THÈSE

présentée à

Université Scientifique et Médicale de Grenoble
Institut National Polytechnique de Grenoble

pour obtenir le grade de
DOCTEUR ÈS SCIENCES MATHÉMATIQUES

par

Patrick COUSOT

MÉTHODES ITÉRATIVES DE CONSTRUCTION ET D'APPROXIMATION DE POINTS FIXES D'OPÉRATEURS MONOTONES SUR UN TREILLIS, ANALYSE SÉMANTIQUE DES PROGRAMMES.

Thèse soutenue le 21 mars 1978 devant la Commission d'Examen :

Président : L. BOLLIET

Examinateurs : C. BENZAKEN
Ph. JORRAND
B. LORHO
C. PAIR
F. ROBERT
M. SINTZOFF

THÈSE

présentée à

Université Scientifique et Médicale de Grenoble
Institut National Polytechnique de Grenoble

pour obtenir le grade de
DOCTEUR ÈS SCIENCES MATHÉMATIQUES

par

Patrick COUSOT

MÉTHODES ITÉRATIVES DE CONSTRUCTION ET D'APPROXIMATION DE POINTS FIXES D'OPÉRATEURS MONOTONES SUR UN TREILLIS, ANALYSE SÉMANTIQUE DES PROGRAMMES.

Thèse soutenue le 21 mars 1978 devant la Commission d'Examen :

Président : L. BOLLIET

Examinateurs : C. BENZAKEN
Ph. JORRAND
B. LORHO
C. PAIR
F. ROBERT
M. SINTZOFF

ABSTRACT INTERPRETATION: ITERATIVE METHODS FOR FIXED POINT CONSTRUCTION AND APPROXIMATION OF MONOTONE OPERATORS ON LATTICES, SEMANTICS-BASED STATIC PROGRAM ANALYSIS

CHAPTER 1.

INTRODUCTION

1. INTRODUCTION

Semantic program analysis consists in the determination of the conditions in which the run-time execution of a program terminates, does not terminate, or leads to an error (either because the rules of good usage of a programming language have not been respected, or because the program does not correspond to its specification). The semantic analysis of a program must also allow us to determine, at each point of the program, the properties of the objects manipulated by the program.

We propose a theory of the semantic analysis of programs providing a unified framework to perform analyses, from the most precise ones, such as those performed to justify the total correctness of programs, to the coarsest, such as those used in compilation. In our opinion, there is no lack of continuity between these two extremes, and the theory we propose allows the construction of a continuous range of applications, from exact analysis to the most approximate analyses. Since we care about practical applications, we have devoted part of our efforts to build up a model leading to automatized solutions (some automatizations having effectively been realized) to economically relevant problems.

- In most systems for program verification, the semantic analysis of the program to be verified must be done by the programmer, who has to provide documentation of the program, often extensive due to the large amount of details. Now, if we exclude the specification of the output describing the problem to be solved, a fair amount of this documentation can be constructed from the text of the program (with the certainty that this documentation and the program itself are in accordance).

- Program debugging techniques, still widely used in the computer software industry, can be partly avoided (at least to remove programming mistakes, if not design mistakes), using our techniques of automatic semantic analysis of programs, and this, without waiting for the ten, or more, years necessary for theorem prover based techniques of program verification to be made practical. On the other hand, it is certain that the methods we propose are complementary and offer, for some kinds of analyses, a very profitable cost/benefit ratio.

- In high-level languages, the programmer is encouraged to formulate his or her algorithms in abstract terms, appropriate to the problem to be solved. To make an automatic choice for an effective program implementation, one has to make a rather precise semantic analysis of it.

- Almost all the definitions of classic programming languages contain various restrictions which are necessary for the programs to be meaningful, but cannot generally be checked syntactically. One should indeed know the domain of variable values. The classic solution of run-time tests is generally considered unacceptable because of its cost. Only an automatic semantic analysis of programs can provide an economically viable solution.

- Most optimization techniques used in the compilation of programs can only be implemented when the conditions ensuring the equivalence between the transformed program and the original one are satisfied, as well as conditions ensuring an actual performance improvement. When there is a doubt, the classic option consists in considering the most pessimistic hypothesis. A deeper semantic analysis of the program could avoid this.

Generally, the development of a theory of the semantic analysis of programs leading to automatized applications is justified by the technical resolution of the software reliability and the software efficiency problems. It is, in our opinion, complementary to the efforts which are currently made to elevate the Art of Programming to the status of Science.

Let us now give a very short summary of the contents of this thesis:

We will reduce the problem of the determination of semantic properties of a program to the problem of computing the extreme fixpoints of monotone operators on a complete lattice. After this introduction, the second chapter is then dedicated to the mathematical study of fixpoint theorems in complete lattices. We provide a constructive demonstration of Tarski's theorem, showing that the set of fixpoints of a monotone operator F on a complete lattice L is the image of L by the pre-closure operator defined by means of transfinite iteration limits. It is a matter of showing how classic iterative methods can be adapted to converge starting from any point, and also, to reach fixpoints other than the least and the greatest ones. This also allows us to define the union and the intersection in the lattice of the fixpoints of F in a constructive way, that is, by recurrences on F. We obtain, as a particular case, the theorem of construction of the least fixpoint of a continuous operator. We will then consider some systems of monotone fixpoint equations in a complete lattice. After recalling the formal resolution method by variable elimination, we will demonstrate a convergence result of chaotic iterative methods, asynchronous iterative methods, and asynchronous iterative methods with memory. This opens the way to the resolution of systems of monotone equations on a lattice using several processors computing in parallel, without the need for any synchronization.

In the third chapter, the problem of semantic analysis of programs is studied independently from the problem of language definition, within a very general framework studying the behavior of a discrete dynamic system. A program is a discrete dynamic system as long as it defines a transition relation (or a transition function, if it is deterministic) between the states of memory preceding or following the execution of any elementary instruction. To study the behavior of a discrete dynamic system, it is necessary to characterize the set of reachable states satisfying a given entry specification, or else, to characterize the set of ascendants satisfying a given exit specification.

In other words, it is necessary to determine the weakest precondition, with respect to the entry states, so that the system may evolve towards a state satisfying a given postcondition, or the strongest postcondition characterizing the states towards which the system evolves, starting from any entry state, and satisfying a given precondition. We will show that these conditions are obtained as solutions of fixpoint equations, or of equation systems when the set of states of the dynamic discrete system is partitioned. We will then formalize the operational semantics of a simple programming language, corresponding to sequential iterative programs, and we will show how a program defines a discrete dynamic system. We will then apply the results obtained by the analysis of discrete dynamic system behaviors to the semantic analysis of programs. This leads us to define forward and backward deductive semantics of programs, generalizing the classic forward program verification method of Floyd–Naur and the backward method of Hoare–Dijkstra to techniques that formalize the semantics of programming languages. In fact, the forward and backward deductive semantics define the conditions in which a program successfully completes, does not complete, or leads to an error, as a solution of semantic equation systems associated with the program. Both semantics can be used to characterize, at each point of the program, the set of descendants of the entry states and the set of ascendants of the exit states. As a consequence, they are equivalent, as both allow an exact semantic analysis of programs.

After showing that the exact semantic analysis of programs consists in solving equation systems, keeping in mind that the solutions to these equations are not automatically computable, but wishing, at the same time, to design automatic analysis techniques, we are forced to limit ourselves to approximate automatic analyses. So, in Chapter Four, we will study computation methods to approximate fixpoints of monotone operators on a lattice. To effectively compute under- and over-approximations of the solutions of an equation system, we essentially propose two complementary methods. They consist, on the one hand, in simplifying the equations to solve and, on the other hand, in accelerating the convergence of iterative methods of fixpoint construction. To accelerate the convergence of an iteration which does not stabilize naturally in a fixed number of steps, we propose to extrapolate, while computing the terms of the sequence of iterates, to obtain an approximation of its limit in a finite number of steps. Similarly to iterative methods with convergence acceleration, the simplification of equations is widely used in numerical analysis but, for our needs, we have to study them in a purely algebraic framework. To simplify the semantic equation systems associated with the programs for each particular problem of semantic analysis of programs, we propose to ignore *a priori* certain properties and only keep the program properties which are meaningful for this specific application. From an algebraic point of view, this approximation is formalized as a closure operator in the domain of the equations to solve, a closure operator that we will define, in an equivalent way, by a Moore family, a congruence relation, or a pair of adjoint functions. Different approximate analyses can be combined by combining the corresponding closure operators and, in particular, the lattice of closure operators formalizes the design of a hierarchy of approximations, depending on their precision.

In Chapter Five we develop automatic semantic program analysis methods, therefore necessarily approximate. In order to perform an approximate semantic analysis of a program, we propose to compute an approximation of the forward and backward systems of semantic equations associated with this program. Having chosen a particular class of program properties providing useful answers to a given problem, we will show how the results of Chapter Four allow us to design an algorithm which can automatically perform the analysis of any program for this class of properties. The design of this algorithm is based on the choice of a closure operator defining a space of approximate properties, as well as the rules of construction of systems of simplified equation systems associated with a program. To solve these equations, we will use an iterative method. Extrapolation operations will be designed when convergence must be accelerated. We will illustrate our approach by giving some examples of approximate semantic analyses of programs. After briefly examining many different classic examples related to program optimization, we will consider applications to the discovery of pointer properties, the determination of the types of variables in high-level languages without declarations, the analysis of the interval of values of numeric variables, and also the discovery of linear relations of equality or inequality between the variables of a program.

Chapter Six discusses recursive procedures, whose analysis is more complex than that of sequential iterative programs, given that it is necessary to consider functional equations of the form $f(x) = F(f)(x)$, and no longer equations of the form $x = f(x)$. We will use the same approach as for iterative sequential programs, by defining a deductive semantics, and then, by introducing some approximation methods which, in fact, generalize the study of Chapter Four to the case of functional equation systems.

CHAPTER 2.

FIXPOINT THEOREMS ON COMPLETE LATTICES

2. FIXPOINT THEOREMS ON COMPLETE LATTICES

2. FIXPOINT THEOREMS ON COMPLETE LATTICES

We reduce the problem of determining the semantic properties of a program to the problem of solving an equation system $X = F(X)$, whose solution (or fixpoint F) characterizes the properties of the program. Chapters 3, 5, and 6 show that it is wise to choose a complete lattice L as a model of the properties to discover and to express the equation system $X = F(X)$ with the help of a monotone operator F on L.

Admitting this, the aim of this chapter is to remind, to improve, and to establish a certain number of mathematical results which will be useful to solve a system of monotone equations in a complete lattice.

The first three paragraphs (2.1, 2.2, 2.3) entail some brief reminders of complete lattices and, amongst others, of the image of a complete lattice by a closure operator.

In Paragraph 2.4, we will show that the set of monotone operators on a complete lattice is the image of the set of all operators on this lattice by a functional which is a closure operator.

The fundamental result in order to solve a monotone fixpoint equation in a complete lattice is that of Tarski [1955] which ensures the existence of solutions, in particular a least and a greatest solution. We also know how to build these extreme solutions by transfinite induction (or countable induction, using a supplementary hypothesis of continuity). In Paragraph 2.5, we give a constructive proof of Tarski's theorem, whose originality lies in defining the set of fixpoints of a monotone operator F on a complete lattice L as an image of L by pre-closure operators defined by means of transfinite iteration limits. It is a matter of showing how classic iterative methods can be adapted to converge starting from any point, and also to show how to reach fixpoints other than the least and the greatest ones. This also allows us to define the join and the meet within the lattice of the fixpoints of F in a constructive way, that is by, induction using F.

In Paragraphs 2.6 and 2.7, we will show how the classic hypothesis of continuity consists in fact in only considering monotone operators whose fixpoints are limits of “countable” iterations.

Then, we will focus on solving a system of monotone fixpoint equations in a complete lattice: after stating the formal resolution method by variable elimination (2.8), we will prove convergence results for several iterative methods: chaotic iteration, asynchronous iteration, and asynchronous iteration with memory (2.9). This opens up a research field, in particular, towards the use of several processors computing in parallel (without any synchronisation) to solve such equation systems.

2.1 COMPLETE LATTICES

As reference textbooks on ordered sets and lattices, we suggest Birkhoff [1967], Bourbaki [1967], Grätzer [1971], and Szász [1971]. The notions stated in Paragraphs 2.1, 2.2, 2.3 are devoted to establishing the basic notation and terminology.

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a *complete lattice* with respect to the *partial order* $\sqsubseteq$. By definition, any subset S of L has a *least upper bound* (also called *join*, and denoted as $\sqcup S$) and a *greatest lower bound* (also called *meet*, and denoted as $\sqcap S$) in L, which in particular implies that L has a least element (*infimum* $\bot = \sqcap L$) and a greatest element (the *supremum* $\top = \sqcup L$). If $x, y \in L$, we denote $\sqcup\{x, y\}$ as $x \sqcup y$ and $\sqcap\{x, y\}$ as $x \sqcap y$, and also $\{\{x \sqsubset y\} \Leftrightarrow \{x \sqsubseteq y \ \underline{\text{and}}\ x \neq y\}\}$, $\{\{x \sqsupseteq y\} \Leftrightarrow \{y \sqsubseteq x\}\}$, and $\{\{x \sqsupset y\} \Leftrightarrow \{y \sqsubset x\}\}$.

Let μ be any ordinal and $\langle x^\delta : \delta \in \mu\rangle$ be a collection of elements in L. We say that $\langle x^\delta : \delta \in \mu\rangle$ is an *ascending chain* if $\{\forall \delta, \eta \in \mu, \{\{\delta \leq \eta\} \Rightarrow \{x^\delta \sqsubseteq x^\eta\}\}\}$ and that $\langle x^\delta :$

$\delta \in \mu\rangle$ is a *strictly ascending chain* if and only if $\{\forall \delta, \eta \in \mu, \{\{\delta < \eta\} \Rightarrow \{x^\delta \sqsubset x^\eta\}\}\}$. The dual notions are that of *descending chain* and *strictly descending chain.*

We say that a lattice satisfies the *ascending chain condition* (or analogously the *maximal condition*) if any strictly ascending chain is finite. The dual notion is that of *descending chain condition* (or analogously *minimal condition*).

2.2 COMPLETE LATTICE OF THE OPERATORS ON A COMPLETE LATTICE

We denote by $E \to E'$ the set of functions from the set E to the set E' defined everywhere on E (an analogous classic notation is ${E'}^E$).

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice, the elements in $L \to L$ are called *operators* on L. The set of operators on L is a complete lattice $(L \to L)(\sqsubseteq', \bot', \top', \sqcup', \sqcap')$ with respect to the partial order $\sqsubseteq'$ defined as $\{\forall f, g \in (L \to L), \{f \sqsubseteq' g\} \Leftrightarrow \{\forall x \in L, f(x) \sqsubseteq g(x)\}\}$. Exploiting the lambda notation from Church [1951], we have $\bot' = \boldsymbol{\lambda} x \bullet \bot$, $\top' = \boldsymbol{\lambda} x \bullet \top$, $\sqcup' = \boldsymbol{\lambda} S \bullet (\boldsymbol{\lambda} x \bullet \sqcup \{f(x) : f \in S\})$ and $\sqcap' = \boldsymbol{\lambda} S \bullet (\boldsymbol{\lambda} x \bullet \sqcap \{f(x) : f \in S\})$. In order to simplify the notation, we will omit the primes and leave the distinction between $(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ and $(\sqsubseteq', \bot', \top', \sqcup', \sqcap')$ to be purely contextual.

2.3 IMAGE OF A COMPLETE LATTICE BY A CLOSURE OPERATOR

In order to study a subset R of a complete lattice L, it is common and useful to represent it as the image $f(L) = R$ of L by an operator f on L. Indeed, the properties of f often supply relevant information on R. In this sense, we will often encounter (§2.4, 2.5.3, 2.6, 5, 6) the specific and important case where f is a closure operator and, in this case, the subset R of L is a complete lattice on which we are able to construct the least upper bound and greatest lower bound.

Recall that an operator ρ on an ordered set $L(\sqsubseteq)$ is an *upper closure operator* on L if and only if it is *monotone* (synonymous *isotone*, *increasing*, namely $\{\forall x, y \in L, \{x \sqsubseteq y\} \Rightarrow \{\rho(x) \sqsubseteq \rho(y)\}\}$), *extensive* ($\boldsymbol{\lambda} x \bullet x \sqsubseteq \rho$), and *idempotent* ($\rho = \rho \circ \rho$).

THEOREM 2.3.0.1 *Ward [1942, Thm. 4.1]*

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and ρ be an upper closure operator on L, then the image $\rho(L)$ of L by ρ is a complete lattice $\rho(L)(\sqsubseteq, \rho(\bot), \top, \boldsymbol{\lambda} S \bullet \rho(\sqcup S), \sqcap)$.

DEFINITION 2.3.0.2 *Szász [1971, p. 50]*

An operator f on the complete lattice $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ is a *complete join-morphism* if and only if $\{\forall S \subseteq L, f(\sqcup S) = \sqcup f(S)\}$. The dual notion is that of *complete meet-morphism*. An operator is a *complete morphism* if it is both a complete join- and meet-morphism.

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ and $M(\sqsubseteq, \bot', \top', \sqcup', \sqcap')$ be two complete lattices such that $M \subseteq L$. We say that M is a *sub-join-semi-lattice* of L if $\sqcup' = \sqcup$ (dually *sub-meet-semi-lattice*) and we say that M is a *sub-lattice* of L if $\sqcup' = \sqcup$ and $\sqcap' = \sqcap$.

PROPOSITION 2.3.0.3 *Ward [1942, p. 193]*

The image $\rho(L)$ of a complete lattice $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ by an upper closure operator ρ on L is a complete sub-lattice of L if and only if ρ is a complete join-morphism.

PROPOSITION 2.3.0.4 *Monteiro & Ribeiro [1942]*

(a) - Let ρ be an upper closure operator on an ordered set $L(\sqsubseteq)$. For any $x \in L$ the set $\{y \in \rho(L) : x \sqsubseteq y\}$ is non-empty and contains a least element which is $\rho(x)$.

(b) - Conversely, if R is a subset of L such that for any $x \in L$ the set $\{y \in R : x \sqsubseteq y\}$ contains a least element $\rho(x)$, then ρ is an upper closure operator and $R = \rho(L)$.

From the above results, we are able to prove, by duality, analogous properties for lower closure operators:

An operator ρ defined on an ordered set $L(\sqsubseteq)$ is a *lower closure operator* on L if it is monotone, idempotent, and *reductive* ($\rho \sqsubseteq \boldsymbol{\lambda} x \cdot x$).

COROLLARY 2.3.0.5

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and ρ a lower closure operator on L, then the image $\rho(L)$ of L by ρ is a complete lattice $\rho(L)(\sqsubseteq, \bot, \rho(\top), \sqcup, \boldsymbol{\lambda} S \cdot \rho(\sqcap S))$.

$\rho(L)$ is a complete sub-lattice of L if and only if ρ is a complete meet-morphism.

COROLLARY 2.3.0.6

A subset R of a complete lattice $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ is the image $\rho(L)$ of L by a lower closure operator ρ on L if and only if, for each $x \in L$, the set $\{y \in R : y \sqsubseteq x\}$ is non-empty and contains a greatest element equal to $\rho(x)$.

These few properties of closure operators are the only ones that we will need in Chapter 2. For the needs of Chapters 5 and 6, our study will be completed in Chapter 4.

2.4 THE COMPLETE LATTICE OF MONOTONE OPERATORS ON A COMPLETE LATTICE

An operator f on $L(\sqsubseteq)$ is *monotone* if $\{\forall x \in L, \{x \sqsubseteq y\} \Rightarrow \{f(x) \sqsubseteq f(y)\}\}$. When L is a complete lattice this definition is equivalent to (Bourbaki [1967, Ex. 10, p. 102]):

- An operator f on the complete lattice $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ is monotone if and only if $\{\forall S \subseteq L, \sqcup f(S) \sqsubseteq f(\sqcup S)\}$.
- An operator f on the complete lattice $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ is monotone if and only if $\{\forall S \subseteq L, f(\sqcap S) \sqsubseteq \sqcap f(S)\}$.

It is well known that monotone operators on a complete lattice L form a complete sub-lattice of $(L \rightarrow L)$ (for example Bourbaki [1967, Ex. 11.d, p. 103]). The proof that we give is interesting because it presents a proof method that will be used often later in this thesis. Suppose that we are to study the set R of elements of a complete lattice L satisfying a given property P. If we can find an operator ρ on L such that, for each x in L, the set of elements of L greater than or equal to x satisfying P is non-empty and admits a least element equal to $\rho(x)$, then we know that ρ is an upper closure operator (2.3.0.4) and $R = \rho(L)$ is a complete lattice (2.3.0.1). Moreover R is a complete sub-lattice of L when ρ is a complete join-morphism (2.3.0.3).

Therefore, the set of monotone operators on a complete lattice L is a sub-lattice of the lattice $(L \rightarrow L)$ of operators on L, and is also the image of $(L \rightarrow L)$ by an upper closure operator *mon* that we can define as follows:

<u>DEFINITION</u> 2.4.0.1

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice. We denote by *mon* the operator defined on $(L \rightarrow L)$ as follows:

$$mon \quad = \quad \boldsymbol{\lambda} f \bullet (\boldsymbol{\lambda} x \bullet \sqcup \{f(y) : (y \in L) \ \underline{\text{and}}\ (y \sqsubseteq x)\})$$

THEOREM 2.4.0.2

Let f be an operator on L, then $mon(f)$ is the least monotone operator on L greater than or equal to f.

Proof: Let a and b be elements in L such that $a \sqsubseteq b$. For any y in L, $\{y \sqsubseteq a\}$ implies $\{y \sqsubseteq b\}$. Thus, $\sqcup\{f(y) : y \sqsubseteq a\} \sqsubseteq \sqcup\{f(y) : y \sqsubseteq b\}$, which proves that $mon(f)(a) \sqsubseteq mon(f)(b)$, that is to say, that $mon(f)$ is a monotone operator on L.

For any a in L, we have $f(a) \sqsubseteq \sqcup\{f(y) : y \sqsubseteq a\}$ because $\sqsubseteq$ is reflexive. It follows that $mon(f)$ is greater than or equal to f.

Let g be a monotone operator on L such that $f \sqsubseteq g$. For any y in L, $f(y) \sqsubseteq g(y)$, which implies that, for any a in L, $mon(f)(a) = \sqcup\{f(y) : y \sqsubseteq a\} \sqsubseteq \sqcup\{g(y) : y \sqsubseteq a\} \sqsubseteq \sqcup\{g(y) : g(y) \sqsubseteq g(a)\} \sqsubseteq g(a)$, which shows that $mon(f)$ is the least monotone operator on L greater than or equal to f.
End of proof.

By applying Theorem 2.3.0.4.(b) we obtain:

COROLLARY 2.4.0.3

mon is an upper closure operator on $(L \rightarrow L)$ and $mon(L \rightarrow L)$ is the set of all monotone operators on L.

Theorems 2.3.0.1 and 2.3.0.3 allow us to restate a known result:

THEOREM 2.4.0.4 *Bourbaki [1967, p. 163]*

The set of all monotone operators on a complete lattice L is a complete sublattice $(\sqsubseteq, \boldsymbol{\lambda} x \bullet \bot, \boldsymbol{\lambda} x \bullet \top, \sqcup, \sqcap)$ of $(L \rightarrow L)$.

By duality we obtain the following:

COROLLARY 2.4.0.5

Let f be an operator on L. Then $\boldsymbol{\lambda} x \bullet \sqcap \{f(y) : (y \in L) \underline{\text{and}} (x \sqsubseteq y)\}$ is the greatest monotone operator on L which is smaller than or equal to f. Moreover $\boldsymbol{\lambda} f \bullet (\boldsymbol{\lambda} x \bullet \sqcap \{f(y) : (y \in L) \underline{\text{and}} (x \sqsubseteq y)\})$ is a lower closure operator on $(L \rightarrow L)$ and the set of monotone operators on L is a complete sub-lattice of $(L \rightarrow L)$, which is the image of $(L \rightarrow L)$ by this lower closure operator.

2.5 CONSTRUCTIVE VERSION OF TARSKI'S FIXPOINT THEOREM

The fundamental result by Tarski [1955] shows that the set *fp*(f) of all *fixpoints* of an operator f on a non-empty complete lattice $L(\sqsubseteq, \perp, \top, \sqcup, \sqcap)$ (i.e., *fp*$(f) = \{x \in L : f(x) = x\}$) is a non-empty complete lattice with respect to the order $\sqsubseteq$ (although it is not necessarily a sub-lattice of L). The proof given by Tarski is based on the definition of the least fixpoint *lfp*(f) of f by *lfp*$(f) = \sqcap\{x \in L : f(x) \sqsubseteq x\}$. The upper limit of $S \subseteq$ *fp*(f) in *fp*(f) is then *lfp*$(f \mid \{x \in L : (\sqcup S) \sqsubseteq x\})$ and the proof closes by applying the duality principle.

Computer scientists as well as numerical analysts (Amann [1976]) that use Tarski's theorem usually prefer to define *lfp*(f) as $\bigsqcup_{i<\omega} f^i(\perp)$, where ω is the first limit ordinal. This iterative method goes back to the first recursion theorem of Kleene [1952] and is used by Tarski [1955] for complete morphisms. More generally this requires an additional hypothesis of continuity (see for instance Kolodner [1968]) which is strictly stronger than monotonicity (it is required that $f(\bigsqcup_{i<\omega} f^i(\perp)) = \bigsqcup_{i<\omega} f^{i+1}(\perp))$. From a more abstract point of view, the continuity hypothesis is not necessary and we can use instead transfinite iterations similar to those used in Devidé [1964], Hitchcock & Park [1973], and Pasini [1974]. In order to give a constructive version of Tarski's theorem we will use the ideas from Cousot & Cousot [1977a] and Manna & Shamir [1977] to compute fixpoints of an operator f in two steps: a first, increasing iteration sequence from an arbitrary starting point d which converges towards a post-fixpoint p of f (i.e., $f(p) \sqsubseteq p$), and a second, decreasing iteration sequence from p towards a fixpoint q. Let $q = f^\star(d)$, we prove that the set $\mathit{fp}(f)$ of fixpoints of f is the image of the complete lattice L by the function $f^\star$ which is a pre-closure operator, implying that $\mathit{fp}(f) = f^\star(L)$ is a complete lattice.

2.5.1 Definition of a transfinite iteration

The class Ord of ordinals is well ordered by $\leq \; = \{(x, y) : (x, y \in \text{Ord}) \ \underline{\text{and}} \ ((x \in y) \ \underline{\text{or}} \ (x = y))\}$. We will denote by 0 the ordinal of the empty set and by $\bigcup\limits_{i \in I} \delta_i$ the ordinal which is the upper limit of the family of ordinals $\{\delta_i : i \in I\}$. Recall that, if δ is a *limit ordinal*, then $\bigcup\limits_{\alpha < \delta} \alpha = \delta$. Otherwise, δ is a *successor ordinal* and $\bigcup\limits_{\alpha < \delta} \alpha = (\delta - 1)$ where the predecessor of δ is denoted as $\delta - 1$.

DEFINITION 2.5.1.0.1 *Increasing iteration*

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $\mu(L)$ the least ordinal such that the cardinal of the class $\{\delta : \delta \in \mu(L)\}$ is strictly greater than the cardinal $\mathit{card}(L)$ of L. Assume also that $f \in \mathit{mon}(L \to L)$.

The *increasing iteration starting from $d \in L$ and defined by f* is a sequence $\langle x^\delta : \delta \in \mu(L) \rangle$ of elements of L defined by transfinite induction as follows:

$$\begin{array}{llll}
\text{(a) -} & x^0 & = & d \\
\text{(b) -} & x^\delta & = & f(x^{\delta-1}) \quad \text{for each successor ordinal } \delta \in \mu(L) \\
\text{(c) -} & x^\delta & = & \bigsqcup\limits_{\alpha < \delta} x^\alpha \quad \text{for each limit ordinal } \delta \in \mu(L)
\end{array}$$

In general, the elements of an increasing iteration may not be comparable, instead we will use the term “increasing” because we will always choose starting points ensuring that the increasing iteration is indeed an ascending chain.

DEFINITION 2.5.1.0.2 *Decreasing iteration*

The *decreasing iteration starting from* $d \in L(\sqsubseteq, \perp, \top, \sqcup, \sqcap)$ *and defined by* $f \in mon(L \rightarrow L)$ is a sequence $\langle x^\delta : \delta \in \mu(L)\rangle$ of elements in L defined by transfinite induction as follows:

$$\begin{array}{llll}
\text{(a) -} & x^0 & = & d \\
\text{(b) -} & x^\delta & = & f(x^{\delta-1}) \quad \text{for each successor ordinal } \delta \in \mu(L) \\
\text{(c) -} & x^\delta & = & \displaystyle\prod_{\alpha<\delta} x^\alpha \quad \text{for each limit ordinal } \delta \in \mu(L)
\end{array}$$

DEFINITION 2.5.1.0.3 *Limit of a stationary transfinite sequence*

We say that the sequence $\langle x^\delta : \delta \in \mu\rangle$ of elements in $L(\sqsubseteq, \perp, \top, \sqcup, \sqcap)$ is *stationary* if and only if $\{\exists \varepsilon \in \mu : \{\forall \beta \in \mu : \{\beta \geq \varepsilon\} \Rightarrow \{x^\varepsilon = x^\beta\}\}\}$, in this case the *limit* of this sequence is x^ε. We will note $luis(f)(d)$ (respectively $llis(f)(d)$) the limit of a stationary increasing iteration sequence (respectively decreasing) starting from d and defined by $f \in mon(L \rightarrow L)$.

LEMMA 2.5.1.0.4

Let $L(\sqsubseteq, \perp, \top, \sqcup, \sqcap)$ be a complete lattice and $\langle x^\delta : \delta \in \mu(L)\rangle$ be an increasing iteration sequence (respectively decreasing) starting from $d \in L$ and defined by $f \in mon(L \rightarrow L)$. If $\langle x^\delta : \delta \in \mu(L)\rangle$ is an ascending chain (respectively descending), then it is stationary and its limit is the least fixpoint of f greater than or equal to d (respectively the greatest fixpoint of f smaller than or equal to d).

Proof: We consider the case of an increasing iteration and assume that $\{\forall\varepsilon, \{\varepsilon \in \mu(L) \ \underline{\text{and}}\ (\varepsilon + 1) \in \mu(L)\} \Rightarrow \{x^\varepsilon \neq x^{\varepsilon+1}\}\}$. Then, by hypothesis, $\langle x^\delta : \delta \in \mu(L)\rangle$ is an ascending chain which is strictly ascending and the classes $\{x^\delta : \delta \in \mu(L)\}$ and $\{\delta : \delta \in \mu(L)\}$ are equinumerous and, by definition of $\mu(L)$, $card(\{x^\delta : \delta \in \mu(L)\}) > card(L)$. However, as f is defined everywhere on L, we know that $\{\forall\delta \in \mu(L), x^\delta \in L\}$ from which follows that $card(\{x^\delta : \delta \in \mu(L)\}) \leq card(L)$. By contradiction, we have proved that $\{\exists\varepsilon \in \mu(L) : (\varepsilon + 1) \in \mu(L)$ and $x^{\varepsilon} = x^{\varepsilon+I}\}$.

