in S_1\} \cup \{\text{expand}_{V_1 \setminus V_2} c | c \in S_2\}, V_1 \cup V_2 \}.$$  

We define $\mathcal{C}_D \[ \cdot \] : \mathbb{G}^D \times \mathbb{I}^D \rightarrow \bigcup_{V \in \wp_f(\mathcal{V})} (\wp(D_V) \times \{V\})$ as follows.

$$\mathcal{C}_D \[ \langle c, V \rangle \] I = \{c\}, V$$

$$\mathcal{C}_D \[ G_1 \text{ and } G_2 \] I = \mathcal{C}_D \[ G_1 \] \otimes^D \mathcal{C}_D \[ G_2 \] I$$

$$\mathcal{C}_D \[ G_1 \text{ or } G_2 \] I = \mathcal{C}_D \[ G_1 \] \oplus^D \mathcal{C}_D \[ G_2 \] I$$

$$\mathcal{C}_D \[ p(x_1, \ldots, x_n) \] I = \{\text{rename}_{D_{\{x_1, \ldots, x_n\}} \rightarrow D_{\{x_1, \ldots, x_n\}}} I(p), \{x_1, \ldots, x_n\}\}.$$
Note that the variable renaming at procedure call is accomplished through the use of the renaming operation rather than through the combined application of a diagonal element and of the cylindrification operation (compare this with Definition 2.4). We will discuss later the many advantages of this change.

The lemma below assures that $\mathcal{CA} D \parallel$ is well defined, i.e., the signatures of the operations over $D$ are respected.

**Lemma 2.21.** Given $G \in G^D$ and $I \in I^D$ we have $\mathcal{CA} D [G] I = \langle S, V \rangle$ with $S \in \wp(D_V)$.

We can define now the elastic immediate consequence operator.

**Definition 2.22.** Given a program $P \in P^D$, the elastic immediate consequence operator $T_P^{el} : I^D \to I^D$ is defined as

$$T_P^{el}(I)(p^n) = \begin{cases} 
\text{rename}_{D_{\{y_1, \ldots, y_n\}}}^{D_{\{y_1, \ldots, y_n\}}} \text{expand}_{D_{\{y_1, \ldots, y_n\}}}^{D_{\{y_1, \ldots, y_n\}}} \text{restrict}_{V \setminus \{y_1, \ldots, y_n\}} I(S) & \text{if } p(y_1, \ldots, y_n) \leftarrow G \in P \text{ and } \mathcal{CA} D [G] I = \langle S, V \rangle \\
\emptyset & \text{otherwise,}
\end{cases}$$

for every predicate symbol $p^n \in \Pi$.

**Proposition 2.23.** Given a program $P \in P^D$, $T_P^{el}$ is a continuous map.

By using Theorem 1.2, Proposition 2.23 allows us to give the following definition.

**Definition 2.24.** Given a program $P \in P^D$, we define its elastic semantics as

$$\mathcal{S}_P^{el} = \bigcup_{n \geq 0} T_P^{el} \downarrow_n (\bot).$$

### 2.5.1 Elastic semantics for call patterns

An elastic semantics for call patterns is similar to the elastic semantics for computed answers defined above. For the sake of completeness, we give below its definition.

**Definition 2.25.** A call pattern elastic constraint system is an elastic constraint system (Definition 2.17) with distinguished elements $\delta_{x, \bar{x}}^{D_V}$ for any $V \in \wp_f(V)$ and $\bar{x}$ and $\bar{y}$ in $V$.

**Definition 2.26.** A call pattern elastic interpretation over $D$ is a map $I : \Pi \to \bigcup_{V \in \wp(\pi \cup X)} \wp(D_V \setminus \{y, z\} \cup (D_V \times \Pi))$ such that $I(p^n) \in \wp(D_{\{x_1, \ldots, x_n\}} \cup (D_K \cup \{y, z\} \times \Pi))$ for every $p^n \in \Pi$. The set of call pattern elastic interpretations over $D$ is denoted by $I^D_{cp}$. Call pattern elastic interpretations are ordered by set inclusion: $I_1 \leq I_2$ if and only if $I_1(p) \subseteq I_2(p)$ for every $p \in \Pi$. The set $I^D_{cp}$ is a complete lattice w.r.t. the
\( \leq \) ordering. The least upper bound and greatest lower bound operations are \( \cup \) and \( \cap \) defined as

\[
(\cup_{j \in J} \{I_j\})(p) = \cup_{j \in J} (I_j(p)) , \quad (\cap_{j \in J} \{I_j\})(p) = \cap_{j \in J} (I_j(p)) ,
\]

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

We define the evaluation of a goal (query, input) in an interpretation in a way similar to that of Definition 2.9. However, we use a smaller set of variables at any step of the following definition.

**Definition 2.27.** Given \( \{V_1, V_2\} \subseteq \wp(\mathcal{V}) \), \( S_1 \subseteq D_{V_1 \setminus K} \cup (D_{V_1} \times \Pi) \) and \( S_2 \subseteq D_{V_2 \setminus K} \cup (D_{V_2} \times \Pi) \), we define

\[
\langle S_1, V_1 \rangle \otimes^D_{cp} \langle S_2, V_2 \rangle = \left\{ \begin{array}{ll}
\{c'_1 \times^D_{D(1 \cup V_2) \setminus K} c'_2 | c_1 \in S_1, c_2 \in S_2, \\
c'_1 = \text{expand}_{D_{V_1 \setminus K \setminus V_1}} c_1, \\
c'_2 = \text{expand}_{D_{V_2 \setminus K \setminus V_2}} c_2, \\
c'_1 \times^D_{D(1 \cup V_2) \setminus K} c'_2 \text{ is defined} \}
\end{array} \right\} \cup
\left\{ \begin{array}{ll}
\{\langle c_1, p \rangle | \langle c_1, p \rangle \in S_1, V_1 \cup V_2 \}
\end{array} \right\},
\]

and

\[
\langle S_1, V_1 \rangle \otimes^D_{cp} \langle S_1, V_2 \rangle = \langle \text{expand}_{D_{V_1 \setminus K \setminus V_1}} c | c \in S_1 \rangle \cup
\left\{ \begin{array}{ll}
\langle \text{expand}_{D_{V_2 \setminus K \setminus V_1}} c, p \rangle | \langle c, p \rangle \in S_1 \}
\end{array} \right\} \cup
\left\{ \begin{array}{ll}
\langle \text{expand}_{D_{V_2 \setminus K \setminus V_2}} c | c \in S_2 \}
\end{array} \right\} \cup
\left\{ \begin{array}{ll}
\langle \text{expand}_{D_{V_1 \setminus V_2}} c, p \rangle | \langle c, p \rangle \in S_2, V_1 \cup V_2 \}
\end{array} \right\}.
\]

Given \( S \in \bigcup_{V \in \wp(V)} \wp(D_{V \setminus K} \cup (D_V \times \Pi)) \), we define

\[
\text{rename}^D_{x \rightarrow y}(S) = \text{rename}^D_{x \rightarrow y} \{ s \in S | s \in D_{V \setminus K} \} \cup \{ \langle \text{rename}^D_{x \rightarrow y} s, p \rangle | \langle s, p \rangle \in S \}
\]

\[
\text{expand}^D_{x}(S) = \text{expand}^D_{x \setminus K} \{ s \in S | s \in D_{V \setminus K} \} \cup \{ \langle \text{expand}^D_{x} s, p \rangle | \langle s, p \rangle \in S \}
\]

\[
\text{restrict}^D_{x}(S) = \text{restrict}^D_{x \setminus K} \{ s \in S | s \in D_{V \setminus K} \} \cup \{ \langle \text{restrict}^D_{x} s, p \rangle | \langle s, p \rangle \in S \}.
\]
Moreover, we define \( \mathcal{CP}^D \) : \( \mathcal{G}^D \times \mathcal{I}^D \to \bigcup_{V \in \wp_f(\mathcal{V})} \left( \wp(D_{V \setminus K} \cup (D_V \times \Pi)) \times \{V\} \right) \) as

\[
\begin{align*}
\mathcal{CP}^D[\langle c, V \rangle] & = \{\langle c, V \rangle\} \\
\mathcal{CP}^D[G_1 \text{ and } G_2] & = \mathcal{CP}^D[G_1] \otimes_{cp} \mathcal{CP}^D[G_2] \\
\mathcal{CP}^D[G_1 \text{ or } G_2] & = \mathcal{CP}^D[G_1] \oplus_{cp} \mathcal{CP}^D[G_2] \\
\mathcal{CP}^D[p(x_1, \ldots, x_n)] & = \langle \langle \delta^{D}_{\{x_1, \ldots, x_n\} \cup K}, p \rangle \rangle \cup \text{rename}_{\{x_1, \ldots, x_n\} \cup \{y_1, \ldots, y_n\}} I(p), \{x_1, \ldots, x_n\} \cup K \rangle.
\end{align*}
\]

The lemma below assures that \( \mathcal{CA}^D \) is well defined, i.e., the signatures of the operations over \( D \) are respected.

**Lemma 2.28.** Given \( G \in \mathcal{G}^D \) and \( I \in \mathcal{I}^D \), we have \( \mathcal{CP}^D[G]I = \langle S, V \rangle \) with \( S \in \wp(D_{V \setminus K} \cup (D_V \times \Pi)) \).

We can define now the call pattern elastic immediate consequence operator.

**Definition 2.29.** Given a program \( P \in \mathbb{P}^D \), the call pattern elastic immediate consequence operator \( T_{cp,ela}^P : \mathcal{I}^D \to \mathcal{I}^D \) is defined as

\[
T_{cp,ela}^P(I)(p^n) = \begin{cases} 
\text{rename}_{\{y_1, \ldots, y_n\} \cup K} \text{ expand}_{\{y_1, \ldots, y_n\} \cup K} D_{V \setminus \{y_1, \ldots, y_n\} \cup K} (S) \\
\text{if } p(y_1, \ldots, y_n) \leftrightarrow G \in P \text{ and } \mathcal{CP}^D[G]I = \langle S, V \rangle \\
\emptyset \\
on \text{ otherwise,} 
\end{cases}
\]

for every predicate symbol \( p^n \in \Pi \).

**Proposition 2.30.** Given a program \( P \in \mathbb{P}^D \), \( T_{cp,ela}^P \) is a continuous map.

By using Theorem 1.2, Proposition 2.30 allows us to give the following definition.

**Definition 2.31.** Given a program \( P \in \mathbb{P}^D \), we define its call pattern elastic semantics as

\[
S_{cp,ela}^P = \bigcup_{n \geq 0} T_{cp,ela}^P \upharpoonright_n(\perp).
\]

### 2.5.2 Elastic semantics over Herbrand

In this subsection we show how to instantiate the elastic semantics described this section with the set of existential Herbrand constraints defined in Section 2.4.

We have already defined the \( \times^R \) operation (Definition 2.14) and diagonal elements over \( H_V \) (Definition 2.16). The other three operations required by Definition 2.17 are given below.
Definition 2.32 (Restriction, expansion and renaming on $H_V$). Given $V \in \mathcal{P}_f(V)$, we define $\text{restrict}^V_n : H_V \mapsto H_{V \setminus n}$, with $n \in V$, $\text{expand}^V_x : H_V \mapsto H_{(V \setminus x) \cup x}$, with $x \in V$ and $n \notin V$ as

\[
\begin{align*}
\text{restrict}^V_n (\exists_W c) &= \exists_{W \cup N} c \left[ N/n \right] \quad \text{with } N \in \mathcal{W} \text{ fresh,} \\
\text{expand}^V_x (\exists_W c) &= \exists_W c, \\
\text{rename}^V_{x \rightarrow n} (\exists_W c) &= \exists_W (c[n/x]) .
\end{align*}
\]

An interesting result is that the cylindrification operation (Definition 2.15) can be defined in terms of the three functions above.

Proposition 2.33. Given $V \in \mathcal{P}_f(V)$, $x \in V$ and $\exists_W c \in H_V$, we have

\[
\exists^V_x (\exists_W c) = \text{restrict}^V_n (\text{expand}^V_{x \cup n} (\text{rename}^V_{x \rightarrow n} (c))) ,
\]

with $n \in V \setminus V$.

### 2.6 An equivalence result

In this section we show that the classical semantics over existential Herbrand constraints coincides with the elastic semantics over the same set of constraints, both for computed answers and for call patterns. This result allows us to use the elastic semantics as an optimised version of the classical semantics over existential Herbrand constraints.

We know that a program to be used with the elastic semantics contains the explicit information about the variables of its constraints (Definition 2.18), while this is implicit in a program to be used with the classical semantics (Definition 2.2). In the case of existential Herbrand constraints, the variables of a constraint can be obtained from the same constraint since

\[
\exists_W c \in H_{\text{vars}(c) \cap V} .
\]

Therefore, with an abuse of notation we can say that $\mathbb{P}^H \subseteq \mathbb{P}^f$ and $\mathbb{G}^H \subseteq \mathbb{G}^f$ for every $V \in \varphi_f(V)$. By using this notation, what we are going to prove in this section is that $S_P = S^{\text{ela}}_P$ and $S^H_P = S^{\text{ela}}_P$ for every $P \in \mathbb{P}^H$.

Assume $I \in \mathbb{H}$ be an elastic interpretation. If $\pi \subseteq V$ then $I$ is an interpretation as defined in Definition 2.3, i.e., $I \in \mathbb{H}_V$. Therefore, the propositions below are well given.

Proposition 2.34. Let $V \in \varphi_f(V)$ be such that $\pi \subseteq V$. Let $G \in \mathbb{G}^H$ and $I \in \mathbb{H}$. Then $\mathcal{C}A^H \llbracket G \rrbracket I = (\mathcal{C}A^{\text{H}_V} \llbracket G \rrbracket I, V')$ with $V' \subseteq V$.

Proposition 2.35. Let $V \in \varphi_f(V)$ be such that $\pi \subseteq V$. Let $P \in \mathbb{P}^H$ and $I \in \mathbb{H}$. We have $T_P(I) = T^{\text{ela}}_P(I)$. 
Theorem 2.36. Let $V \in \varphi_f(V)$ be such that $\pi \subseteq V$. Let $P \in \mathbb{P}^{H_V}$. We have
\[ S_P = S_P^{da}. \]

Identical arguments can be used for call patterns.

Proposition 2.37. Let $V \in \varphi_f(V)$ be such that $\pi \cup K \subseteq V$. Let $G \in \mathbb{G}^{H_V}$ and $I \in \mathbb{I}^{H_{cp}}$. Then $\mathcal{C}^{\mathbb{P}}[G]I = (\mathcal{C}^{\mathbb{P}}[G], V')$ with $V' \subseteq V$.

Proposition 2.38. Let $V \in \varphi_f(V)$ be such that $\pi \cup K \subseteq V$. Let $P \in \mathbb{P}^{H_V}$ and $I \in \mathbb{I}^{H_{cp}}$. We have $T^{cp}_P(I) = T^{cp,da}_P(I)$.

Theorem 2.39. Let $V \in \varphi_f(V)$ be such that $\pi \cup K \subseteq V$. Let $P \in \mathbb{P}^{H_V}$. We have
\[ S^{cp}_P = S^{cp,ela}_P. \]

2.6.1 Classical vs elastic

Theorems 2.36 and 2.39 guarantee that we can compute the classical semantics over existential Herbrand constraints by using its elastic versions. There are many advantages of an elastic semantics w.r.t. a classical semantics.

1. Since we designed the elastic semantics with efficiency issues in mind (see Section 2.5), it is not surprising that an elastic immediate consequence operator allows a more efficient calculation of its (possibly abstract) Kleene sequence (Definitions 2.24 and 2.31). This is because if a constraint contains a small subset of the whole set of program variables, that subset only is taken into account during the computation of the abstract operations. Moreover, an elastic semantics uses an explicit renaming operation for procedure call (Definitions 2.20 and 2.27) rather than a combination of diagonal elements, conjunction and cylindrification (Definitions 2.4 and 2.9). Since the abstract counterparts of the renaming operation are usually very efficient, this still reduces the computational complexity of the calculation of the Kleene sequence.

2. Since constraints in goals are required to belong to $H_V$ but $V$ is not fixed as in the case of the classical semantics, we can abstract them by using as small a set $V$ as possible. This has important consequences in the space requirement of the abstract analysis, since the dimension of the abstraction of a constraint, in general, grows quickly with the dimension of $V$.

3. The abstract counterparts of the renaming operation are usually very precise, while the composition of the abstract counterparts of diagonal elements, conjunction and cylindrification can introduce some loss of precision. This justifies the experimental observation that an elastic semantics leads in general to more precise abstract analyses than its classical version (the expand operation does not introduce any loss of precision, in general, since its abstract counterparts are usually very precise).
The above considerations show that an elastic semantics must be preferred in general to its classical version. Note that these considerations are derived by practical as well as experimental arguments.

2.7 Collecting semantics

We are interested in using the semantics for computed answers (Section 2.3), its elastic version (Section 2.5), the semantics for call patterns (Subsection 2.3.1) or its elastic version (Subsection 2.5.1) instantiated with existential Herbrand constraints as concrete semantics for abstract interpretation of logic programs. In order to do this, these semantics must be put in their collecting version [26].

**Definition 2.40.** The collecting version of a semantics over the constraint system $D = \{D_V\}_{V \in \mathcal{P}(V)}$ is the same semantics over the power-set constraint system $\varphi(D) = \{\varphi(D_V)\}_{V \in \mathcal{P}(V)}$ whose operations are the point-wise extension of the operations over $D$. Namely, if $\text{op}^D : D_V \rightarrow D_V$, then $\text{op}^{\varphi(D)} : \varphi(D_V) \rightarrow \varphi(D_V)$ is defined as $\text{op}^{\varphi(D)}(s) = \{\text{op}^D(s) \mid s \in S$ and $\text{op}^D(s)$ is defined$. Distinguished elements of $D$ are mapped in elements of $\varphi(D)$ as their singleton set. For instance, $\delta^{\varphi(D)}_{x,y} = \{\delta^D_{x,y}\}$.

Since the operations over the collecting semantics are defined through the same operations over the non collecting semantics, the collecting semantics inherits some properties of the non collecting one. For instance, Theorems 2.36 and 2.39 still hold for the collecting semantics over the constraint system $\varphi(H)$.

2.8 Abstract compilation

Abstraction compilation [44, 6] is an application of abstract interpretation designed for the semantics-based analysis of programs. The idea is that the constraints contained in a program are abstracted once and for all at the beginning of the computation of the abstract fixpoint. This leads, in general, to a more efficient computation of the abstract fixpoint.

Assume we have two constraint systems $C = \{C_V\}_{V \in \mathcal{P}(V)}$ and $A = \{A_V\}_{V \in \mathcal{P}(V)}$, such that, for every $V \in \mathcal{P}(V)$, $C_V$ is partially ordered w.r.t. $\leq$, $A_V$ is partially ordered w.r.t. $\preceq$ and there is a Galois connection $\langle \alpha_V, \gamma_V \rangle$ between $C_V$ and $A_V$ such that all the operators and the diagonal elements of Definition 2.1 (and 2.17, if we are interested in an elastic semantics) are correct. The Galois connection can be extended into a Galois connection between $(\varphi(C_V), \subseteq)$ and $(\varphi(A_V), \subseteq)$ by defining $\alpha(P) = \{\alpha(c) \mid c \in P\}$ for any $P \in \varphi(C_V)$ and $Q_1 \subseteq Q_2$ 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.
Definition 2.41 (Abstract compilation). Let $C$ and $A$ be as above. Given a program $P \in \mathbb{P}^{C\downarrow}$, we define $\alpha(P) \in \mathbb{P}^{A\downarrow}$ as the program obtained from $P$ by substituting the constraints in the clauses with their abstraction through $\alpha_V$. Similarly, if $P \in \mathbb{P}^C$, then $\alpha(P) \in \mathbb{P}^A$ is the elastic program obtained from $P$ by substituting every constraint $\langle c, V \rangle$ in the clauses of $P$ with $\langle \alpha_V(c), V \rangle$.

Since all the immediate consequence operators we have defined in this chapter work over set-theoretical complete lattices and are additive, by the general theory of fixpoint and abstract interpretation we conclude that, for every $P \in \mathbb{P}^{C\downarrow}$, we have

$$\alpha(\lfloor p(T_P) \rfloor) \sqsubseteq \lfloor p(T_{\alpha(P)}) \rfloor,$$

since $\alpha \circ T_P \sqsubseteq T_{\alpha(P)} \circ \alpha$, and

$$\alpha(\lfloor p(T_P) \rfloor) = \lfloor p(T_{\alpha(P)}) \rfloor,$$

whenever $T_P$ is precise w.r.t. $T_{\alpha(P)}$, i.e., $\alpha \circ T_P = T_{\alpha(P)} \circ \alpha$.

The same results hold for $T^{\text{ela}}_P$, $T^{\text{cp}}_P$ and $T^{\text{cp,ela}}_P$.

2.9 Conclusions

We have shown how a concrete semantics for pure logic languages can be computed in a bottom-up fashion, and we have shown how this computation can be optimised through the use of an elastic version of the semantics.

Often, the collecting semantics is assumed to use one constraint to denote a procedure, instead of a set of constraints. This requires that the constraint system is a complete lattice, since a least upper operation is used during the computation of the abstract fixpoint. We do not discuss this point further here. We just observe that this approach, which leads to a semantics which is an abstract interpretation of that defined in Section 2.7, is not applicable to the case of the semantics of real logic languages like Prolog, where sequences of constraints must be used [77]. Moreover, note that, if this approach has to be used, an algorithm for computing the least upper bound must be provided on the abstract constraint system.

The definitions and the results of this chapter can be extended to deal with some meta-logical features of real logic languages like Prolog and Gödel. For a detailed treatment of this subject, see [76, 77]. A framework for the abstraction of these semantics is provided in [78].
2.10 Proofs

Proofs of Section 2.3

Proposition 2.6. It is a simple corollary of Definition 2.5, since by structural induction on a goal $G$ it is easy to prove that

$$\mathcal{A}^{D_v} [I_j] = \cup_{j \in J} (\cup_{j \in J} (\mathcal{A}^{D_v} [G] [I_j]))$$

for every $\{I_j\}_{j \in J} \subseteq \mathbb{P}^{D_v}$ with $J \subseteq \mathbb{N}$.

Proposition 2.11. Like in Proposition 2.6.

Proofs of Section 2.4

Proposition 2.13. Given $\theta \in \text{sol}_V(\exists_W c)$ and $\theta' = \theta \sigma$ for some $\sigma \in \Theta^V$, there must be $\theta'' \in \Theta^V_{\cup \mathbb{V}}$ such that $\theta = \theta'|_V$ and $c \theta''$ is true. We have $\theta'|_V = (\theta' \sigma)|_V$ and $\theta'' \sigma$ is idempotent since $\theta \sigma$ is. Given $t_1 = t_2 \in c$, we have $t^1(\theta'(\theta'' \sigma)|_W) = t^1 \theta \sigma(\theta'' \sigma)|_W = t^2(\theta'(\theta'' \sigma)|_W) = t^2 \theta \sigma(\theta'' \sigma)|_W$. This means that $c \theta'(\theta'' \sigma)|_W$ is true, i.e., $\theta' = (\theta'(\theta'' \sigma)|_W)|_V \in \text{sol}_V(\exists_W c)$.

Proofs of Section 2.5

Lemma 2.21. By straightforward induction over $G$. For $G = p(y_1, \ldots, y_n)$, the result follows since $I(p) \in D_{\{t_1, \ldots, t_n\}}$ and by the signature of rename. For the case $G = (c, \mathbb{V})$, we have the thesis since $G \in \mathbb{G}^D$ and (Definition 2.18) we know that we must have $c \in \mathbb{D}_V$.

Proposition 2.23. Like in Proposition 2.6.

Lemma 2.28. By straightforward induction over $G$. For $G = p(y_1, \ldots, y_n)$, the result follows since $I(p) \in D_{\{t_1, \ldots, t_n\} \cup \mathbb{K}}$ and by the signature of rename. For the case $G = (c, \mathbb{V})$, we have the thesis since $G \in \mathbb{G}^D$ and (Definition 2.18) we know that we must have $c \in \mathbb{D}_V$.

Proposition 2.30. Like in Proposition 2.6.

Proposition 2.33.

$$\text{restrict}_{\mathbb{V} \cup \mathbb{N}} (\text{expand}_{\mathbb{V} \cup \mathbb{N}} (\text{rename}_{\mathbb{V} \rightarrow \mathbb{V}} (c))) = \text{restrict}_{\mathbb{V} \cup \mathbb{N}} (\text{expand}_{\mathbb{V} \cup \mathbb{N}} (\exists_W (c[n/x])))$$

$$= \text{restrict}_{\mathbb{V} \cup \mathbb{N}} (\exists_W (c[n/x]))$$

$$= \exists_{\mathbb{V} \cup \mathbb{N}} ((c[n/x])[N/n])$$

(since $n \in \mathbb{V} \setminus \mathbb{V}$).

$$= \exists_{\mathbb{V} \cup \mathbb{N}} (c[N/x]) = \exists_{\mathbb{V} \cup \mathbb{N}} (c[N]) = \exists_{\mathbb{V} \cup \mathbb{N}} (c[N]) .$$

□
2.10. Proofs

Proofs of Section 2.6

Lemma 2.42. Given \( \exists_W c \in H_V, x \in V \) and \( n \in \mathcal{V} \setminus V \), we have
\[
\exists_x^{H_V \cup n} \left( \delta_{x(n)}^{H_V \cup n} \ast^{H_V \cup n} \exists_W c \right) = \text{rename}^{H_V}_{x \to n} (\exists_W c).
\]

Proof. If \( x \not\in \text{rng}(c) \) we have
\[
\exists_x^{H_V \cup n} \left( \delta_{x(n)}^{H_V \cup n} \ast^{H_V \cup n} \exists_W c \right) = \exists_x^{H_V \cup n} \exists_W (c \cup \{ n = c(x) \})
\]
(assuming \( N \in \mathcal{W} \) fresh) = \( \exists_W (c \setminus \{ x = c(x) \}) \cup \{ N = c(x), n = c(x) \} \)
\[
\simeq \exists_W (c \setminus \{ x = c(x) \}) \cup \{ n = c(x) \}
\]
= \( \exists_W (c[n/x]) = \text{rename}^{H_V}_{x \to n} (\exists_W c) \).

If \( x \in \text{rng}(c) \) we have
\[
\exists_x^{H_V \cup n} \left( \delta_{x(n)}^{H_V \cup n} \ast^{H_V \cup n} \exists_W c \right) = \exists_x^{H_V \cup n} (\exists_W (c[n/x] \cup \{ x = n \}))
\]
(assuming \( N \in \mathcal{W} \) fresh) = \( \exists_W (c[n/x] \cup \{ N = n \}) \)
\[
\simeq \exists_W c[n/x] = \text{rename}^{H_V}_{x \to n} (\exists_W c).
\]

Lemma 2.43. Given \( V \in \omega_f(\mathcal{V}), \{z_1, \ldots, z_n\} \subseteq V, \{x_1, \ldots, x_n\} \subseteq \mathcal{V} \setminus V \) and \( h \in H_V \), for every \( n \geq 1 \) we have
\[
\exists_{x_1, \ldots, x_n}^{H_V \cup \{z_1, \ldots, z_n\}} \left( \delta_{x_1(n)}^{H_V \cup \{z_1, \ldots, z_n\}, x_1(n)} \ast^{H_V \cup \{z_1, \ldots, z_n\}} h \right) = \exists_{z_1}^{H_V \cup \{z_1, \ldots, z_n\}} \left( \delta_{z_1(n), z_1}^{H_V \cup \{z_1, \ldots, z_n\}} \ast^{H_V \cup \{z_1, \ldots, z_n\}} \ldots \right.
\]
= \( \exists_{z_1}^{H_V \cup \{z_1, \ldots, z_n\}} \exists_W \text{mgu}(\{x_i = z_i \mid i = 1, \ldots, n+1\} \cup c) \)
\[
\exists_{z_1}^{H_U \cup \{N_1, \ldots, N_{n+1}\}} \exists_W \text{mgu}(\{x_1 = N_1\} \cup \text{mgu}(\{x_i = N_i \mid i = 1, \ldots, n+1\} \cup c[N_1/z_1] \cdots [N_{n+1}/z_{n+1}]) \)
\]
= \( \exists_{z_1}^{H_U \cup \{x_1, \ldots, x_{n+1}\}} \exists_W (c[x_1 = z_1] \cup \text{mgu}(\{x_1 = N_1 \mid i = 1, \ldots, n+1\}) \cup c[N_2/z_2] \cdots [N_{n+1}/z_{n+1}]) \)
\]
= \( \exists_{z_1}^{H_U \cup \{x_1, \ldots, x_{n+1}\}} \{x_1 = z_1\} \ast^{H_U \cup \{x_1, \ldots, x_{n+1}\}} h \)
\]
= \( \exists_{z_1}^{H_U \cup \{x_1, \ldots, x_{n+1}\}} \{x_1 = z_1\} \ast^{H_U \cup \{x_1, \ldots, x_{n+1}\}} h \)
\]
\[ \text{mgu}(\{x_i = N_i \mid i = 2, \ldots, n+1\} \cup c[N_2/z_2] \cdots [N_{n+1}/z_{n+1}]) \)
\]
= \( \exists_{z_1}^{H_U \cup \{x_2, \ldots, x_{n+1}\}} \{x_1 = z_1\} \ast^{H_U \cup \{x_1, \ldots, x_{n+1}\}} h \)
\]
\[ \text{mgu}(\{x_i = N_i \mid i = 2, \ldots, n+1\} \cup c[N_2/z_2] \cdots [N_{n+1}/z_{n+1}]) \)
\]
and the thesis follows by inductive hypothesis. □

**Corollary 2.44.** Given $V' \in \wp(V)$, $\{t_1, \ldots, t_n\} \subseteq V'$ and $\{x_1, \ldots, x_n\} \subseteq V'$ disjoint and $h \in H_{V' \setminus \{x_1, \ldots, x_n\}}$, we have

$$\exists^{H_{V'}}_{\{t_1, \ldots, t_n\}}(\delta^{H_{V'}}_{\{t_1, \ldots, t_n\} \setminus \{x_1, \ldots, x_n\}} \times^{H_{V'}} h) = \text{rename}_{\{t_1, \ldots, t_n\} \setminus \{x_1, \ldots, x_n\}}^{H_{V'}} h.$$  

**Proof.** By Lemma 2.43 and iterated application of Lemma 2.42. □

**Proposition 2.34.** By structural induction on $G$, by using the fact that $\text{expand}$ leaves an existential Herbrand constraint unchanged. The only difficult case is that for procedure call, which has been proved apart in Corollary 2.44 by choosing $V' = V$. □

**Proposition 2.35.** Let $p^n \in \Pi$. If there is no clause for $p$ in $P$ we have $T_P(I)(p) = \emptyset = T_P^{\text{def}}(I)(p)$. Otherwise, let $(y_1, \ldots, y_n) \leftrightarrow G$ be such a clause and $\mathcal{CA}^H[G][I] = \langle \mathcal{CA}^{H_{V'}}[G][I], V' \rangle$ with $V' \subseteq V$ (by Proposition 2.34). We have

$$T_P(I)(p) = \exists^{H_V}_{\{y_1, \ldots, y_n\}} \left( \left( \delta^{H_V}_{\{y_1, \ldots, y_n\} \setminus \{y_1, \ldots, y_n\}} \right) \otimes^{H_V} \mathcal{CA}^{H_{V'}}[G][I] \right)$$

$$= \exists^{H_V}_{\{y_1, \ldots, y_n\}} \left( \left( \delta^{H_V}_{\{y_1, \ldots, y_n\} \setminus \{y_1, \ldots, y_n\}} \right) \otimes^{H_V} \exists^{H_{V'}}_{\{y_1, \ldots, y_n\}} \mathcal{CA}^{H_{V'}}[G][I] \right).$$

Since $V' \subseteq V$ and $\mathcal{CA}^{H_{V'}}[G][I] \in \wp(H_{V'})$ (Lemma 2.21) and the variables in $\pi$ do not occur in the constraints of $G$, we have

$$T_P(I)(p) = \exists^{H_V}_{\{y_1, \ldots, y_n\}} \left( \left( \delta^{H_V}_{\{y_1, \ldots, y_n\} \setminus \{y_1, \ldots, y_n\}} \right) \otimes^{H_V} \exists^{H_{V'}}_{\{y_1, \ldots, y_n\}} \mathcal{CA}^{H_{V'}}[G][I] \right)$$

and since $\mathcal{CA}^{H_{V'}}[G][I] \in \wp(H_{\{y_1, \ldots, y_n\}})$, by Corollary 2.44 we have that

$$T_P(I)(p) = \text{rename}_{\{y_1, \ldots, y_n\} \setminus \{y_1, \ldots, y_n\}}^{H_V \setminus \{y_1, \ldots, y_n\}} \mathcal{CA}^{H_{V'}}[G][I].$$

Since $\text{expand}$ leaves the existential Herbrand constraints unchanged, we have

$$T_P(I)(p) = \text{rename}_{\{y_1, \ldots, y_n\} \setminus \{y_1, \ldots, y_n\}}^{H_{V'} \setminus \{y_1, \ldots, y_n\}} \text{expand}_{\{y_1, \ldots, y_n\} \setminus \{y_1, \ldots, y_n\}}^{H_{V'} \setminus \{y_1, \ldots, y_n\}} \mathcal{CA}^{H_{V'}}[G][I],$$

i.e., $T_P(I)(p) = T_P^{\text{def}}(I)(p)$. □

**Theorem 2.36.** Straight from Definitions 2.7 and 2.24, by Proposition 2.35. □

**Proposition 2.37.** Like Proposition 2.34. □
Proposition 2.38. Let $p^n \in \Pi$. If there is no clause for $p$ in $P$ we have $T^c_P(I)(p) = \emptyset = T^c_{P^c}(I)(p)$. Otherwise, let $p(y_1, \ldots, y_n) \leftarrow G$ be such a clause and let $CP^H[G][I] = \langle CP^H[G][I], V' \rangle$ with $V' \subseteq V$ (by Proposition 2.37). We can rewrite $T^c_P(I)(p)$ as

$$\exists_{\Pi} \left( \exists_{\Pi} \left( \exists_{\Pi} \right) \left( \exists_{\Pi} \right) \right)$$

$$= \exists_{\Pi} \left( \exists_{\Pi} \right)$$

Since $V' \subseteq V$ and $CP^H[G][I] \in \varphi(H_{V'} \cup (H_{V'} \times \Pi))$ (Lemma 2.28), and the variables in $\pi$ do not occur in the constraints of $G$, by letting $T = \{y_1, \ldots, y_n\} \cup K \cup \pi$ we have that $T^c_P(I)(p)$ is

$$\exists_{\Pi} \left( \exists_{\Pi} \right)$$

Since $\text{restrict}_{V' \cup \{K \cup \{y_1, \ldots, y_n\}\}} CP^H[G][I] \in \varphi(H_T \cup (H_T \times \Pi))$, by Corollary 2.44 (with $V' = \{y_1, \ldots, y_n\}$) we have that

$$T^c_P(I)(p) = \text{rename}_{K \cup \{y_1, \ldots, y_n\}} \text{restrict}_{V' \cup \{K \cup \{y_1, \ldots, y_n\}\}} CP^H[G][I],$$

and since $\text{expand}$ leaves the existential Herbrand constraints unchanged, we can rewrite $T^c_P(I)(p)$ as

$$\text{rename}_{K \cup \{y_1, \ldots, y_n\}} \text{expand}_{V' \cup \{K \cup \{y_1, \ldots, y_n\}\}} \text{restrict}_{V' \cup \{K \cup \{y_1, \ldots, y_n\}\}} CP^H[G][I],$$

i.e., $T^c_P(I)(p) = T^c_{P^c}(I)(p)$. □

Theorem 2.39. Straight from Definitions 2.12 and 2.31, by using Proposition 2.38. □
Chapter 3  **Downward closed properties**

You’re the ground I feed on.

Björk,

*Bachelorette*, 1997

In this chapter we study the analysis of downward closed properties of logic programs, which are a very abstract presentation of types. We generalise the construction of the traditional domains for groundness analysis to a very large class of downward closed properties. This is done in such a way that the results enjoy the good properties of those domains. Namely, we obtain abstract domains with a clear representation made of logical formulas and with optimal and well-known abstract operations. Moreover, they can be built by using the linear refinement technique, and, therefore, are provably *optimal* and enjoy the condensing property, which is very important for a goal-independent analysis.

Part of this chapter has been published in [47].

3.1 **Introduction**

A downward (instantiation) closed set of terms represents a property which is maintained during the computation. As is common in work on type analysis, we call such a property a type. Type analysis is therefore just the upward approximation of the success set of a program through types. Type analysis of logic programs is important for verification as well as optimisation of unification. Note that other interesting properties of logic programs, like freeness and sharing, are not types since they are not downward closed.

An important and simple type, which distinguishes whether a term contains variables or not, is groundness [22, 3, 23]. The usual domain for groundness analysis, $Pos$, features some desirable properties: simplicity, human readability, effectivity, usefulness. Moreover, it has been shown [72] that $Pos$ is condensing and is the most precise domain for groundness analysis that does not consider the name of the functors in a term. All these good properties of the $Pos$ domain should have encouraged a generalisation of $Pos$ to a general type domain. Instead, type domains
have been developed, up to now, in a way totally independent from the domain \textit{Pos} for groundness analysis. In [74] the authors define generic type domains with dependencies but, again, this is not a real generalisation of \textit{Pos}, though it can be used for abstract compilation. The only paper which generalises the definition of the domain \textit{Pos} to generic types is [16]. However, in this paper it is assumed that the usual properties of \textit{Pos} hold for all the type domains. For instance, logical conjunction between formulas is used as conjunction operator and Schröder elimination as cylindrification operator. It is not obvious at all that these operators, which are optimal in the case of groundness analysis [23], are even correct in the general case of type analysis and no proof is given. Although in [17] a domain with properties similar to those of \textit{Pos} is built, the way it is constructed is not a generalisation of that used for \textit{Pos} [72].

The type analysis presented in [36] has the same power as a generic type analysis based on \textit{Pos}. However, it is not based on abstract interpretation. Instead, it uses \textit{pre-interpretedations} to model the abstract behaviour of a constraint. Pre-interpretations are extensional descriptions of the models of propositional formulas. This means that, in general, the use of propositional formulas should lead to a more efficient analysis. Moreover, in Chapter 4 we will show that type variables allow us to represent type properties in a very compact and efficient way. We do not know how type variables can be used in their approach. The expressive power of the domains of [36] is smaller than that of our domains, since they constrain a ground term to belong to one type only.

In [58] it is shown that a hierarchy of domains for type analysis can be defined in the same way as it has been done in the case of groundness analysis [72]. Transfinite formulas, which are needed to deal with polymorphism, are used to represent the type domain and generalise the finite formulas of \textit{Pos}. It is shown how a finite representation of these formulas can be achieved through the use of type variables. Two operators, logical conjunction and Schröder elimination, are used as abstract operators over transfinite formulas and are the generalisation of the same operators used for the \textit{Pos} domain for groundness analysis.

In this chapter we use the abstract interpretation framework [26] to generalise the correctness results of [23] and [58] to a much larger class of type domains. Namely, we show that the abstract operators defined in these two papers are correct for every type domain, including then groundness and non-freeness, which were not covered in [58]. Moreover, we present a very weak condition, sufficient to entail the optimality of the abstract operators for a large class of type systems, including groundness, non-freeness and the type systems considered in [58] when speaking of correctness. Note that no optimality result were given in [58], while [23] proves optimality for the case of groundness only. The optimality result is important since it shows that, when our sufficient condition is satisfied, 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.

The correctness and optimality result proved in [23] for the two operations over
the formulas of $\text{Pos}$ cannot be easily lifted to a generic type domain. Given a different type domain, every proof should be rewritten. This is because a direct (Galois) connection between the concrete domain of existential Herbrand constraints and the abstract domain for groundness analysis is defined, and this connection is not parametrised with respect to a generic type.

Instead, we use a two-step generic abstraction from existential Herbrand constraints into a generic type domain. In the first step we show that the collecting domain of existential Herbrand constraints can be abstracted into the set of downward closed sets of substitutions, which we call $\text{Down}$. A downward closed set of substitutions represents the union of the solutions of a set of existential Herbrand constraints. We provide the optimal abstract operators on $\text{Down}$ induced by the corresponding operators on existential Herbrand constraints. Note that this first step is independent from the type analysis at hand. We consider the particular analysis only in the second step when we abstract $\text{Down}$ into a type domain of transfinite formulas. Since the distance between type domains and $\text{Down}$ is much shorter than the distance between type domains and existential Herbrand constraints, the definition of a Galois connection between $\text{Down}$ and the type domain is rather obvious. Moreover, it is quite easy to provide the optimal operators on the type domains induced by the optimal operators on $\text{Down}$.

