UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

DOBLE GRADO DE INGENIERÍA INFORMÁTICA Y MATEMÁTICAS

TRABAJO DE FIN DE GRADO 2017/2018

TOWARDS COMPUTING DISTANCES AMONG ABSTRACT INTERPRETATIONS

Autor: Ignacio Casso San Román

Director: Francisco J. López Fraguas

Co-director: Manuel V. Hermenegildo

Madrid, September 15, 2018

Contents

Acknowledgments

I would like to express my gratitute and dedicate this work to all the people that have supported me throughout its development and my university years.

To my advisors Jose Francisco Morales and Manuel Hermenegildo at the IMDEA Software Institute and Francisco Javier López Fraguas at UCM, for their supervision and continuous support and for giving me the opportunity to work on this exciting topic. They shared with me a lot of their expertise and research insight, as well as their general wisdom and energy.

To my colleagues at IMDEA and the CLIP group, and especially to my office mates Isabel García and Nataliia Stulova, for being always there to solve doubts and hear my complaints, and for their valuable advice.

To the many current and past members of the CLIP group who made the Ciao Prolog and the CiaoPP preprocessor systems possible, on which my work is based.

And of course, to my friends and family, for their love, support and patience.

Finally, I am grateful to the institutions which have funded my research activities: the IMDEA Software Institute, the Universidad Complutense de Madrid (UCM), the Madrid Regional Government under the *N-GREENS* program, and the Spanish MINECO under the *TRACES* project.

Abstract

Abstract interpretation is a technique which safely approximates the execution of programs. These aproximations can then be used by static analysis tools to reason about properties that hold for all possible executions, in order to optimize, verify or debug programs, among other applications. Different abstractions, called abstract domains, and analysis algorithms, computing the fixpoints involved in different ways, are used in this process, resulting in different aproximations, all of which are correct but may have different precision.

This use of abstract interpretations is purely *qualitative*: it relies on an order $\sqsubseteq$ in the abstract domains and the fact that one abstract interpretation over-aproximates or under-aproximates the actual (or some given) semantics of programs. A *quantitative* use of abstract interpretations is not covered by the existing theory, that is, there is no way to measure how close two abstract interpretations are to each other, even when one over-aproximates the other. However, the structure of abstract domains and (logic) programs suggests that one could define distances and metrics among those abstract domains and abstract interpretations, and those distances could arguably find many applications, such as comparing the precision of different analyses.

In this work we develop theory and tools to work with abstract interpretations *quantitatively*, in the context of the Ciao and CiaoPP environment. First, we develop a theory for distances in abstract domains and propose distances for CiaoPP domains. Later, we extend those distances to distances between whole analyses of programs. Finally, we apply successfully those distances in experiments to measure the precision of different analyses.

Keywords: Abstract interpretation, static analysis, logic programming, metrics, distances, complete lattices, program semantics, program comparison.

Resumen

La interpretación abstracta es una técnica que permite aproximar correctamente la semántica de los programas. Para ello se usan distintas abstracciones, llamadas dominios abstractos, y algoritmos de análisis, lo cual da lugar a aproximaciones correctas con distinta precisión. Esas aproximaciones son después usadas por herramientas de análisis estático para razonar acerca de las propiedades que se cumplen para todas las posibles ejecuciones del programa, y así poder optimizar, verificar o depurar los programas, entre otras aplicaciones.

Este uso de la técnica de interpretación abstracta es totalmente *cualitativo*: se basa en una relación de orden $\sqsubseteq$ en los dominios abstractos y en el hecho de que una interpretación abstracta sobreaproxime la semántica real (o una dada) del programa en cuestión. La teoría actual no contempla un uso *cuantitativo* de las interpretaciones abstractas, es decir, no permite medir cómo de parecidas son dos interpretaciones abstractas. Sin embargo, la estructura de los dominios abstractos y de los programas sugiere que se podrían definir distancias en esos dominios y entre interpretaciones abstractas, las cuales se podrían usar para medir la precisión de distintos análisis, entre otras aplicaciones.

En este trabajo proponemos teoría y herramientas para trabajar *cuantitativamente* con la técnica de interpretación abstracta, en el contexto del lenguaje de programación Ciao y su preprocesador CiaoPP. En primer lugar se desarrollan bases teóricas para definir distancias en dominios abstractos y se proponen algunas distancias para los dominios usados en CiaoPP. Después se extienden esas distancias en dominios a distancias entre interpretaciones abstractas completas de los programas. Finalmente, se aplican esas distancias en experimentos para medir la precisión de distintos análisis.

Palabras clave: Interpretación abstracta, análisis estático, programación lógica, métricas, distancias, retículos completos, semántica de programas, comparación de programas.

Chapter 1

Introduction

Abstract interpretation [10] is a well-known theory and framework for modeling static analysis and constructing static analyzers. The basic idea behind it is to interpret (i.e execute) a program over a special abstract domain D_α, in order to approximate the collecting semantics of the program in the concrete and intended domain D (i.e., the set of all states potentially reached in an execution). During the process of abstract interpretation, each operation μ_D used during the normal execution is interpreted as a corresponding abstract operation μ_{D_α} in D_α. If the abstract domain and those abstract operations obey some safety properties (e.g., D and D_α are complete lattices and enjoy a Galois connection, the abstract operations are monotonous in D_α, all ascending chains in the lattice are finite or there is a suitable *widening* operator, etc.), then the analysis is ensured to terminate and the execution over D_α to over-approximate every possible execution in D. Then it can be reasoned safely (but perhaps imprecisely) about the properties that hold for all the executions in D, and that information can be used to verify, optimize, or debug the program, among other applications.

The technique of abstract interpretation has been shown to be practical and effective for building static analysis tools, starting from early successes in applications such as automatic program parallelization [9] (by the PLAI analyzer and parallelizer, now contained within the CiaoPP system used in this work) to the verification of the primary flight control software of the Airbus A340 by the Astrée analyzer [11], to gradually becoming a mainstream tool. Indeed, abstract interpretation is currently in regular use for code verification at all major computer companies such as Google, Facebook, or Microsoft.

However, abstract interpretation has, of course, also its limitations. One of them is its exclusively qualitative nature: currently there is barely any theory or technique to work with abstract interpretations in a quantitative way. By design, abstract interpretation is safe and correct, but that notion of safety or correctness is of topological nature, and we would like to work with abstract interpretations as something more metric. For example, we know that the semantics computed is an over-approximation of the real one, but the theory does not provides a way to tell how precise that approximation is. Even if we knew the actual semantics, there is no way to measure the difference between it and the computed one.

Our goal for this thesis is developing theory and techniques for working with abstract

interpretations and abstract domains in a quantitative way. The context will be that of logic programming: we will be working with Ciao [18], a logic-based programming language, and CiaoPP [20], its state-of-the-art preprocessor based on abstract interpretation. Our approach will be the following: defining distances among abstract domains and abstract interpretations. We will not deal with the abstract interpretation process and the analyzers, but with its results: the elements of the abstract domains and the semantics of the programs analyzed over those domains.

Our working plan will be the following. First we will try to develop a solid theory of distances in abstract domains and define a few distances for the most common CiaoPP domains. Then we will try to extend them to distances between analysis results or abstract semantics. And finally, we will propose and implement a few experiments to test and evaluate those distances, mainly focused in measuring the precision of an analysis, by computing the distance between the true semantics and the one inferred by the preprocessor.

The rest of this document is structured as follows: in Chapter 2 we provide an overview on abstract interpretation of logic programming and Ciao and its preprocessor. In Chapter 3 we develop theory for defining distances in abstract domains and propose distances for some Ciao domains. In Chapter 4 we try to extend those distances to distances between analyses of a program. In Chapter 5, we evaluate those distances with a few experiments. In Chapter 6, we provide our conclusions. The Ciao code developed can be found here.

Chapter 2

Preliminaries

In this chapter we provide a quick overview on some of the required background for this work, that is, abstract interpretation of logic programs and the Ciao programming language and its preprocessor. Other background, like the basics of logic programming and its terminology [23],[1], is assumed to be already known by the reader.

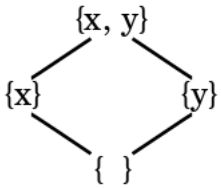
2.1 Abstract Interpretation of Logic Programs

In this section we provide an overview on the basic concepts and ideas of abstract interpretation, specifically in logic programs. First, we introduce the required background on lattice theory [2]. Then, we explain the basic ideas behind the technique, illustrating them with an example. And finally, we explain how abstract interpretation is applied and implemented in the context of logic programs and logic programs semantics. For the two later points we follow the description in [29].

2.1.1 Basic Lattice Theory Definitions

Definition 1 (Partial Order)**.** *A* partial order $(X, \sqsubseteq)$ *is a binary relation* $\sqsubseteq$ *on a set* X *that is reflexive, transitive and antisymmetric.*

Definition 2 (Hasse diagram)**.** *A finite partially ordered set can be represented graphically with a* Hasse diagram*. In that diagram, each element is represented as a vertice in the plane, and two related elements are connected by a segment that goes upward from the lower to the greater. Below it can be found an example for the powerset of the set* $\{x, y\}$ *ordered by inclusion.*



Definition 3. *Let* $(X, \sqsubseteq)$ *be a partial order and* $a, b \in X$*. The* greatest lower bound *of* a *and* b*, denoted by* $a \sqcap b$*, is the greatest element in* X *that is still lower than both of them. Formally,* $a \sqcap b$ *verifies the following properties:*

- $(a \sqcap b) \sqsubseteq a$
- $(a \sqcap b) \sqsubseteq b$
- *if* $c \sqsubseteq a$ *and* $c \sqsubseteq b$ *then* $c \sqsubseteq (a \sqcap b)$.

The greatest lower bound of a lattice, if exists, must be unique*. The greatest lower bound of two elements* a *and* b *is also called the* infimum *(or* meet*) of* a *and* b*.*

Definition 4. *Given a partial order* $(X, \sqsubseteq)$ *and* $a, b \in X$*, the* least upper bound *of* a *and* b*, denoted by* $a \sqcup b$*, is the smallest element in* X *that is still greater than both of them. Formally,* $a \sqcup b$ *verifies the following properties:*

- $a \sqsubseteq (a \sqcup b)$
- $b \sqsubseteq (a \sqcup b)$
- *if* $a \sqsubseteq c$ *and* $b \sqsubseteq c$ *then* $(a \sqcup b) \sqsubseteq c$*.*

The least upper bound of a lattice, if exists, must be unique*. The least upper bound of two elements* a *and* b *is also called the* supremum *(or* join*) of* a *and* b*.*

Definition 5 (Lattice)**.** *A* lattice *is a poset for which any two elements* a *and* b *have a great lower bound and a least upper bound*

Definition 6 (Complete Lattice)**.** *We extend in the natural way the definition of supremum and infimum to subsets of L. Thus, a lattice* $(L, \sqsubseteq)$ *is* complete *if every subset* S *(possibly infinite) of* L *has both a supremum* $\sup(S)$ *and an infimum* $\inf(S)$*. The maximum element of a a complete lattice,* $\sup(L)$ *is called top or* $\top$*, and the minimum,* $\inf(L)$*, bottom or* $\bot$*.*

Definition 7. *Let* $(L_1, \sqsubseteq_1)$ *and* $(L_2, \sqsubseteq_2)$ *be two partially ordered sets. Let* $f : L_1 \longrightarrow L_2$ *and* $g : L_2 \longrightarrow L_1$ *be two applications such that*

$$\forall x \in L_1, y \in L_2 : f(x) \sqsubseteq_1 y \iff x \sqsubseteq_2 g(y)$$

Then the quadruple $\langle L_1, f, L_2, g \rangle$ *is a* Galois connection*, written*

$$L_1 \underset{f}{\overset{g}{\leftrightarrows}} L_2$$

If $\alpha \circ \gamma = id$*, then the quadruple is called a* Galois insertion

2.1.2 Abstract Interpretation

Abstract interpretation [10] is an elegant and useful technique for performing a global analysis of a program in order to compute, at compile-time, characteristics of the terms to which the variables in that program will be bound at run-time for a given class of queries. In principle, such an analysis could be done by an interpretation of the program which, starting from the set of all possible queries, computed the *set of all possible substitutions* (collecting semantics) which can occur at each step of execution. If a property holds for all the substitutions considered, then that property can be assumed in the compilation of the program, i.e., in general the aim of the analysis is to show that given some property p and a set of substitutions Θ, $\forall\theta, \theta \in \Theta \Rightarrow p(\theta)$. However, the computed sets of substitutions can in general be infinite and thus such an approach can lead to non-terminating computations.

Abstract interpretation offers an alternative in which the program is interpreted using *abstract substitutions* instead of actual substitutions. An abstract substitution is a finite representation of a, possibly infinite, set of actual substitutions in the concrete domain. The set of all possible abstract substitutions for a clause represents an "abstract domain" (for that clause) which is usually a complete lattice or cpo of finite height. Abstract substitutions and sets of concrete substitutions are related via a pair of functions referred to as the *abstraction* (α) and *concretization* (γ) functions. In addition, each primitive operation u of the language (unification being a notable example) is abstracted to an operation u' over the abstract domain. Soundness of the analysis requires that each concrete operation u be related to its corresponding abstract operation u' as follows: for every x in the concrete computational domain, $u(x) \sqsubseteq \gamma(u'(\alpha(x)))$.

More formally, abstract interpretation relies on two core concepts: the notion of *approximation* and the notion of *finite representation*. Approximation is based on the observation that if we construct a set $\Theta_a \supseteq \Theta$, and prove that $\forall\theta, \theta \in \Theta_a \Rightarrow p(\theta)$, then the property holds also for Θ. Θ_a is said to be a *safe approximation* of Θ. Any function (such as unification, for example) can also be approximated in a similar way. In particular, a semantic function for a program can be approximated: let the meaning of a program P be a mapping F_P from input to output, input and output values $\in$ "standard" domain D: $F_P : D \to D$. Let's 'lift' this meaning to map sets of inputs to sets of outputs $F_P^* : \wp(D) \to \wp(D)$ where $F_P^*(S) = \{F_P(x) | x \in S\}$ and $\wp(S)$ denotes the powerset of S. A function $G : \wp(D) \to \wp(D)$ is a *safe approximation* of F_P^* if $\forall S, S \in \wp(D), G(S) \supseteq F_P^*(S)$. Using the notion of approximation, properties which are proved using G hold for F_P^*.

The second basic concept is that of *finite representation*. The domain $\wp(D)$ can be represented by an "abstract" domain D_α whose elements are finite representations of (possibly) infinite objects in $\wp(D)$. Thus, in the case of analyzing substitutions in a clause, the concrete domain D is the set of all substitutions for the variables in that clause. The abstract domain D_α is then the set of all "abstract substitutions," an abstract substitution λ being a finite representation of a, possibly infinite, set of actual substitutions. The representation of $\wp(D)$ by D_α is expressed by a (monotonic) function called a *concretization function*: $\gamma : D_\alpha \to \wp(D)$ such that $\gamma(\lambda) = d$ if d is the largest element (under $\subseteq$) of $\wp(D)$ that λ describes. Note that $(\wp(D), \subseteq)$ is obviously a complete lattice. We can also define (not strictly needed) a (monotonic) *abstraction function* $\alpha : \wp(D) \to D_\alpha$, where $\alpha(d) = \lambda$

if λ is the "least" element of D_α that describes d.

An *abstract semantic function* can then be defined as $F_\alpha : D_\alpha \to D_\alpha$ which is a safe approximation of the standard semantic function if $\forall \lambda, \lambda \in D_\alpha, \gamma(F_\alpha(\lambda)) \supseteq F^*_P(\gamma(\lambda))$. It is then possible to prove a property of the output of a given class of inputs represented by λ by proving that all elements of $\gamma(F_\alpha(\lambda))$ have such property.

The set inclusion relation among sets of substitutions in the concrete domain induces an ordering relation in the abstract domain herein represented by "$\sqsubseteq$." Under this relation the abstract domain is usually a complete lattice or cpo of finite height (more precisely "ascending chain finite" [24]), such finiteness required, in principle, for termination of fixpoint computations, and $(D, \alpha, \wp(D), \gamma)$ is a Galois insertion.

Example 1. Using the notation of the previous paragraph, and denoting with $Pvar$ the set of variables in the program, which we consider to be bound at run time to real numbers, then the concrete domain would be $D = \wp(Pvar \to \mathbb{R})$. If we denote $\mathbb{I}$ the set of all intervals in $\mathbb{R}$ with integer endpoints, that is, $\mathbb{I} = \{(l, u) \mid l, u \in \mathbb{Z},\ l <= u\}$, then we could define an abstract domain $D_\alpha = (Pvar \to \mathbb{I})$. This domain is called the *intervals* domain, and informally it abstracts a set of real numbers to the smallest interval that contains it. The concretization function would be $\gamma(\{X_1 \leftarrow I_1, \ldots, X_n \leftarrow I_n\}) = \{\{X_1 \leftarrow r_1, \ldots, X_n \leftarrow r_n\} \mid \forall i = 1 \ldots n,\ r_i \in I_i\}$. The abstraction function would be $\alpha(\Theta) = \{X_1 \leftarrow (l_1, u_1), \ldots, X_n \leftarrow (l_n, u_n)\}$, where $\forall i = 1, \ldots, n,\ l_i = sup\{z \mid z \in \mathbb{Z}, \forall \theta \in \Theta,\ z \leq \theta X_i\},\ u_i = inf\{z \mid z \in \mathbb{Z}, \forall \theta \in \Theta,\ z \geq \theta X_i\}$. $\mathbb{I}$ can be given an order relation $\sqsubseteq$, such that $(l_1, u_1) \sqsubseteq (l_2, u_2) \iff l_2 \leq l_1 \leq u_1 \leq u_2$, and that order relation can be lifted elementwise to an order relation in D_α, under which it would be a complete lattice if we include $\bot$ and $\top$ elements, and $(D, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (D_\alpha, \sqsubseteq)$ would be a Galois insertion. A safe aproximation of the operation $X = Y + Z$ would map an abstract substitution λ to $\lambda[X \leftarrow (l_1 + l_2, u_1 + u_2)]$, where $\lambda X = (l_1, u_1), \lambda Y = l_2, u_2$

□

2.1.3 Abstract Interpretation of Logic Progams

In the case of abstract interpretation for logic programs, the input to the abstract interpreter is a set of clauses (the program) and set of "query forms" i.e., names of predicates which can appear in user queries and their abstract substitutions. The goal of the abstract interpreter is then to compute the set of abstract substitutions which can occur at all points of all the clauses that would be used while answering all possible queries which are concretizations of the given query forms. It is convenient to give different names to abstract substitutions depending on the point in a clause to which they correspond. Consider, for example, the clause h :- $p_1, \ldots, p_n$. Let λ_i and λ_{i+1} be the abstract substitutions to the left and right of the subgoal $p_i, 1 \leq i \leq n$ in this clause. See figure 2.1(b).