Assume that $\{\forall\beta, \{\varepsilon \leq \beta < \gamma < \mu(L)\} \Rightarrow \{x^{\varepsilon} = x^{\beta}\}\}$. Then, if γ is a successor ordinal, $x^{\varepsilon} = f(x^{\varepsilon}) = x^{\varepsilon+1} = x^{\gamma-1} = f(x^{\gamma-1}) = x^{\gamma}$ by Definition 2.5.1.0.1.(b) and by induction hypothesis. Otherwise, γ is a limit ordinal and $x^{\gamma} = \bigsqcup_{\alpha<\gamma} x^{\alpha} = (\bigsqcup_{\alpha<\varepsilon} x^{\alpha}) \sqcup (\bigsqcup_{\varepsilon\leq\beta<\gamma} x^{\beta}) \sqsubseteq x^{\varepsilon}$. As $\langle x^{\delta} : \delta \in \mu(L)\rangle$ is a descending chain and $\varepsilon < \gamma$, we have $d = x^0 \sqsubseteq x^{\varepsilon} \sqsubseteq x^{\gamma}$ and, by antisymmetry, $d \sqsubseteq x^{\varepsilon} = x^{\gamma}$. By transfinite induction, we conclude that $\langle x^{\delta} : \delta \in \mu(L)\rangle$ is stationary and $d \sqsubseteq x^{\varepsilon} = x^{\varepsilon+1} = f(x^{\varepsilon})$.

Assume that there exists $p \in fp(f)$ such that $d \sqsubseteq p$. We prove that $\forall\delta \in \mu(L), x^{\delta} \sqsubseteq p$. For $\delta = 0$, we have $x^0 = d \sqsubseteq p$. Assume that for each β such that $\beta < \delta < \mu(L)$ we have $x^{\beta} \sqsubseteq p$. If δ is a successor ordinal, then, by monotonicity, $x^{\delta} = f(x^{\delta-1}) \sqsubseteq f(p) = p$. If δ is a limit ordinal, then $x^{\delta} = \bigsqcup_{\beta<\delta} x^{\beta} \sqsubseteq p$ because p is an upper bound of all β such that $\beta < \delta$. By transfinite induction, $\forall\delta \in \mu(L), x^{\delta} \sqsubseteq p$ and this is true in particular for the iteration limit.

End of proof.

2.5.2 Increasing iteration starting from a pre-fixpoint

DEFINITION 2.5.2.0.1 *Pre-fixpoint and post-fixpoint of an operator*

Let $L(\sqsubseteq)$ be an ordered set and $f \in (L \rightarrow L)$, then the set of *pre-fixpoints* of f is $prefp(f) = \{x \in L : x \sqsubseteq f(x)\}$. Dually, the set of *post-fixpoints* of f is $postfp(f) = \{x \in L : f(x) \sqsubseteq x\}$.

THEOREM 2.5.2.0.2

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice. An increasing iteration $\langle x^{\delta} : \delta \in \mu(L)\rangle$ starting from $d \in prefp(f)$ and defined by $f \in mon(L \rightarrow L)$ is a stationary ascending chain and its limit $luis(f)(d)$ is the least fixpoint of f greater than or equal to d.

Proof: Because $x^0 = d \sqsubseteq f(d) = x^1$ and f is monotone, it is easy to show by transfinite induction that $\langle x^{\delta} : \delta \in \mu(L)\rangle$ is an ascending chain which is therefore

stationary. As a consequence of Lemma 2.5.1.0.4, its limit $luis(f)(d)$ is the least fixpoint of f greater than or equal to d.
End of proof.

By applying the duality principle we have:

COROLLARY 2.5.2.0.3

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice. A decreasing iteration sequence $\langle x^\delta : \delta \in \mu(L) \rangle$ starting from $d \in postfp(f)$ and defined by $f \in mon(L \to L)$ is a stationary descending chain and its limit $llis(f)(d)$ is the greatest fixpoint of f smaller than or equal to d.

Remark 2.5.2.0.4 Extreme fixpoints of a monotone operator

As $\bot \in prefp(f)$ and $\top \in postfp(f)$ for all f, $f \in mon(L \to L)$ admits at least two (non necessarily distinct) fixpoints. Moreover, as any fixpoint of f is included between $\bot$ and $\top$, f admits a least fixpoint $lfp(f) = luis(f)(\bot)$ and a greatest fixpoint $gfp(f) = llis(f)(\top)$.
End of remark.

PROPOSITION 2.5.2.0.5

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $f \in mon(L \to L)$, then the restriction $\big(luis(f) \mid prefp(f)\big)$ of $luis(f)$ to $prefp(f)$ is an upper closure operator on $prefp(f)$ and $fp(f) = luis(f)(prefp(f))$.

Proof: $fp(f)$ is a subset of $prefp(f)$ which, by Theorem 2.5.2.0.2, is such that for each $x \in prefp(f)$ the set $\{y \in fp(f) : x \sqsubseteq y\}$ admits a least element $luis(f)(x)$ and, as a consequence of Theorem 2.3.0.4.(b), this implies that $\big(luis(f) \mid prefp(f)\big)$ is an upper closure operator on $prefp(f)$ and $fp(f) = luis(f)(prefp(f))$.
End of proof.

By applying the duality principle we have:

COROLLARY 2.5.2.0.6

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $f \in \mathit{mon}(L \to L)$, then the restriction of $\mathit{llis}(f)$ to $\mathit{postfp}(f)$ is a lower closure operator on $\mathit{postfp}(f)$ and $\mathit{fp}(f) = \mathit{llis}(f)(\mathit{postfp}(f))$.

PROPOSITION 2.5.2.0.7

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $f, g \in \mathit{mon}(L \to L)$ such that $f \sqsubseteq g$. Then $\{\forall d \in \mathit{prefp}(f), \mathit{luis}(f)(d) \sqsubseteq \mathit{luis}(g)(d)\}$.

Proof: Because $f \sqsubseteq g$, we have $\mathit{prefp}(f) \subseteq \mathit{prefp}(g)$ and the proof follows by transfinite induction on the increasing iterations defined by f and g.
End of proof.

PROPOSITION 2.5.2.0.8

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ and $L'(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be complete lattices, $f \in \mathit{mon}(L \to L)$, $g \in \mathit{mon}(L' \to L')$, and h be a complete join-morphism from L to L' such that $h \circ f = g \circ h$. Then $\{\forall d \in \mathit{prefp}(f), h(\mathit{luis}(f)(d)) = \mathit{luis}(g)(h(d))\}$.

Proof: Let $\mu = \underline{\max}(\mu(L), \mu(L'))$ and $\langle X^\delta : \delta \in \mu \rangle$ be the increasing iteration defined by f starting from d. Because h is a complete join-morphism from L to L', it is monotone and therefore $\forall d \in \mathit{prefp}(f)$ we have $d \sqsubseteq f(d)$, which implies $h(d) \sqsubseteq h(f(d)) = g(h(d))$, that is, $h(d) \in \mathit{prefp}(g)$. Let $\langle Y^\delta : \delta \in \mu \rangle$ be the increasing iteration defined by g and starting from $h(d)$. We prove that $\forall \delta, h(X^\delta) = Y^\delta$. We have that $h(X^0) = h(d) = Y^0$. Assume by induction hypothesis that $\forall \alpha < \delta, h(X^\alpha) = Y^\alpha$. If δ is a successor ordinal, then, in particular, $h(X^{\delta-1}) = Y^{\delta-1}$ and $h(X^\delta) = h(f(X^{\delta-1})) = g(h(X^{\delta-1})) = g(Y^{\delta-1}) = Y^\delta$. If δ is a limit ordinal, then $h(X^\delta) = h(\bigsqcup_{\alpha<\delta} X^\alpha) = \bigsqcup_{\alpha<\delta} h(X^\alpha) = \bigsqcup_{\alpha<\delta} Y^\alpha = Y^\delta$ because h is a complete join-morphism and by induction hypothesis. By transfinite induction, we conclude that $\forall \delta \in \mu, h(X^\delta) = Y^\delta$. Let X^ε and $Y^{\varepsilon'}$ be respectively the limits of $\langle X^\delta : \delta \in \mu \rangle$ and

$\langle Y^\delta : \delta \in \mu\rangle$. It follows that $h(luis(f)(d)) = h(X^\varepsilon) = h(X^{\underline{\max}(\varepsilon,\varepsilon')}) = Y^{\underline{\max}(\varepsilon,\varepsilon')} = Y^{\varepsilon'} = luis(g)(h(d))$.

End of proof.

2.5.3 Constructive characterization of the sets of pre- and post-fixpoints of a monotone operator on a complete lattice

PROPOSITION 2.5.3.0.1

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $f \in mon(L \to L)$. For each $d \in L$, the increasing iteration sequences starting from d and defined by $\boldsymbol{\lambda} x \bullet x \sqcup f(x)$ and $\boldsymbol{\lambda} x \bullet d \sqcup f(x)$ are stationary ascending chains. Their limits $luis(\boldsymbol{\lambda} x \bullet x \sqcup f(x))(d)$ and $luis(\boldsymbol{\lambda} x \bullet d \sqcup f(x))(d)$ are equal to the least post-fixpoint of f greater than or equal to d.

Proof: Each element d in L is a pre-fixpoint of both $\boldsymbol{\lambda} x \bullet x \sqcup f(x)$ and $\boldsymbol{\lambda} x \bullet d \sqcup f(x)$ which are, by hypothesis $f \in mon(L \to L)$, monotone operators on L. Theorem 2.5.2.0.2 implies that the increasing iterations $\langle x^\delta : \delta \in \mu(L)\rangle$ and $\langle y^\delta : \delta \in \mu(L)\rangle$ starting from d and defined respectively by $\boldsymbol{\lambda} x \bullet x \sqcup f(x)$ and $\boldsymbol{\lambda} x \bullet d \sqcup f(x)$ are stationary ascending chains with respective limits x^{ε_1} and y^{ε_2}.

As x^{ε_1} is the least among the fixpoints of $\boldsymbol{\lambda} x \bullet x \sqcup f(x)$ greater than d and because $\{\forall p \in L, \{p = p \sqcup f(p)\} \Leftrightarrow \{f(p) \sqsubseteq p\}\}$, we conclude that $x^{\varepsilon_1} = luis(\boldsymbol{\lambda} x \bullet x \sqcup f(x))(d)$ is the least among the fixpoints of f greater than or equal to d.

As $y^{\varepsilon_2} = d \sqcup f(y^{\varepsilon_2})$, we have $f(y^{\varepsilon_2}) \sqsubseteq y^{\varepsilon_2}$ and, therefore, $d \sqsubseteq y^{\varepsilon_2}$ implies $x^{\varepsilon_1} \sqsubseteq y^{\varepsilon_2}$. It is easy to prove by transfinite induction on δ that $\{\forall \delta \in \mu(L), y^\delta \sqsubseteq x^\delta\}$. Therefore, $y^{\varepsilon_2} = y^{\underline{\max}(\varepsilon_1,\varepsilon_2)} \sqsubseteq x^{\underline{\max}(\varepsilon_1,\varepsilon_2)} = x^{\varepsilon_1}$ and, by antisymmetry, $x^{\varepsilon_1} = y^{\varepsilon_2}$, and we have $luis(\boldsymbol{\lambda} x \bullet x \sqcup f(x))(d) = luis(\boldsymbol{\lambda} x \bullet d \sqcup f(x))(d)$.

End of proof.

THEOREM 2.5.3.0.2 *The lattice of post-fixpoints of a monotone operator*

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $f \in \mathit{mon}(L \to L)$. The set of post-fixpoints of f is a non-empty complete lattice:

$$\mathit{postfp}(f)(\sqsubseteq, \mathit{lfp}(f), \top, \boldsymbol{\lambda}\, S \bullet\ \mathit{luis}(\boldsymbol{\lambda}\, z \bullet z \sqcup f(z))(\sqcup S), \sqcap)$$

where the least fixpoint of f is $\mathit{lfp}(f) = \sqcap\{x \in L : f(x) \sqsubseteq x\} = \mathit{luis}(f)(d)$ for any $d \in L$ such that $d \sqsubseteq \mathit{lfp}(f)$.

Proof: By Propositions 2.5.3.0.1, 2.3.0.4, and 2.3.0.1, $\mathit{postfp}(f)$ is a complete lattice $(\sqsubseteq,\ \mathit{luis}(\boldsymbol{\lambda}\, z \bullet z \sqcup f(z))(\bot),\ \top,\ \boldsymbol{\lambda}\, S \bullet\ \mathit{luis}(\boldsymbol{\lambda}\, z \bullet z \sqcup f(z))(\sqcup S),\ \sqcap)$.

Theorem 2.5.3.0.1 implies that $\mathit{luis}(\boldsymbol{\lambda}\, z \bullet z \sqcup f(z))(\bot) = \mathit{luis}(\boldsymbol{\lambda}\, z \bullet \bot \sqcup f(z))(\bot)$ $= \mathit{luis}(f)(\bot) = \mathit{lfp}(f)$ (Remark 2.5.2.0.4). As $\mathit{lfp}(f)$ is the infimum of $\mathit{postfp}(f)$, we have $\mathit{lfp}(f) = \sqcap\, \mathit{postfp}(f)$.

Finally, let $d \in L$ be such that $d \sqsubseteq \mathit{lfp}(f)$ and $\langle x^\delta : \delta \in \mu(L)\rangle$, $\langle y^\delta : \delta \in \mu(L)\rangle$, and $\langle z^\delta : \delta \in \mu(L)\rangle$ be the increasing iterations defined by f starting respectively from $\bot$, d, and $\mathit{lfp}(f)$. By transfinite induction, it is immediate to prove that $\{\forall \delta \in \mu(L), x^\delta \sqsubseteq y^\delta \sqsubseteq z^\delta = \mathit{lfp}\, f\}$. Because $\langle x^\delta : \delta \in \mu(L)\rangle$ is stationary with limit $\mathit{lfp}(f)$, it follows immediately that $\langle y^\delta : \delta \in \mu(L)\rangle$ is stationary with limit $\mathit{lfp}(f)$. (Observe that $\langle y^\delta : \delta \in \mu(L)\rangle$ is not necessarily an ascending chain.)

End of proof.

By applying the duality principle, we have:

COROLLARY 2.5.3.0.3 *The lattice of pre-fixpoints of a monotone operator*

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $f \in \mathit{mon}(L \to L)$. The set of pre-fixpoints of f is a non-empty complete lattice:

$$\mathit{prefp}(f)(\sqsubseteq, \bot, \mathit{gfp}(f), \sqcup, \boldsymbol{\lambda}\, S \bullet\ \mathit{llis}(\boldsymbol{\lambda}\, z \bullet z \sqcap f(z))(\sqcap S))$$

where the greatest fixpoint of f is $\mathit{gfp}(f) = \sqcup\{x \in L : x \sqsubseteq f(x)\} = \mathit{llis}(f)(d)$ for any $d \in L$ such that $\mathit{gfp}(f) \sqsubseteq d$.

2.5.4 Unary monotone polynomials on a complete lattice defined by a family of monotone operators

This paragraph stands aside from our preliminaries concerning the constructive version of Tarski's fixpoint theorem. We address the matter of giving an answer to the following question stated in Chapter 5: Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $\langle f_i : i \in I \rangle$ a family of monotone operators on L. What is the least and the greatest element in L that we can compute starting from a set $S \subseteq L$ of initial values by applying $\sqcup$, $\sqcap$, and f_i? The notion of computation being defined as that of unary polynomials on the algebra $\langle L; \sqcup, \sqcap, \langle f_i : i \in I \rangle \rangle$, we are left to determine the least and greatest polynomials on this algebra.

DEFINITION 2.5.4.0.1 *Unary polynomials*

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $\langle f_i : i \in I \rangle$ a family of elements in $mon(L \rightarrow L)$. The set $\mathbb{P}$ of *unary polynomials* on the algebra $\langle L; \sqcup, \sqcap, \langle f_i : i \in I \rangle \rangle$ is the least subset of $(L \rightarrow L)$ such that:

(a) - $(\boldsymbol{\lambda}\, x \bullet x) \in \mathbb{P}$

(b) - If $P \in \mathbb{P}$ and $i \in I$ then $(f_i \circ P) \in \mathbb{P}$

(c) - If $\langle P_\gamma : \gamma \in J \rangle \subseteq \mathbb{P}$ then $(\bigsqcup_{\gamma \in J} P_\gamma) \in \mathbb{P}$ and $(\bigsqcap_{\gamma \in J} P_\gamma) \in \mathbb{P}$

It follows immediately from this definition that unary polynomials on $\langle L; \sqcup, \sqcap, \langle f_i : i \in I \rangle \rangle$ are monotone operators on L.

THEOREM 2.5.4.0.2

Any unary polynomial on $\langle L; \sqcup, \sqcap, \langle f_i : i \in I \rangle \rangle$ is smaller than or equal to $luis(\boldsymbol{\lambda}\, z \bullet z \sqcup (\bigsqcup_{i \in I} f_i(z)))$ and greater than or equal to $llis(\boldsymbol{\lambda}\, z \bullet z \sqcap (\bigsqcap_{i \in I} f_i(z)))$.

Proof: Let $\overline{F} = luis(\boldsymbol{\lambda}\, z \bullet z \sqcup (\bigsqcup_{i \in I} f_i(z))$ and $\underline{F} = llis(\boldsymbol{\lambda}\, z \bullet z \sqcap (\bigsqcap_{i \in I} f_i(z))$. We know that $\overline{F}, \underline{F} \in mon(L \rightarrow L)$. By Proposition 2.5.3.0.1 and its dual statement, we deduce that $\mathit{luis}(\overline{F})(x)$ and $\mathit{llis}(\underline{F})(x)$ are defined for all x in L. The proof follows by structural induction:

(a) - $\mathit{luis}(\overline{F})$ is extensive (Theorem 2.5.2.0.5) and $\mathit{llis}(\underline{F})$ is reductive (Theorem 2.5.2.0.6). It follows that, for all x in L, we have $\mathit{llis}(\underline{F})(x) \sqsubseteq x \sqsubseteq \mathit{luis}(\overline{F})(x)$.

(b) - Let P be a unary polynomial such that $\mathit{llis}(\underline{F}) \sqsubseteq P \sqsubseteq \mathit{luis}(\overline{F})$. Then, for any $i \in I$, by monotonicity we have $f_i \circ \mathit{llis}(\underline{F}) \sqsubseteq f_i \circ P \sqsubseteq f_i \circ \mathit{luis}(\overline{F})$. Moreover, $\mathit{llis}(\underline{F}) = \underline{F} \circ \mathit{llis}(\underline{F}) \sqsubseteq f_i \circ \mathit{llis}(\underline{F}) \sqsubseteq f_i \circ P \sqsubseteq f_i \circ \mathit{luis}(\overline{F}) \sqsubseteq \overline{F} \circ \mathit{luis}(\overline{F}) = \mathit{luis}(\overline{F})$.

(c) - Let $\langle P_\gamma : \gamma \in J\rangle$ be a family of unary polynomials such that $\mathit{llis}(\underline{F}) \sqsubseteq P_\gamma \sqsubseteq \mathit{luis}(\overline{F})$ for any $\gamma \in J$. By definition of the least upper bound, we have $\mathit{llis}(\underline{F}) \sqsubseteq \bigsqcup_{\gamma \in J} P_\gamma \sqsubseteq \mathit{luis}(\overline{F})$ and, analogously, $\mathit{llis}(\underline{F}) \sqsubseteq \bigsqcap_{\gamma \in J} P_\gamma \sqsubseteq \mathit{luis}(\overline{F})$.

End of proof.

2.5.5 Constructive characterisation of the set of all fixpoints of a monotone operator on a complete lattice

THEOREM 2.5.5.0.1 *Constructive version of Tarski[1955]'s theorem*

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $f \in \mathit{mon}(L \to L)$. The set $\mathit{fp}(f)$ of fixpoints of f is a complete lattice:

$$\mathit{fp}(f)(\sqsubseteq, \mathit{lfp}(f), \mathit{gfp}(f), \boldsymbol{\lambda} S \bullet \mathit{luis}(f)(\sqcup S), \boldsymbol{\lambda} S \bullet \mathit{llis}(f)(\sqcap S))$$

Proof: As a consequence of Theorems 2.5.2.0.2 and 2.5.2.0.5, $\mathit{fp}(f)$ is the image of $\mathit{prefp}(f)$ by the upper closure operator $\mathit{luis}(f)$ and, as $\mathit{prefp}(f)$ is a complete lattice $(\sqsubseteq, \bot, \mathit{gfp}(f), \sqcup, \boldsymbol{\lambda} S \bullet \mathit{llis}(\boldsymbol{\lambda} z \bullet z \sqcap f(z))(\sqcap S))$ (Theorem 2.5.3.0.3), Theorem 2.3.0.1 implies that $\mathit{fp}(f)$ is a complete lattice for the partial order $\sqsubseteq$ with infimum $\mathit{luis}(f)(\bot) = \mathit{lfp}(f)$ and where the least upper bound operation is $\boldsymbol{\lambda} S \bullet \mathit{luis}(f)(\sqcup S)$. By duality,

$\mathit{fp}(f)$ is the image of the complete lattice $\mathit{postfp}(f)(\sqsubseteq, \mathit{lfp}(f), \top, \boldsymbol{\lambda} S \bullet\ \mathit{luis}(\boldsymbol{\lambda} z \bullet z \sqcup f(z))(\sqcup S), \sqcap)$ (Theorem 2.5.3.0.2) by the lower closure operator $\mathit{llis}(f)$ (Theorem 2.5.2.0.6) and, therefore (Theorem 2.3.0.3), it is a complete lattice for the partial order $\sqsubseteq$ with supremum $\mathit{llis}(f)(\bot) = \mathit{gfp}(f)$ and where the operation of greatest lower bound is $\boldsymbol{\lambda} S \bullet \mathit{llis}(f)(\sqcap S)$.
End of proof.

COROLLARY 2.5.5.0.2

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice, $d \in L$, and $f \in \mathit{mon}(L \rightarrow L)$. Then, $\mathit{luis}(f) \circ \mathit{llis}(\boldsymbol{\lambda} z \bullet z \sqcap f(z))(d)$ and $\mathit{llis}(f) \circ \mathit{luis}(\boldsymbol{\lambda} z \bullet z \sqcup f(z))(d)$ are fixpoints of f that are (respectively) greater than or equal to any fixpoint of f which is smaller than or equal to d, and smaller than or equal to any fixpoint of f which is greater than or equal to d. Moreover, $\mathit{luis}(f) \circ \mathit{llis}(\boldsymbol{\lambda} z \bullet z \sqcap f(z))(d) \sqsubseteq \mathit{llis}(f) \circ \mathit{luis}(\boldsymbol{\lambda} z \bullet z \sqcup f(z))(d)$.

Proof: Let $a, b \in L$ be such that $f(a) = a \sqsubseteq d \sqsubseteq b = f(b)$. Then, by monotonicity, $a = \mathit{luis}(f) \circ \mathit{llis}(\boldsymbol{\lambda} z \bullet z \sqcap f(z))(a) \sqsubseteq \mathit{luis}(f) \circ \mathit{llis}(\boldsymbol{\lambda} z \bullet z \sqcap f(z))(d) \sqsubseteq \mathit{luis}(f) \circ \mathit{llis}(\boldsymbol{\lambda} z \bullet z \sqcap f(z))(b) = b$. Likewise, $a \sqsubseteq \mathit{llis}(f) \circ \mathit{luis}(\boldsymbol{\lambda} z \bullet z \sqcup f(z))(d) \sqsubseteq b$.

Let $p = \mathit{llis}(\boldsymbol{\lambda} z \bullet z \sqcap f(z))(d)$ and $q = \mathit{luis}(\boldsymbol{\lambda} z \bullet z \sqcup f(z))(d)$. Let $S = \{x \in L : p \sqsubseteq x \sqsubseteq q\}$, it is a complete sub-lattice of L with infimum p and supremum q. By Theorem 2.5.3.0.1 and its dual, $p \sqsubseteq f(p)$ and $f(q) \sqsubseteq q$, therefore, $f(S) \subseteq S$. Theorem 2.5.5.0.1 implies that the least fixpoint of the restriction of f to S is $\mathit{luis}(f)(p)$ while the greatest fixpoint of the restriction of f to S is $\mathit{llis}(f)(q)$, which proves that $\mathit{luis}(f) \circ \mathit{llis}(\boldsymbol{\lambda} z \bullet z \sqcap f(z))(d) \sqsubseteq \mathit{llis}(f) \circ \mathit{luis}(\boldsymbol{\lambda} z \bullet z \sqcup f(z))(d)$.
End of proof.

DEFINITION 2.5.5.0.3 *Pre-closure operators of a complete lattice* (Ladegaillerie [1973])

A *lower pre-closure operator* $\underline{\rho}$ of a complete lattice $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ is an operator on L that is monotone, idempotent, and that satisfies the *lower connectivity axiom* $\{\forall x \in L : \underline{\rho}(x \sqcap \underline{\rho}(x)) = \underline{\rho}(x)\}$.

Dually, an *upper pre-closure operator* $\overline{\rho}$ on a complete lattice L is an operator on L that is monotone, idempotent, and that satisfies the *upper connectivity axiom* $\{\forall x \in L : \overline{\rho}(x \sqcup \overline{\rho}(x)) = \overline{\rho}(x)\}$.

COROLLARY 2.5.5.0.4

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ and $f \in \mathit{mon}(L \to L)$. The set $\mathit{fp}(f)$ of fixpoints of f is the image of L by the lower pre-closure operator $\mathit{luis}(f) \circ \mathit{llis}(\boldsymbol{\lambda}\, z \bullet z \sqcap f(z))$ and the image of L by the upper pre-closure operator $\mathit{llis}(f) \circ \mathit{luis}(\boldsymbol{\lambda}\, z \bullet z \sqcup f(z))$.

Proof: $\mathit{luis}(f) \circ \mathit{llis}(\boldsymbol{\lambda}\, z \bullet z \sqcap f(z))$ is a lower pre-closure operator on L because it is the composition of an upper closure operator $\mathit{luis}(f)$ and a lower closure operator $\mathit{llis}(\boldsymbol{\lambda}\, z \bullet z \sqcap f(z))$ (2.5.2.0.5, 2.5.3.0.1, 2.5.2.0.6, and Ladegaillerie [1973, Prop. 5,p. 41]).

End of proof.

PROPOSITION 2.5.5.0.5 *Park [1969]*

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap, \neg)$ be a complete boolean lattice and $f \in \mathit{mon}(L \to L)$. Then, $\boldsymbol{\lambda}\, x \bullet \neg f(\neg x) \in \mathit{mon}(L \to L)$, $\mathit{gfp}(f) = \neg\, \mathit{lfp}(\boldsymbol{\lambda}\, x \bullet \neg f(\neg x))$, $\mathit{lfp}(\boldsymbol{\lambda}\, x \bullet \neg f(\neg x)) \sqcap \mathit{lfp}(f) = \bot$, and $\{\{\mathit{lfp}(\boldsymbol{\lambda}\, x \bullet \neg f(\neg x)) \sqcup \mathit{lfp}(f) = \top\} \Leftrightarrow \{\mathit{lfp}(f) = \mathit{gfp}(f)\}\}$.

2.5.6 Non-computability of the fixpoints of a monotone operator of a complete lattice

We show that, in general, the extreme fixpoints of a monotone operator defined on an arbitrary complete lattice are not computable. This result points towards a potential difficulty and not an impossibility. It is does not exclude special cases in which the fixpoints are computable (e.g., when the lattice satisfies chain conditions). It tells us, however, that, in general, we have to be content with the computation of approximations of the extreme fixpoints (because they are non computable or simply because they are too complex to compute).

PROPOSITION 2.5.6.0.1

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be an arbitrary complete lattice and f an arbitrary monotone operator on L. The problem of computing $\mathit{lfp}(f)$ and $\mathit{gfp}(f)$ is undecidable.

Proof: We prove that if we are able to compute $\mathit{lfp}(f)$ for arbitrary L and $f \in \mathit{mon}(L \to L)$, we could solve a modified version of the "Post correspondence problem" which is known to be undecidable (Hopcroft & Ullman [1969]). The problem is defined as follows: "given two arbitrary sequences $A = A_1, \ldots, A_k$ and $B = B_1, \ldots, B_k$ of k non-empty chains on an alphabet with at least two characters, is there a sequence $i_1, \ldots, i_n$ of integers such that $A_1, A_{i_1}, \ldots, A_{i_n} = B_1, B_{i_1}, \ldots, B_{i_n}$?"

Let $C = \{1, \ldots, k\}$, and let $C^\star$ be the set of possibly empty chains on the alphabet C. Let $C^{1\star} = \{1s : s \in C^\star\}$. Let $\wp(C^{1\star})$ be the set of all subsets of $C^{1\star}$. Let $F = \boldsymbol{\lambda} E \bullet (\{\{1\}\} \cup \{\bigcup_{i=1}^{k} \{si : s \in x\} : x \in E\})$. It is a monotone operator on $\wp(C^{1\star})$. Consider the sequence $E^0 = \{1\}$, $E^\delta = F(E^{\delta-1})$ if δ is a successor ordinal, and $E^\delta = \bigcup_{\alpha<\delta} E^\alpha$ if δ is a limit ordinal. It is an ascending chain for the order $\subseteq$ with limit $L = \mathit{lfp}(F)$ because $\{1\} \subseteq \mathit{lfp}(F)$. Let us define a partial order on L as $\{\forall x, y \in L, \{x \sqsubseteq y\} \Leftrightarrow \{\{x = y\}$ $\underline{\text{or}}$ $\{\exists \delta \in \mu(\wp(C^{1\star})) : (x \in E^\delta)$ $\underline{\text{and}}$ $(y \notin E^\delta)\}\}\}$. The relation $\sqsubseteq$ is trivially reflexive. Assume there exists $x, y \in L$ such that $x \sqsubseteq y$, $y \sqsubseteq x$, and $x \neq y$. Then, $\exists \delta, \eta : x \in E^\delta, y \notin E^\delta$ and also $y \in E^\eta$, $x \notin E^\eta$. It is impossible that $\eta = \delta$. If $\delta < \eta$, then $x \in E^\delta \subseteq E^\eta$ contradicting $x \notin E^\eta$, otherwise $\eta < \delta$ and $y \in E^\eta \subseteq E^\delta$, contradicting $y \notin E^\delta$. *Ad absurdum* $\sqsubseteq$ is antisymmetric. Assume $x \sqsubseteq y$ and $y \sqsubseteq z$. If $x = y$ or $y = z$, then $\sqsubseteq$ is trivially transitive, otherwise $\exists \delta, \eta$ such that $x \in E^\delta$, $y \notin E^\delta$, $y \in E^\eta$, and $z \notin E^\eta$. We have necessarily $\delta \neq \eta$. If $\delta < \eta$, then

$E^\delta \subseteq E^\eta$, $x \in E^\eta$, and $z \notin E^\eta$, therefore, $x \sqsubseteq z$, otherwise $\eta < \delta$ and $y \in E^\eta \subseteq E^\delta$, contradicting $y \notin E^\delta$. We conclude that $\sqsubseteq$ is a partial order for L. Assume $x \neq y$, $x \not\sqsubseteq y$, and $y \not\sqsubseteq x$. Then, $\forall \delta, x \notin E^\delta$ or $y \in E^\delta$ and $\forall \eta, y \notin E^\eta$ and $x \in E^\eta$, in particular, for $\delta = \eta$, we obtain a contradiction which shows that $\sqsubseteq$ is a total order. L is therefore a chain with infimum $\{1\}$. Moreover, L has a supremum $\top$, otherwise the chain $\langle E^\delta : \delta \in \mu(\wp(C^{1\star}))\rangle$ would not be stationary. Indeed, assume $\forall y \in L, \exists x \in L, x \not\sqsubseteq y$. Then, $x \neq y$ and $\forall \delta$, we have $x \notin E^\delta$ and $y \in E^\delta$. In particular, for the limit $L = E^\varepsilon$, we have $x \notin L$, which is a contradiction. By reductio ad absurdum, $\exists \top \in L : \forall x \in L, x \sqsubseteq \top$. Therefore, L is a complete lattice.