This chapter is organised as follows. Section 3.2 introduces the domain $\text{Down}$, whose constraints are downward closed sets of substitutions and a subset $\text{Sol}$ of $\text{Down}$ which is isomorphic to $H$, with the optimal counterparts over $\text{Sol}$ of the operators over $H$. Section 3.3 defines the domain $\varphi(H)$ used for a collecting version of the semantics on $H$ and its isomorphic counterpart $\varphi(\text{Sol})$, and shows that $\varphi(\text{Sol})$ can be abstracted into our concrete domain $\text{Down}$. This time, the whole domain $\text{Down}$ is used, rather than its subset $\text{Sol}$. In the rest of the chapter, the abstraction is type-specific. Namely, Section 3.4 defines a generic type system and its induced type domain $T$ formed by transfinite formulas. Moreover, it provides correct operators on $T$ corresponding to the operators on $\text{Down}$, and shows sufficient conditions for their optimality. Since the operators on $\text{Down}$ have been shown optimal with respect to the operators over $\varphi(\text{Sol})$, which is isomorphic to $\varphi(H)$, these correctness and optimality results can be extended from $\varphi(H)$ to $T$. Section 3.5 shows that our work generalises some old results and provides logical domains for type analysis of logic programs. Section 3.6 applies the linear refinement technique to develop the abstract domains for modelling conjunction in a very precise way. We conclude in Section 3.7. The picture below synthesises the various domains considered in this chapter, and their relationships as abstraction (represented by horizontal arrows) or lifting to the powerset (represented by vertical arrows).
collecting $\varphi(H) \leftrightarrow \varphi(Sol) \rightarrow \text{Down} \rightarrow T$

<table>
<thead>
<tr>
<th>non collecting $H \leftrightarrow Sol$</th>
<th>type dependent abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>type independent abstractions</td>
<td></td>
</tr>
</tbody>
</table>

3.2 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 $\theta$, i.e., which can be derived from $\theta$ with computation. For instance, if in a program point we have $y = f(x)$, then it is possible that, as computation proceeds, we have $y = f(g(w))$ and $x = g(w)$. It is not possible, however, that $y = g(w)$. With this interpretation in mind, we can say that a downward closed set of substitutions contains exactly all substitutions which are consistent with the prosecution of computation. For instance, if $S_1$ is the (downward closed) set of substitutions which are consistent with a procedure call $p_1$ and $S_2$ is the (downward closed) set of substitutions which are consistent with a procedure call $p_2$, then $S_1 \cap S_2$ is the (downward closed) set of substitutions which are consistent with the calls $p_1, p_2$ and $p_2, p_1$.

**Definition 3.1 (The Down constraint system).** We define the constraint system (Definition 2.1) $\text{Down} = \{\text{Down}_V\}_{V \in \mathcal{P}(V)}$ where $\text{Down}_V = \varphi(\Theta_V)$. Let $\{S, S_1, S_2\} \subseteq \text{Down}_V$ and $x \in V$. We define

$$S_1 \times_{\text{Down}_V} S_2 = S_1 \cap S_2 \quad \exists_{x \in \text{Down}_V} S \left\{ \theta' \in \Theta_V \mid \text{there exist } \sigma \in S[n/x], \theta \in \Theta_{V \mid n \cdot V} \text{ such that } \theta \leq_{V \mid n \cdot V} \sigma \text{ and } \theta' = \theta|_V \right\},$$

where $n \in V \setminus V$ is fresh and $S[n/x] = \{\sigma[n/x] \mid \sigma \in S\}$.

Given $\langle x_1, \ldots, x_n \rangle$ and $\langle y_1, \ldots, y_n \rangle$ in $V$, we define

$$\partial_{\text{Down}_V}^{\langle x_1, \ldots, x_n \rangle, \langle y_1, \ldots, y_n \rangle} = \{ \theta \in \Theta_V \mid \theta(x_i) = \theta(y_i) \text{ for all } i = 1, \ldots, n \}.$$

While the definition of conjunction is the classical one for the case of downward closed sets of substitution (see, for instance, [72]), 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.

Definition 3.1 should be read as follows. In order to compute the cylindrification of a set $S$ of substitutions, we consider $x$ as a new variable $n$, we instantiate all the substitutions which are obtained from $S$ in such a way, then we select those instantiations $\theta$ such that $\theta|_V$ does not contain $n$. 


3.2. The domains \textit{Down} and \textit{Sol}

Consider for instance a procedure \( p(y) \leftarrow \{ y = f(x) \} \). The set of substitutions consistent with the body of the procedure is \( S = \downarrow \{ \theta \} \) where \( \theta = \{ y \mapsto f(x) \} \). Note that \( \theta' = \{ y \mapsto f(g(x)) \} \notin S \) since we consider idempotent instantiation. Consider now the set of substitutions consistent with the procedure call \( p(y) \). The variable \( x \) in the body of the procedure is existentially quantified. Therefore, it is \textit{not} the same as \( x \) outside the procedure \( p \). This means that \( \theta' \) is consistent with the procedure call \( p(y) \). Indeed, the procedure call \( p(y) \) made in a store in which \( y = f(g(x)) \) succeeds. In Definition 3.1 we consider \( x \) as a new variable \( n \), then we instantiate this new variable in every possible way, even with a term which contains \( x \).

The conjunction operator is closed on \( \text{Down}_V \), as it can be checked easily. The same holds for cylindrification.

\textbf{Proposition 3.2.} Let \( V \in \varphi_f(V) \) and \( S, S_1, S_2 \subseteq \Theta_V \).

\begin{itemize}
  \item[i)] If \( S = \downarrow S \) then \( \exists_x \text{Down}_V S = \downarrow (\exists_x \text{Down}_V S) \) (cylindrification is closed on the set of downward closed sets of substitutions).
  \item[ii)] If \( S_1 \subseteq S_2 \), then \( \exists_x \text{Down}_V S_1 \subseteq \exists_x \text{Down}_V S_2 \) (cylindrification is monotonic).
  \item[iii)] \( S \subseteq \exists_x \text{Down}_V (S) \) (cylindrification is extensive).
\end{itemize}

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.

\textbf{Proposition 3.3.} Let \( V \in \varphi_f(V) \). If \( \Sigma \) contains at least a constant and a functor symbol, then \( \{ \text{sol}_V(h) \mid h \in H_V \} \subseteq \varphi_f(\Theta_V) \).

In spite of this result, the following proposition shows that \( \star_{\text{Down}_V} \) and \( \exists_{\text{Down}_V} \) are closed on the set \( \text{sol}_V(H_V) \).

\textbf{Proposition 3.4.} Let \( V \in \varphi_f(V) \), \( \{ h, h_1, h_2 \} \subseteq H_V \) and \( \bar{x}, \bar{y} \subseteq V \).

\begin{itemize}
  \item[i)] \( \text{sol}_V(h_1) \star_{\text{Down}_V} \text{sol}_V(h_2) = \text{sol}_V(h_1 \star_{H_V} h_2) \),
  \item[ii)] \( \exists_x \text{Down}_V \text{sol}_V(h) = \text{sol}_V (\exists_x^{H_V} h) \) for every \( x \in V \),
  \item[iii)] \( \text{sol}_{x,\bar{y}} = \text{sol}_V (\text{sol}_{x,\bar{y}}^{H_V}) \).
\end{itemize}

Therefore, we can introduce the following definition.

\textbf{Definition 3.5 (The \textit{Sol} constraint system).} We define \( \text{Sol} = \{ \text{Sol}_V \}_{V \in \varphi_f(V)} \), where \( \text{Sol}_V = \{ \text{sol}_V(h) \mid h \in H_V \} \). The \( \star_{\text{Sol}_V} \) and \( \exists_{\text{Sol}_V} \) operators on \( \text{Sol}_V \) are the restriction of the corresponding operators of \( \text{Down}_V \) to \( \text{Sol}_V \). The diagonal elements of \( \text{Sol}_V \) are the diagonal elements 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 \( H_V \). Therefore, we can see \( \text{Sol} \) as an alternative presentation of \( H \), and we know that the corresponding operations coincide (Proposition 3.4). The usefulness of \( \text{Sol} \) is that of presenting an existential Herbrand constraint as a downward closed set of substitutions, i.e., the set of its solutions. This will be very important in the next section.

#### 3.3 The collecting semantics

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

We use \( \cup \) as abstraction map from \( \varphi(\text{Sol}_V) \) into \( \text{Down}_V \). The map \( \cup \) is onto from \( \varphi(\text{Sol}_V) \) into \( \text{Down}_V \). This is because every \( d \in \text{Down}_V \) can be written as the infinite union \( d = \bigcup \{\downarrow \{\theta\} \mid \theta \in d\} \) and every single \( \downarrow \{\theta\} \) belongs to \( \text{Sol}_V \), as shown by the following proposition.

**Proposition 3.6.** Given \( V \in \varphi_f(\mathcal{V}) \) and \( \theta \in \Theta_V \) we have \( \text{sol}_V(\text{Eq}(\theta)) = \downarrow \{\theta\} \).

It can be shown that \( \cup \) is additive from \( \varphi(\text{Sol}_V) \) into \( \text{Down}_V \).

**Proposition 3.7.** Given \( V \in \varphi_f(\mathcal{V}) \) and \( \{p_i\}_{i \in I} \subseteq \varphi(\text{Sol}_V) \) with \( I \subseteq \mathbb{N} \), we have

\[
\bigcup \left( \bigvee_{i \in I} \varphi(\text{Sol}_V) p_i \right) = \bigvee_{i \in I} \text{Down}_V \left( \bigcup p_i \right).
\]

The above proposition and the fact that \( \cup \) is onto entail that \( \cup \) induces an Galois insertion from \( \varphi(\text{Sol}_V) \) into \( \text{Down}_V \).

Moreover, it can be shown that the operators and the diagonal elements on \( \text{Down}_V \) are the best abstraction through \( \cup \) of the corresponding operators and diagonal elements on \( \varphi(\text{Sol}_V) \). Actually, the operators on \( \text{Down}_V \) are precise.

**Proposition 3.8.** Let \( V \in \varphi_f(\mathcal{V}) \), \( \{p_1, p_2, p\} \subseteq \varphi(\text{Sol}_V) \) and \( \bar{x} \) and \( \bar{y} \) in \( V \).

1) \( \bigcup (p_1 \star \varphi(\text{Sol}_V) p_2) = (\bigcup p_1) \star \text{Down}_V (\bigcup p_2) \).

2) \( \bigcup (\exists x \varphi(\text{Sol}_V) p) = \exists x \text{Down}_V (\bigcup p) \).

3) \( \bigcup (\delta_{x, \bar{y}}) = \delta_{x, \bar{y}} \text{Down}_V \).
3.4 Type systems and type constraint systems

We have shown that the collecting semantics over existential Herbrand constraints can be abstracted into a semantics over the domain \(\downarrow\). This domain can be used to represent every downward closed property of logic programs. In general, the full power of \(\downarrow\) 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 \(\downarrow\) into a domain for a specific type system, where a type system specifies which downward closed properties we are interested in. Note that from now on the abstraction process becomes type-dependent.

**Definition 3.9.** A type system is a triple \(\langle \Delta, \Sigma, I \rangle\) where \(\Delta\) and \(\Sigma\), called type signature and term signature, respectively, are sets of function symbols with associated arities and \(I(\mathfrak{f}^n)\) is a total map from \((\varphi(\mathfrak{t}(\Sigma, \mathcal{V})))^n\) to \(\varphi(\mathfrak{t}(\Sigma, \mathcal{V}))\) for every \(\mathfrak{f}^n \in \Delta\).

We evaluate elements of \(\mathfrak{t}(\Delta, \emptyset)\) by defining

\[ [\mathfrak{f}^0]I = I(\mathfrak{f}^0), \quad [\mathfrak{f}^n(t_1, \ldots, t_n)]I = I(\mathfrak{f}^n)([t_1]I, \ldots, [t_n]I). \]

We write the arity of functors only when it is not clear from the context.

Consider a term signature \(\Sigma\). We make some examples of type systems.

**Example 3.10 (Type system for groundness).** Let \(G = \langle \{g^0\}, \Sigma, I \rangle\) and let \(I(g) = \mathfrak{t}(\Sigma, \emptyset)\). The functor symbol \(g\) is interpreted as the set of terms which do not contain variables.

**Example 3.11 (Type system for non freeness).** Let \(NF = \langle \{nf^0\}, \Sigma, I \rangle\) where \(I(nf) = \mathfrak{t}(\Sigma, \mathcal{V}) \setminus \mathcal{V}\). The functor symbol \(nf\) is interpreted as the set of terms which are not in \(\mathcal{V}\), i.e., the set of non free terms.

**Example 3.12 (Type system for integers, lists and trees).** Let \(ILT = \langle \{int^0, list^1, tree^1\}, \Sigma, I \rangle\), where \(\Sigma\) contains the set \(\{0^0, s^1, [0^1, [0^1]], void^0, tree^3\}\) and

\[
I(int) = \mu i.\{0\} \cup \{s(n) \mid n \in i\}
\]

\[
I(list) = \lambda \tau, \mu l.\{[\tau] \cup \{[h[t] \mid h \in \tau, t \in l\}\}
\]

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

Note that \(list\) and \(tree\) are unary functors. Therefore, we can write \(list(int)\) and \(tree(list(int))\). The type system \(ILT\) allows to represent integers and polymorphic lists and trees.

Given a type system, we can define a type constraint system. Type constraint systems are formed by transfinite propositional formulas, defined below.
Definition 3.13. Given a type signature $\Delta$ and $V \in \wp(\forall V)$, the set $\Phi_{\Delta,V}$ of transfinite formulas over $\Delta$ and $V$ is defined as the least set such that $x \in t$ is a transfinite formula, where $x \in V$ and $t \in \text{terms}(\Delta, \emptyset)$, if $S$ is a set of transfinite formulas then $\forall S$ and $\exists S$ are transfinite formulas, if $\phi_1$ and $\phi_2$ are transfinite formulas then $\phi_1 \Rightarrow \phi_2$ is a transfinite formula and if $\phi$ is a transfinite formula then $\neg \phi$ is a transfinite formula. We write $\phi_1 \land \phi_2$ for $\forall \{\phi_1, \phi_2\}$, true for $\forall \emptyset$ and false for $\forall \emptyset$. 

Definition 3.14 (Type constraint system). Let $T = \langle \Delta, \Sigma, I \rangle$ be a type system. A type constraint system for $T$ is a constraint system (Definition 2.1) $T = \{T_V\}_{V\in \wp(\forall V)}$ where $T_V \subseteq \Phi_{\Delta,V}$ is closed w.r.t. $\land$ and contains the diagonal elements 

$\exists^T_{x_1, \ldots, x_n, y_1, \ldots, y_n} = \land_{d \in \text{terms}(\Delta, \emptyset), i=1, \ldots, n} \{ (x_i \in d \Rightarrow y_i \in d) \land (y_i \in d \Rightarrow x_i \in d) \}$ 

for every $\langle x_1, \ldots, x_n \rangle$ and $\langle y_1, \ldots, y_n \rangle$ in $V$, and is closed with respect to the operations\(^1\) 

$\phi_1 \times^T_{x} \phi_2 = \phi_1 \land \phi_2$ 

$\exists^T_{x} \phi = \lor \{ \phi[P/x] \mid \text{there exists } t \in \text{terms}(\Sigma, V) \text{ such that } P = \{ d \in \text{terms}(\Delta, \emptyset) \mid t \in \llbracket d \rrbracket_I \} \}$, 

where, for any $P \in \wp(\text{terms}(\Delta, \emptyset))$, 

$(x \in t)[P/x] = \begin{cases} true & \text{if } \cap_{p \in P} \llbracket p \rrbracket_I \subseteq \llbracket t \rrbracket_I \\ false & \text{otherwise} \end{cases}$ 

$(y \in t)[P/x] = (y \in t)$ if $x \neq y$ 

$(\land S)[P/x] = \land \{ s[P/x] \mid s \in S \}$ 

$(\lor S)[P/x] = \lor \{ s[P/x] \mid s \in S \}$ 

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

$(\neg \phi)[P/x] = \neg (\phi[P/x])$.

Note that if $\text{terms}(\Delta, \emptyset)$ 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 constraint systems of Examples 3.10 and 3.11, cylindrification becomes the classical Schröder elimination [3], since $\text{terms}(\Delta, \emptyset)$ contains only a constant symbol.

Definition 3.15. Given a type system $T = \langle \Delta, \Sigma, I \rangle$, we define the map $\llbracket \cdot \rrbracket_T :$ 

\(^{1}\)The operation $\exists^T_{x}$ generalises the Schröder elimination defined as $\exists_x(\phi) = \phi[true/x] \lor \phi[false/x]$ [3, 23].
\[ \Phi_{\Delta, V} \rightarrow (\Theta_V \rightarrow \{0, 1\}) \]

\[ [x \in t]_{T \sigma} = \begin{cases} 1 & \text{if } \sigma(x) \in [t]_I \\ 0 & \text{otherwise} \end{cases} \quad \langle \land S \rangle_{T \sigma} = \begin{cases} 1 & \text{if } [\phi]_{T \sigma} = 1 \\ 0 & \text{for every } \phi \in S \end{cases} \]

\[ [\lor (S)]_{T \sigma} = \begin{cases} 1 & \text{if there exists } s \in S \\ 0 & \text{such that } [s]_{T \sigma} = 1 \end{cases} \quad [\phi_1 \Rightarrow \phi_2]_{T \sigma} = \begin{cases} 1 & \text{if when } [\phi_1]_{T \sigma} = 1 \\ 0 & \text{then } [\phi_2]_{T \sigma} = 1 \end{cases} \]

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

When it is clear from the context, we write \([\;]\) for \([\;]_{T}\). Given \(V \in \wp_f(V)\) 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 is called \((T, V)\)-equivalence. Again, we drop the subscripts \(T\) and \(V\) when it is clear from the context.

From now on, if \(T = \{T_V\}_{V \in \wp_f(V)}\) is a type constraint system for \(T\) then, for each \(V \in \wp_f(V)\), each transfinite formula in \(T_V\) denotes its \((T, V)\)-equivalence class. Note that, for every \(V \in \wp_f(V)\), \(T_V\) is a complete lattice w.r.t. \(\leq_{T, V}\), since by Definition 3.14 it is completely \(\land\)-closed and topped (remember that \(\text{true} = \land \emptyset\)). It has \(\text{true}\) as top, \(\text{false}\) as bottom, \(\lor\) as least upper bound operator and \(\land\) as greatest lower bound operator. Since we consider equivalence classes of transfinite formulas, we must check that the operations \(\star_{T_V}\) and \(\exists_{T_V}\) of Definition 3.14 are independent from the representatives chosen for the \((T, V)\)-equivalence classes. This is obvious for conjunction. In the case of cylindrification, it is true if the following property holds.

\textbf{P1:}\ Let \(T = \{T_V\}_{V \in \wp_f(V)}\) be a type constraint system for the type system \(\langle \Delta, \Sigma, I \rangle\).

For every \(V \in \wp_f(V)\) and \(\{\phi_1, \phi_2\} \subseteq T_V\) such that \(\phi_1 \equiv_{T_V} \phi_2\), we have \(\phi_1[P/x] = \phi_2[P/x]\) for any \(x \in V\) and \(P = \{d \in \text{terms}(\Delta, \emptyset) \mid t \in [d]_I\}\) with \(t \in \text{terms}(\Sigma, V)\).

We want to show now how, for any \(V \in \wp_f(V)\), 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.

\textbf{Definition 3.16.}\ Given a type constraint system \(T = \{T_V\}_{V \in \wp_f(V)}\) for the type system \(\langle \Delta, \Sigma, I \rangle\) and \(\phi \in T_V\), we define

\[ \gamma_{T_V}(\phi) = \{\theta \in \Theta_V \mid \text{for all } \sigma \in \Theta_V \text{ such that } \sigma \leq \theta \text{ we have } [\phi]_{T \sigma} = 1\} \]

Note that \(\gamma_{T_V}(\phi) \in \text{Down}_V\) for any \(\phi \in T_V\). Given \(\phi_1, \phi_2 \in T_V\), we define \(\phi_1 \equiv_{T_V} \phi_2\) if and only if \(\gamma_{T_V}(\phi_1) = \gamma_{T_V}(\phi_2)\). This equivalence relation is called \(\gamma_{T_V}\)-equivalence. When it is clear from the context, we drop the subscript \(T_V\) from \(\gamma_{T_V}\).
It is easy to realise that $\equiv$ entails $\equiv_\gamma$, though the converse does not hold in general, as shown below.

**Example 3.17.** Consider the type system $G$ of Example 3.10 and the type constraint system $\{\Phi_{\Delta, V}\}_{V \in \mathcal{V}(\mathcal{V})}$ for $G$. Consider $V$ such that $\{x, y\} \subseteq V$. Let $\phi_1 = false$ and $\phi_2 = x \in g \Rightarrow false$. Note that there is no $\theta$ such that $[\phi_1]_G \theta = 1$. But any substitution $\theta$ such that $\theta(x)$ is not ground satisfies $[\phi_2]_G \theta = 1$. Hence $\phi_1$ and $\phi_2$ are not $(G, V)$-equivalent. However, $\gamma(\phi_1) = \emptyset$ and $\gamma(\phi_2) = \emptyset$. This is obvious for $\phi_1$. For $\phi_2$, assume $\theta \in \gamma(\phi_2)$. Any instance of $\theta$ must belong to $\gamma(\phi_2)$. But this is a contradiction as soon as we consider an instance $\theta'$ of $\theta$ which makes $x$ ground, because in such a case it would be $\lbrack false \rbrack_\gamma \theta = 1$.

**Proposition 3.18.** Given a type constraint system $T = \{T_V\}_{V \in \mathcal{V}(\mathcal{V})}$ for a type system $T$, $\gamma_T$ is co-additive and $\gamma_T(T_V)$ is a Moore family of $Down_V$ for any $V \in \mathcal{V}(\mathcal{V})$.

Since $\gamma$ is co-additive and $Down_V$ and $T_V$ are complete lattices, we conclude that $\gamma$ is the concretisation map of a Galois connection from $Down_V$ into $T_V$. Let $\alpha$ (i.e., $\alpha_T$) be the corresponding abstraction map. Since, in general, $\gamma$ is not one-to-one (Example 3.17), we conclude that this Galois connection is not always a Galois insertion. However, we do not like useless elements in $T_V$. Therefore, it would be nice if we had a Galois insertion instead of just a Galois connection. This holds if the property below is satisfied.

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

**Definition 3.19.** Given a type system $T = \langle \{c^0\}, \Sigma, I \rangle$ and $V = \{x_1, \ldots, x_n\} \in \mathcal{V}(\mathcal{V})$, $\phi \in \Phi_{\Delta, V}$ is called *positive* if and only if for any ground term $t \in \text{terms}(\Sigma, \emptyset)$ such that $t \in [c]I$, we have $[\phi][x_1 \mapsto t, \ldots, x_n \mapsto t] = 1$.

**Definition 3.20.** A type constraint system $T = \{T_V\}_{V \in \mathcal{V}(\mathcal{V})}$ for the type system $\langle \Delta, \Sigma, I \rangle$ is *positive* if $\Delta$ is formed by just one constant, i.e., $\Delta = \{c^0\}$ for some $c$, and for every $V \in \mathcal{V}(\mathcal{V})$ and every $\phi \in T_V$, $\phi$ is positive (Definition 3.19).

A type constraint system $T = \{T_V\}_{V \in \mathcal{V}(\mathcal{V})}$ for the type system $T = \langle \Delta, \Sigma, I \rangle$ is *structural* if and only if $T$ is structural, i.e., if and only if for every $V \in \mathcal{V}(\mathcal{V})$ and $T \in \mathcal{V}(\text{terms}(\Sigma, V))$ there exists $\sigma \in \Theta_{V, \emptyset}$ such that, for every $t \in T$ and every $d \in \text{terms}(\Delta, \emptyset)$ we have $t \in [d]I$ if and only if $t\sigma \in [d]I$.

The idea underlying the definition of structural type systems is that every finite set of terms can be instantiated into a finite set of ground terms with the same type properties as the original terms. The following proposition shows the importance of positive or structural type constraint systems.

**Proposition 3.21.** Every positive or structural type constraint system satisfies both properties $P1$ and $P2$. 

From the general theory of abstract interpretation (Section 1.6), we know that the operations $\star_{DownV}$ and $\exists_{DownV}$ and the elements $\delta_{DownV}$ induce their optimal counterparts $\star_{TV}$, $\exists_{TV}$ and $\delta_{TV}$ over $TV$. Proposition 3.22 shows how the operations and diagonal elements of Definition 3.14 relate to these optimal induced operations and elements. Moreover, it shows the importance of positive or structural type constraint systems.

**Proposition 3.22.** Let $T = \{TV\}_{V \in \mathcal{V}(\mathcal{V})}$ be a type constraint system for the type system $(\Delta, \Sigma, I)$, satisfying property $P1$.

i) The operator $\star_{TV}$ is always correct w.r.t. $\star_{DownV}$, 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) Given $x$ and $y$ in $V$, $\delta_{TV}^{xy}$ is a correct approximation of $\delta_{DownV}^{xy}$, and it is its best possible approximation if $T$ is positive or structural.

In conclusion, a positive or structural type constraint system $T = \{TV\}_{V \in \mathcal{V}(\mathcal{V})}$ enjoys interesting properties. Namely, cylindrification is well-defined (property $P1$), a Galois insertion can be established between $DownV$ and $TV$ (property $P2$), the operators over $TV$ are the best possible approximations of the corresponding operators over $DownV$ and the diagonal elements of $TV$ are exactly the abstraction of the diagonal elements of $DownV$ (Proposition 3.22).

It is not easy to apply directly the definition of structural type constraint system (Definition 3.20). The proposition below provides a sufficient condition which entails that a type constraint system is structural. Moreover, it shows how large the class of structural type constraint systems is.

**Proposition 3.23.** Let $T = \{TV\}_{V \in \mathcal{V}(\mathcal{V})}$ be a type constraint system for the type system $(\Delta, \Sigma, I)$. 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 $t' \in \|d\|I$, every term obtained from $t'$ by substituting some occurrences of $t$ with variables of $V$ is still in $\|d\|I$. Then $T$ is structural.

As a corollary of the above proposition, we have that any type system defined through the type specification language of [7] is structural, provided the term signature $\Sigma$ contains a symbol $c^0$ which is not used in the rules which define the type system. Indeed, we can use $t = c$ in Proposition 3.23.

### 3.4.1 Other operators

In Section 2.5 we have shown that an elastic semantics needs some more operators than those introduced in Definition 3.14. However, if a downward closed set of
substitutions is contained in the concretisation of a tranfinite formula $\phi$ over a finite set $V$ of variables, then its expansion to a set $V' \supseteq V$ of variables is contained in the concretisation of the same formula over $V'$. This means that we do not need to keep the number of variables small explicitly in the case of type analysis. Therefore, an elastic semantics is useless in this case.

Note that a renaming operation would be useful for avoiding to rename through the use of a diagonal element and a cylindrification operation. We claim that the following operator is the optimal approximation of the concrete operation $\text{rename}^{\eta(H_V)}$.

**Definition 3.24.** Given a type constraint system $T = \{T_V\}_{V \in \mathcal{V}_{f}(\mathcal{V})}$, $V \in \mathcal{V}_{f}(\mathcal{V})$ and $x \notin V$, we define

$$\text{rename}^{T_{V_{x \mapsto n}}} (\phi) = \phi[n/x].$$

### 3.5 Applications

We apply here the theory developed in the previous section to some type systems.

**Example 3.25.** Consider the type system of Example 3.10 and consider the type constraint system $\text{Pos}_{\{g\}} = \{\text{Pos}_{\{g\}, V}\}_{V \in \mathcal{V}_{f}(\mathcal{V})}$ where $\text{Pos}_{\{g\}, V}$ is formed by the set of positive transfinite formulas in $\Phi_{\{g\}, V}$. Note that this set is finite. In [3] it is shown that the set of positive formulas is closed under conjunction and cylindrification. Moreover, it contains the diagonal elements. The type constraint system $\text{Pos}_{\{g\}}$ is well-known [22, 3, 23].

A formula is called definite if the set of its propositional models is closed under instantiation. Consider the type constraint system $\text{Def}_{\{g\}} = \{\text{Def}_{\{g\}, V}\}_{V \in \mathcal{V}_{f}(\mathcal{V})}$ where $\text{Def}_{\{g\}, V}$ is the set of (positive) definite formulas in $\Phi_{\{g\}, V}$. This type constraint system is also well-known [3]. In [3] it is shown that the set of definite formulas is closed under conjunction and cylindrification and contains the diagonal elements.

Since $\text{Pos}_{\{g\}}$ and $\text{Def}_{\{g\}}$ are positive, we conclude that they are related to Down through a Galois insertion and that the operations and diagonal elements of Definition 3.14 are the best possible approximations of the corresponding operations and diagonal elements in Down (Proposition 3.22).

**Example 3.26.** The same construction done above for groundness can be applied to non-freeness (Example 3.11). The resulting type constraint systems will be called $\text{Pos}_{\{nf\}}$ and $\text{Def}_{\{nf\}}$. Note that they are positive type constraint systems.

Since Proposition 3.22 considers every positive type constraint system, we have generalised the result contained in [23] 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 and diagonal elements of Definition 3.14 as done in [16], although in [16], however, no justification of the correctness of this approach is provided.
Example 3.27. Given the type system of Example 3.12, consider the type constraint system $ILT = \{ILT_V\}_{V \in \varphi_\eta(V)}$ where $ILT_V = \Phi_{\Delta_V}$. Since Proposition 3.23 holds with $t = s([])$, we conclude that $ILT$ is structural. Therefore, we know that it is related to $Down$ through a Galois insertion and that the operators and diagonal elements of Definition 3.14 are optimal with respect to the corresponding operators and diagonal elements of $Down$.

One could wonder why for groundness and non-freeness we consider only positive formulas (Examples 3.25 and 3.26), while we consider the whole set of formulas for $ILT$. This is because the type systems $G$ and $NF$ are not structural. Therefore, if we want to use Proposition 3.22 to obtain the optimality of the operations and diagonal elements, we must consider only positive formulas. Moreover, we know that if property P2 (entailed by the fact that a type constraint system is positive) does not hold, we obtain a Galois connection rather than a Galois insertion. For instance, we do not need those formulas which do not represent computation closed properties of variables. Since, in the case of groundness or non-freeness, every variable can always eventually belong to the type, we do not need formulas like $\neg(x \in g)$, whose concretisation is the empty set. This is not true for the type constraint system $ILT$. Indeed, the formula $\neg(x \in \text{int})$ has a clear meaning as a computation closed property of the variable $x$. Namely, it says that $x$ is not and can never become an integer. This is possible if, for instance, $x$ is bound to the empty list.

Note that $ILT_V$ is not a finite set, although $V$ is finite. This is because polymorphism allows to write arbitrarily complex types. For the same reason, the definition of the cylindrification operator (Definition 3.14) is not an effective algorithm on $ILT_V$. In [58] it is shown how to overcome these problems using a domain of type dependencies with type variables which is an abstract interpretation of our type constraint systems of transfinite formulas. This issue will be discussed in the next chapter.

3.6 Type systems and linear refinement

We have shown that every type constraint system induces an abstract interpretation of $Down$ (Proposition 3.18). In this section, we want to show how a hierarchy of abstract interpretations of $Down$ can be defined starting from a basic one modelling just the type properties of interest. Every domain in this hierarchy will be the linear refinement of the previous one w.r.t. concrete conjunction. This means that our hierarchy will be a chain of domains which induce an approximation of concrete conjunction which is more and more precise as long as we proceed in the refinement.

This section generalises an analogous result about groundness analysis contained in [72].

Definition 3.28. Let $T = (\Delta, \Sigma, I)$ be a type system and $V \in \varphi_\eta(V)$. We define

$$v_t = \{\theta \in \Theta_V \mid \theta(v) \in \llbracket t \rrbracket I\}$$
for any \( v \in V \) and \( t \in \text{terms}(\Delta, \emptyset) \), and

\[
\begin{align*}
Basic_{T,V}^i & = \bigwedge \{ v_t \mid v \in V \text{ and } t \in \text{terms}(\Delta, \emptyset) \} \\
Basic_{T,V}^{i+1} & = Basic_{T,V}^i \to^{d_{\text{Down}}} Basic_{T,V}^i \quad \text{for } i \geq 0.
\end{align*}
\]

We show now what is \( -\circ^{d_{\text{Down}}} \) and that \( Basic_{T,V}^{i+1} \) contains \( Basic_{T,V}^i \), which is why in Definition 3.28 we used the simplified form of linear refinement given by Equation (1.4).

**Proposition 3.29.** Let \( V \in \varphi_f(\mathcal{V}) \).

i) If \( \{d_1, d_2\} \subseteq D_{\text{Down}} \) then

\[
d_1 -\circ^{d_{\text{Down}}} d_2 = \{\theta \in \Theta_{T} \mid \text{for all } \sigma \leq \theta \text{ if } \sigma \in d_1 \text{ then } \sigma \in d_2\}.
\]

ii) Given a type system \( T \), \( Basic_{T,V}^i \subseteq Basic_{T,V}^{i+1} \) for any \( i \geq 0 \).

In general, \( Basic_{T,V}^{i+1} \) induces a more precise conjunction operation than \( Basic_{T,V}^i \). However, it is the case that, for a large class of type systems, this chain is finite. Therefore, a best domain exists for the abstract analysis of a given set of types.

**Proposition 3.30.** Let \( T = \langle \Delta, \Sigma, I \rangle \) be a structural type system (Definition 3.20). For any \( V \in \varphi_f(\mathcal{V}) \) we have

i) \( Basic_{T,V}^2 = Basic_{T,V}^i \) for any \( i \geq 2 \).

ii) \( Basic_{T,V}^2 = (Basic_{T,V} \to^{d_{\text{Down}}} Basic_{T,V}) \to^{d_{\text{Down}}} Basic_{T,V} \).

iii) \( Basic_{T,V}^2 \) is condensing in the sense of [73].

Note that we do not know if the analogous version of Proposition 3.30 holds for the type system of a positive type constraint system (Definition 3.20). However, in [72] it is shown that Proposition 3.30 holds for the type system for groundness of Example 3.10, and that \( Basic_{\Delta,V}^2 \) is isomorphic to \( Pos_{\{g\}}(\mathcal{V}) \) (Example 3.25). The proofs of [72] hold for the type system for non freeness too (Example 3.11). However, they cannot be generalised to every type system of a positive type constraint system.

We show now that the domains \( Basic_{T,V}^i, Basic_{T,V}^i \) and \( Basic_{T,V}^2 \) can be represented by suitable type constraint systems.

**Definition 3.31.** Let \( \langle \Delta, \Sigma, I \rangle \) be a type system. Let \( V \in \varphi_f(\mathcal{V}) \). We define

\[
\begin{align*}
\text{And}_{\Delta,V} &= \{ \land S \mid S \subseteq \{ v \in t \mid v \in V \text{ and } t \in \text{terms}(\Delta, \emptyset) \} \} \\
\text{Or}_{\Delta,V} &= \{ \lor S \mid S \subseteq \{ v \in t \mid v \in V \text{ and } t \in \text{terms}(\Delta, \emptyset) \} \text{ and } S \neq \emptyset \} \\
\text{Def}_{\Delta,V} &= \{ A_1 \Rightarrow A_2 \mid \{ A_1, A_2 \} \subseteq \text{And}_{\Delta,V} \} \\
\text{Pos}_{\Delta,V} &= \{ A \Rightarrow O \mid A \in \text{And}_{\Delta,V} \text{ and } O \in \text{Or}_{\Delta,V} \}.
\end{align*}
\]

Moreover, we define the type constraint systems \( \text{Def}_{\Delta} = \{ \text{Def}_{\Delta,V} \}_{V \in \varphi_f(\mathcal{V})} \) and \( \text{Pos}_{\Delta} = \{ \text{Pos}_{\Delta,V} \}_{V \in \varphi_f(\mathcal{V})} \).
Proposition 3.32. Let $\langle \Delta, \Sigma, I \rangle$ be a type system and $V \in \varphi_f(V)$. Letting $\gamma$ denote $\gamma_{\Psi_{\Delta, V}}$, we have

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

ii) $\gamma(\lor(S)) = \bigcup_{\phi \in S} \gamma(\phi)$ if $\lor(S) \in \text{Or}_{\Delta, V}$.

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

Corollary 3.33. Given a type system $\langle \Delta, \Sigma, I \rangle$, structural or such that $\Delta = \{c^0\}$ for some $c$, and $V \in \varphi_f(V)$, we have that $\text{Def}_{\Delta, V}$ is isomorphic to $\text{Basic}^1_{\Delta, V}$ and $\text{Pos}_{\Delta, V}$ is isomorphic to $\text{Basic}^2_{\Delta, V}$.

As a result of the above corollary and of Proposition 3.30, we can say that, for structural type constraint systems, $\text{Pos}_{\Delta}$ can be used to approximate the concrete conjunction in the best possible way (among the abstract domains which do not consider the name of the functors in a term). Remember that, for positive or structural type constraint systems, we know the best possible approximations of conjunction, cylindrification and diagonal elements (Proposition 3.22).

3.6.1 Negative information

Let $T = \langle \Delta, \Sigma, I \rangle$ be a type system. Given $V \in \varphi_f(V)$, $x \in V$ and $d \in \text{terms}(\Delta, \emptyset)$, we have

$$\gamma(x \in d \Rightarrow \text{false}) = \{\theta \in \Theta_V \mid \text{for all } \sigma \leq \theta \text{ we have } [x \in d]\sigma = 0\}.$$  

This means that $\gamma(x \in d \Rightarrow \text{false})$ contains the set of substitutions which map $x$ in a term which is not and can never be instantiated to a term of the type $d$. This is a form of intuitionistic negation. The ability to represent negative information seems a distinguishing feature of our approach. Negative information is extremely useful in practice. In the next chapter we will show an example of analysis where it plays a key role.

3.7 Conclusions

We have defined a large class of type domains that enjoy the same desirable properties of the well-known domain for groundness analysis [22, 3, 23]. This leads to the use of transfinite formulas and operators on transfinite formulas for the type analysis of logic programs. In [58] the authors show how to overcome some limitations of this approach. Namely, they tackle the problem of the finiteness of the analysis. We will give a more detailed presentation of this issue in the next chapter.

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, so that there is one universal condition that includes the positive and structural conditions as special cases.
3.8 Proofs

Proofs of Section 3.2

Proposition 3.34. Given $S \in \varphi \downarrow (\Theta_V)$ and $x \in V$, we have

$$\exists_{x}^{\text{Down}} \, S = \downarrow (S'_{V \setminus x}) ,$$

where

$$S' = \left\{ \theta \{ x \mapsto u \} \in \Theta_V^{[x]} \mid u \in \text{terms}(\Sigma, V) \text{ and } \theta \in S \right\} .$$

Proof. Let $\theta' \in \exists_{x}^{\text{Down}} \, S$. By definition, we have $\theta' = \theta[V, \sigma_{\leq V \cup n} \sigma, \sigma = \sigma'[n/x]$ and $\sigma' \in S$ for suitable $\theta$, $\sigma$ and $\sigma'$. Hence $\theta = \sigma''$ for a suitable $\sigma''$. Since $\sigma''$ is idempotent, $\sigma'(\sigma''|_{V \setminus x})$ is idempotent too and in $\Theta_V$. Moreover, it belongs to $S$ by downward closure and we have:

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

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

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

$$= \sigma'[n/x](y)\sigma'' = \sigma'(y)[n/x]\sigma''$$

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

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

Therefore,

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

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

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

Therefore, $\theta' \in \exists_{x}^{\text{Down}} \, S$.  

Proposition 3.2.

i) Proposition 3.34 shows that it is the downward closure of a set of substitutions.
ii) We have \( S'_1 = \{ \{ x \mapsto u \} \in \Theta^{(x)}_V \mid u \in \text{terms}(\Sigma, V) \text{ and } \theta \in S_1 \} \subseteq \{ \{ x \mapsto u \} \in \Theta^{(x)}_V \mid u \in \text{terms}(\Sigma, V) \text{ and } \theta \in S_2 \} = S'_2. \) Therefore, we have \( \downarrow(S'_1|_{V\setminus x}) \subseteq \downarrow(S'_2|_{V\setminus x}) \) and by Proposition 3.34 we have the thesis.

iii) Let \( S' \) be as in Proposition 3.34. Let \( \theta \in S. \) Choosing \( u = x, \) we have \( \theta \in S'. \) If \( x \notin \text{dom}(\theta) \) then \( \theta|_{V\setminus x} = \theta \) and \( \theta \in \downarrow(S'|_{V\setminus x}). \) If \( x \in \text{dom}(\theta) \) then \( \theta = \theta|_{V\setminus x}\{x \mapsto \theta(x)\}. \) Therefore, even in this case we have \( \theta \in \downarrow(S'|_{V\setminus x}). \) By Proposition 3.34 we have the thesis.

\[ \square \]

**Proposition 3.3.** We assume the constant in \( \Sigma \) to be \( a \) and the functor to be the unary functor \( f, \) though the proof can be generalised easily to greater arities.

The non strict inclusion is a consequence of the fact that \( \text{sol}_V(\exists_W c) \) is a downward closed set of substitutions. The strict inclusion follows from the fact that every set on the left is recursive [27], while some sets on the right are not. Indeed, given a substitution \( \theta \in \Theta_V \) we can check easily if it 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 [63] applied to \( c\theta. \) On the contrary, there are downward closed sets of substitutions which are not recursive. Indeed, given a Turing machine \( M, \) seen as a partial map \( M : \mathbb{N} \rightarrow \mathbb{N}, \) such that \( M(i) \) is defined if and only if the machine \( M \) terminates on input \( i, \) yielding the result \( M(i), \) we can define the downward closed set of substitutions:

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

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

\[ \square \]

**Proposition 3.4.** Assume \( h_1 = \exists_W c_1 \) and \( h_2 = \exists_W c_2 \) with \( W_1 \cap W_2 = \emptyset. \)

Let \( \theta' \in \text{sol}_V(h_1) \ast^{hv}_V \text{sol}_V(h_2). \) Then \( \theta' = \theta|_V, \theta \in \Theta_{V\cup W_1\cup W_2}\) and \( c_1 \theta \) and \( c_2 \theta \) are true. Hence, we have \( \theta|_{V\cup W_1} \in \Theta_{V\cup W_1\cup W_2}\) and \( c_1 \theta|_{V\cup W_1} \) is true for \( i = 1, 2. \) Therefore, \( \theta' = (\theta|_{V\cup W_1})|_V \) belongs to \( \text{sol}_V(h_i) \) for \( i = 1, 2. \) This means that \( \theta' \in \text{sol}_V(h_1) \ast  \text{sol}_V(h_2) = \text{sol}_V(h_1) \ast^{down}_V \text{sol}_V(h_2). \)

Conversely, let \( \theta' \in \text{sol}_V(h_1) \ast^{down}_V \text{sol}_V(h_2) = \text{sol}_V(h_1) \cap \text{sol}_V(h_2). \) Hence there exist \( \theta_1 \in \Theta_{V\cup W_1}\) and \( \theta_2 \in \Theta_{V\cup W_2}\) such that \( \theta_i|_V = \theta' \) and \( c_i \theta_1 \) 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 \( \theta = \theta' \theta_1 \theta_2 \) which is such that \( \theta \in \Theta_{V\cup W_1\cup W_2}\) and \( c_i \theta \) is true for \( i = 1, 2. \) Therefore, \( \theta = \theta|_V \in \text{sol}_V(h_1 \ast^{hv}_V h_2). \)

\[ \square \]

**Proposition 3.4.** We have to prove that

\[ \exists_x^{down} \text{sol}_V(\exists_W c) = \text{sol}_V(\exists_W c[V/N]) \]
for every \( x \in V \).