Definition 8. *λ_i and λ_{i+1} are, respectively, the* abstract call substitution *and the* abstract success substitution *for the subgoal p_i. For this same clause, λ_1 is the* abstract entry substitution *(also represented as β_{entry}) and λ_{n+1} is the* abstract exit substitution *(also represented as β_{exit}).*

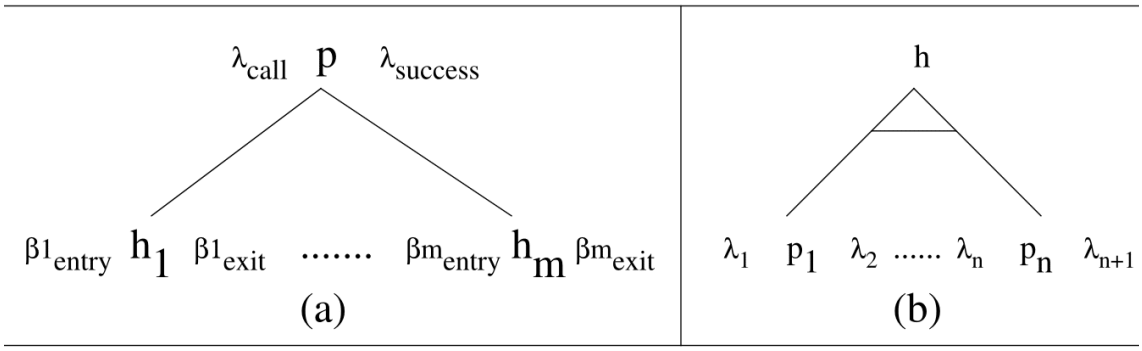


Figure 2.1: Illustration of the abstract interpretation process.

Control of the interpretation process can itself proceed in several ways, a particularly useful and efficient one being to essentially follow a top-down strategy starting from the query forms.[1] The following description is based on the top-down framework of Bruynooghe [3].

In a similar way to the concrete top-down execution, the abstract interpretation process can be represented as an abstract AND-OR tree, in which AND-nodes and OR-nodes alternate. A clause head h is an AND-node whose children are the literals in its body $p_1, \ldots, p_n$ (figure 2.1(b)). Similarly, if one of these literals p can be unified with clauses whose heads are $h_1, \ldots, h_m$, p is an OR-node whose children are the AND-nodes $h_1, \ldots, h_m$ (figure 2.1(a)). During construction of the tree, computation of the abstract substitutions at each point is done as follows:

- *Computing success substitution from call substitution*: Given a call substitution λ_{call} for a subgoal p, let $h_1, \ldots, h_m$ be the heads of clauses which unify with p (see figure 2.1(a)). Compute the entry substitutions $\beta 1_{entry}, \ldots, \beta m_{entry}$ for these clauses. Compute their exit substitutions $\beta 1_{exit}, \ldots, \beta m_{exit}$ as explained below. Compute the success substitutions $\lambda 1_{success}, \ldots, \lambda m_{success}$ corresponding to these clauses. The success substitution $\lambda_{success}$ is then the *least upper bound* (LUB) of $\lambda 1_{success}, \ldots, \lambda m_{success}$. Of course the LUB computation is dependent on the abstract domain and the definition of the $\sqsubseteq$ relation.

- *Computing exit substitution from entry substitution*: Given a clause h :- $p_1, \ldots, p_n$ whose body is non-empty and an entry substitution λ_1, λ_1 is the call substitution for p_1. Its success substitution λ_2 is computed as above. Similarly, $\lambda_3, \ldots, \lambda_{n+1}$ are computed. Finally, λ_{n+1} is obtained, which is the exit substitution for this clause. See figure 2.1(b). For a unit clause (i.e. whose body is empty), its exit substitution is the same as its entry substitution.

This conceptual model if coded directly does not render an efficient implementation. However, efficient fixpoint frameworks (using memo tables) have been developed, in particular the PLAI algorithm [27, 28, 30], that result in very efficient and effective analyses.

[1] More precisely, this strategy can be seen as a *top-down driven* bottom up computation, since some degree of fixpoint, bottom up computation is required in the presence of recursive predicates.

2.2 Ciao

Ciao [18] is a modern, general-purpose, modular, multiparadigm programming language with an advanced programming enviroment, which includes a static analyzer based in abstract interpretation, with which we will be conducting the practical part of this work. In the cited paper a detailed description of Ciao and its more attractive characteristics can be found. In this section we will just introduce the Ciao components most relevant to our work: its assertion framework and its powerful preprocessor, CiaoPP [20].

Assertions are linguistic constructs which allow expressing properties of programs. Syntactically they appear as an extended set of declarations, and semantically they allow talking about preconditions, (conditional-) postconditions, whole executions, program points, etc. The preprocessor, CiaoPP, is a collection of static analysis and debugging tools designed to optimize and verify Ciao programs. Its main component is an abstract interpretation based static analyzer, capable of inferring properties of the predicates and literals of a program, including types, variable instantiation properties (e.g modes or sharing), non-failure and determinacy, computational cost, etc. Both the assertions framework and the preprocessor are deeply connected, and in a way it could be said that the assertions are the interface to the analyzer for the programmer: they allow to specify and guide the analysis, and the result of the analysis is usually outputed as Ciao assertions. Let us see the more relevant details about both:

2.2.1 Ciao Assertions

Ciao assertion language syntax and meaning: For clarity of exposition, we will focus on the most commonly-used subset of the ciao assertion language: `pred` assertions. A detailed description of the full language can be found in Puebla *et al* [33] or in the Ciao reference manual [4].

Such `pred` assertions allow specifying certain conditions on the state (current substitution) that must hold at certain points of program execution. They are very useful for detecting deviations of behavior (symptoms) with respect to such assertions, or to ensure that no such deviations exist (correctness). In particular, they allow stating sets of *preconditions* and *conditional postconditions* for a given predicate. Such `pred` assertions take the form:

$$\texttt{:- pred } Head \texttt{ : } Pre \texttt{ => } Post\texttt{.}$$

where $Head$ is a normalized atom that denotes the predicate that the assertion applies to, and the Pre and $Post$ are conjunctions of "`prop`" atoms, i.e., of atoms whose corresponding predicates are declared to be *properties* [33, 35]. Both Pre and $Post$ can be empty conjunctions (meaning true), in that case they can be omitted. The following example illustrates the basic concepts involved:

Example 2. The following assertions describe different modes for calling a `length` predicate: either for `(1)` generating a list of length `N`, `(2)` to obtain the length of a list `L`, or `(3)` to check the length of a list.

Note also the definition of the `list/1` property (in this case a regular type) in line 7. Other properties (`int/1` –a base regular type, and `var/1` –a mode) are assumed to be

```
:- pred length(L,N) : (var(L), int(N))  => list(L). %(1)
:- pred length(L,N) : (var(N), list(L)) => int(N).  %(2)
:- pred length(L,N) : (list(L), int(N)).            %(3)

:- prop list/1.
list([]).
list([_|T]) :- list(T).
```

loaded from the libraries (`native_props` in Ciao for these properties).

□

Assertion status: Each assertion can be in a *verification status*, marked by prefixing the assertion itself with the keywords `check`, `trust`, `true`, `checked`, and `false`. This specifies respectively whether the assertion is provided by the programmer and is to be checked or to be trusted, or is the output of static analysis and thus correct (safely approximated) information, or the result of processing an input assertion and proving it correct or false. The `check` status is assumed by default when no explicit status keyword is present (as in the examples so far).

Uses of assertions: Assertions find many uses in ciao, ranging from testing [26] to verification [35, 31], documentation [17], or guiding the analysis [5, 16]. For our work, the most relevant is the later: guiding the static analysis. We can specify the abstract queries for the abstract interpretation of the program with *entry* assertions (e.g `:- entry length(L,N) : (var(L), int(N))`). We can provide the true semantics of the program at certain points with *trust* assertions, and the preprocessor will consider that information as truthful during the analysis . We can even describe the semantics of code not yet written or written in other languages, which allows analyzing partially developed code. And of course, we can specify the intended semantics of our programs with normal *check pred* assertions, which will be checked against the result of the analysis.

2.2.2 The Ciao verification framework and the Ciao preprocessor

We now describe the Ciao verification framework [7, 19, 34], which is implemented in the Ciao preprocessor, CiaoPP. Figure 2.2 depicts the overall architecture. Hexagons represent tools and arrows indicate the communication paths among them. It is a design objective of the framework that most of this communication be performed also in terms of assertions. This has the advantage that at any point in the process the information is easily readable by the user. The input to the process is the user program, *optionally* including a set of assertions; this set always includes any assertion present for predicates exported by any libraries used (left part of Figure 2.2).

Run-time checking of assertions: after (assertion) normalization in the *Assertion Normalizer* component (which, e.g., takes away syntactic sugar) the *RT-check* module transforms the program by adding run-time checks to it that encode the meaning of the asser-

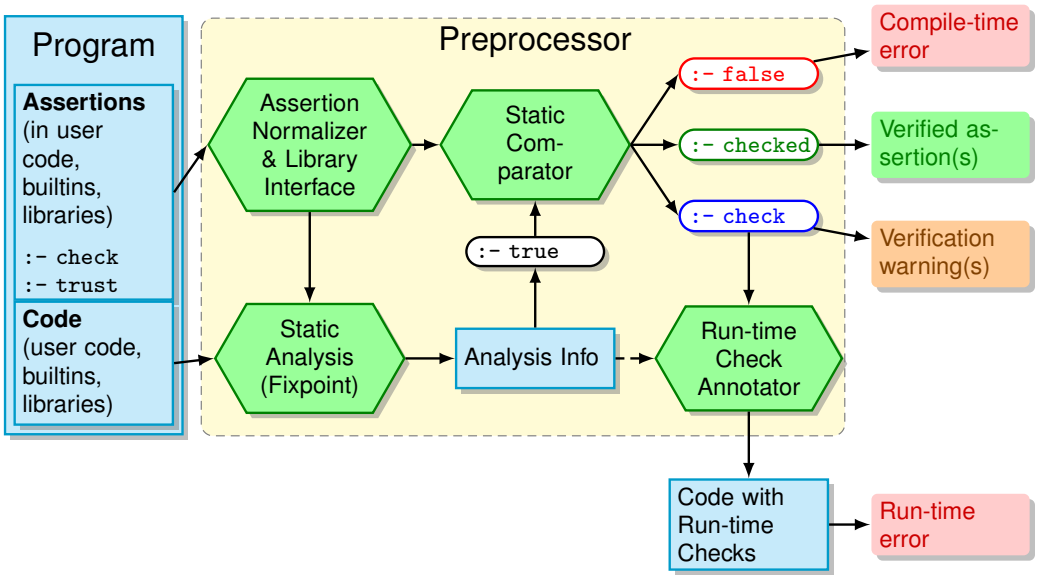


Figure 2.2: The Ciao Verification Framework.

tions (assume for now that the *Comparator* simply passes the assertions through). Note that the fact that properties are written in the source language and *runnable* is very useful in this process. Failure of these checks raises run-time errors referring to the corresponding assertion. *Correctness* of the transformation requires that the transformed program only produce an error if the assertion is in fact violated.

Compile-time checking of assertions: even though run-time checking can detect violations of specifications, it cannot guarantee that an assertion holds. Also, it introduces run-time overhead. The framework performs compile-time checking of assertions by *comparing* the results of static analysis (Figure 2.2) with the assertions. This analysis is typically performed by abstract interpretation or any other mechanism that provides *safe* upper or lower approximations of relevant properties, so that comparison with assertions is meaningful despite precision losses in the analysis. The type of analysis may be selected by the user or determined automatically based on the properties appearing in the assertions. Analysis results are given using also the assertion language, to ensure interoperability and make them understandable by the programmer. As a possible result of the comparison, assertions may be proved to hold, in which case they get `checked` status – Figure 2.2. As another possible result, assertions can be proved not to hold, in which case they get `false` status and a *compile-time error* is reported. Even if a program contains no assertions, it can be checked against the assertions contained in the libraries used by the program, potentially catching bugs at compile time. Finally, and most importantly, if it is not possible to prove nor to disprove (part of) an assertion, then such assertion (or part) is left as a `check` assertion, for which optionally run-time checks can be generated as described above. This can optionally produce a *verification warning*.

2.2.3 CiaoPP abstract domains

CiaoPP has a wide variety of abstract domains to perform analysis with. We will be working mainly with the two most used: *shfr* and *eterms*. The former abstracts information about the sharing and freeness of the variables, and the later about their shape or type. In this section we introduce the basics of both of them, that is, their lattice structure and their abstraction and concretization functions. A more formal definition, including the abstract operations required to perform the analysis, can be found in [29] and [38].

The sharing-freenes domain

The *shfr* domain is defined as $D_\alpha = \wp(\wp(Pvar)) \times \wp(Pvar \to \{G, F, NF\})$, where $\wp$ denotes the powerset of a set, and $Pvar$ the set of variables of interest in a clause. Each abstract substitution in the domain is therefore a 2-tuple. Intuitively, the first element holds the *sharing* information, while the second holds the *freeness* information.

The *sharing* component provides information about *potential* aliasing and variable sharing among the program variables (as well as groundness). The sharing component of the abstract substitution for a clause is defined to be *a set of sets of program variables* in that clause. Informally, a set of program variables appears in the sharing component if the terms to which these variables are bound share a variable.

More formally, a (concrete) substitution for the variables for a clause is a mapping from the set of program variables in that clause (*Pvar*) to terms that can be formed from the universe of all variables (*Uvar*), and the constants and the functors in the given program and query. We consider only idempotent substitutions.

The function *Occ* takes two arguments, θ (a substitution) and U (a variable in *Uvar*) and produces the set of all program variables $X \in Pvar$ such that U occurs in $vars(X\theta)$. The domain of a substitution θ is written as $dom(\theta)$. The instantiation of a term t under a substitution θ is denoted as $t\theta$ and $vars(t\theta)$ denotes the set of all variables in $t\theta$.

Definition 9 (Occ)**.**

$$Occ(\theta, U) = \{X | X \in dom(\theta), U \in vars(X\theta)\}$$

The sharing component of the abstraction of a substitution θ is defined as:

Definition 10 (Abstraction(sharing) of a substitution)**.**

$$\mathcal{A}_{sharing}(\theta) \quad = \quad \{Occ(\theta, U) \mid U \in Uvar\}$$

The *freeness* component of an abstract substitution for a clause gives the mapping from its program variables to an abstract domain $\{G, F, NF\}$ of freeness values i.e. $D_\alpha\text{-}freeness = \wp(Pvar \to \{G, F, NF\})$. X/G means that X is bound to only *ground* terms at run-time. X/F means that X is free, i.e., it is not bound to a term containing a functor. X/NF means that X is *potentially* non-free, i.e., it can be bound to terms which

have functors. The three freeness values are related to each other by the following partial order: $\bot \sqsubseteq F \sqsubseteq NF,\ \ \bot \sqsubseteq G \sqsubseteq NF$

More formally, the freeness value of a term is defined as follows:

Definition 11 (Abstraction(freeness) of a Term)**.**

$$\mathcal{A}_{freeness}(Term) = \begin{cases} if & vars(Term) = \varnothing \quad then\ \ G \\ if & vars(Term) = \{Y\} \wedge Term \equiv Y \quad then\ \ F \\ else & NF \end{cases}$$

Both components of the abstract domain could be abstract domains on their own, and in fact, the sharing component is. In Ciao that domain is called *share* [27], and we will also be using it in our work.

The regular types domain

A *regular type* [12] is a type representing a class of terms that can be described by a deterministic regular term grammar. A *regular term grammar*, or grammar for short, describes a set of finite terms constructed from a finite alphabet $\mathcal{F}$ of *ranked function symbols* or *functors*. A grammar $G = (S, \mathcal{T}, \mathcal{F}, \mathcal{R})$ consists of a set of non-terminal symbols $\mathcal{T}$, one distinguished symbol $S \in \mathcal{T}$, and a finite set $\mathcal{R}$ of productions $T \longrightarrow rhs$, where $T \in \mathcal{T}$ is a non-terminal and the right hand side rhs is either a non-terminal or a term $f(T_1, \ldots, T_n)$ constructed from an n-ary function symbol $f \in \mathcal{F}$ and n non-terminals.

The non-terminals $\mathcal{T}$ are *types* describing (ground) terms built from the functors in $\mathcal{F}$. The concretization $\gamma(T)$ of a non-terminal T is the set of terms derivable from its productions, that is,

$$\begin{aligned} \gamma(T) &= \bigcup_{(T \longrightarrow rhs) \in \mathcal{R}} \gamma(rhs) \\ \gamma(f(T_1, \ldots, T_n)) &= \{f(t_1, \ldots, t_n) \mid t_i \in \gamma(T_i)\} \end{aligned}$$

To be able to describe terms containing numbers and variables we introduce two distinguished symbols **num** and **any**, plus an additional $\bot$. The concretization of **num** is the set of all numbers, the concretization of **any** is the set of all terms (including variables), and the concretization of $\bot$ is the empty set of terms. These symbols are non-terminals but they are considered terminals to the effect of regarding a grammar as deterministic.

Non-terminals, and therefore, grammars are ordered by the relation of *containment*, or type *inclusion*, $(T_1 \sqsubseteq T_2) \Leftrightarrow \gamma(T_1) \subseteq \gamma(T_2)$. Based on this, the set of all grammars $\mathcal{G}$ is a complete lattice with top element **any** and bottom element $\bot$. The least upper bound is given by type *union*, $(T_1 \sqcup T_2)$, and the greatest lower bound by type *intersection*, $(T_1 \sqcap T_2)$.

