Abstract Domains in Constraint Programming

Series Editor
Narendra Jussien

Abstract Domains in Constraint Programming

Marie Pelleau

ISTE PRESS

ELSEVIER

To Eliott, Joaquim and Ambrym

First published 2015 in Great Britain and the United States by ISTE Press Ltd and Elsevier Ltd

ISTE Press Ltd
27-37 St George's Road
London SW19 4EU
UK

www.iste.co.uk

Elsevier Ltd
The Boulevard, Langford Lane
Kidlington, Oxford, OX5 1GB
UK

www.elsevier.com

Notices

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods, professional practices, or medical treatment may become necessary.

Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

British Library Cataloguing in Publication Data
A CIP record for this book is available from the British Library
Library of Congress Cataloging in Publication Data
A catalog record for this book is available from the Library of Congress
ISBN 978-1-78548-010-2

Printed and bound in the UK and US

Preface

Constraint programming aims at solving hard combinatorial problems, with a computation time increasing in practice exponentially. Today, the methods are efficient enough to solve large industrial problems in a generic framework. However, solvers are dedicated to a single variable type: integer or real. Solving mixed problems relies on *ad hoc* transformations. In another field, abstract interpretation offers tools to prove program properties by studying an abstraction of their concrete semantics, that is, the set of possible values of the variables during an execution. Various representations for these abstractions have been proposed. They are called abstract domains. Abstract domains can mix any type of variables, and even represent relationship between the variables. In this book, we define abstract domains for constraint programming so as to build a generic solving method, dealing with both integer and real variables. We will also study the octagons abstract domain already defined in abstract interpretation. Guiding the search by the octagonal relations, we obtain good results on a continuous benchmark. Then, we define our solving method using abstract interpretation techniques in order to include existing abstract domains. Our solver, AbSolute, is able to solve mixed problems and use relational domains.

Marie PELLEAU
February 2015

Introduction

Recent advances in computer science are undeniable. Some are visible, and others are less known to the general public: today, we are able to quickly solve many problems that are known to be difficult (requiring a long computation time). For instance, it is possible to automatically place thousands of objects of various shapes in a minimum number of containers in tens of seconds, while respecting specific constraints: accessibility of goods, non-crush, etc. [BEL 07]. Constraint programming (CP) formalizes such problems using constraints that describe a result we want to achieve (accessibility of certain objects, for example). These constraints come with efficient algorithms to solve greatly combinatorial problems. In another research area, semantics, abstract interpretation (AI) attacks an insoluble problem in the general case: the correction of programs. With strong theoretical tools developed from its creation (fixed-point theorems), AI manages to prove the properties of programs. In this area, the effectiveness of methods makes it possible for impressive applications to be solved: tools in AI have, for instance, managed to prove that there was no overflow error in the flight controls of the Airbus A380 which contains almost 500,000 lines of code.

The work presented in this book is at the interface between CP and AI, two research areas in computer science with *a priori* quite different problematics. In CP, the goal is usually to obtain a good computation time for problems that are, in general, nondeterministic polynomial

time (NP), or to extend existing tools to handle more problems. In AI, the goal is to analyze very large programs by capturing a maximum of properties. Despite their differences, there is a common concern in these two disciplines: identifying an impossible or difficult (computationally) space to compute precisely (the solutions set in CP and the semantics of the program in AI). It concerns computing the relevant overapproximations of this space. CP proposes methods to carefully surround this space (consistency and propagation), always with Cartesian overapproximations (boxes in $\mathbb{R}^n$ or $\mathbb{Z}^n$). AI uses often less accurate overapproximations but not only Cartesian: they may have various different shapes (not only boxes but also octagons, ellipsoids, etc.). These non-Cartesian approximations facilitate more properties to be captured.