Consider now the operator f on L defined as $\boldsymbol{\lambda} x \bullet \underline{\text{if}}\ \ \{\exists s \in x : s = 1i_1 \ldots i_n\ \underline{\text{and}}\ A_1, A_{i_1}, \ldots, A_{i_n} = B_1, B_{i_1}, \ldots, B_{i_n}\}\ \ \underline{\text{then}}\ \ x\ \ \underline{\text{else}}\ \ \bigcup_{i=1}^{k} \{si : s \in x\}\ \ \underline{\text{endif}}$. f is monotone, therefore it admits a least fixpoint. Moreover, the modified Post correspondence problem has a solution if and only if the least fixpoint of f is different from $\top$. If $\mathit{lfp}(f)$ was computable, then this problem would not be undecidable.
End of proof.

2.6 THE COMPLETE LATTICE OF UPPER-CONTINUOUS OPERATORS ON A COMPLETE LATTICE

DEFINITION 2.6.0.1 *Continuity*

Let ω be the first infinite limit ordinal.

We say that the operator f on $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ is *upper-continuous* if and only if, for any ascending chain $\langle x^\delta : \delta \in \omega\rangle$, we have $f(\bigsqcup_{\delta \in \omega} x^\delta) = \bigsqcup_{\delta \in \omega} f(x^\delta)$.

We say that an operator f on $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ is *lower-continuous* if and only if, for any descending chain $\langle x^\delta : \delta \in \omega\rangle$, we have $f(\bigsqcap_{\delta \in \omega} x^\delta) = \bigsqcap_{\delta \in \omega} f(x^\delta)$

An operator f on L is *continuous* if it is both upper and lower-continuous.

The name "continuity" is justified by the fact that Definition 2.6.0.1 is equivalent to the topological notion of continuity for a suitably chosen topology on L (for this, it is necessary to suppose some additional hypotheses on L, see Scott [1972], Cousot & Cousot [1977d]).

Note that, if f is a monotone operator on L, then $f(\bigsqcup_{\delta \in \mu} x^\delta) = \bigsqcup_{\delta \in \mu} f(x^\delta)$ and $f(\bigsqcap_{\delta \in \mu} x^\delta) = \bigsqcap_{\delta \in \mu} f(x^\delta)$ hold for finite ascending (respectively, descending) chains $\langle x^\delta : \delta \in \mu \rangle$ (that is to say, when $\mu < \omega$). In a lattice satisfying the ascending (respectively, descending) chain condition, all infinite chains are stationary and, therefore, all monotone operators are upper (respectively lower) continuous. Moreover, note that upper or lower continuity implies monotonicity. Finally, if an operator is a complete join-morphism (Definition 2.3.0.2) (respectively complete meet-morphism), then it is upper-continuous (respectively lower continuous).

We now characterise the set of upper-continuous operators on a complete lattice L as the image of $(L \to L)$ by a lower closure operator *ucont*. Contrarily to what has been done for monotone operators (Paragraph 2.4), we have not found any suitable definition for *ucont* which is simple enough to ensure some practical usefulness. Thus, we will be satisfied with simply proving the existence of *ucont*.

THEOREM 2.6.0.2

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice, then there exists a lower closure operator *ucont* on $(L \to L)$ such that *ucont*$(L \to L)$ is the set of upper-continuous operators on L. Moreover, for any f in $(L \to L)$, *ucont*(f) is the greatest upper-continuous operator on L smaller than or equal to f.

Proof: Let C be the subset of the complete lattice $(L \to L)$ corresponding to the upper-continuous operators on L. Then $(\boldsymbol{\lambda} x \cdot \bot) \in C$ and for any non-empty subset S of C we have $(\sqcup S) \in C$. Indeed, let $\langle x^\delta : \delta \in \omega \rangle$ be an ascending chain in L. For any f in S we have $f(\bigsqcup_{\delta \in \omega} x^\delta) = \bigsqcup_{\delta \in \omega} f(x^\delta)$. So, $\bigsqcup_{f \in S} f(\bigsqcup_{\delta \in \omega} x^\delta) = \bigsqcup_{f \in S} \bigsqcup_{\delta \in \omega} f(x^\delta) = \bigsqcup_{\delta \in \omega} \bigsqcup_{f \in S} f(x^\delta)$, that is to say, $(\sqcup S)(\bigsqcup_{\delta\in\omega} x^\delta) = \bigsqcup_{\delta\in\omega}(\sqcup S)(x^\delta)$.

Let *ucont* be the function from $(L \longrightarrow L)$ to itself defined as:

$$ucont \quad = \quad \boldsymbol{\lambda} f \bullet \sqcup \{C \cap \{g \in (L \longrightarrow L) : f \sqsupseteq g\}\}$$

Then, *ucont* is a lower closure operator where the closed elements are the elements in C. For any $f \in (L \longrightarrow L)$, $ucont(f)$ is the greatest element of C smaller than or equal to f because, if $g \in ucont(L \longrightarrow L)$ and $f \sqsupseteq g$, then $ucont(f) \sqsupseteq ucont(g) = g$.
End of proof.

2.7 FIXPOINT THEOREM FOR UPPER CONTINUOUS OPERATORS ON A COMPLETE LATTICE

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice such that $\mu(L) > \omega$. A necessary and sufficient condition for an increasing iteration $\langle x^\delta : \delta \in \mu(L)\rangle$ defined by $f \in (L \longrightarrow L)$ and starting from $d \in L$ which is an ascending chain to be stationary after ω is:

$$\begin{array}{lll}
 & \{\forall \delta \in \mu(L), \{\delta \geq \omega\} \Rightarrow \{x^\omega = x^\delta\}\} & \\
\Leftrightarrow & \{x^\omega = x^{\omega+1}\} & \text{(Lemma 2.5.1.0.4)} \\
\Leftrightarrow & \{x^\omega = f(x^\omega)\} & \text{(Definition 2.5.1.0.1.(b))} \\
\Leftrightarrow & \{\bigsqcup\limits_{\alpha<\omega} x^\alpha = f(\bigsqcup\limits_{\alpha<\omega} x^\alpha)\} & \text{(Definition 2.5.1.0.1.(c))} \\
\Leftrightarrow & \{\bigsqcup\limits_{\alpha<\omega} x^{\alpha+1} = f(\bigsqcup\limits_{\alpha<\omega} x^\alpha)\} & \text{(because } \{\{\alpha < \omega\} \Leftrightarrow \{(\alpha+1) < \omega\}\}) \\
\Leftrightarrow & \{\bigsqcup\limits_{\alpha<\omega} f(x^\alpha) = f(\bigsqcup\limits_{\alpha<\omega} x^\alpha)\} & \text{(Definition 2.5.1.0.1.(b))}
\end{array}$$

We can observe that the upper-continuous hypothesis is “designed” so as to avoid transfinite iterations to be considered (more precisely, iterations with strictly more than ω terms). Actually, this hypothesis is slightly too strong because it is not required for the chain $\langle x^\delta : \delta \in \omega\rangle$ to satisfy Definition 2.5.1.0.1. By applying Theorem 2.5.2.0.2 we have:

PROPOSITION 2.7.0.1

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $f \in \mathit{ucont}(L \rightarrow L)$. An increasing iteration $\langle x^\delta : \delta \in \underline{\min}(\mu(L), \omega)\rangle$ starting from $d \in \mathit{prefp}(f)$ and defined by f is a stationary ascending chain, and its limit $\mathit{luis}(f)(d)$ is the least fixpoint of f greater than or equal to d.

2.8 FORMAL METHOD TO SOLVE A SYSTEM OF MONOTONE FIXPOINT EQUATIONS BY MEANS OF VARIABLE ELIMINATION

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice, then the set L^n of n-tuples of elements in L is a complete lattice for the partial order:

$$\{(X_1, \ldots, X_n) \sqsubseteq (Y_1, \ldots, Y_n)\} \quad \Leftrightarrow \quad \{\forall i \in [1, n], X_i \sqsubseteq Y_i\}$$

Then we have:

$$\begin{array}{rcl} \bigsqcup\limits_{i\in\Delta}(X_1^i, \ldots, X_n^i) & = & (\bigsqcup\limits_{i\in\Delta} X_1^i, \ldots, \bigsqcup\limits_{i\in\Delta} X_n^i) \\ \bigsqcap\limits_{i\in\Delta}(X_1^i, \ldots, X_n^i) & = & (\bigsqcap\limits_{i\in\Delta} X_1^i, \ldots, \bigsqcap\limits_{i\in\Delta} X_n^i) \end{array}$$

so that the infimum of L^n is $(\bot, \ldots, \bot)$ and the supremum is $(\top, \ldots, \top)$. With a slight abuse of notation we will write $L^n(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$.

Consider a system of equations with n variables $(X_1, \ldots, X_n) \in L^n$ having the following form:

$$\left\{ \begin{array}{lcl} X_1 & = & F_1(X_1, \ldots, X_n) \\ \ldots & & \\ X_n & = & F_n(X_1, \ldots, X_n) \end{array} \right.$$

where $\{\forall i \in [1,n], F_i \in (L^n \to L)\}$, which we will denote as $X = F(X)$ where $F \in (L^n \to L^n)$. If $F_i (i = 1, \ldots, n)$ are monotone, then F is monotone and Theorem 2.5.5.0.1 gives us a constructive definition of the solutions for this system of equations.

More generally, with each equation $X = F(X)$ on a lattice L, we can associate a system of equations on a direct decomposition of L having the same solutions up to an isomorphism:

PROPOSITION 2.8.0.1

Let $L(\sqsubseteq, \perp, \top, \sqcup, \sqcap)$ and $\forall i \in [1,n]$, $M_i(\sqsubseteq, \perp, \top, \sqcup, \sqcap)$ be complete lattices such that $\prod_{i=1}^{n} M_i$ is a *direct decomposition* of L (that is to say that there exists decomposition morphisms $\sigma_i \in (L \to M_i)$, such that $\sigma = \boldsymbol{\lambda} x \bullet (\sigma_1(x), \ldots, \sigma_n(x))$ is a complete isomorphism from L to $\prod_{i=1}^{n} M_i$). Let $F \in mon(L \to L)$, then the set $fp(F)$ of fixpoints of F and the set of solutions of the system of equations:

$$\left\{ \begin{array}{rcl} X_i & = & \sigma_i \circ F \circ \sigma^{-1}(X_1, \ldots, X_n) \\ i & = & 1, \ldots, n \end{array} \right.$$

are completely isomorphic by σ.

Additionally to the iterative method, a formal computation proceeding by variable elimination also allows solving the system of equations. Variable X_i is eliminated by solving the equation $X_i = (\boldsymbol{\lambda} Y \bullet F_i(X_1, \ldots, Y, \ldots, X_n))(X_i)$ in terms of the X_j $(j \neq i)$ considered here as free variables, then substituting the solution at all occurrences of X_i inside the remaining equations. This process is repeated until all variables have been eliminated.

PROPOSITION 2.8.0.2 *Bekić [1969], Leszczylowski [1971]*

Let $F \in mon(L^{n+m} \to L^n)$ and $G \in mon(L^{n+m} \to L^m)$. We will note $(X, Y) = (X_1, \ldots, X_n, Y_1, \ldots, Y_m)$ when $X \in L^n$ and $Y \in L^m$. We consider the system of equations,

$$(1) \quad \begin{cases} X & = & F(X, Y) \\ Y & = & G(X, Y) \end{cases}$$

the *resolvent* $R = \boldsymbol{\lambda} Y \cdot \mathit{lfp}(\boldsymbol{\lambda} X \cdot F(X, Y))$, and the system of equations,

$$(2) \quad \begin{cases} X & = & R(Y) \\ Y & = & G(R(Y), Y) \end{cases}$$

We note $\mathit{fp}(i)$ and $\mathit{lfp}(i), (i = 1, 2)$ respectively the set of solutions of the system of equations (i) and its least solution. Then, $\mathit{fp}(2) \subseteq \mathit{fp}(1)$ and $\mathit{lfp}(1) = \mathit{lfp}(2)$.

Proof: If $Y, Z \in L^m$, then $\{Y \sqsubseteq Z\} \Rightarrow \{(X, Y) \sqsubseteq (X, Z)\} \Rightarrow \{\boldsymbol{\lambda} X \cdot F(X, Y) \sqsubseteq \boldsymbol{\lambda} X \cdot F(X, Z)\} \Rightarrow \{\mathit{lfp}(\boldsymbol{\lambda} X \cdot F(X, Y)) \sqsubseteq \mathit{lfp}(\boldsymbol{\lambda} X \cdot F(X, Z))\} \Rightarrow \{R(Y) \sqsubseteq R(Z)\} \Rightarrow \{R \in mon(L^m \to L^n)\}$, therefore, $\mathit{fp}(2)$ is non-empty.
Let $(A_2, B_2) \in \mathit{fp}(2)$ be a fixpoint of (2). Then, $A_2 = R(B_2)$, that is to say, $\mathit{lfp}(\boldsymbol{\lambda} X \cdot F(X, B_2)) = A_2$, then $F(A_2, B_2) = A_2$ and $B_2 = G(R(B_2), B_2)$, and so, $B_2 = G(A_2, B_2)$, which implies that (A_2, B_2) is a solution of the system of equations (1), that is to say, $\mathit{fp}(2) \subseteq \mathit{fp}(1)$.

In order to show that, in general, $\mathit{fp}(2) \neq \mathit{fp}(1)$, we consider $L = \{\bot, \top\}$, $F = \boldsymbol{\lambda}(X, Y) \cdot [X \sqcap Y]$, and $G = \boldsymbol{\lambda}(X, Y) \cdot [X \sqcup Y]$. The resolvent is $R = \boldsymbol{\lambda} Y \cdot \mathit{lfp}(\boldsymbol{\lambda} X \cdot F(X, Y)) = \boldsymbol{\lambda} Y \cdot \mathit{lfp}(\boldsymbol{\lambda} X \cdot [X \sqcap Y]) = \boldsymbol{\lambda} Y \cdot \bot$. The system of equations (1) admits a solution $(\top, \top)$ which is not a solution of (2).

As $\mathit{lfp}(2) \in \mathit{fp}(1)$, we have $\mathit{lfp}(1) \sqsubseteq \mathit{lfp}(2)$. Let (A_1, B_1) be a fixpoint of (1), then we have $F(A_1, B_1) = A_1$ and therefore $F(A_1, B_1) \sqsubseteq A_1$, which implies that A_1 is a post-fixpoint of $\boldsymbol{\lambda} X \cdot F(X, B_1)$ from which it follows that $\mathit{lfp}(\boldsymbol{\lambda} X \cdot F(X, B_1)) \sqsubseteq A_1$, and so, $R(B_1) \sqsubseteq A_1$. As $(R(B_1), B_1) \sqsubseteq (A_1, B_1)$ and G is monotone, $G(R(B_1), B_1) \sqsubseteq$

$G(A_1, B_1) \sqsubseteq B_1$ because (A_1, B_1) is a post-fixpoint of (1). We deduce that (A_1, B_1) is a post-fixpoint of (2), which implies that $\mathit{lfp}(2) \sqsubseteq (A_1, B_1)$, so that, in particular, $\mathit{lfp}(2) \sqsubseteq \mathit{lfp}(1)$ and, by antisymmetry, $\mathit{lfp}(2) = \mathit{lfp}(1)$.
End of proof.

2.9 CHAOTIC, ASYNCHRONOUS, AND ASYNCHRONOUS WITH MEMORY ITERATIVE METHODS TO SOLVE A SYSTEM OF MONOTONE FIXPOINT EQUATIONS ON A COMPLETE LATTICE

The increasing iteration starting from $D \in L^n$ and defined by $F \in \mathit{mon}(L^n \to L^n)$ (Definition 2.5.1.0.1) proceeds as follows:

- $X^0 = D$
- $\begin{cases} X_i^\delta = F_i(X_1^{\delta-1}, \ldots, X_n^{\delta-1}) \\ i = 1, \ldots, n \end{cases}$ if δ is a successor ordinal
- $X^\delta = \bigsqcup\limits_{\alpha<\delta} X^\alpha$ if δ is a limit ordinal

which indeed corresponds to the method of successive approximations. We can think that the Gauss–Seidel's method:

$$\begin{cases} X_1^\delta = F_1(X_1^{\delta-1}, \ldots, X_n^{\delta-1}) \\ \ldots \\ X_i^\delta = F_i(X_1^\delta, \ldots, X_{i-1}^\delta, X_i^{\delta-1}, \ldots, X_n^{\delta-1}) \\ \ldots \\ X_n^\delta = F_n(X_1^\delta, \ldots, X_{n-1}^\delta, X_n^{\delta-1}) \end{cases}$$

consisting in continually re-injecting inside the computation the last resultants of the computation itself, enforces convergence and reduces memory usage. Robert [1976a] shows that in general these methods are not equivalent. Without strong enough hypotheses on L^n and F, it is possible that either the method of successive approximations or Gauss–Seidel converges while the other diverges.

We are going to show that, when L is a complete lattice and $F \in mon(L^n \rightarrow L^n)$, this phenomenon is impossible, that is, both strategies produce the same result. More generally, in Paragraph 2.9.1, we show that the hypothesis of monotonicity allows us to use any *chaotic strategy*, that is, in the iteration we can randomly determine at each step which components are computed as a function of the preceding iteration (on the condition that we never indefinitely forget any component) and obtain the convergence towards the same fixpoint as with the method of successive approximations.

In Paragraph 2.9.2, this result is generalised to the case of *asynchronous iterative methods*, offering the possibility of delays when accessing the previously computed iterations, therefore corresponding to the use of many processors working in parallel without (necessarily) being synchronised.

In fact, chaotic and asynchronous iterations are only particular cases of the *iterative asynchronous methods with memory* considered in Paragraph 2.9.3. The iterative asynchronous methods with memory permit not only the decomposition of the computation into groups of concurrently evolving components, but also the decomposition according to the structure of each component, as different values for each variable can be used when computing a component.

In practice, we consider iterative methods converging in a finite number of steps, but we will prove all our convergence results for transfinite iterations anyway. From an abstract point of view, this allows us to separate the problem of the termination of the iteration (number of iterations) from that of its convergence (stabilisation of the iteration). (For example, in order to find X such that $X \sqsubseteq lfp(F)$, we can compute the increasing iteration $\langle X^\delta : \delta \in \mu(L)\rangle$ starting from $\perp$ and defined by F, and stop the computation after some number of steps fixed beforehand or when precision is sufficient. The algorithm is correct because, from an abstract point of view, every term X^{δ} is smaller than the limit $luis(F)(\bot) = lfp(F)$ of the transfinite iteration, therefore, only the very first terms are effectively computed.)

2.9.1 Convergence of chaotic iterations

DEFINITION 2.9.1.0.1 *Increasing chaotic iteration*

- Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice, n a strictly positive integer, and $F \in mon(L^n \to L^n)$.
- Let $\langle J^{\delta} : \delta \in \mathrm{Ord}\rangle$ be a sequence of subsets of $\{1, \ldots, n\}$ such that:

(a) - $\{\forall \delta \in Ord, \forall i \in \{1, \ldots, n\}, \exists \alpha \geq \delta : i \in J^{\alpha}\}$

- The *increasing chaotic iteration starting from $D \in L^n$ and defined by F and $\langle J^{\delta} : \delta \in \mathrm{Ord}\rangle$* is a sequence $\langle X^{\delta} : \delta \in \mu(L)\rangle$ of elements in L^n defined by transfinite induction as follows:

$$\begin{array}{lcll}
\text{(b) - } X^0 & = & D & \\
\text{(c) - } X_i^{\delta} & = & X_i^{\delta-1} & \text{for any successor ordinal } \delta \text{ and } i \notin J^{\delta} \\
\text{(d) - } X_i^{\delta} & = & F_i(X^{\delta-1}) & \text{for any successor ordinal } \delta \text{ and } i \in J^{\delta} \\
\text{(e) - } X_i^{\delta} & = & \bigsqcup\limits_{\alpha<\delta} X_i^{\alpha} & \text{for any limit ordinal } \delta
\end{array}$$

This definition can be interpreted by considering that, at step δ of the computation, only the components $i \in J^{\delta}$ evolve in view of the values obtained in previous steps. The condition 2.9.1.0.1.(a) imposes that no component is forever omitted. The abstract case 2.9.1.0.1.(e) has been added to the classic definition in order to make the definition compatible with 2.5.1.0.1.

For instance, the method of successive approximations consists in taking $\{\forall \delta \in Ord,\ J^{\delta} = \{1, \ldots, n\}\}$ while Gauss–Seidel's method corresponds to choosing $J^{\delta} = \{1\}$ if $\delta = 1$ or δ is the successor of a limit ordinal and $J^{\delta} = \{1 + (j \ \underline{\text{modulo}}\ n)\}$ if δ is a successor ordinal and $J^{\delta-1} = \{j\}$.

THEOREM 2.9.1.0.2

An increasing chaotic iteration starting from $D \in \mathit{prefp}(F)$ and defined by $F \in \mathit{mon}(L^n \to L^n)$ and $\langle J^\delta : \delta \in \mathrm{Ord}\rangle$ is a stationary ascending chain with limit $\mathit{luis}(F)(D)$.

Proof: This is a special case of Theorem 2.9.2.0.2. In order to prove additionally that it is an ascending chain, see Cousot & Cousot [1977e].
End of proof.

As all the components $X_i^{\delta+1}$ (such that $i \in J^{\delta+1}$) are evaluated using X^δ, Definition 2.9.1.0.1 implicitly assumes that the computation is made by a single sequential process or by several synchronized parallel processes (for instance, one for each $i \in J^{\delta+1}$). The definition of asynchronous iterations instead avoids the synchronization constraint by allowing delays when accessing previously computed iterates.

2.9.2 Convergence of asynchronous iterations

DEFINITION 2.9.2.0.1 *Increasing asynchronous iterations*

- Let $\langle J^\delta : \delta \in \mathrm{Ord} \rangle$ be a sequence of elements of $\{1, \ldots, n\}$ such that:

(a) - $\{\forall \delta \in \mathrm{Ord}, \forall i \in \{1, \ldots, n\}, \exists \alpha \geq \delta : i = J^\alpha\}$

- Let $\langle S^\delta : \delta \in \mathrm{Ord} \rangle$ be a sequence of elements in Ord^n such that:

(b) - $\{\forall i \in \{1, \ldots, n\}, \forall \delta \in \mathrm{Ord}, S_i^\delta < \delta\}$

(c) - $\{\forall \delta \in \mathrm{Ord}, \forall i \in \{1, \ldots, n\}, \exists \beta \geq \delta : \{\forall \alpha \geq \beta, \delta \leq S_i^\alpha\}\}$

(d) - $\{\forall \beta, \delta \in \mathrm{Ord}, \{\beta \text{ is a limit ordinal } \underline{\text{and}}\ \beta < \delta\} \Rightarrow \{\forall i \in \{1, \ldots, n\}, \beta \leq S_i^\delta\}\}$

- Let F be a monotone operator on a complete lattice L^n. An *increasing asynchronous iteration for F starting from $D \in L^n$ and defined by $\langle J^\delta : \delta \in \mathrm{Ord} \rangle$ and* $\langle S^\delta : \delta \in \mathrm{Ord} \rangle$ is a sequence $\langle X^\delta : \delta \in \mathrm{Ord} \rangle$ of elements in L^n defined by transfinite induction as follows:

$$\begin{array}{lcll}
\text{(e) - } X^0 & = & D & \\
\text{(f) - } X_i^\delta & = & X_i^{\delta-1} & \text{for any successor ordinal } \delta \text{ and } i \notin J^\delta \\
\text{(g) - } X_i^\delta & = & F_i(X_1^{S_1^\delta}, \ldots, X_n^{S_n^\delta}) & \text{for any successor ordinal } \delta \text{ and } i = J^\delta \\
\text{(h) - } X_i^\delta & = & \bigsqcup\limits_{\alpha < \delta} X_i^\alpha & \text{for any limit ordinal } \delta
\end{array}$$

The definition of asynchronous iterations is due to numeric analysts Chazan & Miranker [1969] but the form given here is more similar to the version of Baudet [1976]. We have nevertheless simplified this definition by assuming that $\forall \delta \in \mathrm{Ord}, J^\delta \in \{1.....n\}$ instead of $J^\delta \subseteq \{1.....n\}$. This does not introduce any restriction: indeed, when several components are updated at the same time, we consider from an *abstract* point of view that they have been computed the same way but updated at different times. (For example, the method of successive approximations corresponds to the choice $J^{\alpha+j_n+i} = i$ and $S_k^{\alpha+j_n+i} = \alpha + j_n$ for any limit ordinal α and integers $j \geq 0$, $1 \leq i \leq n$, and $1 \leq k \leq n$. More generally, chaotic iterations are a particular case of asynchronous iterations.) We also added the rule 2.9.2.0.1.(h) as well as the condition 2.9.2.0.1.(d) so that the notion of asynchronous iteration is compatible with transfinite iterations 2.5.1.0.1.

Asynchronous iterations offer a model for potential computations on a multiprocessor. A global memory, initialised with D, is accessible by each processor that can read and write each component X_i of the global memory X. These operations are mutually exclusive in time and can therefore be considered as instantaneous. (Depending on the size of the memories X_i $(i = 1, \ldots, n)$, this mutual exclusion is assured by hardware or software solutions, this is the only elementary synchronisation that is needed between the different processors.)

We interpret $\langle \delta : \delta \in \mathrm{Ord} \rangle$ as a growing sequence of instants δ in which a read or a write of component X_i takes place. When a processor is idle, it is assigned the evaluation of an arbitrary component of the system of equations. Definition 2.9.2.0.1 indicates that, at time δ, a processor ends the evaluation of the ith component such that $i = J^\delta$. The corresponding value X_i^δ is instantaneously written to memory X_i. Evaluating X_i^δ consists in reading the value $X_1^{S_1^\delta}$ from memory X_1 at time S_1^δ, ..., reading X_n at time S_n^δ, applying F_i to the arguments $X_1^{S_1^\delta}, \ldots, X_n^{S_n^\delta}$ and writing the corresponding value $X_i^\delta = F_i(X_1^{S_1^\delta}, \ldots, X_n^{S_n^\delta})$ at time δ into X_i. All the components X_j of X such that $j \neq J^\delta$ are not modified at time δ.

We should point out that no synchronisation is necessary between the processors contributing to this computation and the distribution of the tasks for the different processors is free. In particular, a processor that breaks down can be eliminated from the pool of available processors and replaced in its task by other, working processors. The distribution of tasks for the different processors must in all cases respect the condition 2.9.2.0.1.(a) which states that no component can be definitely abandoned.

(This hypothesis is a bit too strong given that, for example, a constant component is always stable. To make our formalism lighter, we do not consider these situations.)

It is also natural to assume that the evaluation of each $F_i(X_1^{S_1^\delta},\ldots,X_n^{S_n^\delta})$ should last a finite time (but not necessarily uniformly bounded). Therefore, for each δ, the duration of the computation $\underline{\overset{n}{\max}}_{i=1}(\delta - S_i^\delta)$ must be finite. This is what the condition 2.9.2.0.1.(c) expresses under a slightly different form: for each δ, there exists $\beta \geq \delta$ such that, after time β, no processor can finish a computation that was started at the time δ.

The hypothesis that the elementary evaluation of a component should last a finite time should hold even if we consider transfinite iterations. The condition 2.9.2.0.1.(d) is at this point necessary to keep this fact into account. By reduction ad absurdum, let us suppose that the computation of X_i^δ, started at the time $\alpha = \underline{\overset{n}{\max}}_{i=1}(S_i^\delta)$, takes r time units (where $r < \omega$ because the computation length can be arbitrarily long, but finite). We have $(\alpha + r) = \delta$. Let us suppose that there exists a limit ordinal β and $j \in \{1,\ldots,n\}$ such that $S_j^\delta < \beta < \delta$. Then $\alpha < \beta < \delta$ and, as β is a limit ordinal $(\alpha + 1) < \beta$, then, by finite induction, $(\alpha + r) < \beta < \delta$, contradicting $(\alpha + r) = \delta$.

THEOREM 2.9.2.0.2

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice, n a natural number, and $F \in mon(L^n \rightarrow L^n)$. An asynchronous increasing iteration for F starting from $D \in prefp(F)$ and defined by $\langle J^\delta : \delta \in \mathrm{Ord}\rangle$, $\langle S^\delta : \delta \in \mathrm{Ord}\rangle$ is a stationary sequence with limit $luis(F)(D)$.

Proof: This is a particular case of Theorem 2.9.3.0.9 where $m = 1$.

End of proof.

2.9.3 Convergence of asynchronous iterations with memory

DEFINITION 2.9.3.0.1 *Increasing asynchronous iteration with memory*

- Let $\langle J^\delta : \delta \in \text{Ord}\rangle$ be a sequence of elements in $\{1, \ldots, n\}$ such that:

(a) - $\{\forall \delta \in \text{Ord}, \forall i \in \{1, \ldots, n\}, \exists \alpha \geq \delta : i = J^\alpha\}$

- Let $\langle S^\delta : \delta \in \text{Ord}\rangle$ be a sequence of elements in $(\text{Ord}^n)^m$ such that:

(b) - $\{\forall i \in \{1, \ldots, n\}, \forall j \in \{1, \ldots, m\}, \forall \delta \in \text{Ord}, (S^\delta_j)_i < \delta\}$

(c) - $\{\forall \delta \in \text{Ord}, \forall i \in \{1, \ldots, n\}, \forall j \in \{1, \ldots, m\}, \exists \beta \geq \delta : \{\forall \alpha \geq \beta, \delta \leq (S^\alpha_j)_i\}\}$

(d) - $\{\forall \beta, \delta \in \text{Ord}, \{\beta \text{ is a limit ordinal } \underline{\text{and}}\ \beta < \delta\}$
$\Rightarrow \quad \{\forall i \in \{1, \ldots, n\}, \forall j \in \{1, \ldots, m\}, \beta \leq (S^\delta_j)_i\}\}$

- Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and F a monotone application from $(L^n)^m$ to L^n. An *increasing asynchronous iteration with memory for F starting from $D \in L^n$ and defined by $\langle J^\delta : \delta \in \text{Ord}\rangle$ and $\langle S^\delta : \delta \in \text{Ord}\rangle$* is a sequence $\langle X^\delta : \delta \in \text{Ord}\rangle$ of elements of L^n defined by transfinite induction as follows:

(e) - $X^0 = D$

(f) - $X^\delta_i = X^{\delta-1}_i$ for any successor ordinal δ and $i \neq J^\delta$

(g) - $X^\delta_i = F_i(Z^1, \ldots, Z^n)$ for any successor ordinal δ and $i = J^\delta$ where $\forall j \in \{1, \ldots, m\}, \forall i \in \{1, \ldots, n\}, Z^j_i = X_i^{(S^\delta_j)_i}$

(h) - $X^\delta_i = \bigsqcup_{\alpha < \delta} X^\alpha_i$ for any limit ordinal δ

We observe that, whenever $m = 1$, this definition is equivalent to Definition 2.9.2.0.1.