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

\[
\theta'' = \theta'|_V = (\sigma|_V = (\sigma''[n/x])|_V \in \text{sol}_V(\exists_W \cup c|N/x]) \).

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

\[
\theta = (c[N/x]|_V \cup \text{dom}(c) \cap (\text{dom}(\theta) \cup x) \subseteq \{x\}) \text{ and } \text{rng}(\theta') \cap (\text{dom}(\theta) \cup x) \subseteq \{x\}.
\]

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

\[
\text{dom}(\theta'') \cap \text{rng}(\theta'') \subseteq (\text{dom}(c) \cap V) \cup x \cap (\text{rng}(c) \cup \text{rng}('') \subseteq \{x\}.
\]

The thesis follows by Proposition 3.34. \( \square \)

**Proposition 3.4.iii.** Given \( \langle x_1, \ldots, x_n \rangle \) and \( \langle y_1, \ldots, y_n \rangle \) in \( V \), we have to prove that \( \delta_{\Downarrow_V} = \text{sol}_V(\delta_{\Downarrow_V}^{H_V}) \). We have \( \theta \in \delta_{\Downarrow_V}^{H_V} \) if and only if \( \theta(x_i) = \theta(y_i) \) for every \( i = 1, \ldots, n \), if and only if \( (x_i = y_i) \theta \) is true for every \( i = 1, \ldots, n \), which is true if and only if \( \theta \) belongs to \( \text{sol}_V(\delta_{\Downarrow_V}^{H_V}) \). \( \square \)

**Proofs of Section 3.3**

**Proposition 3.6.** Since \( \theta \in \text{sol}_V(\text{Eq}(\theta)) \), by downward closeness of \( \text{sol}_V(\text{Eq}(\theta)) \) we conclude that \( \text{sol}_V(\text{Eq}(\theta)) \subseteq V \). Conversely, assume \( \sigma \in \text{sol}_V(\text{Eq}(\theta)) \). This means that \( \text{Eq}(\theta)|_V \sigma \) is true, i.e., that \( \sigma(x) = \theta(x)|_V \sigma \) for every \( x \in V \). This entails that \( \sigma = \theta \sigma \), i.e., \( \sigma \leq \theta \). Therefore, we conclude that \( \sigma \in V \), and the converse inclusion is proved. \( \square \)

**Proposition 3.7.** Since \( \varphi(\text{Sol}_V) \) and \( \Downarrow_V \) are set-theoretical lattices, union is their least upper bound operator. Therefore,

\[
\cup (\text{Sol}_V \cup \{p_i \mid i \in I\}) = \cup (\cup p_i \mid i \in I) = \text{Sol}_V \Downarrow \cup p_k.
\]

\( \square \)
Proofs of Section 3.4

We rewrite the cylindrification operator of Definition 3.14 in a way which simplifies the following proofs.

Definition 3.35. Given \( t \in \text{terms}(\Sigma, V) \), we define

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

\[
(y \in d)[t/x] = (y \in d) \\
(\land S)[t/x] = \land \{s[t/x] \mid s \in S\} \\
(\lor S)[t/x] = \lor \{s[t/x] \mid s \in S\} \\
(\phi_1 \Rightarrow \phi_2)[t/x] = (\phi_1[t/x]) \Rightarrow (\phi_2[t/x]) \\
(\neg \phi)[t/x] = \neg(\phi[t/x])
\]

Proposition 3.36. Given a type constraint system \( T = \{ T_V \}_{V \in \psi_f(V)} \) for a type system \( \langle \Delta, \Sigma, I \rangle \), \( V \in \psi_f(V) \) and \( x \in V \),

\[
\exists x^{T_V} \phi = \lor \{ \phi[t/x] \mid t \in \text{terms}(\Sigma, V) \}
\]

for any \( \phi \in T_V \).

Proof. For any \( t \in \text{terms}(\Sigma, V) \), let \( P_t = \{ d \in \text{terms}(\Delta, \emptyset) \mid t \in [d]I \} \). We have \( \phi[t/x] \equiv \phi[P/x] \), which entails the thesis.
Proposition 3.18. For the co-additivity of \( \gamma \), let \( S \subseteq T_V \). We have
\[
\gamma(\wedge S) = \{ \theta \in \Theta_V \mid \text{for all } \sigma \leq \theta \text{ we have } [\phi][\sigma] = 1 \text{ for any } \phi \in S \} = \bigcap_{\phi \in S} \{ \theta \in \Theta_V \mid \text{for all } \sigma \leq \theta \text{ we have } [\phi][\sigma] = 1 \} = \bigcap_{\phi \in S} \gamma(\phi) .
\]
Moreover, since \( \wedge \emptyset \in T_V \) and \( \gamma(\wedge \emptyset) = \Theta_V \), which is the top of \( \text{Down}_V \), we conclude that \( \gamma(T_V) \) is topped. Finally, since for any \( S \subseteq T_V \) we know that \( \wedge S \in T_V \), the above result about co-additivity entails that the set \( \gamma(T_V) \) is completely \( \cap \)-closed.

Definition 3.37. Given a type constraint system \( T = \{ T_V \}_{V \in \Phi(\nu)} \) for the type system \( \langle \Delta, \Sigma, I \rangle \) and \( \{ \theta_1, \theta_2 \} \subseteq \Theta_V \), \( \theta_1 \) and \( \theta_2 \) are type-equivalent if and only if for every \( v \in V \) and every \( d \in \text{terms}(\Delta, \Sigma, I) \) we have \( \theta_1(v) \in [d]I \) if and only if \( \theta_2(v) \in [d]I \).

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

Proposition 3.38. Given a type constraint system \( T = \{ T_V \}_{V \in \Phi(\nu)} \) for the type system \( \langle \Delta, \Sigma, I \rangle \), two type-equivalent substitutions \( \theta_1 \) and \( \theta_2 \) in \( \Theta_V \) and \( \phi \in \Phi_{\Delta[V]} \), we have \([\phi]_{\theta_1} = [\phi]_{\theta_2}\).

Proof. By simple induction on the structure of transfinite formulas. \( \square \)

Proposition 3.21. Let \( T = \{ T_V \}_{V \in \Phi(\nu)} \) be a type constraint system for the type system \( \langle \Delta, \Sigma, I \rangle \). Consider property P1 first. Let \( \phi_1, \phi_2 \in T_V \). Assume by contradiction that \( \phi_1 \equiv \phi_2 \) but \( \phi_1[t/x] \not\equiv \phi_2[t/x] \). Then we can assume without any loss of generality that there exists a substitution \( \theta \) such that \([\phi_1[t/x]]\theta = 1 \) and \([\phi_2[t/x]]\theta = 0 \).

If \( T \) is positive, let \( \Delta = \{ c \} \) be its only type. If \([c]I \neq \emptyset \), let \( t' \) be a ground term in \([c]I \) (otherwise, we do not need such a term). Define \( \theta' \) such that
\[
\theta'(x) = \begin{cases} 
  t' & \text{if } t \in [c]I \\
  x & \text{otherwise}
\end{cases}
\]
\[
\theta'(v) = \begin{cases} 
  t' & \text{if } \theta(v) \in [c]I \\
  v & \text{otherwise}
\end{cases}
\]
for every \( v \in V \setminus x \). The substitutions \( \theta|_{V \setminus x} \) and \( \theta'|_{V \setminus x} \) are type-equivalent by construction. We have
\[
1 = [\phi_1[t/x]]\theta = [\phi_1[t/x]]\theta'.
\]
(by the choice of \( t' \))
\[
[\phi_1]_{\theta'} = [\phi_2]_{\theta'}
\]
(since \( \phi_1 \equiv \phi_2 \))
\[
[\phi_2[t/x]]\theta = 0 ,
\]
which is a contradiction.

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

Consider now property P2. Given $\phi_1, \phi_2 \in T_V$ such that $\phi_1 \neq \phi_2$, we would like to show that $\gamma(\phi_1) \neq \gamma(\phi_2)$. Let $\theta$ be such that $[\phi_1]\theta = 1$ and $[\phi_2]\theta = 0$ (this is possible since $\phi_1 \neq \phi_2$, 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 $[\phi_2]\theta' = [\phi_2]\theta = 0$ entails $\theta' \notin \gamma(\phi_2)$, this will entail the thesis.

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

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

The substitutions $\theta$ and $\theta'$ are type-equivalent by construction. Moreover, every instance of $\theta'$ or is type-equivalent to $\theta'$ or puts every variable into the type $c$. Since $\phi_1$ is positive, we have $\theta' \in \gamma(\phi_1)$, as required.

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

**Proposition 3.22.i.** We have to show that

$$
\alpha \left( \gamma(\phi_1) \downarrow_{V} \gamma(\phi_2) \right) \leq \phi_1 \downarrow_{T_V} \phi_2
$$

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

$$
\alpha \left( \gamma(\phi_1) \downarrow_{V} \gamma(\phi_2) \right) = \alpha \left( \gamma(\phi_1) \cap \gamma(\phi_2) \right) \\
\text{(Proposition 3.18)} = \alpha(\gamma(\phi_1 \land \phi_2)) \\
\text{(\alpha \gamma is reductive)} \leq \phi_1 \land \phi_2 = \phi_1 \downarrow_{T_V} \phi_2.
$$

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

**Proposition 3.22.ii.** For the result about correctness it suffices to show that

$$
\exists_{\phi}\downarrow_{V} \left( \gamma(\phi) \right) \leq \gamma \left( \exists^T_{\phi} (\phi) \right),
$$

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 $\gamma$.

Let $\theta \in \exists_{\phi}\downarrow_{V} \left( \gamma(\phi) \right)$. Then there exists $\theta' \in \gamma(\phi)$ such that $\theta = (\theta'|_{V \setminus \{x \mapsto u\}})\sigma$, for suitable $\sigma \in \Theta^V$ and $u \in \text{terms}(\Sigma, V)$ (Proposition 3.34). Then

$$
\theta|_{V \setminus \{x \mapsto u\}} = ((\theta'|_{V \setminus \{x \mapsto u\}})\sigma)|_{V \setminus \{x \mapsto u\}} = ((\theta'|_{\{x \mapsto u\}})\sigma)|_{V \setminus \{x \mapsto u\}}.
$$
Let $\theta'' = (\theta'\{x \mapsto u\})\sigma$. We have $\theta'' \leq \theta'$. Then $[\phi]\theta'' = 1$, which entails that $[\phi[\theta''(x)/x]](\theta'\sigma)|_{V\setminus x} = 1$. This means that $[\exists^T_x \phi]\theta = 1$, because $\theta|_{V\setminus x} = \theta''|_{V\setminus x}$. Since this is true for every $\theta \in \exists^T_x \phi(\gamma(\phi))$ and $\exists^T_x \phi(\gamma(\phi))$ is downward closed (Proposition 3.2.i), we have the thesis.

Let $T$ be positive or structural. Assume by contradiction that $\alpha(\exists^T_x \phi) < \exists^T_x \phi$. Then there exists $\theta$ such that $[\alpha(\exists^T_x \phi)]\theta = 0$ and $[\exists^T_x \phi]\theta = 1$. We show that there exists a substitution $\theta'$ type-equivalent to $\theta$ such that $\theta' \in \exists^T_x \phi(\gamma(\phi))$. By extensivity (Proposition 3.2.iii), it follows that $\theta' \in \gamma(\exists^T_x \phi(\gamma(\phi)))$, i.e., $[\alpha(\exists^T_x \phi(\gamma(\phi)))]\theta' = 1$, which is a contradiction because $\theta$ and $\theta'$ are type-equivalent and we know that $[\alpha(\exists^T_x \phi(\gamma(\phi)))]\theta = 0$.

If $T$ is positive, then let $e_0$ be its only type. Let $t$ be a ground term in $[c]I$, if $[c]I$ is not empty (otherwise, we do not need such a $t$). Let $z \in V$ arbitrary. Let us define

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

for every $v \in V$. The substitutions $\theta$ and $\theta'$ are type-equivalent by construction. Then $[\exists^T_x \phi]\theta' = 1$. Therefore, there exists a term $t'$ such that $[\phi[t'/x]]\theta' = 1$, i.e., $[\phi](\theta'|_{V\setminus x}\{x \mapsto t''\}) = 1$, where $t'' = \begin{cases} t & \text{if } t' \in [c]I \\ z & \text{otherwise}. \end{cases}$

Let $\theta'' = \theta'|_{V\setminus x}\{x \mapsto t''\}$. Every instance of $\theta''$ whether is type-equivalent to $\theta''$ or puts every variable inside the type $c$. Since $\phi$ is positive, we have $\theta'' \in \gamma(\phi)$. Then $\theta' \in \exists^T_x \phi(\gamma(\phi))$ (Proposition 3.34).

If $T$ is structural, we know that there exists a grounding substitution $\theta'$ which is type-equivalent to $\theta$. Then $[\exists^T_x \phi]\theta' = 1$. Then there exists a term $t'$ such that $[\phi[t'/x]]\theta' = 1$, i.e., $[\phi](\theta'|_{V\setminus x}\{x \mapsto t''\}) = 1$, where $t''$ is a ground instance of $t'$ with the same type properties as $t'$ (we can find such a $t''$ since $T$ is structural). Let $\theta'' = \theta'|_{V\setminus x}\{x \mapsto t''\}$. It is grounding for $V$. Therefore, every instance of $\theta''$ is $\theta''$ itself and $\theta'' \in \gamma(\phi)$. Then $\theta' \in \exists^T_x \phi(\gamma(\phi))$ (Proposition 3.34).

**Proposition 3.22.iii.** Let $\theta \in \delta^T_{x,y}$, with $x = \langle x_1, \ldots, x_n \rangle$ and $y = \langle y_1, \ldots, y_n \rangle$ for some $n \geq 1$. We have $\theta(x_i) = \theta(y_i)$ for every $i = 1, \ldots, n$. Therefore, every $\sigma \leq \theta$ is such that $\sigma(x_i) = \sigma(y_i)$ for every $i = 1, \ldots, n$. This means that $\theta \in \gamma(\delta^T_{x,y})$, and $\delta^T_{x,y} \geq \alpha(\delta^T_{x,y})$.

Assume now that $T$ is positive or structural. If, by contradiction, the strict inclusion holds, there exists a substitution $\theta$ such that $[\delta^T_{x,y}]\theta = 1$ but $[\alpha(\delta^T_{x,y})]\theta = 0$. Since $[\delta^T_{x,y}]\theta = 1$, we conclude that for every $i = 1, \ldots, n$ and every $d \in \text{terms}(\Delta, \emptyset)$, if $\theta(x_i) \in [d]I$, then $\theta(y_i) \in [d]I$ and vice versa.
Assume $\Delta = \{d\}$. If $[d]I \neq \emptyset$, then let $t$ be a ground term such that $t \in [d]I$. Let us define the idempotent substitution

$$
\theta'(x_i) = \begin{cases} 
  t & \text{if } \theta(x_i) \in [d]I \\
  \theta(y_i) & \text{otherwise}
\end{cases}
$$

$$
\theta'(y_i) = \begin{cases} 
  t & \text{if } \theta(y_i) \in [d]I \\
  \theta(y_i) & \text{otherwise}
\end{cases}
$$

$$
\theta'(x) = \theta(x) \quad \text{if } x \notin \{x_1, \ldots, x_n, y_1, \ldots, y_n\}.
$$

By definition, $\theta$ and $\theta'$ are type-equivalent. By Proposition 3.38, this entails that $[\alpha(\delta_{x,y}^{\text{Down}_V})]\theta' = 0$, i.e., $\theta' \not\in \gamma(\delta_{x,y}^{\text{Down}_V})$. But $\theta'(x_i) = \theta'(y_i)$ for every $i = 1, \ldots, n$. Then $\theta' \in \delta_{x,y}^{\text{Down}_V}$. This is a contradiction since $\delta_{x,y}^{\text{Down}_V} \subseteq \gamma(\delta_{x,y}^{\text{Down}_V})$ (Proposition 3.2.iii).

If the type system contains more than one type, it must be structural. Therefore, we can define a substitution $\theta'$ grounding for $V$ which is type-equivalent to $\theta$ an such that $\theta'(x_i) = \theta'(y_i)$ for every $i = 1, \ldots, n$. The thesis follows as in the case above. \(\square\)

**Proposition 3.2.3.** Given $V \in \psi(I)\{V\}$ and $\{t_1, \ldots, t_n\} \subseteq \text{terms}(\Sigma, V)$, consider $\sigma \in \Theta^V$ such that $\sigma(v) = t$ for any $v \in V$. For any $i = 1, \ldots, n$ and $d \in \text{terms}(\Delta, \emptyset)$, if $t_i \in [d]I$ then $t_i\sigma \in [d]I$ by downward closedness of types. Conversely, if $t_i\sigma \in [d]I$ we conclude that $t_i \in [d]I$ by the choice of $t$, since $t_i$ is obtained from $t_i\sigma$ by substituting some instances of $t$ with variables in $V \subseteq V$. Therefore, $T$ is structural. \(\square\)

## Proofs of Section 3.6

**Proposition 3.29.**

i)

$$
d_1 \rightarrow_{\text{Down}_V} d_2 = \bigcup \{d \in \text{Down}_V \mid d_1 \cap d \subseteq d_2\}
$$

$$
= \{\theta \in \Theta^V \mid \text{for all } \sigma \leq \theta \text{ if } \sigma \in d_1 \text{ then } \sigma \in d_2\}.
$$

ii) For every $i \geq 0$ we have $\Theta^V \in \text{Basic}^i_{T,V}$ since $\text{Basic}^i_{T,V}$ is a Moore family of $\text{Down}_V$. Therefore, we have $\Theta^V \rightarrow d = d$ by point i and since $d$ is downward closed. This entails that every $d \in \text{Basic}^i_{T,V}$ is contained in $\text{Basic}^{i+1}_{T,V}$.

\(\square\)

**Proposition 3.30.** In [72] it is shown that points i, ii and iii hold if, letting $\{b_i\}_{i \in I}$, $\{c_j\}_{j \in J}$ and $\{d_k\}_{k \in K}$ be in $\{v_t \mid v \in V \text{ and } t \in \text{terms}(\Delta, \emptyset)\}$, with $I, J, K \subseteq \mathbb{N}$ and letting $\theta \in \Theta^V$ be such that $\theta \not\in B \cup C \cup D$, we have $\theta \not\in (B \rightarrow C) \rightarrow D$. By Definition 3.20 we know that there exists $\sigma \in \Theta^V$ such that $\theta\sigma$ is grounding for $V$ and $\theta$ is type-equivalent to $\theta\sigma$ (Definition 3.37). Therefore, every $\sigma' \leq \theta\sigma$ is such that...
\( \sigma' = \theta \sigma \) and \( \sigma' \not\in B \cup D \). We conclude that \( \theta \sigma \in (B \rightarrow C) \) and \( \theta \sigma \not\in D \). This means that \( \theta \sigma \not\in (B \rightarrow C) \rightarrow D \), i.e., \( \theta \not\in (B \rightarrow C) \rightarrow D \) since \( (B \rightarrow C) \rightarrow D \) is downward closed. By using the above mentioned result of [72], we have the thesis. \( \square \)

**Proposition 3.32.**

i) \[
\gamma(\nu \in t) = \{ \theta \in \Theta_V \mid \text{for all } \sigma \leq \theta \text{ we have } \sigma(x) \in [\nu]I \} \\
( [\nu]I \text{ is downward closed}) = \{ \theta \in \Theta_V \mid \theta(x) \in [\nu]I \} = \nu_t .
\]

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

\[
\gamma(\forall(S)) = \left\{ \theta \in \Theta_V \mid \text{for all } \sigma \leq \theta \text{ there exists } \phi \in S \text{ such that } [\phi] \sigma = 1 \right\} \\
(\text{since } \phi = x \in t) = \left\{ \theta \in \Theta_V \mid \text{there exists } \phi \in S \text{ such that } [\phi] \theta = 1 \right\} = \bigcup_{\phi \in S} \{ \theta \in \Theta_V \mid [\phi] \theta = 1 \} \\
(\text{since } \phi = x \in t) = \bigcup_{\phi \in S} \{ \theta \in \Theta_V \mid \text{for all } \sigma \leq \theta \text{ we have } [\phi] \sigma = 1 \} = \bigcup_{\phi \in S} \gamma(\phi) .
\]

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 \} \\
(\text{Proposition 3.29.ii}) = \{ \theta \in \Theta_V \mid [A] \theta = 1 \} \rightarrow \{ \theta \in \Theta_V \mid [O] \theta = 1 \} \\
\left( A \in And_{\Delta,V} \text{ and } O \in Or_{\Delta,V} \right) = \{ \theta \in \Theta_V \mid \text{for all } \sigma \leq \theta \text{ we have } [A] \sigma = 1 \} \rightarrow \\
\{ \theta \in \Theta_V \mid \text{for all } \sigma \leq \theta \text{ we have } [O] \sigma = 1 \} = \gamma(A) \rightarrow \gamma(O) \).
\]

\( \square \)

**Corollary 3.33.** Proposition 3.32 and the co-additivity of \( \gamma \) (Proposition 3.18) entail that \( Basic^1_{\Delta,V} = \gamma(Def_{\Delta,V}) \) and \( Basic^2_{\Delta,V} = \gamma(Pos_{\Delta,V}) \). Moreover, we know that for positive or structural type constraint systems \( \equiv^\gamma \)-equivalence is the same as \( \equiv^\alpha \)-equivalence (Proposition 3.21). We conclude that \( \gamma \) is one-to-one, which entails the thesis. \( \square \)
In this chapter we provide a representation of the $Def_\Delta$ type constraint system, based on finite formulas with type variables, i.e., pure logic programs. This constraint system of pure logic programs allows us to obtain an effective type analysis. Moreover, it makes clear where the abstraction process loses precision to gain effectiveness. We show the application of our domain of logic programs to the type analysis of the classical append/3 procedure, in order to expose all the details of the analysis. Finally, we show the application of a prototypical analyser we have developed to the type analysis of a non-trivial procedure.

Part of this chapter has been published in [58].

4.1 Introduction

Type analysis for untyped logic programs is useful both to the programmer (for debugging and verification) and to the compiler (for code optimisation).

The main contribution of this chapter is to define a representation for the $Def_\Delta$ type constraint system of Definition 3.31. In this way, we base our construction over concrete domains which are defined through the formal methodology of domain refinement (Section 3.6) and are the generalisation of the well-known domain $Def$ for groundness analysis. This approach allows us to split the clean theoretical construction of type constraint systems from the subsequent abstraction into finite formulas with type variables, i.e., pure logic programs, where practical considerations constrain to lose precision to gain the finiteness of the analysis.

The use of logic programs as abstract domains for the analysis of logic programs is not new (see, for example, [33]). However, another contribution of this chapter is to justify their use as the logical consequence of the use of linear refinement.
This chapter is organised as follows. In Section 4.2 we discuss related works. Section 4.3 introduces the constraint system $\text{Prog}^k$ of logic programs as a finite representation of $\text{Def}_{\Delta}$. Section 4.4 shows how to approximate the abstraction function through an algorithmic definition. Section 4.5 shows how information can be extracted from the elements of $\text{Prog}^k$. Section 4.6 shows some examples of analysis. Section 4.7 draws some conclusions and describes some open problems.

### 4.2 Related works

It is hard to compare the various techniques of type analysis in terms of precision, efficiency and generality, because they use different methods and are often based on different assumptions.

Some type inference techniques are similar to those developed for (higher order) functional languages (see, for example, [53, 70, 4, 84]) while others are inspired by program verification methods [2]. Others use type graphs [81, 51]. Finally, there are plenty of type analysis techniques based on abstract interpretation [7, 16, 17, 51, 52, 61, 74, 83, 84].

The basic step in every abstract interpretation approach to type analysis is the choice of the abstract domain, which defines how we assign types to terms. A ground type language does not allow one to handle type dependencies [10]. This is the case of [52, 83]. Some type dependencies among different arguments of a procedure can be expressed by using type variables in the type language. This is a standard solution, used for instance in [7, 51, 84, 17]. The same solution is used in the framework of regular approximations of the success set in [35]. However, the use of type variables does not allow one to express all type dependencies between argument positions. Only in [7, 16] it is shown an example of analysis which explicitly expresses type dependencies. The domains defined in [7] use ground types. There is a notion of well-typed programs which limits the applicability of their approach. Since type variables are not used, a practical use of their domains is impossible. In [16] the authors generalise the construction of the domain $\text{Pos}_{\{g\}}$ for groundness analysis to generic types. As we said in the introduction of Chapter 3, the correctness of their approach was not justified by a formal proof. Moreover, the use of type variables seems more a consequence of a specific implementation design than of an underlying theory.

In general, we do not know of any approach where the abstract domains are developed in an automatic way, and choices about the representation and the algorithms are implied by the same theory of abstract interpretation. Instead, this is a distinguishing feature of our abstract domains for type analysis. As a consequence, we are able to recognise where we lose precision to gain effectivity. Moreover, we can set the desired level of precision of the analysis. Other distinguishing features of our approach are the exploitation of negative information, the use of explicit polymorphic type dependencies, the use of logic programs as abstract domains for
4.3 Logic programs as finite representations of type domains

Transfinite formulas can be used as computational domain if the set of types is finite. In such a case every transfinite formula is isomorphic to a finite formula and the definitions of the abstract operators given in Definition 3.14 are algorithms. Even the equivalence test between two formulas becomes effective, though very expensive, being a classical NP-complete problem [69]. However, a finite set of types is useful for mode analysis (groundness, nonfreeness) but does not allow us to make interesting type analyses involving polymorphic types. When \( \text{terms}(\Sigma, \emptyset) \) is infinite, transfinite formulas are not finitely representable. However, the full power of transfinite formulas is seldom useful. In many cases, a finite formula with type variables can represent the same information. For instance, assuming that we have a polymorphic type \( \text{list} \), we must use the infinite conjunction

\[
\land_{d \in \text{terms}(\Sigma, \emptyset)} (x \in \text{list}(d) \iff (h \in d \land t \in \text{list}(d)))
\]

in order to express the relationship between the variables of the constraint \( \{x = h[t]\} \). By using type variables, the same information can be expressed by the finite formula \( x \in \text{list}(T) \iff (h \in T \land t \in \text{list}(T)) \). Note that this formula can be written as the logic program

\[
\begin{align*}
\text{x(list(T))} & \leftarrow \text{h(T)}, \text{t(list(T))} \\
\text{h(T)} & \leftarrow \text{x(list(T))} \\
\text{t(list(T))} & \leftarrow \text{x(list(T))}
\end{align*}
\]

In this program, the variables of interest \( x, h \) and \( t \) in the example) become constants in the language of the program, while the type variables are the real variables of the program.

In this section, we define a constraint system of logic programs which can be used as a finite representation of transfinite formulas.

In the following, we assume there is an infinite set \( T \) of type variables, denoted by uppercase letters.

A type model provides a set of types for every variable.

**Definition 4.1.** Given a type system \( T = \langle \Delta, \Sigma, I \rangle \), a type model for \( T \) is a map \( M : \mathcal{V} \mapsto \{\{d \in \text{terms}(\Delta, \emptyset) \mid t \in [d][I] \text{ with } t \in \text{terms}(\Sigma, \mathcal{V})\}\}. \)

**Definition 4.2 (The \( \text{Prog}^k \) constraint system).** Let \( T = \langle \Delta, \Sigma, I \rangle \) be a type system, \( V \in \varphi_f(\mathcal{V}) \) and \( k \geq 1 \). A \( k \)-atom for \( V \) and \( \Delta \) is \( \nu(t) \), where \( \nu \in V \)
and \( t \in \text{terms}^k(\Delta, T) \). An element of \( \text{Prog}^k_V \) is a (possibly empty or infinite) set of clauses \( H \leftarrow T \) where \( H \) is a \( k \)-atom for \( V \) and \( \Delta \) and \( T \) is a (possibly empty) set of comma separated \( k \)-atoms for \( V \) and \( \Delta \). A ground instance of a clause \( H \leftarrow T \) is a clause obtained from \( H \leftarrow T \) by substituting every type variable with some element of \( \text{terms}(\Delta, \emptyset) \). We define the constraint system \( \text{Prog}^k_V = \{ \text{Prog}^k_V \}_{V \in \mathcal{V}(\nu)} \) with the operations

\[
P_1 \ast_{\text{Prog}^k_V} P_2 = P_1 \cup P_2
\]

\[
\text{rename}_{x \rightarrow n} P = P[n/x] \quad \text{if } x \in V \text{ and } n \not\in V
\]

\[
\exists_x \text{Prog}^k_V P = (P \cup P') \cap \text{Prog}^k_V \setminus x,
\]

where \( P' \) is the set of clauses which can be obtained by unfolding the clauses in \( P \) of the form \( x(t) \leftarrow B \) in the body of the other clauses\(^1\). Given \( \langle x_1, \ldots, x_n \rangle \) and \( \langle y_1, \ldots, y_n \rangle \) in \( V \), we define the diagonal elements

\[
\delta_{\text{Prog}^k_V}^{(x_1, \ldots, x_n), (y_1, \ldots, y_n)} = \{ x_i(T) \leftarrow y_i(T) \mid i = 1, \ldots, n \} \cup \{ y_i(T) \leftarrow x_i(T) \mid i = 1, \ldots, n \}.
\]

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

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

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

\[
M \models H \leftarrow T \text{ iff } M \models T \text{ entails } M \models H \text{ when } H \leftarrow T \text{ is ground}
\]

\[
M \models H \leftarrow T \text{ iff } M \models H' \leftarrow T' \text{ for every ground instance } H' \leftarrow T' \text{ of } H \leftarrow T,
\]

\[
\quad \text{when } H \leftarrow T \text{ is not ground}
\]

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

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

For any \( V \in \mathcal{V}(\nu) \), the programs in \( \text{Prog}^k_V \) are partially ordered as \( P_1 \leq_T P_2 \) 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 \( P_1 \leq_T P_2 \) and \( P_2 \leq_T P_1 \). The quotient of the set \( \text{Prog}^k_V \) 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., \( \ast_{\text{Prog}^k_V} \)). From now on, every program will stand for its equivalence class.

**Example 4.3.** Figure 4.1 shows a program together with its restrictions w.r.t. \( z \) and \( x \). Note that in \( \exists_x \text{Prog}^k_V P \) we have dropped a tautological clause for \( y(T) \), since we deal with equivalence classes of constraints.

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

\(^1\)This unfolding could be applied more than once, for achieving a better precision. However, practical experiments have shown that one application leads to a sufficiently precise result.
Figure 4.1: A program and two of its cylindrifications.

**Definition 4.4.** Let \( \langle \Delta, \Sigma, I \rangle \) be a type system, \( k \geq 1 \) and \( V \in \wp_f(\mathcal{V}) \). Let \( V_1, \ldots, V_n \) be the type variables in the clause \( H \leftarrow B \) and let \( P \in \text{Prog}^k_V \). We define

\[
v(t) = v \in t \quad B' = \bigwedge_{b \in B} b'
\]

where \( B \) is a set of \( k \)-atoms.

Moreover, we define

\[
\begin{align*}
\gamma^k_H(H \leftarrow B) &= \bigwedge_{t_1, \ldots, t_n \in \text{terms}(\Delta, \emptyset)} H'[t_1/V_1] \cdots [t_n/V_n] \leftarrow B'[t_1/V_1] \cdots [t_n/V_n], \\
\gamma^k_P(P) &= \bigwedge_{c \in P} \gamma^k_c(c).
\end{align*}
\]

When it is clear from the context, we write \( \gamma^k \) for \( \gamma^k_H \).

**Proposition 4.5.** Given a type system \( T \), \( k \geq 1 \) and \( V \in \wp_f(\mathcal{V}) \), \( \gamma^k : \text{Prog}^k_V \rightarrow T_V \) is co-additive.

The operations and diagonal elements of Definition 4.2 are correct or even optimal w.r.t. the corresponding operations and diagonal elements of \( \text{Def}_{\Delta, V} \).

**Proposition 4.6.** Let \( T = \langle \Delta, \Sigma, I \rangle \) be a type system, \( k \geq 1 \) and \( V \in \wp_f(\mathcal{V}) \).

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

ii) The operator \( \text{rename}^{\text{Prog}^k_V} \) is the best possible approximation\(^2\) of \( \text{rename}^{\text{Def}_{\Delta, V}} \).

iii) The operator \( \exists^{\text{Prog}^k_V} \) is correct w.r.t. \( \exists^{\text{Def}_{\Delta, V}} \).

iv) The diagonal elements of \( \text{Prog}^k_V \) are the best possible approximation of the corresponding diagonal elements of \( \text{Def}_{\Delta, V} \).

\(^2\)This is a typical example of how the use of a renaming operation instead of a diagonal element and a cylindrification allows us to attain a better precision (see point iii).
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 4.2 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 boil down to the case of a finite set of types. Indeed, type variables can be bound to terms of arbitrary depth. Therefore, as our concluding examples will show, the restriction on the depth of the terms does not introduce a big loss in precision, thanks to the use of type variables.

The definition of $\exists_x^{\text{Prog}_V^k}$ uses concrete unification between type terms. Since types are partially ordered with respect to subtyping (for instance, $\text{int} \subseteq \text{top}$), the unification procedure used for unfolding might be too coarse. For instance, if we have a clause whose head is $x(\text{list(int)})$ and we try to unfold it in the body of a clause containing $x(\text{list(top)})$, the unification procedure fails. Actually, unfolding 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 cylindricification operator can be improved by using a unification procedure which is aware of subtyping information.

In order to make $\text{Prog}_V^k$ useful for program analysis, we still need to define an algorithm which allows us to abstract a concrete Herbrand constraint into an element of $\text{Prog}_V^k$, and an algorithm which extracts from an element of $\text{Prog}_V^k$ the set of types a variable is bound to.

### 4.4 Abstraction

In this section we define an algorithm which approximates the restriction of the map $\lambda h, \alpha_k (\alpha_{\text{Def}}(\Delta, V) \cup \text{sol}(h)))$ to singleton sets of existential Herbrand constraints.

**Definition 4.7.** Given a type system $T = \langle \Delta, \Sigma, I \rangle$, $k \geq 1$ and $V \in \wp_f(V)$, we assume there is a Prolog procedure $\text{type}/2$ such that, for any $\text{Term} \in \text{terms}(\Sigma, V)$ such that $\text{vars}(\text{Term}) = \{x_1, \ldots, x_n\}$ and $\text{Type} \in \text{terms}(\Delta, T)$, we have that if $\text{type}(\text{Term}, \text{Type})$ yields the computed answer substitution $\theta$ then for every $\sigma \in \Theta_V$ and $\mu : T \rightarrow \text{terms}(\Delta, \emptyset)$ we have $(\text{Term})\sigma \in \text{eval}(\text{Type})\theta \mu$ if and only if $\sigma(x_i) \in \text{eval}(\text{Type})\theta \mu$ for every $i = 1, \ldots, n$.

In words, $\text{type}$ 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.

**Example 4.8.** Figure 4.2 shows an example of such a procedure for the types $\text{top}$, $\text{int}$ and polymorphic $\text{lists}$. A query $\text{type}(\text{[H|T]}, \text{Type})$ yields a computed answer substitution $\{\text{Type} \mapsto \text{list(S)}, \text{H} \mapsto \text{S}, \text{T} \mapsto \text{list(S)}\}$, meaning that the term $\text{[H|T]}$ can be instantiated to a term of type $\text{list(S)}$ if and only if $\text{H}$ is instantiated to a term of type $\text{S}$ and $\text{T}$ to a term of type $\text{list(S)}$, for every instantiation of $\text{S}$. 
4.4. Abstraction

- **meta-clause**
  
  \[
  \text{type}(X, S) : \text{var}(X),! \cdot X = S. \\
  \]

- **the whole universe**
  
  \[
  \text{type}(X, \text{top}). \\
  \]

- **integers:**
  
  \[
  \mu. \lambda = \{0\} \cup \{s(l) \mid l \in \lambda\} \\
  \text{type}(X, \text{int}) : -X = 0. \\
  \text{type}(X, \text{int}) : -X = s(I), \text{type}(I, \text{int}). \\
  \]

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

*Figure 4.2: An example of the procedure type.*

Note that a definition of the type procedure can be derived automatically from the definition of types and that it is 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 [74].

**Definition 4.9.** Given a type system \( T = \langle \Delta, \Sigma, I \rangle \), a procedure type for \( T \), \( k \geq 1 \), \( V \in \mathcal{V} \) and \( c \in C_V \) in normal form, for any \( x \in V \) we define

\[
\alpha^\text{alg}_x(c) = \{x(t^i_x) \leftarrow x_1(t^i_{x_1}), \ldots, x_n(t^i_{x_n}) \mid i = 1, \ldots, m\} \cup \{x_j(t^i_{x_j}) \leftarrow x(t^i_x) \mid j = 1, \ldots, n, i = 1, \ldots, m\},
\]

where \( \text{vars}(c(x)) = \{x_1, \ldots, x_n\} \) and \( \text{type}(c(x), T) \) yields the computed answers \( \{T \mapsto t^i_x, x_1 \mapsto t^i_{x_1}, \ldots, x_n \mapsto t^i_{x_n}\} \) for \( i = 1, \ldots, m \).

We define

\[
\alpha^\text{alg}_V(c) = \bigcup_{x \in V} \alpha^\text{alg}_x(c). 
\]

The following proposition shows that the set of the solution of an existential Hebrand constraint of the form \( \exists_y c \) is correctly approximated by \( \alpha^\text{alg}_V(c) \).

**Proposition 4.10.** Given a type system \( T = \langle \Delta, \Sigma, I \rangle \), a procedure type for \( T \), \( k \geq 1 \), \( V \in \mathcal{V} \) in normal form and \( x \in V \), we have

\[
\downarrow c \subseteq \gamma^\text{Def}_{\Delta, V} \gamma^k \alpha^\text{alg}_V(c). 
\]

The definition of the abstraction map can be improved by using the negative information contained in an existential Herbrand constraint. We do not consider this improved version here, though it has been implemented in the prototypical analyser used in Subsection 4.6.2.
4.5 Information extraction

We consider now the problem of how information can be extracted from an abstract constraint \( P \in \text{Prog}_k^V \). 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 \( \gamma^k(P) \leq v \in d \). Since \( v \in d = \gamma^k(\{v(d) \leftarrow \}) \), the following result allows us to compare \( M_T(P) \) with \( M_T(\{v(d) \leftarrow \}) \) instead of their concretisation through \( \gamma^k \).

**Proposition 4.11.** Let \( T = \langle \Delta, \Sigma, I \rangle \) be a type system, \( k \geq 1 \) and \( V \in \varphi_I(V) \). For any \( \{P_1, P_2\} \subseteq \text{Prog}_k^V \) we have that
\[
P_1 \leq P_2 \text{ entails } \gamma^k_T(P_1) \leq \gamma^k_T(P_2).
\]

Since in general we have an infinite set of type models, Proposition 4.11 does not provide an algorithm for checking if \( \gamma^k(P) \leq \gamma^k(\{v(d) \leftarrow \}) \). This can be achieved, instead, by using the following result.

**Proposition 4.12.** Let \( T = \langle \Delta, \Sigma, I \rangle \) be a type system, \( k \geq 1 \), \( V \in \varphi_I(V) \), \( P \in \text{Prog}_k^V \), \( v \in V \) and \( d \in \text{terms}(\Delta, T) \). If \( v(d) \) can be derived from \( P \) by resolution (i.e., there is a refutation of \( v(d) \leftarrow \) in \( P \) with \( \varepsilon \) as computed answer substitution), then \( P \leq \{v(d) \leftarrow \} \).

In general, the resolution process is not finite. Therefore, we must halt after \( n \) steps of resolution at most. The greater is \( n \), the more precise will be the entailment check. Moreover, note that resolution is not complete w.r.t. our entailment check. Consider for instance the case \( P = \{v(\text{int}) \leftarrow \} \) and \( d = \text{top} \). We have \( P \leq \{v(\text{top}) \leftarrow \} \) 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.

4.6 Examples

We show the application of the domain \( \text{Prog}_k^V \) to the type analysis of two procedures. The first one is the classical \texttt{append/3} procedure which appends two lists. We compute its denotation by hand, step by step. That way, we expose all the ideas described in this chapter. The second example is the type analysis of a complex procedure. This time, an automatic analyser is used, embedding negative and some subtyping information. This example allows us to describe a prototypical type analyser which uses \( \text{Prog}_k^V \).

4.6.1 Type analysis of append/3

Consider the well-known procedure \texttt{append/3} which appends two lists.
\[
\text{append([], L, L).}
\text{append([H|T], L, [H|A]) : -append(T, L, A).}
\]
We want to compute its computed answer semantics using the $Prog^2_V$ constraint system. We use a version of the $T_P$ operator of Definition 2.5 which uses rename instead of diagonal elements and cylindrification.