In this book, we exploit the similarities of these overapproximation methods to integrate AI tools in the methods of CP. We redefine tools in CP from notions of AI (abstract domains). This is not only an intellectual exercise. Indeed, by generalizing the description of overapproximations, there is a significant gain in the expressiveness of CP. In particular, the problems are treated uniformly for real and integer variables, which is not currently the case. We also develop the octagon abstract domain, showing that it is possible to exploit the relationships captured by this particular domain to solve continuous problems more effectively. Finally, we perform the opposite task: we define CP as an abstract operation in AI, and develop a solver capable of handling practically all abstract domains.

I.1. Context

As mentioned before, the CP and AI have a common concern: computing efficiently and as accurately as possible an approximation of a difficult or impossible space. However, the issues and problems of these two areas are different, and hence so are their fields of application.

I.1.1. *Constraint programming*

CP, whose origins date back to 1974 [MON 74], is based on the formalization of problems such as a combination of first-order logic formulas, i.e. the constraints. A constraint defines a relationship between the variables of a problem: for example, two objects placed in the same container have an empty geometric intersection, that is to say, a heavy object should be placed under a fragile object. This is known as declarative programming. CP provides efficient generic solution methods for many combinatorial problems. Academic and industrial applications are varied: job-shop scheduling problems [GRI 11, HER 11a], design of substitution tables in cryptography [RAM 11], scheduling problems [STØ 11], prediction of the ribonucleic acid (RNA) secondary structure in biology [PER 09], optical network design [PEL 09] or automatic harmonization in music [PAC 01].

One of the limitations of the expressiveness of CP methods is that they are dedicated to the nature of the problem: solvers used for discrete variable problems are fundamentally different from techniques dedicated to continuous variable problems. In a way, the semantics of the problem is different depending on whether one deals with discrete or continuous problems.

However, many industrial problems are mixed: they contain both integer and real variables. This is, for example, the case of the problem of fast power grid repair after a natural disaster [SIM 12] to restore the power as quickly as possible in the affected areas. In this problem, we try to establish a plan of action and determine the routes that should be used by repair crews. Some of the variables are discrete; for example, each device (generator, line) is associated with a Boolean variable, indicating whether it is operational or not. Others are real, as the electrical power on a line. Another example of application is the design of the topology of a multicast transmission network [CHI 08]: we want to design a network that is reliable. A network is said to be reliable when it is still effective even when one of its components is defective, so that all user communications can pass into the network with the least possible delay. Again, some of the variables are integers (the

number of lines in the network) while others are continuous (the flow of information passing over the network average).

The convergence of discrete and continuous constraints in CP is both an industrial need and a scientific challenge.

I.1.2. *Abstract interpretation*

The basis of AI was established in 1976 by Cousot and Cousot [COU 76]. AI is the theory of semantic approximation [COU 77b] in which one of the applications is programs proof. The goal is to verify and prove that a program does not contain a bug, that is to say, runtime errors. Industrial stakes are high. Indeed, many bugs have made history, such as the Year 2000 bug, or Y2K, due to system design error. On January 1, 2000, some systems showed the date of January 1, 1900. This bug may be repeated on January 19, 2038, on some UNIX systems [ROB 99]. Another example of a bug is that of the infamous inaugural flight of the Ariane 5 rocket, which, due to an error in the navigation system, caused the destruction of the rocket only 40 s after takeoff.

Every day, new softwares are being developed, corresponding to thousands or millions of lines of code. To test or verify these programs manually would require a considerable amount of time. The soundness of programs cannot be proven in a generic way; thus, AI implements methods to automatically analyze certain properties of a program. The analyzers are based on operations on the semantics of programs, that is, the set of values that can be taken by the variables of the program during its execution. By computing an overapproximation of these semantics, the analyzer can, for example, prove that the variables do not take values beyond the permitted ranges (*overflow*).

Many analyzers are developed and used for various application areas, such as aerospace [LAC 98, SOU 07], radiation [POL 06] and particle physics [COV 11].