In an abstract interpretation-based type analysis, a type is used as an abstract description of a set of terms. Given variables of interest $\{x_1, \ldots, x_n\}$, any substitution $\theta = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ can be approximated by an *abstract substitution* $\{x_1 \leftarrow T_{x_1}, \ldots, x_n \leftarrow T_{x_n}\}$ where $t_i \in \gamma(T_{x_i})$ and each type $T_{x_i} \in \mathcal{G}$. We will write abstract substitutions as tuples $\langle T_1, \ldots, T_n \rangle$. Concretization is lifted up to abstract substitutions straightforwardly, except that we consider a distinguished abstract substitution $\bot$ as a representa-

tive of any $\langle T_1, \ldots, T_n \rangle$ such that there is $T_i = \bot$. An ordering on the domain is obtained as the natural element-wise extension of the ordering on types. The domain is a lattice with bottom element $\bot$ and top element $\langle T_1, \ldots, T_n \rangle$ such that $T_1 = \ldots = T_n =$ **any**. The greatest lower bound and lowest upper bound domain operations are lifted also element-wise. Using the adjoin α of γ as abstraction function, it can be shown that $(\wp(\Theta), \alpha, \Omega, \gamma)$ is a Galois insertion, where Θ is the domain of concrete substitutions and Ω that of abstract substitutions.

Other abstract domains

We will also be working with two other abstract domains. One of them is the *definiteness* abstract domain or *def* [25, 13]. For each variable X, it expresses if it is ground or which sets of variables uniquely determine X, that is, making those variables ground makes X ground too. The other is a simple *groundness* domain implemented in the CiaoPP system by Claudio Vaucheret. For each variable, this domain expresses if it is ground or not ground, or potentially both, in a completely analogous way to the *freeness* component in *shfr*. The domain is $D_{gr} = \wp(Pvar \rightarrow \{G, NG, ANY\})$, the relation between the groundness values is $\bot \sqsubseteq G \sqsubseteq ANY$, $\bot \sqsubseteq NG \sqsubseteq ANY$, and the abstraction function is

$$\mathcal{A}_{groundness}(Term) = \begin{cases} if & vars(Term) = \varnothing \quad then \;\; G \\ else & NG \end{cases}$$

Chapter 3

Distances in Abstract Domains

The first natural step towards comparing quantitatively abstract interpretations is to be able to compare quantitatively elements of abstract domains. To that end, we would want to endow those abstract domains with some kind of metric. The purpose of this chapter is to propose such metrics for some abstract domains in CiaoPP, like *shfr* and *eterms*. In order to do that, we first explore some theory and insights for defining distances in arbitrary abstract domains.

3.1 Abstract Distances

Definition 12 (Metric)**.** *A metric on a set S is a function $d : S \times S \to \mathbb{R}$ satisfying:*

- *Non-negativity:* $\forall x, y \in S,\ d(x, y) \geq 0$.
- *Identity of indiscernibles:* $\forall x, y \in S,\ d(x, y) = 0 \iff x = y$.
- *Symmetry:* $\forall x, y \in S,\ d(x, y) = d(y, x)$.
- *Triangle inequality:* $\forall x, y, z \in S,\ d(x, z) \leq d(x, y) + d(y, z)$.

A set S in which a metric is defined is called a metric space.

We can relax some of the above conditions and get new and more general notions of metric spaces. Two of them that are relevant in our work are the following:

A pseudometric is a metric where the two different elements are allowed to have distance 0. We call the left implication of the identity of indiscernibles, weak identity of indiscernibles.

A semimetric is a metric that does not necessarily fulfill the triangle inequality.

We want to extend that definition to abstract domains to define a notion of metric in them, which we will call abstract distance. However, we cannot use that definition as it is. On the one hand, an abstract domain is not an arbitrary set: it has some properties, like a lattice structure and a Galois connection with a concrete domain, which should be reflected in our definition of abstract distance. On the other, we do not know yet if

that definition is general enough: perhaps some of those conditions would be too big of a restriction for our abstract distances.

Let us study then which properties could be expected of an abstract distance, and we will try later to define formally the notion of abstract distance from that. We will include examples of distances fulfilling those properties in the intervals domain, which we saw in example 1

Apart from the common definition of a metric, there are mainly two points to take into account when defining an abstract distance, and from which we will derive new sets of properties.

The first one is the relation of the abstract domain with the concrete domain, and how an abstract distance is interpreted under that relation. In that sense, we can observe that a distance d_α in an abstract domain will induce a distance in the concrete one, as $d(A, B) = d_\alpha(\alpha(A), \alpha(B))$, and vice versa: a distance d in the concrete domain induces an abstract distance in the abstract one, as $d_\alpha(a, b) = d(\gamma(a), \gamma(b))$. We will see later that these induced distances inherit most metric properties. Thus, an abstract distance can be interpreted as an abstraction of a distance in the concrete domain, or as way to define a distance in it. In fact, it is when is interpreted that way that an abstract distance makes more sense from a semantics point of view. From that characterization, and studying what properties the distances in the concrete domain should fulfill, we can derive new properties for our abstract distances.

The other aspect to take into account is the structure of the abstract domain as a complete lattice. The distance defined should respect that structure, and thus we will derive properties that relate that distance with the order relation $\sqsubseteq$ in the domain.

We will see that the set of properties that come from the lattice structure and the set that comes from the relation with a distance in the concrete domain are closely related. That is not surprising at all, since in fact the order in the abstract domain is just induced by the inclusion order in the concrete one, and most desired properties in the concrete domain will be related to that order.

Let us see then what properties should be expected for our abstract distances considering all this.

Metric Properties

First of all, we have all the properties that define a metric. Those are the *non-negativity*, the *identity of indiscernibles*, the *symmetry* and the *triangle inequality*. We would expect them both in the abstract distance and in the related distances in the concrete domain. In fact, the following proposition says that both are almost equivalent.

Proposition 1. *Let us consider an abstract domain D_α, that abstracts the concrete domain D, with abstraction function $\alpha : D \to D_\alpha$ and concretization function $\gamma : D_\alpha \to D$. Both domains are complete lattices and α and γ form a Galois connection. Then:*

(1) If $d_\alpha : D_\alpha \times D_\alpha \to \mathbb{R}$ is a metric in the abstract domain, then $d : D \times D \to \mathbb{R}$, $d(A, B) = d_\alpha(\alpha(A), \alpha(B))$ is a pseudometric in the concrete domain.

(2) If $d : D \times D \to \mathbb{R}$ is a metric in the concrete domain, then $d_\alpha : D_\alpha \times D_\alpha \to$

$\mathbb{R},\ d_\alpha(a,b) = d(\gamma(a),\gamma(b))$ *is a pseudometric in the abstract domain. If the Galois connection is a Galois insertion, then* d *is a full metric.*

Proof. (1) d is a pseudometric:

- Non-negativity: $d(A,B) = d_\alpha(\alpha(A),\alpha(B)) \geq 0$, since d_α is non-negative
- Weak identity of indiscernibles : $d(A,A) = d_\alpha(\alpha(A),\alpha(A)) = 0$, since d_α fulfills the identity of indiscernibles
- Symmetry: $d(A,B) = d_\alpha(\alpha(A),\alpha(B)) = d_\alpha(\alpha(B),\alpha(A)) = d(B,A)$, since d_α is symmetric
- Triangle inequality: $d(A,C) = d_\alpha(\alpha(A),\alpha(C)) \leq d_\alpha(\alpha(A),\alpha(B)) + d_\alpha(\alpha(B),\alpha(C)) = d(A,B) + d(B,C)$, since d_α fulfills the triangle inequality

(2) d_α is a pseudometric: analogous. Besides, if the Galois connection is a Galois insertion, then γ is injective (otherwise, $\exists\ a \neq b \in D_\alpha\ t.q\ \gamma(a) = \gamma(b) \implies \alpha(\gamma(a)) = \alpha(\gamma(b)) \implies a = b$, which is absurd). But then $d_\alpha(a,b) = 0 \implies d(\gamma(a),\gamma(b)) = 0 \implies \gamma(a) = \gamma(b) \implies a = b$, and therefore d_α is a full metric

□

Example 3. $d(I_1,I_2) = \begin{cases} 0\ if\ I_1 = I_2 \\ 1\ otherwise \end{cases}$ *, fulfills trivially all of those properties.*

Order Preserving Properties

It is reasonable to expect, for both $(D,\subseteq)$ and $(D_\alpha,\sqsubseteq)$, that given a chain of elements, closer elements in the chain should have smaller distances to each other. Let us define this property:

Definition 13 (Order-preserving function)**.** *Let* D *be a complete lattice, and* $d : D \times D \to \mathbb{R}$ *a function.*

We will say that d *is order-preserving if* $\forall a,b,c \in D,\ a \sqsubseteq b \sqsubseteq c \implies d(a,b) \leq d(a,c)\ ,\ d(b,c) \leq d(a,c)$.

We will say that d *is strictly order-preserving if* $\forall a,b,c \in D,\ a \sqsubset b \sqsubset c \implies d(a,b) < d(a,c)\ ,\ d(b,c) < d(a,c)$.

The following propositions relates that property for distances in the abstract domain and in the concrete domain:

Proposition 2. *Let us consider a concrete and an abstract domains* D, D_α*, with abstractions and concretization functions* $\alpha : D \to D_\alpha$ *and* $\gamma : D_\alpha \to D$*, which form a Galois connection. Then:*

(1) If $d_\alpha : D_\alpha \times D_\alpha \to \mathbb{R}$ *is order-order preserving, then* $d : D \times D \to \mathbb{R},\ d(A,B) = d_\alpha(\alpha(A),\alpha(B))$ *is too.*

(2) If $d : D \times D \to \mathbb{R}$ *is order-preserving, then* $d_\alpha : D_\alpha \times D_\alpha \to \mathbb{R},\ d_\alpha(a,b) = d(\gamma(a),\gamma(b))$ *is too. If the Galois connection is a Galois insertion and* d *is strictly order-preserving, then* d_α *is too.*

Proof. (1) If $A \sqsubseteq B \sqsubseteq C$, then $\alpha(A) \sqsubseteq \alpha(B) \sqsubseteq \alpha(B)$, since α is monotonic. But then $d(A,B) = d_\alpha(\alpha(A),\alpha(B)) \leq d_\alpha(\alpha(A),\alpha(C)) = d(A,C)$, since d_α is order-preserving.

(2) Order-preserving: Analogous. Strictly order-preserving: The same reasoning, but observing that d_α is now strictly monotonic, since it is injective.

□

Example 4. *In the intervals domain,* $d(I_1,I_2) = \begin{cases} 0, \ if \ I_1 = I_2 = \bot \\ u - l, \ if \ I_1 = (l,u), I_2 = \bot \ or \ I_1 = \bot, I_2 = (l,u) \\ |l_1 - l_2| + |u_1 - u_2|, \ if \ I_1 = (l_1,u_1), I_2 = (l_2,u_2) \end{cases}$ *is strictly order-preserving.*

Diamond Inequalities

If we consider the Hasse diagram of the lattices $(D,\sqsubseteq)$ and $(D_\alpha,\sqsubseteq)$, (even if they are not finite), a few other properties come to our mind.

One could be that, if we have four elements a,b,c,d in the lattice, such that $c \sqcap d \sqsubseteq a \sqcap b \sqsubseteq a \sqcup b \sqsubseteq c \sqcup d$, then $d(a,b)$ should be lower or equal than $d(c,d)$. However, is straightforward to see that this implies that $d(a,b) = d(a \sqcap b, a \sqcup b)$, and the later implies the former if d is order preserving, and that condition is too strong. We can weaken it using $\sqsubset$ instead. Let us formalize all of this:

Definition 14 (Diamond inequalities)**.** *Let D be a complete lattice, and $d : D \times D \to \mathbb{R}$ a function.*

We will say that d fulfills the diamond equality if $\forall a,b \in D, \ d(a,b) = d(a \sqcap b, a \sqcup b)$

We will say that d fulfills the strong diamond inequality if $\forall a,b,c,d \in D, \ c \sqcap d \sqsubseteq a \sqcap b \sqsubseteq a \sqcup b \sqsubseteq c \sqcup d \implies d(a,b) \leq d(c,d)$

We will say that d fulfills the diamond inequality if $\forall a,b,c,d \in D, \ c \sqcap d \sqsubset a \sqcap b \sqsubseteq a \sqcup b \sqsubset c \sqcup d \implies d(a,b) \leq d(c,d)$

Observation 1. *Let D be a complete lattice and $d : D \times D \to \mathbb{R}$ a distance in it. d fulfills the strong diamond inequality $\iff$ d is order-preserving and fulfills the diamond equality.*

Example 5. *The distance defined in example 4 does not fulfill the diamond inequality, since for example if $I_1 = (0,1), I_2 = (2,3)$, then $d(I_1,I_2) = 4$, $d(I_1 \sqcap I_2, d_1 \sqcup I_2) = 3$. However, it is not difficult to see that if fulfills it when $I_1 \sqcap I_2 \neq \bot$.*

As opposed to the previous cases, this property is not necessarily inherited from distance in the concrete or abstract domain to the other. If (D_1,α,D_2,γ) is a Galois connection, then it can be shown that $\alpha(a \sqcup b) = \alpha(a) \sqcup \alpha(b))$ and $\gamma(a \sqcap b) = \gamma(a) \sqcap \gamma(b)$. However, the same can not be said about the meet of the abstraction or the join of the concretization, and that is why those properties are not inherited

Other Properties

Those are the basic properties to be expected in general for all abstract distances. However, there are many more properties that would be too strong of a requirement and

would only be fulfilled by some abstract distances, but that can still be useful to consider, both from a theoretical and practical point of view. In particular, the lattice theory is very rich, and for every type of lattice (modular, distributive, etc), there could be found new interesting properties. Nonetheless, we will only consider one more.

This last property does not come from the mathematical nature or the structure of abstract domains, but from the practical nature of our abstract distances. This property will help us work better with them as a tool and is the following.

Definition 15. *Let D be a complete lattice, and $d : D \times D \to \mathbb{R}$ a distance in D.*
We will say that d is normalized if $\forall a, b \in D,\ d(a, b) \leq 1$, and $d(\bot, \top) = 1$

Abstract distance definition

Let us now consider all those properties one by one. We will decide which of them should be included in our definition of abstract distance, and which others are just desirable or useful from a theoretic point of view, but should not be imposed. Basically, the properties inherited by the abstraction of a distance will be imposed, and all others not, seems it reasonable to expect the abstraction of a distance to be an abstract distance.

- *Non-negativity* and *symmetry*: These two properties will be imposed in our distances. It is true that signed distances could make sense in the context of ordered sets and talk about orientation ($d(a, b) = -d(b, a)$, $d(a, b) > 0 \iff a \sqsupset b$), but we are not going to consider that.

- *Identity of indiscernibles*: We will impose the left implication (i.e the *weak identity of indiscernibles*), but not the other, allowing pseudometrics to be abstract distances. One of the reasons is that it is not inherited by the abstraction of a distance, unless the domains enjoy a Galois insertion. But in general, we could say that dealing with abstract domains involve in some cases operations that lose precision, and that loss of precision will make necessary in some cases to allow two different elements to have distance 0. We will see examples in later sections when we introduce distances that are only pseudometrics.

- *Triangle inequality*: We will not impose this property, but its weaker version in lattices, the *weak triangle inequality*, to be introduced later. The reason for this is that we find the original triangle inequality to be too restrictive for unrelated elements of a lattice, as will become apparent in later sections. As an example, the distance defined in example 4 does not fulfill it.

- *Order-preserving*: We will imposed this property, although not the strict version, since is not necessarily inherited by the abstraction of a distance.

- *Diamond inequalities*: We will not include any of these conditions either, since they are not inherited by the abstraction of a distance.

- Normalized: We will not include this condition either, since is nature is practical and not theoretical. However, we will expect our distances to fulfill it, and it can be noted that any distance can be normalized dividing it by $d(\bot, \top)$.

Therefore, an abstract distance is defined as follows:

Definition 16 (Abstract Distance)**.** *Let us consider an abstract domain D_α, that abstracts the concrete domain D, with abstraction function $\alpha : D \to D_\alpha$ and concretization function $\gamma : D_\alpha \to D$. An abstract distance in D_α is an application $d_\alpha : D_\alpha \times D_\alpha \to \mathbb{R}$ fulfilling the following properties:*

- *Non-negativity:* $\forall a, b \in D_\alpha,\ d_\alpha(a, b) \geq 0$.
- *Weak identity of indiscernibles:* $\forall a \in D_\alpha,\ d_\alpha(a, a) = 0$.
- *Symmetry:* $\forall a, b \in D_\alpha,\ d_\alpha(a, b) = d_\alpha(b, a)$.
- *Order-preserving:* $a, b, c \in D_\alpha,\ a \sqsubseteq b \sqsubseteq c \implies d_\alpha(a, b) \leq d_\alpha(a, c)\ ,\ d_\alpha(b, c) \leq d_\alpha(a, c)$.
- *Weak Triangle inequality:* $\forall a, b, c \in D_\alpha,\ a \sqsubseteq b \sqsubseteq c \implies d_\alpha(a, c) \leq d_\alpha(a, b) + d_\alpha(b, c)$.

Observation 2. *It is straightforward to see that if d_1, d_2 are abstract distances and $\lambda \in \mathbb{R}^+$, then $d_1 + d_2$ and λd_1 are abstract distances too.*

However, as we have seen there is a rich set of properties that an abstract distance can fulfill, and we have just chosen that definition to be the most general possible. In most situations, we will just talk about an abstract distance fulfilling a given subset of those properties, and in general we will expect well-behaved abstract distances to be full metrics and strictly order-preserving, and to fulfill the diamond inequality.

Let us now compile some existing theory and ideas that might help us define distances fulfilling those properties.