The first step of the analysis consists in the transformation of the program in the abstract syntax of Definition 2.2. The result is

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

The second step consists in the abstraction of the program (Definition 2.41) through the $\alpha_{alg}$ map of Definition 4.9. We assume to use a type system containing polymorphic lists, integers and a top type. Therefore, we use the type procedure of Figure 4.2. The result of the abstraction of the program is the following.

\[ x(\text{list}(T)) \leftarrow \]
\[ append(x, y, z) \leftarrow y(T) \leftarrow z(T) \text{ or } z(T) \leftarrow y(T) \]
\[ \begin{cases} 
  x(\text{list}(T)) \leftarrow h(T), t(\text{list}(T)) \\
  h(T) \leftarrow x(\text{list}(T)) \\
  t(\text{list}(T)) \leftarrow x(\text{list}(T)) \\
  z(\text{list}(T)) \leftarrow h(T), a(\text{list}(T)) \\
  h(T) \leftarrow z(\text{list}(T)) \\
  a(\text{list}(T)) \leftarrow z(\text{list}(T))
\end{cases} \]

By definition we have

\[ T_P \uparrow_{\emptyset}(append) = \emptyset . \]

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

\[ T_P \uparrow_1(\emptyset)(append) = \left\{ \begin{array}{l}
  t_1(\text{list}(T)) \leftarrow \\
  t_2(T) \leftarrow t_3(T) \\
  t_3(T) \leftarrow t_2(T)
\end{array} \right\} . \]

Note that variables have been renamed into $t$ variables, following Definition 2.5, though an explicit renaming operation has been applied instead of a diagonal element and cylindrification.

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 $CA^{Prog^2_V} [\text{append}(t, y, a)] T_P \uparrow_1(\emptyset)$. We follow definition 2.4, though an explicit renaming operation is applied instead of a diagonal
The second step is the computation of $\mathcal{C}A^{\text{Prog}^2_v}[\text{append}(t, y, a)] T_P \uparrow 1(\emptyset)$.

Following Definition 2.4 we have

$$
\begin{align*}
\mathcal{C}A^{\text{Prog}^2_v}[\text{append}(t, y, a)] T_P \uparrow 1(\emptyset) \\
&= \mathcal{C}A^{\text{Prog}^2_v}[Q \; T_P \uparrow 1(\emptyset) \otimes \mathcal{C}A^{\text{Prog}^2_v}[\text{append}(t, y, a)] T_P \uparrow 1(\emptyset)] \\
&= \{Q\} \otimes \mathcal{C}A^{\text{Prog}^2_v}[\text{append}(t, y, a)] T_P \uparrow 1(\emptyset)
\end{align*}
$$

$$
\begin{align*}
&= \{Q\} \otimes \mathcal{C}A^{\text{Prog}^2_v} \begin{cases}
  t(\text{list}(T)) \leftarrow t(\text{list}(T)) \\
y(T) \leftarrow y(T) \\
a(T) \leftarrow a(T)
\end{cases}
\end{align*}
$$

Following Definition 2.5, we must rename the program $R$ and cylindrify w.r.t. the variables $\{t_1, t_2, t_3\}$. The renaming operation yields

$$
\text{rename}_{\rho(\text{Prog}^2_v)}(\{R\}) = \begin{cases}
  t(\text{list}(T)) \leftarrow t(\text{list}(T)) \\
t_2(T) \leftarrow a(T) \\
a(T) \leftarrow t_2(T) \\
t_1(\text{list}(T)) \leftarrow h(T), t(\text{list}(T)) \\
h(T) \leftarrow t_1(\text{list}(T)) \\
t(\text{list}(T)) \leftarrow t_1(\text{list}(T)) \\
t_3(\text{list}(T)) \leftarrow h(T), a(\text{list}(T)) \\
h(T) \leftarrow t_3(\text{list}(T)) \\
a(\text{list}(T)) \leftarrow t_3(\text{list}(T))
\end{cases}.
$$
We perform the cylindrification operation one variable at a time.

\[
\begin{align*}
\forall_{\{l\}}^{\text{Prog}^2} (\{R^1\}) &= \left\{ \begin{array}{l}
\ell_2(T) \leftarrow a(T) \\
a(T) \leftarrow \ell_2(T) \\
\ell_1(\text{list}(T)) \leftarrow h(T) \\
h(T) \leftarrow \ell_1(\text{list}(T)) \\
\ell_3(\text{list}(T)) \leftarrow h(T), \ell_3(\text{list}(T)) \\
h(T) \leftarrow \ell_3(\text{list}(T)) \\
a(\text{list}(T)) \leftarrow \ell_3(\text{list}(T)) \\
\end{array} \right\}, \\
R^2
\end{align*}
\]

\[
\begin{align*}
\forall_{\{a\}}^{\text{Prog}^2} (\{R^2\}) &= \left\{ \begin{array}{l}
\ell_2(\text{list}(T)) \leftarrow \ell_3(\text{list}(T)) \\
\ell_1(\text{list}(T)) \leftarrow h(T) \\
\ell_3(\text{list}(T)) \leftarrow h(T), \ell_2(\text{list}(T)) \\
h(T) \leftarrow \ell_1(\text{list}(T)) \\
h(T) \leftarrow \ell_3(\text{list}(T)) \\
\end{array} \right\}, \\
R^2
\end{align*}
\]

\[
T_P \uparrow_2 (\emptyset)(\text{append}) = \left\{ \begin{array}{l}
\ell_1(\text{list}(T)) \leftarrow \ell_3(\text{list}(T)) \\
\ell_2(T) \leftarrow \ell_3(T) \\
\ell_3(T) \leftarrow \ell_2(T) \\
\end{array} \right\} \cup \forall_{\{h\}}^{\text{Prog}^2} (\{R^3\})
\]

\[
= \left\{ \begin{array}{l}
\ell_1(\text{list}(T)) \leftarrow \ell_3(\text{list}(T)) \\
\ell_2(T) \leftarrow \ell_3(T) \\
\ell_3(T) \leftarrow \ell_2(T) \\
\end{array} \right\}.
\]

Since it can be shown that \(T_P \uparrow_3 (\emptyset) = T_P \uparrow_2 (\emptyset)\), we have \(S_P = T_P \uparrow_2 (\emptyset)\).

Assume we are interested in the abstract behaviour of the procedure \text{append} when it is called with its first and second argument bound to lists. This means that we want to compute

\[
\mathcal{C}A^{\text{Prog}^2} \left[ \begin{array}{l}
x(\text{list}(\text{top})) \leftarrow \\
y(\text{list}(\text{top})) \leftarrow \\
\text{and append}(x, y, z) \\
\end{array} \right] S_P
\]

which is

\[
\left\{ \begin{array}{l}
x(\text{list}(\text{top})) \leftarrow x(\text{list}(\text{top})) \leftarrow \\
y(\text{list}(\text{top})) \leftarrow y(\text{list}(\text{top})) \leftarrow \\
x(\text{list}(\text{top})) \leftarrow , x(\text{list}(\text{top})) \leftarrow z(\text{list}(\text{top})) \\
y(T) \leftarrow z(T) \phantom{a} y(\text{list}(\text{top})) \leftarrow z(\text{list}(\text{top})) \\
z(T) \leftarrow y(T) \phantom{a} z(\text{list}(\text{top})) \leftarrow x(\text{list}(\text{top})), y(\text{list}(\text{top})) \\
\end{array} \right\}.
\]

Since from both programs it is possible to derive the fact \(z(\text{list}(\text{top}))\) by resolution, we conclude that \(z\) is bound to a list after the call of \text{append} with its first two arguments bound to lists (Proposition 4.12).
\begin{verbatim}
int(0).
\texttt{int(s(I)):-int(I)}.

\texttt{derivative(x,s(0)).}
\texttt{derivative(X,0):-int(X)}.
\texttt{derivative(X*Y,(D*Y)+(X*DY)):-derivative(X,DX),derivative(Y,DY).}
\texttt{derivative(X+Y,DX+DY):-derivative(X,DX),derivative(Y,DY).}
\texttt{derivative(-X,-(DX)):-derivative(X,DX).}
\texttt{derivative(X-Y,DX-DY):-derivative(X,DX),derivative(Y,DY).}
\texttt{derivative(X*K,D*K*DX*(K-s(0)))):-derivative(K,DK).}
\texttt{derivative(exp(X),DX*exp(X)):-derivative(X,DX).}
\texttt{derivative(sin(X),DX*cos(X)):-derivative(X,DX).}
\texttt{derivative(cos(X),-(DX*sin(X))):-derivative(X,DX).}
\end{verbatim}

Figure 4.3: The \texttt{derivative/2} procedure.

4.6.2 Type analysis of \texttt{derivative/2}

We have implemented a small analyser for pure logic programs. It transforms a logic program into the abstract syntax of Definition 2.2, then abstracts the program by using a generic domain. Finally, it computes the abstract fixpoint and allows us to evaluate queries. We have used $Prog_V^k$ as abstract domain. The domain can be specialised w.r.t. a given set of types, through the specification of the \texttt{type} procedure of Definition 4.7. The domain implements negative information.

Consider the program shown in Figure 4.3. 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

\[ (x(\text{algebraic}) \leftarrow ) \text{ and derivative}(x,y) \]

in the abstract fixpoint computed by the analyser\textsuperscript{3}. The result is the set of constraints shown in Figure 4.4. If the predicate \texttt{false} is derivable by resolution from a constraint, then that constraint can be dropped. 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. This means that our analyser allows us to conclude that the derivative of an algebraic expression is an algebraic expression.

\textsuperscript{3}Actually, the query is \texttt{derivative(algebraic,top)}, but we use the syntax of Definition 2.2 to make things consistent.
<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>
<tr>
<td>y(int) ←</td>
<td>y(int) ←</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>false ← x(int)</td>
</tr>
<tr>
<td>false ← x(int)</td>
<td>false ← y(int)</td>
</tr>
<tr>
<td>false ← y(algebraic)</td>
<td>false ← y(int)</td>
</tr>
<tr>
<td>false ← y(int)</td>
<td>x(algebraic) ←</td>
</tr>
<tr>
<td>x(algebraic) ←</td>
<td>x(expr) ←</td>
</tr>
<tr>
<td>x(expr) ←</td>
<td>y(algebraic) ←</td>
</tr>
<tr>
<td>y(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>false ← x(algebraic)</td>
</tr>
<tr>
<td>false ← y(int)</td>
<td>false ← x(int)</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>x(expr) ←</td>
<td>false ← y(int)</td>
</tr>
<tr>
<td>x(algebraic) ← x(algebraic) ←</td>
<td>x(expr) ← y(expr)</td>
</tr>
<tr>
<td>y(algebraic) ← y(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>false ← 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) ←</td>
<td>x(expr) ←</td>
</tr>
<tr>
<td>y(expr) ←</td>
<td>x(expr) ←</td>
</tr>
<tr>
<td>y(expr) ←</td>
<td>x(expr) ←</td>
</tr>
</tbody>
</table>

Figure 4.4: The set of constraints computed for our query.
Note that this result has been possible only by using negative information, since the constraints we have dropped do not allow one to derive the fact $y(\text{algebraic})$.

## 4.7 Conclusions

This chapter has presented a generic domain for the type analysis of pure logic programs, involving polymorphic types and dependencies among types. We have used the abstract interpretation framework to relate our domain to the underlying domains of transfinite formulas defined in Chapter 3. Those domains can be developed through the formal methodology of linear refinement (see Section 3.6), and enjoy many interesting properties of isomorphism and optimality (see Chapter 3). This means that we have built practical domains over theoretical domains of transfinite formulas. The advantage of this approach is that it is clear where we decide to lose precision to gain the effectivity of the analysis. For instance, this happens with the choice of a finite $k$ in $\text{Prog}_k^V$, and with the use of an imprecise but efficient cylindrification operation (Proposition 4.6).

The main result is that logic programs can be used as abstract domains. Even logic variables have a clear meaning as type variables. We claim that this is not restricted to the particular case of the analysis of logic programs, but is a general result which applies to other programming paradigms.

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. This distinguishes neatly our approach from the many others contained in the literature. However, we do not claim that ours is the best framework for type analysis. Other approaches may be faster, simpler or even more precise than ours. A detailed comparison of the results is almost impossible without a benchmark evaluation, since every framework uses a different notion of semantics or domain for analysis, and is intended for even more different applications.

We are left with several important open problems.

- The use of programs for representing the $\text{Pos}_\Delta$ type constraint system (Definition 3.31) should be investigated. We think that $\text{Pos}_\Delta$ should be represented by disjunctive logic programs instead of traditional logic programs.

- The operation $\exists^{\text{Prog}}_k^V$ is just correct (Proposition 4.6). We are interested in an algorithm for the optimal cylindrification operation.

- We know that the algorithm for approximating the abstraction map is just correct (Proposition 4.10). 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 [45].
Subtyping information should be extracted from a type specification and used for improving the precision of the analysis.
4.8 Proofs

Proofs of Section 4.3

**Proposition 4.5.** Given \( \{P_1, P_2\} \subseteq \text{Prog}^k_V \), we have

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

\( \square \)

**Proposition 4.6.**

i) Straight from Proposition 4.5.

ii) By Definition 3.24 and 4.2

iii) First we show that \( \gamma^k(P) \leq \gamma^k(P' \cap \text{Prog}^k_{V\setminus x}) \). Indeed, consider a clause \( c \in P' \cap \text{Prog}^k_{V\setminus x} \). It is the unfolding of some clauses \( c_1, \ldots, c_n \) of \( P \) in a clause \( c_{n+1} \) of \( P \). Every ground instance \( c^g \) of \( c \) is the unfolding of suitable ground instances \( c^g_1, \ldots, c^g_n \) of \( c_1, \ldots, c_n \) in a suitable ground instance \( c^g_{n+1} \) of \( c_{n+1} \). Therefore, if \( \gamma^k(P) \| \theta = 1 \) then \( \| c^g_i \| \theta = 1 \) for \( i = 1, \ldots, n+1 \) and \( \| c^g \| \theta = 1 \). Since this is true for every \( c \in P' \cap \text{Prog}^k_{V\setminus x} \), we have that \( \gamma^k(P' \cap \text{Prog}^k_{V\setminus x}) \| \theta = 1 \).

We show now that \( \exists^\text{Def}\Delta, V \gamma^k(P) \leq \gamma^k(P \cap \text{Prog}^k_{V\setminus x}) \). Let \( \exists^\text{Def}\Delta, V \gamma^k(P) \| \theta = 1 \). Then \( \| \gamma^k(P)[S/x] \| \theta = 1 \) for a suitable \( S \in \varphi(\text{terms}(\Delta, \emptyset)) \). Consider a clause \( H \leftarrow B \in P \cap \text{Prog}^k_{V\setminus x} \). Since \( H \leftarrow B \in P \) and \( x \) does not occur in \( H \leftarrow B \), we have \( H'[t_1/V_1] \cdots [t_n/V_n] \leftarrow B'[t_1/V_1] \cdots [t_n/V_n] \| \theta = 1 \), for every \( \{t_1, \ldots, t_n\} \subseteq \text{terms}(\Delta, \emptyset) \), where \( V_1, \ldots, V_n \) are the type variables of \( H \leftarrow B \). This means that \( \| \gamma^k(P \cap \text{Prog}^k_{V\setminus x}) \| \theta = 1 \).

Using the above facts we conclude that

\[
\exists^\text{Def}\Delta, V \gamma^k(P) = \exists^\text{Def}\Delta, V (\gamma^k(P) \land \gamma^k(P' \cap \text{Prog}^k_{V\setminus x}))
= (\exists^\text{Def}\Delta, V \gamma^k(P)) \land \gamma^k(P' \cap \text{Prog}^k_{V\setminus x})
\leq \gamma^k(P \cap \text{Prog}^k_{V\setminus x}) \land \gamma^k(P' \cap \text{Prog}^k_{V\setminus x})
= \gamma^k((P \cap \text{Prog}^k_{V\setminus x}) \cup (P' \cap \text{Prog}^k_{V\setminus x}))
= \gamma^k(\exists^\text{Prog}^k_{V}(P)).
\]

iv) By Definitions 4.4 and 3.14.

\( \square \)
Proposition 4.10. Since $c$ can be seen as a substitution, consider $\sigma \leq c$. We show that $[[\gamma^k \alpha^d(c)]]\sigma = 1$, i.e., that $[[\gamma^k \alpha^d(c)]]\sigma = 1$ for every $x \in V$. Let $x \in V$. We must show that for any $H \leftarrow B \in \alpha^d(c)$ and any $\mu : \mathcal{T} \rightarrow \text{terms}(\Delta, \emptyset)$ we have $[H^' \mu \leftarrow B^' \mu]\sigma = 1$. By Definition 4.9 we know that the clauses in $\alpha^d(c)$ are generated correspondingly to any computed answer substitution of $\text{type}(c(x), T y p e)$. Consider one such computed answer $\theta = \{T y p e \leftrightarrow t^i, x_1 \leftrightarrow t^i_{x_1}, \ldots, x_n \leftrightarrow t^i_{x_n}\}$, where $\text{vars}(c(x)) = \{x_1, \ldots, x_n\}$. We have

\[
\begin{align*}
[[x_1(t^i_{x_1}), \ldots, x_n(t^i_{x_n})]'_\mu]\sigma &= 1 \\
\iff [x_i \in t^i_{x_i} \land \ldots \land x_n \in t^i_{x_n}]\sigma &= 1 \\
\iff [x_j \in t^i_{x_j}]\sigma &= 1 & \text{for all } j = 1, \ldots, n \\
\iff \sigma(x_j) \in [t^i_{x_j}]I & \text{for all } j = 1, \ldots, n \\
& \iff \sigma(x_j) \in [\theta(x_j)]I & \text{for all } j = 1, \ldots, n \\
(\sigma \leq c) & \iff \sigma(x) \in [\nu^i I] \\
& \iff [[x \in \nu^i]']\sigma = 1. 
\end{align*}
\]

Therefore, for all $i = 1, \ldots, m$ we have $[[x(t^i) \leftarrow x_1(t^i_{x_1}), \ldots, x_n(t^i_{x_n})]'_\mu]\sigma = 1$ and $[[x_1(t^i_{x_1}), \ldots, x_n(t^i_{x_n})]'_\mu]\sigma = 1$ for any $j = 1, \ldots, n$, which proves the thesis (because of Definition 4.9).

Proposition 4.11. Assume $[[\gamma^k(P_1)]]\emptyset = 1$. Let $M(v) = \{d \in \text{terms}(\Delta, \emptyset) \mid \theta(v) \in [d]I\}$. For every clause $H \leftarrow B \in P_1$ and every $\mu : \mathcal{T} \rightarrow \text{terms}(\Delta, \emptyset)$ we have $[[H^' \mu \leftarrow B^' \mu]]\emptyset = 1$. This means that $M \models H \mu \leftarrow B \mu$. Therefore, $M \models P_1$. Since $P_1 \leq P_2$ we have $M \models P_2$. This means that for every clause $H \leftarrow B \in P_2$ and every $\mu : \mathcal{T} \rightarrow \text{terms}(\Delta, \emptyset)$ we have $M \models H \mu \leftarrow B \mu$, which entails that $[[H^' \mu \leftarrow B^' \mu]]\emptyset = 1$. We conclude that $[[\gamma^k(P_2)]]\emptyset = 1$.

Proposition 4.12. Consider $\mu : \mathcal{T} \rightarrow \text{terms}(\Delta, \emptyset)$. Since $\nu(d)$ is derivable from $P$, then $\nu(d)_\mu$ is derivable from $P$. We show that if $M \models P$ then $M \models \nu(d)_\mu$. Since this is done for every $\mu$, we have the thesis. We proceed by induction on the number $n$ of the steps of resolution. If $n = 1$ then there exists a clause $\nu(t) \leftarrow \in P$ such that $t_{\mu'} = d_{\mu}$ for a suitable $\mu'$. Since $M \models \nu(t) \leftarrow$, we have $M \models \nu(t)_{\mu'} \leftarrow$, i.e., $M \models \nu(d)_{\mu} \leftarrow$. Assume the result is true for $n \geq 1$ and that $\nu(d)_\mu$ is derivable from $P$ by $n + 1$ steps of resolution. Then there exists a clause $\nu(t) \leftarrow B_1, \ldots, B_m \in P$ such that $t_{\mu'} = d_{\mu}$ for a suitable $\mu'$ and $B_i_{\mu'}$ is derivable from $P$ in $n$ steps for any $i = 1, \ldots, m$. By inductive hypothesis, we conclude that $P \leq B_i_{\mu'}$ for any $i = 1, \ldots, m$. Since $M \models P$, we have $M \models B_i_{\mu}$ for any $i = 1, \ldots, m$, and since $M \models \nu(t)_{\mu'} \leftarrow B_i_{\mu'}$, $B_i_{\mu}$ we have $M \models \nu(t)_{\mu'}$, i.e., $M \models \nu(d)_{\mu} \leftarrow$. \qed
We address the problem of how linear refinement should be used in order to improve significantly the precision of a basic domain. Moreover, we show how linearly refined domains can be represented through data structures, and how the abstract operations can be implemented by algorithms which are correct for every abstract property. Finally, the issue of the complexity of the analysis is addressed. We show that linear refinement can be seen as a compilation technique. Therefore, it enjoys good complexity results and can be used fruitfully for modular analysis. This last aspect is very important in the case of incremental analysis or in the case of analysis in the presence of third-party code, whose source is not available for copyright reasons.

5.1 Introduction

We know that linear refinement was meant to be an automatic way for improving the precision of a domain w.r.t. the approximation of some concrete operation [39, 73]. However, there is no guarantee that the improved domain reaches some significant level of precision. The fact that linear refinement was applied only to the very special case of groundness analysis [72] has hidden this problem for a while.

Another problem is related to the algorithmic definitions of the abstract operations on a domain. In [39, 73] a domain is just a set of sets of concrete objects, i.e., the set of fixpoints of an upper closure operator. The concrete objects in a set enjoy the same abstract properties. This point of view is theoretically clean and allows to simplify notation and proofs. We have used this approach in Chapter 3 and we will use it again in this and the next chapters. However, we must provide names for sets in order to implement an abstract domain in an analysis engine. Moreover, we know that, given a domain, the abstract operations are induced by the domain itself [25, 26]. Therefore, [39, 73] do not consider the problem of how the abstract operations can be implemented by suitable algorithms. However, [25, 26] do not
provide an explicit definition of the induced operations. The formula they provide cannot be automatically translated into an algorithm.

A problem strictly related to that of the algorithmic definition of the abstract operations is that of the complexity of the analysis. It is not obvious that the best possible approximation of a concrete operation enjoys good computational complexity. In general, we can admit a loss of precision in the name of better performance. Therefore, the optimal operations induced by an abstract domain have just a theoretical interest.

Finally, [39, 73] do not consider the way an abstraction map can be computed through an algorithm. However, we must provide such an algorithm, since it is used during the analysis process.

In this chapter we address all these problems and we provide practical recommendations to the designer of abstract domains. Moreover, we provide a generic way for representing linearly refined domains, and generic algorithms which are always correct w.r.t. concrete conjunction, restriction, expansion, renaming, cylindrification and union. These algorithms have predictable and reasonable computational complexity. In the next chapter we apply these results to the design of a domain for non pair-sharing and freeness analysis of logic programs.

The chapter is organised as follows. Section 5.2 shows how the linear refinement technique should be used in order to obtain useful arrows. Section 5.3 provides a refinement operation on representations for every refinement operation on domains. In Section 5.4 we consider a formalisation of good properties of existential Herbrand constraints. Section 5.5 provides algorithms for all the abstract operations, which are correct for every good property. Section 5.6 defines a heuristics for the definition of the abstraction map, and shows that this map can be seen as a compiler from constraints to dependencies. Section 5.7 considers the issue of the complexity of the analysis, giving some advice for keeping this complexity as small as possible. We conclude in Section 5.8.

5.2 Improving precision

In the case of groundness analysis, [72] shows that a significant precision can be attained by linearly refining a basic domain which models just groundedness. The same has been done in Section 3.6 by using domains which model basic type properties. This might suggest that it is a general result. Instead, [46] shows that the linear refinement of a basic domain for freeness does not feature a useful level of precision. The same will be shown in Chapter 6 for the case of a basic domain for non pair-sharing. In general, we can hope that a good level of precision can be achieved by repeated application of linear refinement. However, there is no theoretical evidence of this result. Moreover, iterated refinement leads to domains which are large or even huge (consider the case of non pair-sharing in Chapter 6), whose implementation is difficult to devise.
In this section we provide an inductive way for representing domains obtained as Moore closure, reduced product and linear refinement of simpler domains. This means that for every refinement operator of domains we have a corresponding refinement operator of representations.

In the following, we assume the complete lattice \( \langle C, \leq \rangle \) to be our concrete domain.
Definition 5.1. A representation of \( D \subseteq C \) is a finite set \( R \) of data structures together with an onto map \( \text{unrep} : R \to D \) and a map \( \text{dim} : R \to \mathbb{N} \). We say that \( r \in R \) represents every \( d \in D \) such that \( d \leq \text{unrep}(r) \) and has dimension \( \text{dim}(r) \).

We define \( \text{dim}(R) = \max \{ \text{dim}(r) \mid r \in R \} \). We assume that \( \text{dim}(R) \) is the time required for reading, writing and comparing elements of \( R \).

If we have a representation for a set of abstract properties, we know how to represent its Moore closure.

Definition 5.2. Assume we have a finite set of abstract properties \( D \subseteq C \) which is represented by \( R \). Define \( \wedge R \), where \( \text{unrep}(\{r_1, \ldots, r_n\}) = \prod C \{\text{unrep}(r_1), \ldots, \text{unrep}(r_n)\} \) and \( \text{dim}(r) = \sum_{i \in r} \text{dim}(i) \) for \( r \in \wedge R \). We often write \( r_1 \cdots r_n \) for \( \{r_1, \ldots, r_n\} \) and \( s_1 \cdots s_m \) for \( \cup_{i=1,\ldots,m} s_i \), where \( \{s_1, \ldots, s_m\} \subseteq \wedge K \) for \( i = 1, \ldots, m \).

If we have a representation for two domains, we know how to represent their reduced product.

Definition 5.3. Assume \( D_1 \subseteq C \) and \( D_2 \subseteq C \) are represented by \( R_1 \) and \( R_2 \), respectively. Then \( D_1 \cap D_2 \) is represented by \( R_1 \cap R_2 = R_1 \times R_2 \) where \( \text{unrep}(\langle r_1, r_2 \rangle) = \prod C \{\text{unrep}(r_1), \text{unrep}(r_2)\} \) and \( \text{dim}(\langle r_1, r_2 \rangle) = \text{dim}(r_1) + \text{dim}(r_2) \) for every \( r_1 \in R_1 \) and \( r_2 \in R_2 \).

If we have a representation of two domains, we know how to represent their linear arrows.

Definition 5.4. Assume \( D_1 \subseteq C \) and \( D_2 \subseteq C \) are represented by \( R_1 \) and \( R_2 \), respectively. If \( \exists : C \times C \to C \), we define the representation

\[
\{r_1 \rightarrow^{\exists} r_2, r_1 \leftarrow^{\exists} r_2 \mid r_1 \in R_1 \text{ and } r_2 \in R_2\}
\]

of \( \{a \rightarrow^{\exists} b, a \leftarrow^{\exists} b \mid a \in D_1 \text{ and } b \in D_2\} \), whose elements can be seen as pairs endowed with a directionality flag. Moreover, we define \( \text{unrep}(r_1 \rightarrow^{\exists} r_2) = \text{unrep}(r_1) \rightarrow^{\exists} \text{unrep}(r_2) \), \( \text{unrep}(r_1 \leftarrow^{\exists} r_2) = \text{unrep}(r_1) \leftarrow^{\exists} \text{unrep}(r_2) \) and \( \text{dim}(r_1 \rightarrow^{\exists} r_2) = \text{dim}(r_1) + \text{dim}(r_2) \). When it is clear from the context, we drop the \( \exists \) superscript.

If \( D \subseteq C \) is represented by \( R \), we can represent \( D \rightarrow^{\exists} D \) as

\[
R \rightarrow^{\exists} R = R \cap \bigcup \{r_1 \rightarrow^{\exists} r_2, r_2 \leftarrow^{\exists} r_1 \mid \{r_1, r_2\} \subseteq R\}.
\]

If Equation (1.4) can be used, this representation can be simplified since we do not need the reduced product operation. Moreover, if \( D = \wedge K \) and \( R = \wedge R' \), where \( R' \) is a representation of \( K \), from Equation (1.5) we conclude that a representation of \( D \rightarrow^{\exists} D \) is

\[
R \rightarrow^{\exists} R = R \cap \bigcup \{r_1 \rightarrow^{\exists} r_2, r_2 \leftarrow^{\exists} r_1 \mid r_1 \in R \text{ and } r_2 \in R'\}.
\]
As above, if Equation (1.6) can be used, this representation can be further simplified since we do not need the reduced product operation.

In the following, we will always assume that both these simplifications can be applied. Therefore, when referring to \( R \rightarrow \mathcal{R} \), we mean

\[
\bigcup \{ r_1 \rightarrow \mathcal{R} r_2, r_2 \rightarrow \mathcal{R} r_1 \mid r_1 \in R \text{ and } r_2 \in R' \}.
\]

### 5.4 Variable-related and local properties

In Section 5.5 we provide correct algorithms over the representation of linearly refined domains. Except for the case of a generic algorithm for \( \mathcal{R} \rightarrow \mathcal{R} \), which can be provided for every linearly refined domain, we will consider there the concrete domain and the operations of the logic programming case. However, note that those results can be easily generalised to the case of store-based programming languages.

When considering logic programming, we restrict to the case of (local) variable-related properties.

**Definition 5.5.** A variable-related property is a pair

\[
\text{property} = \langle \{\text{property}_V\}_{V \in \varphi_f(V)}, \text{target} \rangle,
\]

where \( \text{target} : \varphi_f(V) \rightarrow \varphi(\varphi_f(V)) \) is monotonic and, for any \( V \in \varphi_f(V) \), \( \text{target}(V) \subseteq \varphi(V) \) and \( \text{property}_V \subseteq H_V \times \text{target}(V) \). Its width is

\[
\text{width}(\text{property}) = \max \{ \#V' \mid V \in \varphi_f(V), V' \in \text{target}(V) \}.
\]

It is local if and only if for every \( V \in \varphi_f(V) \) the following conditions hold

i) for every \( x \in V \setminus V \), \( \exists_w c \in H_V[x], V' \in \text{target}(V) \) and \( N \in \mathcal{W} \) fresh, we have \( \text{property}_V(\exists_w c[V/N/x], V') \) if and only if \( \text{property}_V(\exists_w c, V') \);

ii) for every \( \exists_w c \in H_V, V' \in \text{target}(V) \) and \( \{v_1, v_2\} \subseteq V \), \( \text{property}_V(\exists_w c, V') \) if and only if \( \text{property}_V(\exists_w (c[v_1/v_2, v_2/v_1]), V'[v_1/v_2, v_2/v_1]) \);

iii) given \( x \in V \), for every \( \{\exists_w c_1, \exists_w c_2\} \subseteq H_V[x] \) and \( V' \in \text{target}(V) \) such that \( x \in V' \), we have \( \text{property}_V(\exists_w c_1, V') \) if and only if \( \text{property}_V(\exists_w c_2, V') \).

The target of a variable-related property tells which are the sets of variables over which the property is meaningful. The width of a variable-related property is the maximum dimension of the its target.

**Example 5.6.** Groundness \([22, 3, 23, 72]\) is a local variable-related property, defined as 

\[
\text{groundness} = \langle \{\text{groundness}_V\}_{V \in \varphi_f(V)}, \lambda V.\{v \mid v \in V\} \rangle \text{ where, for every } h \in H_V, V \in \varphi_f(V) \text{ and } v \in V, \text{groundness}_V(\exists_w c, \{v\}) \text{ is true if and only if } \text{vars}(c(v)) = \emptyset. \text{ Its width is 1.} \]
Example 5.7. Freeness [29, 66, 12, 15, 21, 65, 79, 55, 46] is a local variable-related property, defined as \( \text{freeness} = \{ \{ \text{freeness}_V \} \subseteq V \mid \lambda V. \{ \{ v \mid v \in V \} \} \) where, for every \( V \subseteq \varphi_f(\mathcal{V}) \), \( h \in H^V \) and \( v \in V \), \( \text{freeness}_V(\exists_{\mathcal{W}}c, \{ v \}) \) is true if and only if \( c(v) \in \mathcal{V} \cup \mathcal{W} \). Its width is 1.

Example 5.8. Sharing [56, 49, 12, 54, 55, 66, 75] is a local variable-related property, defined as \( \text{sharing} = \{ \{ \text{sharing}_V \} \subseteq V \mid \lambda V. \{ \{ v \mid v \in V \} \} \) where, for every \( V \subseteq \varphi_f(\mathcal{V}) \), \( h \in H^V \) and \( V' \subseteq V \), \( \text{sharing}_V(\exists_{\mathcal{W}}c, V') \) is true if and only if there exists \( v \in V \cup \mathcal{W} \) such that \( V' = \{ v' \in V \mid v' \in \text{vars}(c(v')) \} \). Its width is \( \infty \), since an arbitrarily large set of variables can share a given variable.

Example 5.9. Linearity [12, 54] is a local variable-related property. Namely, let \( \text{linearity} = \{ \{ \text{linearity}_V \} \subseteq V \mid \lambda V. \{ \{ v \mid v \in V \} \} \) where, for every \( V \subseteq \varphi_f(\mathcal{V}) \), \( h \in H^V \) and \( v \in V \), we have \( \text{linearity}_V(\exists_{\mathcal{W}}c, \{ v \}) \) is true if and only if every \( v' \in V \) occurs at most once in \( c(v) \). Its width is 1.

As the above examples show, the notion of local variable-related property is very general. Actually, a non local variable-related property is a rather artificial concept.

Example 5.10. Let \( V \subseteq \varphi_f(\mathcal{V}) \), \( y \in V \) and \( x \in \mathcal{V} \setminus V \). For any \( \exists_{\mathcal{W}}c \in H^V \) and \( V' \subseteq V \) we define \( \text{weird}_V(\exists_{\mathcal{W}}c, V') \) if and only if \( x \not\in \text{rng}(c) \). Given \( V' \subseteq V \), we have \( \text{weird}_V(\exists_{\mathcal{W}}c, V') \) but not \( \text{weird}_{V \cup \mathcal{W}}(\{ y \mapsto x \}, V') \). Then \( \{ \{ \text{weird}_V \} \subseteq V \mid \lambda V. \{ \{ v \mid v \in V \} \} \) violates condition i of Definition 5.5.

A set of variable-related properties induce a set of subsets of \( H^V \) for any \( V \subseteq \varphi_f(\mathcal{V}) \). We provide a representation for these sets.

Definition 5.11. Let \( P = \{ p_i \mid i = 1, \ldots, n \} \) be a finite set of variable-related properties. For any \( V \subseteq \varphi_f(\mathcal{V}) \), \( i = 1, \ldots, n \) and \( V' \subseteq t_i(V) \) we define

\[
p_i(V')_V = \{ h \in H^V \mid (p_i)_V(h, V') \}
\]

When it is clear from the context, we drop \( p_i \) and/or \( V \) from \( p_i(V')_V \). For any \( V \subseteq \varphi_f(\mathcal{V}) \), the properties in \( P \) induce the subset of \( \varphi(H^V) \)

\[
\text{Kernel}(P)_V = \{ p_i(V')_V \mid 1 \leq i \leq n \text{ and } V' \subseteq t_i(V) \}
\]

A representation of \( \text{Kernel}(P)_V \) is

\[
\text{RepKernel}(P)_V = \{ p_i(V') \mid 1 \leq i \leq n \text{ and } V' \subseteq t_i(V) \}
\]

An object \( p_i(V') \) can be implemented as a pair \( (p_i, V') \). We write \( p_i(p_1, \ldots, p_n) \) for \( p_i(\{ v_1, \ldots, v_n \}) \) and we drop the tag \( p_i \) when it is obvious from the context. In the same situation, we write \( v \) for \( p_i(v) \), where \( v \in V \). The map \( \text{unrep}_V \) is defined as \( \text{unrep}_V(p_i(V')) = p_i(V')_V \) and the map \( \text{dim}_V \) is defined as \( \text{dim}_V(p_i(V')) = 1 \).

Given a set of variable-related properties \( P \), Definition 5.11 allows us to define a domain \( \lambda \text{Kernel}(P)_V \) whose representation can be obtained from \( \text{RepKernel}(P)_V \) through Definition 5.2. The boldface notation of Definition 5.11 is extended to its elements. Therefore, \( r_V \) stands for \( \text{unrep}_V(r) \) for any \( r \in \lambda \text{Kernel}(P)_V \). The linear refinement of \( \lambda \text{Kernel}(P)_V \) can be represented through Definitions 5.4 and 5.3.
5.4.1  Definite and possible analyses

Definite analysis approximates a property from below. This means that the analyser computes a subset of the sets of targets of the properties of interest such that those properties hold for those sets in a given program point. If a superset is provided, we speak of possible analysis.

The domain \(\lambda \text{Kernel}(P)_V\) and its refinement can be used for definite analysis of the properties in \(P\). Note that every possible analysis of a property \(p\) can be seen as the definite analysis of the opposite property \(\text{non} p\) of \(p\).

**Definition 5.12.** Given a variable-related property \(p = \langle\{p\}_v \in \phi_f(V), t\rangle\), the opposite property is \(\text{non} p = \langle\{\text{non} p\}_v \in \phi_f(V), t\rangle\) such that, given \(V \in \phi_f(V), h \in H_V\) and \(V' \in t(V)\), \(\text{non} p\) holds if and only if \(p\) does not hold.

Note that if \(p\) is local then \(\text{non} p\) is local and \(\text{width}(\text{non} p) = \text{width}(p)\).

**Example 5.13.** Following Definition 5.12, non groundness is the local variable-related property non groundness, where groundness has been defined in Example 5.6. Given \(V \in \phi_f(V), v \in V\) and \(\exists W c \in H_V\) we have non groundness \(V(\exists W c, \{v\})\) if and only if \(\text{vars}(c(v)) \neq \emptyset\). Its width is 1.

5.5  Generic algorithms

In this section we tackle the problem of the definition of generic algorithms over \(R \rightarrow^{\mathbb{R}} R\), where \(R\) is the representation of a domain \(D \subseteq C\) defined as \(D = \lambda \text{Kernel}\). These algorithms are said generic since they are not related to the specific property modelled by \(D\), but are correct for every possible \(D\).

The advantages of having generic algorithms are many. Firstly, a generic algorithm can be used as the starting point for the definition of a specialised algorithm for a given property. Moreover, it can be used whenever a specialised algorithm could not be provided. Finally, a generic algorithm can be implemented in a generic analyser, i.e., an analyser which is parametric w.r.t. the property of interest. Therefore, clever implementations of the generic algorithm affect the efficiency of every analysis performed by using the generic analyser.

We do not claim that the generic algorithms we are going to define are always precise enough for practical use. Similarly, we do not claim that more precise generic algorithms could not be defined. However, we know that ours are precise enough for the case of non pair-sharing and freeness analysis (Chapter 6), and enjoy good complexity results.

In Subsection 5.5.1 we consider generic algorithms for \(\mathbb{R} \rightarrow^{\mathbb{R}} R\) and \(\cup R \rightarrow^{\mathbb{R}} R\). In this case, we do not make any restriction about the concrete domain \(C\) and the operation \(\mathbb{X}\), except from \(\mathbb{X}\) being a quantale. This is because the knowledge about the structure of \(C\) and the definition of \(\mathbb{X}\) has been hidden inside the abstraction map, which must be aware of what \(C\) and \(\mathbb{X}\) are in order to abstract an object of
C into a set of arrows, as shown in Section 5.6. However, this is not the case for other operations, possibly not related to \( E \). Therefore, in Subsections 5.5.3, 5.5.4, 5.5.5 and 5.5.6 we come back to the particular case of local variable-related properties of existential Herbrand constraints, and we provide generic algorithms for the operations of Section 2.4 and Subsection 2.5.2. However, we claim that the results of these subsections are general enough to be applied with slight modifications to other programming paradigms, where the notion of existential Herbrand constraint has been replaced by a notion of store.

5.5.1 Generic algorithms for \( \otimes^{R \to E} R \) and \( \cup^{R \to E} R \)

Since the domain \( D \to^{E} D \) is the linear refinement of \( D \) w.r.t. the operation \( \otimes \), it is not surprising that a generic algorithm for the abstract counterpart \( \otimes^{R \to E} R \) of \( \otimes \) can be easily defined by using the semantics of the arrows.

Definition 5.14. An arrow \( r \in R \to^{E} R \) is said tautological if and only if \( c \leq \text{unrep}(r) \) for every \( c \in C \). Given \( \{A_1, A_2\} \subseteq R \to^{E} R \) and a set \( T \subseteq R \to^{E} R \) of tautological arrows\(^1\), we define

\[
A_1 \otimes^{R \to E} R A_2 = \text{unrep}(A_1, A_2) \cup \text{unrep}(A_2, A_1) \cup \text{unrep}(A_1, A_2) \cup \text{unrep}(A_2, A_1),
\]

where

\[
\text{unrep}(B_1, B_2) = \{ r \leftarrow l_1 \cdots l_n \mid r \leftarrow r_1 \cdots r_n \in B_1 \cup T \text{ and } r_i \leftarrow l_i \in B_2 \cup T \text{ for every } i = 1, \ldots, n \},
\]

\[
\text{unrep}(B_1, B_2) = \{ l_1 \cdots l_n \rightarrow r \mid r_1 \cdots r_n \rightarrow r \in B_2 \cup T \text{ and } l_i \rightarrow r_i \in B_1 \cup T \text{ for every } i = 1, \ldots, n \}.
\]

Proposition 5.15. If \( \otimes \) is a quantale, then the operation \( \otimes^{R \to E} R \) is correct w.r.t. \( \otimes \) and has worst case time complexity \( O((a_1 + t)(a_2 + t)\dim(Kernel)) \), where \( a_1 \), \( a_2 \) and \( t \) are the dimensions of its first and second argument and of the set \( T \) of tautological arrows, respectively.