I.2. Problematic

In this book, we focus on CP solving methods, known as *complete*, that find the solution set or prove that it is empty, if necessary. These methods are based on an exhaustive search of the space of all possible values, also called search space. Using operations to restrict the space to visit (consistency and propagation), these methods can be accelerated. Existing methods are dedicated to a certain type of variables, discrete or continuous. Facing a mixed problem, containing both discrete and continuous variables, CP offers no real solution and the techniques available are often limited. Typically, variables are artificially transformed so that they are all discrete as in the solver Choco [CHO 10], or all continuous as in the solver RealPaver [GRA 06]. In AI, analyzed programs often, if not always, contain different types of variables. Theories of AI integrate many types of domains, and helped develop analyzers uniformly dealing with discrete and continuous variables.

We propose to draw inspiration from the work of the AI community on the different types of domains to provide new solving methods in CP. These new methods should be able, in particular, to approximate with various shapes and solve mixed problems.

I.3. Outline of the book

This book is organized as follows: Chapter 1 gives the mandatory notions of AI and CP to understand our work and an analysis of the similarities and differences between these two research areas. Based on the similarities identified between CP and AI, we define abstract domains for CP in Chapter 2, with a resolution based on these abstract domains. The use of an example of abstract domain existing in AI in CP, the octagons, is detailed in Chapter 3. Chapter 4 deals with the solving method implementation details presented in Chapter 2 for octagons. Finally, Chapter 5 redefines the concepts of CP using the techniques and tools available in AI to define a method called abstract resolution. A prototype implementation, as well as experimental results, is finally presented.

I.4. Contributions

The work of this book aims to design new solving techniques for CP. There are two parts in this work. In the first part, the abstract domains are defined for CP, so as mandatory operators for the solving process. These new definitions allow us to define a uniform resolution framework that no longer depends on the variables type or on the representation of the variables values. An example of a solver using the octagon abstract domain and respecting the framework is implemented in a continuous solver Ibex [CHA 09a], and tested on examples of continuous problems. In the second part, the different CP operators needed to solve are defined in AI, allowing us to define a solving method with the existing operators in AI. This method was then implemented over Apron [JEA 09], a library of abstract domains.

Most theoretical and practical results of Chapters 2–5 are the subject of publications in conferences or journals [TRU 10, PEL 11, PEL 13, PEL 14].

1

State of the Art

In this chapter, we present the notions upon which abstract interpretation (AI) is based and the principles of constraint programming (CP). We do not provide an exhaustive presentation of both areas, but rather give the notions needed for the understanding of this book. The concepts discussed include those of partially ordered sets, lattice and fixpoint, which are at the basis of the underlying theories in both fields. It also includes the in-place tools, such as narrowing and widening operators in AI or consistency and splitting operators in CP. Finally, the chapter presents an analysis of the similitudes between an AI and CP upon which rely the works presented in this book.

1.1. Abstract interpretation

The founding principles of AI were introduced in 1976 by Patrick and Cousot [COU 76]. In this section, we only present some aspects of AI that will be needed afterward. For a more complete presentation, see [COU 92a, COU 77a].

1.1.1. *Introduction to abstract interpretation*

One of the applications of AI is to automatically prove that a certain type of bug does not exist in a program and that there is no error during a program execution. Let us see an example.

EXAMPLE 1.1 (Backtrace).– Consider the following program:

1: real x, y
2: $x \leftarrow 2$
3: $y \leftarrow 5$
4: $y \leftarrow y * (x - 2)$
5: $y \leftarrow x/y$

The backtrace for this program is:

line	x	y	
1	?	?	
2	2	?	
3	2	5	
4	2	0	
5	2	NaN	*Error: division by zero*

In toy examples like this one, the backtrace allows us to quickly detect that the program contains errors. However, real-life programs are more complex and larger in terms of lines of code; thus, it is impossible to try all the possible executions. Moreover, the Halting's theorem states that it is undecidable to prove that a program terminates.