3.2 Distances between sets

As we have explained, a distance in an abstract domain between two abstract substitutions corresponds to a distance in the concrete domain between their respective concretizations, which are sets of concrete logic substitutions in our case. Therefore, if we had a distance $d : \wp(\Theta) \times \wp(\Theta) \to \mathbb{R}$ between such sets, we could immediately derive an abstract distance D_α in the abstract domain from it. The problem is, of course, that the concrete substitutions space is infinite, non-metric (a priori) and not structured (a priori), so defining distance between sets there would be much harder than defining distances in the abstract domains, which are in some cases the opposite, and which arise to solve that very same problem (at least the infinity). Furthermore, it could be challenging to operate with the concretizations of elements of abstract domains, since they could be infinite and lead to too costly or non-terminating computations. However, it still makes sense to investigate existing distances between sets in the literature and see how they could connect to our problem. Besides, some abstract domains in Ciao are set-based, and some of the distances listed here might be applicable as abstract distances for them.

Symmetric difference distance

The first distance that comes to our mind is the symmetric difference distance [22], which is defined as follows:

Definition 17 (Symmetric difference distance)**.** *The symmetric difference distance between two finite sets A, B is $d(A, B) = |A \cup B \setminus A \cap B| = |A \cup B| - |A \cap B|$.*

This distance cannot be directly translated to a distance in an abstract domain, since the concretization of an abstract substitution will most likely be not finite. However, we can get still some useful insights from this distance.

Working with the concrete domain as a lattice $(D, \subseteq)$, that distance fulfills that $d(A, B) = d(A \cup B, A \cap B) = d(lub(A, B), glb(A, B))$. That would translate to a distance in the abstract domain defined as $d_\alpha(a, b) = d_\alpha(a \sqcup b, a \sqcap b)$, and then we would only have to define the abstract distance between elements related by $\sqsubseteq$, which seems easier and would take advantage of the *lub* and *glb* operations already implemented for the domain.

Observation 3. *Let $d_\sqsubseteq$ be a partial distance in D_α defined in $\{(a, b) | a, b \in D_\alpha, a \sqsubseteq b\}$ and d_α a distance in D_α defined as $d_\alpha(a, b) = d_\sqsubseteq(a \sqcap b, a \sqcup b)$. If $d_\sqsubseteq$ is non-negative, symmetric, order preserving, and fulfills the weak identity of indiscernibles and weak triangle inequality, then d_α is an abstract distance, which trivially fulfills the diamond inequalities.*

We can take the translation further and assign a finite size $|\ |$ to each element of the abstract domain, which somehow abstracts the size in the concrete domain. Then we could define the distance as $d_\alpha(a, b) = |a \sqcup b| - |a \sqcap b|$.

Proposition 3. *Let us consider an abstract domain D_α and a positive and monotonic application $size : D_\alpha \to \mathbb{R}$. Let us define $d_\alpha : D_\alpha \times D_\alpha \to \mathbb{R}$ as $d_\alpha(a, b) = size(a \sqcup b) - size(a \sqcap b)$. Then d_α is an abstract distance which trivially fulfills the diamond inequality.*

Proof.

- Non-negativity: Trivial, since $a \sqcap b \sqsubseteq a \sqcup b$ and $size$ is monotonic.
- Symmetry: Trivial, since $a \sqcup b = b \sqcup a, a \sqcap b = b \sqcap a$
- Weak identity of indiscernibles: Trivial, since $a = a \sqcap a = a \sqcup a$
- Order preserving: Trivial, since $size$ is monotonic. In fact, if size is strictly monotonic d_α will be strictly order-preserving
- Weak triangle inequality: Trivial. In fact we have the equality: $a \sqsubseteq b \sqsubseteq c \implies d_\alpha(a, c) = size(c) - size(b) + size(b) - size(a) = d(a, b) + d(b, c)$.

□

Observation 4. *In the proposition above, if $size(\bot) = 0$ and $size(\top) = 1$, then d_α is normalized. If $size$ is strictly monotonic, then d_α is strictly order-preserving and fulfills the identity of indiscernibles.*

Example 6. *In the intervals domain, we could assign a size* $size((l,u)) = l-u$, $size(\bot) = 0$*, and define a distance* d *as* $d(I_1, I_2) = size(I_1 \sqcup I_2) - size(I_1 \sqcap I_2)$.

Finally, even if A or B are infinite, if we redefine d as $d(A,B) = (|A \cup B| - |A \cap B|)/|D|$ instead, we could approximate that distance probabilistically, sampling the concrete substitutions space. For a sample of N concrete substitutions, if M of them are contained in $\gamma(a) \cup \gamma(b) \setminus \gamma(a) \cap \gamma(B)$, the $d(a,b)$ would be M/N. The problem with this kind of distances is that they are non-deterministic, which in the context of static analysis is always something to avoid, and that they are not guaranteed to fulfill most properties expected of an abstract distance. However they can still be useful, even if only to test a distance like the one proposed above (where $N \to \infty$, both distances should be similar). We will talk about probabilistic distances in another section.

Jaccard distance

Another known distance between finite sets is the Jaccard [22] distance, defined as follows:

Definition 18. *The Jaccard index of two finite sets* A, B *is* $J(A,B) = |A \cap B|/|A \cup B| \in [0,1]$.

The Jaccard distance between two finite sets A, B *is* $d_J(A,B) = 1 - J(A,B)$.

As we did with the symmetric difference distance, if we define a size $|\,|$ in the abstract domain, we can define an abstract distance as $d_\alpha(a,b) = 1 - (size(a \sqcap b)/size(a \sqcup b)$.

Proposition 4. *Let us consider an abstract domain* D_α *and a positive monotonic application* $size : D_\alpha \to \mathbb{R}$*. Let us define* $d_\alpha : D_\alpha \times D_\alpha \to \mathbb{R}$ *as* $d_\alpha(a,b) = 1 - \frac{size(a \sqcap b)}{size(a \sqcap b)}$*. Then* d_α *is an abstract distance which trivially fulfills the diamond equality.*

Proof.

- Non-negativity: $size$ monotonic $\implies size(a \sqcup b) \geq size(a \sqcap b) \implies d_\alpha(a,b) >= 0$
- Weak identity of indiscernibles: Trivial. If $size$ is strictly monotonic, then the full identity also holds: $d_\alpha(a,b) = 0 \iff size(a \sqcap b) = size(a \sqcup b) \iff a \sqcap b = a \sqcup b \iff a = b$
- Symmetry: Trivial
- Order-preserving: $a \sqsubseteq b \sqsubseteq c \implies d_\alpha(a,b) = 1 - \frac{size(a)}{size(b)} \leq 1 - \frac{size(a)}{size(c)} = d_\alpha(a,c)$, and analogous for $d_\alpha(b,c)$
- Weak triangle inequality: We need to prove that if $a \sqsubset b \sqsubset c$, then $1 - \frac{size(a)}{size(b)} + 1 - \frac{size(b)}{size(c)} \geq 1 - \frac{size(a)}{size(c)}$. That is true if $size(a)size(b) + size(b)size(c) \geq size(a)size(c) + size(b)size(b)$, but that is straightforward to check knowing that $size(a) \leq size(b) \leq size(c)$ and expressing $size(b), size(c)$ as $size(a)+n, size(a)+n+m$, with $n, m \geq 0$.

□

Example 7. *In the interval domains, we could define a distance as* $d(I_1, I_2) = 1 - \frac{size(I_1 \sqcap I_2)}{size(I_1 \sqcup I_2)}$*, where* $size((l,u)) = u - l,\ size(\bot) = 0, size(\top) = \infty$.

We can also adopt a probabilistic approach equivalent to the one proposed with the symmetric difference distance, both to define a new non-deterministic distance and to test the distance above.

Hausdorff distance

Another very known distance between sets, this time not necessarily finite but in a metric space, is the Hausdorff [22] distance. It is defined as follows:

Definition 19. *Let* A, B *be two sets in a metric space* S *with metric* d*. The Hausdorff distance between* A *and* B *is defined as the maximum between* $\sup_{a \in A} \inf_{b \in B} d(a,b)$ *and* $\sup_{b \in B} \inf_{a \in A} d(a,b)$.

Although this distance is defined for infinite sets, it might still not be computable if the sets are not finite, so we can not get directly an abstract distance from it. It also would need the set of concrete substitutions to be metric, so we would need to define a distance in it. Finally, it is undefined for the empty set, and therefore it would be undefined in the abstract domain for $\bot$.

For all these reasons it is harder to derive abstract distances from the Hausdorff distance in the concrete domain. A probabilistic approach would also have much worse behaviour than in the previous cases. However, as we said, some of our domains are set based, and for them we could apply the Hausdorff distance directly.

Example 8. *The intervals domain is partially set-based (it is not completely because the least upper bound is not the union of intervals), so we could use there the Hausdorff distance. It is easy to check that it would be an abstract distance if it was extended to be defined on* $\bot$ *(although it could not be extended in a way that the triangle inequality held).*

3.3 Distances in complete lattices

We can take advantage of the structure of the abstract domains as completed lattices to help defining distances in them. Let us see a few examples.

Distance between related elements of the lattice

It is seems arguably easier to define a distance $d_\sqsubseteq$ between two elements a, b in a complete lattice D when $a \sqsubseteq b$. That distance could later be extended to a general distance $d : D \times D \to \mathbb{R}$ between arbitrary elements as a function of $d_\sqsubseteq(a, a \sqcup b)$, $d_\sqsubseteq(b, a \sqcup b)$, $d_\sqsubseteq(a \sqcap b, a)$, $d_\sqsubseteq(a \sqcap b, b)$, and $d_\sqsubseteq(a \sqcap b, a \sqcup b)$. We have seen an example in previous sections: the distance defined as $d(a,b) = d_\sqsubseteq(a \sqcap b, a \sqcup b)$. This is also worth from an implementation point of view since there are already efficient algorithms for computing the *glb* and *lub* in CiaoPP domains.

Observation 5. *Let D be a complete lattice and $d_{\sqsubseteq} : \{(a,b) | a,b \in D,\ a \sqsubseteq b\} \to \mathbb{R}$, an application which is (with a little definition abuse) non-negative, symmetric, order-preserving and fulfills the weak identity of indiscernibles and triangle inequality. Let us define $d : D \times D \to \mathbb{R}$ as $d(a,b) = \lambda_1(d_{\sqsubseteq}(a, a \sqcup b) + d_{\sqsubseteq}(b, a \sqcup b)) + \lambda_2(d_{\sqsubseteq}d(a \sqcap b, a) + d_{\sqsubseteq}(a \sqcap b, b)) + \lambda_3 d(a \sqcap b, a \sqcup b)$. Then, if $\lambda_1, \lambda_2, \lambda_3 \geq 0$, d is an abstract distance.*

Example 9. *A few examples of distances defined this way would be:*

- *distance-to-lub:* $d(a,b) = \frac{d_{\sqsubseteq}(a, a \sqcup b) + d_{\sqsubseteq}(b, a \sqcup b)}{2}$.
- *distance-to-glb:* $d(a,b) = \frac{d_{\sqsubseteq}(a \sqcap b, a) + d_{\sqsubseteq}(a \sqcap b, b)}{2}$.

Discrete distances

If a complete lattice D has finite height, or finite ascending or descending chains, we could assign a positive weight to each edge of the lattice's Hasse diagram, and define the distance between two related elements $a, b \in D$, with $a \sqsubseteq b$, as the sum of the weights of the edges of the chain $a = d_0 \sqsubseteq d_1 \sqsubseteq ... \sqsubseteq d_n = b$. We could later extend that distance to arbitrary elements as we saw in the previous section.

Example 10. *A simple example would be when all the weights are 1, and then the distance would be the number of "steps" from on element to the other. In the case of the intervals domain, that distance would be again $d_{\sqsubseteq}((l1, u1), (l2, u2)) = |l1 - l2| + |u2 - u1|$, which we have already shown to be an abstract distance.*

Since there are usually more than one of those chains from on element to another, the sum of the weights in all of them should be unique. This might seem like a strong requirement , but actually it is not. Consider for example the interval domains, where the endpoints of the intervals are integers, and where we assign a weight of 1 to each "edge" of the Hasse diagram. Let us consider now two intervals $(l_1, u_1) \sqsubset (l_2, u_2)$, and $n = (l_1 - l_2) + (u_2 - u_1)$. Then it is straightforward to check that there will be $n!$ chains between them, but all of them have sum of weights n. The same could be said for discrete set-based lattices where the join and meet are the union and intersection respectively.

Observation 6. *If the sums of weights of a chain from $a \in D$ to $b \in D$ is unique, the sum of the weights from $\bot$ to each element of D would define a monotonic size in D, and the distance between two elements $a, b \in D$ proposed would just be the difference between their sizes. Therefore it is an abstract distance, as seen in section 3.2*

Valuations

Definition 20. *If D is a lattice, a function $v : D \to \mathbb{R}$ is called a valuation [2] if $\forall a, b \in D,\ v(a) + v(b) = v(a \sqcup b) + v(a \sqcap b)$*

If it is monotonous, then it is said that the valuation is isotone, and if it is strictly monotonous it is said that the valuation is positive.

Some definitions of valuation also impose that $v(\bot) = 0$.

Example 11. *In $\mathbb{R}^n$, with the order $(x_1,\ldots,x_n) \sqsubseteq (y_1,\ldots,y_n) \iff \forall i, x_i \leq y_i$, any linear function $c(x) = c_1x_1 + \ldots + c_nx_n$ is a valuation, isotone if all c_i are non-negative, and positive if they are all positive.*

Proposition 5. *If D is a lattice and $v : D \to \mathbb{R}$ a isotone valuation in it, the function $d : D \times D \to \mathbb{R}$, $d(a,b) = v(a \sqcup b) - v(a \sqcap b)$, defines a pseudometric in D. If v is positive, then it defines a metric. Furthermore, in both cases the function is order preserving, and therefore it is an abstract distance.*

Proof.

- Non-negativity, symmetry, identity of indiscernibles, order-preserving: trivial, and analogous to proof 3.2
- Triangle inequality: We first prove that $d(a \sqcup x, a \sqcup y) + d(a \sqcap x, a \sqcap y) <= d(x,y)$ (1). By definition, the left hand of the expression is $v(a \sqcup x \sqcup y) - v((a \sqcup y) \sqcap (a \sqcup y)) + v((a \sqcap x) \sqcup (a \sqcap y)) - v(a \sqcap x \sqcap y)$, and this is at most $v(a \sqcup x \sqcup y) - v(a \sqcup (x \sqcap y)) + v(a \sqcap (x \sqcup y)) - v(a \sqcap x \sqcap y)$, since $(a \sqcup (x \sqcap y)) \sqsubseteq (a \sqcup x) \sqcap (a \sqcup y)$ and $(a \sqcap x) \sqcup (a \sqcap y) \sqsubseteq (a \sqcap (x \sqcup y))$ (distributive inequalities). But that is equal to $v(a \sqcup x \sqcup y) + v(a \sqcap (x \sqcup y)) - v(a \sqcup (x \sqcap y)) - v(a \sqcap x \sqcap y) = v(a) + v(x \sqcup y) - v(a) - v(x \sqcap y) = d(x,y)$. Now we can prove the triangle inequality: $d(x,y) + d(y,z) = d(x \sqcup y, x \sqcap y) + d(y \sqcup z, y \sqcap z) = d(x \sqcup y, y) + d(y, x \sqcap y) + d(y \sqcup z, y) + d(y, y \sqcap z) \geq d(x \sqcup y \sqcup z, y \sqcup z) + d(y \sqcup z, y) + d(y, x \sqcap y) + d(x \sqcap y, x \sqcap y \sqcap z) = d(x \sqcup y \sqcup z, y) + d(y, x \sqcap y \sqcap z) = d(x \sqcup y \sqcup z, x \sqcap y \sqcap z) \geq d(x \sqcup y, x \sqcap y) = d(x,y)$, where he have used that $d(x \sqcup y \sqcup z, y \sqcup z) \leq d(x \sqcup y, y)$ and $d(x \sqcap y, x \sqcap y \sqcap z) \leq d(y, y \sqcap z)$, which is deduced from (1).

□

This is not something new, we already proposed an instance of this kind of distances when we tried to extend the symmetric difference distance in the concrete domain to the abstract domain. The only difference is the new condition $v(a,b) = v(a \sqcap b) + v(a \sqcup b)$. In fact, valuations become specially relevant as distances in abstract domains when they are considered from that perspective, that means, when the valuation somehow abstracts in the concrete domain a notion of something like size, amount of information or precision. Considering it that way also gives new significance to the definition of valuation, which might originally seem a little arbitrary. In the concrete domain, which is set based, $v(a) + v(b) = v(a \sqcap b) + v(a \sqcup b)$ can be interpreted as the well know formula $|A \cup B| = |A| + |B| - |A \cap B|$.

Therefore, in order to define an abstract distance in an abstract domain, it would be enough to define a valuation that abstracts size in the concrete domain. However, it should be noted that this might not be trivial, since the loss of precision if the join operation will make the equation $v(a) + v(b) = v(a \sqcap b) + v(a \sqcup b)$ more difficult to be fulfilled than in the set-based lattices.

Example 12. *In the interval domains, the length of the intervals could be a good candidate to be a valuation, but it is not. The reason is that the join operation loses precision.*

Consider $I_1 = (0,1),\ I_2 = (2,3)$. *Then* $I_1 \sqcup I_2 = (0,3),\ I_1 \sqcap I_2 = \bot$, *and* $v(I_1) + v(I_2) = 2 \neq 3 = v(I_1 \sqcup I_2) + v(I_1 \sqcap I_2)$.

However, in a set-based domain like share, the cardinal of the set is a valuation.

It can be shown that a candidate to a valuation is in fact one if $v(a \sqcup b) = v(a) + v(b)$ whenever $a \sqcap b = \bot$. It can also be proved that if we define the Jaccard distance for a valuation, it would fulfill the triangle inequality.

Cartesian product of lattices

Definition 21. *We define the cartesian product of complete lattices* $(D_1, \sqsubseteq_1), \ldots, (D_n, \sqsubseteq_n)$ *as* $(D, \sqsubseteq)$, *where* $D = D_1 \times \ldots \times D_n$, *and* $\sqsubseteq$ *is defined as follows:*

$(a_1, ..., a_n) \sqsubseteq (b_1, ...b_n) \iff_{def} \forall i = 1...n,\ a_i \sqsubseteq_i b_i$

With that order, it is straightforward to check that $(D, \sqsubseteq)$ *is a complete lattice and the meet, join, top and bottom result as follows:*

- $(a_1, ..., a_n) \sqcup (b_1, ...b_n) =_{def} (a_1 \sqcup_1 b_2, ..., a_n \sqcup_n b_n)$
- $\top = (\top_1, \ldots, \top_n)$.
- $(a_1, ..., a_n) \sqcap (b_1, ...b_n) =_{def} (a_1 \sqcap_1 b_2, ..., a_n \sqcap_n b_n)$
- $\bot = (\bot_1, \ldots, \bot_n)$.