Let σ be the function from L^n to $(L^n)^m$ such that $\{\forall X \in L^n, \forall i \in \{1, \ldots, n\}, (\sigma(X))_i = X\}$. Let F be a monotone function from $(L^n)^m$ to L^n. We define a fixpoint of F as any element X in L^n such that $X = F(X, \ldots, X)$, namely $X = F(\sigma(X))$. Because $F \circ \sigma \in mon(L^n \to L^n)$, all the results from Paragraph 2.5 can be applied to

$F \circ \sigma$.

In order to present a practical application of Definition 2.9.3.0.1, assume that we have to compute $luis(f)(D)$ where $f \in mon(L^n \to L^n)$ and $D \in prefp(f)$, and that we can find a natural number m and $F \in ((L^n)^m) \to L^n)$ such that $luis(f)(D) = luis(F \circ \sigma)(D)$. We can then use whatever increasing asynchronous iteration with memory we wish to compute it as we are going to show that it is a stationary sequence with limit $luis(F \circ \sigma)(D)$.

For instance, let $f = \boldsymbol{\lambda} X \bullet (g(X) \sqcup h(X))$. Then, $luis(f)(D) = luis(F \circ \sigma)(D)$ where $F = \boldsymbol{\lambda}(X, Y) \bullet (g(X) \sqcup h(Y))$. A possible iteration with two memories for F is:

$$\begin{cases} X^0 & = & \bot \\ X^1 & = & \bot \\ X^{\delta+2} & = & F(X^{\delta+1}, X^{\delta}) \end{cases}$$

which is equivalent to two collateral iterations:

$$\begin{cases} X^0 & = & \bot \\ X^{\delta+1} & = & g(X^{\delta}) \sqcup Y^{\delta} \end{cases} \qquad \begin{cases} Y^0 & = & \bot \\ Y^{\delta+1} & = & h(X^{\delta}) \end{cases}$$

which are a natural decomposition of the computation which cannot be described by Definition 2.9.2.0.1.

In the rest of the paragraph, we consider an increasing asynchronous iteration with m memories $\langle X^{\delta} : \delta \in \text{Ord}\rangle$ for a monotone function F from $(L^n)^m$ to L^n starting from a pre-fixpoint D of F ($D \sqsubseteq F \circ \sigma(D)$) and defined by $\langle J^{\delta} : \delta \in \text{Ord}\rangle$ and $\langle S^{\delta} : \delta \in \text{Ord}\rangle$ as in Definition 2.9.3.0.1.

LEMMA 2.9.3.0.2

$$\{\forall \delta \in \text{Ord}, D \sqsubseteq X^{\delta} \sqsubseteq luis(F \circ \sigma)(D)\}$$

Proof: By transfinite induction on δ by considering the fact that $D \sqsubseteq F \circ \sigma(D) \sqsubseteq luis(F \circ \sigma)(D) = F \circ \sigma(luis(F \circ \sigma)(D))$ (Theorem 2.5.2.0.2). More details can be found in Cousot [1977d].

End of proof.

DEFINITION 2.9.3.0.3

The condition 2.9.3.0.1.(a) implies that for all $\delta \in \text{Ord}$, there is an ordinal $\Pi(\delta)$ defined as:

$$\Pi(\delta) \quad = \quad \underline{\min}\{\alpha \in \text{Ord} : (\delta \leq \alpha) \ \underline{\text{and}}\ (\{1, \ldots, n\} = \{J^\beta : \delta \leq \beta \leq \alpha\})\}$$

Intuitively, between the steps δ and $\Pi(\delta) + 1$, all the components have been relaxed at least one time, that is to say

$$\{\forall i \in \{1, \ldots, n\}, \forall \delta \in \text{Ord}, \exists \beta \in \text{Ord} : (\delta \leq \beta \leq \Pi(\delta))\ \underline{\text{and}}\ (i = j^\beta)\}$$

DEFINITION 2.9.3.0.4

The condition 2.9.3.0.1.(c) implies that for all $\delta \in \text{Ord}$, there exists an ordinal $\lambda(\delta)$ defined as:

$$\lambda(\delta) \quad = \quad \underline{\min}\{\beta \in \text{Ord} : \forall j \in \{1, \ldots, m\}, \forall i \in \{1, \ldots, n\}, \forall \alpha \geq \beta, \delta \leq (S_j^\alpha)_i\}$$

Intuitively, a computation that ends at step $\lambda(\delta) + 1$ cannot have read the components intervening in the computation before step δ, that is to say

$$\{\forall j \in \{1, \ldots, m\}, \forall i \in \{1, \ldots, n\}, \forall \alpha \geq \lambda(\delta), \delta \leq (S_j^\alpha)_i\}$$

DEFINITION 2.9.3.0.5

We denote by $\langle \eta^\delta : \delta \in \mathrm{Ord}\rangle$ the transfinite sequence of ordinals defined by transfinite induction as follows:

$$\begin{array}{llll} \text{(a) -} & \eta^0 & = & 0 \\ \text{(b) -} & \eta^\delta & = & \Pi(\lambda(\eta^{\delta-1}))+1 \quad \text{if } \delta \text{ is a successor ordinal} \\ \text{(c) -} & \eta^\delta & = & \bigcup\limits_{\alpha<\delta} \eta^\alpha \quad \text{if } \delta \text{ is a limit ordinal} \end{array}$$

LEMMA 2.9.3.0.6

The sequence $\langle \eta^\delta : \delta \in \mathrm{Ord}\rangle$ is strictly increasing and for any limit ordinal δ, η^δ is also a limit ordinal.

Proof: We prove that $\{\forall\gamma, \beta \in \mathrm{Ord}, \{\beta < \gamma\} \Rightarrow \{\eta^\beta < \eta^\gamma\}\}$ by transfinite induction on γ. The case $\gamma = 0$ is trivial. Assume that for all $\gamma < \delta$ we have $\{\forall \beta \in \mathrm{Ord}, \{\beta < \gamma\} \Rightarrow \{\eta^\beta < \eta^\gamma\}\}$. If δ is a successor ordinal, then for any ordinal $\beta < \delta$, we have, if $\beta < (\delta - 1)$, then $\eta^\beta < \eta^{\delta-1} < \Pi(\lambda(\eta^{\delta-1})) + 1 = \eta^\delta$ by induction hypothesis and $\Pi(\lambda(\eta)) \geq \eta$ for all $\eta \in \mathrm{Ord}$, otherwise $\beta = (\delta-1)$ and $\eta^\beta = \eta^{\delta-1} < \Pi(\lambda(\eta^{\delta-1}))+1 = \eta^\delta$. If δ is a limit ordinal and $\beta < \delta$, then there exists ε such that $\beta < \varepsilon < \delta$ and by induction hypothesis $\eta^\beta < \eta^\varepsilon \leq \bigcup\limits_{\alpha<\delta} \eta^\alpha$. By transfinite induction $\langle \eta^\delta : \delta \in \mathrm{Ord}\rangle$ is strictly increasing.

Let δ be a limit ordinal and β be an ordinal such that $\beta < \eta^\delta$. In order to prove that η^δ is a limit ordinal, it is sufficient to prove that there exists an ordinal γ such that $\beta < \gamma < \eta^\delta$. Because $\eta^\delta = \bigcup\limits_{\alpha<\delta} \eta^\alpha$, there exists $\alpha < \delta$ such that $\beta < \eta^\alpha$. Because $\langle \eta^\delta : \delta \in \mathrm{Ord}\rangle$ is strictly increasing, $\eta^\alpha < \eta^\delta$ and, by transitivity, $\beta < \eta^\alpha < \eta^\delta$ and η^δ is a limit ordinal.

End of proof.

DEFINITION 2.9.3.0.7

We denote by $\langle B^\delta : \delta \in \mathrm{Ord}\rangle$ the sequence of elements of L^n defined by transfinite induction as follows:

$$\begin{array}{llll} \text{(a) -} & B^0 & = D & \\ \text{(b) -} & B^\delta & = F(\sigma(B^{\delta-1})) & \text{if } \delta \text{ is a successor ordinal} \\ \text{(c) -} & B^\delta & = \bigsqcup_{\alpha<\delta} B^\alpha & \text{if } \delta \text{ is a limit ordinal} \end{array}$$

LEMMA 2.9.3.0.8

$$\{\forall \beta, \delta \in \mathrm{Ord}, \{\eta^\delta \leq \beta\} \Rightarrow \{B^\delta \sqsubseteq X^\delta\}\}$$

Proof: By induction on δ (case 1, 2, 3).

Case 1: If $\delta = 0$, then $\forall \beta \geq \eta^0 = 0$, and we have $B^0 = D \sqsubseteq X^\beta$ (Lemma 2.9.3.0.2).

Case 2: Assuming that δ is a successor ordinal and that the lemma holds for all $\delta' < \delta$, we prove by transfinite induction on β that the lemma holds also for δ (case 2.1, 2.2, 2.3).

Case 2.1: If $\beta = \eta^\delta$, then, by 2.9.3.0.5.(b), $\eta^\delta = \Pi(\lambda(\eta^{\delta-1})) + 1$. Therefore, for all $i = 1, \ldots, n$ there is a greatest ordinal ε such that $\lambda(\eta^{\delta-1}) \leq \varepsilon \leq \beta$ and $i = J^\varepsilon$. As a consequence of 2.9.3.0.1, we have therefore $X_i^\varepsilon = X_i^{\varepsilon+1} = \ldots = X_i^\beta$ with $X_i^\varepsilon = F_i(Z^1, \ldots, Z^m)$. As $\varepsilon \geq \lambda(\eta^{\delta-1})$, we have $\{\forall j \in \{1, \ldots, m\}, \forall k \in \{1, \ldots, n\}, \eta^{\delta-1} \leq (S_j^\varepsilon)_k < \varepsilon\}$. By induction hypothesis, it follows that $B_k^{\delta-1} \sqsubseteq X_k^{(S_j^\varepsilon)_k} = Z_k^j$. We deduce that $\{\forall j \in \{1, \ldots, m\}, B^{\delta-1} \sqsubseteq Z^j\}$, then, by 2.9.3.0.7.(b) and monotonicity, we obtain $B_i^\delta = F_i(\sigma(B^{\delta-1})) \sqsubseteq F_i(Z^1, \ldots, Z^m) = X_i^\varepsilon = X_i^\beta$, so that $B_i^\delta \sqsubseteq X_i^{\eta^\delta}$.

Case 2.2: Assume that β is a successor ordinal and that for all α such that $\eta^\alpha \leq \alpha < \beta$ we have $B^\delta \sqsubseteq X^\alpha$. We prove that $\{\forall i \in \{1, \ldots, n\}, B_i^\delta \sqsubseteq X_i^\beta\}$. If $i \neq J^\beta$, then $B_i^\beta \sqsubseteq X_i^{\beta-1} = X_i^\beta$, otherwise $i = J^\beta$ and $X_i^\beta = F_i(Z^1, \ldots, Z^m)$.

Because $\beta > \eta^\delta > \Pi(\lambda(\eta^{\delta-1})) \geq \lambda(\eta^{\delta-1})$, we know that $\{\forall j \in \{1, \ldots, m\}, \forall k \in \{1, \ldots, n\}, \eta^{\delta-1} \leq (S_j^\delta)_k\}$, which implies that $B_k^{\delta-1} \sqsubseteq X_k^{(S_j^\beta)_k}$ by induction hypothesis. Then, $\forall j \in \{1, \ldots, m\}$ we have $B^{\delta-1} \sqsubseteq Z^j$ and, by monotonicity, $B_i^\delta = F_i(\sigma(B_i^{\delta-1})) \sqsubseteq F_i(Z^1, \ldots, Z^m) = X_i^\beta$.

Case 2.3: Assume that β is a limit ordinal and that, for any ordinal α such that $\eta^\delta \leq \alpha < \beta$, we have $B^\delta \sqsubseteq X^\alpha$. Thus, $B^\delta \sqsubseteq \bigsqcup_{\eta^\alpha \leq \alpha < \beta} X^\alpha \sqsubseteq \bigsqcup_{\alpha < \beta} X^\alpha = X^\beta$.

Case 3: Assuming that δ is a limit ordinal and that the lemma holds for all $\delta' < \delta$, we prove by transfinite induction on β that the lemma holds for δ (case 3.1, 3.2, 3.3).

Case 3.1: If $\beta = \eta^\delta$, then η^δ is also a limit ordinal (Lemma 2.9.3.0.6). By induction hypothesis on δ, we have $\forall \gamma < \delta, \beta^\gamma \sqsubseteq X^{\eta^\gamma}$, and so, $B^\delta = \bigsqcup_{\gamma < \delta} B^\gamma \sqsubseteq \bigsqcup_{\gamma < \delta} X^{\eta^\gamma}$. Because $\langle \eta^\gamma : \gamma \in \mathrm{Ord} \rangle$ is strictly increasing $\{\{\gamma < \delta\} \Rightarrow \{\eta^\gamma < \eta^\delta\}\}$ and therefore $\bigsqcup_{\gamma < \delta} X^{\eta^\gamma} \sqsubseteq \bigsqcup_{\eta^\gamma < \eta^\delta} X^{\eta^\gamma} \sqsubseteq \bigsqcup_{\alpha < \eta^\delta} X^\alpha = X^{\eta^\delta}$. By transitivity, $B^\delta \sqsubseteq X^\beta$.

Case 3.2: Assume that β is a successor ordinal and that, for any ordinal α such that $\eta^\delta \leq \alpha < \beta$, we have $B^\delta \sqsubseteq X^\alpha$; we prove that $\forall i \in \{1, \ldots, n\}, B_i^\delta \sqsubseteq X_i^\beta\}$. If $i \neq J^\beta$, then $B_i^\delta \sqsubseteq X_i^{\beta-1} = X_i^\beta$, otherwise $i = J^\beta$ and $X_i^\beta = F_i(Z^1, \ldots, Z^m)$. Because $\eta^\delta < \beta$ and η^δ is a limit ordinal, 2.9.3.0.1.(b) and 2.9.3.0.1.(d) imply that $\{\forall j \in \{1, \ldots, m\}, \forall k \in \{1, \ldots, n\}, \eta^\delta \leq (S_j^\beta)_k < \beta\}$. Therefore, by induction hypothesis, $\{\forall j \in \{1, \ldots, m\}, B_k^\delta \sqsubseteq X_k^{(S_j^\beta)_k} = Z^j\}$. As $\langle B^\delta : \delta \in \mathrm{Ord} \rangle$ is an ascending chain (Theorem 2.5.2.0.2), we obtain by monotonicity $B_i^\delta \sqsubseteq F_i(\sigma(B^\delta)) \sqsubseteq F_i(Z^1, \ldots, Z^m) = X_i^\beta$.

Case 3.3: If β is a limit ordinal and, for any ordinal α such that $\eta^\delta \leq \alpha < \beta$ we have $B^\delta \sqsubseteq X^\alpha$, then $B^\delta \sqsubseteq \bigsqcup_{\eta^\delta \leq \alpha < \beta} X^\alpha \sqsubseteq \bigsqcup_{\alpha < \beta} X^\alpha = X^\beta$.

End of proof.

THEOREM 2.9.3.0.9

An increasing asynchronous iteration with m memories for a monotone function F from $(L^n)^m$ to L^n (where $L(\sqsubseteq, \perp, \top, \sqcup, \sqcap)$ is a complete lattice) starting from a pre-fixpoint D of F is a stationary sequence with limit $luis(F \circ \sigma)(D)$.

Proof: The sequence $\langle B^\delta : \delta \in \text{Ord}\rangle$ is a stationary ascending chain with limit $luis(F \circ \sigma)(D)$ (Theorem 2.5.2.0.2). Therefore, there exists an ordinal $\varepsilon \in \mu(L)$ such that $\forall \gamma \geq \varepsilon, luis(F \circ \sigma)(D) = B^\gamma$. Therefore, $\forall \beta \geq \eta^\varepsilon$ we have $luis(F \circ \sigma)(D) = B^\varepsilon \sqsubseteq X^\beta \sqsubseteq luis(F \circ \sigma)(D)$ (Lemmas 2.9.3.0.9 and 2.9.3.0.2). Note that $\langle X^\delta : \delta \in \text{Ord}\rangle$ is not necessarily an ascending chain.
End of proof.

2.10 BIBLIOGRAPHIC NOTES

The fundamental bibliography to be consulted on ordered sets and lattices is the following: Birkhoff [1967], Bourbaki [1967], Grätzer [1971], and Szász [1971].

We will go back, more at length, on the closure operators on a lattice in Paragraph 4.2 and the corresponding bibliography is to be found in Paragraph 4.4.

Many authors have pointed out that the fixpoint theorem of Knaster [1928] and Tarski [1955] can be proved with some weaker hypotheses than the completeness (see, amongst others, Abian & Brown [1961], Höft & Höft [1976], Pasini [1974], Pelczar [1961], Markowsky [1976], Ward Jr. [1957], Wolk [1957]). Likewise, in our constructive version of Tarski's theorem, we require a hypothesis on the existence of bounds of certain chains and not the less general hypothesis that the lattice is complete. A generalisation is therefore possible to weaken the hypothesis of completeness, but we will not need it. Tarski [1955] (followed by de Maar [1964], Markowsky [1976], Pelczar [1971], Smithson [1973], Wong [1967], ...) also states a theorem on the common fixpoints of a family of commuting monotone operators in a complete lattice. It is also possible to give a constructive version of it (see Cousot & Cousot [1977d]).

Theorem 2.5.6.0.1 on the undecidability of computing the fixpoints of a monotone operator in a complete lattice generalises the theorem [7] of Kam & Ullman [1977] on the non-existence of an algorithm to perform an optimal analysis of the data-flow of a program.

The notion of chaotic iteration comes from the field of numerical analysis and dates back from Southwell [1955] and Ostrowski [1955]. Chazan & Miranker [1969] introduced the notion of chaotic iteration with delays to enable parallel computations in multiprocessors. They showed that a chaotic iteration with delays converges towards the solution of the equation $x = Ax + b$ (where A is a $n \times n$ matrix of reals and b a vector column of reals) if and only if $\rho(|A|) < 1$ (where $|A|$ is the $n \times n$ matrix obtained by replacing each element of A with its absolute value and $\rho(|A|)$ is the spectral radius of $|A|$). These results were extended to some non-linear problems and convergence studies were carried out for some hypotheses other than the hypothesis of contraction. Amongst others, we wish to single out the work of the numerical analysis teams of Grenoble (Abtroun [1977], Charnay [1975], Mahjoub [1977], Robert [1974]), of Besançon (Comte [1976], El Tarazi [1976], Jacquemard [1977], Luong [1975], Miellou [1975a,1975b]), and of the Carnegie-Mellon University (Baudet [1976], Traub [1964]). In particular, the term "asynchronous iteration" comes from Baudet [1976] who generalises the notion of delayed chaotic iterations by eliminating the hypothesis that delays are limited by a fixed maximum delay (Miranker [1977]). Miellou [1977] independently obtained Theorem 2.9.3.0.9 with more restrictive hypotheses than ours. Notably, L is a normal lattice of Banach and F is semi-continuous, which is a hypothesis similar to upper-continuity (2.6.0.1) and not required to provide a constructive version of Tarski's theorem. Also, regarding Definition 2.9.3.0.1 Miellou [1977] adds the hypothesis (using our notations) $\{\forall i \in \{1, \ldots, n\}, \forall j \in \{1, \ldots, m\}, \forall \delta, \delta' \in \mathrm{Ord}, \{\delta \leq \delta'\} \Rightarrow \{(S_j^{\delta})_i \leq (S_j^{\delta'})_i\}\}$. This specific hypothesis for the study of monotone algorithms makes the proof of convergence easier, because it implies that an asynchronous iteration with memory is an ascending chain, with the inconvenient, though, of needing a synchronisation among the processors participating in the computing. As shown in Miranker [1977], an asynchronous algorithm can be superior to an algorithm forbidding to regress while computing due to the waiting time resulting from the synchronisation of processors.

We have not given any criterion for the stopping of asynchronous iterations. It is a programming problem that is handled by Dijkstra [1977] within a very general framework.

The problem of choosing an optimal chaotic strategy, in numerical analysis, dates back to Stein & Rosenberg [1948]. Though being definitely interesting in a practical sense, this problem has not yet received, in numerical analysis, any completely satisfactory answer, apart from some particular cases (for example, Abtroun [1977]) or by interactive methods (Mahjoub [1977]). As for the case of the equation systems we consider in the semantic analysis of programs, the problem is being studied by R. Cousot (see Remark 4.1.2.0.7.(b)).

CHAPTER 3.

BEHAVIOR OF A DISCRETE DYNAMIC SYSTEM, EXACT SEMANTIC ANALYSIS OF PROGRAMS, AND APPLICATIONS

3. BEHAVIOR OF A DISCRETE DYNAMIC SYSTEM, EXACT SEMANTIC ANALYSIS OF PROGRAMS, AND APPLICATIONS

3. BEHAVIOR OF A DISCRETE DYNAMIC SYSTEM, EXACT SEMANTIC ANALYSIS OF PROGRAMS, AND APPLICATIONS

We shall define and study the behavior of discrete dynamic systems in Paragraph 3.1. Discrete dynamic systems provide a very general framework, where we can express and solve problems of semantic analyses of programs, such as the verification of the total or partial correctness of a program, or the inference of the most general invariant at each point of a program. A program is a discrete dynamic system in the sense that it defines a transition relation between memory states before and after the execution of each elementary instruction (if the program is deterministic, this transition relation becomes a transition function). In order to study the behavior of a discrete dynamic system, we need to characterize the set of descendants of the set of all the states that satisfy some given entry specification, or characterize the set of ancestors of the set of states that satisfy some given exit specification. We show that these sets can be computed as solutions of fixpoint equations, or of systems of equations when the set of states of the dynamic system is partitioned.

In Paragraph 3.2, we define the operational semantics of a simple programming language corresponding to sequential iterative programs, with instructions for assignment, conditional branching, and unconditional branching. (We defer recursive programs to Chapter 6, so as not to address all issues at the same time.)

The results of Paragraph 3.1 are applied in Paragraph 3.3, so as to define the forward deductive semantics of the programming language under consideration. A program is mapped to a system of semantic equations by expressing the set of possible states of variables at any point a_j in the program as a function the possible states at the points a_i that are encountered before a_j during an execution of the program.

The least solution of this system defines the set of states of variables at any point in the program, during any execution of this program starting from an initial state which satisfies the given entry specification.

Then, we show in Section 3.4 how the forward deductive semantics can justify Floyd–Naur's method to verify the partial correctness of a program. This method is then extended to proofs of total correctness. Then, we show that performing the symbolic execution of a program in fact consists in solving the system of forward semantic equations using a chaotic iteration strategy.

In Paragraph 3.5, we apply the results of Paragraph 3.1 in order to define a backward deductive semantics for the programming language we consider. The program is mapped to a system of semantic equations, where the set of possible states of variables at any point a_i in the program is expressed as a function of all possible sates of variables at points a_j that follow a_i during an execution of the program. The solutions of this system of backward semantic equations characterize the set of states m of variables for which an execution of the program from a_i in state m terminates in a state which satisfies the given exit specification, terminates without satisfying this exit specification, never terminates, or leads to a semantic error.

Then, in Paragraph 3.5, we will justify the proof method introduced by Hoare to verify the partial correctness of a program, and its extension by Dijkstra to proofs of total correctness. Then, we remark that the forward and backward reasoning techniques are in fact equivalent.

Finally, in Paragraph 3.6, we present results on the combination of forward and backward semantic analyses, which we will use in Chapter 5.

3.1 DISCRETE DYNAMIC SYSTEMS

3.1.1 Notion of discrete dynamic system

DEFINITION 3.1.1.0.1 *Discrete dynamic system*

A *discrete dynamic system* is a 5-tuple $(\mathcal{S}, \tau, \nu_\varepsilon, \nu_\sigma, \nu_\xi)$ defined as follows:

(a) - $\mathcal{S}$ is the *set of states* of the system

(b) - $\tau \in ((\mathcal{S} \times \mathcal{S}) \rightarrow \mathcal{B})$ where $\mathcal{B} = \{\underline{\text{true}}, \underline{\text{false}}\}$ is a *transition relation* between each state and its successors

(c) - $\nu_\varepsilon \in (\mathcal{S} \rightarrow \mathcal{B})$ defines its *entry states*

(d) - $\nu_\sigma \in (\mathcal{S} \rightarrow \mathcal{B})$ defines its *exit states*

(e) - $\nu_\xi \in (\mathcal{S} \rightarrow \mathcal{B})$ defines its *erroneous states*

and which satisfies the following conditions:

(f) - the set of states is not empty: $\{\mathcal{S} \neq \emptyset\}$

(g) - the entry states are exogenous:

$$\{\forall e_1, e_2 \in \mathcal{S}, \{\tau(e_1, e_2)\} \Rightarrow \{\underline{\text{not}}(\nu_\varepsilon(e_2))\}\}$$

(h) - the exit states are stable:

$$\{\forall e_1, e_2 \in \mathcal{S}, \{\nu_\sigma(e_1) \ \underline{\text{and}} \ \tau(e_1, e_2)\} \Rightarrow \{e_1 = e_2\}\}$$

(i) - the sets of entry states, exit states, and erroneous states are pairwise disjoint:

$$\{\forall i, j \in \{\varepsilon, \delta, \xi\}, \{i \neq j\} \Rightarrow \{\forall e \in \mathcal{S}, \underline{\text{not}}(\nu_i(e) \ \underline{\text{and}} \ \nu_j(e))\}\}$$

DEFINITION 3.1.1.0.2

Let $\tau \in ((\mathcal{S} \times \mathcal{S}) \rightarrow \mathcal{B})$. The *inverse* of τ is $\tau^{-1} = \boldsymbol{\lambda}(e_1, e_2) \bullet \tau(e_2, e_1)$. A transition relation (and by extension, a discrete dynamic system $\pi(\mathcal{S}, \tau, \nu_\varepsilon, \nu_\sigma, \nu_\xi)$) is:

(a) - *total* if and only if

$$\{\forall e_1 \in \mathcal{S}, \exists e_2 \in \mathcal{S} : \tau(e_1, e_2)\}$$

(b) - *partial* if and only if it is not total

(c) - *functional* or *deterministic* if and only if:

$$\{\forall e_1, e_2, e_3 \in \mathcal{S}, \{\tau(e_1, e_2) \ \underline{\text{and}}\ \tau(e_1, e_3)\} \Rightarrow \{e_2 = e_3\}\}$$

(d) - *injective* if and only if τ^{-1} is deterministic

(e) - *invertible* if and only if it is total and injective

(f) - *without error recovery* if and only if:

$$\{\forall e_1, e_2 \in \mathcal{S}, \{\nu_\xi(e_1) \ \underline{\text{and}}\ \tau(e_1, e_2)\} \Rightarrow \{\nu_\xi(e_2)\}\}$$

$\llcorner$

The notion of discrete dynamic system is obviously very general. It applies not only to computer systems but also to economic or biological systems, provided that the model of the system to study evolves according to discrete time steps. In particular, discrete dynamic systems can model sequential and parallel programs.

Example 3.1.1.0.3 Sequential programs

A sequential program *à la* FORTRAN π with n global variables $X_1, \ldots, X_n$ with values in $\mathcal{U}_1, \ldots, \mathcal{U}_n$ and α labels (or program points) $a_1, \ldots, a_\alpha$ is a discrete dynamic system $(\mathcal{S}, \tau, \nu_\varepsilon, \nu_\sigma, \nu_\xi)$. A state $e \in \mathcal{S}$ is a pair $\langle m,\ c\rangle$ where $m \in (\mathcal{U}_1 \times \ldots \times \mathcal{U}_n)$ is a memory state, which maps each program variable X_i to a value in $\mathcal{U}_i$ and $c \in$

$\{a_1, \ldots, a_\alpha, \underline{\text{error}}\}$ is a control state that is the label of the instruction of the program that should be executed next, or that indicates an error occurred during the execution of the program. The transition relation maps each state $\langle m,\ c\rangle$ to a unique successor state $\langle m',\ c'\rangle = \tau(\langle m,\ c\rangle)$ resulting from the execution of the atomic instruction at label c in program π. Since τ is a function, the system is deterministic. We observe that the system is not injective in general. For instance, if we consider the program defined by "for all integer x, $\tau(\langle a_\varepsilon,\ x\rangle) = \langle a_\sigma,\ x^2\rangle$", then $\tau(\langle a_\varepsilon,\ 2\rangle) = \langle a_\sigma,\ 4\rangle$ <u>and</u> $\tau(\langle a_\varepsilon,\ -2\rangle) = \langle a_\sigma,\ 4\rangle$ but $\langle a_\varepsilon,\ 2\rangle \neq \langle a_\varepsilon,\ -2\rangle$. When the execution of the labelled instruction c is not defined, for a memory state m (for instance, this is the case when a run-time error is caused by a division by zero), we let $\tau(\langle m,\ c\rangle) = \langle m,\ \underline{\text{error}}\rangle$. The consequence of this choice is that the system is total. In practice, when a run-time error is detected, the execution of the program is stopped. The definition $\tau(\langle m, \underline{\text{error}}\rangle) = \langle m,\ \underline{\text{error}}\rangle$ is a good model for this situation since the system under study is without error recovery. Finally, if we assume that the program has a unique entry point a_ε and a unique exit point a_σ, we define $\nu_\varepsilon = \boldsymbol{\lambda}\,\langle m,\ c\rangle \bullet (c = a_\varepsilon)$, $\nu_\sigma = \boldsymbol{\lambda}\,\langle m,\ c\rangle \bullet (c = a_\sigma)$, and $\nu_\xi = \boldsymbol{\lambda}\,\langle m,\ c\rangle \bullet (c = \underline{\text{error}})$.
End of example.