Nowadays, computer science is omnipresent and critical programs may contain thousands or even millions of lines of code [HAV 09]. In these programs, execution errors are directly translated into significant cost. For example, in 1996, the destruction of the Ariane 5 rocket was due to an integer overflow [LIO 96]. In 1991, American Patriot missiles failed to destroy an enemy Scud missile killing 28 soldiers due to a rounding error that had been propagated through the computations [MIS 92]. We must, therefore, ensure that such programs do not have any execution errors. Moreover, this should be done in a reasonable time without having to run the program. Indeed, sending probes into space just to check whether the program is correct, in the sense that it does not contain execution errors, is not a viable solution from an economical and ecological point of view. This is where AI comes into play. One of its applications is to verify that a program is correct during the compilation process, and thus before it is executed. The main idea is to study the values that can be taken by the variables

throughout the program. We call *semantics* the set of these values and *specification* the set of all the desired behaviors such as "never divided by zero". If the semantics meets all the given specifications, then we can say that the program is correct.

An important application of AI is the design of static program analyzers that are correct and complete. An analyzer is said to be correct if it answers that a program is correct only when the program does not contain any execution error. There exist several static analyzers; we distinguish two types: the correct static analyzers, such as Astrée [BER 10] and Polyspace [POL 10], and the non-correct static analyzers, such as Coverity [COV 03]. All these analyzers have industrial applications. For instance, Astrée was able to automatically prove the absence of runtime errors in the primary flight control software of the Airbus A340 fly-by-wire system. More recently, it analyzed the electric flight control code for the Airbus A380 [ALB 05, SOU 07]. Polyspace was used to analyze the flight control code for the Ariane 502 rocket [LAC 98] and verify security softwares of nuclear installations [POL 06]. As for Coverity, it has been used to verify the code of the curiosity Mars Rover [COV 12] and ensure the accuracy of the Large Hadron Collider (LHC) software [COV 11], the particle accelerator that led to the discovery of the Higgs Boson particle.

Computing the real semantics, called *concrete semantics*, is very costly and undecidable in the general case. Indeed, Rice's theorem states that any non-trivial property formulated only on the inputs and outputs of a program is undecidable. Thus, the analyzers compute an overapproximation of the concrete semantics, the *abstract semantics*. The first step is to associate a function with each instruction of the program. This function modifies the set of possible values for the variables with respect to the instruction. Thus, the program becomes a composition of these functions and the set of observable behaviors corresponds to a fixpoint of this composition. The second step is to define an abstract domain to restrict the expressivity by keeping only a subset of properties on the program variables. An abstract domain is a computable data structure used to depict some of the program

properties. Moreover, abstract domains come with efficient algorithms to compute different operations on the concrete semantics and allow the fixpoint to be computed in a finite time. The analyzer always observes a superset of program behaviors; thus, all the found properties by the analyzer are verified for the program. However, it can omit some of them.

EXAMPLE 1.2 (Deducted properties).– Consider the following program:

1: real x, y
2: $x \leftarrow random(1, 10)$
3: **if** $x \bmod 2 = 0$ **then**
4: $\quad y \leftarrow x/2$
5: **else**
6: $\quad y \leftarrow x - 1$
7: **end if**

As the variable x takes its value between 1 and 10 (instruction 2), we can deduce that variable y takes its value between 0 and 8. However, for every execution, $y < x$ is an omitted property by the intervals abstract domain but can be found with the polyhedra abstract domain.

In the following sections we present the theoretical notions upon which the AI relies.

1.1.2. *General presentation*

The AI underlying theory uses notions of fixpoints and lattices. These notions are recalled in this section.

1.1.2.1. *Lattices*

Lattices are a well-known notion in computer science. Here, they are used to express operations on abstract domains and require some properties, such as being closed and complete.

DEFINITION 1.1 (Poset).– *A relation $\sqsubseteq$ on a non-empty set $\mathcal{D}$ is a* partial order *(Po) if and only if it is reflexive, antisymmetric and transitive. A set with a partial order is called* partially ordered set *(poset). If they exist, we denote by $\bot$ the least element and by $\top$ the greatest element of $\mathcal{D}$.*

EXAMPLE 1.3 (Partially ordered set).– Let $\mathbb{F}$ be the set of floating-point numbers according to the IEEE norm [GOL 91]. For $a, b \in \mathbb{F}$, we can define $[a, b] = \{x \in \mathbb{R}, a \leq x \leq b\}$ as the real interval delimited by the floating-point numbers a and b, and $\mathbb{I} = \{[a, b], a, b \in \mathbb{F}\}$ as the set of intervals. Given an interval $I \in \mathbb{I}$, we denote by $\underline{I}$ (respectively, $\overline{I}$) its lower (respectively, upper) bound and for all point x, $\underline{x}$ (respectively, $\overline{x}$) its lower (respectively, upper) floating approximation.