Many abstract domains are of this nature. The most common cases is when they are a combination of two separate domains or when the properties they express about the variables are independent for each variable and not relational. An example of the former could be *shfr* and an example for the later could be *eterms*.

However, in the case of the cartesian product of abstract domains $D_1, \ldots, D_n$, not all elements in $D_1 \times \ldots \times D_n$ will belong to the new abstract domain. In particular, all elements of the lattice $(a_1, \ldots, a_n)$ for which $\exists i\ t.q\ a_i = \bot$ are identified as the bottom element of the cartesian product lattice, since their concretization is $\varnothing$. And the same would happen to other sets of elements which share concretization. This is needed if there is to be a Galois insertion between the concrete and new abstract domains. Therefore, in the new domain the greatest lower bound would be defined as follows:

$$(a_1, ..., a_n) \sqcap (b_1, ...b_n) = \begin{cases} (a_1 \sqcap_1 b_1, ..., a_n \sqcap_n b_n)\ if\ \forall i = 1...n,\ a_i \sqcap_i b_i \neq \bot_{D_i} \\ \bot\ otherwise \end{cases}$$

Let us study how could we extend distances in lattices to distances in the cartesian product of lattices, taking into account this last consideration. A first approach could be to treat the lattice as a vectorial space, and adapt well-known distances for vectorial spaces.

Example 13. *We could adapt the euclidean distance and define* d *as* $d((a_1, ..., a_n), (b_1, ..., b_n)) = \sqrt{d_1(a_1, b_1)^2 + ... + d_n(a_n, b_n)^2}$.

We could also use any other Minkowski distance and define d *as* $d((a_1, ..., a_n), (b_1, ..., b_n)) = \sqrt[p]{d_1(a_1, b_1)^p + ... + d_n(a_n, b_n)^p}$

Proposition 6. *Let $D_1, \ldots, D_n$ be complete lattices with an abstract distances $d_1, \ldots, d_n$, and let $d_L : L \times L \to \mathbb{R}$, where $L = D_1 \times \ldots \times D_n$, be the Minkowski extension of those distances as defined above. Then d_L is also an abstract distance. Furthermore, if $d_1, \ldots, d_n$ fulfill the identity of indiscernibles or the triangle equality, so does d_L.*

Proof.

- Non-negativity, symmetry, identity of indiscernibles: trivial
- order-preserving: $(a_1, \ldots, a_n) \sqsubseteq (b_1, \ldots, b_n) \sqsubseteq (c_1, \ldots, c_n) \implies \forall i, a_i \sqsubseteq b_i \sqsubseteq c_i$, and therefore, since $\forall i, d_i$ is an abstract distance, then $\forall i, d_i(a_i, b_i) \leq d_i(a_i, c_i)$, and $d(a_1, \ldots, a_n), (b_1, \ldots, b_n) = \sqrt[p]{d_1(a_1, b_1)^p + \ldots + d_n(a_n, b_n)^p} \leq \sqrt[p]{d_1(a_1, c_1)^p + \ldots + d_n(a_n, c_n)^p} = d(a_1, \ldots, a_n), (c_1, \ldots, c_n)$. $d(b^n, c^n) \leq d(a^n, c^n)$ is analogous
- Weak triangle inequality: We only need to check that if $\forall i, d_i(a_i, c_i) \leq d_i(a_i, b_i) + d_i(b_i, c_i)$, then $\sqrt[p]{d_1(a_1, b_1)^p + \ldots + d_n(a_n, b_n)^p} \leq \sqrt[p]{d_1(a_1, c_1)^p + \ldots + d_n(a_n, c_n)^p} + \sqrt[p]{d_1(a_1, c_1)^p + \ldots + d_n(a_n, c_n)^p}$, but that is straightforward to check if we square both sides to p. The triangle inequality is analogous.

□

There are many other distances in the literature that extend a vector of distances to a scalar distance, for example in the field of machine learning, which handles large data sets for which the euclidean distance is not a good metric. However, our lattices will have small dimensions (it is rare to have more than a few variables in an abstract substitution, and in fact many abstract domains are too exponential to work with that), so we will only consider the extensions proposed.

3.4 Distances through other domains

Let us suppose that we have two abstract domains, D_{α_1} and D_{α_2}, that are similar, in the sense that they abstract similar properties of the concrete domain D. Let us suppose too that we already have an abstract distance $d_{\alpha_2} : D_{\alpha_2} \times D_{\alpha_2} \to \mathbb{R}$, and that we want to define another in D_{α_1}. Since both domains are similar, we could try to 'translate' the abstract substitutions from D_{α_1} to D_{α_2}, and use the distance there. We would get a distance $d_{\alpha_1} : D_{\alpha_1} \times D_{\alpha_1} \to \mathbb{R}$ defined as $d_{\alpha_1}(a, b) = d_{\alpha_2}(f(a), f(b))$, where $f : D_{\alpha_1} \to D_{\alpha_2}$ is that 'translation' function. If γ_1 is the concretization function of D_{α_1} and α_2 the abstraction function of D_{α_2}, then we can define that translation function as $f(a) = \alpha_2(\gamma_1(a))$.

Proposition 7. *Let D_1, D_2 be abstract domains, with Galois connections to a concrete domain $(D, D_{\alpha_1}, \alpha_1, \gamma_1)$ and $(D, D_{\alpha_2}, \alpha_2, \gamma_2)$, and let $d_{\alpha_2} : D_{\alpha_2} \times D_{\alpha_2} \to \mathbb{R}$ an abstract distance in D_2.*

Let us define $d_{\alpha_1} : D_{\alpha_1} \times D_{\alpha_1} \to \mathbb{R}$ as $d_{\alpha_1}(a, b) = d_{\alpha_2}(\alpha_2(\gamma_1(a)), \alpha_2(\gamma_1(b)))$. Then d_{α_1} is also an abstract distance. Furthermore, if d_{α_2} fulfills the triangle inequality so does d_{α_1}.

Proof.

- Non-negativity, simmetry, weak identity of indiscernibles, triangle inequality (if d_{α_2} too): Straightforward, analogous to 3.1
- Order-preserving, weak triangle inequality: Straightforward, considering that $\alpha_2 \circ \gamma_1$ is monotonic, since both α_2 and γ_1 are and the composition of monotonic applications is monotonic. Analogous to 3.1

□

Observation 7. *There are some desirable properties of our distances that are not inherited with this construction, like the right implication of the identity of indiscernibles or being strictly order-preserving, and we can ask ourselves under which conditions they would. As we have seen in other sections, the answer is when $\alpha_2 \circ \gamma_1$ in injective. Informally, we could say that this happens when the translation does not lose precision, or in other words when D_{α_1} is strictly more abstract w.r.t D than D_{α_2}. We could model this behaviour saying that there is a Galois insertion $(D_{\alpha_1}, \alpha', D_{\alpha_2}, \gamma')$ such that $\alpha_2 = \alpha_1 \circ \alpha'$, $\gamma_2 = \gamma_1 \circ \gamma'$.*

This technique allows to reuse already defined distances for an abstract domain in another similar abstract domain. If the original domain is strictly more expressive than the other, then no precision will be lost. For example, if we have an abstract distance in *shfr*, it makes sense to define an abstract distance in share as $d_{share}(Sh_1, Sh_2) = d_{shfr}((Sh_1, \top), d(Sh_2, \top))$, where $\top$ refers to the top element of the freeness component ($\{X_1/nf, ..., X_n/nf\}$). It can also be done the other way around, translating a substitution in an abstract domain to another in a strictly more expressive abstract domain. That makes sense when only some of the information that an abstract domain captures is important. For example, if we have an abstract distance in the share domain, and want to compare two abstract substitutions in shfr, and for our application the freeness information is not important, then we can use the distance $d_{shfr}((Sh_1, \_), (Sh_2, \_)) = d_{share}(Sh_1, Sh_2)$, where $\_$ here denotes any possible freeness component. Finally, we can also do this when both abstract domains are not comparable in terms of being more or less abstract or expressive. Some precision will be lost, but if the domains are similar it can make sense from a semantic point of view. An example could be distances in the domain *share* through the domain *def* or viceversa.

But this is not only useful to define an abstract distance in an abstract domain, but also to define a distance $d_{\alpha_{1,2}} : D_{\alpha_1} \times D_{\alpha_2} \to \mathbb{R}$ between different, but similar, abstract domains. We will not formalize the notion of distance between elements of different abstrat domains, but it is an idea with many applications and that will appear again in later chapters. For a situation like that, in which we want to compare two abstract substitutions in different abstract domains, we just have to translate the abstract substitutions to a reference domain, which could be either of the original ones, or other, and compute the distance there. Again, we could compare them with respect to some property they both express in a reference domain more abstract than both; we could compare them in a domain more expressive than both, and in that case it would be like comparing their concretizations; or we could just compare them in a domain not related in that sense to them, and lose precision. We will see examples of this in section 3.5.5.

3.5 Distances in Ciao Domains

In this section we apply some of the ideas of the previous sections to define distances in some common Ciao domains: *gr*, *share*, *shfr* and *eterms*.

3.5.1 Distance in the *groundness* domain

Let us start with one of the simplest domains in Ciao: *gr*, which is defined in section 2.2.3. Let us try to apply the ideas explored in previous sections to define abstract distances in it.

First of all, *gr* is a cartesian product domain in the sense of section 3.3. If we define a distance in the core domain (i.e *gr* for just one variable), then we can follow the ideas defined in that section to extend it to an abstract distance for the domain *gr* with arbitrary number of variables.

The domain for just one variable is the simplest domain possible: apart from $\bot$ and $\top$ (*any*), it has only two unrelated elements: g and ng. Let us call that domain D_{gr}. We will define the following distance for it:

Definition 22 (d_{gr})**.** *The distance* $d_{gr} : D_{gr} \times D_{gr} \to \mathbb{R}$ *is defined as* $d_{gr}(\bot, g) = d_{gr}(\bot, ng) = d_{gr}(g, \top) = d_{gr}(ng, \top) = \frac{1}{2},\ d_{gr}(\bot, \top) = 1$

It is easy to check that this distance can be interpreted, among others, as a discrete distance, and as a valuation, and therefore it is an abstract distance.

We propose the following extension from d_{gr} to a distance $d^n_{gr} : D^n_{gr} \times D^n_{gr} \to \mathbb{R}$, where $D^n_{gr} = D_{gr} \times ...^n \times D_{gr}$.

Definition 23 (d^n_{gr})**.** $d^n_{gr}(\lambda_1, \lambda_2) = d_{\sqsubseteq}(\lambda_1 \sqcap \lambda_2, \lambda_1 \sqcup \lambda_2)$*, where*
$d_{\sqsubseteq}(\{X_1/gr_{1,1}, ..., X_n/gr_{n,1}\},\ \{X_1/gr_{1,2}, ..., X_n/gr_{n,2}\} =$
$\sqrt[2]{d_{gr}(gr_{1,1}, gr_{1,2})^2 + ... + d_{gr}(gr_{n,1}, gr_{n,2})^2}$

This distance is the euclidean extension of d_{gr} as defined in section 3.3, and therefore it is is an abstract distance, since d_{gr} was. On the following we will consider it the standard distance for *gr*, that is just D^n_{gr} when the abstract substitutions operate on n variables, and we will call it the $groundness$ distance.

3.5.2 Distances in the *sharing* domain

Let us try to propose a few distance for the domain *share*, defined in section 2.2.3, applying the insights of previous sections.

Normalized symmetric difference distance

The domain *share* is a finite set-based domain, that is, its elements are finite sets and the join and meet operations are the intersection and union of sets respectively. Therefore a first idea could be to use the symmetric difference distance. As we say in previous

sections, that distance can be interpreted as the distance induced by a valuation, where $v(\lambda) = |\lambda|$, and as a discrete distance based on number of steps, so it is an abstract distance. Let us define it, but normalizing it so its range is $[0, 1]$:

Definition 24 (Normalized symmetric difference distance for *sharing*)**.** *The normalized symmetric difference distance* $d_{share_{nsdd}} : D^n_{share} \times D^n_{share} \to \mathbb{R}$ *is defined as*
$d_{share_{nsdd}}(Sh_1, Sh_2) = (|Sh_1 \cup Sh_2| - |Sh_1 \cap Sh_2|)/2^n$*, where*
n *denotes the number of variables in the domain of the substitutions, and* $|\top| = 2^n$

Observation 8. *As we saw in section 3.3, since* $|\,|$ *is strictly monotonic in shfr,* $d_{share_{nsdd}}$ *is strictly order-preserving and a full metric. It also fulfills trivially the diamond inequalities.*

Jaccard distance

We could also follow the ideas in section 3.2 and use the Jaccard distance.

Definition 25 (Jaccard Distance in *sharing*)**.** *The Jaccard distance* $d_{share_{jac}} : D^n_{share} \times D^n_{share} \to \mathbb{R}$ *is defined as*
$d_{share_{jac}}(Sh_1, Sh_2) = 1 - \frac{|Sh_1 \cap Sh_2|}{|Sh_1 \cap Sh_2|}$

As it is shown in section 3.2, since $|\,|$ is strictly monotonic then $d_{share_{jac}}$ is an abstract distance, which fulfills the diamond inequalities and it strictly order-preserving.

Sum of minimum distances

The previous distance does not take into account the elements of the sets in a *share* substitution. Those sets are also sets, of the variables involved in the susbtitution, and it could be argued that they are not all the same. For example, $d_{share_{nsdd}}(\{\{X, Y, Z\}\}, \{\{X, Y\}\}) = d_{share_{nsdd}}(\{\{X, Y, Z\}\}, \{\{U, V\}\})$, when the first should probably be lower.

We could fix that introducing a metric in $\wp(PVar)$, and using distances for sets in metric spaces, like Hausdorff. As metric in $\wp(PVar)$, we propose again the symmetric difference distance: $d_{PVar}(S_1, S_2) = (|S_1 \cup S_2| - |S_1 \cap S_2|)$.

Hausdorff does not seem like a good distance for this, since it does not take into account the number of points in which two set differs. We propose using instead the *sum of minimum distances*:

Definition 26 (Normalized sum of minimum distances for *sharing*)**.** *The normalized sum of minimum distances* $d_{share_{nsmd}} : D^n_{share} \times D^n_{share} \to \mathbb{R}$ *is defined as*
$d_{share_{nsmd}}(Sh_1, Sh_2) = \frac{\sum_{S_1 \in Sh_1} d'(S_1, Sh_2) + \sum_{S_2 \in Sh_2} d'(S_2, Sh_1)}{\sum_{i=0}^{n} \binom{n}{i}(n-i)}$*, where*
$d'(S, SS) = \min_{S' \in SS} d_{PVar}(S, S')$ *and* n *denotes the number of variables in the domain of the substitutions, and* $|\top| = 2^n$
The denominator is just a factor to normalized the distance so its range is $[0, 1]$

The main problem of this distance is that is undefined for $\bot$ as it is. Several extensions to $\bot$ could be discuss, but we will just say that $d_{share_{nsms}}(\bot, \_) = d_{share_{nsms}}(\_, \bot) = 1$. The proof that this is in fact an abstract distance is not as trivial as others, but it is again straightforward, and not worth to include here.

3.5.3 Distances in the Sharing-Freeness Domain

Let us define now a few distances in the domain *shfr*, defined in section 2.2.3.

The first component of *shfr* is just *share*, so we will recycle the distances there and combine them with some distance in the *freeness* component. That component is in turn very similar to *gr*, so we could define a distance similar to d_{gr}. Let us do so:

Definition 27 (*Freeness* distance)**.** *Let D_{fr} be the freeness component of the domain shfr, that is, the lattice $(\{\bot, G, F, NF\}, \sqsubseteq)$, where $G \sqsubseteq NF, F \sqsubseteq NF$.*

We define the freeness distance $d_{fr} : D_{fr} \times D_{fr} \to \mathbb{R}$ as $d_{fr}(\bot, NF) = d_{fr}(G, F) = 1$, $d_{fr}(\bot, G) = d_{fr}(\bot, F) = d_{fr}(G, NF) = d_{fr}(F, NF) = 1/2$

We extend that distance to a distance $d^n_{fr} : D^n_{fr} \times D^n_{fr} \to \mathbb{R}$, where $D^n_{fr} = D_{fr} \times ...^n \times D_{fr}$, as:

$d^n_{fr}(\lambda_1, \lambda_2) = d_{\sqsubseteq}(\lambda_1 \sqcap \lambda_2, \lambda_1 \sqcup \lambda_2)$, *where*

$d_{\sqsubseteq}(\{X_1/fr_{1,1}, ..., X_n/fr_{n,1}\}, \{X_1/fr_{1,2}, ..., X_n/fr_{n,2}\} = \frac{\sqrt{d_{fr}(fr_{1,1}, fr_{1,2})^2 + ... + d_{fr}(fr_{n,1}, fr_{n,2})^2}}{\sqrt{n}}$

We will combine the distances in both components using the euclidean distance, and obtain:

Definition 28. *The distances $d_{shfr_{nsdd}}, d_{shfr_{jac}}, d_{shfr_{nsmd}} : D^n_{shfr} \times D^n_{shfr} \to \mathbb{R}$ are defined as:*

$$d_{shfr_{nsdd}}((Sh_1, Fr_1), (Sh_2, Fr_2)) = \frac{\sqrt{d_{share_{nsdd}}(Sh_1, Sh_2)^2 + d_{fr}(Fr_1, Fr_2)^2}}{\sqrt{2}}$$

$$d_{shfr_{jac}}((Sh_1, Fr_1), (Sh_2, Fr_2)) = \frac{\sqrt{d_{share_{jac}}(Sh_1, Sh_2)^2 + d_{fr}(Fr_1, Fr_2)^2}}{\sqrt{2}}$$

$$d_{shfr_{nsmd}}((Sh_1, Fr_1), (Sh_2, Fr_2)) = \frac{\sqrt{d_{share_{nsmd}}(Sh_1, Sh_2)^2 + d_{fr}(Fr_1, Fr_2)^2}}{\sqrt{2}}$$

Since all the original distances are normalized abstract distances, the resulting distances are too, as shown in section 3.3.