3.1.2 Specification of the behavior of a discrete dynamic system

From a state $e_0 \in \mathcal{S}$ in the system $\pi(\mathcal{S}, \tau, \nu_\varepsilon, \nu_\sigma, \nu_\xi)$, can evolve into the states e_1, e_2, $\ldots$, e_i, e_{i+1}, $\ldots$ such that $\tau(e_i, e_{i+1})$. The set of the descendant states from e_0 can be expressed as the transitive closure of τ:

<u>DEFINITION</u> 3.1.2.0.1 *Transitive closure of a relation*

Let $\alpha, \beta \in ((\mathcal{S} \times \mathcal{S}) \to \mathcal{B})$ be two relations between elements of $\mathcal{S}$. We note $\alpha \circ \beta$ the *composition* of α and β, which is defined as:

$$\alpha \circ \beta \quad = \quad \boldsymbol{\lambda}(e_1, e_2) \bullet [\exists e_3 \in \mathcal{S} : \alpha(e_1, e_3) \ \underline{\text{and}} \ \beta(e_3, e_2)]$$

For all integers $n \in \omega$, we define α^n by finite induction as follows:

$$\begin{array}{lllll} \alpha^0 & = & eq & = & \boldsymbol{\lambda}(e_1, e_2) \bullet (e_1 = e_2) \\ \alpha^n & = & \alpha^{n-1} \circ \alpha & = & \alpha \circ \alpha^{n-1} \qquad\qquad \text{for all } n \geq 1 \end{array}$$

The *reflexive transitive closure* of α is $\alpha^\star = \underset{n \in \omega}{\underline{\text{OR}}} \, \alpha^n$

The *strict transitive closure* of α is $\alpha^+ = \alpha \circ \alpha^\star = \alpha^\star \circ \alpha = \underset{n \in \omega}{\underline{\text{OR}}} \, \alpha^{n+1}$

$\llcorner$

The specification of a discrete dynamic system $\pi(\mathcal{S}, \tau, \nu_\varepsilon, \nu_\sigma, \nu_\xi)$ consists in the choice of an entry specification $\phi \in (\mathcal{S} \to \mathcal{B})$ and of an exit specification $\psi \in (\mathcal{S} \to \mathcal{B})$. This specification expresses the intent that, from any entry state satisfying the entry specification ϕ, the system π should evolve into an exit state satisfying the exit condition ψ. In the case of a deterministic discrete dynamic system, we call

- *Verification of totally correct behavior* of π for ϕ and ψ the proof that:

$$\{\forall e_1 \in \mathcal{S}, \{\nu_\varepsilon(e_1) \ \underline{\text{and}} \ \phi(e_1)\} \Rightarrow \{\exists e_2 \in \mathcal{S} : \tau^\star(e_1, e_2) \ \underline{\text{and}} \ \nu_\sigma(e_2) \ \underline{\text{and}} \ \psi(e_2)\}\}$$

- *Verification of termination* the proof that any entry state satisfying the entry specification ϕ causes the system to evolve into an exit state:

$$\{\forall e_1 \in \mathcal{S}, \{\nu_\varepsilon(e_1) \ \underline{\text{and}} \ \phi(e_1)\} \Rightarrow \{\exists e_2 \in \mathcal{S} : \tau^\star(e_1, e_2) \ \underline{\text{and}} \ \nu_\sigma(e_2)\}\}$$

- *Verification of partially correct behavior* of π for ϕ and ψ the proof that, when an entry state satisfying ϕ causes the system to evolve into an exit state, then that exit state satisfies the specification ψ:

$$\{\forall e_2 \in \mathcal{S}, \{\exists e_1 \in \mathcal{S} : \nu_\varepsilon(e_1) \ \underline{\text{and}} \ \phi(e_1) \ \underline{\text{and}} \ \tau^\star(e_1, e_2) \ \underline{\text{and}} \ \nu_\sigma(e_2)\} \Rightarrow \{\psi(e_2)\}\}$$

- *Verification of invariance* of $\beta \in (\mathcal{S} \rightarrow \mathcal{B})$ the proof that any state satisfying β causes the system to evolve into a state satisfying β:

$$\{\forall e_1, e_2 \in \mathcal{S}, \{\beta(e_1) \ \underline{\text{and}}\ \tau^\star(e_1, e_2)\} \Rightarrow \{\beta(e_2)\}\}$$

3.1.3 Fixpoint approach to study the behavior of a discrete dynamic system

In order to study the behavior of a discrete dynamic system, we shall try to characterize the *set of the ancestors of the states which satisfy a condition* $\beta \in (\mathcal{S} \rightarrow \mathcal{B})$, which amounts to determine:

$$\boldsymbol{\lambda}\, e_1 \bullet \{\exists e_2 \in \mathcal{S} : \tau^\star(e_1, e_2) \ \underline{\text{and}}\ \beta(e_2)\} \quad = \quad wp(\tau^\star)(\beta)$$

where:

DEFINITION 3.1.3.0.1

$$\begin{array}{lcl} wp & \in & (((\mathcal{S} \times \mathcal{S}) \rightarrow \mathcal{B}) \rightarrow ((\mathcal{S} \rightarrow \mathcal{B}) \rightarrow (\mathcal{S} \rightarrow \mathcal{B}))) \\ & = & \boldsymbol{\lambda}\, \theta \bullet \{\boldsymbol{\lambda}\, \beta \bullet [\boldsymbol{\lambda}\, e_1 \bullet (\exists e_2 \in \mathcal{S} : \theta(e_1, e_2) \ \underline{\text{and}}\ \beta(e_2))]\} \end{array}$$

For instance, verifying the total correctness of $\pi(\mathcal{S}, \tau, \nu_\varepsilon, \nu_\sigma, \nu_\xi)$ for ϕ and ψ corresponds to proving that:

$$\{\nu_\varepsilon \ \underline{\text{and}}\ \phi\} \quad \Rightarrow \quad \{wp(\tau^\star)(\nu_\sigma \ \underline{\text{and}}\ \psi)\}$$

In the same way, we shall try to characterize the *set of descendants of the states which satisfy condition* $\beta \in (\mathcal{S} \rightarrow \mathcal{B})$, which amounts to determine:

$$\boldsymbol{\lambda}\, e_2 \bullet \{\exists e_1 \in \mathcal{S} : \beta(e_1) \ \underline{\text{and}}\ \tau^\star(e_1, e_2)\} \quad = \quad sp(\tau^\star)(\beta)$$

where:

DEFINITION 3.1.3.0.2

$$\begin{aligned} sp &\in (((\mathcal{S} \times \mathcal{S}) \to \mathcal{B}) \to ((\mathcal{S} \to \mathcal{B}) \to (\mathcal{S} \to \mathcal{B}))) \\ &= \boldsymbol{\lambda}\, \theta \bullet \{\boldsymbol{\lambda}\, \beta \bullet [\boldsymbol{\lambda}\, e_2 \bullet (\exists e_1 \in \mathcal{S} : \beta(e_1) \ \underline{\text{and}}\ \theta(e_1, e_2))]\} \end{aligned}$$

For instance, verifying the partial correctness of π for ϕ and ψ consists in proving that $\{sp(\tau^\star)(\nu_\varepsilon\ \underline{\text{and}}\ \phi)\ \underline{\text{and}}\ \nu_\sigma\} \Rightarrow \{\psi\}$. Similarly, verifying the invariance of β consists in proving that $\beta \Rightarrow \underline{\text{not}}\,(wp(\tau^\star)(\underline{\text{not}}\,(\beta)))$, that is, $sp(\tau^\star)(\beta) \Rightarrow \beta$.

Since $\tau^\star = eq\ \underline{\text{or}}\ \tau^\star \circ \tau = eq\ \underline{\text{or}}\ \tau \circ \tau^\star$, we construct $wp(\tau^\star)$ and $sp(\tau^\star)$ as fixpoints of an equation.

THEOREM 3.1.3.0.3

(a) - $((\mathcal{S}\times\mathcal{S}) \to \mathcal{B})(\Rightarrow, \boldsymbol{\lambda}\,(e_1, e_2) \bullet \underline{\text{false}}, \boldsymbol{\lambda}\,(e_1, e_2) \bullet \underline{\text{true}}, \underline{\text{OR}}, \underline{\text{AND}}, \underline{\text{not}}\,)$ is a complete boolean lattice

(b) - Let $a, b \in ((\mathcal{S} \times \mathcal{S}) \to \mathcal{B})$, then $\boldsymbol{\lambda}\, \alpha \bullet [a\ \underline{\text{or}}\ b \circ \alpha]$ and $\boldsymbol{\lambda}\, \alpha \bullet [a\ \underline{\text{or}}\ \alpha \circ b]$ are complete join-morphisms

(c) - Let $\tau \in ((\mathcal{S} \times \mathcal{S}) \to \mathcal{B})$ and eq be the equality relation

$$\tau^\star \quad = \quad lfp(\boldsymbol{\lambda}\, \alpha \bullet [eq\ \underline{\text{or}}\ \alpha \circ \tau]) \quad = \quad lfp(\boldsymbol{\lambda}\, \alpha \bullet [eq\ \underline{\text{or}}\ \tau \circ \alpha])$$

The following proposition shows that the study can be performed either using sp or wp, using a transition relation or its inverse:

PROPOSITION 3.1.3.0.4

(a) - $\boldsymbol{\lambda}\,\theta \bullet \theta^{-1}$ is a complete automorphism on $((\mathcal{S} \times \mathcal{S}) \to \mathcal{B})$

(b) - $\forall \alpha, \beta \in ((\mathcal{S} \times \mathcal{S}) \to \mathcal{B}), (\alpha \circ \beta)^{-1} = \beta^{-1} \circ \alpha^{-1}$

(c) - $\forall \tau \in ((\mathcal{S} \times \mathcal{S}) \to \mathcal{B}), \forall n \in \omega, (\tau^n)^{-1} = (\tau^{-1})^n$

(d) - $\forall \tau \in ((\mathcal{S} \times \mathcal{S}) \to \mathcal{B}), (\tau^{\star})^{-1} = (\tau^{-1})^{\star}$ and $(\tau^{+})^{-1} = (\tau^{-1})^{+}$

(e) - $\forall \theta \in ((\mathcal{S} \times \mathcal{S}) \to \mathcal{B}), \mathit{sp}(\theta) = \mathit{wp}(\theta^{-1})$ and $\mathit{wp}(\theta) = \mathit{sp}(\theta^{-1})$

LEMMA 3.1.3.0.5

For all $\theta \in ((\mathcal{S} \times \mathcal{S}) \to \mathcal{B})$ and $\beta \in (\mathcal{S} \to \mathcal{B})$ we have:

(a) - $\mathit{wp}(\theta), \boldsymbol{\lambda}\,\theta \bullet (\mathit{wp}(\theta)(\beta)), \mathit{sp}(\theta)$, and $\boldsymbol{\lambda}\,\theta \bullet (\mathit{sp}(\theta)(\beta))$ are $\perp$-strict (i.e., $f(\perp) = \perp$) complete join-morphisms

(b) - If θ is deterministic, then $\mathit{wp}(\theta)$ is a complete meet-morphism. If θ is injective, then $\mathit{sp}(\theta)$ is a complete meet-morphism

Proof:

(a) follows from the fact that, for every family $\langle \beta_i : i \in I \rangle$ of elements in $(\mathcal{S} \to \mathcal{B})$ and every $\beta \in (\mathcal{S} \to \mathcal{B})$, we have $\{\exists e \in \mathcal{S} : \beta(e) \ \underline{\text{and}}\ (\underset{i \in I}{\underline{\text{OR}}}\, \beta_i(e))\} = \underset{i \in I}{\underline{\text{OR}}}\{\exists e \in \mathcal{S} : \beta(e) \ \underline{\text{and}}\ \beta_i(e)\}$.

(b) When θ is deterministic, there exists a function f from $\mathcal{S}$ to $\mathcal{S}$ such that $\{\forall e_1, e_2 \in \mathcal{S}, \{\theta(e_1, e_2)\} \Leftrightarrow \{f(e_1) = e_2\}\}$. For a fixed e_1 and when $\{e \in \mathcal{S} : \theta(e_1, e)\}$ is not empty, we have $\mathit{wp}(\theta)(\underset{i \in I}{\underline{\text{AND}}}\, \beta_i)(e_1) = \{\exists e_2 \in \mathcal{S} : \theta(e_1, e_2) \ \underline{\text{and}}\ (\underset{i \in I}{\underline{\text{AND}}}\, \beta_i(e_2))\} = \underset{i \in I}{\underline{\text{AND}}}\, \beta_i(f(e_1)) = \underset{i \in I}{\underline{\text{AND}}}\{\exists e_2 \in \mathcal{S} : \theta(e_1, e_2) \ \underline{\text{and}}\ \beta_i(e_2)\} = \underset{i \in I}{\underline{\text{AND}}}\, \mathit{wp}(\theta)(\beta_i)(e_1)$. Following 3.1.1.0.2.(d) and 3.1.3.0.4, if θ is injective, then θ^{-1} is deterministic and $\mathit{sp}(\theta) = \mathit{wp}(\theta^{-1})$ is a complete meet-morphism.

End of proof.

THEOREM 3.1.3.0.6

For all $a, b \in ((\mathcal{S} \times \mathcal{S}) \rightarrow \mathcal{B})$ and $\beta \in (\mathcal{S} \rightarrow \mathcal{B})$ we have:

- $\mathit{wp}(\mathit{lfp}(\boldsymbol{\lambda}\, \alpha \bullet [a \;\underline{\text{or}}\; b \circ \alpha]))(\beta)$
$$\begin{aligned} &= \mathit{lfp}(\boldsymbol{\lambda}\, \alpha \bullet [\mathit{wp}(a)(\beta) \;\underline{\text{or}}\; \mathit{wp}(b)(\alpha)]) \\ &= \underset{n \in \omega}{\underline{\text{OR}}}\, \mathit{wp}(b^n)(\mathit{wp}(a)(\beta)) \end{aligned}$$

- $\mathit{sp}(\mathit{lfp}(\boldsymbol{\lambda}\, \alpha \bullet [a \;\underline{\text{or}}\; \alpha \circ b]))(\beta)$
$$\begin{aligned} &= \mathit{lfp}(\boldsymbol{\lambda}\, \alpha \bullet [\mathit{sp}(a)(\beta) \;\underline{\text{or}}\; \mathit{sp}(b)(\alpha)]) \\ &= \underset{n \in \omega}{\underline{\text{OR}}}\, \mathit{sp}(b^n)(\mathit{sp}(a)(\beta)) \end{aligned}$$

$\llcorner$

Proof: Let $h = \boldsymbol{\lambda}\, \theta \bullet [\mathit{wp}(\theta)(\beta)]$, $f = \boldsymbol{\lambda}\, \alpha \bullet [a \;\underline{\text{or}}\; b \circ \alpha]$ and $g = \boldsymbol{\lambda}\, \alpha \bullet [\mathit{wp}(a)(\beta) \;\underline{\text{or}}\; \mathit{wp}(b)(\alpha)]$. Let us prove that $h \circ f = g \circ h$. Indeed, let $\alpha \in ((\mathcal{S} \times \mathcal{S}) \rightarrow \mathcal{B})$, then:

$$\begin{aligned} h \circ f(\alpha) &= \mathit{wp}(a \;\underline{\text{or}}\; b \circ \alpha)(\beta) \\ &= \boldsymbol{\lambda}\, e_1 \bullet [\exists e_2 \in \mathcal{S} : (a(e_1, e_2) \;\underline{\text{or}}\; (b \circ \alpha)(e_1, e_2)) \;\underline{\text{and}}\; \beta(e_2)] \\ &= \boldsymbol{\lambda}\, e_1 \bullet [\exists e_2 \in \mathcal{S} : a(e_1, e_2) \;\underline{\text{and}}\; \beta(e_2)] \;\underline{\text{or}} \\ &\ \boldsymbol{\lambda}\, e_1 \bullet [\exists e_2, e_3 \in \mathcal{S} : b(e_1, e_3) \;\underline{\text{and}}\; \alpha(e_3, e_2) \;\underline{\text{and}}\; \beta(e_2)] \\ &= \mathit{wp}(a)(\beta) \;\underline{\text{or}}\; \boldsymbol{\lambda}\, e_1 \bullet [\exists e_3 \in \mathcal{S} : b(e_1, e_3) \;\underline{\text{or}}\; \mathit{wp}(\alpha)(\beta)(e_3)] \\ &= \mathit{wp}(a)(\beta) \;\underline{\text{or}}\; \mathit{wp}(b)(\mathit{wp}(\alpha)(\beta)) \\ &= g \circ h(\alpha) \end{aligned}$$

f, g, and h are complete join-morphisms, therefore, Proposition 2.5.2.0.8 implies that $h(\mathit{lfp}(f)) = h(\mathit{luis}(f)(\boldsymbol{\lambda}\, (e_1, e_2) \bullet [\underline{\text{false}}])) = \mathit{luis}(g)(h(\boldsymbol{\lambda}\, (e_1, e_2) \bullet [\underline{\text{false}}])) = \mathit{luis}(g)(\boldsymbol{\lambda}\, (e_1, e_2) \bullet [\underline{\text{false}}]) = \mathit{lfp}(g)$.

Let $\langle X^n : n \in \omega \rangle$ be the increasing iteration defined by g and starting from the infimum $\boldsymbol{\lambda}\, e \bullet \underline{\text{false}}$. We have $X^0 = \boldsymbol{\lambda}\, e \bullet \underline{\text{false}}$ and $X^1 = \mathit{wp}(a)(\beta)$ since $\mathit{wp}(b)$ is $\bot$-strict. Let us assume that $X^n = \overset{n-1}{\underset{i=0}{\underline{\text{OR}}}}\, \mathit{wp}(b^i)(\mathit{wp}(a)(\beta))$. Then, $X^{n+1} = g(X^n) = \mathit{wp}(a)(\beta) \;\underline{\text{or}}\; \overset{n-1}{\underset{i=0}{\underline{\text{OR}}}}\, \mathit{wp}(b)(\mathit{wp}(b^i)(\mathit{wp}(a)(\beta))) = \overset{n}{\underset{i=0}{\underline{\text{OR}}}}\, \mathit{wp}(b^i)(\mathit{wp}(a)(\beta))$ since $\mathit{wp}(b)(\mathit{wp}(a)(\beta)) = \mathit{wp}(b \circ$

$a)(\beta)$. By finite induction, we have found the general term of the iteration. By passing to the limit, we obtain $\mathit{lfp}(g) = X^{\omega} = \underset{n\in\omega}{\underline{\mathrm{OR}}}(\overset{n-1}{\underset{i=0}{\underline{\mathrm{OR}}}}\, \mathit{wp}(b^i)(\mathit{wp}(a)(\beta))) = \underset{n\in\omega}{\underline{\mathrm{OR}}}\, \mathit{wp}(b^i)(\mathit{wp}(a)(\beta))$.

Following from Proposition 3.1.3.0.4, a symmetric proof applies to *sp*.

End of proof.

Theorems 3.1.3.0.3 and 3.1.3.0.6 imply:

COROLLARY 3.1.3.0.7

Let $\tau \in ((\mathcal{S} \times \mathcal{S}) \rightarrow \mathcal{B})$, then:

- $\mathit{wp}(\tau^{\star}) \quad = \quad \boldsymbol{\lambda}\,\beta \bullet\ \mathit{lfp}(\boldsymbol{\lambda}\,\alpha \bullet [\beta\ \underline{\mathrm{or}}\ \mathit{wp}(\tau)(\alpha)]) \quad = \quad \underset{n\in\omega}{\underline{\mathrm{OR}}}\, \mathit{wp}(\tau^n)$
- $\mathit{sp}(\tau^{\star}) \quad = \quad \boldsymbol{\lambda}\,\beta \bullet\ \mathit{lfp}(\boldsymbol{\lambda}\,\alpha \bullet [\beta\ \underline{\mathrm{or}}\ \mathit{sp}(\tau)(\alpha)]) \quad = \quad \underset{n\in\omega}{\underline{\mathrm{OR}}}\, \mathit{sp}(\tau^n)$

While computing $\mathit{wp}(\mathit{lfp}(\boldsymbol{\lambda}\,\alpha \bullet [a\ \underline{\mathrm{or}}\ b \circ \alpha]))(\beta)$ is easy thanks to Proposition 2.5.2.0.8, computing $\mathit{wp}(\mathit{lfp}(\boldsymbol{\lambda}\,\alpha \bullet [a\ \underline{\mathrm{or}}\ \alpha \circ b]))(\beta)$ is not straightforward, since $\boldsymbol{\lambda}\,\theta \bullet [\mathit{wp}(\theta)(\beta)] \circ \boldsymbol{\lambda}\,\alpha \bullet [a\ \underline{\mathrm{or}}\ \alpha \circ b] = \boldsymbol{\lambda}\,\alpha \bullet [\mathit{wp}(a)(\beta)]\ \underline{\mathrm{or}}\ \mathit{wp}(\alpha)(\mathit{wp}(b)(\beta))]$ cannot be written as the composition of a function with $\boldsymbol{\lambda}\,\alpha \bullet\ \mathit{wp}(\alpha)(\beta)$. As a consequence, it is easier to express *wp* using *sp*, and then to apply Theorem 3.1.3.0.7. Indeed:

$$\begin{array}{rcl} \mathit{wp} & = & \boldsymbol{\lambda}\,\theta \bullet \{\boldsymbol{\lambda}\,\beta \bullet [\boldsymbol{\lambda}\,\bar{e} \bullet (\exists e_2 \in \mathcal{S} : \theta(\bar{e}, e_2)\ \underline{\mathrm{and}}\ \beta(e_2))]\} \\ & = & \boldsymbol{\lambda}\,\theta \bullet \{\boldsymbol{\lambda}\,\beta \bullet [\boldsymbol{\lambda}\,\bar{e} \bullet (\exists e_2 \in \mathcal{S} : [\exists e \in \mathcal{S} : (e = \bar{e})\ \underline{\mathrm{and}}\ \theta(e, e_2)]\ \underline{\mathrm{and}}\ \beta(e_2)])]\} \\ & = & \boldsymbol{\lambda}\,\theta \bullet \{\boldsymbol{\lambda}\,\beta \bullet [\boldsymbol{\lambda}\,\bar{e} \bullet (\exists e_2 \in \mathcal{S} : \mathit{sp}(\theta)(\boldsymbol{\lambda}\,e \bullet [e = \bar{e}])(e_2)\ \underline{\mathrm{and}}\ \beta(e_2))]\} \end{array}$$

Therefore, 3.1.3.0.7 and 3.1.3.0.4 imply:

COROLLARY 3.1.3.0.8

Let $\tau \in ((\mathcal{S} \times \mathcal{S}) \rightarrow \mathcal{B})$, then:

- $\mathit{wp}(\tau^{\star}) \quad = \quad \boldsymbol{\lambda}\,\beta \bullet \{\boldsymbol{\lambda}\,\bar{e} \bullet [\exists e_2 \in \mathcal{S} : \beta(e_2)\ \underline{\mathrm{and}}\ \mathit{lfp}(\boldsymbol{\lambda}\,\alpha \bullet [\boldsymbol{\lambda}\,e \bullet (e = \bar{e})\ \underline{\mathrm{or}}\ \mathit{sp}(\tau)(\alpha)])(e_2)]\}$
- $\mathit{sp}(\tau^{\star}) \quad = \quad \boldsymbol{\lambda}\,\beta \bullet \{\boldsymbol{\lambda}\,\bar{e} \bullet [\exists e_1 \in \mathcal{S} : \beta(e_1)\ \underline{\mathrm{and}}\ \mathit{lfp}(\boldsymbol{\lambda}\,\alpha \bullet [\boldsymbol{\lambda}\,e \bullet (e = \bar{e})\ \underline{\mathrm{or}}\ \mathit{wp}(\tau)(\alpha)])(e_1)]\}$

This result shows that we can study a discrete dynamic system by solving either a "forward" equation of the form $\alpha = \beta \; \underline{\text{or}} \; sp(\tau)(\alpha)$ or a "backward" equation of the form $\alpha = \beta \; \underline{\text{or}} \; wp(\tau)(\alpha)$.

3.1.4 Discrete dynamic systems with partitioned set of states

We have reduced the study of a discrete dynamic system $\pi(\mathcal{S}, \tau, \nu_\varepsilon, \nu_\sigma, \nu_\xi)$ to the problem of solving an equation associated with the system. When the set of states $\mathcal{S}$ is partitioned, the partition on $\mathcal{S}$ induces a straightforward decomposition of $(\mathcal{S} \to \mathcal{B})$ and then Proposition 2.8.0.1 shows how to map an equation to a system of equations such that the set of solutions of the system is isomorphic to the set of solutions of the equation. This decomposition will allow us to use various techniques for solving systems of monotone equations in a complete lattice which were studied in Paragraphs 2.8 and 2.9.

DEFINITION 3.1.4.0.1 *Partitioned discrete dynamic system*

A partitioned discrete dynamic system is a tuple $(\mathcal{S}, \tau, \nu_1, \ldots, \nu_n, \varepsilon, \sigma, \xi)$ such that:

(a) - $\{n \geq 1\} \; \underline{\text{and}} \; \{\forall i \in [1, n], \nu_i \in (\mathcal{S} \to \mathcal{B})\}$

(b) - $\{\varepsilon, \sigma, \xi \in [1, n]\} \; \underline{\text{and}} \; \{(\mathcal{S}, \tau, \nu_\varepsilon, \nu_\sigma, \nu_\xi) \text{ is a discrete dynamic system}\}$

(c) - $\{\forall e \in \mathcal{S}, \exists i \in [1, n] : \nu_i(e)\}$

(d) - $\{\forall i, j \in [1, n], \forall e \in \mathcal{S}, \{\nu_i(e) \; \underline{\text{and}} \; \nu_j(e)\} \Rightarrow \{i = j\}\}$

For all $i \in [1, n]$, we define:

$$\begin{aligned} \sigma_i &\in ((\mathcal{S} \to \mathcal{B}) \to (\mathcal{S} \to \mathcal{B})) \\ &= \boldsymbol{\lambda} \beta \bullet [\nu_i \; \underline{\text{and}} \; \beta] \end{aligned}$$

$$\sigma \in ((\mathcal{S} \to \mathcal{B}) \to (\sigma_1(\mathcal{S} \to \mathcal{B}) \times \ldots \times \sigma_n(\mathcal{S} \to \mathcal{B})))$$

$$= \boldsymbol{\lambda}\beta \bullet [\sigma_1(\beta), \ldots, \sigma_n(\beta)] = \boldsymbol{\lambda}\beta \bullet \prod_{i=1}^{n} \sigma_i(\beta)$$

$$\begin{array}{rcl} \sigma^{-1} & \in & ((\sigma_1(\mathcal{S} \to \mathcal{B}) \times \ldots \times \sigma_n(\mathcal{S} \to \mathcal{B})) \to (\mathcal{S} \to \mathcal{B})) \\ & = & \boldsymbol{\lambda}\beta \bullet [\underline{\mathrm{OR}}_{i=1}^{n} \beta_i] \end{array}$$

From 3.1.4.0.1.(c), $\sigma^{-1} \circ \sigma = \boldsymbol{\lambda}\beta \bullet [\underline{\mathrm{OR}}_{i=1}^{n} \nu_i \,\underline{\mathrm{and}}\, \beta] = \boldsymbol{\lambda}\beta \bullet \beta$ and, from 3.1.4.0.1.(d), $\sigma_i \circ \sigma^{-1} = \boldsymbol{\lambda}\beta \bullet \sigma_i \circ \sigma^{-1}(\beta_1 \;\underline{\mathrm{and}}\; \nu_1, \ldots, \beta_n \;\underline{\mathrm{and}}\; \nu_n) = \boldsymbol{\lambda}\beta \bullet [\nu_i \;\underline{\mathrm{and}}\; (\underline{\mathrm{OR}}_{j=1}^{n} \beta_j \;\underline{\mathrm{and}}\; \nu_j)] = \boldsymbol{\lambda}\beta \bullet \beta_i$, so that σ is a bijection. The distributivity properties $\beta \;\underline{\mathrm{and}}\; (\underline{\mathrm{OR}}_{j \in J} \beta_j) = \underline{\mathrm{OR}}_{j \in J}(\beta \;\underline{\mathrm{and}}\; \beta_j)$ and $\beta \;\underline{\mathrm{and}}\; (\underline{\mathrm{AND}}_{j \in J} \beta_j) = \underline{\mathrm{AND}}_{j \in J}(\beta \;\underline{\mathrm{and}}\; \beta_j)$ imply that σ is a complete morphism from $(\mathcal{S} \to \mathcal{B})$ to $\prod_{i=1}^{n} \sigma_i(\mathcal{S} \to \mathcal{B})$. Therefore, Propositions 2.5.2.0.8, 2.5.3.0.3, 2.5.3.0.2, and 2.5.5.0.1 show that the sets of pre-fixpoints, post-fixpoints, and fixpoints of $F \in ((\mathcal{S} \to \mathcal{B}) \to (\mathcal{S} \to \mathcal{B}))$ are completely isomorphic by σ to the sets of pre-solutions, post-solutions, and solutions of the system of equations:

$$\left\{ \begin{array}{rcl} x_i & = & \sigma_i \circ F \circ \sigma^{-1}(x_1, \ldots, x_n) \\ i & = & 1, \ldots, n \end{array} \right.$$

In particular, for $F = \boldsymbol{\lambda}\alpha \bullet [\beta \;\underline{\mathrm{or}}\; sp(\tau)(\alpha)]$ and $F = \boldsymbol{\lambda}\alpha \bullet [\beta \;\underline{\mathrm{or}}\; wp(\tau)(\alpha)]$, we get:

PROPOSITION 3.1.4.0.2

Let $\pi(\mathcal{S}, \tau, \nu_1, \ldots, \nu_n, \varepsilon, \sigma, \xi)$ be a partitioned discrete dynamic system. Then, $\forall i \in [1, n], \sigma_i \circ \boldsymbol{\lambda}\alpha \bullet [\beta \;\underline{\mathrm{or}}\; sp(\tau)(\alpha)] \circ \sigma^{-1}$ is equal to:

$$\boldsymbol{\lambda}(\alpha_1, \ldots, \alpha_n) \bullet [(\nu_i \;\underline{\mathrm{and}}\; \beta) \;\underline{\mathrm{or}}\; (\underline{\mathrm{OR}}_{j \in \underline{\mathrm{pred}}_{\tau}(i)} \; sp(\tau_{ji})(\alpha_j))]$$

where:

$\forall i, j \in [1, n], \forall \theta \in ((\mathcal{S} \times \mathcal{S}) \to \mathcal{B}), \theta_{ij} = \boldsymbol{\lambda}(e_1, e_2) \bullet [\nu_i(e_1) \;\underline{\mathrm{and}}\; \theta(e_1, e_2) \;\underline{\mathrm{and}}\; \nu_j(e_2)]$

$$\begin{array}{rcl} \underline{\mathrm{pred}}_{\tau} & \in & [1, n] \to 2^{[1,n]} \\ & = & \boldsymbol{\lambda} i \bullet \{j \in [1, n] : (\exists e_1, e_2 \in \mathcal{S} : \tau_{ji}(e_1, e_2))\} \end{array}$$

Proof:

$\sigma_i(\beta \ \underline{\text{or}}\ sp(\tau)(\sigma^{-1}(\alpha_1,\ldots,\alpha_n)))$ where $(\forall i \in [1,n], \alpha_i \in \sigma_i(\mathcal{S} \rightarrow \mathcal{B}))$

$$\begin{aligned}
&= \sigma_i(\beta)\ \underline{\text{or}}\ \sigma_i(sp(\tau)(\sigma^{-1}(\nu_1\ \underline{\text{and}}\ \alpha_1,\ldots,\nu_n\ \underline{\text{and}}\ \alpha_n)))\\
&= \sigma_i(\beta)\ \underline{\text{or}}\ \boldsymbol{\lambda}\, e_2 \bullet \{\nu_i(e_2)\ \underline{\text{and}}\ sp(\tau)(\mathop{\underline{\text{OR}}}_{j=1}^{n}(\nu_j\ \underline{\text{and}}\ \alpha_j))(e_2)\}\\
&= \sigma_i(\beta)\ \underline{\text{or}}\ \boldsymbol{\lambda}\, e_2 \bullet \{\nu_i(e_2)\ \underline{\text{and}}\ \mathop{\underline{\text{OR}}}_{j=1}^{n}(sp(\tau)(\nu_j\ \underline{\text{and}}\ \alpha_j)(e_2))\} \text{ (from 3.1.3.0.5.(a))}\\
&= \sigma_i(\beta)\ \underline{\text{or}}\ \mathop{\underline{\text{OR}}}_{j=1}^{n} \boldsymbol{\lambda}\, e_2 \bullet \{\exists e_1 \in \mathcal{S} : \alpha_j(e_1)\ \underline{\text{and}}\ \nu_j(e_1)\ \underline{\text{and}}\ \tau(e_1,e_2)\ \underline{\text{and}}\ \nu_i(e_2)\}\\
&\quad \text{(from 3.1.2.0.2)}\\
&= \sigma_i(\beta)\ \underline{\text{or}}\ \mathop{\underline{\text{OR}}}_{j=1}^{n} \boldsymbol{\lambda}\, e_2 \bullet \{\exists e_1 \in \mathcal{S} : \alpha_j(e_1)\ \underline{\text{and}}\ \tau_{ji}(e_1,e_2)\}\\
&= (\nu_i\ \underline{\text{and}}\ \beta)\ \underline{\text{or}} \mathop{\underline{\text{OR}}}_{j \in \underline{\text{pred}}_\tau(i)} sp(\tau_{ji})(\alpha_j) \text{ since } \{j \notin \underline{\text{pred}}_\tau(i)\} \Rightarrow \{\underline{\text{not}}(\tau_{ji}(e_1,e_2))\}
\end{aligned}$$

End of proof.