As we said in Section 2.9, in many cases a least upper bound operation is used by the collecting semantics. In these cases, an approximation of \( \cup : (D \to^{E} D)^2 \to (D \to^{E} D) \) is required.

Definition 5.16. Let \( \{A_1, A_2\} \subseteq R \to^{E} R \). We define

\[
\cup^{R \to E} R (A_1, A_2) = \{ l_1l_2 \rightarrow r \mid l_1 \rightarrow r \in A_1 \text{ and } l_2 \rightarrow r \in A_2 \}.
\]

Proposition 5.17. The operation \( \cup^{R \to E} R \) is correct and has worst case time complexity \( O(a_1a_2\dim(Kernel)) \), where \( a_1 \) and \( a_2 \) are the dimensions of its arguments.

\(^1\)The larger \( T \) is, the more precise \( \otimes^{R \to E} R \) is.
5.5.2 Back to logic programming

As we said at the beginning of this section, from now on we consider the particular case $C = \varphi(H_V)$ and $\Xi = x^{\varphi(H_V)}$. We define a constraint system of representations (Definition 2.1).

**Definition 5.18.** Given a finite set of variable-related properties $P$, a constraint system of representations for $P$ is the constraint system $\text{Rep}(P) = \{\text{Rep}(P)_V\}$, where $\text{Rep}(P)_V = \lambda \text{RepKernel}(P)_V \rightarrow \phi^{\varphi(H_V)} \lambda \text{RepKernel}(P)_V$. This constraint system is provided both in its classical form (Definition 2.1) and in its elastic version (Definition 2.17). The operations $\times^{\text{Rep}(P)_V}$ and $\cup^{\text{Rep}(P)_V}$ are the instantiation of the operations of Definition 5.14 and 5.16 to the case $R = \lambda \text{RepKernel}(P)_V$. Their complexity is $O((a_1+t)(a_2+t))$ and $O(a_1a_2)$, respectively, since $\text{dim}(\text{RepKernel}(P)_V) = 1$. The other operations are those defined in the following of this section. There are two maps $\text{unrep}_V$ and $\text{dim}_V$ for every $V \in \varphi_f(V)$, which are the $\text{unrep}$ and the $\text{dim}$ maps of $\text{Rep}(P)_V$, respectively.

5.5.3 A generic algorithm for $\text{expand}^{\text{Rep}(P)_V}$

Consider the $\text{expand}^{\varphi(H_V)}$ operation induced by the point-wise extension of the operation $\text{expand}^{H_V}$ of Definition 2.32. This operation maps a set in $\varphi(H_V)$ in the same set in $\varphi(H_{V,\xi})$, assuming $x \notin V$.

The following result shows that, for the case of local variable-related properties, if a dependency is correct for $h \in H_V$, then it is correct for $\text{expand}^{H_V}(h)$.

**Proposition 5.19.** Let $P$ be a finite set of local variable-related properties and $V \in \varphi_f(V)$. Given $x \in V \setminus V$ and $l_V \rightarrow \phi^{\varphi(H_V)} r_V \in \lambda \text{Kernel}(P)_V \rightarrow \lambda \text{Kernel}(P)_V$, we have $l_V \rightarrow \phi^{\varphi(H_V)} r_V \subseteq l_{V,\xi} \rightarrow \phi^{\varphi(H_{V,\xi})} r_{V,\xi}$.

As a consequence of the above proposition, the arrows of $A \in \text{Rep}(P)_V$ can be put in $\text{expand}^{\text{Rep}(P)_V}(A)$. The question is whether there are new arrows in $\text{expand}^{\text{Rep}(P)_V}$, possibly involving the newly introduced variable $x$. The proposition below shows that the behaviour of two variables which do not occur in an existential Herbrand constraint is the same.

**Proposition 5.20.** Let $P$ be a finite set of local variable-related properties, $V \in \varphi_f(V)$, $\{v_1, v_2\} \subseteq V$ and $\exists w c \in H_V$ such that $\{v_1, v_2\} \cap (\text{dom}(c) \cup \text{rng}(c)) = \emptyset$. Given $l \rightarrow r \in \lambda \text{Kernel}(P)_V \rightarrow \lambda \text{Kernel}(P)_V$, we have $\exists w c \in l \rightarrow r$ if and only if $\exists w c \in l[v_1/v_2, v_2/v_1] \rightarrow r[v_1/v_2, v_2/v_1]$.

The proposition above suggests to use a distinguished variable $? \in V$ which does not occur in the programs. This means that it does not occur in the existential Herbrand constraints we are dealing with. Whenever we need to know the behaviour of a newly introduced variable $x$ in $\text{expand}^{\text{Rep}(P)_V}(A)$, we can look at all the arrows in $A$ having $?$ on the right and *incarnate* them into new arrows, obtained by substituting
$x$ for $?$. The use of this generic variable is efficient and does not affect the clean construction through linear refinement. However, if the width of a variable-related property is greater than 1, when we incarnate $?$ into $x$ we wish to know how this new $x$ behaves in conjunction with the distinguished variable itself. This means that we need to know the behaviour of a variable which does not occur in the program w.r.t. another variable which does not occur in the program. This is possible if we use two or more distinguished variables. These variables will be referred to as $?_1, ?_2$ and so on. The number of distinguished variables we need is given by the maximum widths of the properties of interest. These arguments lead to the following definition.

**Definition 5.21.** Let $P$ be a finite set of variable-related properties such that $n = \max\{\text{width}(p) \mid p \in P\}$ is finite, $V \in \phi_f(V)$ and $x \in V \setminus V$. Let $?_1, \ldots, ?_n \subseteq V$ and $A \in \text{Rep}(P)_V$. We define

$$\text{expand}^{\text{Rep}(P)_V}_x(A) = A \cup \{l[x/?_1] \rightarrow r[x/?_1] \mid l \rightarrow r \in A\}.$$ 

**Proposition 5.22.** If $P$ is a finite set of local variable-related properties such that $n = \max\{\text{width}(p) \mid p \in P\}$ is finite, $V \in \phi_f(V)$ and $x \in V \setminus V$, then $\text{expand}^{\text{Rep}(P)_V}_x$ is correct w.r.t. $\text{expand}^{\text{Rep}(P)_V}_x$ applied to sets of existential Herbrand constraints where $?_1, \ldots, ?_n$ do not occur. Its worst-case time complexity is linear in the dimension of its argument.

### 5.5.4 A generic algorithm for $\text{restrict}^{\text{Rep}(P)_V}_x$

A generic algorithm for $\text{restrict}^{\text{Rep}(P)_V}_x$ removes all the references to the variable $x$.

**Definition 5.23.** Let $P$ be a finite set of variable-related properties, $V \in \phi_f(V)$, $x \in V$ and $A \in \text{Rep}(P)_V$. Let $X = \{p(V') \mid \langle p, t \rangle, V' \in t(V) \text{ and } x \in V'\}$. We define

$$\text{restrict}^{\text{Rep}(P)_V}_x(A) = \left\{l \setminus X \rightarrow r \setminus X \mid l \rightarrow r \in A, \text{ and for all } \langle p, t \rangle \in P \text{ and } V'' \in t(V) \text{ if } p(V'') \in l \text{ and } x \in V'' \text{ then } p_V(\epsilon, V'') \right\}.$$ 

**Proposition 5.24.** If $P$ is a set of local variable-related properties, $V \in \phi_f(V)$ and $x \in V$, then $\text{restrict}^{\text{Rep}(P)_V}_x$ is correct and has complexity linear in the dimension of its argument.

### 5.5.5 A generic algorithm for $\text{rename}^{\text{Rep}(P)_V}_x$

**Definition 5.25.** Let $P$ be a finite set of variable-related properties, $V \in \phi_f(V)$, $x \in V$, $n \in V \setminus V$ and $A \in \text{Rep}(P)_V$. We define

$$\text{rename}^{\text{Rep}(P)_V}_x(A) = A[n/x].$$
Proposition 5.26. If \( P \) is a finite set of local variable-related properties, \( V \in \wp_f(V), x \in V \) and \( n \in V \setminus V \), then \( \text{rename}^\text{Rep}(P)_x^n \) is correct and has complexity linear in the dimension of its argument.

5.5.6 Other generic algorithms

Definition 5.27. Let \( P \) be a finite set of variable-related properties whose width is finite, \( V \in \wp_f(V), x \in V \) and \( A \in \text{Rep}(P)_V \). We define

\[
\exists^\text{Rep}[P]_V (A) = \text{restrict}^\text{Rep}[P]_{V \cup n} (\text{expand}^\text{Rep}[P]_{V \cup n \setminus x} (\text{rename}^\text{Rep}[P]_{x \to n} (A)))
\]

Proposition 5.28. If \( P \) is a finite set of local variable-related properties such that \( n = \max\{\text{width}(p) \mid p \in P\} \) is finite, \( V \in \wp_f(V) \) and \( x \in V \), then \( \exists^\text{Rep}(P)_x \) is correct w.r.t. \( \exists^\text{Rep}(P)_x \) applied to sets of existential Herbrand constraints where \( ?_1, \ldots, ?_n \) do not occur. Moreover, it has complexity linear in the dimension of its argument.

We consider now the problem of checking when the analyser has reached the fixpoint. Since all the generic algorithms we have provided in this section are monotonic w.r.t. set inclusion, it follows that it suffices to check if two successive iterations of the abstract immediate consequence operator are syntactically the same. Note that, however, \( \text{Rep}(P)_V \), in general, is not isomorphic to the domain \( \land \text{Kernel}(P)_V \to \land \text{Kernel}(P)_V \) it represents. Therefore, syntactical equality might allow more iterations than it is needed. The alternative would be to use semantical equality, by comparing the \text{unrep} of two successive iterations of the abstract immediate consequence operator. However, this would be more time-consuming than syntactical equality, and might not prove useful in real cases.

5.6 Abstraction as compilation

In the previous section we have been able to define generic algorithms for linearly refined domains since the information about the specific property of interest for the analysis has been compiled by the abstraction map. This means that, by using the linear refinement technique, the abstraction map compiles a set of existential Herbrand constraints into the set of arrows they satisfy. These arrows can be then executed by a generic abstract analyser which uses the algorithms of Section 5.5. Note that the compilation is strictly related to the property of interest. Therefore, it is not possible to provide a generic algorithm for the abstraction map. In this section, we provide a heuristics which allows us to define this algorithm in a methodological way once an abstract property is given. We restrict to the case of logic programming, though the ideas contained in this section can be easily generalised to store-based programming languages.
The problem here is the definition of an algorithm which provides a correct representation of the restriction to singletons of the abstraction map \( \alpha_V : \varphi(H_V) \rightarrow \lambda \text{Kernel}(P)_V \bowtie^\varphi(H_V) \lambda \text{Kernel}(P)_V \) for any finite set of variable-related properties \( P \) and any \( V \in \varphi_f(V) \). If we know an approximation of the abstraction of a single binding, we can provide an approximation of the abstraction of an existential Herbrand constraint. Moreover, this approximation can be further refined by adding global information, i.e., information which cannot be derived by observing every single binding, but requires the observation of the whole constraint.

**Definition 5.29.** Given a finite set of variable-related properties \( P, V \in \varphi_f(V) \), \( \alpha^{\text{base}}_V : \{ v = t \mid v \in V, \ t \in \text{terms}(\Sigma, V) \} \rightarrow \text{Rep}(P)_V \) and \( \alpha^{\text{global}}_V : C_V \rightarrow \text{Rep}(P)_V \), we define

\[
\alpha^{\text{local}}_V(v) = \{ k \rightarrow k \mid k \in \text{RepKernel}(P)_V \}
\]

\[
\alpha^{\text{local}}_V(\{ v_1 = t_1, \ldots, v_n = t_n \}) = \alpha^{\text{base}}_V(v_1 = t_1) \bowtie^{\text{Rep}(P)_V} \alpha^{\text{local}}_V(\{ v_2 = t_2, \ldots, v_n = t_n \}),
\]

where \( \{ v_1 = t_1, \ldots, v_n = t_n \} \in C_V \). Finally, we define

\[
\alpha^{\text{alg}}_V(\exists w c) = \exists w^{\text{Rep}(P)}_V \left( \alpha^{\text{local}}_V(c) \cup \alpha^{\text{global}}_V(c) \right).
\]

**Proposition 5.30.** Let \( P \) be a finite set of variable-related properties, \( V \in \varphi_f(V) \) and \( \alpha_V \) the abstraction map induced by the Moore family \( \lambda \text{Kernel}(P)_V \bowtie^\varphi(H_V) \lambda \text{Kernel}(P)_V \). Let \( \alpha^{\text{base}}_V : \{ v = t \mid v \in V, \ t \in \text{terms}(\Sigma, V) \} \rightarrow \text{Rep}(P)_V \) and \( \alpha^{\text{global}}_V : C_V \rightarrow \text{Rep}(P)_V \) be such that

i) \( \{ v = t \} \in \text{unrep}_V(\alpha^{\text{base}}_V(v = t)) \) for any \( v \in V \) and \( t \in \text{terms}(\Sigma, V) \);

ii) \( c \in \text{unrep}_V(\alpha^{\text{global}}_V(c)) \) for any \( c \in C_V \).

Then \( \alpha^{\text{alg}}_V \) is correct w.r.t. \( \alpha_V \).

By using \( \alpha^{\text{alg}}_V \), we can provide correct approximations of the diagonal elements over \( \varphi(H_V) \).

### 5.6.1 Information extraction

Since the result of the analysis of the properties in \( P \) is an element \( A \in \text{Rep}(P)_V \), we are interested in a generic algorithm which allows one to compute a subset of the sets of variables for which the properties of interest hold for any \( h \in \text{unrep}_V(A) \).

**Definition 5.31.** Given a finite set of variable-related properties \( P, \langle p, t \rangle \in P, \ V \in \varphi_f(V) \) and \( A \in \text{Rep}(P)_V \), we define

\[
\text{extract}_{\text{p}}(A) = \{ p(V') \mid V' \in t(V), \ l \rightarrow p(V') \in A \text{ and } \varepsilon \in 1 \}.
\]
The following proposition shows that if $p(V') \in \text{extract}_{V'}(A)$ then $p_V(h,V')$ actually holds for any $h \in \text{unrep}_V(A)$.

**Proposition 5.32.** Let $P$ be a finite set of variable-related properties, $(p,t) \in P$ \( V \in \varphi_f(V) \) and $A \in \text{Rep}(P)_V$. Then $\text{unrep}_V(A) \subseteq \text{extract}_{p_V}(A)$.

### 5.7 The complexity issue

The time and space required for analysing a program are very important since, for complex programs, they might be so large that the analysis cannot be performed in a reasonable time and without consuming the entire memory of the system. At the same time, a practical analyser is not required to perform the analysis in a few seconds’ time. Quoting from [85], this is because

only *production versions* deserve to be compiled with the optimization passes turned on, a production version is compiled once and used thousands, perhaps millions of times, and computers do work overnight.

So we do not participate in the race for the fastest ever analysis, especially when done (as is often the case) at the expense of precision.

However, note that this point of view is quite biased towards optimisation, while analysis is used for debugging and for extracting information useful to the programmer, who might not like working overnight.

The problem of complexity has been largely misunderstood in the context of program analysis. This is because it is not enough to provide complexity results for the abstract operators (like we did in Section 5.5), but we must be able to estimate the number of iterations which are needed to reach the fixpoint. Experimental evaluations show that this number is usually small, though it can arbitrarily grow for some *artificial* programs, up to the number of the elements of the abstract domain. Note that, in our case, the cardinality of the set \( \{ \dim(A) \mid A \in \text{Rep}(P)_V \} \) is exponential in \( \dim(\text{RepKernel}(P)_V) \) which, in turn, is usually polynomial in the number of variables of interest (see Examples 5.6, 5.7, 5.9), or even exponential in the same quantity\(^2\) (see Example 5.8).

This argument entails that the complexity of the analysis can be kept as small as possible by reducing the dimensions of the representation and of its elements. This, in turn, can be achieved

1. by using an elastic version of the semantics, as we said in Section 2.5;

2. by using a collecting semantics which uses a least upper bound operation instead of sets of constraints;

\(^2\)This justifies our refusal, in Section 5.2, of applying linear refinement to already linearly refined domains, in general.
3. by using classes of equivalence of representations. Indeed, since more elements of $R \rightarrow^* R$ may represent the same object, we can choose one whose dimension is minimal, by reducing, at the same time, the cardinality of the set $\{\dim(A) \mid A \in \text{Rep}(P)\}$;

4. by applying the analysis in an incremental way, following the modular structure of complex programs.

Note that point 3 reduces both the dimension of the elements and the dimension of the domain of representation. However, this technique must be used carefully, since the complexity of the reduction must be taken into account.

We think that point 4 can provide real help in reducing the complexity of the analysis of large programs. Note that our approach, based on sets of dependencies, is ideal for modular analysis. Indeed, the denotation of a procedure is not a static object, but contains the description of its behaviour in every possible context.

Linear refinement contributes to the reduction of the complexity of the analysis since

1. it allows precise modular, incremental analysis;

2. it admits generic algorithms (Section 5.5), which can be implemented by using very clever optimisations like structure sharing;

3. the knowledge about the property of interest is hidden inside the abstraction map. This map can be very complex, but it is applied just once, at program abstraction time, while the abstract algorithms are used inside the fixpoint computation, and are efficient since they are generic, blind algorithms which do not consider the abstract property anymore. We spoke of abstraction as compilation (Section 5.6) to emphasise the fact that we apply abstract compilation [6], not just abstract interpretation. The advantages of compilation vs interpretation are well-known;

4. the elements of $R \rightarrow^* R$ can be easily simplified, since their semantics is simple.

In the following, we consider the last point described above, and we provide a simple generic simplification rule for the elements of $R \rightarrow^* R$, which can be improved in specific cases.

### 5.7.1 Reduction rules

A reduction rule transforms every elements of $R \rightarrow^* R$ in an element which represents the same information, without increasing its dimension.

**Definition 5.33.** A reduction rule is a function $\rho : (R \rightarrow^* R) \rightarrow (R \rightarrow^* R)$ such that, for every $A \in R \rightarrow^* R$,
i) \[ \text{unrep}(A) = \text{unrep}(\rho(A)) \];

ii) \[ \text{dim}(A) \geq \text{dim}(\rho(A)) \].

The condition ii means that the time required for computing \( \text{op}(\rho(A)) \) is not greater than the time required for computing \( \text{op}(A) \), as order of magnitude, where \( \text{op} \) is any of the generic operators of Section 5.5. Since, in general, \( \rho(R \rightarrow^E R) \subseteq R \rightarrow^E R \), the number of iterations required for reaching the abstract fixpoint cannot grow by applying \( \rho \) after every iteration of the abstract immediate consequence operator.

We provide now an example of reduction rule, which is generic in the sense of Section 5.5. It is based on the consideration that if an arrow in \( A \in R \rightarrow^E R \) is entailed by another arrow in \( A \cup T \) then it can be removed.

**Definition 5.34.** For any \( A \in R \rightarrow^E R \), we define

\[
\chi(A) = \{ l \rightarrow r \in A \mid \text{there is not } l' \rightarrow r \in A \cup T \text{ such that } l' \subset l \},
\]

where \( T \) is the same set of tautological arrows used in Definition 5.14.

**Proposition 5.35.** The map \( \chi \) is a reduction rule.

The \( \chi \) reduction rule can be applied to any linearly refined domain. If we restrict to the case of a finite set of local variable-related properties whose width is finite, we can provide another reduction rule, which can be applied after \( \chi \). Its goal is to reduce the number of arrows for the distinguished variables. Indeed, two distinguished variables behave exactly in the same way, as a consequence of Proposition 5.20. Therefore, we can represent the behaviour of the distinguished variables by using only \( ?_1 \) in the clauses where just one distinguished variable occurs, by using only \( ?_1 \) and \( ?_2 \) in the clauses where just two distinguished variables occur and so on. This leads to the following definition.

**Definition 5.36.** Let \( P \) be a finite set of variable-related properties whose width is finite. Let \( n = \max \{ \text{width}(p) \mid p \in P \} \) and \( A \in \text{Rep}(P)_V \). Let \( ?(l \rightarrow r) \) be the set of distinguished variables which occur in \( l \rightarrow r \). We define

\[
\psi(A) = (A \setminus \{ a \in A \mid ?(a) \neq \emptyset \}) \cup \cup \{ a[?_1/v_1] \cdots [?_m/v_m] \mid \text{a} \in A, \ ?(a) = \{ v_1, \ldots, v_m \} \}.
\]

**Proposition 5.37.** Under the hypotheses of Definition 5.36, \( \psi \) is a reduction rule.

### 5.8 Conclusions

In this chapter we have described some solutions to typical problems of the linear refinement technique. Namely, we have considered the issue of the precision,
representation and complexity of the refined domains. We have shown that precision can be improved by studying the relationships among abstract properties, in such a way that the refined domains feature useful arrows. Moreover, for many domain refinement operations we have provided corresponding representation refinement operations. This means that a generic representation is available for the linear refinement of a basic domain modelling just the property of interest. We have provided generic algorithms over the representation of every linearly refined domain which implement the concrete operations of the collecting semantics of logic programs, and which feature predictable computational complexity. Finally, we have shown how the time and space complexity of the analysis can be kept as small as possible.

The logical consequence of this work is that the definition and implementation of the linear refinement of a basic domain of analysis is almost automatic now. The user must provide just the abstraction maps \( \alpha^\text{base}_V \) and \( \alpha^\text{global}_V \) of Definition 5.29 and a set of tautological arrows (Definition 5.14). Then he can whether use proper instantiations of the generic algorithms of Section 5.5 or provide optimised versions based on the particular properties of interest. He may as well provide specific reduction rules, based on the semantics of these properties.

As said in Section 5.7, the linear refinement technique might be of great help for keeping the computational complexity of the analysis small. This is because it allows one to push the idea of abstract compilation to its extreme consequences, and allows a compositional, incremental analysis of programs.

Note that the ability of linear refinement to provide a flexible denotation for a procedure, which can be composed in every context and still lead to precise results, is very important when the source code of a module or library is not available, for copyright reasons. In such cases, the copyright holder can provide the module together with its abstract denotation. This would not be possible if the language of denotations were too poor to represent the evolution of the abstract property in every possible context.

The generic algorithms described in Section 5.5 lead to correct but different results, w.r.t. precision, if we substitute their arguments with other representations which still represent the same information. Therefore, an open problem is the definition of generic algorithms which do not feature this behaviour.

In Chapter 6 we consider the instantiation of the framework defined in this chapter to the case of the analysis of non pair-sharing and freeness of variables for pure logic programs. This will prove the usefulness of our framework for the solution of a non trivial case of analysis.
5.9 Proofs

Proofs of Section 5.5

**Proposition 5.15.** Given \(\{A_1, A_2\} \subseteq R - \equiv R\), we have to prove that
\[
\text{unrep}(A_1) \otimes \text{unrep}(A_2) \leq \text{unrep}(A_1 \otimes^{R - \equiv R} A_2),
\]
i.e., that for every \(c_1 \leq \text{unrep}(A_1)\) and \(c_2 \leq \text{unrep}(A_2)\) we have that \(c_1 \otimes c_2 \leq \text{unrep}(A_1 \otimes^{R - \equiv R} A_2)\). Consider an arrow \(l_1 \cdots l_n \rightarrow r \in \rightarrow(A_1, A_2)\). We must prove that \(c_1 \otimes c_2 \leq \text{unrep}(l_1 \cdots l_n \rightarrow r)\). Let \(c \leq \text{unrep}(l_1 \cdots l_n)\). We know that \(c_1 \leq \text{unrep}(l_i \rightarrow r_i)\) for \(i = 1, \ldots, n\), i.e., that \(c_1 \leq \text{unrep}(l_i) \rightarrow \text{unrep}(r_i)\) for \(i = 1, \ldots, n\) and \(c_2 \leq \text{unrep}(r_1 \cdots r_n \rightarrow r)\), i.e., \(c_2 \leq \text{unrep}(r_1 \cdots r_n) \rightarrow \text{unrep}(r)\). We have \(c \otimes (c_1 \otimes c_2) = (c \otimes c_1) \otimes c_2\) by the associativity of \(\otimes\). Since \(c \leq \text{unrep}(l_i)\) and \(c_1 \leq \text{unrep}(r_i)\) for \(i = 1, \ldots, n\), we have \(c \otimes c_1 \leq \bigcap \{\text{unrep}(r_i) \mid i = 1, \ldots, n\}\) = \(\text{unrep}(r_1 \cdots r_n)\), and since \(c_2 \leq \text{unrep}(r_1 \cdots r_n) \rightarrow \text{unrep}(r)\) we conclude that \((c \otimes c_1) \otimes c_2 \leq \text{unrep}(r)\). We can use similar arguments for the other arrows in \(A_1 \otimes^{R - \equiv R} A_2\).

A naive implementation of \(\otimes^{R - \equiv R}\) considers every unfolding of an arrow of \(A_1 \cup T\) with arrows of \(A_2 \cup T\) and vice versa. Therefore, its worst case time complexity is \(O((a_1 + t)(a_2 + t)\dim\text{Kernel})\).

**Proposition 5.17.** Given \(\{A_1, A_2\} \subseteq R - \equiv R\), we have to prove that
\[
\text{unrep}(A_1) \cup \text{unrep}(A_2) \leq \text{unrep}(A_1 \cup^{R - \equiv R} A_2).
\]
For any \(l \rightarrow r \in A_1 \cup^{R - \equiv R} A_2\) we have \(l = l_1l_2\) with \(l_1 \rightarrow r \in A_1\) and \(l_2 \rightarrow r \in A_2\). Then \(\text{unrep}(A_1) \leq \text{unrep}(l_1 \rightarrow r) \leq \text{unrep}(l_1l_2 \rightarrow r)\) and \(\text{unrep}(A_2) \leq \text{unrep}(l_2 \rightarrow r)\). Since this holds for every \(l \rightarrow r \in A_1 \cup^{R - \equiv R} A_2\), we have the thesis.

For the result about complexity, note that a naive algorithm for \(\cup^{R - \equiv R}\) scans the heads of the arrows in \(A_1\) for elements of \(\text{Kernel}\) which belong to the head of some arrow in \(A_2\), and merges their bodies when this happens. Therefore, its complexity is \(O(a_1a_2\dim\text{Kernel})\).

**Proposition 5.19.** Let \(h = \exists_{W,C} c \in 1_V \rightarrow^{H_V} r_V\), \(h' = \exists_{W',C'} c' \in 1_{V\cup W}\) and \(N \in \mathcal{W}\) fresh. We have \(\exists_{W \cup W'} c' [N/x] \in 1_V\) because the properties in \(P\) are local (point i of Definition 5.5). Therefore, we have \(\exists_{W \cup \mathcal{W} \cup N} c' [N/x] \neq H_V \exists_W C \in r_V\) and \(\exists_{W \cup W \cup W'} c' [N/x] \neq H_V \exists_W C \in \exists_{W \cup W \cup W'} \mathcal{M} (c' [N/x] \cup C) = \exists_{W \cup W \cup W'} \mathcal{M} (c' \cup C) [N/x]\), since \(c \in C_V\) and \(x \notin V\). Therefore, \(h \neq H_{V \cup W} h' = \exists_{W \cup W} \mathcal{M} (c' \cup C) \in r_{V \cup W}\) by the locality of the properties in \(P\). This means that \(h \in 1_{V \cup W} \rightarrow^{H_{V \cup W}} r_{V \cup W}\).

\(\square\)
**Proposition 5.20.** Let \(\exists_{W'} c' \in L[v_1/v_2, v_2/v_1]\). By point ii of Definition 5.5, we have \(\exists_{W'} c' \in L[v_1/v_2, v_2/v_1] \in I\). Therefore, we have \(\exists_{W' \cup W} \text{mgu}(c'[v_1/v_2, v_2/v_1] \cup c) = \exists_{W'} c' \in r\). Since \(\{v_1, v_2\} \cap (\text{dom}(c) \cup \text{rng}(c)) = \emptyset\), we have \(\text{mgu}(c'[v_1/v_2, v_2/v_1] \cup c) = \text{mgu}(c \cup c'[v_1/v_2, v_2/v_1])\). By point ii of Definition 5.5, this entails that \(\exists_{W'} c' \in r\) \(\exists_{W'} \text{mgu}(c' \cup c) \in r[v_1/v_2, v_2/v_1]\). The converse holds by symmetry since \(L = L[v_1/v_2, v_2/v_1][v_1/v_2, v_2/v_1]\) and \(R = R[v_1/v_2, v_2/v_1][v_1/v_2, v_2/v_1]\). 

**Proposition 5.22.** We prove that 

\[
\text{expand}_{x}^{(H_v)}(\text{unrep}_{v}(A) \cup U) \subseteq \text{unrep}_{v \cup x}(\text{expand}_{x}^{\text{Rep}(P)}(v)(A)),
\]

where \(U = \{\exists_{W} c \in H_v \mid \{?, \ldots, ?_n\} \cap (\text{dom}(c) \cup \text{rng}(c)) = \emptyset\}\). Note that this result is enough for our purposes, since we are dealing with existential Herbrand constraints in \(U\).

Let \(h \in \text{expand}_{x}^{(H_v)}(\text{unrep}_{v}(A) \cup U)\). Then \(h \in \text{unrep}_{v}(A) \cup U\). Given \(l \rightarrow r \in \text{expand}_{x}^{\text{Rep}(P)}(v)(A)\), we have \(l \rightarrow r \in A\) or \(l \rightarrow r = l'[x/?] \rightarrow r'[x/?]\) for some \(l' \rightarrow r' \in A\). In the first case, we have \(h \in I_{v} \rightarrow l'[v_{x}] \rightarrow r'[v_{x}]\) and by Proposition 5.19 we have \(h \in I_{v \cup x} \rightarrow l'[v_{x}] \rightarrow r'[v_{x}]\). In the second case, we have \(h \in I_{v} \rightarrow l'[v_{x}] \rightarrow r'[v_{x}]\) and, as in the case above, we have \(h \in I_{v \cup x} \rightarrow l'[v_{x}] \rightarrow r'[v_{x}]\). By Proposition 5.20 and the choice of \(h\), we have \(h \in I_{v \cup x} \rightarrow l'[v_{x}] \rightarrow r'[v_{x}]\). Therefore, we have \(h \in \text{unrep}_{v \cup x}(\text{expand}_{x}^{\text{Rep}(P)}(v)(A))\).

For the complexity of the procedure, it suffices to note that a naive algorithm for \(\text{expand}_{x}^{\text{Rep}(P)}(v)(A)\) copies the arrows of \(A\) \((O(\text{dim}(A)))\) and adds new instances of the arrows where \(?_1\) occurs \((O(\text{dim}(A)))\).

**Proposition 5.24.** We have to prove that 

\[
\text{restrict}_{x}^{(H_v)}(\text{unrep}_{v}(A)) \subseteq \text{unrep}_{v \setminus x}(\text{restrict}_{x}^{\text{Rep}(P)}(v)(A)).
\]

Let \(h = \exists_{W} c \in \text{restrict}_{x}^{(H_v)}(\text{unrep}_{v}(A))\). Let \(l \setminus X \rightarrow r \setminus X \in \text{restrict}_{x}^{\text{Rep}(P)}(v)(A)\). It must be \(l \rightarrow r \in A\). Let \(h' = \exists_{W} c' \in H_{Y \setminus X}\) such that \(h' \in l \setminus X\). Since for every \(p(V''') \in l \cap X\), with \(\langle p, t \rangle \in P\) and \(V''' \in t(V)\), we have \(p_{V}(c, V''') = \text{local}\), by point iii of Definition 5.5 we have \(p_{V}(h', V''')\), i.e., \(h' \in p(V''')_{v}\) (Definition 5.11). Moreover, since \(h' = \exists_{W} c' \equiv \exists_{W \cup N} c[N/x]\), from point i of Definition 5.5, we have 

\[
\begin{align*}
\text{if } h' \in \text{p}(V''')_{v} \text{ for every } p(V''') \in l \setminus X \\
\text{if } \text{propery}_{y \setminus x}(h', V''') \text{ for every } p(V''') \in l \setminus X \\
\text{if } \text{propertyl}_{y}(h', V''') \text{ for every } p(V''') \in l \setminus X \\
\text{if } h' \in \text{p}(V''')_{v} \text{ for every } p(V''') \in l \setminus X \\
\text{if } h' \in l \setminus X.
\end{align*}
\]

In conclusion, we have \(h' \in I_{v}\), and since \(h \in I_{v} \rightarrow r_{v}\), we have \(h' \in r_{v}\), i.e., \(h' \in r_{v} \in r \setminus X_{v}\). Since \(\{h, h'\} \subseteq H_{v \setminus x}\), we have \(h' \in r_{v} \).
\( c \) = \( \exists W \cup W(c' \cup c)[N/x] \) and, as done above, from \( h' \overset{H}{\ast} h \in r \setminus X_v \) we conclude \( h' \overset{H}{\ast} h \in r \setminus X_v \). This means that \( h \in \text{unrep}_V(l \setminus X \rightarrow r \setminus X) \). Since this is true for every arrow in \( \text{restrict}_{x}^{\text{Rep}(P)} \), we have the thesis.

For the result about complexity, note that a naive algorithm for \( \text{restrict}_{x}^{\text{Rep}(P)} \) copies the arrows of \( A \) and removes the references to \( x \) (\( O(\text{dim}(A)) \)). \( \square \)

**Proposition 5.26.** We have to prove that

\[
\text{rename}_{x \rightarrow n}^{(H_v)}(\text{unrep}_V(A)) \subseteq \text{unrep}_{V \setminus x}(\text{rename}_{x \rightarrow n}^{\text{Rep}(P)}(A)) .
\]

Let \( h \in \text{rename}_{x \rightarrow n}^{(H_v)}(\text{unrep}_V(A)) \). We have \( h \in H_{(V \setminus x) \setminus n} \) and \( h = \exists W(c[n/x]) \) where \( \exists Wc \in \text{unrep}_V(A) \). Let \( l[n/x] \rightarrow r[n/x] \in \text{rename}_{x \rightarrow n}^{\text{Rep}(P)}(A) \). Let \( h' = \exists Wc' \in l[n/x]_{(V \setminus x) \setminus n} \). We have

\[
\exists Wc' \in l[n/x]_{(V \setminus x) \setminus n}
\]

(point i of Def. 5.5) if \( \exists Wc' \in l[n/x]_{V \setminus n} \)

(point ii of Def. 5.5) if \( \exists Wc'[x/n, n/x] \in l[n/x][x/n, n/x]_{V \setminus n} \)

if \( \exists Wc'[x/n] \in l_{V \setminus n} \).

Since \( \exists Wc \in l_{V \setminus n} \rightarrow r_{V} \), we have \( \exists Wc \overset{H}{\ast} r_{V} \), \( r_{V} \in r_{V} \). This means that \( \exists W \cup W(c'[n/x] \cup c'[x/n] \in r_{V} \). As done above, this entails that \( \exists W \cup W(c[n/x] \cup c') \in r[n/x]_{(V \setminus x) \setminus n} \), i.e., \( h \overset{H}{\ast} r_{(V \setminus x) \setminus n} \), \( h' \in r[n/x]_{(V \setminus x) \setminus n} \). Therefore, we have \( h \in l[n/x]_{(V \setminus x) \setminus n} \rightarrow r[n/x]_{(V \setminus x) \setminus n} \), i.e., \( h \in \text{unrep}_{V \setminus x}(l[n/x] \rightarrow r[n/x]) \).

Since this is true for every arrow in \( \text{rename}_{x \rightarrow n}^{\text{Rep}(P)}(A) \), we have the thesis.

For the result about complexity, note that a naive algorithm for \( \text{rename}_{x \rightarrow n}^{\text{Rep}(P)}(A) \) copies \( A \) by substituting \( n \) for \( x \) (\( O(\text{dim}(A)) \)). \( \square \)

**Proposition 5.28.** By Proposition 2.33 and Propositions 5.26, 5.22 and 5.24. \( \square \)

**Proofs of Section 5.6**

**Proposition 5.30.** We prove by induction on \( n \) that

\[
\{v_1 = t_1, \ldots , v_n = t_n\} \in \text{unrep}_V(\alpha_{V}^{\text{local}}(\{v_1 = t_1, \ldots , v_n = t_n\})) .
\]

If \( n = 0 \) we have \( \varepsilon \in \text{unrep}_V(\alpha_{V}^{\text{local}}(\varepsilon)) \) since given \( h \in k_{V} \), with \( k \in \text{RepKernel}(P)_V \), we have \( h \overset{H}{\ast} \varepsilon = h \in k_{V} \). If the result is true for \( n \), we have \( \{v_1 = t_1\} \in \text{unrep}_V(\alpha_{V}^{\text{base}}(v_1 = t_1)) \) by point i, and \( \{v_2 = t_2, \ldots , v_{n+1} = t_{n+1}\} \in \alpha_{V}^{\text{local}}(\{v_2 = t_2, \ldots , v_{n+1} = t_{n+1}\}) \) by inductive hypothesis. The result follows by Proposition 5.15 (remember that we assume existential Herbrand constraints to be in normal form).
Moreover, since by point ii we have \( c \in \unrep_V(\alpha^{\text{global}}_V(c)) \), we conclude that \( c \in \unrep_V(\alpha^{\text{local}}_V(c) \cup \alpha^{\text{global}}_V(c)) \). By Proposition 5.28 we have

\[
\exists w c \in \unrep_V(\alpha^{\text{alg}}_V(\exists w c)) ,
\]
since \( \{?, \ldots, ?, n\} \cap (\text{dom}(c) \cup \text{rng}(c)) = \emptyset \). This entails that, for every \( h \in H_V \) where \( \{?, \ldots, ?, n\} \) do not occur, we have

\[
\alpha_V(h) = \cap \{d \in \chi \text{Kernel}(P)_V \rightarrow \chi \text{Kernel}(P)_V \mid h \in d\}
\subseteq \cap \{l \rightarrow r \mid l \rightarrow r \in \alpha_{V}^{\text{alg}}(h)\} = \unrep_V(\alpha_{V}^{\text{alg}}(h)) .
\]

\[\Box\]

**Proposition 5.32.** Let \( h \in \unrep_V(A) \) and \( p(V') \in \text{extract}_{\mathcal{P}_V}(A) \). Then \( h \in \mathbb{1} \rightarrow p(V') \) and \( \varepsilon \in \mathbb{1} \). This entails that \( \varepsilon \star^H_V h = h \in p(V') \). Since this is true for every \( p(V') \in \text{extract}_{\mathcal{P}_V}(A) \), we have the thesis. \[\Box\]

**Proofs of Section 5.7**

**Proposition 5.35.** Let \( A \in R \rightarrow^{\mathcal{E}} R \). Condition ii of Definition 5.33 is obviously satisfied since \( \chi \) can only remove arrows. Consider condition i, now. For what we have just said, we have \( \unrep(A) \subseteq \unrep(\chi(A)) \). Let \( c \in \unrep(\chi(A)) \). Consider an arrow \( l \rightarrow r \in A \). If \( l \rightarrow r \in \chi(A) \) then \( c \in \mathbb{1} \rightarrow r \). Otherwise, there must be an arrow \( l' \rightarrow r \in \chi(A) \cup T \) such that \( l' \in l \), i.e., \( \mathbb{1} \supseteq l \). If \( l' \rightarrow r \in \chi(A) \) then \( c \in \mathbb{1} \rightarrow r \subseteq \mathbb{1} \rightarrow r \), otherwise \( c \in C = \mathbb{1} \rightarrow r \subseteq \mathbb{1} \rightarrow r \). In any case we have \( c \in \mathbb{1} \rightarrow r \). Since this is true for every \( l \rightarrow r \in A \), we conclude that \( c \in \unrep(A) \). \[\Box\]

**Proposition 5.37.** The map \( \psi \) removes arrows and, sometimes, substitutes arrows with other arrows of the same dimension. Therefore, we have \( \dim(\psi(A)) \leq \dim(A) \). We want to prove that \( \unrep_V(A) = \unrep_V(\psi(A)) \). Let \( h \in \unrep_V(A) \) and \( l \rightarrow r \in \unrep_V(\psi(A)) \). If \( l \rightarrow r \in A \) then \( h \in \mathbb{1} \rightarrow \mathbb{r} \). Otherwise \( l[v_1/?1] \cdots [v_m/?m] \rightarrow r[v_1/?1] \cdots [v_m/?m] \in A \), where \( ?(l \rightarrow r) = \{?, \ldots, ?, m\} \) and \( \{v_1, \ldots, v_m\} \subseteq \{?, \ldots, ?, m\} \). From \( h \in \unrep_V(A) \) and Proposition 5.20 we have \( h \in \mathbb{1} \rightarrow \mathbb{r} \). Since this is true for every \( l \rightarrow r \in \psi(A) \), we have \( h \in \unrep_V(\psi(A)) \). The other inclusion follows in a similar way. \[\Box\]
In this chapter we use the framework of generic linear refinement defined in Chapter 5 to construct a domain for non pair-sharing and freeness analysis. The resulting domain is strictly more precise than that for sharing and freeness analysis defined by Jacobs and Langen. Moreover, it can be used for abstract compilation, while Jacobs and Langen’s domain can be used only for abstract interpretation. This shows that our framework is strong enough to be applied to non trivial cases of analysis.