3.5.4 Distances in the Regular Types Domain

Let us try to define an abstract distance for the the domain *eterms*, defined in section 2.2.3. This domain is just the cartesian product of $\mathcal{G}$ by itself n times, where n is the number of variables in the clause, so we can concentrate on defining a distance d_G between two types $T_1, T_2 \in \mathcal{G}$, and that distance can later be extended to an abstract distance d_{eterms} in D_{eterms}, using the notions seen in section 3.3. Likewise, to that end we can define a distance $d_{\sqsubseteq}$ between T_1 and T_2 when $T_1 \sqsubseteq T_2$, and d_G can later be defined as $d_G(T, T') = d_{\sqsubseteq}(T \sqcap T', T \sqcup T')$, as proposed in sections 3.2 and 3.3. Therefore, we will focus on defining just $d_{\sqsubseteq}$, trying to apply some of the ideas explored in previous sections.

One of them was to define distances based on the Hasse diagram or the number of steps from an element to another, but it can not be applied in this case since $\mathcal{G}$ has not finite height or ascending chains. Another was to define a monotonic size or a valuation for types in $\mathcal{G}$. If we work with the normal form $G = (S, \mathcal{T}, \mathcal{F}, \mathcal{R})$ of a type T, which is unique for regular types, we could try to define a size for T from it. However this is not

an easy task. We could try to define the size as a function of the number of terminals, non-terminals and productions, but that would really be measuring the complexity of a grammar, not its size. For example, both the smallest and biggest grammars are quite simple: the grammar for $\bot$ has no productions and the grammar for $\top$ has just one. It would also be difficult to enforce monotony with this approach, while keeping distances bounded. We propose some ideas:

- We could define $size(T) = |\mathcal{R}_f| + |\mathcal{R}_\infty||\mathcal{F}|$, where we have partitioned the set of productions $\mathcal{R}$ into the set of finite productions $\mathcal{R}_f$ and the set of not finite or recursive productions $\mathcal{R}_\infty$. This size is not monotonic, so it will not give us a distance. However, it could be applied to the difference between two types, which CiaoPP is able to compute, and that would result in a semimetric.
- We could extend the definition of abstract distance to a function $d : D \times D \to \mathbb{R} \cup \{\infty\}$, and follow the same approach, defining a monotonic size $size : D \to \mathbb{R} \cup \infty$, with $size(any) = \infty$. For example, the size of a type could be the number of productions from the initial symbol of the grammar, or ∞ is the production $S \to$ **any** belong to $\mathcal{R}$.
- We could try to bound the growth of the size will keeping it monotonic with a transformation like $size \to 1 - \frac{1}{size}$, with $size \geq 1$, although this particular transformation grows close to 1 too fast.

Another approach, proposed in section 3.2, could be to define a metric in the substitutions space Θ, extend it to a Hausdorff distance $d_{\mathcal{H}}$ in the concrete domain, and derive from it an abstract distance $d_\alpha(a,b) = d_{\mathcal{H}}(\gamma(a),\gamma(b))$ in the abstract domain, as seen in section 3.1. We will do something similar, defining a metric d_{term} in the space of ground and non-cyclic (i.e finite) terms and extending it to a distance between two types $T_1, T_2 \in \mathcal{G}$ as a Hausdorff distance. We define d_{term} recursively as follows:

$$d_{term}(f(x_1,\ldots,x_n), g(y_1,\ldots,y_m)) = \begin{cases} if & f/n \neq g/m \quad then \ 1 \\ else & p\sum_{i=1}^{n} \frac{1}{n} d_{term}(x_i, y_i) \end{cases}$$

where $p \in (0,1)$ is a parameter of the distance. The resulting distance between two types $G_1 = (S_1, \mathcal{T}_1, \mathcal{F}_1, \mathcal{R}_1)$ and $G_2 = (S_2, \mathcal{T}_2, \mathcal{F}_2, \mathcal{R}_2)$ would be $d'(S_1, S_2)$, which is defined recursively, and with a little abuse of notation, as follows:

$$d'(S_1,S_2) = \begin{cases} if & \exists\,(S_1 \to f(T_1,\ldots,T_n)) \in \mathcal{R}_1 \wedge \nexists (S_2 \to f(T'_1,\ldots,T'_n)) \in \mathcal{R}_2 \quad then \ 1 \\ if & \exists\,(S_2 \to f(T_1,\ldots,T_n)) \in \mathcal{R}_2 \wedge \nexists (S_1 \to f(T'_1,\ldots,T'_n)) \in \mathcal{R}_1 \quad then \ 1 \\ else & max\{p\sum_{i=1}^{n} \frac{1}{n} d(T_i,T'_i) \mid (S_1 \to f(T_1,\ldots,T_n)) \in \mathcal{R}_1 \wedge \\ & \qquad\qquad (S_2 \to f(T'_1,\ldots,T'_n)) \in \mathcal{R}_2\} \end{cases}$$

Observation 9. *As it is, the distance is not well-defined. On the one hand, it is undefined for the type $\bot$. However it can be easily extended as $d(\bot,\_) = d(\_,\bot) = 1$. On the other, its computation may be non-terminating, since the recursion is not necessarily limited (although it can be proved that the resulting infinite sums would always converge). But this problem can be easily overcome pruning the recursion at some point, for example at*

some depth or when the weight (the accumulated $\frac{p}{n}$ factor) goes below some threshold, or not so easily computing the convergent sums.

It is straightforward to check that d_{terms} is a metric, and therefore the induced Hausdorff distance $d_{\mathcal{H}}$ between set of terms is too. We conclude that $d_{\mathcal{G}} : \mathcal{G} \times \mathcal{G} \to \mathbb{R}$, $d_{\mathcal{G}}(T_1, T_2) = d_{\mathcal{H}}(\gamma(T_1), \gamma(T_2))$ is an abstract distance, as shown in section 3.1, and noting that $d_{\mathcal{H}}$ is of course order-preserving. However, we will not prove that the proposed distance d' is in fact $d_{\mathcal{G}}$

Finally, we propose one more idea, which we will not develop. We could work with a graph representation of types, with terminals and non terminals as nodes and productions as edges, and use existing distances between graphs in the literature. This approach could also be used to define distances less semantic and more topological, that is, distances that work with the shape of a type modulo renamed functors.

3.5.5 Distance between elements of different aliasing-mode domains

To conclude this section, we revisit the ideas introduced in section 3.4 and propose informally a few new abstract domains and distances in them to use as reference to compare abstract substitutions in different aliasing-mode abstract domains, like *shfr, share* or *def*. Before that, we should remark that any of the native aliasing-mode domains and distances proposed for them so far would accomplish that task. For example, we can compare abstract substitutions in different domains with respect to the groundness information they capture, using *gr* as reference domain and implementing the translation function from those domains to *gr*. An of course, the domains could be the same and then the result would be a proper abstract distance. The domains proposed are the following:

- **All possible aliasing-mode information**

  We have explained in previous sections that a good way to compare two abstract substitutions is to compare their corresponding sets of logic substitutions in the concrete domain, but that a problem is that those sets will not be finite. However, the mode and sharing possibilities between the variables of those substitutions are finite. We can abstract away the things that make that set infinite (e.g terms shape) and consider only whether variables are free, ground or partially instantiated, and how variables depend on each other. We would get an new abstract domain, set based but finite, and strictly more expressive than any other native aliasing/mode domain. Therefore, we could define simple set-based distances in it and use it as a reference domain for computing distance in those aliasing/mode domains.

- **Pair-sharing**

  A reference domain useful for abstract domains that express sharing between variables (*share, shfr, def*) is pair-sharing, which indicates which pairs of variables possibly share. It is similar to *share*, but with pair of variables instead of sets. It is neither more nor less abstract than the other aliasing-mode domains (although, e.g., *share/shfr* capture independence and grounding dependencies [8]), but it is

still useful, since we can define set-based distances in it and it is easy to implement transformations from other domains to it.

- **Graphs**

  Another very expressive aliasing/mode abstract domain could be the following. We keep in one component the information about a variable being ground, partially instantiated, or free. And in another we express the sharing and dependencies between variables with a graph. If the graph has labels in the edges indicating which variable introduces the sharing, then this domain is strictly more expressive than any other Ciao aliasing/mode abstract domain, and if not it is still quite expressive. The advantage of this representation is that it allows us to use existing and established distances between graphs from the literature.

Chapter 4

Distances between analyses

Now that we have distances in the abstract domains, we can start to work quantitatively with abstract interpretations. For some applications, like semantic search [14] –looking for code that complies with a given set of assertions– those abstract distances suffice, but for some others we need more. For example, in some situations we might want to compute the distance between two complete analyses of a program. That is precisely what we pursue in this chapter: to extend our abstract distances to distances between analyses, i.e., distances between whole abstract interpretations of a program.

We will consider only the case of two analyses of the same program with the same entries. The case of two analyses over different programs, even if one is just a slight modification of another, is much more complex and out of the scope of this work. In the last section of this chapter we will devote a few lines to it, exposing the challenges involved and some possible initial approaches, as well as some speculative applications.

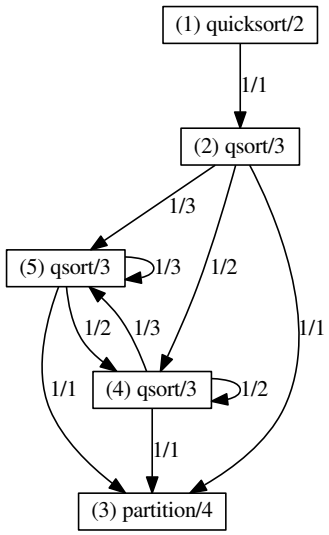
Before trying to define distances between analyses, let us recall first what is the object we will be working with. As we saw in the preliminary chapter (2), an analysis, or rather its result, for a given entry, is the abstract resolution tree of the program for that entry. That is like the concrete resolution tree for a concrete query, but with abstract call and success substitutions over the underlying abstract domain instead of concrete ones. As in the concrete case, the tree might be infinite, but only because it repeats call patterns (e.g a descendant of a node expresses a call to the same predicate and with the same abstract substitution than its ascendant), and then the tree can be represented in a finite way as a cyclic tree or a graph.

Example 14. Let us consider as an example the simple program in Figure 4.1, which uses an *entry* assertion to specify the initial abstract query of the analysis [34]. If we analyze it with the *groundness* domain, the result can be represented with a graph as seen in Figure 4.2.

That graph is a finite representation of an infinite abstract and-or tree. The nodes in the graph correspond to and-nodes $\langle L, \lambda^c, \lambda^s \rangle$ in the analysis tree, where the literals L, abstract call substitutions λ^c and abstract success substitutions λ^s are specified below the graph. The labels in the edge indicate to which program point each node corresponds: if one node is connected to its predecessor by an arrow with label i/j, then that node corresponds to the j-th literal of the i-th clause of the predicate indicated by the predecessor.

```
:- module(quicksort,[quicksort/2],[assertions]).

:- use_module(partition,[partition/4]).

:- entry quicksort(Xs,Ys) : (ground(Xs), var(Ys)).

quicksort(Xs,Ys) :-
    qsort(Xs,Ys,[]).

qsort([X|Xs],Ys,TailYs) :-
    partition(Xs,X,L,R),
    qsort(R,R2,TailYs),
    qsort(L,Ys,[X|R2]).
qsort([],Ys,Ys).
```

Figure 4.1: Quicksort, using difference lists.



(1) $\langle quicksort(Xs,Ys),\ \{Xs/g,Ys/ng\},\ \{Xs/g,Ys/g\}\rangle$
(2) $\langle qsort(Xs,Ys,[]),\ \{Xs/g,Ys/ng\},\ \{Xs/g,Ys/g\}\rangle$
(3) $\langle partition(Xs,X,L,R),\ \{Xs/g,X/g,L/ng,R/ng\},\ \{Xs/g,X/g,L/g,R/g\}\rangle$
(4) $\langle qsort(Xs,Ys,Zs),\ \{Xs/g,Ys/ng,Zs/g\},\ \{Xs/g,Ys/g,Zs/g\}\rangle$
(5) $\langle qsort(Xs,Ys,[Z|Zs]),\ \{Xs/g,Ys/ng,Z/g,Zs/g\},\ \{Xs/g,Ys/g,Z/g,Zs/g\}\rangle$

Figure 4.2: Analysis of `quicksort/2`.

The or-nodes are left implicit.

□

We would like to compare two analyses like the one above. We propose in the following a few methods to do so, that vary in their complexity and applications. We will explain them and give some examples, but as opposed to the previous chapter we will not do it in a formal and theoretic way, i.e., we will not define formally the notion of distance between analyses and abstract interpretations or develop theory for it, nor will we define formally each distance. Since this part of the work is more speculative, and still at the idea stage, such formalization is left as future work. Note, however, that if we did, we would define these distances as non-negative, symmetric, and fulfilling the left implication of the identity of indiscernibles.

4.1 First approach: top results of the analysis

As explained in the preliminary sections, the input to the static analysis is a set of entries or abstract queries (abstract call substitutions for some predicates). The analysis computes the abstract execution tree for those abstract queries.

Let us consider that we only have one of those entries. The call substitution for the top node of the abstract execution tree will be the one indicated by that entry, and the success substitution for that node will be the "abstract answer." The rest of the tree will indicate how that answer is obtained.

If we want to compare two analyses of the same program with the same entry, we could use only that abstract answer or top result to compare them and leave aside the rest of the tree. We would be throwing away part of the analysis, but it is fine for a naive attempt to compare them. In fact, in some situations we might consider that abstract answer to be the true result of the analysis and the rest of the tree just the resolution of that result, the same way as when one only cares about the answers/solutions of a predicate in the concrete case, and not their computation, i.e., the input-output semantics of the program. Therefore, for those situations this naive distance could be even more suitable than other more complete ones. An example could be if we are trying to compute the difference between the characteristics of the interface of a program that two different analyses infer.

If the analyses have instead more entries, but still the same ones, we could compute the same way a vector of distances, and obtain a scalar distance for it. This could be done with a simple average or the euclidean distance, as we did in section 3.3, or adapting any other well-known vectorial distance in the literature. Additionally, if the user specifies a weight for each entry, the final distance could just be the weighted average of the distances for those entries.

Observation 10. *If in a distance defined that way, the sum of the weights is 1, and the abstract distance we are using for the underlying domain is normalized, then the distance between the analyses will still fulfill the convenient property of being in the interval [0,1].*

A distance defined this way is a pseudo-metric, if the underlying abstract distance is too. However, it will never be a full metric, since it is easy to find two different analyses with the same top success substitution.

4.2 Second approach: program points

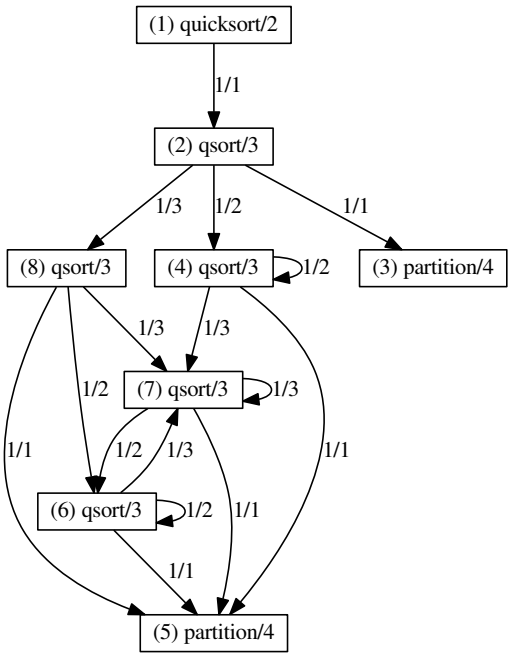
We wonder now how could we define a distance that takes the whole analysis into account. If the result of the two analyses has the same shape, and only differs in the abstract substitutions, which would all refer to the same graph points in both analyses, then we could compute again the distances point to point (i.e., abstract substitution to abstract substitution) using our abstract distance in the underlying abstract domain. We would get a set of distances with some structure, and compute a scalar distance from it. For example, if we consider both abstract trees computed by the analyses for a given entry and they had the same shape, we would get a tree of distances (the same tree, but with distances instead of abstract substitutions), and try to get a distance from it.

The problem is that even if the program is the same, the shape of that resulting tree of the analysis is not. It is not only its representation as a finite cyclic tree what is different (one analysis could repeat call patterns for the same predicate and therefore cycle before the other, etc), but also the abstract execution tree itself might differ. Therefore we need to find a common representation for both analyses. In the previous section we did already that: since the program is the same, the predicates are the same, and we considered only the top result (i.e., the root of the tree) for them, throwing away most of the analyses information. In this section we propose another approach: considering the information inferred by the analyses for each program point.

Even if the analyses are different, the program points are the same since both of them are analyses of the same program. We could compile all the information inferred by each analysis for each program point, and forget about its context in the abstract tree. In fact, that is the purpose of the analysis in many situations, for example when it is done to optimize code but without specialization (creating procedure versions): in that case all that matters is the substitutions with which a program point can be called or succeeds, and not which traces lead to those calls. Later we could compute distances program point to program point, and think how to get a scalar distance from that.

To compile all the analysis information for a program point, all we need to do is collect all pairs of abstract call and success substitutions that appear for that program point in the analysis tree, and get from it a new pair: the new call and success substitutions will be the join of those call and success substitutions respectively, so it over-approximates all possible call and success substitutions for the program point.