Let $\mathbb{I}^n$ be the set of Cartesian products of n intervals. The set $\mathbb{I}^n$ with the inclusion relation $\subseteq$ is a partially ordered set.

REMARK 1.1.– Note that the inclusion relation $\subseteq$ is a partial order. Thus, for any non-empty set E, $\mathcal{P}(E)$ with this relation is a partially ordered set.

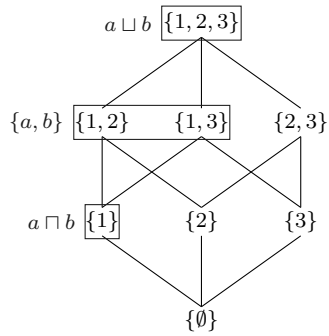
DEFINITION 1.2 (Lattice).– *A partially ordered set $(\mathcal{D}, \sqsubseteq, \sqcup, \sqcap)$ is a* lattice *if and only if for $a, b \in \mathcal{D}$, the pair $\{a, b\}$ has a least upper bound (lub) denoted by $a \sqcup b$, and a greatest lower bound (glb) denoted by $a \sqcap b$. A lattice is said to be* complete *if and only if any subset has both a least upper bound and a greatest lower bound.*

In a lattice, any *finite* subset has a least upper bound and a greatest lower bound. In a complete lattice, any subset has a least upper bound and a greatest lower bound, even if the subset is not finite. Thus, a complete lattice has a greater element denoted by $\top$, and a least element denoted by $\bot$.

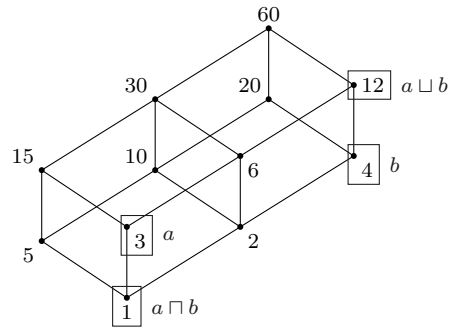
REMARK 1.2.– Notice that any finite lattice is automatically complete.

Figure 1.1 gives examples of partially ordered sets represented with Hasse diagrams. The first figure (Figure 1.1(a)) corresponds to the power sets of the set $\{1, 2, 3\}$ with the set inclusion $\subseteq$. This partially ordered set is finite and has a least element $\{\emptyset\}$ and a greatest element

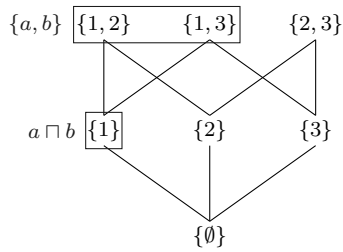
$\{1, 2, 3\}$. Hence, it is a complete lattice. For instance, the pair $\{\{1, 2\}, \{1, 3\}\}$ has a least upper bound $\{1, 2\} \cap \{1, 3\} = \{1\}$ and a greatest lower bound $\{1, 2\} \cup \{1, 3\} = \{1, 2, 3\}$.