Since $(\tau_{ij})^{-1} = (\tau^{-1})_{ji}$ and using Proposition 3.1.3.0.4, we immediately obtain the following proposition:

PROPOSITION 3.1.4.0.3

Let $\pi(\mathcal{S},\tau,\nu_1,\ldots,\nu_n,\varepsilon,\sigma,\xi)$ be a *partitioned discrete dynamic system*. Then, $\forall i \in [1,n], \sigma_i \circ \boldsymbol{\lambda}\, \alpha \bullet [\beta\ \underline{\text{or}}\ wp(\tau)(\alpha)] \circ \sigma^{-1}$ is equal to:

$$\boldsymbol{\lambda}\,(\alpha_1,\ldots,\alpha_n) \bullet [(\nu_i\ \underline{\text{and}}\ \beta)\ \underline{\text{or}}\ (\mathop{\underline{\text{OR}}}_{j \in \underline{\text{succ}}_\tau(i)} wp(\tau_{ij})(\alpha_j))]$$

where

$$\begin{aligned}
\underline{\text{succ}}_\tau &\in [1,n] \rightarrow 2^{[1,n]}\\
&= \boldsymbol{\lambda}\, i \bullet \{j \in [1,n] : (\exists e_1, e_2 \in \mathcal{S} : \tau_{ij}(e_1,e_2))\}
\end{aligned}$$

3.1.5 Properties of partitioned discrete dynamic systems

From Definitions 3.1.1.0.2.(c) and (d) and Proposition 3.1.3.0.4, the study of $\mathit{wp}(\tau^\star)$ when τ is deterministic and the study of $\mathit{sp}(\tau^\star)$ when τ is injective are symmetric. However, we shall present our results only for deterministic systems, since injective programs are not common. We have seen that $\mathit{wp}(\tau^\star) = \boldsymbol{\lambda}\,\beta \bullet\ \mathit{lfp}(\boldsymbol{\lambda}\,\alpha \bullet [\beta\ \underline{\text{or}}\ \mathit{wp}(\tau)(\alpha)])$. We will now focus on the properties of the other fixpoints of $\boldsymbol{\lambda}\,\alpha \bullet [\beta\ \underline{\text{or}}\ \mathit{wp}(\tau)(\alpha)]$.

<u>PROPOSITION</u> 3.1.5.0.1

Let $\pi(\mathcal{S}, \tau, \nu_\varepsilon, \nu_\sigma, \nu_\xi)$ be a deterministic discrete dynamic system, $\beta \in (\mathcal{S} \to \mathcal{B})$ such that $\{\forall e_1, e_2 \in \mathcal{S}, \{\beta(e_1)\ \underline{\text{and}}\ \tau(e_1, e_2)\} \Rightarrow \{e_1 = e_2\}\}$ and $\alpha \in (\mathcal{S} \to \mathcal{B})$ be a post-fixpoint of $\boldsymbol{\lambda}\,x \bullet [\beta\ \underline{\text{or}}\ \mathit{wp}(\tau)(x)]$. Then, $\mathit{sp}(\tau^\star)(\alpha) \Rightarrow \alpha$.

Proof: First, let us prove that $\{\mathit{sp}(\tau)(\alpha) \Rightarrow \alpha\}$. For any e in $\mathcal{S}$, we have $\mathit{sp}(\tau)(\alpha)(e) = \{\exists e_1 \in \mathcal{S} : \alpha(e_1)\ \underline{\text{and}}\ \tau(e_1, e)\} \Rightarrow \{\exists e_1 \in \mathcal{S} : \alpha(e_1)\ \underline{\text{and}}\ [\beta(e_1)\ \underline{\text{or}}\ \mathit{wp}(\tau)(\alpha)(e_1)]\ \underline{\text{and}}\ \tau(e_1, e)\}$.
Since $\{\{\beta(e_1)\ \underline{\text{and}}\ \tau(e_1, e)\} \Rightarrow \{e_1 = e\}\}$ and since π is deterministic $\{\{\tau(e_1, e_2)\ \underline{\text{and}}\ \tau(e_1, e)\} \Rightarrow \{e_2 = e\}\}$, we get $\mathit{sp}(\tau)(\alpha)(e) \Rightarrow \{\exists e_1, e_2 \in \mathcal{S} : [\alpha(e_1)\ \underline{\text{and}}\ (e_1 = e)]\ \underline{\text{or}}\ [\alpha(e_2)\ \underline{\text{and}}\ (e_2 = e)]\} \Rightarrow \alpha(e)$, therefore, $(\alpha\ \underline{\text{or}}\ \mathit{sp}(\tau)(\alpha)) \Rightarrow \alpha$. Since α is a post-fixpoint of $\boldsymbol{\lambda}\,x \bullet [\alpha\ \underline{\text{or}}\ \mathit{sp}(\tau)(x)]$, Theorems 3.1.3.0.5 and 2.5.3.0.2 imply that $\mathit{sp}(\tau^\star)(\alpha) = \mathit{lfp}(\boldsymbol{\lambda}\,x \bullet [\alpha\ \underline{\text{or}}\ \mathit{sp}(\tau)(x)]) \Rightarrow \alpha$.
End of proof.

<u>THEOREM</u> 3.1.5.0.2

Let $\tau \in ((\mathcal{S} \times \mathcal{S}) \to \mathcal{B})$ be a total and deterministic transition relation and $\beta \in (\mathcal{S} \to \mathcal{B})$, then:

$$\underline{\text{not}}\,[\mathit{wp}(\tau^\star)(\beta)] \quad = \quad \mathit{gfp}(\boldsymbol{\lambda}\,\alpha \bullet [\underline{\text{not}}\,(\beta)\ \underline{\text{and}}\ \mathit{wp}(\tau)(\alpha)])$$

Proof: From Theorem 3.1.3.0.7, we know that $\underline{\text{not}}\,[\mathit{wp}(\tau^\star)(\beta)] = \underline{\text{not}}\,[\mathit{lfp}(\boldsymbol{\lambda}\,\alpha \bullet [\beta\ \underline{\text{or}}\ \mathit{wp}(\tau)(\alpha)])] = \underline{\text{not}}\,[\mathit{lfp}(\boldsymbol{\lambda}\,\alpha \bullet [\underline{\text{not}}\,(\underline{\text{not}}\,(\beta)\,\underline{\text{and}}\,\underline{\text{not}}\,(\mathit{wp}(\tau)(\underline{\text{not}}\,(\underline{\text{not}}\,(\alpha)))))])]$ which is equal to $\mathit{gfp}(\boldsymbol{\lambda}\,\alpha\bullet[\underline{\text{not}}(\beta)\ \underline{\text{and}}\ \underline{\text{not}}(\mathit{wp}(\tau)(\underline{\text{not}}(\alpha)))])$ from Proposition 2.5.5.0.5. Since τ is total and deterministic, there exists a total function $\bar{\tau}\in(\mathcal{S}\to\mathcal{B})$ such that $\{\forall e_1,e_2\in\mathcal{S},\{\tau(e_1,e_2)\}\Leftrightarrow\{\bar{\tau}(e_1)=e_2\}\}$. Therefore, $\underline{\text{not}}(\mathit{wp}(\tau)(\underline{\text{not}}(\alpha)))(e_1)=\underline{\text{not}}(\underline{\text{not}}(\alpha(\bar{\tau}(e_1))))=\alpha(\bar{\tau}(e_1))=\mathit{wp}(\tau)(\alpha)(e_1)$ and, finally, $\underline{\text{not}}[\mathit{wp}(\tau^\star)(\beta)]=\mathit{gfp}(\boldsymbol{\lambda}\,\alpha\bullet[\underline{\text{not}}(\beta)\,\underline{\text{and}}\ \mathit{wp}(\tau)(\alpha)])$.

End of proof.

PROPOSITION 3.1.5.0.3

Let $\pi(\mathcal{S},\tau,\nu_\varepsilon,\nu_\sigma,\nu_\xi)$ be a *deterministic discrete dynamic system* and $\phi,\psi\in(\mathcal{S}\to\mathcal{B})$, then:

(a) - $[\mathit{sp}(\tau^\star)(\phi)\ \underline{\text{and}}\ \mathit{wp}(\tau^\star)(\nu_\sigma\ \underline{\text{and}}\ \psi)]=\mathit{sp}[\tau^\star][\mathit{wp}(\tau^\star)(\nu_\sigma\ \underline{\text{and}}\ \psi)\ \underline{\text{and}}\ \phi]$

(b) - $[\mathit{wp}(\tau)(\psi)\ \underline{\text{and}}\ \mathit{sp}(\tau^\star)(\phi)]\Rightarrow\mathit{wp}[\tau][\psi\ \underline{\text{and}}\ \mathit{sp}(\tau^\star)(\phi)]$

$\mathit{sp}(\tau^\star)(\phi)\ \underline{\text{and}}\ \mathit{wp}(\tau^\star)(\psi)$

(c) - $=\ \mathit{lfp}(\boldsymbol{\lambda}\,\alpha\bullet[\mathit{sp}(\tau^\star)(\phi)\ \underline{\text{and}}\ (\psi\ \underline{\text{or}}\ \mathit{wp}(\tau)(\alpha))]\}$

(d) - $=\ \mathit{lfp}(\boldsymbol{\lambda}\,\alpha\bullet[\mathit{wp}(\tau^\star)(\phi)\ \underline{\text{and}}\ (\phi\ \underline{\text{or}}\ \mathit{sp}(\tau)(\alpha))]\}$

(e) - $=\ \mathit{lfp}(\boldsymbol{\lambda}\,\alpha\bullet[\mathit{sp}(\tau^\star)(\phi)\ \underline{\text{and}}\ \mathit{wp}(\tau^\star)(\psi)\ \underline{\text{and}}\ (\psi\ \underline{\text{or}}\ \mathit{wp}(\tau)(\alpha))]\}$

(f) - $=\ \mathit{lfp}(\boldsymbol{\lambda}\,\alpha\bullet[\mathit{sp}(\tau^\star)(\phi)\ \underline{\text{and}}\ \mathit{wp}(\tau^\star)(\psi)\ \underline{\text{and}}\ (\phi\ \underline{\text{or}}\ \mathit{sp}(\tau)(\alpha))]\}$

Proof:

(a) - First, let us prove that $\mathit{sp}(\tau^\star)[\mathit{wp}(\tau^\star)(\nu_\sigma\ \underline{\text{and}}\ \psi)]\Rightarrow\mathit{wp}(\tau^\star)(\nu_\sigma\ \underline{\text{and}}\ \psi)$.

$$\begin{aligned}&\mathit{sp}(\tau^\star)(\mathit{wp}(\tau^\star)(\nu_\sigma\ \underline{\text{and}}\ \psi))(e)\\ &\quad=\ \{\exists e_1,e_2\in\mathcal{S},\exists k,l\in\omega:\tau^k(e_1,e_2)\ \underline{\text{and}}\ \nu_\sigma(e_2)\ \underline{\text{and}}\ \psi(e_2)\ \underline{\text{and}}\ \tau^l(e_1,e)\}\end{aligned}$$

- If $k>l$, then $\tau^k=\tau^l\circ\tau^{k-l}$

$$=\ \{\exists e_1,e_2,e_3\in\mathcal{S},\exists k,l\in\omega:\tau^l(e_1,e_3)\ \underline{\text{and}}\ \tau^{k-l}(e_3,e_2)\ \underline{\text{and}}\ \nu_\sigma(e_2)\ \underline{\text{and}}\ \psi(e_2)\ \underline{\text{and}}\ \tau^l(e_1,e)\}$$

Since τ is deterministic, τ^l is deterministic as well, and therefore $\{e_3 =. e\}$

$\Rightarrow$ $\{\exists e_2 \in \mathcal{S}, \exists k, l \in \omega : \tau^{k-l}(e, e_2) \underline{\text{and}}\, \nu_\sigma(e_2) \underline{\text{and}}\, \psi(e_2)\} = wp(\tau^\star)(\nu_\sigma \underline{\text{and}}\, \psi)(e)$

- If $k \leq l$ then $\tau^l = \tau^k \circ \tau^{l-k}$

$=$ $\{\exists e_1, e_2, e_3 \in \mathcal{S}, \exists k, l \in \omega : \tau^k(e_1, e_2) \underline{\text{and}}\, \nu_\sigma(e_2) \underline{\text{and}}\, \psi(e_2) \underline{\text{and}}\, \tau^k(e_1, e_3) \underline{\text{and}}\, \tau^{l-k}(e_3, e)\}$

Since τ is deterministic, τ^k is deterministic as well, and therefore $\{e_3 = e_2\}$.

However, from 3.1.1.0.1.(h) $\{\nu_\sigma(e_2) \underline{\text{and}}\, \tau^{l-k}(e_2, e)\} \Rightarrow \{e_2 = e\}$.

$\Rightarrow$ $\{\exists e_2 \in \mathcal{S}, \exists k, l \in \omega : \tau^{k-l}(e, e_2) \underline{\text{and}}\, \nu_\sigma(e_2) \underline{\text{and}}\, \psi(e_2)\} = wp(\tau^\star)(\nu_\sigma \underline{\text{and}}\, \psi)(e)$

Then,

$sp(\tau^\star)(wp(\tau^\star)(\nu_\sigma \underline{\text{and}}\, \psi) \underline{\text{and}}\, \phi)$

$\Rightarrow$ $sp(\tau^\star)(wp(\tau^\star)(\nu_\sigma \underline{\text{and}}\, \psi)) \underline{\text{and}}\, sp(\tau^\star)(\phi)$ by monotonicity

$\Rightarrow$ $wp(\tau^\star)(\nu_\sigma \underline{\text{and}}\, \psi) \underline{\text{and}}\, sp(\tau^\star)(\phi)$

Reciprocally,

$wp(\tau^\star)(\nu_\sigma \underline{\text{and}}\, \psi) \underline{\text{and}}\, sp(\tau^\star)(\phi)$

$=$ $\boldsymbol{\lambda} e \bullet \{(\exists e_2 \in \mathcal{S} : \tau^\star(e, e_2) \underline{\text{and}}\, \nu_\sigma(e_2) \underline{\text{and}}\, \psi(e_2)) \underline{\text{and}}\, (\exists e_1 \in \mathcal{S} : \phi(e_1) \underline{\text{and}}\, \tau^\star(e_1, e))\}$

Since $\{\exists e_3 \in \mathcal{S} : \tau^\star(e_1, e_3) \underline{\text{and}}\, \tau^\star(e_3, e_2)\} = \tau^\star \circ \tau^\star(e_1, e_2) = \tau^\star(e_1, e_2)$, we get:

$\Rightarrow$ $\boldsymbol{\lambda} e \bullet \{\exists e_1 \in \mathcal{S} : (\exists e_2 \in \mathcal{S} : \tau^\star(e_1, e_2) \underline{\text{and}}\, \nu_\sigma(e_2) \underline{\text{and}}\, \psi(e_2)) \underline{\text{and}}\, \phi(e_1) \underline{\text{and}}\, \tau^\star(e_1, e)\}$

$=$ $sp(\tau^\star)(wp(\tau^\star)(\nu_\sigma \underline{\text{and}}\, \psi) \underline{\text{and}}\, \phi)$

(b) - Let us prove that $[wp(\tau)(\psi) \underline{\text{and}}\, sp(\tau^\star)(\phi)] \Rightarrow wp(\tau)[\psi \underline{\text{and}}\, sp(\tau^\star)(\phi)]$.

$wp(\tau)(\psi) \underline{\text{and}}\, sp(\tau^\star)(\phi)$

$$= \boldsymbol{\lambda} e \bullet \{(\exists e_2 \in \mathcal{S} : \tau(e, e_2) \;\underline{\text{and}}\; \psi(e_2)) \;\underline{\text{and}}\; (\exists e_1 \in \mathcal{S} : \phi(e_1) \;\underline{\text{and}}\; \tau^\star(e_1, e))\}$$

Since $\{\exists e_3 \in \mathcal{S} : \tau^\star(e_1, e_3) \;\underline{\text{and}}\; \tau(e_3, e_2)\} = \tau^\star \circ \tau(e_1, e_2) \Rightarrow \tau^\star(e_1, e_2)$, we get:

$$\Rightarrow \boldsymbol{\lambda} e \bullet \{\exists e_2 \in \mathcal{S} : \psi(e_2) \;\underline{\text{and}}\; (\exists e_1 \in \mathcal{S} : \phi(e_1) \;\underline{\text{and}}\; \tau^\star(e_1, e_2)) \;\underline{\text{and}}\; \tau^\star(e, e_2)\}$$

$$= \mathit{wp}(\tau)(\psi \;\underline{\text{and}}\; \mathit{sp}(\tau^\star)(\phi))$$

(c) - Let $\langle X^\delta : \delta \leq \omega\rangle$ and $\langle Y^\delta : \delta \leq \omega\rangle$ be the increasing iterations starting from the infimum $\boldsymbol{\lambda} \beta \bullet \underline{\text{false}}$ of $(\mathcal{S} \to \mathcal{B})$ and respectively defined by $\boldsymbol{\lambda} \alpha \bullet [\psi \;\underline{\text{or}}\; \mathit{wp}(\tau)(\alpha)]$ and $\boldsymbol{\lambda} \alpha \bullet [\mathit{sp}(\tau^\star)(\phi) \;\underline{\text{and}}\; (\psi \;\underline{\text{or}}\; \mathit{wp}(\tau)(\alpha))]$. We know that $(X^0 \;\underline{\text{and}}\; \mathit{sp}(\tau^\star)(\phi)) = \boldsymbol{\lambda} \beta \bullet \underline{\text{false}} = Y^0$. Let us assume that $(X^{\delta-1} \;\underline{\text{and}}\; \mathit{sp}(\tau^\star)(\phi)) \Rightarrow Y^{\delta-1}$, then:

$X^\delta \;\underline{\text{and}}\; \mathit{sp}(\tau^\star)(\phi)$

$$= (\psi \;\underline{\text{or}}\; \mathit{wp}(\tau)(X^{\delta-1})) \;\underline{\text{and}}\; \mathit{sp}(\tau^\star)(\phi)$$

$$\Rightarrow (\psi \;\underline{\text{or}}\; (\mathit{wp}(\tau)(X^{\delta-1}) \;\underline{\text{and}}\; \mathit{sp}(\tau^\star)(\phi))) \;\underline{\text{and}}\; \mathit{sp}(\tau^\star)(\phi)$$

$$\Rightarrow (\psi \;\underline{\text{or}}\; \mathit{wp}(\tau)(X^{\delta-1} \;\underline{\text{and}}\; \mathit{sp}(\tau^\star)(\phi))) \;\underline{\text{and}}\; \mathit{sp}(\tau^\star)(\phi) \text{ from (b)}$$

$$\Rightarrow (\psi \;\underline{\text{or}}\; \mathit{wp}(\tau)(Y^{\delta-1})) \;\underline{\text{and}}\; \mathit{sp}(\tau^\star)(\phi) \text{ by induction assumption and monotonicity}$$

$$= Y^\delta$$

Then, $(X^\omega \,\underline{\text{and}}\, \mathit{sp}(\tau^\star)(\phi)) = ((\underset{\alpha<\omega}{\underline{\text{OR}}}\, X^\alpha) \,\underline{\text{and}}\, \mathit{sp}(\tau^\star)(\phi)) = (\underset{\alpha<\omega}{\underline{\text{OR}}}(X^\alpha \,\underline{\text{and}}\, \mathit{sp}(\tau^\star)(\phi))) \Rightarrow \underset{\alpha<\omega}{\underline{\text{OR}}}\, Y^\alpha = Y^\omega$. We get:

$\mathit{sp}(\tau^\star)(\phi) \;\underline{\text{and}}\; \mathit{wp}(\tau^\star)(\psi)$

$$= \mathit{sp}(\tau^\star)(\phi) \;\underline{\text{and}}\; X^\omega \text{ from 3.1.3.0.7, 3.1.3.0.5, and 2.7.0.1}$$

$$\Rightarrow Y^\omega = \mathit{lfp}(\boldsymbol{\lambda} \alpha \bullet [\mathit{sp}(\tau^\star)(\phi) \;\underline{\text{and}}\; (\psi \;\underline{\text{or}}\; \mathit{wp}(\tau)(\alpha))])$$

$$\Rightarrow \mathit{lfp}(\boldsymbol{\lambda} \alpha \bullet [\mathit{sp}(\tau^\star)(\phi)]) \;\underline{\text{and}}\; \mathit{lfp}(\boldsymbol{\lambda} \alpha \bullet [\psi \;\underline{\text{or}}\; \mathit{wp}(\tau)(\alpha)]) \text{ since } \mathit{lfp} \text{ is monotone}$$

$$= \mathit{sp}(\tau^\star)(\phi) \;\underline{\text{and}}\; \mathit{wp}(\tau^\star)(\psi)$$

By antisymmetry, $(sp(\tau^\star)(\phi)\,\underline{\text{and}}\,wp(\tau^\star)(\psi)) = \mathit{lfp}(\boldsymbol{\lambda}\,\alpha \bullet [sp(\tau^\star)(\phi)])\,\underline{\text{and}}\,(\psi\,\underline{\text{or}}\,wp(\tau)(\alpha))])$.
(d), (e), and (f) can be proved in a similar manner.
End of proof.

3.2 DEFINITION OF THE OPERATIONAL SEMANTICS OF A PROGRAMMING LANGUAGE

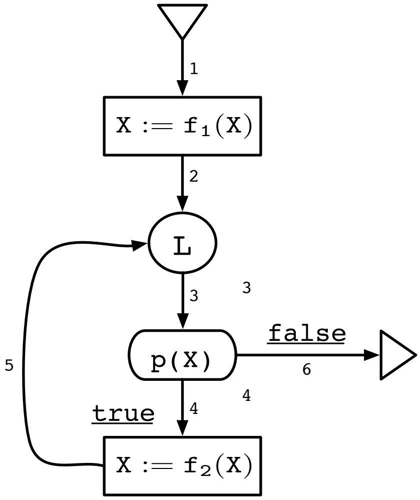
We define the operational semantics of a simple programming language, which allows us to build sequential, iterative programs using assignment, conditional branching, and unconditional branching instructions. We deliberately ignore the issues related to the programming methodology (which would lead us to choose a higher-level language rather than flowcharts), and the issues related to the definition of complex information structures. Our purpose is to show how a program defines a discrete dynamic system.

3.2.1 Abstract syntax of programs

A program with n variables $X = X_1, \ldots, X_n$ will be depicted by a finite connected flowchart with a unique entry node, a unique exit node, and assignment nodes, test nodes, and junction nodes (which correspond to labels in the program). For instance, the following program:

```
{1}
          $X := f_1(X)$ ;
{2}
     L:
{3}
          if p(X) then
{4}
               $X := f_2(X)$ ;
{5}
               goto L ;
           endif ;
{6}
```

will be depicted by the following flowchart:



We choose to follow a rather intuitive presentation since the methods to define formally the context-dependent syntax of programming languages and the techniques to transform a concrete representation of a program (as a string composed of characters and conforming to the concrete syntax) into an abstract representation (as abstract trees or flowcharts) are known (see for instance Lorho [1974]).

3.2.2 Operational semantics of the language

3.2.2.1 Set of states $\mathcal{S}$

Let π be a program with n variables $X_1, \ldots, X_n$ with values in $\mathcal{U}$ and α program points (or edges in the corresponding flowchart) $a_1, \ldots, a_\alpha$. A state $e \in \mathcal{S}$ is a pair $\langle m,\, c\rangle$ where $m \in \mathcal{U}^n$ is the memory state and $c \in \{a_1, \ldots, a_\alpha, \underline{\text{error}}\}$ is the control state.

3.2.2.2 Transition function $\bar{\tau}$

Since we consider deterministic programs, the transition relation $\tau \in ((\mathcal{S} \times \mathcal{S}) \rightarrow \mathcal{B})$ is defined by a transition function $\bar{\tau}$ as $\tau = \boldsymbol{\lambda}\,(e_1, e_2) \bullet [\bar{\tau}(e_1) = e_2]$.

We denote by $dom(f)$ the domain of definition of a partial function f. Then, the transition function $\bar{\tau}$ corresponding to the program π is defined for all $m \in \mathcal{U}^n$ by the following rule patterns:

- $\bar{\tau}(\langle m,\, \underline{\text{error}}\rangle) = \langle m,\, \underline{\text{error}}\rangle$

- If a_1 is the entry point of an assignment instruction $X := f(X)$ the exit point of which is a_2m and if we use the same notation for the syntactic expression f with n variables $X_1, \ldots, X_n$ and for the partial function from $\mathcal{U}^n$ to $\mathcal{U}^n$ it denotes, then we get:

  $$\bar{\tau}(\langle m,\, a_1\rangle) \quad = \quad \underline{\text{if}}\ m \in dom(f)\ \ \underline{\text{then}}\ \ \langle f(m),\, a_2\rangle\ \ \underline{\text{else}}\ \ \langle m,\, \underline{\text{error}}\rangle\ \underline{\text{endif}}$$

- If a is the entry point of a test instruction $p(X)$ the $\underline{\text{true}}$ and $\underline{\text{false}}$ exit points of which are respectively a_t and a_f, and if we use the same notation for the boolean syntactic expression p with n variables $X_1, \ldots, X_n$ for the partial function from $\mathcal{U}^n$ to $\mathcal{B} = \{\underline{\text{true}}, \underline{\text{false}}\}$ it denotes, then we get:

  $$\begin{array}{lcl} \bar{\tau}(\langle m,\, a\rangle) & = & \underline{\text{if}}\ m \in dom(p)\ \ \underline{\text{then}} \\ & & \quad \underline{\text{if}}\ p(m)\ \ \underline{\text{then}}\ \ \langle m,\, a_t\rangle\ \ \underline{\text{else}}\ \ \langle m,\, a_f\rangle\ \underline{\text{endif}} \\ & & \underline{\text{else}} \\ & & \quad \langle m,\, \underline{\text{error}}\rangle \\ & & \underline{\text{endif}} \end{array}$$

- If a_1 is the program point preceding a label L (or entry edge to a junction node L) and a_2 is the program point following this label (or exit edge from a junction node L), i.e.:

  ```
  {a1}              or   {a1}                      or            L :
           L :                      goto L ;              {a2}
  {a2}                              . . .                            . . .
                                 L :                      {a1}
                         {a2}                                        goto L ;
  ```

  then:

  $$\bar{\tau}(\langle m,\, a_1\rangle) \quad = \quad \langle m,\, a_2\rangle$$

- If a_σ is the exit point of the program, then:

$$\bar{\tau}(\langle m,\, a_\sigma\rangle) \quad = \quad \langle m,\, a_\sigma\rangle$$

3.2.2.3 Partitioned deterministic total discrete dynamic system defined by a sequential program

A sequential program π with n variables $X_1, \ldots, X_n$ with value in $\mathcal{U}$ and α program points $a_1, \ldots, a_\alpha$ (where a_ε is the entry point and a_σ is the exit point) defines a partitioned deterministic total discrete dynamic system $\pi(\mathcal{S}, \tau, \nu_1, \ldots, \nu_\alpha, \nu_\xi, \varepsilon, \sigma, \xi)$ where $\mathcal{S}$ and τ are defined as above, $\nu_\xi = \boldsymbol{\lambda}\,\langle m,\, c\rangle \bullet (c = \underline{\text{error}})$, and for all $i = 1, \ldots, \alpha, \nu_i = \boldsymbol{\lambda}\,\langle m,\, c\rangle \bullet (c = a_i)$.

We say that the execution of program π from the initial memory state m_1

- leads to a *semantic error* if and only if $\{\exists m_2 \in \mathcal{U}^n : \tau^\star(\langle m_1,\, a_\varepsilon\rangle, \langle m_2,\, \underline{\text{error}}\rangle)\}$
- *terminates* if and only if $\{\exists m_2 \in \mathcal{U}^n : \tau^\star(\langle m_1,\, a_\varepsilon\rangle, \langle m_2,\, a_\sigma\rangle)\}$

3.3 FORWARD DEDUCTIVE SEMANTICS

In order to perform a semantic analysis of program π, that is, to study the behavior of the discrete dynamic system defined by π, we will characterize the behavior of the descendants of the entry states which satisfy an entry condition $\phi \in \mathcal{P}_n$ where $\mathcal{P}_n = (\mathcal{U}^n \to \mathcal{B})$. Thus, it amounts to determining $sp(\tau^\star)(\nu_\varepsilon \ \underline{\text{and}}\ \bar{\phi})$ where $\bar{\phi} = \boldsymbol{\lambda}\,\langle m,\, c\rangle \bullet \phi(m)$. Since the set $\mathcal{S}$ of states is partitioned, Propositions 3.1.3.0.7 and 3.1.4.0.2 show that $sp(\tau^\star)(\nu_\varepsilon \ \underline{\text{and}}\ \bar{\phi})$ is isomorphic to the least solution of *a system of forward semantic equations* associated with program π, and of the form:

$$\left\{\begin{array}{lcl} P_i & = & (\nu_i \ \underline{\text{and}}\ \nu_\varepsilon \ \underline{\text{and}}\ \bar{\phi}) \ \underline{\text{or}}\ (\underset{j \in \underline{\text{pred}}(i)}{\underline{\text{OR}}} \ sp(\tau_{ji})(P_j)) \\ \\ i & = & 1, \ldots, \alpha, \xi \end{array}\right.$$

For all $i = 1, \ldots, \alpha$, we shall choose $P_i = (\mathcal{U}^n \to \mathcal{B})$ which is simpler than $P_i \in \sigma_i(\mathcal{S} \to \mathcal{B}) = \sigma_i([\{a_1, \ldots, a_\alpha, \underline{\text{error}}\} \times \mathcal{U}^n] \to \mathcal{B})$ but is equivalent, since $\sigma_i(\mathcal{S} \to \mathcal{B})$

and $(\mathcal{U}^n \rightarrow \mathcal{B})$ are isomorphic, by the complete isomorphism $\iota_i = \boldsymbol{\lambda}\,\beta \bullet \{\boldsymbol{\lambda}\,m \bullet [\beta(\langle m, a_i\rangle)]\}$ the inverse of which is $\iota_i^{-1} = \boldsymbol{\lambda}\,\beta \bullet \{\boldsymbol{\lambda}\,\langle m,\, c\rangle \bullet [\beta(m)]\}$.