Example 15. The analysis shown in Figure 4.2 has only one triple $\langle L, \lambda^c, \lambda^s \rangle$ for each program point. Let us consider a different analysis for the same program (Example 4.1), in which there is no information about the imported predicate `partition/4`, and therefore the analysis needs to assume the most general abstract substitution on success for calls to that predicate. Figure 4.3 shows the result of the analysis in the same manner as Figure 4.2 does. We observe that this time there are program points which have more that one triple in the analysis. Let us denote each program point as `P/A/N/M`, where that represents the `M`-th literal of the `N`-th clause of the predicate `P/A`. The correspondence between program points and analysis nodes is the following:



(1) $\langle quicksort(Xs,Ys),\ \{Xs/g,Ys/ng\},\ \{Xs/g,Ys/any\}\rangle$
(2) $\langle qsort(Xs,Ys,[]),\ \{Xs/g,Ys/ng\},\ \{Xs/g,Ys/any\}\rangle$
(3) $\langle partition(Xs,X,L,R),\ \{Xs/g,X/g,L/ng,R/ng\}),\ \{Xs/g,X/g,L/any,R/any\})\rangle$
(4) $\langle qsort(Xs,Ys,Zs),\ \{Xs/any,Ys/ng,Zs/g\}),\ \{Xs/any,Ys/any,Zs/g\})\rangle$
(5) $\langle partition(Xs,X,L,R),\ \{Xs/any,X/any,L/ng,R/ng\}),\ \{Xs/any,X/any,L/any,R/any\})\rangle$
(6) $\langle qsort(Xs,Ys,Zs),\ \{Xs/any,Ys/ng,Zs/any\},\ \{Xs/any,Ys/any,Zs/any\}\rangle$
(7) $\langle qsort(Xs,Ys,[Z|Zs]),\ \{Xs/any,Ys/ng,Z/any,Zs/any\}),\ \{Xs/any,Ys/any,Z/any,Zs/any\})\rangle$
(8) $\langle qsort(Xs,Ys,[Z|Zs]),\ \{Xs/any,Ys/ng,Z/g,Zs/any\},\ \{Xs/any,Ys/any,Z/g,Zs/any\}\rangle$

Figure 4.3: Analysis of `quicksort/2`.

`quicksort/2/0` (entry)	`quicksort/2/1/1`	`qsort/3/1/1`	`qsort/3/1/2`	`qsort/3/1/3`
(1)	(2)	(3), (5)	(4), (6)	(7), (8)

The resulting single triples $\langle L, \lambda^c, \lambda^s \rangle$ for each program point will be the following:

`quicksort/2/0` (entry)	(1)	$\langle quicksort(Xs,Ys),\ \{Xs/g,Ys/ng\},\ \{Xs/g,Ys/any\}\rangle$
`quicksort/2/1/1`	(2)	$\langle qsort(Xs,Ys,[]),\ \{Xs/g,Ys/ng\},\ \{Xs/g,Ys/any\}\rangle$
`qsort/3/1/1`	(3) '$\sqcup$' (5)	$\langle partition(Xs,X,L,R),\ \{Xs/any,X/any,L/ng,R/ng\}),\ \{Xs/any,X/any,L/any,R/any\})\rangle$
`qsort/3/1/2`	(4) '$\sqcup$' (6)	$\langle qsort(Xs,Ys,Zs),\ \{Xs/any,Ys/ng,Zs/any\},\ \{Xs/any,Ys/any,Zs/any\}\rangle$
`qsort/3/1/3`	(7) '$\sqcup$' (8)	$\langle qsort(Xs,Ys,[Z\|Zs]),\ \{Xs/any,Ys/ng,Z/any,Zs/any\},\ \{Xs/any,Ys/any,Z/g,Zs/any\}\rangle$

□

After doing that, two challenges remain: defining a distance between each program point, and defining a scalar distance from the distances between program points.

The first one is not as simple as in the previous section, since the two call substitutions

might differ, so we cannot just use the distance between the success substitutions. However we could just compute the distance between the two call substitutions and the two success substitutions independently and combine them. A simple average or the euclidean distance would suffice.

The scalar distance could be obtained from a weighted average of the distances between the program points, as we did in the previous section. The weights could be again specified by the user, constant, or even inferred from the program structure or the (analogous) structure of the analysis graph.

Example 16. Let us compare the two analyses shown in Figures 4.2 and 4.3 for the program introduced in Example 4.1. We already have their representation as one triple $\langle L, \lambda^c, \lambda^s \rangle$ for each program point. The distances for each program point, computed as the average of the distance between its abstract call substitution and the distance between its abstract success substitution, is the following:

`quicksort/2/0` (entry)	`quicksort/2/1/1`	`qsort/3/1/1`	`qsort/3/1/2`	`qsort/3/1/3`
0.354	0.354	0.427	0.454	0.467

The final distance between the analysis could be the average of all of them, 0.411. Alternatively, we could assign different weights to each program point taking into account the structure of the program, and use a weighted average as final distance. For example, we could assign the weights of the table below, which would yield the final distance 0.378.

`quicksort/2/0` (entry)	`quicksort/2/1/1`	`qsort/3/1/1`	`qsort/3/1/2`	`qsort/3/1/3`
$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{1}{12}$

□

Observation 11. *A distance defined this way will be at most a pseudo metric. It cannot be a full metric since we lose information when we consider only the program points.*

There are many lines of investigation and experimentation for this approach of considering only the program points for the distances. We have just considered and sketched the most basic. In particular, another interesting variant could be working with the whole set of pairs of abstract call-success substitutions, instead of computing their least upper bound. The problem of defining distances between those sets would be closely related to the problem of defining distances between multivariant analyses, which is out of the scope of this work. It could be also interesting to explore more complex ways of combining the distance for the program points considering the structure of the program.

4.3 Third approach: whole abstract execution tree

In the previous section we said that the abstract tree computed by two analyses for the same program and the same entry would not necessarily have the same shape, and therefore the abstract resolution tree was unfit for the approach proposed there. However, after a few transformations that do not change at all the meaning of the analyses, the two

abstract trees can be made to have the same shape. Let us see it (note that we are talking about the actual tree, not its finite representation as a graph or cyclic tree).

Given the abstract execution tree of two analyses for the same program and entry, if we traverse the shape of the trees following the abstract execution flow, it is easy to see that the shape can only differ for one reason: a node in one tree is unreachable in the other. For example, an abstract call substitution might be compatible (i.e., unifies abstractly) with a clause, but the corresponding abstract call in the other tree does not, and therefore the or-node for that clause does not appear. Or the abstract success substitution for one literal in a clause is bottom in one analysis (i.e., that goal will always fail with those call substitutions) and the next literal in the clause is not explored, but that does not happen in the other analysis, where the and-node for that literal is explored and does appear in the tree. In both cases we can just add the missing nodes with bottom abstract call and success substitutions, recursively if needed, and the shape of the tree would be the same again for both analyses. That does not change the meaning of the analyses, and in fact it could be argued that those nodes belong to the real abstract tree and the way to represent them is just omitting them.

We can therefore assume that both trees have the same shape, and adopt an analogous approach to that of the previous section. That is, compute distances point to point (abstract substitution to abstract substitution) in the tree, and define a scalar distance from the resulting tree of distances. We propose the following: first, define a distance between each node of the tree, and then, define the final distance as a weighted average of those node distances.

For the first we could just compute the distance between the call substitutions and between the success substitutions, and combine them with the euclidean distance or just the average. For the later we propose the following algorithm: (1) We start with a weight of 1 for all the nodes, a factor $p \in (0, 1]$, and the root node. (2) We have a weight w and a node. We assign a weight of $p \cdot w$ to that node, split a weight of $(1-p)w$ among its children, and go to (2) for each children with its corresponding part of that splitted weight. If an or-node has no children, we just assign the whole weight w to it.

The idea of this assignment of weights is that, depending on the value p, we consider more relevant the distance between the upper nodes than the distance between the deeper ones. We care more about the more external behavior of the program, so to speak, but we do not want to miss any of the information of the analysis. In fact, indeed we do not miss any information, and it is easy to foresee that if this distance was formally defined, it would be the first of the ones proposed to fulfill the identity of indiscernibles [1], that is, that two analyses have distance 0 only if they are actually the same.

Example 17. Let us apply this algorithm to compute the distance between the two analyses shown in Figures 4.2 and 4.3. The first levels of the analysis and-or tree are shown in Figure 4.4. The or-nodes are omitted (both in the tree and in the weight assignment). Each and-node is a quintuple (P, Id_1, Id_2, D, W): P is the predicate corresponding to that program point, I_1 is the identifier of the node in analysis 4.2 corresponding to that and-node, I_2 is the analogous in analysis 4.3, D is the distance between the two nodes,

[1] If the underlying distance does too, of course.

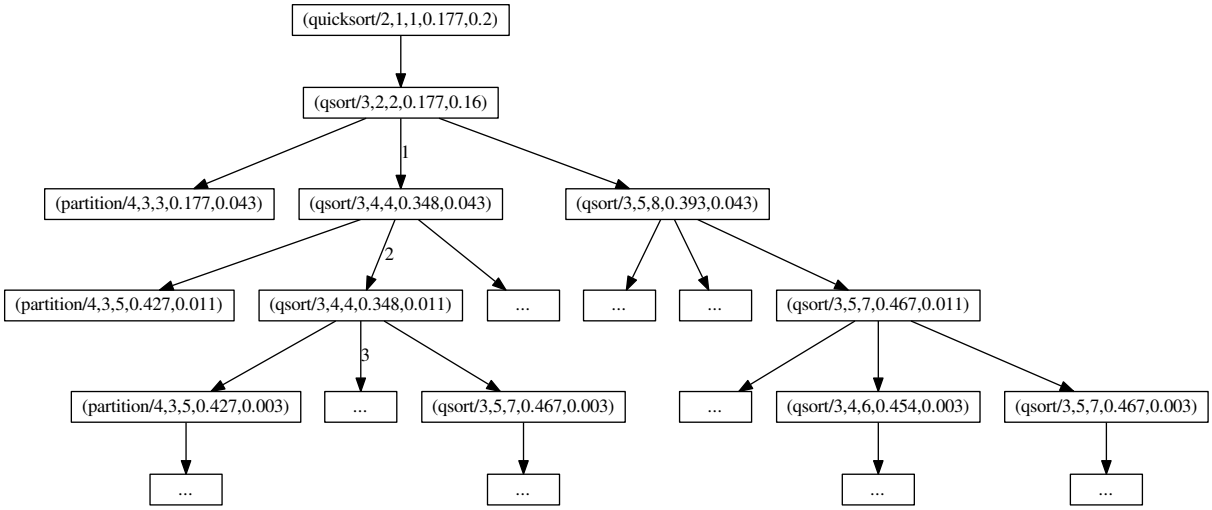


Figure 4.4: 3rd approach: whole abstract execution tree

and W is the corresponding weight to that node. We use a factor $p = \frac{1}{5}$, and the average of the distance between the call substitutions and the distance between the success substitutions as distance between nodes, using the *groundness* distance (seen in Section 3.5.1) as underlying abstract distance for the domain.

□

Clearly, and as seen in Figure 4.4, that weight assignment is non terminating when the tree is not finite, as it will most likely be. We propose the following solutions:

- At some depth of the tree, just assign all the weight to the current node, and do not visit its children.
- When the weight to be assigned is below some threshold, assign all of it to the current node and stop deepening into the tree.
- Compute the "fixpoint". It is clear that the non-finiteness of the process comes from visiting the same pairs of nodes again and again with ever decreasing weights, and the sum of those weights is convergent. An algorithm similar to the fixpoint algorithm used during the analysis and with slight adaptations could be applied to find the limit of that sum.

  Example 18. Let us consider the previous example and Figure 4.4. If we follow the tree through the edges labelled 1,2,3..., we observe that we are visiting the same node over and over with decreasing weights $0.043, 0.011, 0.003 \ldots = w\frac{1}{5} + w\frac{4}{5}\frac{1}{3}\frac{1}{5} + w\frac{4}{5}\frac{1}{3}\frac{4}{5}\frac{1}{3}\frac{1}{5} + \ldots$, where $w = 1\frac{4}{5}\frac{1}{1}\frac{4}{5}\frac{1}{3}$. The sum of those weights converges and can be easily calculated (it is $\frac{1}{5}w\sum_{i=0}^{\infty}(\frac{4}{5}\frac{1}{3})^i = \frac{1}{5}w\frac{15}{11}$).

  □

Observation 12. *If the underlying abstract distance is a metric, the distance proposed between analyses will be a metric too. If the underlying abstract distance is normalized, the distance proposed will have range* $[0, 1]$ *too.*

All the above has been defined for analyses with just one entry. When dealing with several entries but still the same for both analyses, the considerations from previous sections are enough.

4.4 Distance between analyses of different programs

Comparing analyses of two different programs is much more complicated, due to the absence of that common structure of the analysis that we have exploited in the previous sections. In some situations we could adopt an approach like in Section 4.1 and care only about the top result, but again, depending on the application this may or may not be too naive (for example, again it could be useful for semantic code search or interface comparison). We can also adopt an approach like in Section 4.3 and compute node distances as long as the two analyses have the same structure, and return the maximum node distance when they differ. This would be a decent measure of program similarity, but of course it also has limitations, and the same could be said about a naive adaptation of the approach of Section 4.2.

In this scenario, abstract distances tolerant to syntactic noise, like permutations or different number of variables in the domain of the substitutions, or renamed functors or predicates, become more relevant too. But that poses an additional challenge, since it is difficult to combine distances that behave well both semantically and syntactically.

Clearly, as we consider more general classes of programs to compare, the problem becomes broader and more difficult to treat in a general way. However, there are some interesting applications for it that we should mention, and to which it might be possible to find more ad-hoc solutions that might still rely on our abstract distances in the domains. Some of those applications could be comparing the semantics of two programs, detecting or measuring the effectiveness of an obfuscation, measuring the loss of precision in the analysis after optimizations or obfuscations, detecting plagiarism, etc. We return to this topic of applications in Chapter 6.

Chapter 5

Evaluation and Experiments

In order to evaluate the potential of the distances that we have defined in real applications, we need to evaluate their quality. In this chapter we do precisely that, first testing individually that their behavior is the expected and their implementation correct, and later applying them in some small experiments.

5.1 Evaluation of proposed distances

Checking the expected behavior of the proposed distances is challenging, first of all because that expected behavior is not really defined. We have imposed certain properties on our abstract distances, which he have made sure they hold by construction for each distance defined, and otherwise their expected behavior can vary depending on their application.

Therefore, in order to evaluate our abstract distances, we will just compute, for each domain, all proposed and implemented distances for a predefined set of pairs of abstract substitutions, and check manually that the implementation is correct, their complexity is acceptable, and the results fall within the expected margins. We show the results for the domain *shfr*. The tested distances are $d_{shfr_{nsdd}}$ (section 3.5.2), $d_{shfr_{nsmd}}$ (3.5.2) and $d_{shfr_{jacc}}$ (3.5.2). The computed distances can be seen in table 5.1.

The behavior of the analysis distances is more complicated to evaluate manually, since it implies understanding the whole semantics and analyses of the programs in which they are applied, which is not as simple and visual as in the case of the abstract domains. And we would need to that for several programs to get reliable conclusions. Doing it automatically is also too complicated, since we don't have a programmable notion of a distance being accurate. Therefore we evaluate the behavior of the distances between analyses directly with the experiments.

	λ_1	λ_2	$d_{shfr_{nsdd}}$	$d_{shfr_{nsmd}}$	$d_{shfr_{jac}}$
(1)	$\bot$	$\top$	1	1	1
(2)	λ_2^a	λ_2^b	0.392	0.373	0.637
(3)	λ_3^a	λ_3^b	0.713	0.791	0.713
(4)	λ_4^a	λ_4^b	0.94	1	0.94
(5)	λ_5^a	λ_5^b	0.222	0.212	0.408
(6)	λ_6^a	λ_6^b	0.418	0.412	0.54
(7)	λ_7^a	λ_7^b	0.408	0.312	0.408
(8)	λ_8^a	λ_8^b	0.177	0.118	0.177

$\lambda_2^a =$ $(\{\{X\},\{X,Y\},\{Y\}\},$ $\{X/nf,Y/nf,Z/g\})$	$\lambda_6^a =$ $(\{\{X\}\},\{X/f,Y/g,Z/g\})$
$\lambda_2^b =$ $(\{\},\{X/g,Y/g,Z/g\})$	$\lambda_6^b =$ $(\{\},\{X/g,Y/g,Z/g\})$
$\lambda_3^a =$ $(\top,\{X/nf,Y/nf,Z/nf\})$	$\lambda_7^a =$ $(\top,\{X/nf,Y/nf,Z/nf\})$
$\lambda_3^b =$ $(\{\},\{X/g,Y/g,Z/g\})$	$\lambda_7^b =$ $(\{\{X\},\{X,Y\},\{Y\}\},$ $\{X/nf,Y/nf/Z/g\})$
$\lambda_4^a =$ $(\top,\{X/f,Y/f,Z/f\})$	$\lambda_8^a =$ $\top$
$\lambda_4^b =$ $(\{\},\{X/g,Y/g,Z/g\})$	$\lambda_8^b =$ $(\{\{X\},\{X,Y\},\{X,Y,Z\},$ $\{X,Z\},\{Y\}\},\top)$
$\lambda_5^a =$ $(\{\{X\}\},\{X/nf,Y/g,Z/g\})$	
$\lambda_5^b =$ $(\{\},\{X/g,Y/g,Z/g\})$	

Table 5.1: Test of distances in *shfr*.

5.2 Experiments

Another way to test our distances is to propose experiments and check that their results match the expected intuitions. We have done so for a few experiments regarding measuring and comparing the precision of the analyses. There is a wide variety of analysis algorithms in CiaoPP (i.e., different fixpoints, widenings, domains, etc), that vary in precision and cost, and their qualitative relative precisions are well known. However, even if it is known that an analysis is by construction more precise than other, there are no tools in CiaoPP to quantify and measure that difference, which would be necessary for example to establish which analysis has the best trade-off between cost and precision. The experiments we have proposed deal with this particular problem, and our results both let us evaluate the quality of our distances and mark a first step towards solid techniques for measuring analysis precision.