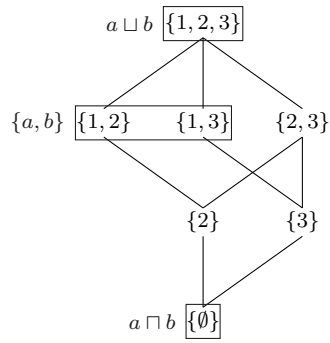
a) $(\mathcal{P}(\{1, 2, 3\}), \subseteq, \cup, \cap)$



b) Set of all the divisors of 60 ordered by the divisibility relation



c) $(\mathcal{P}(\{1, 2, 3\})\backslash\{1, 2, 3\}, \subseteq, \cup, \cap)$



d) $(\mathcal{P}(\{1, 2, 3\}) \setminus \{1\}, \subseteq, \cup, \cap)$

Figure 1.1. *Examples of partially ordered sets represented with Hasse diagram*

The second figure (Figure 1.1(b)) corresponds to the set of the divisors of 60: $\{1, 2, 3, 5, 6, 10, 12, 15, 20, 30, 60\}$, with the divisibility relation. Similarly, this partially ordered set has a least element 1 and a greatest element 60. Thus, it is a complete lattice. The pair $\{3, 4\}$ has a greatest lower bound 1 (their greatest common divisor) and a least upper bound 12 (their least common multiple).

On the contrary, the third example (Figure 1.1(c)) is not a lattice. Indeed, the pair $\{\{1,2\},\{1,3\}\}$ does not have a least upper bound. Likewise, if the element $\{\emptyset\}$ is removed from the lattice (Figure 1.1(a)), the partially ordered set obtained is no longer a lattice. However, removing any element that is neither the least nor the greatest element of the lattice does not change the fact that it is a lattice as shown in Figure 1.1(d).

EXAMPLE 1.4 (Lattice).– It is easily verified that the partially ordered set $(\mathbb{I}^n, \subseteq, \cup, \cap)$ with the least element $\bot = \emptyset$ and the greatest element $\top = \mathbb{F}^n$ is a complete lattice. Let $I = I_1 \times \cdots \times I_n$ and $I' = I'_1 \times \cdots \times I'_n$ be any two elements of $\mathbb{I}^n$. The pair $\{I, I'\}$ has a *glb*

$$I \cap I' = [\max(\underline{I_1}, \underline{I'_1}), \min(\overline{I_1}, \overline{I'_1})] \times \ \ldots \ \times [\max(\underline{I_n}, \underline{I'_n}), \min(\overline{I_n}, \overline{I'_n})]$$

and a *lub*

$$I \cup I' = [\min(\underline{I_1}, \underline{I'_1}), \max(\overline{I_1}, \overline{I'_1})] \times \ \ldots \ \times [\min(\underline{I_n}, \underline{I'_n}), \max(\overline{I_n}, \overline{I'_n})]$$

It follows that any subset has a least upper bound and a greatest lower bound, and therefore $(\mathbb{I}^n, \subseteq, \cup, \cap)$ is a lattice. Moreover, this lattice is finite; thus, $(\mathbb{I}^n, \subseteq, \cup, \cap)$ is a complete lattice.

Lattices are the base set upon which rely the abstract domains in AI. An important feature of the abstract domains is that they can be linked by a Galois connection. Note that some abstract domains do not have a Galois connection (see remark 1.5). Galois connections have been applied to the semantics by Cousot and Cousot in [COU 77a] as follows.

DEFINITION 1.3 (Galois connection).– *Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be the two partially ordered sets; a Galois connection is defined by two morphisms, an abstraction $\alpha: \mathcal{D}_1 \ \rightarrow \ \mathcal{D}_2$ and a concretization $\gamma : \mathcal{D}_2 \rightarrow \mathcal{D}_1$ such that:*

$$\forall X_1 \in \mathcal{D}_1, X_2 \in \mathcal{D}_2, \alpha(X_1) \sqsubseteq X_2 \iff X_1 \sqsubseteq \gamma(X_2)$$

Galois connections are usually represented as follows:

$$\mathcal{D}_1 \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{D}_2$$

REMARK 1.3.– An important consequence of this definition is that the functions α and γ are monotonic for the order $\sqsubseteq$ [COU 92a], that is:

$$\forall X_1, Y_1 \in \mathcal{D}_1, X_1 \sqsubseteq Y_1 \Rightarrow \alpha(X_1) \sqsubseteq \alpha(Y_1),$$

and

$$\forall X_2, Y_2 \in \mathcal{D}_2, X_2 \sqsubseteq Y_2 \Rightarrow \gamma(X_2) \sqsubseteq \gamma(Y_2)$$

REMARK 1.4.– This definition implies that $(\alpha \circ \gamma)(X_2) \sqsubseteq X_2$ and $X_1 \sqsubseteq (\gamma \circ \alpha)(X_1)$. X_2 is said to be a correct approximation (or abstraction) of X_1.