Since the entry states are exogenous (3.1.1.0.1.(g)), the set $\underline{\text{pred}}(\varepsilon)$ is empty, and therefore, $P_\varepsilon = \phi$. For all $i \neq \varepsilon$, the predicate (ν_i $\underline{\text{and}}$ ν_ε) is false (3.1.4.0.1.(d)), and thus:

$$P_i \;=\; \underset{j \in \underline{\text{pred}}(i)}{\underline{\text{OR}}} \;\iota_i[sp(\tau_{ji})(\iota_j^{-1}[P_j])]$$

We need to compute:

$$\begin{aligned}
&\iota_i[sp(\tau_{ji})(\iota_j^{-1}[P_j])]\\
&\quad= \iota_i[sp(\tau_{ji})(\boldsymbol{\lambda}\,\langle m,\, c\rangle \bullet [P_j(m)])]\\
&\quad= \iota_i[\boldsymbol{\lambda}\,\langle m_2,\, c_2\rangle \bullet \{\exists\langle m_1,\, c_1\rangle \in \mathcal{S} : P_j(m_1)\ \underline{\text{and}}\ \tau_{ji}(\langle m_1,\, c_1\rangle, \langle m_2,\, c_2\rangle)\}]\\
&\quad= \boldsymbol{\lambda}\,m_2 \bullet \{\exists\langle m_1,\, c_1\rangle \in \mathcal{S} : P_j(m_1)\ \underline{\text{and}}\ \tau_{ji}(\langle m_1,\, c_1\rangle, \langle m_2,\, a_i\rangle)\}\\
&\quad= \boldsymbol{\lambda}\,m_2 \bullet \{\exists\langle m_1,\, c_1\rangle \in \mathcal{S} : P_j(m_1)\ \underline{\text{and}}\ \nu_j(\langle m_1,\, c_1\rangle)\ \underline{\text{and}}\ \tau(\langle m_1,\, c_1\rangle, \langle m_2,\\
&\qquad a_i\rangle)\ \underline{\text{and}}\ \nu_i(\langle m_2,\, a_i\rangle)\}\\
&\quad= \boldsymbol{\lambda}\,m_2 \bullet \{\exists\langle m_1,\, c_1\rangle \in \mathcal{S} : P_j(m_1)\ \underline{\text{and}}\ (c_1 = a_j)\ \underline{\text{and}}\ \tau(\langle m_1,\, c_1\rangle, \langle m_2,\, a_i\rangle)\}\\
&\quad= \boldsymbol{\lambda}\,m_2 \bullet \{\exists m_1 \in \mathcal{U}^n : P_j(m_1)\ \underline{\text{and}}\ \bar{\tau}(\langle m_1,\, a_j\rangle) = \langle m_2,\, a_i\rangle\}
\end{aligned}$$

- If a_i is the exit point of an assignment instruction $X := f(X)$, then $\underline{\text{pred}}(i) = \{j\}$ where a_j is the entry point of this instruction, thus:

$$\begin{aligned}
P_i &= \boldsymbol{\lambda}\,m_2 \bullet \{\exists m_1 \in \mathcal{U}^n : P_j(m_1)\ \underline{\text{and}}\ [(m_1 \in dom(f))\ \underline{\text{and}}\ (\langle f(m_1),\, a_i\rangle = \langle m_2,\\
&\quad a_i\rangle)]\}\\
&= \boldsymbol{\lambda}\,m_2 \bullet \{\exists m_1 \in \mathcal{U}^n : P_j(m_1)\ \underline{\text{and}}\ m_1 \in dom(f)\ \underline{\text{and}}\ f(m_1) = m_2\}\\
&= \underline{\text{assign}}(f)(P_j) \quad \text{(by definition of } \underline{\text{assign}})
\end{aligned}$$

- If a_i is the $\underline{\text{true}}$ exit point of a test instruction $p(X)$, then $\underline{\text{pred}}(i) = \{j\}$ where a_j is the entry point of this instruction, thus:

$$\begin{aligned}
P_i &= \boldsymbol{\lambda}\,m_2 \bullet \{\exists m_1 \in \mathcal{U}^n : P_j(m_1)\ \underline{\text{and}}\ \bar{\tau}(\langle m_1,\, a_j\rangle) = \langle m_2,\, a_i\rangle\}\\
&= \boldsymbol{\lambda}\,m_2 \bullet \{\exists m_1 \in \mathcal{U}^n : P_j(m_1)\ \underline{\text{and}}\ (m_1 \in dom(p))\ \underline{\text{and}}\ p(m_1)\ \underline{\text{and}}\ (\langle m_1,\\
&\quad a_i\rangle = \langle m_2,\, a_i\rangle)\}
\end{aligned}$$

$$\begin{aligned} &= \boldsymbol{\lambda} m_2 \bullet \{P_j(m_2)\ \underline{\mathrm{and}}\ (m_2 \in dom(p))\ \underline{\mathrm{and}}\ p(m_2)\} \\ &= \underline{\mathrm{test}}(p)(P_j) \quad \text{(by definition of } \underline{\text{test}}\text{)} \end{aligned}$$

- Similarly, if a_i is the false exit point of a test instruction $p(X)$ and a_j is the entry point of that instruction, then:

$$P_i \quad = \quad \underline{\mathrm{test}}(\underline{\mathrm{not}}\,(p))(P_j)$$

- If a_i is the program point following a label that follows program point a_j where $j \in \underline{\mathrm{pred}}(i)$, then:

$$\begin{aligned} P_i &= \underset{j \in \underline{\mathrm{pred}}(i)}{\underline{\mathrm{OR}}} \quad \iota_i[sp(\tau_{ji})(\iota_j^{-1}[P_j])] \\ &= \underset{j \in \underline{\mathrm{pred}}(i)}{\underline{\mathrm{OR}}} \quad \boldsymbol{\lambda} m_2 \bullet \{\exists m_1 \in \mathcal{U}^n : P_j(m_1)\ \underline{\mathrm{and}}\ \bar{\tau}(\langle m_1,\, a_j\rangle) = \langle m_2,\, a_i\rangle\} \\ &= \underset{j \in \underline{\mathrm{pred}}(i)}{\underline{\mathrm{OR}}} \quad \boldsymbol{\lambda} m_2 \bullet \{\exists m_1 \in \mathcal{U}^n : P_j(m_1)\ \underline{\mathrm{and}}\ (\langle m_1,\, a_i\rangle = \langle m_2,\, a_i\rangle)\} \\ &= \underset{j \in \underline{\mathrm{pred}}(i)}{\underline{\mathrm{OR}}} \quad P_j \end{aligned}$$

- Since any execution error terminates the execution of the program, the program has no error recovery, thus, the error states are stable. As a consequence, for all $i = 1, \ldots, \alpha$, P_i is independent from P_ξ. Thus, when solving the system of forward equations associated with the program, we can ignore the equation that defines P_ξ, which could be evaluated last, if required for the application considered. In this case:

$$\begin{aligned} P_\xi &= \underset{j \in \underline{\mathrm{pred}}(\xi)}{\underline{\mathrm{OR}}} \quad \iota_\xi[sp(\tau_{j\xi})(\iota_j^{-1}[P_j])] \\ &= \underset{j \in \underline{\mathrm{pred}}(\xi)}{\underline{\mathrm{OR}}} \quad \boldsymbol{\lambda} m_2 \bullet \{\exists m_1 \in \mathcal{U}^n : P_j(m_1)\ \underline{\mathrm{and}}\ \bar{\tau}(\langle m_1,\, a_j\rangle) = \langle m_2,\, \underline{\mathrm{error}}\rangle\} \\ &= P_\xi\ \underline{\mathrm{or}}\ \underset{j \in \underline{\mathrm{at}}(\pi)}{\underline{\mathrm{OR}}}\ \boldsymbol{\lambda} m_2 \bullet [P_j(m_2)\ \underline{\mathrm{and}}\ m_2 \notin dom(\underline{\mathrm{expr}}(j))] \end{aligned}$$

where $\underline{\mathrm{at}}(\pi)$ denotes the set of $j \in [1, \alpha]$ such that a_j is the entry point of an assignment instruction or of a test instruction in program π and $\underline{\mathrm{expr}}(j)$ denotes the function f of the assignment instruction $X := f(X)$ or predicate p of the test instruction $p(X)$ the entry point of which is a_j.

To summarize:

<u>DEFINITION</u> 3.3.0.1 *System of forward semantic equations associated with a program and an entry specification*

Let π be a program with n variables with values in $\mathcal{U}$ and α program points $a_1, \ldots, a_\alpha$ (where a_ε is the entry point) and an entry specification $\phi \in \mathcal{P}_n = (\mathcal{U}^n \rightarrow \mathcal{B})$. Let $P_1, \ldots, P_\alpha$ be variables with values in $\mathcal{P}_n$. Then, the *system of forward semantic equations* $P = F_\pi(\phi)(P)$ *associated with* (π, ϕ) is defined by the following rules:

- If a_i is the entry point of the program, then $P_i = \phi$

- If a_j is the entry point of an assignment instruction $X := f(X)$ the exit point of which is a_i, then $P_i = \underline{\text{assign}}(f)(P_j)$ where $\underline{\text{assign}} = \boldsymbol{\lambda} f \bullet \{\boldsymbol{\lambda} P \bullet [\boldsymbol{\lambda} m_2 \bullet (\exists m_1 \in \mathcal{U}^n : P(m_1) \ \underline{\text{and}} \ (m_1 \in dom(f)) \ \underline{\text{and}} \ (m_2 = f(m_1)))]\}$

- If a_j is the entry point of a test instruction $p(X)$ and a_i is the <u>true</u> (respectively <u>false</u>) exit point of this instruction, then $P_i = \underline{\text{test}}(p)(P_j)$ (respectively $P_i = \underline{\text{test}}(\underline{\text{not}}(p))(P_j)$) where $\underline{\text{test}} = \boldsymbol{\lambda} p \bullet \{\boldsymbol{\lambda} P \bullet [\boldsymbol{\lambda} m \bullet (P(m) \ \underline{\text{and}} \ (m \in dom(p)) \ \underline{\text{and}} \ p(m))]\}$

- If a_i is the program point following a label that follows the program points $a_{j_1}, \ldots, a_{j_k}$, then $P_i = \underset{l=1}{\overset{k}{\underline{\text{OR}}}} P_{j_l}$

From 3.1.3.0.5.(a), 3.1.3.0.7, and 3.1.4.0.2, we derive the proposition:

<u>PROPOSITION</u> 3.3.0.2 *Property of the least fixpoint of the system of forward semantic equations*

The operator $F_\pi(\phi)$ on $(\mathcal{P}_n)^\alpha$ is a complete join-morphism. The least fixpoint $(P_1, \ldots, P_\alpha)$ of $F_\pi(\phi)$ is such that, for all $i \in [1, \alpha]$, we have:

$$P_i \quad = \quad \boldsymbol{\lambda}\, m_2 \bullet \{\exists m_1 \in \mathcal{U}^n : \phi(m_1) \ \underline{\text{and}}\ \tau^\star(\langle m_1,\, a_\varepsilon\rangle, \langle m_2,\, a_i\rangle)\}$$

PROPOSITION 3.3.0.3 *Conjunction and disjunction of entry specifications*

$\boldsymbol{\lambda}\, \phi \bullet [\mathit{lfp}(F_\pi(\phi))]$ is a complete join-morphism. If π is injective, it is a complete meet-morphism.

3.4 TECHNIQUES FOR PROGRAM ANALYSIS BASED ON THE FORWARD DEDUCTIVE SEMANTICS

We use the results of Paragraph 3.3 in order to prove the method of Floyd and Naur to verify the partial correctness of a program, and then to extend this method to verify the total correctness of a program. Then, we justify the criterion for termination of programs by Katz and Manna. We illustrate the techniques for the analysis of the conditions of termination with no error, of non-termination, and of incorrect execution of programs, based on the forward deductive semantics. We also show how to use the forward deductive semantics in order to characterize, at each point in a program, the set of descendants of the initial states which satisfy an entry condition. Last, we show that the symbolic execution of a program consists in solving the system of forward semantic equations associated with the program using a chaotic iteration strategy.

3.4.1 Justification of the method by Floyd and Naur to verify the partial correctness of a program

Floyd [1967] and Naur [1966] have justified their method to verify the partial correctness of programs by a reasoning based on the operational semantics. The forward deductive semantics makes a more elegant proof possible.

Let π be a program with n variables with values in $\mathcal{U}$, α program points $a_1, \ldots, a_\alpha$ (the entry point and exit point are respectively denoted as a_ε and a_σ) and $\mathcal{P}_n = (\mathcal{U}^n \rightarrow \mathcal{B})$. A proof of partial correctness for program π, the entry specification ϕ, and the exit specification ψ consists in proving that:

$$\{\forall m_2 \in \mathcal{U}^n, [\exists m_1 \in \mathcal{U}^n : \phi(m_1) \ \underline{\text{and}}\ \tau^\star(\langle m_1,\ a_\varepsilon\rangle, \langle m_2,\ a_\sigma\rangle)] \Rightarrow [\psi(m_2)]\}$$

If we can *guess* a post-fixpoint $P \in (\mathcal{P}_n)^\alpha$ of $F_\pi(\psi)$ and such that $\{P_\sigma \Rightarrow \psi\}$, then Theorem 2.5.3.0.2 shows that $\{[\mathit{lfp}(F_\pi(\phi))] \Rightarrow P\}$, and so, $\{[\mathit{lfp}(F_\pi(\phi))]_\sigma \Rightarrow \psi\}$, and Proposition 3.3.0.2 implies that program π is partially correct for (ϕ, ψ) since we have:

$$\boldsymbol{\lambda}\, m_2 \bullet \{\exists m_1 \in \mathcal{U}^n : \phi(m_1) \ \underline{\text{and}}\ \tau^\star(\langle m_1,\ a_\varepsilon\rangle, \langle m_2,\ a_\sigma\rangle)\} \Rightarrow \psi$$

In practice, P is specified by providing only the loop invariants since the other invariants can be derived by replacing the loop invariants with their values in the system of equations.

We have proved that the method by Floyd–Naur is valid, but also that it is complete, which means that (de Bakker & Merteens [1975]), if π is partially correct for (ϕ, ψ), there always exists some $P \in (\mathcal{P}_n)^\alpha$ which allows us to prove it with this method, that is, we can find P such that $\{[F_\pi(\phi)(P) \Rightarrow P] \ \underline{\text{and}}\ [F_\sigma \Rightarrow \psi]\}$. Using the above method, this is achieved by letting $P = \mathit{lfp}(F_\pi(\phi))$.

3.4.2 Extension of the method by Floyd and Naur to verify the total correctness of a program

A proof of the total correctness of a program π for an entry specification ϕ and an exit specification ψ consists in proving that:

$$\{\nu_\varepsilon \ \underline{\text{and}}\ \phi\} \quad \Rightarrow \quad \{\mathit{wp}(\tau^\star)(\nu_\sigma \ \underline{\text{and}}\ \psi)\}$$

Proposition 3.1.3.0.8 allows us to use a system of forward semantic equations, so as to carry out this proof, since it is equivalent to proving that:

$$\{\nu_\varepsilon \ \underline{\text{and}}\ \phi\} \quad \Rightarrow \quad \{\boldsymbol{\lambda}\,\bar{e} \bullet [\exists e_1 \in \mathcal{S} : \nu_\sigma(e_1)\ \underline{\text{and}}\ \psi(e_1)\ \underline{\text{and}}\ \mathit{lfp}(\boldsymbol{\lambda}\,\alpha \bullet [\boldsymbol{\lambda}\,e \bullet (e = \bar{e})\ \underline{\text{or}}\ \mathit{sp}(\tau)(\alpha)])(e_1)]\}$$

that is, up to an isomorphism:

$$\phi \quad \Rightarrow \quad \boldsymbol{\lambda}\,\bar{m} \bullet \{\exists m_1 \in \mathcal{U}^n : [\mathit{lfp}(F_\pi(\boldsymbol{\lambda}\,m \bullet (m = \bar{m})))]_\sigma(m_1)\ \underline{\text{and}}\ \psi(m_1)\}$$

In particular, the weakest entry specification that guarantees the termination of program π is:

$$\boldsymbol{\lambda}\,\bar{m} \bullet \{\exists m_1 \in \mathcal{U}^n : [\mathit{lfp}(F_\pi(\boldsymbol{\lambda}\,m \bullet (m = \bar{m})))]_\sigma(m_1)\}$$

3.4.3 Justification of the criterion for the termination of programs by Katz and Manna

Using our notations, the criterion for the termination of programs given by Katz & Manna [1976] is the following: "if, for all $P \in (\mathcal{P}_n)^\alpha$ such that P is a set of invariants of program π, we can prove that $\{\forall \bar{m} \in \mathcal{U}^n, \phi(\bar{m}) \Rightarrow P_\sigma(\bar{m})\}$, then the execution of program π terminates for all the memory states which satisfy ϕ." In this statement, an invariant P_i at point a_i of program π is informally defined as "a predicate about the variables which holds true for the current value of the variables each time point i is reached during an execution starting from the initial state $\bar{m}$." The method used to verify that a predicate P is an invariant is the same method as Floyd–Naur's, that is, following Paragraph 3.4.1, we check that $P \Leftarrow F_\pi(\boldsymbol{\lambda}\,x \bullet (x = \bar{m}))(P)$. Formally, the termination criterion by Katz and Manna is thus:

$$\phi \quad \Rightarrow \quad \boldsymbol{\lambda}\,\bar{m} \bullet [\exists m_1 \in \mathcal{U}^n : \underline{\text{AND}}\{P_\sigma(m_1) : P \Leftarrow F_\pi(\boldsymbol{\lambda}\,x \bullet (x = \bar{m}))(P)\}]$$

From Theorem 2.5.3.0.2, we get:

$$\underline{\text{AND}}\{P : P \Leftarrow F_\pi(\boldsymbol{\lambda}\,x \bullet (\bar{x} = \bar{m}))(P)\} \quad = \quad \mathit{lfp}(F_\pi(\boldsymbol{\lambda}\,x \bullet (x = \bar{m})))$$

and we have established our criterion.

Katz and Manna have noted that their criterion cannot be used in practice, since there usually exists infinitely many invariants associated with a program. Thus, it is rarely possible to discover the general form of the fixpoints of $F_\pi(\boldsymbol{\lambda} x \bullet (x = \bar{m}))$.

On the opposite, our criterion is based on $lfp(F_\pi(\boldsymbol{\lambda} x \bullet (x = \bar{m})))$, which can be computed by successive approximations, by guessing the general term of the sequence, and proving by induction that the shape of these terms is well chosen, and by passing to the limit to the ω^{th} term (Theorem 2.7.0.1). Of course, this process can be used only for manual computations, and an exact computation is not possible in many cases where guessing the shape of the general term is very hard. We will give a few examples of exact resolution of semantic equations later in this section and in Paragraph 3.6. We will study constructive methods for the approximation of these exact solutions in Chapter 4.

3.4.4 Characterization of the descendants of the entry states and conditions for the termination, non-termination, and erroneous execution of a program

Let us consider the following program, where x is an integer variable that takes values in the range $[-b+1, b]$ of machine integers:

```
{1}
    while   x ≥ 1000   do
{2}
        x := x + α ;
{3}
    redo ;
{4}
```

We shall assume that α is a positive integer constant. By definition of the "while" iteration instruction, this program is equivalent to the program:

```
{1}
    L :
        if x ≥ 1000 then
```

```
{2}
                x := x + $\alpha$ ;
{3}
                goto L ;
            endif ;
{4}
```

The system of forward semantic equations associated with the latter program and the entry specification $\boldsymbol{\lambda} x \bullet (x = \bar{x})$ is:

$$\left\{\begin{array}{lcl} P_1 & = & \boldsymbol{\lambda} x \bullet [x = \bar{x}] \\ P_2 & = & \underline{\text{test}}(\boldsymbol{\lambda} x \bullet [x \geq 1000])(P_1 \ \underline{\text{or}}\ P_3) \\ P_3 & = & \underline{\text{assign}}(\boldsymbol{\lambda} x \bullet [x + \alpha])(P_2) \\ P_4 & = & \underline{\text{test}}(\boldsymbol{\lambda} x \bullet [x < 1000])(P_1 \ \underline{\text{or}}\ P_3) \end{array}\right.$$

More simply, the equation that defines P_2 is:

$$\begin{array}{lcl} P_2 & = & \underline{\text{test}}(\boldsymbol{\lambda} x \bullet [x \geq 1000])(\boldsymbol{\lambda} x \bullet (x = \bar{x}) \ \underline{\text{or}}\ \underline{\text{assign}}(\boldsymbol{\lambda} x \bullet [x + \alpha])(P_2)) \\ & = & \boldsymbol{\lambda} x \bullet \{-b + 1 \leq x \leq b) \ \underline{\text{and}}\ (1000 \leq x) \ \underline{\text{and}}\ [(x = \bar{x}) \ \underline{\text{or}}\ (\exists y : (-b + 1 \leq \\ & & y + \alpha \leq b) \ \underline{\text{and}}\ (x = y + \alpha) \ \underline{\text{and}}\ P_2(y))]\} \\ & = & \boldsymbol{\lambda} x \bullet \{(1000 \leq x \leq b) \ \underline{\text{and}}\ [(x = \bar{x}) \ \underline{\text{or}}\ P_2(x - \alpha)]\} \end{array}$$

The least fixpoint is obtained by successive approximations, starting from the infimum $\boldsymbol{\lambda} x \bullet [\underline{\text{false}}]$ (Theorem 2.5.3.0.2):

$$\begin{array}{lcl} P_2^0 & = & \boldsymbol{\lambda} x \bullet [\underline{\text{false}}] \\ P_2^1 & = & \boldsymbol{\lambda} x \bullet \{(1000 \leq x \leq b) \ \underline{\text{and}}\ (x = \bar{x})\} \\ P_2^2 & = & \boldsymbol{\lambda} x \bullet \{(1000 \leq x \leq b) \underline{\text{and}} [(x = \bar{x}) \underline{\text{or}} ((1000 \leq x - \alpha \leq b) \underline{\text{and}} (x - \alpha = \bar{x}))]\} \\ & = & \boldsymbol{\lambda} x \bullet \{(1000 \leq \bar{x}) \ \underline{\text{and}}\ (x \leq b) \ \underline{\text{and}}\ [(x = \bar{x}) \ \underline{\text{or}}\ (x = \bar{x} + \alpha)]\} \end{array}$$

For the induction step, let us assume that:

$$P_2^k \ = \ \boldsymbol{\lambda} x \bullet \{(1000 \leq \bar{x}) \ \underline{\text{and}}\ (x \leq b) \ \underline{\text{and}}\ \underset{j=0}{\overset{k-1}{\underline{\text{OR}}}}(x = \bar{x} + j\alpha)\}$$

and let us verify that P_2^{k+1} is of the same form:

$$\begin{array}{rcl} P_2^{k+1} & = & \boldsymbol{\lambda} x \bullet \{(1000 \leq x \leq b) \ \underline{\text{and}} \ [(x = \bar{x}) \ \underline{\text{or}} \ P_2^k(x - \alpha)]\} \\ & = & \boldsymbol{\lambda} x \bullet \{(1000 \leq x \leq b) \ \underline{\text{and}} \ [(x = \bar{x}) \ \underline{\text{or}} \ ((1000 \leq \bar{x}) \ \underline{\text{and}} \ (x \leq b + \alpha) \ \underline{\text{and}} \\ & & \underset{j=0}{\overset{k-1}{\underline{\text{OR}}}}(x = \bar{x} + (j+1)\alpha))]\} \\ & = & \boldsymbol{\lambda} x \bullet \{(1000 \leq \bar{x}) \ \underline{\text{and}} \ (x \leq b) \ \underline{\text{and}} \ \underset{j=0}{\overset{k}{\underline{\text{OR}}}}(x = \bar{x} + j\alpha)\} \end{array}$$

By induction on k, we have found the general term of the sequence, so that the least fixpoint can be obtained by passing to the limit (Theorem 2.7.0.1 and Proposition 3.3.0.2):

$$P_2^\omega \ = \ \underset{k \in \omega}{\underline{\text{OR}}} P_2^k \ = \ \boldsymbol{\lambda} x \bullet \{(1000 \leq \bar{x}) \ \underline{\text{and}} \ (x \leq b) \ \underline{\text{and}} \ (\exists j \geq 0 : x = \bar{x} + j\alpha)\}$$

The other components can be obtained by replacing P_2 with P_2^ω in the initial system of equations, which results in:

$$\left[\begin{array}{rcl} P_1^\omega & = & \boldsymbol{\lambda} x \bullet \{x = \bar{x}\} \\ P_2^\omega & = & \boldsymbol{\lambda} x \bullet \{(1000 \leq \bar{x}) \ \underline{\text{and}} \ (x \leq b) \ \underline{\text{and}} \ (\exists j \geq 0 : x = \bar{x} + j\alpha)\} \\ P_3^\omega & = & \boldsymbol{\lambda} x \bullet \{(1000 \leq \bar{x}) \ \underline{\text{and}} \ (x \leq b) \ \underline{\text{and}} \ (\exists j \geq 1 : x = \bar{x} + j\alpha)\} \\ P_4^\omega & = & \boldsymbol{\lambda} x \bullet \{(x = \bar{x}) \ \underline{\text{and}} \ (-b + 1 \leq x < 1000)\} \end{array}\right.$$

From Proposition 3.3.0.2, we have obtained a characterization of the possible values of x at each program point, and for any execution of the program starting from an initial value of x which satisfies the entry specification $\phi = \boldsymbol{\lambda} x \bullet (x = \bar{x})$.

In Paragraph 3.4.2, we have seen that the weakest entry specification that ensures the termination of the program with no semantic error is:

$$\begin{array}{l} \boldsymbol{\lambda} \bar{x} \bullet \{\exists x_1 : P_4^\omega(x_1)\} \\ \quad = \ \boldsymbol{\lambda} \bar{x} \bullet \{-b + 1 \leq \bar{x} < 1000\} \end{array}$$

The set of the entry states leading to a semantic error is characterized by:

$$\begin{aligned}
&\boldsymbol{\lambda}\,\bar{x} \bullet \{\exists x_1 : P_\xi^\omega(x_1)\}\\
&\quad = \boldsymbol{\lambda}\,\bar{x} \bullet \{\exists x : [(P_1^\omega(x)\ \underline{\text{or}}\ P_3^\omega(x))\ \underline{\text{and}}\ \underline{\text{not}}\,(-b+1 \leq x \leq b)]\ \underline{\text{or}}\ [P_2^\omega(x)\ \underline{\text{and}}\\
&\qquad \underline{\text{not}}\,(-b+1 \leq x+\alpha \leq b)]\}\\
&\quad = \boldsymbol{\lambda}\,\bar{x} \bullet \{[(\bar{x} < -b+1)\ \underline{\text{or}}\ (b < \bar{x})\ \underline{\text{or}}\ [(\alpha \neq 0)\ \underline{\text{and}}\ (1000 \leq \bar{x} \leq b)]\}
\end{aligned}$$

The set of entry states for which the program does not terminate is characterized by:

$$\begin{aligned}
&\boldsymbol{\lambda}\,\bar{x} \bullet \{\exists x_1 : \underline{\text{not}}\,(P_\xi^\omega(x_1)\ \underline{\text{or}}\ P_4^\omega(x_1))\}\\
&\quad = \boldsymbol{\lambda}\,\bar{x} \bullet \{(1000 \leq x \leq b)\ \underline{\text{and}}\ (\alpha = 0)\}
\end{aligned}$$

3.4.5 Symbolic execution

Let π be a program with n variables $x = (x_1, \ldots, x_n)$ and let $\bar{x} = (\bar{x}_1, \ldots, \bar{x}_n)$ be Skolem constants associated with undetermined, fixed elements of $\mathcal{U}^n$. We show that the symbolic execution of program π consists in computing the least solution of the system of forward semantic equations $P = F_\pi(\boldsymbol{\lambda}\,x \bullet (\phi(\bar{x})\ \underline{\text{and}}\ (x = \bar{x})))(P)$ associated with π, using any chaotic iteration strategy (Cousot & Cousot [1977e]).

Let $P^0, \ldots, P^n, \ldots, P^\omega$ be an increasing chaotic iteration sequence starting from the infimum P^0 of $(\mathcal{P}_n)^\alpha$ and defined by $F_\pi(\boldsymbol{\lambda}\,x \bullet (\phi(\bar{x})\ \underline{\text{and}}\ (x = \bar{x})))$. Each term P_i^n is of the form:

$$\boldsymbol{\lambda}\,x \bullet \{\underset{j \in \Delta}{\underline{\text{OR}}}[Q_j(\bar{x})\ \underline{\text{and}}\ x = E_j(\bar{x})]\}$$

where Q_j and E_j are formal expressions that depend on the initial value $\bar{x}$ of the variables and where none of the variables $x_1, \ldots, x_n$ appear as a free variable. By convention, this equals $\boldsymbol{\lambda}\,x \bullet (x = \underline{\text{false}})$ when $\Delta = \phi$.

In order to show that P_i^n can be written this way, it is sufficient to note that $\underline{\text{assign}}(f)$ and $\underline{\text{test}}(p)$ are strict and that:

- $\underline{\text{assign}}(f)(\boldsymbol{\lambda} x \bullet \{\underset{j\in\Delta}{\underline{\text{OR}}}[Q_j(\bar{x})\ \underline{\text{and}}\ x = E_j(\bar{x})]\})$

$$= \quad \boldsymbol{\lambda} x \bullet \{\underset{j\in\Delta}{\underline{\text{OR}}}[(Q_j(\bar{x})\ \underline{\text{and}}\ E_j(\bar{x}) \in \mathit{dom}(f))\ \underline{\text{and}}\ x = f(E_j(\bar{x}))]\}$$

- $\underline{\text{test}}(p)(\boldsymbol{\lambda} x \bullet \{\underset{j\in\Delta}{\underline{\text{OR}}}[Q_j(\bar{x})\ \underline{\text{and}}\ x = E_j(\bar{x})]\})$

$$= \quad \boldsymbol{\lambda} x \bullet \{\underset{j\in\Delta}{\underline{\text{OR}}}[(Q_j(\bar{x})\ \underline{\text{and}}\ E_j(\bar{x}) \in \mathit{dom}(f)\ \underline{\text{and}}\ p(E_j(\bar{x})))\ \underline{\text{and}}\ x = E_j(\bar{x})]\}$$

- $\underset{k=1}{\overset{l}{\underline{\text{OR}}}}(\boldsymbol{\lambda} x \bullet \{\underset{j_k\in\Delta_k}{\underline{\text{OR}}}\ [Q_{j_k}(\bar{x})\ \underline{\text{and}}\ x = E_{j_k}(\bar{x})]\})$

$$= \quad \boldsymbol{\lambda} x \bullet (\underline{\text{OR}}\{[Q_{j_k}(\bar{x})\ \underline{\text{and}}\ x = E_{j_k}(\bar{x})] : j_k \in \bigcup_{k=l}^{l} \Delta_k\})$$

Let us consider the expression:

$$P \quad = \quad \underset{j\in\Delta}{\underline{\text{OR}}}\, C_j \qquad \text{where } C_j = \boldsymbol{\lambda} x \bullet \{Q_j(\bar{x})\ \underline{\text{and}}\ x = E_j(\bar{x})\}$$

Then, we can view each C_j as a characterization for an execution path of π where $Q_j(\bar{x})$ describes the conditions that should be verified so that any execution of π starting from a_ε and with the initial state of variables $\bar{x}$ visits program point a_i in a state where the variables x are equal to $E_j(\bar{x})$.

Equivalently, we will write P as a *symbolic context* $\{\langle Q_j(\bar{x}),\ (E_j)_1(\bar{x}),\ \ldots,\ (E_j)_n(\bar{x})\rangle : j \in \Delta\}$ where ϕ corresponds to $\boldsymbol{\lambda} x \bullet [\underline{\text{false}}]$.

For instance, let us consider the following program:

```
{1}
      while   x ≥ y   do
{2}
            x := x − y ;
{3}
      redo ;
{4}
```

The corresponding system of forward semantic equations is:

$$\begin{cases} P_1 & = & \{\langle \underline{\text{true}},\ \bar{x},\ \bar{y}\rangle\} \\ P_2 & = & \underline{\text{test}}(\boldsymbol{\lambda}\,(x,y)\bullet[x \geq y])(P_1\ \underline{\text{or}}\ P_3) \\ P_3 & = & \underline{\text{assign}}(\boldsymbol{\lambda}\,(x,y)\bullet[x-y, y])(P_2) \\ P_4 & = & \underline{\text{test}}(\boldsymbol{\lambda}\,(x,y)\bullet[x < y])(P_1\ \underline{\text{or}}\ P_3) \end{cases}$$

We will assume that the program manipulates integers (so that we can assume that $x - y$ is always defined).