Part of this chapter has been published in [59].

6.1 Introduction

Pair-sharing analysis [75, 5] is concerned with determining a superset of the set of pairs of variables which, in a given program point, can be bound to two terms which share some variable. Pair-sharing analysis is a particular case of set-sharing analysis [56, 49, 12, 54, 55, 66, 13], where not only pairs but generic sets of variables are considered. (Pair-)sharing analysis is useful for avoiding occur-check [75] and for automatic program parallelisation [49, 66]. As stressed in [5], pair-sharing information is actually needed in program analysis and transformation, and set-sharing information is redundant w.r.t. pair-sharing information.

Freeness analysis [12, 14, 55, 66, 13, 46] is concerned with determining a subset of variables which are guaranteed to be bound to some variable in a given program point. Freeness analysis is useful for optimising unification, for goal reordering and for avoiding type checking. It is well known that performing sharing and freeness analysis in conjunction improves the precision of both [49, 66].

The first contribution of this chapter is the definition of a domain for non pair-sharing and freeness as linear refinement of a basic domain. The refined domain has various advantages w.r.t. the Sharing ∩ Free domain [56, 49, 66]. First, since it is
defined through a formal and automatic technique, definitions and proofs are very simple, compared to those used for \textit{Sharing} \cap \textit{Free}. Moreover, it is more precise than \textit{Sharing} \cap \textit{Free}, since it contains more precise information about groundness, functor names and linearity. Finally, its representation and algorithms are derived in an automatic way, by using the framework of Chapter 5. As a consequence, our domain can be used for abstract compilation, while \textit{Sharing} \cap \textit{Free} can be used for abstract interpretation only. Another contribution of this chapter is to show, within the linear refinement framework, why and how sharing information interacts with freeness information.

The chapter is organised as follows. Sections 6.2 and 6.3 introduce two basic domains for non pair-sharing and freeness analysis, respectively, and show that their linear refinements do not lead to useful domains. In Section 6.4 we justify this result and in Section 6.5 we combine the two analyses by using a domain which is defined as the linear refinement of a basic domain for both non pair-sharing and freeness. It is shown to be more precise than the domain of [56, 49, 66]. Sections 6.6, 6.7 and 6.8 apply to our basic domain the framework of Chapter 5. Section 6.9 shows an example of abstract computation. We conclude in Section 6.10.

### 6.1.1 Related works

Almost all works about sharing analysis are not amenable for abstract compilation [56, 49, 12, 54, 55, 66]. Moreover, they have been developed without using a generic framework like that of Chapter 5.

To the best of our knowledge, only [18] and [13] provide abstract domains for sharing analysis which can be used for abstract compilation. The domain in [18] is isomorphic to the \textit{Sharing} domain of [56, 49, 66]. This means that, when used for abstract compilation, it must be coupled with a domain for freeness (see Example 6.8). For instance, in Chapter 7 we describe a domain for freeness which can be used for abstract compilation. The domain in [13] models sharing, freeness and groundness. Its precision is comparable to that of the \textit{Sharing} \times \textit{Free} domain of [49, 66].

Our domain leads to a more precise results than those considered above (Proposition 6.16). Moreover, we claim that our construction is more general, since it is just an instantiation of the framework of Chapter 5. We do not need to define partial orderings, representations and correct algorithms, since they are implied by the theory.

### 6.2 Non pair-sharing analysis

Sharing analysis is concerned with determining a superset of the set of program variables which share a variable in a given program point. The usual domain for sharing analysis [49, 56] is defined as $\varphi(\wp(V))$, for any $V \in \wp_f(V)$. For instance,
6.2. Non pair-sharing analysis

letting $V = \{v, x, y, z\}$, the Herbrand constraint $\{x = f(y), z = g(y, v)\}$ is abstracted to the abstract element $\emptyset, \{x, y, z\}, \{v, z\}$, representing the fact that $x$, $y$ and $z$ share $y$ and $v$ and $z$ share $v$.

It has been noted that the applications of sharing analysis always consider sharing information about pairs of variables, and that sharing information about sets of variables is redundant for pair-sharing information [5]. Therefore, in the following we consider pair-sharing information, i.e., we compute a superset of the set of pairs of variables which actually share in a given program point. This is exactly the same as computing a subset of the set of pairs of variables which are guaranteed not to share in a given program point. We prefer this second point of view, since definite information is more intuitive than possible information. The simplest definition of an abstract domain for non pair-sharing coincides with the same property we want to observe.

**Definition 6.1.** Non pair-sharing [5] is a local variable-related property (Definition 5.5) defined as $nsh = \{nsh_V\}_{v \in \forall f(V)} \lambda V.\{\{v_1, v_2\} \mid \{v_1, v_2\} \subseteq V\}$ where, for every $\exists w \in H_V$ and $\{v_1, v_2\} \subseteq V$, $nsh_V(\exists w, \{v_1, v_2\})$ is true if and only if $\text{vars}(c(v_1)) \cap \text{vars}(c(v_2)) = \emptyset$. We have $\text{width}(nsh) = 2$.

**Definition 6.2.** Given $V \in \forall f(V)$, we define

$$NSh_V = \bigcup \text{Kernel}(\{nsh\}_V) .$$

As we said in Definition 5.11, we always remove the tag $nsh$ since it is obvious in the context of this chapter.

**Example 6.3.** Let $V = \{v, x, y, z\}$ and $h = \{x = f(y), z = g(y, v)\}$. We have $h \in (x, v)$ (i.e., $h \in nsh(x, v)$) since $x$ and $v$ do not share any variable in $h$. But $h \notin (z, y)$, since $z$ and $y$ share $y$ in $h$.

Note that $(v, v) = v_g$ (Definition 3.28 and Example 3.10). Indeed, a variable does not share with itself if and only if it is ground.

The domain $NSh_V$ is isomorphic to the domain for pair-sharing defined in [5] as an abstraction of the larger domain for set-sharing analysis presented in [49].

The optimal approximation of concrete conjunction is indeed very poor from the viewpoint of precision. This is because $NSh_V$ is not precise enough to distinguish existential Herbrand constraints whose behaviour, w.r.t. concrete conjunction, is quite different.

**Example 6.4.** The domain $NSh_V$ identifies $\varepsilon$ with $h = \{\exists w | x = f(w, w)\}$ (i.e., they are abstracted into the same element of $NSh_V$). However, these two constraints have very different sharing behaviour when unified, for instance, with $\{x = f(y, z)\}$.

This imprecision was avoided in [5, 49] through the use of a hybrid conjunction procedure, which computes the conjunction of an abstract element with a concrete
element\(^1\) and which allows us to distinguish \(\varepsilon\) from \(h\). This approach has the advantage of providing a more precise result for abstract conjunction than dealing with two abstract elements (the first one and the abstraction of the second). However, it does not allow to apply abstract compilation \([44, 6]\) for sharing analysis. Moreover, the approach of \([5, 49]\), even using the hybrid unification procedure, is sometimes imprecise.

**Example 6.5.** The sharing domains of \([5, 49]\) cannot distinguish between the existential Herbrand constraints \(h_1 = \{x = \mathbf{f}(y), v = \mathbf{f}(z)\}\) and \(h_2 = \{x = \mathbf{f}(y), v = \mathbf{g}(z)\}\). However, they have very different sharing behaviour when unified with \(\{x = v\}\). Namely, the fact that \(y\) and \(z\) are made to share depends on whether \(h_1\) or \(h_2\) is unified with \(\{x = v\}\).

To solve these problems, we linearly refine \(\text{NSh}_V\) w.r.t. concrete conjunction.

### 6.2.1 The domain \(\text{NSh}_V \rightarrow^{\#(\text{H}_V)} \text{NSh}_V\)

Consider the refined domain \(\text{NSh}^1_V = \text{NSh}_V \rightarrow^{\#(\text{H}_V)} \text{NSh}_V\). The following result shows that we cannot use the simplified Equation (1.4) and the generic framework of Chapter 5. This complicates the implementation of the domain.

**Proposition 6.6.** Given \(V \in \varphi_f(V)\) such that \(\#V \geq 3\), we have

\[
\text{NSh}_V \not\subseteq \text{NSh}_V \rightarrow^{\#(\text{H}_V)} \text{NSh}_V.
\]

The domain \(\text{NSh}^1_V\) is precise enough to solve the problem shown by Example 6.4.

**Example 6.7.** Consider Example 6.4. The constraint \(\varepsilon\) belongs to \((\mathbf{y}, \mathbf{z}) \rightarrow (\mathbf{y}, \mathbf{z})\), while \(h\) does not (consider its conjunction with \(\{x = \mathbf{f}(y, z)\}\)).

Since \(\text{NSh}_V \subseteq \text{NSh}^1_V\), the above example shows that \(\text{NSh}^1_V\) is strictly more precise than \(\text{NSh}_V\) for pair-sharing information, which, as proved in \([5]\), is as precise as Jacobs and Langen’s domain \([49]\) w.r.t. pair-sharing information.

However, \(\text{NSh}^1_V\) is not precise enough yet. For instance, it cannot solve the problem shown by Example 6.5. This negative result is not very relevant, since the imprecision shown by Example 6.5 does not affect the usefulness of the analysis (the analysis described in \([56, 66]\) has been used with good results, though it cannot solve this problem). Instead, the following example shows a very great source of imprecision.

**Example 6.8.** Consider \(V = \{v, x, y, z\}\), \(h_1 = \{x = \mathbf{f}(y, z)\}\) and \(h_2 = \{x = v\}\). The optimal approximation \(\ast_{\text{NSh}^1_V}\) of concrete conjunction is such that \(\alpha(h_1) \ast_{\text{NSh}^1_V} \alpha(h_2) \not\subseteq (\mathbf{y}, \mathbf{z})\), i.e., it is not able to conclude that \(y\) and \(z\) do not share in the concrete conjunction of \(h_1\) and \(h_2\) (see the proof at the end of the chapter). Note

\(^1\)Actually, an approximation of concrete conjunction has never been explicitly defined.
that this problem holds for $NSh_V$ and Sharing, too. Indeed, $NSh_V \subseteq NSh^1_V$ and $NSh_V$ is isomorphic to the pair-sharing domain of [5], provably as precise as Sharing w.r.t. pair-sharing information.

This imprecision could be overcome if we knew that $x$ is free in $h_2$. Example 6.8 shows that the freeness of a variable cannot be expressed in $NSh^1_V$. This source of imprecision of $NSh^1_V$ (and, therefore, of $NSh_V$) is the same that led researchers to compute freeness information about program variables in order to improve sharing information [56, 66]. Note that here we have a formal proof of the fact that sharing information needs freeness information for better precision, unless we accept to compute the huge and complex second refinement $NSh^2_V = NSh^1_V \rightarrow^\varphi(h_V) NSh^1_V$ (with no guarantee of solving the problem).

### 6.3 Freeness analysis

Freeness analysis is concerned with determining a subset of program variables which are guaranteed to be bound to variables, in a given program point. The simplest domain for freeness analysis coincides with the same property (Example 5.7).

**Definition 6.9.** Given $V \in \varphi_f(V)$, we define

$$Free_V = \bigcup \text{Kernel} \{\text{freeness}\}_V.$$

As usual, we drop the tag *freeness*.

**Example 6.10.** Consider $V = \{k, x, y, z\}$ and $h = \exists_w \{x = f(y), z = w\}$. We have $h \in kxyz$, since $k$, $y$ and $z$ are free in $h$, but $h \notin kxyz$, since $x$ is not free in $h$.

The domain $Free_V$ is not precise enough for a practical use.

**Example 6.11.** Consider the existential Herbrand constraints $h_1 = \{x = a\}$ and $h_2 = \{y = a\}$ over $V = \{x, y, z\}$. Their unification leaves $z$ free. A useful domain for freeness analysis must capture this behaviour. However, $\alpha(h_1) \star Free_V \alpha(h_2) \not\subseteq z$ (see the proof at the end of the chapter).

We can try to solve this problem by linearly refining the domain $Free_V$.

#### 6.3.1 The domain $Free_V \rightarrow\varphi(h_V) Free_V$

Consider the refined domain $Free^1_V = Free_V \rightarrow\varphi(h_V) Free_V$. The following result shows that we cannot use the simplified Equation (1.4) and the generic framework of Chapter 5. This complicates the implementation of the domain.

**Proposition 6.12.** Given $V \in \varphi_f(V)$ with $\#V \geq 2$, we have

$$Free_V \not\subseteq Free_V \star\varphi(h_V) Free_V.$$
The imprecision of Example 6.11 is not overcome.

**Example 6.13.** Let $h_1$ and $h_2$ be as in Example 6.11. We have $\alpha(h_1) \neq^\text{Free}_V \alpha(h_2) \not\in z$ (see the proof at the end of the chapter).

In Chapter 7 we show how this problem can be overcome by using only freeness information. However, that solution is not general, while in this chapter we want to apply the linear refinement technique in its general form.

### 6.4 A pragmatic look at linear refinement

Example 6.8 shows that the refinement $\text{NSh}^1_V$ of the basic domain for non pair-sharing analysis is not precise enough. Example 6.13 shows that the refinement $\text{Free}^1_V$ of the basic domain for freeness analysis is not precise enough too. If we are interested in improving the precision of non pair-sharing or freeness analysis we have two possibilities. The first is to perform further refinements, hopefully forcing the imprecision to disappear. The second is to apply the suggestion of Section 5.2. This means redesigning the basic domain, by adding the information which is needed in order to obtain useful arrows by refinement.

This last alternative does not look attractive at a first glance. Linear refinement was in fact originally presented as an *automatic* methodology, able to improve the precision of the abstract version of a given concrete operation [32, 40]. Instead, the second approach reintroduces a non methodological choice about the information which is needed in order to obtain more useful arrows. However, we want to convince the reader that in this case the first approach is the wrong one, at least if we are interested in an abstract domain with a *computationally interesting* representation and simple *algorithmic* definitions for its abstract operators.

It is true that the refinement of an already refined domain admits a very simple representation as arrows of arrows. However, this representation is huge and impractical. Moreover, as we perform more refinements, the abstraction function and the abstract operators are more difficult to devise, and computationally more expensive. Note that these remarks are not always true (consider type analysis, for instance).

However, a strong argument against blindly refining a domain can be found if we look at Examples 6.8 and 6.11. Consider $h_1$ and $h_2$ as in Example 6.8. We are not able to write any arrow\(^2\) $a = l \rightarrow (y, z)$ with $l \in \text{NSh}^1_V$, $h_2 \in l$ and $h_1 \in a$, i.e., we cannot provide any sufficient condition for the non sharing of $y$ and $z$ after the conjunction of $h_1$ with another constraint, in such a way that it holds for $h_2$. We would like to write the arrow $x(y, z) \rightarrow (y, z)$ for $h_1$, meaning that if $x$ is free and $y$ and $z$ do not share in $h_2$ then $y$ and $z$ do not share in $h_1 \neq^H_V h_2$. We think that freeness information cannot be expressed as a property of non sharing

\(^2\)The only non tautological arrow for $(y, z)$ correct for $h_1$ is $(x, x) \rightarrow (y, z)$, but $h_2 \not\in (x, x)$ since $x$ is not ground in $h_2$. 
of pairs of variables, however complex it might be. In any case, this question has little practical interest since a very simple solution to the above problem exists, i.e., allowing freeness information to appear in the left of arrows for non pair-sharing information.

Consider $h_1$ and $h_2$ as in Example 6.11 now. Again, we proved that we are not able to write any arrow $a = l \rightarrow z$ with $l \in \text{Free}_V^1$, $h_2 \in l$ and $h_1 \in a$. Therefore, we cannot provide any sufficient condition for the freeness of $z$ after the conjunction of $h_1$ with another constraint, such that it holds for $h_2$. We would like to write the arrow $z(x, z) \rightarrow z$ for $h_1$, meaning that if $z$ is free and $x$ and $z$ do not share in $h_2$ then $z$ is free $h_1 *^r h_2$. We can do this by allowing sharing information to appear in the left of arrows for freeness information.

Formally, the above remarks mean that we want to compute the linear refinement of a basic domain containing both non pair-sharing and freeness information. Actually, this section has provided a reformulation inside the domain refinement framework of the already known result about sharing/freeness interaction [56, 66]. We think that the domain refinement theory gives to this problem a better perspective and a greater generality.

In the following of this chapter, we apply the framework of Chapter 5 to the definition of the linear refinement of a basic domain containing both non pair-sharing and freeness information.

### 6.5 Non pair-sharing and freeness analysis

In the previous section we have shown that it is very natural to define a domain for non pair-sharing and freeness analysis in such a way that its linear refinement is very precise. The definition below formalises this idea.

**Definition 6.14.** Given $V \in \wp_f(V)$, we define

$$
\text{NShFree}_V = \bigcup \text{Kernel}(\text{nsh, freeness})_V
$$

$$
\text{NShFree}_V^1 = \text{NShFree}_V \triangleright^* \text{NShFree}_V^1
$$

The result below is relevant since it allows us to work with arrows only, rather than with arrows and basic elements.

**Proposition 6.15.** Given $V \in \wp_f(V)$, $\text{NShFree}_V \subseteq \text{NShFree}_V \triangleright^* \text{NShFree}_V^1$.

As a consequence, $\text{NShFree}_V \subseteq \text{NShFree}_V^1 = \text{NShFree}_V \triangleright^* \text{NShFree}_V^1$. In general, this inclusion is strict.

**Proposition 6.16.** Given $V \in \wp_f(V)$, $\text{NShFree}_V^1$ contains information about functions and linearity, and more precise groundness information than $\text{NShFree}_V$.
Note that all the above kinds of information are relevant for non pair-sharing analysis. For instance, we do not need to know that \( x \) and \( y \) do not share in \( h \) in order to conclude that \( v \) and \( v' \) do not share in \( h \star h' \{ x = f(v), y = g(v') \} \), provided that \( x \) and \( y \) are free\(^3\) in \( h \). The importance of groundness information for non pair-sharing analysis is obvious. If \( x \) is ground then it does not share with any variable. Linearity information is useful too. For instance, even if we know that \( x \) is free and \( y \) and \( z \) do not share in some constraint \( h \), we cannot conclude that \( y \) and \( z \) do not share in \( h \star h' \) \( h_1 \), where \( h_1 = \exists_w \{ x = f(y, z), v = g(w, w) \} \) (consider the case \( h = \{ v = g(y, z) \} \), while we can, if we take \( h_2 = \exists_{w_1, w_2} \{ x = f(y, z), v = g(w_1, w_2) \} \) in \( h \star h' \) \( h_2 \). In \( \text{NShFree}_v \), this is captured by the fact that \( h_1 \not\subseteq x(y, z) \rightarrow (y, z) \), while \( h_2 \in x(y, z) \rightarrow (y, z) \).

### 6.6 Representation and algorithms

A representation of \( \text{NShFree}_v \) is \( \text{Rep}(\{ \text{nsh, freeness} \})_v \), following Definition 5.18. We write \( \text{Rep}_v \) for \( \text{Rep}(\{ \text{nsh, freeness} \})_v \).

Since \( \text{nsh} \) has width 2 and \( \text{freeness} \) has width 1, we use two distinguished variables \( ?_1 \) and \( ?_2 \). Since both non pair-sharing and freeness are local variable-related properties, we can instantiate the algorithms of Section 5.5, obtaining the following correct algorithms.

**Definition 6.17.** Let \( V \in \wp_f(V) \) and \( \{ A, A_1, A_2 \} \subseteq \text{Rep}_V \). The instantiation of the algorithms of Section 5.5 are

\[
A_1 \star_{\text{Rep}_V} A_2 = \neg \neg (A_1, A_2) \cup \neg (A_2, A_1),
\]

where\(^4\)

\[
\neg \neg (B_1, B_2) = \left\{ l_1 \cdots l_n \rightarrow r \mid \begin{array}{l} r_1 \cdots r_n \rightarrow r \in B_2 \cup T \\ l_i \rightarrow r_i \in B_1 \cup T \text{ for every } i = 1, \ldots, n \end{array} \right\}
\]

and \( T = \{ (v_1, v_1) \rightarrow (v_1, v_2) \mid \{ v_1, v_2 \} \subseteq V \} \cup \{ (v, v) \rightarrow (v, v) \mid v \in V \} \cup \{ v(v, v) \rightarrow (v_1, v_2) \mid \{ v, v_1, v_2 \} \subseteq V \} \). Note that the arrows in \( T \) are tautological since, when a variable does not share with itself, it must be ground and, therefore, it cannot share with any other variable. Moreover, a ground variable cannot lose its groundness.

\(^3\)The freeness of \( x \) and \( y \) is necessary. Indeed, consider the cases \( h = \{ y = g(x) \} \) and \( h = \{ x = f(y) \} \) (Thanks to Francesca Scozzari for this observation).

\(^4\)The map \( \star_{\text{Rep}_V} \) is commutative. Therefore, we do not need the leftward arrows.
Finally, it is not possible that a variable is ground and free at the same time.

\[
\exp_{x}^{\text{Rep}_{V}}(A) = A \cup \{ [x/\_1] \mapsto r [x/\_1] \mid l \mapsto r \in A \},
\]

\[
\rest_{x}^{\text{Rep}_{V}}(A) = \{ l \setminus X \mapsto r \setminus X \mid l \mapsto r \in A, \ r \setminus X \neq \emptyset \text{ and } (v, v) \not\in \{ \}
\text{ where } X = \{ x \} \cup \{(x, v) \mid v \in V \},
\]

\[
\rename_{x \rightarrow n}^{\text{Rep}_{V}}(A) = A [ n / x ],
\]

\[
\exists_{x}^{\text{Rep}_{V}}(A) = \rest_{x}^{\text{Rep}_{V \cup n}}(\exp_{x}^{\text{Rep}_{V \cup n}}(\rename_{x \rightarrow n}^{\text{Rep}_{V}}(A))) \cup \text{Rep}_{V}(A_{1}, A_{2}) = \{ l_{1}l_{2} \mapsto r \mid l_{1} \mapsto r \in A_{1} \text{ and } l_{2} \mapsto r \in A_{2} \}.
\]

### 6.7 Abstraction and information extraction

#### 6.7.1 Abstraction

Let \( v = t \), with \( v \in V \) and \( t \in \text{terms}(\Sigma, V) \), be a single binding. Its abstraction consists in three sets of arrows, i.e., arrows for groundness, arrows for non pair-sharing and arrows for freeness. Note that the arrows for groundness are a particular case of arrows for non pair-sharing, but it is simpler to deal with them separately.

**Definition 6.18.** Given \( V \in \phi_{f}(\mathcal{V}) \), \( v \in V \) and \( t \in \text{terms}(\Sigma, V) \), we define

\[
\alpha_{V}^{\text{base}}(v = t) = \alpha_{V}^{\text{ground}}(v = t) \cup \alpha_{V}^{\text{nsh}}(v = t) \cup \alpha_{V}^{\text{free}}(v = t).
\]

We write \( t(v_{1}, \ldots, v_{n}) \) for a term \( t \) whenever \( \text{vars}(t) = \{ v_{1}, \ldots, v_{n} \} \). When \( n = 0 \) then \( t(v_{1}, \ldots, v_{n}) \) is a ground term. We assume variables with different names to be different variables. We define

\[
\alpha_{V}^{\text{ground}}(v = t(v_{1}, \ldots, v_{n})) = \begin{cases} (v_{1}, v_{1}) \ldots (v_{n}, v_{n}) \mapsto (v, v), \\ (v, v) \mapsto (v_{1}, v_{1}), \ldots \\ (v, v) \mapsto (v_{n}, v_{n}) \end{cases} \quad n \geq 0,
\]

\[
\alpha_{V}^{\text{nsh}}(x = t) = \cup \{ v, v' \in V \alpha_{v, v'}^{\text{nsh}}(x = t)
\]

\[
\alpha_{v, v'}^{\text{nsh}}(v = t(v', v_{1}, \ldots, v_{n})) = \{ \}
\quad n \geq 0
\]

\[
\alpha_{v, v'}^{\text{nsh}}(x = t(v', v_{1}, \ldots, v_{n})) = \{ x(v, v') \mapsto (v', v'), (x, x) \mapsto (v, v') \}
\quad n \geq 0
\]

\[
\alpha_{v, v'}^{\text{nsh}}(v = t) = \{ \mapsto (v, v') \}
\quad \text{t ground}
\]

\[
\alpha_{v, v'}^{\text{nsh}}(v = t(v_{1}, \ldots, v_{n})) = \{ (v', v)(v'_{1}, v_{1}) \ldots (v', v_{n}) \mapsto (v', v) \}
\quad n \geq 1
\]

\[
\alpha_{v, v'}^{\text{nsh}}(x = t(v_{1}, \ldots, v_{n})) = \{ (v', v)(v', v_{1})(v'_{1}, v_{1}) \ldots (v', v_{n}) \mapsto (v', v) \}
\quad n \geq 0
\]

\[
\alpha_{v, v'}^{\text{nsh}}(x = t) = \{ (v, v') \mapsto (v, v') \}
\quad \text{t ground}
\]

\[
\alpha_{v, v'}^{\text{nsh}}(x = t(v_{1}, \ldots, v_{n})) = \begin{cases} (v, v')(v, v_{1})(v_{1}, v_{1}) \ldots (v, v_{n}) \mapsto (v, v'), \\ (v, v')(v', v_{1})(v'_{1}, v_{1}) \ldots (v', v_{n}) \mapsto (v, v') \end{cases} \quad n \geq 1
\]
Chapter 6. Non pair-sharing and freeness

\[
\alpha^\text{free}_v(x = t) = \bigcup_{v \in V} \alpha^\text{free}_v(x = t)
\]

\[
\alpha^\text{free}_v(v = x) = \{vx \rightarrow v\}
\]

\[
\alpha^\text{free}_v(v = t(v_1, \ldots, v_n)) = \{\} \quad t(v_1, \ldots, v_n) \not\in V, \ n \geq 0
\]

\[
\alpha^\text{free}_v(x = t(v, v_1, \ldots, v_n)) = \{vx \rightarrow v\} \quad n \geq 0
\]

\[
\alpha^\text{free}_v(x = y) = \{v(x, v)(y, v) \rightarrow v, vxy \rightarrow v\}
\]

\[
\alpha^\text{free}_v(x = t(v_1, \ldots, v_n)) = \{v(x, v)(v_1, v) \cdots (v_n, v) \rightarrow v, vx(x, v) \rightarrow v\}
\]

Condition i of Proposition 5.30 is satisfied.

**Lemma 6.19.** Given \(V \in \wp_f(V), z \in V\) and \(t \in \text{terms}(\Sigma, V)\), we have \(\{z = t\} \in \text{unrep}_V(\alpha^\text{base}_V(z = t))\).

The map \(\alpha^\text{global}_V\) considers global dependencies, i.e., dependencies which are lost if we consider each binding in isolation. These dependencies are originated by syntactical properties of a constraint \(c \in C_V\) (see Proposition 6.16 and its proof).

**Definition 6.20.** Given \(V \in \wp_f(V)\) and \(c \in C_V\) we define

\[
\alpha^\text{global}_V(c) = \{\text{global}_c(l) \rightarrow r \mid l \rightarrow r \in \alpha^\text{local}_V(c)\},
\]

where \(\alpha^\text{local}_V\) is defined in Definition 5.29 and

\[
\text{global}_c(e_1 \cdots e_n) = \text{global}_c(e_1) \cdots \text{global}_c(e_n)
\]

\[
e_i \in V \cup \{(v_1, v_2) \mid \{v_1, v_2\} \subseteq V\} \quad \text{for} \quad i = 1, \ldots, n,
\]

\[
\text{global}_c(v) = v \quad v \in V
\]

\[
\text{global}_c((v_1, v_2)) = \begin{cases} 
    v_1v_2 & \text{if mgu}(c(v_1), c(v_2)) \text{ does not exist} \\
    (v_1, v_2) & \text{otherwise.}
\end{cases}
\]

Condition ii of Proposition 5.30 is satisfied.

**Lemma 6.21.** Given \(V \in \wp_f(V)\) and \(c \in C_V\), we have \(c \in \text{unrep}_V(\alpha^\text{global}_V(c))\).

By using Proposition 5.30, we conclude that the map \(\alpha^\text{alg}_V\) of Definition 5.29 is correct.

### 6.7.2 Information extraction

Consider now the problem of computing the set of variables which are free and the set of pairs of variables which do not share in every existential Herbrand constraint in \(\text{unrep}_V(A)\), with \(A \in \text{Rep}_V\). These two sets can be under approximated by the maps \(\text{extract}_\text{free}_V\) and \(\text{extract}_\text{rsh}_V\), obtained by instantiating Definition 5.31.
Definition 6.22. The maps \(\text{extract\_free}_V\) and \(\text{extract\_nsh}_V\) are obtained from Definition 5.31. They are

\[
\text{extract\_free}_V(A) = \{ v \in V \mid l \to v \in A \text{ and } (v',v') \not\subseteq l \text{ for any } v' \in V \},
\]

\[
\text{extract\_nsh}_V(A) = \{ (v_1,v_2) \mid l \to (v_1,v_2) \in A \text{ and } (v,v) \not\subseteq l \text{ for any } v \in V \}.
\]

Note that the definition of \(\text{extract\_free}\) can be improved.

Proposition 6.23. Let \(V \in \wp_f(V)\), \(A \in \text{Rep}_V\) and

\[F = \{ v \in V \mid l \to v \in A \text{ and } v'(v',v') \not\subseteq l \text{ for any } v' \in V \}.
\]

We have \(\text{unrep}_V(A) \subseteq F\).

The hypothesis on \(l\) is necessary. Indeed, consider \(A = \{ xv(v,v) \to x \}\). We have \(\text{unrep}_V(A) \not\subseteq x\), since \(\text{unrep}_V(A) = H_V\), being \(xv(v,v) \to x\) a tautological arrow.

6.8 Reduction rules

We know that the reduction rule of Definition 5.34 is correct for every linearly refined domain. We instantiate that rule to the case of non pair-sharing and freeness, by using the set \(T\) defined in Definition 6.17. Similarly, we instantiate the reduction rule for distinguished variables provided by Definition 5.36.

Definition 6.24. Let \(V \in \wp_f(V)\) and \(A \in \text{Rep}_V\). The instantiation of Definitions 5.34 and 5.36 to the case of non pair-sharing and freeness is

\[
\chi(A) = \begin{cases} 
0 \to r \in A, \\
1 \to v \in A \text{ if } V \not\subseteq l, \\
2 \to v \not\subseteq l, \\
3 \to v \not\subseteq l, \\
4 \to v \not\subseteq l, \\
5 \to v \not\subseteq l.
\end{cases}
\]

\[
\psi(A) = (A \setminus \{ a \in A \mid ?(a) \neq \emptyset \}) \cup \{ a[?_1/?_2] \mid a \in A \text{ and } ?(a) = \{?_2\} \}.
\]

It could seem that a further reduction rule, specific to the case of non pair-sharing and freeness, would be that of substituting \(l \to v\) with \(l \setminus \{(v',v') \mid v' \in V\} \to v\). The idea is that groundness cannot entail freeness. However, this intuition is wrong. Indeed, groundness does help freeness in a Herbrand constraint like \(h = \{ x = f(a) \}\). Namely, we have \(h \in \wp(v(x,x) \to v)\), while \(h \not\subseteq v \to v\). This is made apparent if we consider the conjunction \(\{ x = f(v) \} \times h\), where \(v\) loses its freeness as a consequence of the non groundness of \(x\).
Figure 6.1: The abstract constraint $\chi \left(\alpha_{[1, 2, x, y, z]}^{ad_q}(\{x = f(y, z)\})\right)$.
Figure 6.2: The abstract constraint \( \psi \chi \left( \alpha_{\{?1,?2,?x,?y,?z\}}^{\text{alg}} \{ \{x = f(y, z)\} \} \right) \).

Figure 6.3: The abstract constraint \( \psi \chi \left( \alpha_{\{?1,?2,?x,?v\}}^{\text{alg}} \{ \{x = v\} \} \right) \).
\[
\psi_X \left( \alpha_{\text{alg}} \left[ \mathcal{X}^{\text{rep}}_{\{1,1,x,y,z\}} \right] \right) = \psi_X \left( \alpha_{\text{alg}} \left[ \mathcal{X}^{\text{rep}}_{\{1,1,x,y,z\}} \right] \right) = \psi_X \left( \alpha_{\text{alg}} \left[ \mathcal{X}^{\text{rep}}_{\{1,1,x,y,z\}} \right] \right) = \psi_X \left( \alpha_{\text{alg}} \left[ \mathcal{X}^{\text{rep}}_{\{1,1,x,y,z\}} \right] \right)
\]
Figure 6.5: The abstract constraint $\psi^\text{Rep}_{\{y,z\}}(\alpha^\text{Alg}_{\{?_1?, ?_2?, x, v\}}(\{x = v\}))$.

Figure 6.6: Some arrows of the conjunction.
6.9 Example

We have implemented the $Rep_V$ domain and its algorithms through a simple Prolog program which can be found at the address

http://www.di.unipi.it/~spoto/nshfree.pl

We use this program to show an example of computation over $Rep_V$.

Consider $V_1 = \{?_1, ?_2, x, y, z\}$ and $h_1 = \{x = f(y, z)\}$. The abstraction $\alpha_{V_1}^{adg}(h_1)$ contains 3851 arrows. Its reduction through $\chi$ yields the 60 arrows of Figure 6.1. Its further reduction through $\psi$ yields the 41 arrows of Figure 6.2.

Consider $V_2 = \{?_1, ?_2, x, v\}$ and $h_2 = \{x = v\}$. The abstraction $\alpha_{V_2}^{adg}(h_2)$ contains 805 arrows. Its reduction through $\chi$ contains 29 arrows and its further reduction through $\psi$ yields the 20 arrows of Figure 6.3.

We want to compute the conjunction of $h_1$ and $h_2$. First of all, we need to expand both constraints to the minimal common set of variables. This is accomplished through the $expand^{Rep_V}$ map. In Figures 6.4 and 6.5 we show the results of these expansions. We expand $h_2$ first w.r.t. $y$ and then w.r.t. $z$. This second expansion adds more arrows than the first one since some new arrows for $y$ contain the distinguished variable $?_1$. Note that this allows to obtain arrows having $(y, z)$ in their head. We can compute the conjunction of these expansions now. Let us call $A$ the result. The constraint $A$ contains, among others, the arrows shown in Figure 6.6. From Definition 6.22 we have

\[
extract\_free_{\{?_1, ?_2, x, y, z\}}(A) \supseteq \{y, z\} \\
extract\_nsh_{\{?_1, ?_2, x, y, z\}}(A) \supseteq \{(y, z)\}
\]

This means that in $h_1 \star^H h_2$ the variables $y$ and $z$ are free and do not share. Remember that this result could not be obtained by using the unrefined domain $NSh_V$ (Example 6.8), unless we accept to use the hybrid unification procedure of [5, 49]. Instead, our approach has been able to obtain this result by using conjunction between two abstract objects.

6.10 Conclusions

We have defined a domain for non pair-sharing and freeness analysis which is more precise than the standard domain defined in [49, 66] and can be used for abstract compilation. Beyond the precision issue, the main contribution of this chapter has been to show the applicability of linear refinement and of the framework of Chapter 5 to a difficult case of program analysis.

It would be interesting to study more precise abstract algorithms and abstraction map than those derived automatically by using the framework of Chapter 5. Our example (Section 6.9) has shown that the reduction rules play a fundamental role
in keeping the abstract objects small. Future research should be oriented towards the definition of more powerful reduction rules than those used in this chapter.
6.11 Proofs

Proofs of Section 6.2

**Proposition 6.6.** Consider \( \{x, y\} \subseteq V \), with \( x \neq y \). We show that the point \((x, y) \in \text{NSh}_V \) is not contained in \( \text{NSh}_V \times^{a(h_V)} \text{NSh}_V \). Every point of this last set is the intersection \( \cap_{i \in I} a_i \), with \( I \subseteq \mathbb{N} \), of a set of arrows having their right hand side in \( \{(v_1, v_2) \mid (v_1, v_2) \subseteq V\} \). We can assume without any loss of generality that this intersection is minimal, i.e., that no \( a_i \) can be left out. We prove that \( \cap_{i \in I} a_i \neq (x, y) \). Assume by contradiction that \( \cap_{i \in I} a_i = (x, y) \).

As a first step, we prove that there is no \( a_i \) such that \( a_i = 1 \rightarrow r \) with \( r \neq (x, y) \). Indeed, assume, without any loss of generality, that \( a_i = 1 \rightarrow (v_1, v_2) \), with \( v_1 \not\in \{x, y\} \). If \( 1 \subseteq (v_1, v_1) \) or \( 1 \subseteq (v_2, v_2) \) then the arrow would be tautological and could be left out. Since this is a contradiction, we can assume \( 1 \not\subseteq (v_1, v_1) \) and \( 1 \not\subseteq (v_2, v_2) \). Let \( h = \{v_1 = v_2\} \). We have \( h \in (x, y) \) by the hypothesis \( v_1 \not\in \{x, y\} \). Let \( h' = \{v = a \mid 1 \subseteq (v, v)\} \). The constraint \( h' \) makes ground exactly those variables which are required to be ground by \( l \). Note that, as said before, \( v_1 \) and \( v_2 \) are unbound in \( h' \). As a consequence, we have that \( h \times^{a(h)} h' = h' \cup \{v_1 = v_2\} \not\in (v_1, v_2) \).

Since \( h' \in l \) by construction, we have \( h \not\in a_i \). But \( h \in (x, y) \). This proves that every \( a_i \) must have \((x, y)\) as its right hand side.

We prove now that every non tautological arrow \( l \rightarrow (x, y) \) which contains \((x, y)\) must be such that \( l \subseteq (v, v) \) for every \( v \in V \setminus \{x, y\} \). Indeed, assume \( l \not\subseteq (v, v) \) with \( v \in V \setminus \{x, y\} \). If \( l \subseteq (x, x) \) or \( l \subseteq (y, y) \) then \( a_i \) would be tautological. Therefore, we can assume \( l \not\subseteq (x, x) \) and \( l \not\subseteq (y, y) \). Consider \( h = \{v = f(x, y)\} \subseteq (x, y) \) and \( h' = \exists_{\{w\}} \{(v = f'(w, w)) \cup \{v' = a \mid 1 \subseteq (v', v')\}\} \). We have \( h' \in l \) by the hypothesis on \( v \). Since \( h \times^{a(h)} h' = \{v = f(x, x), x = y\} \cup \{v' = a \mid 1 \subseteq (v', v')\} \), we have \( h \times^{a(h)} h' \not\subseteq (x, y) \). This shows that \( h \not\in a_i \). This is a contradiction since \((x, y) \subseteq a_i \) and \( h \in (x, y) \).

In conclusion, every \( a_i \) must have the form \( l \rightarrow (x, y) \) with \( l \subseteq (v, v) \) for every \( v \in V \setminus \{x, y\} \). Moreover, we can assume \( l \not\subseteq (x, x) \) and \( l \not\subseteq (y, y) \), otherwise \( a_i \) would be tautological. Let \( v \in V \setminus \{x, y\} \). We can find such a \( v \) since \#V \geq 3. Let \( h = \{v = x, y = x\} \not\subseteq (x, y) \). Given \( h' \in l \), since \( v \) is ground in \( h' \), we conclude that if \( h \times^{a(h)} h' \) exists then \( h \times^{a(h)} h' \in (x, y) \). Therefore, we have \( h \in \cap_{i \in I} a_i \). This proves that \( \cap_{i \in I} a_i \neq (x, y) \). \( \square \)

**Example 6.8.** Assume by contradiction that \( a(h_1) \times^{\text{NSh}_V} a(h_2) \subseteq (y, z) \). By definition of \( \times^{a(h_V)} \), this means that \( a(h_1) \subseteq (a(h_2) \rightarrow (y, z)) \). Therefore, every \( h \in a(h_2) \) is such that its concrete conjunction with \( h_1 \) does not make \( y \) and \( z \) to share. Since\(^5\) \( a(h_2) = \cap A \), where \( A = \{(v, v), (v, z), (x, y), (x, z), (y, z), (x, x) \rightarrow (v, v), (v, v) \rightarrow (x, x), (v, y)(x, y) \rightarrow (v, y), (x, x) \rightarrow (v, y), (v, z)(x, z) \rightarrow (v, z), (x, x) \rightarrow (v, z), (x, y)(v, y) \rightarrow (x, y), (v, v) \rightarrow (x, y), (v, y) \rightarrow (x, y), (v, z)(v, z) \rightarrow (v, z)\} \), we can conclude that if \( h \times^{a(h)} h' \) exists then \( h \times^{a(h)} h' \in (x, y) \). Therefore, we have \( h \in \cap_{i \in I} a_i \). This proves that \( \cap_{i \in I} a_i \neq (x, y) \).

---

\(^5\)This abstraction has been computed by enumeration, by removing those elements that are incorrect for \( h_2 \), or redundant, or tautological, and by proving that the others are correct for \( h_2 \).
We can assume without any loss of generality that this intersection is minimal, i.e., that no contained in \( y = z \) and its unification with \( h_2 \) leaves \( z \) non free.

**Proofs of Section 6.3**

**Example 6.11.** In \( \text{Free}_V \), \( h_1 \) is abstracted to \( yz \) and \( h_2 \) to \( xz \). We have \( yz \not\in \text{Free}_V \) which is not contained in \( z \) (this is because \( \{x = a, y = z\} \) is abstracted to \( yz \) as \( h_1 \), and its unification with \( h_2 \) leaves \( z \) non free).