REMARK 1.5.– Note that for a given abstract domain there can be no abstraction function. For instance, the polyhedra abstract domain has no abstraction function. Indeed, there exist an infinity of approximations of a circle with a polyhedron. Therefore, there is no Galois connection for the polyhedra abstract domain.

Figure 1.2 shows three different approximations for a circle using polyhedra. As there exists an infinity of tangent to the circle, there potentially exists a polyhedron with an infinite number of sides exactly approximating the circle.

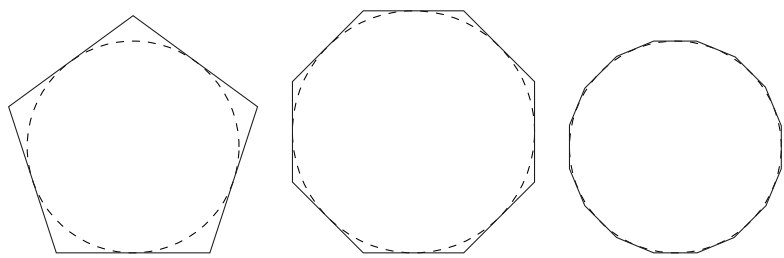


Figure 1.2. *Approximations for a circle*

Galois connections are used in CP even though they are not named. For instance, they are used when solving continuous problems. Indeed, as the intervals with real bounds are not computer representable, they

are approximated as intervals with floating-point bounds. The transition from one representation to another forms a Galois connection as shown in the following example.

EXAMPLE 1.5 (Galois connection).– Let $\mathbb{J}$ be the set of intervals with real bounds. Given two partially ordered sets $(\mathbb{I}^n, \subset)$ and $(\mathbb{J}^n, \subset)$, there exists a Galois connection:

$$\mathbb{J}^n \underset{\alpha_{\mathbb{I}}}{\overset{\gamma_{\mathbb{I}}}{\leftrightarrows}} \mathbb{I}^n$$
$$\alpha_{\mathbb{I}}([x_1, y_1] \times \cdots \times [x_n, y_n]) = [\underline{x_1}, \overline{y_1}] \times \cdots \times [\underline{x_n}, \overline{y_n}]$$
$$\gamma_{\mathbb{I}}([a_1, b_1] \times \cdots \times [a_n, b_n]) = [a_1, b_1] \times \cdots \times [a_n, b_n]$$

In this example, the abstraction function $\alpha_{\mathbb{I}}$ transforms a Cartesian product of real bound intervals into a Cartesian product of floating-point bounds intervals. It approximates each real bound by the closest floating-point number rounded in $\mathbb{F}$ in the corresponding direction. As for the concretization function, it is straightforward since a floating-point number is also a real.

1.1.2.2. *Concrete/abstract*

The concrete domain, denoted by $\mathcal{D}^\flat$, corresponds to the values that can be taken by the variables throughout the program $\mathcal{D}^\flat = \mathcal{P}(V)$ with V a set. Computing the concrete domain can be undecidable; thus, an approximation is accepted. The approximation of the concrete domain is called an abstract domain and is denoted by $\mathcal{D}^\sharp$. If there exists a Galois connection between the concrete domain and the abstract domain, $\mathcal{D}^\flat \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{D}^\sharp$, then any concrete function $f^\flat$ in $\mathcal{D}^\flat$ as an abstraction $f^\sharp$ in $\mathcal{D}^\sharp$ such that

$$\forall X^\sharp \in \mathcal{D}^\sharp, (\alpha \circ f^\flat \circ \gamma)(X^\sharp) \sqsubseteq f^\sharp(X^\sharp)$$

This is a consequence of remark 1.4. Moreover, the abstract function $f^\sharp$ is said to be optimal if and only if $\alpha \circ f^\flat \circ \gamma = f^\sharp$.