The idea, as we said, is to apply this to different native CiaoPP analysis algorithms for the same program. However, in CiaoPP those analyses will mostly differ in things that might be unnecessarily complex to consider for an experiment, like different fixpoints approximations or widenings. Working with them would force us to deal with the internal CiaoPP implementation, and would make understanding the results harder. Therefore,

Module/Predicate		Ver. 1	Ver. 2	Ver. 3	Ver. 4	Ver. 5
quicksort	trusts	2	1	1	0	-
qsort/2	precision	0.0	0.422	0.817	0.817	-
aiakl	trusts	2	1	1	0	-
goal/2	precision	0	0	0.212	0.212	-
rdtok	trusts	3	1	1	1	-
read_tokens/2	precision	0.001	0.001	0.003	0.003	-
progeom	trusts	4	3	2	1	0
pds/2	precision	0.0	0.252	0.297	0.297	0.297

Table 5.2: (Loss of) precision of the analysis over different variants of the same benchmark with different trust assertions.
In the table, there are two rows for each benchmark, the first indicates how many trust assertions each version has and the second shows the measured (loss of) precision.

for our experiments we have chosen to artificially create different analyses with different precision for our programs, whose differences are more visual and simpler to understand.

In section 5.2.1, we propose inducing a gain or loss of precision in the same analysis for the same program, having different versions of the program that just differ in that they have different *trust* assertions [16] to indicate to the analysis the true semantics of the program at some points. In section 5.2.2 we translate whole analyses performed over different aliasing-mode domains, to analyses over the domain *gr*, and compare their precision there.

5.2.1 Analysis Precision and Trust Assertions

As we mentioned in the preliminary section, Ciao assertions with status *trust* indicate to the analyzer that it can consider the information of the assertions as truthful during the analysis. As a result, the analysis yields more precise results where it could otherwise lose precision, maybe because there are side-effects in the program, because no information is available for imported code, or just because it is unable of inferring the actual semantics of a predicate.

For this experiment, we focus on the second point: the analysis in CiaoPP is modular, but in the current default settings it does not analyze other modules to infer the semantics of imported predicates. However, if that semantics is specified with trust assertions, either in the module that uses the predicate or the module that exports it, the analysis will work with it as if it was information inferred by the analyzer itself.

Our experiment consists of the following steps. First, we get a program written across different modules, without cyclic dependencies between them. Then we analyze each module separately, and annotate the semantics computed by the analysis with trust assertions (CiaoPP can do this automatically). If we now analyzed again the main module, the analysis would gain precision due to those trust assertions. We will do exactly this, but performing the analysis several times, using different combinations of trust assertions

```
:- module(_, [qsort/2], [assertions, nativeprops]).

:- use_module(partition, [partition/4]).
:- use_module(mylists, [my_append/3]).

:- entry qsort(As,Bs) : ( ground(As), var(Bs) ).

qsort([],[]).
qsort([X|L],SortL) :-
        partition(L,X,L1,L2),
        qsort(L1,SortL1),
        qsort(L2,SortL2),
        my_append(SortL1,[X|SortL2],SortL).

:- trust pred qsort(As,Bs)
         : ( mshare([[Bs]]), var(Bs), ground([As]) )
        => ground([As,Bs]).

:- trust pred partition(_A,_1,Left,Right)
         : ( mshare([[Left],[Right]]), var(Left), var(Right),
             ground([_A,_1]) )
        => ground([_A,_1,Left,Right]).

:- trust pred my_append(_A,L,_B)
         : ( mshare([[_B]]), var(_B), ground([_A,L]) )
        => ground([_A,L,_B]).
```

Figure 5.1: `qsort` program with `trust` assertions.

(commenting and uncommenting them). We thus produce a set of analyses with different (losses of) precision, which we will measure as the distance to the most precise analysis (the one with all trust assertions). If our distances are correct, the measured precision will be increasing as the subsets of commented *trust* assertions grow, and hopefully those distances will be related somehow to the number of trust assertions used.

The benchmarks used can be found in the code repository, at the path `src/experiments/trusts/benchs`. The analyses have been done over the domain *shfr*, using the abstract distance $d_{shfr_{ndss}}$ and the third distance between analyses proposed (same as example 17). The results are shown in Table 5.2. Here we explain one of them: the analysis of the program *qsort.pl* with different trust assertions.

Example 19. The program is really simple and is shown in Figure 5.1.

The true semantics of the program can be read in the *trust* assertions. The loss of precision of the analysis with different combinations of those trust assertions is the following:

- None: 0.861
- *partition/4* assertion: 0.389
- *my_append/3* assertion: 0.861
- Both assertions: 0.0

Note how a significant amount of precision is lost when none of the assertions is used (recall that with the distances we are working with, the maximum is 1). This is because basically all that the actual semantics and the analysis have in common is some call patterns (first argument of *qsort/2* is ground, third and forth arguments of *partition/4* are free variables, and third argument of *my_append/3* is a free variable too). On the success patterns, they always differ almost as much as they could: on the actual semantics everything is ground, and on the analysis everything is almost the $\top$ substitution of the domain *shfr*, since the analysis cannot infer anything. Adding the trust assertions *my_append/3* changes nothing, since when the analysis reaches the point where that predicate is used in the program, precision has already been lost in *partition/4* and the analysis is already working with $\top$ abstract substitutions.

□

We conclude that the loss of precision when using strictly modular analyses is too big, as we suspected, and is not worth the gain in analysis run time. However, Ciao offers solutions for this. To begin with, all Ciao libraries are annotated with trust assertions, so the programmer does not need to worry when using native code. Also, multi-modular analysis [32, 6, 37] and incremental analysis [15, 21, 36] are available in Ciao. But in general, this experiment proves the great utility of *trust* assertions in Ciao as a way to improve analysis precision, specially considering that there are sources of precision loss, like side-effects or foreign code, that cannot be avoided without explicit use of those assertions.

5.2.2 Analysis Precision vs. Analysis Cost

As we mentioned in section 3.4, different abstract domains can be compared with respect to some of the information they express. For this experiment, we propose to analyze the same program over the domains *shfr*, *share*, and *def*, and measure the precision of the analyses with respect to the groundness information they infer.

In order to do that, we translate the analyses from the original domain to *gr*. That translation is done replacing all the abstract substitutions λ_D in the original domain in the analysis AND-OR tree, with abstract substitutions λ_{gr} in *gr*. The translation is defined as $\lambda_{gr} = \alpha_{gr}(\gamma_D(\lambda_D))$, where α_{gr} is the abstraction function of the domain *gr* and γ_D is the concretization function of the domain D. That means that the new abstract substitution in *gr* is an over-approximation of the abstract substitution in D, and the result is an AND tree that still over-approximates the semantics of the program.

Then, we compute the actual semantics of the program. This can be done specifying it with trust assertions if it is known, or approximating it as the intersection of all the translated analysis, which implies a loss of precision but allows in exchange to automate the process for arbitrary benchmarks with unknown semantics.

Finally, we can measure the (loss of) precision of an analysis as the difference between the actual semantics in *gr* and the semantics in *gr* computed by the analysis. In our case, we also measure the analysis run-time cost for data completion.

Table 5.3 shows the results, for a few benchmarks, some of them developed ad-hoc and others classical benchmarks for static analysis in logic programming. The bench-

Module	Predicate	*shfr*		*share*		*def*	
		prec.	**cost**	**prec.**	**cost**	**prec.**	**cost**
simple.pl	p/1	0.0	8.20	0.000	7.40	0.000	7.44
qsort.pl	quicksort/2	0.0	7.56	0.140	5.72	0.140	5.76
qsort_v2.pl	quicksort/2	0.0	5.88	0.087	5.64	0.087	6.00
append.pl	append/3	0.0	5.84	0.395	5.62	0.395	5.76
bid.pl	bid/4	0.0	7.24	0.126	7.04	0.126	7.20
boyer.pl	tautology/1	0.0	12.32	0.022	10.44	0.023	10.24
deriv.pl	d/3	0.0	7.36	0.121	5.84	0.121	5.96
fib.pl	fib/2	0.0	6.80	0.047	5.12	0.057	5.00
grammar.pl	parse/2	0.0	5.76	0.062	5.36	0.062	5.64
hanoiapp.pl	shanoi/5	0.0	5.56	0.137	5.48	0.137	5.48
mmatrix.pl	mmultiply/3	0.0	7.12	0.144	5.72	0.299	5.72
occur.pl	occurall/3	0.0	7.52	0.144	5.96	0.144	5.84
peephole.pl	peephole_opt/2	0.0	12.88	0.123	11.48	0.135	10.60
progeom.pl	pds/2	0.0	7.36	0.072	5.60	0.086	5.96
qsortapp.pl	qsort/2	0.0	7.28	0.155	5.36	0.155	5.64
query.pl	query/1	0.0	7.68	0.069	6.20	0.069	6.12
tak.pl	tak/4	0.0	7.92	0.150	6.28	0.218	5.64
zebra.pl	zebra/7	0.0	7.04	0.287	7.00	0.287	6.56

Table 5.3: Loss of precision of the analysis over the domains *shfr*, *share* and *def* analyses with respect to *groundness* semantics, and runtime cost in milliseconds of those analysis.

marks and code for the experiment can be found in the code repository, in directories `src/experiments/precision-vs-cost/benchmarks/` and `src/experiments/precision-vs-cost/` respectively. The results align with our a-priori knowledge: that *shfr* is strictly more expressive than both other domains, and that *share* is more precise than *def*, although their precision inferring groundness is usually the same. Therefore, when the best-approximation semantics is computed as the intersection of all three analyses, it is just going to be the semantics computed by *shfr*, and that is why the loss of precision for that domain is always 0. However, we still compute that distance as a sanity check. The difference in precision between *shfr* and the two others comes from the fact that *shfr* can sometimes detect non-groundness (when a variable is free), but it does not detect groundness better than *share*.

Chapter 6

Conclusions

Abstract interpretation is a powerful static analysis technique, but part of its full potential is not reached because of the lack of techniques to work with it in a quantitative way, a shortcoming that has not been properly addressed yet. We have tried to approach this problem, exploiting the structure of the abstract domains and logic programs to define metrics among abstract interpretations that respect their semantic nature and allow us to work quantitatively with them. We have developed a basic theory of metrics in abstract domains, and proposed algorithms to extend them to distances between the abstract semantics of programs. We have implemented some of the proposed distances and experimented with them in the context of the Ciao preprocessor. As a result we have been able to measure the precision of different analysis of some programs in CiaoPP.

We believe that our results are encouraging and that, with some further work and refinement, the distances defined can be integrated into CiaoPP and enhance some components of the Ciao environment. Furthermore, since CiaoPP is capable of transforming foreign languages to Horn clauses, all those applications of our distances can be extended to other programming languages and paradigms. At the same time, clearly the proposed approaches cannot cover all the fields and problems that would in theory benefit from quantitative techniques for abstract interpretation. We feel that the range of applications in which those quantitative techniques could be applied is too wide and varied to be covered by a single approach, and that some very useful applications of distances between abstract interpretations are in need of further investigation. In particular, more work is needed in comparing abstract interpretations for two non-identical programs, a problem with promising potential applications. However, we believe that our idea of metrics in abstract domains could be key in all that future work.

6.1 Applications and Future Work

In this work we have focused on only one of the applications of abstract distances: measuring precision in analysis, computing the distance between the actual and the inferred semantics. However, there are many applications and open lines of investigation that have not been discussed, and where the ideas developed in this document could prove to be useful.

First of all, measuring analyses has many applications in itself, apart from making it possible to establish the quality of different analysis approaches (i.e different domains, fixpoints, widenings...). For example, it allows checking how different programming methodologies or optimizations affect the analysis precision. We have already shown it for the use of trust assertions, but it could be worth investigating it too for things like the use of cuts, higher order programming, catch-all clauses, folding or unfolding predicates of goals, transformations to better exploit indexing, use of runtime checks, etc). In the same line, this technique could also be used to measure the effectiveness of an obfuscation, by checking how much precision loss it induces.

Another application could be found in semantic code browsing [14]. Ciao implements a tool that allows to search code in a library specifying its (external) semantics in some abstract domain, instead of using other traditional techniques like signature matching. That tool could be enhanced with the use of abstract distances, showing similar results when an exact match is not found. However, to fully exploit our distances in this scenario, we would need to find a way to deal with permutations or different number of variables in a clause. The same ideas applied here could also be applied to the field of program synthesis, and we could adopt a static analysis approach to the problem of checking how close a generated program is to a specified behaviour.

Other applications could be found in the field of software metrics. Some of our distances, like the ones based on valuations (section 3.3), are based on a notion of size or norm for an abstract substitution. Those norms can have applications on their own. For example, if they are extended to whole abstract interpretations, a small *share* norm could indicate that a program is more parallelizable, and a small *eterms* norm can indicate that a program is better-typed and therefore safer.

Finally, our abstract distances could be used in a different approach to measuring precision of the analysis: an approach that measures it from construction. We could investigate how to measure an analysis precision by measuring the imprecision introduced at each step of the analysis by an abstract operation, like the greatest lower bound of the result for all the clauses at some point of the AND-OR tree.

Bibliography

[1] Krzysztof R. Apt. *From Logic Programming to Prolog*. Computer Science. Prentice Hall, 1997.

[2] Garrett Birkhoff. *Lattice Theory 3rd ed*. American Mathematical Society, 1967.

[3] M. Bruynooghe. A Framework for the Abstract Interpretation of Logic Programs. Technical Report CW62, Department of Computer Science, Katholieke Universiteit Leuven, October 1987.

[4] F. Bueno, D. Cabeza, M. Carro, M. V. Hermenegildo, P. López-García, and G. Puebla-(Eds.). The Ciao System. Ref. Manual (v1.13). Technical report, School of Computer Science, T.U. of Madrid (UPM), 2009. Available at `http://ciao-lang.org`.

[5] F. Bueno, D. Cabeza, M. V. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.

[6] F. Bueno, M. García de la Banda, M. V. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey. A Model for Inter-module Analysis and Optimizing Compilation. In *Logic-based Program Synthesis and Transformation*, number 2042 in LNCS, pages 86–102. Springer-Verlag, March 2001.

[7] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. V. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press.

[8] F. Bueno and M. García de la Banda. Set-Sharing is not always redundant for Pair-Sharing. In *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, number 2998 in LNCS, Heidelberg, Germany, April 2004. Springer-Verlag.

[9] F. Bueno, M. García de la Banda, and M. V. Hermenegildo. Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. *ACM Transactions on Programming Languages and Systems*, 21(2):189–238, March 1999.

[10] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *ACM Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM Press, 1977.

[11] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astreé analyzer. In *14th European Symposium on Programming, ESOP 2005*, pages 21–30, April 2005.

[12] P.W. Dart and J. Zobel. A Regular Type Language for Logic Programs. In *Types in Logic Programming*, pages 157–187. MIT Press, 1992.

[13] M. García de la Banda and M. V. Hermenegildo. A Practical Approach to the Global Analysis of Constraint Logic Programs. In *1993 International Logic Programming Symposium*, pages 437–455. MIT Press, October 1993.

[14] I. Garcia-Contreras, J. F. Morales, and M. V. Hermenegildo. Semantic Code Browsing. *Theory and Practice of Logic Programming, 32nd Int'l. Conference on Logic Programming (ICLP'16) Special Issue*, 16(5-6):721–737, October 2016.

[15] I. Garcia-Contreras, J. F. Morales, and M. V. Hermenegildo. Towards Incremental and Modular Context-sensitive Analysis. In *Technical Communications of the 34th International Conference on Logic Programming (ICLP 2018)*, OpenAccess Series in Informatics (OASIcs). Dagstuhl Press, July 2018. (Extended Abstract).

[16] I. Garcia-Contreras, J.F. Morales, , and M. V. Hermenegildo. Multivariant Assertion-based Guidance in Abstract Interpretation. In *Pre-proceedings of the 28th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'18)*, September 2018.

[17] M. V. Hermenegildo. A Documentation Generator for (C)LP Systems. In *International Conference CL 2000*, volume 1861 of *LNAI*, pages 1345–1361. Springer-Verlag, July 2000.

[18] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J.F. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming*, 12(1–2):219–252, January 2012. http://arxiv.org/abs/1102.5497.

[19] M. V. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.

[20] M. V. Hermenegildo, G. Puebla, F. Bueno, and P. Lopez-Garcia. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The

Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

[21] M. V. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, March 2000.

[22] Michael Levandowsky and David Winter. Distance between sets. *Nature*, 234:34–25.

[23] J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.

[24] K. Marriott and H. Søndergaard. Semantics-based dataflow analysis of logic programs. *Information Processing*, pages 601–606, April 1989.

[25] K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. Technical report 93/7, Univ. of Melbourne, 1993.

[26] E. Mera, P. López-García, and M. V. Hermenegildo. Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In *25th Int'l. Conference on Logic Programming (ICLP'09)*, volume 5649 of *LNCS*, pages 281–295. Springer-Verlag, July 2009.

[27] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*, pages 166–189. MIT Press, October 1989.

[28] K. Muthukumar and M. Hermenegildo. Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, April 1990.

[29] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables through Abstract Interpretation. pages 49–63. MIT Press, 1991.

[30] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.

[31] P. Pietrzak, J. Correas, G. Puebla, and M. V. Hermenegildo. Context-Sensitive Multivariant Assertion Checking in Modular Programs. In *LPAR'06*, number 4246 in LNCS, pages 392–406. Springer-Verlag, November 2006.

[32] P. Pietrzak, J. Correas, G. Puebla, and M. V. Hermenegildo. A Practical Type Analysis for Verification of Modular Prolog Programs. In *PEPM'08*, pages 61–70. ACM Press, January 2008.

[33] G. Puebla, F. Bueno, and M. V. Hermenegildo. An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, 2000.

[34] G. Puebla, F. Bueno, and M. V. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. V. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[35] G. Puebla, F. Bueno, and M. V. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, number 1817 in LNCS, pages 273–292. Springer-Verlag, March 2000.

[36] G. Puebla and M. V. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *International Static Analysis Symposium (SAS 1996)*, number 1145 in Lecture Notes in Computer Science, pages 270–284. Springer-Verlag, September 1996.

[37] G. Puebla and M. V. Hermenegildo. Some Issues in Analysis and Specialization of Modular Ciao-Prolog Programs. In *Special Issue on Optimization and Implementation of Declarative Programming Languages*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.

[38] C. Vaucheret and F. Bueno. More Precise yet Efficient Type Inference for Logic Programs. In *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *Lecture Notes in Computer Science*, pages 102–116. Springer-Verlag, September 2002.