**Proposition 6.12.** Consider \( x \in V \). We show that the point \( x \in \text{Free}_V \) is not contained in \( \text{Free}_V \cap \text{Free}_V \). Every point of this last set is the intersection \( \cap_{i \in I} a_i \), with \( I \subseteq \mathbb{N} \), of a set of arrows having their right hand side in \( \{v \mid v \in V\} \). We can assume without any loss of generality that this intersection is minimal, i.e., that no \( a_i \) can be left out. We prove that \( \cap_{i \in I} a_i = x \). Assume by contradiction that \( \cap_{i \in I} a_i = \neq x \).

As a first step, we prove that there is no \( a_i \) such that \( a_i = \rightarrow v \) with \( v 
eq x \).

Indeed, in such a case we would have \( h = \{v = a\} \not\subseteq a_i \), since \( \varepsilon \in I \) and \( \varepsilon \preceq h = h \neq v \). But \( h \in x \), which is a contradiction since \( x \subseteq a_i \). Therefore, every \( a_i \) must have the form \( \rightarrow x \) for some \( l \in \text{NSh}_V \). Since \( \# V \geq 2 \), there exists \( y \in V \), \( y \neq x \). Consider \( h = \{y = a\} \in x \). We have \( h \not\subseteq a_i \) since \( \{y = x\} \in I \) and \( \{y = x\} \preceq h = \{x = a, y = a\} \not\subseteq x \). Therefore, we cannot have \( x \subseteq a_i \). This proves that \( I \) is empty, i.e., \( \cap_{i \in I} a_i = H_v \not= x \), a contradiction.

**Example 6.13.** The constraint \( h_1 \) belongs to \( yz \). We want to show that \( yz \) is the abstraction of \( h_1 \) in \( \text{Free}_V^1 \). Indeed, if \( h_1 \in l \rightarrow r \) and \( l \rightarrow r \neq H_v \), then \( r \neq H_v \). Since \( h_3 = \{x = y, z = y\} \) belongs to every element of \( \text{Free}_V \) and therefore to \( l \), we would have \( h_3 \preceq h_1 \in r \), which is a contradiction, since \( r \neq H_v \) and all three variables are made non free by the unification. Then \( h_1 \) is abstracted to \( yz \) in \( \text{Free}_V^1 \). By symmetry, \( h_2 \) is abstracted to \( xz \) in \( \text{Free}_V^1 \). The same argument used in Example 6.11 in the case of \( \text{Free}_V \) shows that the best correct approximation of the \( \star^{H_v} \) operation does not belong to \( z \) when applied to \( \alpha(h_1) \) and \( \alpha(h_2) \). Therefore, \( \text{Free}_V \) is useless in practice.

**Proofs of Section 6.5**

**Proposition 6.15.** Let \( F = V \cap A \), where \( A = \cap \{(v_1, v_2) \mid \{v_1, v_2\} \subseteq V, v_1 \neq v_2\} \). It can be easily seen that \( F = \{\varepsilon\} \). Since \( F \in \text{NShFree}_V \), we have that \( F \rightarrow e = e \in \text{NShFree}_V^1 \) for every \( e \in \text{NShFree}_V \), and the inclusion follows.

**Proposition 6.16.**
\(NShFree^1\) contains information about functors. Consider \(h_1\) and \(h_2\) as in Example 6.5. They cannot be distinguished by \(NShFree^1\). However, \(h_2\) belongs to the arrow \(\xv(x, z)(y, v)(y, z) \to (y, z)\), while \(h_1\) does not\(^6\).

\(NShFree^1\) contains better groundness information than \(NShFree^g\). Assume \(V = \{x, y, v\}\). The set of existential Herbrand constraints such that \(x\) is ground or \(y\) is ground does not belong to \(NSh^V\) \([24]\). The same holds for \(NShFree^g\), as it can be checked by case analysis (freeness information is of no help). However, this point belongs to \(NShFree^1\), since

\[
H_V \rightarrow (x, y) = x_g \cup y_g.
\]

Remember that \(v_g = \{\exists w, c \in H_v \mid \text{vars}(c(v)) = \emptyset\}\) (Definition 3.28 and Example 3.10). Indeed, assume that \(h = \exists w c \in H_v \rightarrow (x, y)\). The variables \(x\) and \(y\) cannot belong to \(\text{rng}(c)\), otherwise the constraint \(\text{h}^\ast \rightarrow \text{h}^\ast\) \(\text{h}\) is defined and makes \(x\) and \(y\) to share. The same argument can be used to show that both \(x\) and \(y\) must belong to \(\text{dom}(c)\). Moreover, since \(\varepsilon \in H_v\), we must have \(\text{vars}(c(x)) \cap \text{vars}(c(y)) = \emptyset\). Assume \(v_x \in \text{vars}(c(x)), v_y \in \text{vars}(c(y))\) and let \(h' = \exists w \{x = c(x)[w/v_x] ; y = c(y)[w/v_y]\}. We have that \(h' \rightarrow h\) is defined and makes \(x\) and \(y\) to share. Therefore, the only possibility if that \(x\) and \(y\) belong to \(\text{dom}(c)\) and \(c(x)\) or \(c(y)\) is ground. Conversely, given \(\exists w c \in H_v\) such that \(c(x)\) or \(c(y)\) is ground, we have \(\exists w c \in H_v \rightarrow (x, y)\).

\(NShFree^1\) contains linearity information. Let \(V \in \varphi_f(V)\), \(\{v, x, y\} \subseteq V\), \(h_1 = \exists_{w_1, w_2} \{v = f(w_1, w_2)\}\) and \(h_2 = \exists_{w} \{v = f(w, w)\}\). The constraint \(h_1\) binds \(v\) to a linear term, while \(h_2\) binds \(v\) to a non-linear term. The constraints \(h_1\) and \(h_2\) cannot be distinguished in \(NShFree^g\). However, in \(NShFree^1\) we have \(h_1 \in (x, y) \rightarrow (x, y)\) and \(h_2 \not\in (x, y) \rightarrow (x, y)\).

\(\square\)

## Proofs of Section 6.7

**Lemma 6.19.** It suffices to prove that every arrow \(1 \rightarrow r \in \alpha_V^{\text{ground}}(z = t) \cup \alpha_V^{\text{sh}}(z = t) \cup \alpha_V^{\text{free}}(z = t)\) is such that \(\{z = t\} \in 1 \rightarrow r\).

This is true if \(1 \rightarrow r \in \alpha_V^{\text{ground}}(z = t)\), since \((v, v) = v_g\) and \(\alpha_V^{\text{ground}}(z = t)\) contains obvious groundness dependencies.

For the arrows contained in \(\alpha_V^{\text{sh}}\) and \(\alpha_V^{\text{free}}\), given \(h = \exists w c \in 1\), we have \(h \rightarrow (z = t) = \exists w \text{mgu}(c \text{mgu}(c(z) = tc))\), if it is defined. Let \(c' = \text{mgu}(c(z) = tc)\). Since \(\text{dom}(c') \cap \text{dom}(c) = \emptyset\), we have \(h \rightarrow (z = t) = \exists c(c')\), if it exists. Assume it exists. We consider every single arrow contained in \(\alpha_V^{\text{sh}}(z = t) \cup \alpha_V^{\text{free}}(z = t)\) and we prove that \(h' = \exists w(c c') \in r\). In this proof, if \(t(v_1, \ldots, v_n)\) is a term, we write \(t(t_1, \ldots, t_n)\) for \(t(v_1, \ldots, v_n)[t_1/v_1] \cdots [t_n/v_n]\).\(^6\)

\(^6\)It does actually belong to \((x, y)(x, z)(y, v)(y, z) \rightarrow (y, z)\).
Consider \( x(v, v') \rightarrow (v, v') \in \alpha_{V}^{nsh}(x = t(v, v', v_1, \ldots, v_n)) \), with \( n \geq 0 \). Since \( c(x) \in V \), we have \( c' = \{ c(x) = t(c(v), c(v'), c(v_1), \ldots, c(v_n)) \} \). Note that \( c(x) \notin \text{vars}(c(v)) \cup \text{vars}(c(v')) \), since otherwise \( c' \) and \( h' \) would not exist. Therefore, we have \( (cc')(v) = c(v)c = c(v) \) and \( (cc')(v') = c(v')c = c(v') \), and from \( h \in (v, v') \) we conclude that \( \text{vars}(cc')(v) \cap \text{vars}(cc')(v') = \emptyset \). Therefore, \( h' \in (v, v') \).

Consider \( (x, x) \rightarrow (v, v') \in \alpha_{V}^{nsh}(x = t(v, v', v_1, \ldots, v_n)) \), with \( n \geq 0 \). Since \( x \) is ground in \( h \), we conclude that \( v \) and \( v' \) are made ground in \( h' \). Therefore, \( h' \in (v, v') \).

Consider \( \rightarrow (v, v') \in \alpha_{V}^{nsh}(v = t) \), with \( t \) ground. Since \( v \) is ground in \( h' \), we have \( h' \in (v, v') \).

Consider \( (v', v)(v_1, v_1) \cdots (v, v_n) \rightarrow (v', v) \in \alpha_{V}^{nsh}(v = t(v_1, \ldots, v_n)) \), with \( n \geq 1 \). We have \( c' = \text{mgu}\{c(v) = t(c(v_1), \ldots, c(v_n))\} \) with \( \text{dom}(c') \cup \text{rng}(c') \subseteq \text{vars}(c(v)) \cup \bigcup_{i=1}^{n} \text{vars}(c(v_i)) \) and from the fact that \( h \in (v', v)(v_1, v_1) \cdots (v', v_n) \) we have \( \text{dom}(c') \cap \text{vars}(c(v')) = \text{rng}(c') \cap \text{vars}(c(v')) = \emptyset \). Therefore, we have \( (cc')(v) = c(v)c', (cc')(v') = c(v')c = c(v') \) and \( \text{vars}(cc')(v') = \text{vars}(cc')(v') \subseteq (\text{vars}(c(v)) \cup \text{rng}(c')) \cap \text{vars}(c(v')) = \text{rng}(c') \cap \text{vars}(c(v')) = \emptyset \). Therefore, \( h' \in (v, v') \).

Consider \( (v, v') \rightarrow (v', v') \in \alpha_{V}^{nsh}(v = t) \), with \( t \) ground. We have that \( c' = \text{mgu}\{c(x) = t\} \) is such that \( c'(y) \) is ground for every \( y \in \text{dom}(c') \). Therefore, we have \( \text{vars}(cc')(v') \cap \text{vars}(cc')(v') = \text{vars}(c(v)) \cap \text{vars}(c(v')) \subseteq \text{vars}(c(v)) \cap \text{vars}(c(v')) = \emptyset \) since \( h \in (v, v') \). Therefore, \( h' \in (v, v') \).

Consider \( (v, v')(v', x)(v_1, v_1) \cdots (v, v_n) \rightarrow (v, v') \in \alpha_{V}^{nsh}(x = t(v_1, \ldots, v_n)) \) with \( n \geq 1 \). We have \( c' = \text{mgu}\{c(x) = t(c(v_1), \ldots, c(v_n))\} \), with \( \text{dom}(c') \cup \text{rng}(c') \subseteq \text{vars}(c(x)) \cup \bigcup_{i=1}^{n} \text{vars}(c(v_i)) \) and since \( h \in (v, v')(v', x)(v', v_1) \cdots (v', v_n) \) we conclude that \( \text{dom}(c') \cap \text{vars}(c(v')) = \text{rng}(c') \cap \text{vars}(c(v')) = \emptyset \). Therefore, we have \( (cc')(v) = c(v)c' \) and \( (cc')(v') = c(v')c = c(v') \). Then \( \text{vars}(cc')(v') \cap \text{vars}(cc')(v') = \text{vars}(c(v)c') \cap \text{vars}(c(v)c') \subseteq (\text{vars}(c(v)) \cup \text{rng}(c')) \cap \text{vars}(c(v)c') = \text{rng}(c') \cap \text{vars}(c(v)c') = \emptyset \), and \( h' \in (v, v') \). The same proof holds for its symmetrical arrow.

Consider \( vx \rightarrow v \in \alpha_{V}^{free}(x = x) \). We have \( c' = \{ c(v) = c(x) \} \). Since \( \{ c(v), c(x) \} \subseteq V \), we have \( (cc')(v) = c(v)c = c(v) \in V \) and \( h' \in v \).

Consider \( vx \rightarrow v \in \alpha_{V}^{free}(x = t(v_1, \ldots, v_n)) \), with \( n \geq 0 \). Since \( c(x) \in V \), we have \( c' = \{ c(x) = t(c(v), c(v_1), \ldots, c(v_n)) \} \). If \( c(v) \neq c(x) \), we have \( (cc')(v) = c(v)c = c(v) \in V \). Otherwise, \( c' \) exists only if \( t(v_1, \ldots, v_n) \equiv v \). In this case, we have \( (cc')(v) = c(v)c = v \in V \). In both cases we have \( (cc')(v) \in V \) and \( h' \in v \).

Consider \( v(x, v)(y, v) \rightarrow v \in \alpha_{V}^{free}(x = y) \). We have \( c' = \text{mgu}\{c(x) = c(y)\} \). Since \( c(v) \in V \) and \( \text{dom}(c') \subseteq \text{vars}(c(x)) \cup \text{vars}(c(y)) \), from \( h \in (x, v)(y, v) \) we
conclude that \( c(v) \not\in \text{dom}(c') \). Therefore, \((cc')(v) = c(v)c' = c(v) \in V\) and \(h \in v\).

Consider \( vxy \to v \in \alpha_V^{\text{free}}(x = y) \). We have \( c' = \{c(x) = c(y)\} \) and since \( \{c(v), c(x), c(y)\} \subseteq V \) we conclude that \((cc')(v) = c(v)c' \in V\). Therefore, \( h' \in v\).

Consider \( v(x,v)(v_1,v) \cdots (v_n,v) \to v \in \alpha_V^{\text{free}}(x = t(v_1, \ldots, v_n)) \), with \( n \geq 0 \) and \( t(v_1, \ldots, v_n) \not\in V \). We have \( c' = \text{mgu}\{c(x) = t(c(v_1), \ldots, c(v_n))\} \). Since \( c(v) \in V \) and \( \text{dom}(c') \subseteq \text{vars}(c(x)) \cup \bigcup_{i=1, \ldots, n} \text{vars}(c(v_i)) \), from the fact that \( h \in (x, v)(v_1, v) \cdots (v_n, v) \) we conclude that \( c(v) \not\in \text{dom}(c') \), \((cc')(v) = c(v)c' = c(v) \in V \) and \( h' \in v\).

Consider \( vx(x,v) \to v \in \alpha_V^{\text{free}}(x = t(v_1, \ldots, v_n)) \), with \( t(v_1, \ldots, v_n) \not\in V \) and \( n \geq 0 \). Since \( c(x) \in V \) we have \( c' = \{c(x) = t(c(v_1), \ldots, c(v_n))\} \) and from \( h \in v(x, v) \) we conclude that \( c(v) \neq c(x) \). Therefore, we have \((cc')(v) = c(v)c' = c(v) \in V \) and \( h' \in v\).

**Lemma 6.21.** By Lemma 6.19 and Proposition 5.15 we know that every arrow \( l \to r \in \alpha_V^{\text{local}}(c) \) is such that \( c \in \text{unrep}_V(l \to r) \). We prove that if we substitute \((v_1, v_2) \in l \) with \( v_1, v_2 \), provided \( \text{mgu}(c(v_1), c(v_2)) \) does not exist, the resulting arrow \( l' \to r \) is such that \( c \in \text{unrep}_V(l' \to r) \). Let \( h' = \exists_{W'} c' \in \{c(x) = t(c(v_1), \ldots, c(v_n))\} \) then \( h' \in 1 \) and \( h' \not\in W' \) and from \( \{c(v_1), c(v_2)\} \subseteq V \) we conclude that \( c'(v_1) = c'(v_2) \). Since \( \text{mgu}(c(v_1), c(v_2)) \) does not exist, we have that \( h' \not\in W' \) and does not exist too.

**Proposition 6.23.** Given \( l \to v \in A \) and \( v' \in V \), if \( v'(v', v') \subseteq l \) then \( l \to v = H_V \) and \( \text{unrep}_V(A) = \text{unrep}_V(A \setminus \{l \to v\}) \). Therefore, we can assume that for any \( l \to v \in A \) and \( v' \in V \) whether \( v' \in l \) or \( (v', v') \in l \), but not both. For every \( l \to v \in A \), let \( \exists_{W} c \in l \to v, \theta = \{v'' = a | v'' \in V \cup W\} \) and \( h = \{v'' = c(v'') \theta | v'' \in V \} \). By definition and the hypothesis on \( l \), we have \( h \in 1 \). Therefore, we have \( h \not\in W \exists_{W} c \in v \). Since \( v \) is upward closed, we have \( \exists_{W} c \in v \). This is true for every \( l \to v \in A \), i.e., \( \exists_{W} c \in \cap_{l \to v \in A} v = F \).
Chapter 7  Freeness in isolation

Libertà va cercando.

Dante Alighieri,
Purgatorio, I, 71, 13th century

As we have seen in Chapter 6, it is not possible to define a precise domain for freeness analysis as the linear refinement of a basic domain for freeness analysis, without the help of any auxiliary property. In this chapter we solve this problem through a non standard use of linear refinement. The result is a precise domain for definite freeness analysis which consists of sets of dependencies between sets of variables. We provide explicit and effective definitions of the operations for this domain which we show to be safe with respect to the concrete operations. We illustrate how the domain may be used in a practical analysis by means of an example.

Part of this chapter has been published in [46].

7.1 Introduction

Freeness analysis is concerned with the computation of a set of variables (called free variables) which, at a given program point, are guaranteed to be only bound to variables. As pointed out in [66], the information collected by a freeness analysis is useful for increasing the power of non strict independent AND-parallelism, for optimising unification, for goal ordering and for the avoidance of type checking. Moreover, for sharing analysis (which deals with the possible sharing of terms among variables), freeness is an important component that can improve both the efficiency and the precision of the analyser.

Freeness analysis has received relatively little attention from the research community, compared for instance with groundness analysis [3, 23]. If we consider the freeness domains proposed in the literature, we find that two different approaches are taken. The first considers freeness as a mode [29, 66]. In this case, a substitution is simply abstracted as the set of free variables (that is, those which are mapped by the substitution to variables). This basic approach is extremely imprecise. To improve precision, [29] combines modes with the information about the set of variables...
present in the term bound to each variable, whereas [66] combines freeness with sharing analysis, to improve both the sharing and the freeness components. Both [29] and [66] show that the basic freeness abstraction is not acceptable at all. This is because even if it is known that a variable is free in the substitutions $\theta_1$ and $\theta_2$, there is no guarantee that it will be free in their most general unifier. Then the most precise abstraction of unification is so imprecise to be useless. Compare this with the case of groundness, where the simple abstraction of a substitution into the set of variables that are ground allows the definition of an abstract unification operation which is imprecise but still useful. The second approach abstracts a substitution into an abstract set of equations, which can be seen as an approximate description of the concrete set of equations represented by the substitution itself [12, 15, 21, 65, 79]. This means that part of the functor structure of the substitution is maintained in the abstract sets of equations, resulting in an extremely precise analysis but with rather complex abstract operations. In [55] an intermediate approach is proposed. Here the terms are fully abstracted apart from the positions of their variables. These are maintained by means of paths that define their positions in the term.

In [30, 37] an abstract domain for definite freeness analysis is defined. Though the general case of constraint logic programming is considered and no use of the refinement technique is made, their notion of constrained sets of variables is very similar to the notion of internal arrows we will define in Section 7.3.

The contribution of this chapter is in the definition of a new domain for definite freeness analysis which is able to express freeness dependencies without the help of any auxiliary domain (like sharing, for instance). This domain is constructed by means of a non standard linear refinement operation. We provide a representation for this domain and safe operations on this representation. Since every element of the representation can be put in a compact normal form, the freeness analysis computed with this domain is definitely more efficient than that computed by using the domain for non pair-sharing and freeness of Chapter 6. However, no information about non pair-sharing is provided. We show that our domain is contained in the Sharing $\cap$ Free domain of [56, 66]. This means that its precision is no more than that of [56, 66]. The advantage of our domain w.r.t. that of [56, 66] is its natural definition, together with the simplicity of its representation and its abstract operators. Moreover, our domain can be used for abstract compilation.

The chapter is organised as follows. Section 7.2 recalls the problem which occurred in Chapter 6 while defining a precise domain for freeness through linear refinement. Section 7.3 shows how to overcome this problem through the use of internal dependencies. Section 7.4 defines a representation of our domain of internal dependencies and Section 7.5 defines the algorithms for the computation of the abstract operators on this representation. Section 7.6 provides an example of freeness analysis of logic programs based on our domain. Section 7.7 compares our work with the standard framework for freeness analysis, which is based on the Sharing $\cap$ Free domain. Section 7.8 draws some conclusions.
7.2 Freeness analysis

In Section 6.3 we have defined a basic domain $\text{Free}_V$ for freeness analysis, and we have shown that it is not precise enough for practical use (Example 6.11). Moreover, in Subsection 6.3.1 we have shown that also its linear refinement w.r.t. the $\star^{\text{Free}_V}$ operation is not able to reach a sensible precision (Example 6.13). As a consequence, in Section 6.4 we have used non pair-sharing information in order to improve the precision of freeness analysis. In this chapter, instead, we want to use linear refinement in a non standard way in order to obtain a precise domain for freeness analysis as a refinement of $\text{Free}_V$, without the help of any auxiliary domain. We say that we use linear refinement in a non standard way since we do not refine $w$, $r$, $t$. We do not claim that this solution is general and can be applied whenever the linear refinement of a basic domain is not precise enough. However, we think that this chapter shows once more the strength of linear refinement, provided it is carefully used.

7.3 Internal freeness dependencies

Consider the constraint $c = \{y = f(a)\}$. We cannot be sure that $x$ is still free after conjunction with a constraint $c'$ where even both $x$ and $y$ are free. However, if $x$ is free in $c'$, instantiation of $x$ in $\text{mgu}(c, c')$ can only be a consequence of the non freeness of $y$ in $c$ which pushes $c'$ to instantiate $x$. Then, from the point of view of $c$, it is an external mechanism. Namely, $c$ instantiates $y$ and $c'$ replies by instantiating $x$ in turn.

Assume we want to model the effects internal to $c$ of the conjunction with $c'$. For instance, in the conjunction of the constraint $c = \{y = f(a), x = v, h = w\}$ with $c' = \{y = f(x), w = a\}$, we want to model the instantiation of $w$ and $h$, but not the instantiation of $x$ and $v$, which, from the point of view of $c$, are the consequence of an external instantiation. This can be achieved by refining the $\text{Free}_V$ domain with respect to the internal operation, which is the element-wise extension of the operation internal defined below.

**Definition 7.1.** Given $V \in \mathcal{V} \cup \mathcal{W}$ and $\{p_1, p_2\} \subseteq \mathcal{V}(H_V)$, we define $\text{internal}^{H_V} = \lambda h'. h. \text{over}(h') \star^{H_V} h$ and $\text{over}(h')$ is $h'$ where all free variables have been renamed into new, overlined, variables. Namely, $\text{over}(\exists \mathcal{V} \mathcal{W} c) = \exists_{\mathcal{V}\mathcal{W}\text{over}(c)} \mathcal{V} \text{over}(c)$, where $\text{over}(\{x_1 = t_1, \ldots, x_n = t_n\}) = \{\text{over}(x_1 = t_1), \ldots, \text{over}(x_n = t_n)\}$ and

\[
\text{over}(x = t) = \begin{cases} 
\text{over}(x) = \text{over}(t) & \text{if } t \in \mathcal{V} \cup \mathcal{W} \\
\phantom{\text{over}(x)} = \text{over}(t) & \text{otherwise}
\end{cases}
\]

\[
\text{over}(f(t_1, \ldots, t_m)) = f(\text{over}(t_1), \ldots, \text{over}(t_m))
\]

\[
\text{over}(v) = \begin{cases} 
\overline{v} & \text{if } v \in \mathcal{V}, \\
\text{with } \overline{v} \in \mathcal{W} \text{ fresh} & \\
v & \text{if } v \in \mathcal{W}.
\end{cases}
\]
Example 7.2. We have that
\[
\text{over}(\{y = f(a), x = v, h = w\}) = \exists_{\{\pi, \tau, \overline{\pi}, \overline{\tau}\}} \{y = f(a), \pi = \overline{\pi}, \tau = \overline{\tau}\} \quad \text{and}
\text{over}(\exists_{\{w\}} \{x = f(w), y = w, h = g(z)\}) = \exists_{\{w, \pi, \tau\}} \{x = f(w), \overline{\pi} = w, h = g(\overline{\tau})\}.
\]

By computing \(\text{mgu(over}(c'), c)\), we can observe in isolation the internal and immediate effects of \(c'\) on \(c\), without bothering about the fact that these effects can reach \(c'\), inducing some new instantiations that can reach \(c\) in turn, and so on.

Definition 7.3. Given \(V \in \varphi_f(V)\), we define the refined domain \({}^1\)
\[
Fr_V = \bigwedge \{a \xrightarrow{\text{internal}} b \mid \{a, b\} \subseteq \text{Free}_V \text{ and } a \subseteq b\}.
\]

The \(\neg \xrightarrow{\text{internal}} \) dependencies, we call internal dependencies, can be seen as the building blocks of the unsatisfactory, global dependencies of Subsection 6.3.1.

Proposition 7.4. Given \(V \in \varphi_f(V)\) we have

i) for every \(\{l, r\} \subseteq \text{Free}_V\),
\[
1 \xrightarrow{\text{internal}} r = \left\{ h' \in H_V \mid \text{for all } h \in H_V, \text{ if } h \in 1 \text{ and } \text{over}(h) \ast h' \in r \right\}
\]

ii) \(\text{Free}_V \subseteq Fr_V\).

As a consequence of point ii of Proposition 7.4, we can use Equation (1.4) instead of Equation (1.3).

From now on, \(\rightarrow\) will stand for \(\xrightarrow{\text{internal}}\). 

Example 7.5. The domain \(Fr_{\{x, y\}}\) is depicted in Figure 7.1. We have \(xy \rightarrow xy \cap x \rightarrow x = xy \rightarrow xy \cap y \rightarrow y = x \rightarrow x \cap y \rightarrow y = \{\varepsilon\}\) and \(xy \rightarrow xy = \{\varepsilon, \{x = y\}\}\).

The point \(x \rightarrow x\) contains \(\varepsilon\) and constraints like \(\{y = f(a)\}\) or \(\exists_{\{w\}} \{y = f(w)\}\).

The point \(xy \rightarrow x\) contains the constraints above, \(\{x = y\}\) and constraints like \(\{y = f(x)\}\). Symmetrically for \(y \rightarrow y\) and \(xy \rightarrow y\). The top element contains the whole set of constraints. Namely, it is the only point containing constraints like \(\{x = f(a), y = f(f(a))\}\). This means that the abstraction of \(\{x = f(a)\}\) is \(y \rightarrow y\), the abstraction of \(\{y = f(a)\}\) is \(x \rightarrow x\) and the abstraction of \(\{x = f(a), y = f(a)\}\) is \(\emptyset \rightarrow \emptyset\).

Example 7.6. Let \(V = \{x, y, w, z\}\). The constraint \(\{x = f(y)\}\) belongs to \(z \rightarrow z\). Then, from the point of view of internal freeness dependencies, \(z\) is free after composition with a constraint where \(z\) is free, where composition means application of the \(\text{internal}^{0(H_V)}\) operator.
Figure 7.1: The domain $Fr_{(x,y)}$ of internal freeness dependencies.

The abstraction map $\alpha^Fr : \wp(H_V) \to Fr_V$ is induced by the abstract domain $Fr_V$. Namely, the pair $\langle \alpha^Fr, \gamma^Fr \rangle$ is defined as $\alpha^Fr(S) = \cap \{ f \in Fr_V \mid S \subseteq f \}$ for every $S \in \wp(H_V)$ and $\gamma^Fr(f) = f$ for every $f \in Fr_V$, and forms a Galois insertion between $\wp(H_V)$ and $Fr_V$.

The concrete operations of Section 6.3 induce corresponding abstract operations on $Fr_V$. We will show that the internal freeness dependencies can be used to model the composition ($\wp(H_V)$ operation) in a precise way. The insight is that to compute the global dependencies between two constraints we have to exploit our knowledge of their internal dependencies.

### 7.4 A representation for $Fr_V$

We can use

$$Rep_V = \{ a \rightarrow b \mid a \in \lambda RepKernel(\{freeness\})_V \text{ and } b \in a \}$$

to represent $Fr_V$. Note that, if $A \in Rep_V$ and $l \rightarrow v \in A$, then $v \in l$. The elements of $Rep_V$ can be simplified by allowing any variable to occur at most once on the right of the arrows. From now on, $Rep_V$ will stand for the set of its constraints which are in normal form. This reduces greatly the dimension of the elements of the representation.

**Definition 7.7.** Given $V \in \wp_f(V)$, a constraint $A \in Rep_V$ is in normal form if and only if $A = \{ l_1 \rightarrow v_1, \ldots, l_n \rightarrow v_n \}$ with $v_i \neq v_j$ for every $i, j = 1, \ldots, n, i \neq j$.

**Proposition 7.8.** Given $V \in \wp_f(V)$, for any $A \in Rep_V$ there is $A' \in Rep_V$ in normal form such that $\text{unrep}_V(A) = \text{unrep}_V(A')$.

---

1 If we remove the condition $a \subseteq b$, the only difference would be that $\emptyset \in Fr_V$. Since no constraint belongs to $\emptyset$, this extra point would be useless.
7.4.1 The abstraction function

We provide now an algorithmic definition of the restriction to singletons of the abstraction map $\alpha^F^r_V : \varphi(H_V) \to Fr_V$ induced by $Fr_V$.

Definition 7.9. Given $V \in \varphi_f(V)$ and $v \in V$, we define $L_c(v) = \{ v' \in V \mid \text{vars}(c(v')) \cap \text{vars}(c(v)) \neq \emptyset \}$ and $\alpha^al_g_V(\exists W c) = \{(L_c(v) \setminus W) \vdash v \mid v \in \text{free}(\exists W c)\}$.

Note that, by definition, $\alpha^al_g(h)$ is in normal form for every $h \in H_V$.

We prove that $\alpha^al_g(h)$ represents exactly $\alpha^F^r_V$.

Proposition 7.10. Given $V \in \varphi_f(V)$ and $h \in H_V$, we have

$$\alpha^F^r_V(\{h\}) = \text{unrep}_V \left( \alpha^al_g(h) \right).$$

Example 7.11. Consider $V = \{x, y, v, z\}$. The table below shows some existential Herbrand constraints and their abstraction through $\alpha^al_g$.

<table>
<thead>
<tr>
<th>$h$</th>
<th>$\alpha^al_g(h)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>${v = f(x), z = a}$</td>
<td>${v : x, v : y \to y}$</td>
</tr>
<tr>
<td>${v = g(x, y), z = f(y)}$</td>
<td>${v : x \to x, v : yz \to y}$</td>
</tr>
<tr>
<td>${v = g(x, y), z = y}$</td>
<td>${v : x \to x, v : yz \to y, v : yz \to z}$</td>
</tr>
<tr>
<td>${v = x, y = x}$</td>
<td>${v : x \to x, v : xy \to y, v : xy \to v, z \to z}$</td>
</tr>
<tr>
<td>$\exists_{(w, s)} {v = f(w, y), s = f(x), z = w}$</td>
<td>${x \to x, y : v \to y, v : z \to z}$</td>
</tr>
</tbody>
</table>

7.4.2 Information extraction

We want an algorithm for computing the set of variables that are free in every existential Herbrand constraint represented by some $A \in \text{Rep}_V$.

Definition 7.12. For any $V \in \varphi_f(V)$ we define the map $\text{free}_V : Fr_V \to \varphi(V)$ as

$$\text{free}_V(A) = \{ v \in V \mid B \to v \in A \}.$$

The following proposition shows that all the variables in $\text{free}_V(A)$ are actually free in the constraints represented by $A$.

Proposition 7.13. Let $V \in \varphi_f(V)$ and $A \in \text{Rep}_V$. Then $\text{unrep}_V(A) \subseteq \text{free}_V(A)$.

\(^2\)Compare this with Definition 6.22.
7.5 The abstract operators

In this section we define the abstract counterparts of the concrete operators of Section 2.4 and Subsection 2.5.2.

Given $V \in \wp_f(V)$ and $\{A_1, A_2\} \subseteq \text{Rep}_V$, their abstract conjunction constructs an arrow for a variable $v$ by unfolding the body of an arrow for $v$ contained in $A_1$ with the arrows in $A_2$ and so on alternately. If this unfolding fails, no arrow is built for the variable $v$.

**Definition 7.14.** Given $V \in \wp_f(V)$ and $\{A_1, A_2\} \subseteq \text{Rep}_V$, we define

$$A_1 \ast_{\text{Rep}_V} A_2 = \left\{ \bigcup_{i \geq 1} \text{dunf}^{i}_{A_1,A_2}(l) \rightarrow v \mid l \rightarrow v \in A_1, \text{dunf}^{i}_{A_1,A_2}(l) \text{ is not fail for every } i \geq 1 \right\},$$

where $\text{dunf}^{1}_{A_1,A_2}(l) = l$, $\text{dunf}^{i}_{A_1,A_2}(l) = \text{unf}(l, A_{1+((i-1) \text{mod} 2)})$ for $i \geq 2$, and

$$\text{unf}(\text{fail}, A) = \text{fail} \quad \text{unf}(\emptyset, A) = \emptyset$$

$$\text{unf}(v_1 \cdots v_n, A) = \begin{cases} l_1 \setminus v_1 \cdots l_n \setminus v_n & \text{where } l_i \rightarrow v_i \in A \text{ for } i = 1, \ldots, n \\ \text{fail} & \text{otherwise}. \end{cases}$$

Note that the above definition provides an algorithm for computing $\ast_{\text{Rep}_V}$, since, $V$ being finite, there must exist two different natural numbers $i, j$, both even or both odd, such that $\text{dunf}^{i}_{A_1,A_2}(l) = \text{dunf}^{j}_{A_1,A_2}(l)$. Then the computation of the union can be stopped at the max($i, j$)-th iteration. Moreover, note that the operation $\ast_{\text{Rep}_V}$ is closed on the set of elements of $\text{Rep}_V$ that are in normal form.

**Example 7.15.** Let $V = \{v, x, y, z\}$. Consider the two sets of arrows $A_1 = \{xy \rightarrow x, vz \rightarrow v, vz \rightarrow z\}$ and $A_2 = \{yz \rightarrow y, x \rightarrow x, v \rightarrow v\}$ in $\text{Rep}_V$. Consider the variable $x$. There is an arrow $xy \rightarrow x$ for $x$ in $A_1$. Then we start unfolding $xy$ and we obtain $\text{dunf}^{1}_{A_1,A_2}(xy) = xy, \text{dunf}^{2}_{A_1,A_2}(xy) = z$, $\text{dunf}^{3}_{A_1,A_2}(xy) = v$ and $\text{dunf}^{4}_{A_1,A_2}(xy) = \emptyset$ for every $i \geq 4$. Hence the abstract conjunction contains the arrow $vxyz \rightarrow x$. If we consider the variable $y$, we do not find any arrow for $y$ in $A_1$. Then no arrow for $y$ is computed by the algorithm. If we consider the variable $v$, we have $\text{dunf}^{1}_{A_1,A_2}(vz) = vz$ and $\text{dunf}^{2}_{A_1,A_2}(vz) = \text{fail}$, since there is no arrow for $z$ in $A_2$. Then the algorithm does not add any arrow for $v$ in the abstract conjunction.

Note that $h_1 = \{y = f(x), z = v\} \in \text{unrep}_V(A_1)$ and $h_2 = \{z = g(y)\} \in \text{unrep}_V(A_2)$. Their concrete conjunction is $h = \{v = g(f(x)), y = f(x), z = g(f(x))\}$. Note that neither $y$ nor $v$ are free in $h$. Then it is correct that the abstract unification algorithm does not contain any arrow for those two variables. Moreover, note that the arrow $vxyz \rightarrow x$, computed by the algorithm, is correct for $h$.

**Proposition 7.16.** Given $V \in \wp_f(V)$, $\ast_{\text{Rep}_V}$ is correct.
Definition 7.17. Given \( V \in \varphi_f(V) \), \( A \in \text{Rep}_V \), \( x \in V \) and \( n \in V \setminus V \), we define

\[
\exists^{\text{Rep}_V}_x(A) = \{x \mapsto x\} \cup \{l \setminus x \mapsto v \mid l \mapsto v \in A \text{ and } v \neq x\}
\]

\[\text{expand}^A_n(A) = A \cup \{n \mapsto n\}\]

\[\text{rename}^A_{x \mapsto n}(A) = A[n/x]\]

\[\text{restrict}^A_n(A) = \{l \setminus x \mapsto v \mid l \mapsto v \in A \text{ and } v \neq x\}\,.
\]

Note that these operations are closed on the set of elements of \( \text{Rep}_V \) that are in normal form.

Proposition 7.18. The operations of Definition 7.17 are (correct and) precise.

Definition 7.19. Given \( V \in \varphi_f(V) \) and \( \{A_1, A_2\} \subseteq \text{Rep}_V \), we define

\[A_1 \cup^{\text{Rep}_V} A_2 = \{l_1l_2 \mapsto v \mid l_1 \mapsto v \in A_1 \text{ and } l_2 \mapsto v \in A_2\}\,.
\]

Proposition 7.20. Given \( V \in \varphi_f(V) \), \( \cup^{\text{Rep}_V} \) is correct.

Definition 7.21. Given \( V \in \varphi_f(V) \), \( \bar{x} = \langle x_1, \ldots, x_n \rangle \) and \( \bar{y} = \langle y_1, \ldots, y_n \rangle \) in \( V \), we define

\[\delta^{\text{Rep}_V}_{\bar{x}, \bar{y}} = \{x_iy_i \mapsto y_i, x_iy_i \mapsto x_i \mid i = 1, \ldots, n\}\,.
\]

Note that \( \delta^{\text{Rep}_V}_{\bar{x}, \bar{y}} \) is in normal form.

Proposition 7.22. Given \( V \in \varphi_f(V) \), \( \bar{x} \) and \( \bar{y} \) in \( V \), \( \delta^{\text{Rep}_V}_{\bar{x}, \bar{y}} \) is the abstraction of \( \delta^{\varphi_f(H_V)}_{\bar{x}, \bar{y}} \).

7.6 An example

We show here a simple example of freeness analysis of logic programs, based on the elastic semantics for call patterns (Subsection 2.5.1). Given an input goal for a procedure and a selection rule, a (concrete) call pattern is the name of a procedure call found during the resolution of the goal, together with the partial Herbrand constraint computed up to that call, restricted to the arguments of the procedure. Note that the input goal itself is a call pattern. Let us call \( \iota_1, \ldots, \iota_n \) the arguments of the input goal and \( \kappa_1, \ldots, \kappa_m \) the arguments of a call pattern found during the resolution of the input goal. The call pattern semantics collects every call pattern found during the resolution of a goal. An abstract call pattern semantics is the same, except that it uses abstract constraints instead of concrete ones. Here, we will use elements of \( \text{Fr}_V \) as the abstract constraints.

Consider the following program, taken from [66], that computes the product of a matrix and a vector:
multiply(\[\emptyset\], V, \[\emptyset\]).
multiply([VM\{MM\}, V, [H\{T\}]): = \text{vmul}(VM, V, H), \text{multiply}(MM, V, T).

\text{vmul}(\[\emptyset\], \[\emptyset\], 0).
\text{vmul}([H\{T\}], [H\{T\}], R): = \text{vmul}(T1, T2, R1), R is (R1+H1*H2).

The call pattern semantics of the predicate \text{vmul} contains just the following two call patterns, both for \text{vmul}:

\[
\begin{align*}
\{t_1k_1 \leftrightharpoons k_1, t_2k_2 \leftrightharpoons k_2, t_3k_3 \leftrightharpoons k_3, t_1k_1 \leftrightharpoons l_1, t_2k_2 \leftrightharpoons l_2, t_3k_3 \leftrightharpoons l_3\}, \text{vmul} \\
\{l_3 \leftrightharpoons l_3, t_1k_1 \leftrightharpoons k_1, t_2k_2 \leftrightharpoons k_2, k_3 \leftrightharpoons k_3\}, \text{vmul}
\end{align*}
\]

The first call pattern corresponds to the same input pattern of the procedure, while the second call pattern corresponds to every recursive call. Both call patterns contain the arrow \(t_i k_i \leftrightharpoons k_i\) for \(i = 1, 2\). The first contains the arrow \(t_3 k_3 \leftrightharpoons k_3\) and the second the arrow \(k_3 \leftrightharpoons k_3\). If we compute the conjunction (\text{Rep}) of this denotation with the input pattern \(\{t_i \leftrightharpoons t_i, k_i \leftrightharpoons k_i \mid 1 \leq i \leq 3\}\), the \(i\)-th argument of every concrete call pattern \((k_i)\) is free. Note that the second abstract call pattern says that the third argument of every concrete call pattern for \text{vmul} \((k_3)\) is free independently from the input pattern (indeed, it contains the arrow \(k_3 \leftrightharpoons k_3\)). This means that in every recursive call found in the concrete resolution, the third argument is always free.