In the following, we will write $\mathcal{D}$ (respectively, f) for a domain (respectively, function) in general (whether it is concrete or abstract).

We will write $\mathcal{D}^\flat$ and $f^\flat$ for a concrete domain and a concrete function, and $\mathcal{D}^\sharp$ and $f^\sharp$ for an abstract domain and an abstract function.

1.1.2.3. *Transfer function*

In order to analyze a program, each line of code is analyzed. To do so, each instruction is associated with a function, called transfer function, which modifies the possible values for the variables.

DEFINITION 1.4 (Transfer function).– *Let C be the line of code to analyze. Given an initial set of states, a transfer function $F : \mathcal{P}(\mathcal{D}) \rightarrow \mathcal{P}(\mathcal{D})$ returns a set of environments corresponding to all the possible accessible states after the execution of C. We will write $\{\!|C|\!\}$ the transfer function for the instruction C and $\{\!|C|\!\}X$ when it is applied to the environments set X.*

EXAMPLE 1.6 (Transfer Function for an Affectation).– Consider the affectation $x \leftarrow expr$ with x a variable and $expr$ any expression. The transfer function $\{\!|x \leftarrow expr|\!\}$ only modifies in the initial environments set the possible values for the variable x.

Let x and y be two variables taken their values in $[-10, 10]$. The transfer function $\{\!|x \leftarrow random(1, 10)|\!\}$ only modifies the values for x. We now have x in $[1, 10]$ and y in $[-10, 10]$.

EXAMPLE 1.7 (Transfer function for a condition).– Let us now consider a Boolean expression; the transfer function only keeps the environments satisfying the Boolean expression. Let x and y be two variables taken their values in $[-10, 10]$. For the Boolean expression $x \leq 0$, the transfer function $\{\!|x \leq 0|\!\}$ filters the values of x so as to satisfy the Boolean expression. We now have x in $[-10, 0]$ and y in $[-10, 10]$.

REMARK 1.6.– In the following, any transfer function is supposed to be monotonic.

1.1.2.4. *Fixpoint*

Abstract interpretation also relies on the notion of fixpoint.

DEFINITION 1.5 (Fixpoint).– *Let F be a function; we called fixpoint of F an element X such that $F(X) = X$. We denote by $\mathrm{lfp}_X F$ a least fixpoint of F greater than X, and $\mathrm{gfp}_X F$ a greatest fixpoint of F smaller than X.*

REMARK 1.7.– Note that when F is monotonic, if the least or the greatest fixpoint exists, then it is unique.

Each instruction of the program is associated with a transfer function. Thus, the program corresponds to a composition of these functions. Proving that the program is correct is equivalent to computing the least fixpoint of this composition of functions. The computation of the fixpoint is mandatory to analyze loops, for instance. The functions associated with the loop are applied several times until the fixpoint is reached.

There exist several possible iterative schemas. Let $(X_1, \ldots, X_n)$ be the set of environments, where X_i corresponds to the environments set for the instruction i. We denote by X_i^j the set of environments for the instruction i at the iteration j. Let F_i be the transfer function for the instruction i. The most common iterative scheme is the Jacobi iteration scheme. The value of X_i^j is computed using the environments sets from the previous iteration:

$$X_i^j = F_i(X_1^{j-1}, \ldots, X_n^{j-1})$$

Another iterative scheme is the Gauss–Seidel iteration scheme. It computes the value of X_i^j using the environments sets already computed at the current iteration and the environments sets from the previous iteration:

$$X_i^j = F_i(X_1^j, \ldots, X_{i-1}^j, X_i^{j-1}, X_{i+1}^{j-1}, \ldots X_n^{j-1})$$

In the examples of this section, we use the Gauss–Seidel iterations.