The denotation of \text{multiply} contains more interesting information. It contains call patterns for \text{multiply} as well as call patterns for \text{vmul}:

\[
\begin{align*}
\{t_1k_1 \leftrightharpoons k_1, t_2k_2 \leftrightharpoons k_2, t_3k_3 \leftrightharpoons k_3, t_1k_1 \leftrightharpoons l_1, t_2k_2 \leftrightharpoons l_2, t_3k_3 \leftrightharpoons l_3\}, \text{multiply} \\
\{t_1k_1 \leftrightharpoons k_1, t_3k_3 \leftrightharpoons k_3\}, \text{multiply} \\
\{t_2k_2 \leftrightharpoons l_2, t_1k_1 \leftrightharpoons k_1, t_2k_2 \leftrightharpoons k_2, t_3k_3 \leftrightharpoons k_3\}, \text{vmul} \\
\{t_1k_1 \leftrightharpoons k_1, t_2k_2 \leftrightharpoons k_2, k_3 \leftrightharpoons k_3\}, \text{vmul} \\
\{t_1k_1 \leftrightharpoons k_1, t_3k_3 \leftrightharpoons k_3\}, \text{vmul} \\
\{t_1k_1 \leftrightharpoons k_1, k_3 \leftrightharpoons k_3\}, \text{vmul}
\end{align*}
\]

Both call patterns for \text{multiply} contain the arrow \(t_3 k_3 \leftrightharpoons k_3\). The presence of this arrow in the second call pattern for \text{multiply} means that our domain has been able to capture the freeness dependency of the variable \(T\) from the third argument of the head of the second clause for \text{multiply}. If we compute the conjunction of the above denotation with an input pattern for \text{multiply} whose third argument \((t_3)\) is free \((\{t_3 \leftrightharpoons t_3, k_i \leftrightharpoons k_i \mid 1 \leq i \leq 3\}\), we conclude that every concrete call pattern for \text{multiply} will have its third argument \((k_3)\) free. As noted in [66], if we assume that \text{multiply} is always called with its third argument free and the first two arguments ground, that information allows the two calls \text{vmul}(VM, V, H) and \text{multiply}(MM, V, T) to be executed in \text{AND-parallelism} [43].
7.7 \( Fr_V \) vs \( Sharing_V \cap Free_V \).

In this section we compare our \( Fr_V \) domain with the reduced product \( Sharing_V \cap Free_V \) defined in [56]. Remember that \( Free_V \) has been defined in Definition 6.9.

The domain \( Sharing_V \) has been defined in [56] as \( Sharing_V = \varphi(V) \), with the concretisation map \( \gamma_{Sharing_V} : Sharing_V \to \varphi(H_V) \) given by

\[
\gamma_{Sharing_V}(S) = \{ c \in \{ v' \in V \mid v \in \text{vars}(c(v')) \} \subseteq S \text{ for every } v \in V \cup W \}.
\]

The domain \( Sharing_V \) collects the set of sets of variables which can share some variable. This means that it is used for possible sharing analysis. Since we prefer to reason about definite analyses, we consider definite non sharing analysis. Therefore, we define the opposite property \( nonsharing \) of the property \( sharing \) of Example 5.8.

**Definition 7.23.** Non sharing is a local variable-related property which is defined as \( nonsharing = \{ \{ nonsharing \}_V \}_{V \in \varphi(V), \lambda V, \varphi(V)} \) where, for every \( h \in H_V \) and \( V' \subseteq V \), \( nonSharing_V(\exists_W c, V') \) is true if and only if there is no \( v \in V \cup W \) such that \( V' = \{ v' \in V \mid v \in \text{vars}(c(v')) \} \), its width is \( \infty \).

**Definition 7.24.** Given \( V \in \varphi_f(V) \), the basic domain for non sharing is

\[
nonSh_V = \bigcup \text{Kernel}(\{ nonSharing \}_V).
\]

As usual, we write \((v_1, \ldots, v_n)\) for \( nonSharing(v_1, \ldots, v_n) \).

**Example 7.25.** Let \( V = \{ s, v, x, y, z \} \) and \( h = \exists_w \{ x = f(v), z = g(v, w), y = w \} \). We have \( h \in (s, x, y)(s, z) \), but \( h \notin (z, y) \), because \( z \) and \( y \) share \( w \), and \( h \notin (x, x) \), since \( x \) shares \( v \) with itself.

The following proposition says that \( nonSh_V \) is \( Sharing_V \), since \( \gamma_{Sharing_V} \) is one-to-one, as proved in [49].

**Proposition 7.26.** Given \( V \in \varphi_f(V) \), we have

\[
nonSh_V = \gamma_{Sharing_V} (Sharing_V) \).
\]

The following proposition shows that every point of \( Fr_V \) is the intersection of a point of \( nonSh_V \) and a point of \( Free_V \).

**Proposition 7.27.** Given \( f \in Fr_V \), let \( \eta : Fr_V \to nonSh_V \cap Free_V \) be defined as \( \eta(f) = NSH_f \cap F_f \), where

\[
NSH_f = \bigcap \left\{ (v_1, \ldots, v_n) \mid \text{there exists } l \mapsto v_i \in Fr_V \text{ such that } f \subseteq l \mapsto v_i \text{ and } \{ v_1, \ldots, v_n \} \not\subseteq l \right\}
\]

\[
F_f = \bigcap \{ v \mid l \mapsto v \in Fr_V \text{ and } f \subseteq l \mapsto v \}.
\]

We have \( \eta(Fr_V) = Fr_V \).
Corollary 7.28. Given $V \in \varphi_f(V)$, we have

$$ Fr_V \subseteq \text{nonSh}_V \cap \text{Free}_V = \gamma_{\text{Sharing}_V}(\text{Sharing}_V) \cap \text{Free}_V. $$

Corollary 7.28 shows that $Fr_V$ cannot be more precise than the reduced product $\gamma_{\text{Sharing}_V}(\text{Sharing}_V) \cap \text{Free}_V$ used in [49] for freeness analysis.

7.8 Conclusions

This chapter presents a new domain $Fr_V$ for freeness analysis constructed through the linear refinement technique. An interesting consequence of our investigation is that the standard approach of linear refinement did not lead to a useful domain. Instead, we found a novel approach which we called internal dependencies.

The precision of $Fr_V$ needs to be compared with alternative domains for freeness analysis, for instance the $\gamma_{\text{Sharing}_V}(\text{Sharing}_V) \cap \text{Free}_V$ domain. We have shown that freeness analysis performed with $\text{Rep}_V$ cannot be more precise than freeness analysis performed with $\gamma_{\text{Sharing}_V}(\text{Sharing}_V) \cap \text{Free}_V$. Our opinion is that the two domains provide the same information with respect to freeness. This argument still needs to be investigated further. It could be proved by using the quotient technique [24].

Condensing does not hold for our domain. As shown in [73], under suitable conditions a domain is condensing if and only if it is closed with respect to $\rightarrow_{*^{(H_V)}}$. Our domain is not closed with respect to such a refinement. This is not surprising, since we refined a basic domain for freeness with respect to an operation that is not $*^{(H_V)}$. Note, however, that our domain still yields correct results using a goal-independent semantics. The actual gain in precision when using a goal-dependent semantics is not easily quantifiable without an experimental evaluation.
7.9 Proofs

Proofs of Section 7.3

Proposition 7.4.

i) We have

\[
\|l \rightarrow \text{internal}_{\varphi(H_V)} r\| = \bigvee_{\varphi(H_V)} \left\{ p \in \varphi(H_V) \left| \text{internal}_{\varphi(H_V)}(l, p) \subseteq \varphi(H_V) \cap r \right. \right\} \\
= \bigcup \left\{ p \in \varphi(H_V) \left| \text{internal}_{\varphi(H_V)}(l, p) \subseteq r \right. \right\} \\
= \left\{ h' \in H_V \left| \text{for all } h \in H_V, \text{ if } h \in l \text{ and } \overline{\text{over}(h) \ast^{H_V} h'} \text{ is defined, then } \overline{\text{over}(h) \ast^{H_V} h'} \in r \right. \right\}
\]

ii) Every \( U \in \text{Free}_V \) is such that \( U = V \rightarrow \text{internal}_{\varphi(H_V)} U \).

\[\square\]

Proofs of Section 7.4

Definition 7.29. Given \( V \in \varphi_f(V) \) and \( h \in H_V \), we say that \( l \rightarrow v \in \text{Fr}_V \) is minimal for \( h \) if and only if \( h \in l \rightarrow v \) and for every \( l' \subseteq l \) we have \( h \notin l' \rightarrow v \).

Lemma 7.30. Given \( V \in \varphi_f(V) \) and \( h \in H_V \), we have that

i) every \( l \rightarrow v \in \alpha^{adg}_V(h) \) is such that \( h \in l \rightarrow v \);

ii) if \( l \rightarrow v \) is minimal for \( h \) then \( l \rightarrow v \in \alpha^{adg}_V(h) \).

Proof.

i) Let \( h = \exists W \cdot c \). Let \( (L_c(v) \setminus W) \rightarrow v \in \alpha^{adg}_V(h) \). By definition of \( \alpha^{adg}_V \) we have \( h \in v \). Let \( h' \in H_V \) be such that \( \overline{\text{over}(h') \ast^{H_V} h} \) is defined and \( h' \in L_c(v) \setminus W \). We can assume \( h' = \exists W' \cdot c' \) with \( W \cap W' = \emptyset \). Since \( h' \in L_c(v) \setminus W \) and \( W \cap W' = \emptyset \), we conclude that no variable in \( L_c(v) \) occurs in \( \overline{\text{over}(c')} \). By definition of \( L_c(v) \), the only way \( v \) can be instantiated is by instantiating some variable in \( L_c(v) \). This means that \( \overline{\text{over}(h') \ast^{H_V} h} \in v \). Then \( h \in L_c(v) \setminus W \rightarrow v \).

ii) Let \( l \rightarrow v \) be minimal for \( h \). We have \( h \in v \), since \( \epsilon \in l \) and \( \overline{\text{over}(\epsilon) \ast^{H_V} h} = h \in v \). Then \( L_c(v) \setminus W \rightarrow v \in \alpha^{adg}_V(h) \) and by point i above we conclude that \( h \in L_c(v) \setminus W \rightarrow v \). Let \( v' \in L_c(v) \). We have three alternatives:

(a) \( v = v' \in c \): let \( h' = \{v' = \mathbf{a}\} \);
(b) \( v' = t(v) \in c \), where \( t(v) \) is a term containing \( v \): let \( h' = \{ v' = t(v)[a/v] \} \);
(c) \( v = v'' \in c \) and \( v' = t(v'') \in c \), where \( t(v'') \) is a term containing \( v'' \): let \( h' = \{ v' = t(v'')[a/v''] \} \).

In all cases, by construction we have over(\( h' \)) \( \times^h v \) is defined and over(\( h' \)) \( \times^h v \) \( h \not\subseteq v \). If \( v' \not\subseteq l \), we would have \( h' \in l \) and \( h \not\subseteq 1 \rightarrow v \), a contradiction.

Therefore, it must be \( L_c(v) \setminus W \subseteq l \), and by minimality of \( 1 \rightarrow v \) for \( h \) we conclude that \( L_c(v) \setminus W = 1 \rightarrow v \).

\( \square \)

**Proposition 7.8.** Assume \( \{ l_1 \rightarrow v, l_2 \rightarrow v \} \subseteq A \) with \( A \in \text{Rep}_V \). We want to show that unrep_v((A \setminus \{ l_1 \rightarrow v, l_2 \rightarrow v \}) \cup \{(l_1 \cap l_2) \rightarrow v \}) = \text{unrep}_v(A) \). It suffices to prove that \( \{ l_1 \cap l_2 \} \rightarrow v = (l_1 \rightarrow v) \cap (l_2 \rightarrow v) \). Suppose \( h \in (l_1 \cap l_2) \rightarrow v \). Then \( h \in l_1 \rightarrow v \) and \( h \in B_2 \rightarrow v \). Conversely, let \( h = \exists c \in (l_1 \rightarrow v) \cap (l_2 \rightarrow v) \) and suppose \( l_1' \rightarrow v \) and \( l_2' \rightarrow v \) are minimal for \( h \). Then \( l_1' \subseteq l_1 \) and \( l_2' \subseteq l_2 \). By Lemma 7.30.ii we conclude that \( l'_1 = l'_2 = L_c(v) \setminus W \), so that \( L_c(v) \setminus W \subseteq l_1 \cap l_2 \). Therefore, by Lemma 7.30.i we have \( h \in (l_1 \cap l_2) \rightarrow v \).

\( \square \)

**Proposition 7.10.** We have \( \alpha^{fr}(\{h\}) = \{ 1 \rightarrow v \in \text{Fr}_V \mid h \in l \rightarrow v \} \) (we can always assume the right hand side to be a single variable). Moreover:

\[
\bigcap \{ 1 \rightarrow v \in \text{Fr}_V \mid h \in l \rightarrow v \}
= \bigcap \{ 1 \rightarrow v \in \text{Fr}_V \mid 1 \rightarrow v \text{ is minimal for } h \}
\]

(Lemma 7.30.ii) \[ \supseteq \{ 1 \rightarrow v \in \text{Fr}_V \mid l \rightarrow v \in \alpha^{alg}(h) \} \]

(Lemma 7.30.i) \[ \supseteq \{ 1 \rightarrow v \in \text{Fr}_V \mid h \in 1 \rightarrow v \} \].

Therefore, \( \alpha^{fr}(\{h\}) = \text{unrep}_v(\alpha^{alg}(h)) \).

\( \square \)

**Proposition 7.13.** Let \( A = \{ l_1 \rightarrow v_1, \ldots, l_n \rightarrow v_n \} \) and consider a constraint \( h \in \text{unrep}_V(A) \). We want to show that \( h \in \text{free}_V(A) \). Remember that the head of every arrow belongs to its body. Therefore, from over(\( \epsilon \)) = \( \epsilon \times^h v = h \) and \( \epsilon \in l_i \) for every \( i = 1, \ldots, n \) we conclude that \( h \in v_i \) for every \( i = 1, \ldots, n \).

\( \square \)

**Proofs of Section 7.5**

**Lemma 7.31.** Given \( V \in  \{\text{Fr}_V(V)\} \), let \( A_1, A_2 \in \text{Rep}_V \) be such that there exists an arrow \( l \rightarrow x \in A_1 \times^v A_2 \). Let \( h_1 = \exists c_1 \in \text{unrep}_V(A_1) \) and \( h_2 = \exists c_2 \in \text{unrep}_V(A_2) \) such that \( h_1 \times^h h_2 \) exists. We can assume that \( W_1 \cap W_2 = \emptyset \). Consider the algorithm in Figure 7.2. Let \( p^{(i)} \) be the value of a variable \( p \) at the \( i \)-th iteration. Let \( D^{(i)} = \text{dunf}_{A_1,A_2}^{l'}(l') \) with \( l' \rightarrow x \in A_1 \). The following properties hold at point \( * \):
\[ i := 1; \]
(a) \textbf{for each } \( x = v \in c_1 \) \textbf{ do } \( c_1 := (c_1 \setminus \{x = v\}) \circ \{v = x\}; \)
(b) \textbf{for each } \( x = v \in c_2 \) \textbf{ do } \( c_2 := (c_2 \setminus \{x = v\}) \circ \{v = x\}; \)

do begin
\( fr := 1 + ((i - 1) \mod 2); \) \( to := 1 + (i \mod 2); \)
\( \theta := \text{mgv}(\{v = t_v \in c_{fr} \mid x \in \text{vars}(t_v)\}); \) \( L := \text{dom}(\theta); \)
\textbf{for each } \( v = v' \in c_{to} \) \textbf{ with } \( v \in L \) \textbf{ do }

(c) \textbf{if } \( v' \in L \text{ then } c_{to} := c_{to} \setminus \{v = v'\} \)
(d) \textbf{else } \( c_{to} := (c_{to} \setminus \{v = v'\}) \cup \{v' = \theta(v)\} \)
\textbf{for each } \( v' = t(v) \in c_{to} \) \textbf{ with } \( v \in L \) \textbf{ do }

(e) \( c_{to} := (c_{to} \setminus \{v' = t(v)\}) \cup \{v' = t(\theta(v))\}; \)
(f) \textbf{ remove every } \( v = v \) \textbf{ from } \( c_{to}; \) \{point *\}
\( i := i + 1; \)
end
\textbf{until } \( c_1, c_2 \) \textbf{ have not changed; }

Figure 7.2: The algorithm used in Lemma 7.31. The variable \( x \) is first moved to the right hand side of equations. Then the constraints of \( c_1 \) containing \( x \) are applied to \( c_2 \). The roles of \( c_1 \) and \( c_2 \) are then swapped alternately.
1. no equation of the form \( v = v \), with \( v \in V \cup W \), belongs to \( c_1^{(i)} \cup c_2^{(i)} \) for every \( i \geq 1 \);

2. \( x \) occurs on the right only of \( c_1^{(i)} \cup c_2^{(i)} \) for \( i \geq 1 \);

3. every \( v \in \bigcup_{1 \leq j \leq i} L^{(j)} \) occurs at most once and on the left in \( c_1^{(i)} \cup c_2^{(i)} \), \( i \geq 0 \);

4. \( L^{(1)} \subseteq D^{(1)} \), \( L^{(2)} \subseteq D^{(2)} \) and \( L^{(i+2)} \setminus L^{(i)} \subseteq D^{(i+2)} \) for \( i \geq 1 \);

5. \( c_1 \cup c_2 \) is equivalent to \( c_1^{(i)} \cup c_2^{(i)} \) for \( i \geq 1 \);

6. \( \{ v \in V \mid (v = t) \in c_1^{(i)} \cup c_2^{(i)}, x \in \text{vars}(t) \} \subseteq l \) for every \( i \geq 1 \).

Moreover, the algorithm terminates.

Proof. First of all, note that if a binding \( b \) is such that \( b \in c_1^{(i+1)} \cup c_2^{(i+1)} \) but \( b \notin c_1^{(i)} \cup c_2^{(i)} \) then \( x \) must occur in \( b \), since the bindings which get added contain \( \theta(v) \) with \( v \in L \) and, therefore, \( x \) occurs in \( \theta(v) \). We will use this property in the proofs below.

1. This property is true at the beginning, since \( c_1 \) and \( c_2 \) are assumed to be in normal form, and statements (a) and (b) just select a variant of \( c_1 \) and \( c_2 \) that has \( x \) on the right. The invariant is maintained by statement (f).

2. Statements (a) and (b) guarantee that \( x \) occurs only on the right hand side of the equations when we enter the cycle. Statement (d) is the only one that can introduce a new variable on the left hand side of the equations. If \( v' \) were \( x \) then \( \theta(v) \) must be \( x \) and the added equation would be \( x = x \), which is removed by statement (f).

3. If \( i = 0 \) the property holds vacuously. Assume \( i > 0 \). Since \( \bigcup_{v \in V} \text{vars}(\theta(v)) \subseteq \bigcup_{v \in V} \text{vars}(t_v) \), no variable in \( \bigcup_{1 \leq j \leq i} L^{(j)} \) can be introduced on the right hand side of an equation. Moreover, no variable in \( L^{(j)}, 1 \leq j \leq i \), can be on the left of two equations. This is because the only statement that introduces an equation with a new variables on the left is statement (d). In such a statement, if \( v' \) is \( x \) then the resulting equation would be \( x = x \) (since \( \theta(v) \) must contain \( x \) by construction and the two constraints are assumed to be unifiable). Then such an equation would have been removed by statement (f). If \( v' \) is not \( x \), since \( v \) cannot be \( x \) by point 2, we conclude that \( v = v' \) does not contain \( x \).

By the property stated above, we conclude that \( v = v' \) belonged to \( c_{t_v} \) before entering the cycle. But \( c_{t_v} \) was then in normal form, so no equation had \( v' \) on the left. The algorithm could not have introduced an equation with \( v' \) on the left, since in such a case \( v' \in L^{(j)} \) with \( 1 \leq j < i \) and by inductive hypothesis \( v' \) could not belong to the right hand side of any equation. This entails that no equation has \( v' \) on the left, and statement (d) maintains the invariant.
4. The set $D^{(1)}$ is such that $D^{(1)} \rightarrow x \in A_1$. Since for every variable $v$ in $L^{(1)}$ there exists $v = t(x) \in c_1^{(1)}$ and $h_1 \in \text{unrep}_v(A_1)$, we conclude that $L^{(1)} \subseteq D^{(1)}$. Since $L^{(1)} \subseteq D^{(1)}$ with $D^{(1)} \rightarrow x \in A_1$ and there exists an arrow $B \rightarrow x$ in $A_1 \ast \text{rep}_v(A_2)$, we conclude that for every variable $v \in L^{(1)}$ there must be an arrow $B_v \rightarrow v$ in $A_2$. Since $h_2 \in \text{unrep}_v(A_2)$ we have

$$L^{(2)} = \{ v \mid v = t \in c^{(2)}_{fr(x)} \text{ and } x \in \text{vars}(t) \}$$

$$\subseteq \{ v \mid v = t(x) \in c_2 \} \cup \{ v' \mid v = v' \in c_2, v \in L^{(1)} \}$$

$$\cup \{ v' \mid v' = t(v) \in c_2, v \in L^{(1)} \}$$

$$\subseteq (B_x \setminus x) \cup \bigcup_{v \in L^{(1)}} (B_v \setminus v) \subseteq D^{(2)}.$$

Assume $v' \in L^{(i+2)}$ with $v' \not\in L^{(i)}$ and $i \geq 1$. Then $v' = t(x) \in c^{(i+2)}_{fr(x)}$ and an equation of the form $v' = t(v)$ or some equations of the form $v = v'$ (but not both) belong to $c^{(i)}_{fr(x)}$ for some $v$, with $x \not\in \text{vars}(t(v))$ and $v \neq v'$ (point 1). We conclude that $v \in L^{(i+1)}$. When $i = 1$, we have $v \in D^{(2)}$. If $i > 1$ by inductive hypothesis we know that $L^{(i+1)} \setminus L^{(i-1)} \subseteq D^{(i+1)}$ and since $v \not\in L^{(i-1)}$ (otherwise it would have been substituted by a term not containing $v$) we conclude that even in this case we have $v \in D^{(i+1)}$. Then $v$ must have been unfolded in the definition of $\ast \text{rep}_v$ with an arrow $B_v \rightarrow v \in A_{fr(x)}$ with $v' \in B_v$. Since $v$ and $v'$ are different variables, we conclude that $v' \in D^{(i+2)}$, and we get the thesis.

5. Statements (a) and (b) clearly maintain the equivalence. In statement (c) $v$ and $v'$ must be both bound in $c^{(i)}_{fr(x)}$ to terms containing $x$, since both belong to $L^{(i)}$. By point 1 we know that $v$ and $v'$ are different variables. By point 4, there must exist $k_v, k_{v'} \geq 0$ such that $v \in D^{(i-2k_v)}$ and $v' \in D^{(i-2k_{v'})}$, with $i - 2k_v \geq 1$ and $i - 2k_{v'} \geq 1$. Then there must be two arrows $l_v \rightarrow v$ and $l_{v'} \rightarrow v'$ in $A_{fr(x)}$. Then $v \in l_v$ and $v' \in l_{v'}$ and there must be two arrows in $A_{fr(x)}$ whose heads are $v$ and $v'$, respectively. This means that $v$ and $v'$ are free in $c^{(i)}_{fr(x)}$ (Proposition 7.13). Then both are bound to the term $x$. This means that the binding $v = v'$ is useless and we maintain the equivalence if we drop it. In statement (d) we bind $v'$ to $\theta(v)$, that is the same term $v$ is bound to $\theta(v)$ is $t_v$ where some variables have been substituted with their value as provided by $c^{(i)}_{fr(x)}$. Then the binding $v = v'$ becomes useless and we maintain the equivalence if we drop it. In statement (e) we just use the application rule of the standard unification algorithm [63]. Then we maintain the equivalence. Statement (f) maintains the equivalence trivially.

6. We have $\{ v \in V \mid (v = t) \in c^{(i)}_1 \cup c^{(i)}_2, x \in \text{vars}(t) \} \subseteq L^{(i)} \cup L^{(i+1)}$ which by point 4 is contained in $\bigcup_{i \geq 1} D^{(i)} = l$. 

Point 3 entails that the algorithm terminates. Indeed, when a variable \( v \) is added in \( L \) due to an equation of the form \( v = t \) with \( x \in \text{vars}(t) \), it gets removed by the remaining equations of \( c_1 \cup c_2 \), as point 3 says. Therefore, at the next iteration \( v \) cannot be applied anymore to other equations. Since the number of variables in \( c_1 \cup c_2 \) is finite, we have the thesis.

**Proposition 7.16.** We have to prove that, given \( \{A_1, A_2\} \subseteq \text{Rep}_V \)

\[
\text{unrep}_V(A_1) *_{\text{pr}(V)} \text{unrep}_V(A_2) \subseteq \text{unrep}_V(A_1 *_{\text{pr}_V} A_2).
\]

Consider two constraints \( h_1 = \exists W_1 c_1 \in \text{unrep}_V(A_1) \) and \( h_2 = \exists W_2 c_2 \in \text{unrep}_V(A_2) \) with \( W_1 \cap W_2 = \emptyset \). If \( h_1 *_{\text{pr}_V} h_2 \) is defined, consider a generic arrow \( l \rightarrow x \in A_1 *_{\text{pr}_V} A_2 \). By Lemma 7.31 we know that \( c_1 \cup c_2 \) can be equivalently rewritten as a constraint \( c'' \) such that, letting \( c = \{b \in c'' \mid x \in \text{vars}(b)\} \) and \( c' = c'' \setminus c \), \( x \) occurs on the right of \( c \) only and the right side of \( c' \) is \( c \). Then \( \{v \mid v = x \in \text{vars}(\text{mgu}(c, \overline{c'}))(v)\} = \emptyset \). By Proposition 7.10 we conclude that \( h_1 *_{\text{pr}_V} h_2 \in \text{unrep}_V(l' \rightarrow x) \), where \( l' = D \setminus W_1 \setminus W_2 \). By point 6 of Lemma 7.31 we know that \( l' \subseteq l \). Then \( h_1 *_{\text{pr}_V} h_2 \in \text{unrep}_V(l \rightarrow x) \). By the generality of \( l \rightarrow x \), we have the thesis.

**Proposition 7.18.** We have to prove that, given \( V \in \varphi(\mathcal{V}) \), \( x \in V \), \( n \in \mathcal{V} \setminus V \) and \( A \in \text{Rep}_V \), we have

\[
\begin{align*}
\exists x^{\text{pr}(V)} \text{unrep}_V(A) &= \text{unrep}_V(\exists x^{\text{pr}_V} A) \\
\text{expand}_n^{\text{pr}(V)} \text{unrep}_V(A) &= \text{unrep}_V(\text{expand}_n^{\text{pr}_V} A) \\
\text{rename}_{x \rightarrow n}^{\text{pr}(V)} \text{unrep}_V(A) &= \text{unrep}_V(\text{rename}_{x \rightarrow n}^{\text{pr}_V} A) \\
\text{restrict}_x^{\text{pr}(V)} \text{unrep}_V(A) &= \text{unrep}_V(\text{restrict}_x^{\text{pr}_V} A)
\end{align*}
\]

Consider the cylindrification operator. We have

\[
\begin{align*}
\text{unrep}_V(\exists x^{\text{pr}_V} A) &= x \rightarrow x \cap \bigcap \{l \mid x \rightarrow v \mid l \rightarrow v \in A \text{ and } v \neq x\} \\
\exists x^{\text{pr}(V)} \text{unrep}_V(A) &= \{\exists_{W \cup N} c[N/x] \mid x \in l \rightarrow v \text{ for all } l \rightarrow v \in A\}.
\end{align*}
\]

Let \( h = \exists_{W} c \in \text{unrep}_V(\exists x^{\text{pr}_V} A) \). Since \( h \in x \rightarrow x \), we can assume that \( x \) does not occur in \( c \). Therefore, \( h = \exists_{W \cup N} c[N/x] \). Let \( l \rightarrow v \in A \). If \( v \) is not \( x \), from \( h \in x \rightarrow x \) we conclude that \( h \in l \rightarrow v \). If \( v = x \), from \( h \in l \setminus x \rightarrow v \) we conclude that \( h \in l \rightarrow v \). Therefore, \( h \in \exists x^{\text{pr}(V)} \text{unrep}_V(A) \). Conversely, assume \( h = \exists_{W \cup N} c[N/x] \in \exists x^{\text{pr}(V)} \text{unrep}_V(A) \). Since \( x \) does not occur in \( h \), we have \( h \in x \rightarrow x \). For any \( l \setminus x \rightarrow v \in A \) we have \( \exists_{W} c \in l \rightarrow v \). By Lemma 7.30.i, we conclude that \( L_{c}(v) \subseteq l \). Since \( L_{c[N/x]}(v) = L_{c}(v) \setminus x \), we have \( L_{c[N/x]}(v) \subseteq l \setminus x \). By Lemma 7.30.i we conclude that \( h \in l \setminus x \rightarrow v \). Therefore, \( h \in \text{unrep}_V(\exists x^{\text{pr}_V} A) \).

The result for the other operations can be proved similarly.
**Proposition 7.20.** Let \( \{A_1,A_2\} \subseteq \text{Rep}_V \). We have to prove that
\[
\text{unrep}_V(A_1) \cup \text{unrep}_V(A_2) \subseteq \text{unrep}_V(A_1 \cup \text{Rep}_V A_2).
\]
If \( h \in \text{unrep}_V(A_1) \cup \text{unrep}_V(A_2) \) then whether \( h \in \text{unrep}_V(A_1) \) or \( h \in \text{unrep}_V(A_2) \). If \( l \rightarrow v \in A_1 \cup \text{Rep}_V A_2 \), then \( l = l_1 l_2 \) with \( l_1 \rightarrow v \in A_1 \) and \( l_2 \rightarrow v \in A_2 \). If \( h \in \text{unrep}_V(A_1) \) we have \( h \in l_1 \rightarrow v \) which entails \( h \in l_1 l_2 \rightarrow v \). If \( h \in \text{unrep}_V(A_2) \) we have \( h \in l_2 \rightarrow v \), which entails \( h \in l_1 l_2 \rightarrow v \). Since this is true for every \( l \rightarrow v \in A_1 \cup \text{Rep}_V A_2 \), we have the thesis. \( \square \)

**Proposition 7.22.** It is a simple corollary of Proposition 7.10. \( \square \)

**Proofs of Section 7.7**

**Proposition 7.26.** It suffices to note that, given \( S \in \varphi(\varphi(V)) \), we have
\[
\gamma_{\text{Sharing}_V}(S) = \bigcap \{(v_1,\ldots,v_n) \mid \{v_1,\ldots,v_n\} \not\subseteq S\}.
\]
Since the function \( \lambda S \in \varphi(\varphi(V)). \bigcap \{(v_1,\ldots,v_n) \mid \{v_1,\ldots,v_n\} \not\subseteq S\} \) ranges over all \( \text{nonSh}_V \), we have the thesis. \( \square \)

**Proposition 7.27.** First we show that \( f \subseteq \text{NSH}_f \cap F_f \). Assume \( (v_1,\ldots,v_n) \subseteq \text{NSH}_f \). Then there exists \( l \rightarrow v_i \in \text{Fr}_V \) such that \( f \subseteq l \rightarrow v_i \) and \( \{v_1,\ldots,v_n\} \not\subseteq l \). Therefore, for any \( h = \exists_W c \in f \) we have \( h \in l \rightarrow v_i \), i.e., \( L_c(v_i) \setminus W \subseteq l \) (Lemma 7.30.ii). Then \( \{v_1,\ldots,v_n\} \not\subseteq L_c(v_i) \setminus W \). Assume by contradiction that there exists \( v \in V \cup W \) such that \( \{v_1,\ldots,v_n\} = \{v' \in V \mid v \in \text{vars}(c(v'))\} \). Since \( v_i \) is free in \( h \) (Proposition 7.13), it must be \( c(v_i) = v \) and by Definition 7.9 it must be \( \{v_1,\ldots,v_n\} = L_c(v_i) \setminus W \), a contradiction. Then we conclude that \( h \in (v_1,\ldots,v_n) \). This proves that \( f \in \text{NSH}_f \). The result \( f \in F_f \) follows by Proposition 7.13.

We prove now that \( \text{NSH}_f \cap F_f \subseteq f \). Consider any arrow \( l \rightarrow v \in \text{Fr}_V \) such that \( f \subseteq l \rightarrow v \). If we prove that every \( h = \exists_W c \in \text{NSH}_f \cap F_f \) is such that \( h \in l \rightarrow v \) we conclude that \( h \in f \), i.e., the thesis. Since \( F_f \subseteq l \rightarrow v \subseteq v \) and \( h \in F_f \), we know that \( v \) is free in \( h \). Then \( h \in L_c(v) \setminus W \rightarrow v \) (Lemma 7.30.i). Then \( h \not\subseteq (v_1,\ldots,v_n) \) where \( L_c(v) \setminus W = \{v_1,\ldots,v_n\} \). From \( h \in \text{NSH}_f \) we conclude that \( L_c(v) \setminus W \subseteq l \), i.e., \( h \in l \rightarrow v \). \( \square \)

**Corollary 7.28.** From Propositions 7.27 and 7.26. \( \square \)
Conclusions

La certidumbre de no haber encontrado ninguna solución para tantos problemas,
sino todo lo contrario:
nuevos y variados problemas
para ninguna solución.

G. G. Márquez,
El amor en los tiempos del cólera, 1985

We have shown that a careful use of linear refinement leads to the definition of abstract domains which are precise, intuitive and can be used for abstract compilation. Although specific implementations of these domains can be provided by exploiting the specific properties of the abstract domain (Chapters 4 and 7), a generic way for implementing a linearly refined domain exists (Chapter 5). This generic construction is strong enough to be instantiated to a full featured non pair-sharing and freeness analysis.

We think that the ideas contained in Chapter 5 can be generalised to the analysis of store-based languages. Therefore, we plan to extend our results to the analysis of traditional properties of imperative languages [1]. The advantage of our framework, w.r.t. the traditional static analyses, is that it can be done through abstract compilation, is very general (Chapter 5) and leads to an input-independent abstract denotation of the procedures. Therefore, it can be used even when the source code of some procedures is not known, as long as their abstract denotation is provided.

There is an increasing interest in secrecy and security analysis of Internet applications, typically written in Java and Java-like languages. We think that the linear refinement technique can provide new domains for precise analysis of these properties. For instance, it has been shown in [82], how secrecy can be modelled by means of an abstract domain that tracks those variables (high variables) that may be affected by the assignment of values to selected input variables. These secret values may only be divulged to a fixed set of trusted procedures. A typical example of information which must not be divulged without proper authorisation is a credit-card number. The domain of [82] cannot express dependencies between the high variables and the other variables so that every procedure must be re-analysed for each possible input. Thus, it is not adequate for a source-independent, modular analysis of a program. One promising way in which a refined and more precise domain, able to be used for a source-independent and modular static analysis may be
obtained, is to apply the linear refinement technique to the simple low/high secrecy domain.

Analysing for security appears more difficult, since security properties are normally rather complex and can only be checked through the use of abstract interpretation together with a language, called the security language, that expresses the actual security properties of applets and with model checking over this language. There are a number of possible security language choices: trace languages [64], security automata [71] and temporal logic [62]. Thus, we first need to compare and select the security language before defining the basic domain for security. Then we can think at applying linear refinement to construct an analysis domain for checking the security via abstract interpretation.

In the context of object-oriented programming languages, as a consequence of class specialisation, method invocations are often virtual, that is, they are only resolved at run-time. Not only does this result in a significant loss of efficiency of the actual program, but it can have a detrimental effect on the analyser. Both the time taken to do the analysis and the precision of the results can be affected. However, many of these virtual invocations cannot arise at run-time and, furthermore, this can be checked by (statically) analysing the program [41, 68]. It would be interesting to investigate this case of program analysis, and how linear refinement can be used in this context. The ideal static analysis of virtual methods specialisation should allow us to analyse every single module in isolation, and then compose the results to obtain the overall analysis results. This is the case if the analysis is compositional w.r.t. the addition of new modules and new classes by other modules. Such a property can be obtained through the use of linear refinement over type variables, in a way which is reminiscent of what we did in Chapter 4.

Some experimental results about the efficiency of the generic analyser of Chapter 5, or of its instantiation in Chapter 6, are needed. Abstract constraints could be very large, in general. Therefore, an implementation will show if this happens for real programs, if this slows down the computation of the fixpoint and how this can be overcome, by using, for instance, stronger and domain-specific reduction rules. Moreover, an efficient and compact representation of the arrows must be devised. Indeed, a naive representation, of sets of sets of variables would be inefficient for implementing the abstract operations, which use inclusion and union of such sets. A possible representation through sequences of bits would not be compact enough when a large set of variables is considered. Therefore, it must be compacted in some way. For instance, bits could be allocated on demand as soon as the set they represent is needed. The specific problems underlying this approach still needs to be investigated.

In conclusion, we think that this thesis has shown that linear refinement is a practical tool to solve real problems which are not necessarily related to the logic programming world. We plan to strengthen this claim in our future work.
Bibliography


Index

\(? (l \rightarrow r), 89\)
\(M_T(P), 60\)
\(T_{cp}, 16\)
\(T_{cp, ela}, 23\)
\(T_P, 14\)
\(T_{ela}, 21\)
\(\exists^{D_V}, 14\)
\(\exists^{cp, V}, 16\)
\(\Phi_{\Delta V}, 40\)
\(\Theta_{t, w}, 6\)
\(\alpha(P), 27\)
\(\exists, 9\)
\(R \rightarrow \mathbb{R}, 82\)
\(\mathcal{CP}^D_{\subseteq}, 23\)
\(\mathcal{CP}^{D_V}_{\subseteq}, 16\)
\(\bigcup^{Rep(P)}_{V}, 82\)
\(\land, 78\)
\(\delta_{Down_{P}}, 36\)
\(\delta_{Solv_{V}}, 37\)
\(\delta_{P_{V}}, 40\)
\(\text{dim}, 78\)
\(\text{dom}, 6\)
\(\text{CA}^{D_V}_{\subseteq}, 14\)
\(\text{Eq}, 17\)
\(\equiv_{\gamma_{V}}, 41\)
\(\equiv_{\gamma}, 60\)
\(\equiv_{T_{V}}, 41\)
\(\equiv_{\text{Prog}^k_{V}}, 60\)
\(\equiv_{\text{Down}_{P}}, 36\)
\(\equiv_{\text{Solv}_{V}}, 37\)
\(\equiv_{\text{Rep}(P)_{V}}, 85\)
\(\equiv_{P_{V}}, 40\)
\(\text{expand}^{Rep(P)_{V}}, 84\)
\(\text{extract\_free}, 105\)
\(\text{extract\_nsh}, 105\)
\(\text{extract\_p}, 86\)
\(\gamma_{T_{V}}, 41\)
\(\text{gfp}, 5\)
\(\varnothing_{[T], 41\}
\(\bigwedge, 39\)
\(T_{n}(x), 5\)
\(\leq_{\gamma_{T_{V}}, 41\}
\(\text{lfp}, 5\)
\(\rightarrow, 78\)
\(\rightarrow_{\exists}, 10\)
\(\mathcal{G}^{D_V}, 13\)
\(\mathcal{G}^D, 19\)
\(\mathcal{I}^{D_V}, 13\)
\(\mathcal{I}^D, 20\)
\(\mathcal{I}^{cp, V}, 15\)
\(\mathcal{I}^D, 21\)
\(\mathcal{P}^{D_V}, 13\)
\(\mathcal{P}^D, 20\)
\(\text{p}(V^V_{V}), 80\)
\(\text{v}_t, 46\)
\(\mathcal{S}_P^{cp}, 16\)
\(\mathcal{S}_P, 14\)
\(\mathcal{S}_P^{cp, ela}, 23\)
\(\mathcal{S}_{ela}, 21\)
\(\text{Basic}^i_{\gamma_{V}}, 46\)
\(\text{C}_V, 16\)
\(\text{Def}_{\Delta}, 46\)
\(\text{Down}, 36\)
\(\text{H}, 17\)
\(\text{H}_{V}, 17\)
\(\text{Kernel}(P)_{V}, 80\)
\(\text{Pos}_{\Delta}, 46\)
\(\text{Prog}^k, 60\)
\(\text{RepKernel}(P)_{V}, 80\)
\(\text{Rep}(P), 83\)
equivalence relations, 3
equivalent exist. Herbr. constraints, 17
existential Herbrand constraints, 17
existential variables, 17
extensivity property, 7
fixpoint, 5
freeness, 80
Galois connection, 7
Galois insertion, 7
generic algorithm, 81
goal, 13
goal-dependent analysis, vi
goal-independent analysis, vi
goal-independent semantics, 11
greatest fixpoint, 5
greatest lower bound, 4
ground term, 6
groundness, 79
Herbrand equations, 16
immediate consequence operator, 14
interpretation, 13
intuitionistic negation, 47
isomorphism, 4
least fixpoint, 5
least upper bound, 4
linear refinement, vii, 9
linearity, 80
local variable-rel. prop., 79
lower bound, 4
magic-set transformation, 15
minimal arrow, 128
monotonic function, 4
Moore closure, 9
Moore family, 9
non groundness, 81
non pair-sharing, 97
non sharing, 126
normal form, 16
one-to-one function, 2
onto function, 2
optimal operator, 9
partial function, 1
partial order, 3
partially ordered set, 3
poset, 3
positive formula, 42
positive type constraint system, 42
possible analysis, 81
post-fixpoint, 5
powerset, 1
pre-fixpoint, 5
precise operator, 9
preorder, 4
preorder on substitutions, 6
program, 13
program variables, 17
quantale, 10
query, 14
quotient set, 3
range of a substitution, 1
reduced product, 9
reduction rule, 88
relation composition, 2
renaming, 6
representation, 78
Schröder elimination, 40
semantics of a program, 14
sharing, 80
solutions of a Herbrand constraint, 17
strict binary relation, 4
structural type const. system, 42
structural type system, 42
substitution, 6
subsumption, 6
taxtological arrows, 82
top element, 4
totally ordered set, 3
transfinite formula, 40
type constraint system, 40
type model, 59
type signature, 39
type system, 39
type variables, 59
type-equivalent substitutions, 52
upper bound, 4
upper closure operator, 5
variable-relat. prop. width, 79
variable-related property, 79
variance, 6
variants, 6