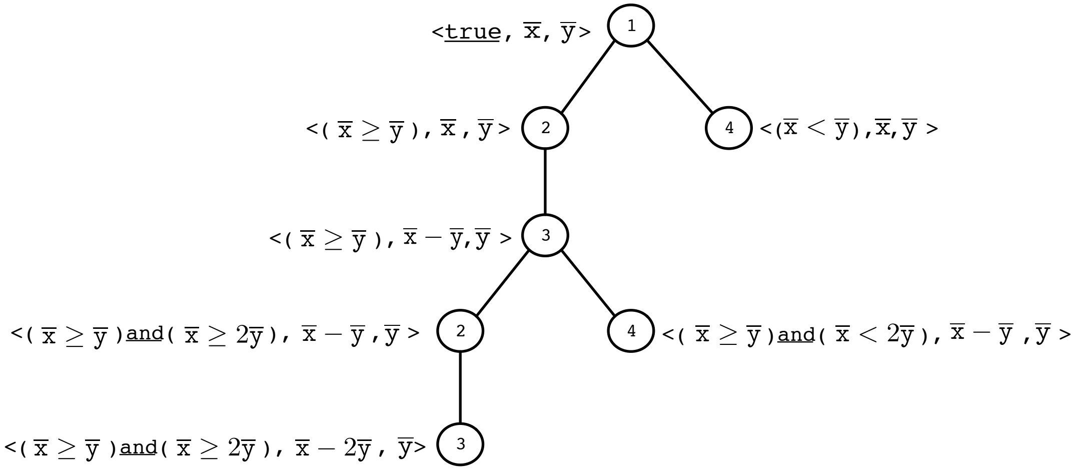
If we choose a chaotic iteration strategy that corresponds to the execution order of the program, we get the first terms of the iteration sequence as follows:

$$\left[\begin{array}{lcl} P_i^0 & = & \phi \quad (i = 1, \ldots, 4) \end{array}\right.$$

$$\left[\begin{array}{lcl} P_1^1 & = & \{\langle \underline{\text{true}},\ \bar{x},\ \bar{y}\rangle\} \\ P_2^1 & = & \underline{\text{test}}(\boldsymbol{\lambda}\,(x,y)\bullet[x \geq y])(P_1^1\ \underline{\text{or}}\ P_3^0) = \{\langle(\bar{x} \geq \bar{y}),\ \bar{x},\ \bar{y}\rangle\} \\ P_3^1 & = & \underline{\text{assign}}(\boldsymbol{\lambda}\,(x,y)\bullet[x-y, y])(P_2^1) = \{\langle(\bar{x} \geq \bar{y}),\ \bar{x}-\bar{y},\ \bar{y}\rangle\} \\ P_4^1 & = & \underline{\text{test}}(\boldsymbol{\lambda}\,(x,y)\bullet[x < y])(P_1^1\ \underline{\text{or}}\ P_3^0) = \{\langle(\bar{x} < \bar{y}),\ \bar{x},\ \bar{y}\rangle\} \end{array}\right.$$

$$\left[\begin{array}{lcl} P_1^2 & = & \{\langle \underline{\text{true}},\ \bar{x},\ \bar{y}\rangle\} \\ P_2^2 & = & \{\langle(\bar{x} \geq \bar{y}),\ \bar{x},\ \bar{y}\rangle, \langle((\bar{x} \geq \bar{y})\ \underline{\text{and}}\ (\bar{x} \geq 2\bar{y})),\ \bar{x}-\bar{y},\ \bar{y}\rangle\} \\ P_3^2 & = & \{\langle(\bar{x} \geq \bar{y}),\ \bar{x}-\bar{y},\ \bar{y}\rangle, \langle((\bar{x} \geq \bar{y})\ \underline{\text{and}}\ (\bar{x} \geq 2\bar{y})),\ \bar{x}-2\bar{y},\ \bar{y}\rangle\} \\ P_4^2 & = & \{\langle(\bar{x} < \bar{y}),\ \bar{x},\ \bar{y}\rangle, \langle(\bar{x} < 2\bar{y}),\ \bar{x}-\bar{y},\ \bar{y}\rangle\} \end{array}\right.$$

After two iterations, we have built the following *symbolic execution tree* (Hantler & King [1976]):


<true, x̄, ȳ>
1
<(x̄ ≥ ȳ), x̄ , ȳ >
2
4
<(x̄ < ȳ),x̄,ȳ >
<(x̄ ≥ ȳ), x̄ − ȳ,ȳ >
3
<(x̄ ≥ ȳ)and(x̄ ≥ 2ȳ), x̄ − ȳ ,ȳ >
2
4
<(x̄ ≥ ȳ)and(x̄ < 2ȳ), x̄ − ȳ ,ȳ >
<(x̄ ≥ ȳ)and(x̄ ≥ 2ȳ), x̄ − 2ȳ , ȳ>
3

We have represented the symbolic context P_i of the program π associated with the point a_i by the set of execution paths associated with the nodes with label i in the symbolic execution tree. We could also represent the symbolic context P_i by the maximal sub-tree of the tree above the leaves of which have label i.

Obviously, if no other specific assumption about $\bar{x}$ and $\bar{y}$ could be made, the next terms of the chaotic iteration sequence would correspond to symbolic trees with greater and greater height, until we would get the tree with infinite height corresponding to $\mathit{lfp}(F_\pi(\boldsymbol{\lambda}\, x \bullet (x = \bar{x})))$.

3.5 BACKWARD DEDUCTIVE SEMANTICS

In order to carry out the semantic analysis of a program π, that is, to study the behavior of the discrete dynamic system defined by π, we have seen in Paragraph 3.1 that we need to consider the ancestors of the final states that satisfy an exit specification $\psi \in \mathcal{P}_n$

where $\mathcal{P}_n = (\mathcal{U}^n \to \mathcal{B})$. Thus, this requires determining $wp(\tau^\star)(\nu_\sigma \ \underline{\text{and}}\ \bar{\psi})$ where $\bar{\psi} = \boldsymbol{\lambda}\,\langle m,\, c\rangle \bullet \psi(m)$. Since the set of states $\mathcal{S}$ is partitioned, Propositions 3.1.3.0.7 and 3.1.4.0.3 show that $wp(\tau^\star)(\nu_\sigma \ \underline{\text{and}}\ \bar{\psi})$ is isomorphic to the least solution of a *system of backward semantic equations* associated with the program π of the form:

$$\begin{cases} P_i &= (\nu_i \ \underline{\text{and}}\ (\nu_\sigma \ \underline{\text{and}}\ \bar{\psi}))\ \underline{\text{or}}\ (\underset{j\in\underline{\text{succ}}(i)}{\underline{\text{OR}}}\ wp(\tau_{ij})(P_j)) \\ i &= 1,\ldots,\alpha,\xi \end{cases}$$

As in Paragraph 3.3, we choose $P_i \in (\mathcal{U}^n \to \mathcal{B})$ rather than $P_i \in \sigma_i(\mathcal{S} \to \mathcal{B})$ since $\sigma_i(\mathcal{S} \to \mathcal{B})$ and $(\mathcal{U}^n \to \mathcal{B})$ are isomorphic by the complete isomorphism $\iota_i = \boldsymbol{\lambda}\,\beta \bullet \{\boldsymbol{\lambda}\,m \bullet [\beta(\langle m,\, a_i\rangle)]\}$ the inverse of which is $\iota_i^{-1} = \boldsymbol{\lambda}\,\beta \bullet \{\boldsymbol{\lambda}\,\langle m,\, c\rangle \bullet [\beta(m)]\}$.
For all $i \neq \sigma$, the predicate $(\nu_i \ \underline{\text{and}}\ \nu_\sigma)$ is false (3.1.4.0.1.(d)), thus:

$$P_i \quad = \quad \underset{j\in\underline{\text{succ}}(i)}{\underline{\text{OR}}}\ \iota_i[wp(\tau_{ij})(\iota_j^{-1}[P_j])]$$

where

$$\begin{aligned} &\iota_i[wp(\tau_{ij})(\iota_j^{-1}[P_j])] \\ &\quad= \iota_i[wp(\tau_{ij})(\boldsymbol{\lambda}\,\langle m,\, c\rangle \bullet [P_j(m)])] \\ &\quad= \iota_i[\boldsymbol{\lambda}\,\langle m_1,\, c_1\rangle \bullet \{\exists\langle m_2,\, c_2\rangle \in \mathcal{S} : \tau_{ij}(\langle m_1,\, c_1\rangle, \langle m_2,\, c_2\rangle)\ \underline{\text{and}}\ P_j(m_2)\}] \\ &\quad= \boldsymbol{\lambda}\,m_1 \bullet \{\exists\langle m_2,\, c_2\rangle \in \mathcal{S} : \tau_{ij}(\langle m_1,\, a_i\rangle, \langle m_2,\, c_2\rangle)\ \underline{\text{and}}\ P_j(m_2)\} \\ &\quad= \boldsymbol{\lambda}\,m_1 \bullet \{\exists\langle m_2,\, c_2\rangle \in \mathcal{S} : \nu_i(\langle m_1,\, a_i\rangle)\ \underline{\text{and}}\ \tau(\langle m_1,\, a_i\rangle, \langle m_2,\, c_2\rangle)\ \underline{\text{and}}\ \nu_j(\langle m_2, \\ &\qquad c_2\rangle)\ \underline{\text{and}}\ P_j(m_2)\} \\ &\quad= \boldsymbol{\lambda}\,m_1 \bullet \{\exists\langle m_2,\, c_2\rangle \in \mathcal{S} : \tau(\langle m_1,\, a_i\rangle, \langle m_2,\, c_2\rangle)\ \underline{\text{and}}\ (c_2 = a_j)\ \underline{\text{and}}\ P_j(m_2)\} \\ &\quad= \boldsymbol{\lambda}\,m_1 \bullet \{\exists m_2 \in \mathcal{U}^n : \bar{\tau}(\langle m_1,\, a_i\rangle) = \langle m_2,\, a_j\rangle\ \underline{\text{and}}\ P_j(m_2)\} \end{aligned}$$

- If a_i is the entry point of an assignment instruction $X := f(X)$, then $\underline{\text{succ}}(i) = \{j\}$ where a_j is the exit point of this instruction, thus:

$$\begin{aligned} P_i &= \boldsymbol{\lambda}\,m_1 \bullet \{\exists m_2 \in \mathcal{U}^n : \bar{\tau}(\langle m_1,\, a_i\rangle) = \langle m_2,\, a_j\rangle\ \underline{\text{and}}\ P_j(m_2)\} \\ &= \boldsymbol{\lambda}\,m_1 \bullet \{\exists m_2 \in \mathcal{U}^n : (m_1 \in \mathit{dom}(f))\ \underline{\text{and}}\ (f(m_1) = m_2)\ \underline{\text{and}}\ P_j(m_2)\} \\ &= \boldsymbol{\lambda}\,m_1 \bullet \{(m_1 \in \mathit{dom}(f))\ \underline{\text{and}}\ P_j(f(m_1))\} \\ &= \underline{\text{assign-1}}(f)(P_j)\ \text{(by definition of } \underline{\text{assign-1}}) \end{aligned}$$

- If a_i is the entry point of the test instruction $p(X)$, then $\underline{\text{succ}}(i) = \{t, f\}$ where a_t and a_f are respectively the $\underline{\text{true}}$ exit point and the $\underline{\text{false}}$ exit point, thus:

$$\begin{aligned}
P_i &= \underset{j \in \{t,f\}}{\underline{\text{OR}}} \ \boldsymbol{\lambda} m_1 \bullet \{\exists m_2 \in \mathcal{U}^n : \bar{\tau}(\langle m_1,\, a_i\rangle) = \langle m_2,\, a_j\rangle \ \underline{\text{and}}\ P_j(m_2)\} \\
&= \quad \boldsymbol{\lambda} m_1 \bullet \{\exists m_2 \in \mathcal{U}^n : (m_1 \in \mathit{dom}(p)) \ \underline{\text{and}}\ p(m_1) \ \underline{\text{and}}\ \langle m_1,\, a_t\rangle = \langle m_2,\, a_t\rangle \ \underline{\text{and}}\ P_t(m_2)\} \\
&\quad \underline{\text{or}} \\
&\quad\quad \boldsymbol{\lambda} m_1 \bullet \{\exists m_2 \in \mathcal{U}^n : (m_1 \in \mathit{dom}(p)) \ \underline{\text{and}}\ \underline{\text{not}}\,(p(m_1)) \ \underline{\text{and}}\ \langle m_1,\, a_f\rangle = \langle m_2,\, a_f\rangle \ \underline{\text{and}}\ P_f(m_2)\} \\
&= \quad \boldsymbol{\lambda} m_1 \bullet \{(m_1 \in \mathit{dom}(p)) \ \underline{\text{and}}\ p(m_1) \ \underline{\text{and}}\ P_t(m_1)\} \\
&\quad \underline{\text{or}} \\
&\quad\quad \boldsymbol{\lambda} m_1 \bullet \{(m_1 \in \mathit{dom}(p)) \ \underline{\text{and}}\ \underline{\text{not}}\,(p(m_1)) \ \underline{\text{and}}\ P_f(m_1)\} \\
&= \underline{\text{test}}(p)(P_t) \ \underline{\text{or}}\ \underline{\text{test}}(\underline{\text{not}}\,(p))(P_f)
\end{aligned}$$

- If a_i is the program point before label L (in a sequence or through an unconditional branching), then $\underline{\text{succ}}(i) = \{j\}$ where a_j is the program point right after L, thus:

$$\begin{aligned}
P_i &= \boldsymbol{\lambda} m_1 \bullet \{\exists m_2 \in \mathcal{U}^n : \bar{\tau}(\langle m_1,\, a_i\rangle) = \langle m_2,\, a_j\rangle \ \underline{\text{and}}\ P_j(m_2)\} \\
&= \boldsymbol{\lambda} m_1 \bullet \{\exists m_2 \in \mathcal{U}^n : \langle m_1,\, a_j\rangle = \langle m_2,\, a_j\rangle \ \underline{\text{and}}\ P_j(m_2)\} \\
&= P_j
\end{aligned}$$

- Since the exit states are stable, we have:

$$\begin{aligned}
\underline{\text{succ}}(\sigma) &= \{j : (\exists e_1, e_2 \in \mathcal{S} : \tau_{\sigma j}(e_1, e_2))\} \\
&= \{j : (\exists e_1, e_2 \in \mathcal{S} : \nu_\sigma(e_1) \ \underline{\text{and}}\ \tau(e_1, e_2) \ \underline{\text{and}}\ \nu_j(e_2))\} \\
&= \{j : (\exists e_1, e_2 \in \mathcal{S} : \nu_\sigma(e_1) \ \underline{\text{and}}\ \tau(e_1, e_2) \ \underline{\text{and}}\ (e_1 = e_2) \ \underline{\text{and}}\ \nu_j(e_2))\} \\
&= \{j : (\exists e_1 \in \mathcal{S} : \nu_\sigma(e_1) \ \underline{\text{and}}\ \tau(e_1, e_1) \ \underline{\text{and}}\ \nu_j(e_1))\} \\
&= \{\sigma\} \text{ (from 3.1.4.0.1.(d))}
\end{aligned}$$

and thus, we get:

$$\begin{aligned}
P_\sigma &= \iota_\sigma[\nu_\sigma \ \underline{\text{and}}\ \iota_\sigma^{-1}(\psi)] \ \underline{\text{or}}\ \iota_\sigma[wp(\tau_{\sigma\sigma})(\iota_\sigma^{-1}[P_\sigma])] \\
&= (\iota_\sigma[\nu_\sigma]\ \underline{\text{and}}\ \psi)\ \underline{\text{or}}\ \boldsymbol{\lambda} m_1 \bullet \{\exists m_2 \in \mathcal{U}^n : \bar{\tau}(\langle m_1, a_\sigma\rangle) = \langle m_2, a_\sigma\rangle\ \underline{\text{and}}\ P_\sigma(m_2)\} \\
&= (\boldsymbol{\lambda} m \bullet (\nu_\sigma(\langle m,\ a_\sigma\rangle))\ \underline{\text{and}}\ \psi)\ \underline{\text{or}}\ \boldsymbol{\lambda} m_1 \bullet \{\exists m_2 \in \mathcal{U}^n : (m_1 = m_2)\ \underline{\text{and}} \\
&\quad P_\sigma(m_2)\} \\
&= \psi\ \underline{\text{or}}\ P_\sigma
\end{aligned}$$

- Last, since the erroneous states are stable, we have the equation $P_\xi = P_\xi$, which we can ignore.

To sum up:

DEFINITION 3.5.0.1 *System of backward semantic equations associated with a program and an exit specification*

Let π be a program with n variables with values in $\mathcal{U}$, α program points $a_1, \ldots, a_\alpha$ (where a_σ is the exit point) and an exit specification $\psi \in \mathcal{P}_n = (\mathcal{U}^n \rightarrow \mathcal{B})$. Let $P_1, \ldots, P_\alpha$ be variables with values in $\mathcal{P}_n$, then the *system of backward semantic equations* $P = B_\pi(\psi)(P)$ *associated with* (π, ψ) is defined by the following rules:

- If a_i is the exit point of the program, then $P_i = \psi$

- If a_i is the entry point of the assignment instruction $X := f(X)$ the exit point of which is a_j, then $P_i = \underline{\text{assign-1}}(f)(P_j)$ where $\underline{\text{assign-1}} = \boldsymbol{\lambda} f \bullet \{\boldsymbol{\lambda} P \bullet [\boldsymbol{\lambda} m \bullet ((m \in dom(f))\ \underline{\text{and}}\ P(f(m)))]\}$

- If a_i is the entry point of the test instruction $p(X)$ the $\underline{\text{true}}$ exit point and the $\underline{\text{false}}$ exit point of which are respectively a_t and a_f, then $P_i = \underline{\text{test}}(p)(P_t)\ \underline{\text{or}}\ \underline{\text{test}}(\underline{\text{not}}(p))(P_f)$

- If a_i is the point before a label followed by program point a_j, then $P_i = P_j$

From 3.1.3.0.5, 3.1.3.0.7, 3.1.4.0.3, 2.8.0.2, and from $lfp(\boldsymbol{\lambda} P_\sigma \bullet [\psi\ \underline{\text{or}}\ P_\sigma]) =$

$lfp(\boldsymbol{\lambda} P_\sigma \bullet [\psi]) = \psi$, we derive the following proposition:

PROPOSITION 3.5.0.2 *Properties of the least fixpoint of the system of backward semantic equations*

The operator $B_\pi(\psi)$ on $(\mathcal{P}_n)^\alpha$ is a complete join- and meet-morphism. The least fixpoint $(P_1, \ldots, P_\alpha)$ of $B_\pi(\psi)$ is such that for all $i \in [1, \alpha]$, we have:

$$P_i \quad = \quad \boldsymbol{\lambda} m_1 \bullet \{\exists m_2 \in \mathcal{U}^n : \tau^\star(\langle m_1,\, a_i\rangle, \langle m_2,\, a_\sigma\rangle) \ \underline{\text{and}}\ \psi(m_2)\}$$

PROPOSITION 3.5.0.3 *Properties of any pre-fixpoint of the system of backward semantic equations*

Let π be a program and $P = B_\pi(\psi)(P)$ be the system of backward semantic equations associated with (π, ψ). For all $i = 1, \ldots, \alpha$ and for all pre-fixpoint P of $B_\pi(\psi)$, we have:

$$\boldsymbol{\lambda} m_2 \bullet \{\exists m_1 \in \mathcal{U}^n : P_i(m_1) \ \underline{\text{and}}\ \tau^\star(\langle m_1,\, a_i\rangle, \langle m_2,\, a_\sigma\rangle)\} \Rightarrow \psi$$

Proof: The proposition follows from 3.1.3.0.7, 3.1.4.0.3, 3.1.5.0.1 with $\beta = (\nu_\sigma \ \underline{\text{and}}\ \iota_\sigma^{-1}(\psi))$, and since any pre-fixpoint of $\boldsymbol{\lambda} P_\sigma \bullet [\psi]$ is also a pre-fixpoint of $\boldsymbol{\lambda} P_\sigma \bullet [P_\sigma \ \underline{\text{or}}\ \psi]$, so that $\iota_\sigma[sp((\tau^\star)_{i\sigma})(\iota_i^{-1}[P_i])] \Rightarrow P_\sigma \Rightarrow \psi$.
End of proof.

PROPOSITION 3.5.0.4 *Properties of the greatest fixpoint of the system of backward semantic equations*

Let π be a program with n variables and α program points $a_1, \ldots, a_\alpha$ and $(Q_1, \ldots, Q_\alpha) = gfp(B_\pi(\boldsymbol{\lambda} X \bullet \underline{\text{true}}))$. Then, $\forall i \in [1, \alpha]$, we have:

$$Q_i \quad = \quad \boldsymbol{\lambda} m_1 \bullet \{\forall m_2 \in \mathcal{U}^n, \underline{\text{not}}\,(\tau^\star(\langle m_1,\, a_i\rangle, \langle m_2,\, \underline{\text{error}}\rangle))\}$$

Proof: We apply Theorems 3.1.3.0.7, 3.1.4.0.3, and 3.1.5.0.2 with $\beta = \nu_\xi$. Therefore, $\underline{\text{not}}(\beta) = \underset{i=1}{\overset{\alpha}{\underline{\text{OR}}}}\, \nu_i$ and the direct decomposition of $\boldsymbol{\lambda}\, \alpha \bullet [\underline{\text{not}}(\nu_\xi)\ \underline{\text{and}}\ wp(\tau)(\alpha)]$ is actually $B_\pi(\boldsymbol{\lambda}\, X \bullet \underline{\text{true}})$, except for the equation defining P_σ, which does not matter since the equations $P_\sigma = \boldsymbol{\lambda}\, X \bullet \underline{\text{true}}$ and $P_\sigma = (\boldsymbol{\lambda}\, X \bullet \underline{\text{true}}\, \underline{\text{and}}\, P_\sigma)$ have the same greatest fixpoint. Therefore:

$$\begin{aligned}
Q_i &= \iota_i[\underline{\text{not}}(wp(\tau^\star)(\nu_\xi))] \\
&= \boldsymbol{\lambda}\, m_1 \bullet \{\underline{\text{not}}(wp(\tau^\star)(\nu_\xi)(\langle m_1,\ a_i\rangle))\} \\
&= \boldsymbol{\lambda}\, m_1 \bullet \{\underline{\text{not}}(\exists \langle m_2,\ c_2\rangle \in \mathcal{S} : \tau^\star(\langle m_1,\ a_i\rangle, \langle m_2,\ c_2\rangle)\ \underline{\text{and}}\ \nu_\xi(\langle m_2,\ c_2\rangle))\} \\
&= \boldsymbol{\lambda}\, m_1 \bullet \{\forall m_2 \in \mathcal{U}^n, \underline{\text{not}}(\tau^\star(\langle m_1,\ a_i\rangle, \langle m_2,\ \underline{\text{error}}\rangle))\}
\end{aligned}$$

End of proof.

Following the definitions given in Paragraphs 3.1.2 and 3.2.2.3, and from the above propositions, we derive:

PROPOSITION 3.5.0.5 *Termination, non-termination, and semantic errors of a program*

Let π be a program with n variables with values in $\mathcal{U}$ and α program points $a_1, \ldots, a_\alpha$. An execution of π starting from program point a_i with initial state of variables $m \in \mathcal{U}^n$

(a) - terminates with no semantic error if and only if $[\mathit{lfp}(B_\pi(\boldsymbol{\lambda}\, m_1 \bullet \underline{\text{true}}))]_i(m)$

(b) - terminates in a state which satisfies the exit condition $\psi \in (\mathcal{U}^n \to \mathcal{B})$ if and only if $[\mathit{lfp}(B_\pi(\psi))]_i(m)$

(c) - does not terminate if and only if $\{[\mathit{gfp}(B_\pi(\boldsymbol{\lambda}\, m_1 \bullet \underline{\text{true}}))]_i(m)\ \underline{\text{and}}\ \underline{\text{not}}([\mathit{lfp}(B_\pi(\boldsymbol{\lambda}\, m_1 \bullet \underline{\text{true}}))]_i)(m)\}$

(d) - leads to a semantic error if and only if $\underline{\text{not}}([\mathit{gfp}(B_\pi(\boldsymbol{\lambda}\, m_1 \bullet \underline{\text{true}}))]_i)(m)$

Propositions 3.1.3.0.5 (for $\theta = \tau^\star$), 3.1.3.0.7, 3.1.4.0.3, and the duality principle imply

PROPOSITION 3.5.0.6 *Conjunction and disjunction of exit specifications*

Let π be a program with n variables, with values in $\mathcal{U}$ and α program points, $\mathcal{P}_n = (\mathcal{U}^n \to \mathcal{B})$, and $P = B_\pi(\psi)(P)$ be the system of backward semantic equations associated with (π, ψ). The functions $\boldsymbol{\lambda}\, \psi \bullet [\mathit{lfp}(B_\pi(\psi))]$ and $\boldsymbol{\lambda}\, \psi \bullet [\mathit{gfp}(B_\pi(\psi))]$ from $\mathcal{P}_n$ to $(\mathcal{P}_n)^\alpha$ are complete join- and meet-morphisms.

3.6 TECHNIQUES FOR THE ANALYSIS OF PROGRAMS BASED ON THE BACKWARD DEDUCTIVE SEMANTICS

We use the results of Paragraph 3.5 to justify Hoare [1969]'s method to verify the partial correctness of programs, and the extension of this method by Dijkstra [1976] to proofs of total correctness. Then, we show how the backward deductive semantics can be used in order to analyze conditions under which a program terminates, does not terminate, or leads to a semantic error. Last, we show how the backward deductive semantics can be used in order to determine, for each program point, the set of all possible states of program variables, during any execution of the program from an initial state which satisfies some given entry specification.

3.6.1 Justification of Hoare's method to verify the partial correctness of a program

Let π be a program with n variables with values in $\mathcal{U}$, α program points $a_1, \ldots, a_\alpha$, and $\mathcal{P}_n = (\mathcal{U}^n \to \mathcal{B})$. A proof of partial correctness of program π for the entry specification ϕ and the exit specification ψ consists in proving that:

$$\{\forall m_2 \in \mathcal{U}^n, [\exists m_1 \in \mathcal{U}^n : \phi(m_1) \ \underline{\text{and}}\ \tau^\star(\langle m_1,\, a_\varepsilon\rangle, \langle m_2,\, a_\sigma\rangle)] \Rightarrow [\psi(m_2)]\}$$

If we can *guess* a pre-fixpoint $P \in (\mathcal{P}_n)^\alpha$ of $B_\pi(\psi)$ such that $\{\phi \Rightarrow P_\sigma\}$, then Proposition 3.5.0.3 shows that π is partially correct for (ϕ, ψ). Therefore, Hoare's method is valid. It is also complete (Cook [1975], Gorelick [1975], Clarke [1977]). Indeed, if π is partially correct for (ϕ, ψ), there always exists $P \in (\mathcal{P}_n)^\alpha$ such that the proof can be performed, that is, such that $\{\{P \Rightarrow B_\pi(\psi)(P)\}\ \underline{\text{and}}\ \{\phi \Rightarrow P_\varepsilon\}\}$. Indeed, choosing $P = \mathit{lfp}(B_\pi(\psi))$ is sufficient since Proposition 3.5.0.5.(b) shows that, if this choice does not work, then the program is incorrect.

3.6.2 Justification of Dijkstra's method to verify the total correctness of a program

A proof of the total correctness of program π for the entry specification ϕ and the exit specification ψ consists in proving that:

$$\{\forall m_1 \in \mathcal{U}^n, [\phi(m_1)] \Rightarrow [\exists m_2 \in \mathcal{U}^n : \tau^\star(\langle m_1,\ a_\varepsilon\rangle, \langle m_2,\ a_\sigma\rangle)\ \underline{\text{and}}\ \psi(m_2)]\}$$

Proposition 3.5.0.2 shows that this amounts to proving that:

$$\{\phi \Rightarrow [\mathit{lfp}(B_\pi(\psi))]_\sigma\}$$

Therefore, the semantics of a program or an instruction is completely defined if we know how to compute $\boldsymbol{\lambda}\,\psi \bullet \{[\mathit{lfp}(B_\pi(\psi))]_\sigma\}$. For instance, the following program pattern with n variables $X = X_1, \ldots, X_n$:

```
while   p(X)   do
     X := f(X) ;
redo ;
```

is, by definition, equivalent to the following program pattern:

```
{1}
     L :
{2}
          if p(X) then
{3}
```

```
                X := f(X) ;
{4}
                goto L ;
            endif ;
{5}
```

The system of backward semantic equations associated with the latter pattern and the exit specification ψ is:

$$\begin{cases} P_1 & = & P_2 \\ P_2 & = & \underline{\text{test}}(p)(P_3)\ \underline{\text{or}}\ \underline{\text{test}}(\underline{\text{not}}\,(p))(P_5) \\ P_3 & = & \underline{\text{assign-1}}(f)(P_4) \\ P_4 & = & P_2 \\ P_5 & = & \psi \end{cases}$$

Proposition 2.8.0.2 allows us to simplify this system into:

$$P_1 \quad = \quad \underline{\text{test}}(p)(\underline{\text{assign-1}}(f)(P_1))\ \underline{\text{or}}\ \underline{\text{test}}(\underline{\text{not}}\,(p))(\psi)$$

the least fixpoint of which is:

$$P_1^{\omega} \quad = \quad \underset{k\in\omega}{\underline{\text{OR}}}((\underline{\text{test}}(p) \circ \underline{\text{assign-1}}(f))^k \circ \underline{\text{test}}(\underline{\text{not}}\,(p)))(\psi)$$

We have recovered the rule for transformation of predicates stated by Dijkstra [1976] for the iteration instruction “while.” Of course, Dijkstra does not use fixpoint equations nor resolution by successive approximations, but he defines the semantics of “while” loops as follows:

$$P_1^{\omega} \quad = \quad \{\exists k \in \omega : I_k\}$$

where

$$\begin{array}{lcl} I_0 & = & (\underline{\text{not}}\,(p)\ \underline{\text{and}}\ \psi) \\ I_k & = & (p\ \underline{\text{and}}\ \underline{\text{assign-1}}(f)(I_{k-1}) \end{array}$$

which is equivalent to the result we have obtained when p is defined for all $X \in \mathcal{U}^n$ since, in this case:

$$\begin{aligned} I_0 &= \underline{\text{test}}(\underline{\text{not}}\,(p))(\psi) \\ I_k &= \underline{\text{test}}(p) \circ \underline{\text{assign-1}}(f)(I_{k-1}) \\ &= ((\underline{\text{test}}(p) \circ \underline{\text{assign-1}}(f))^k \circ \underline{\text{test}}(\underline{\text{not}}\,(p)))(\psi) \end{aligned}$$

3.6.3 Analysis of conditions of termination, non-termination, and erroneous execution of a program based on the backward deductive semantics

Very few articles are devoted to techniques to study the conditions under which a program is incorrect. Proposition 3.5.0.5 is the basis for such a study. For instance, let us consider the following program:

```
{1}
     while   x ≥ 1000   do
{2}
          x := x + α ;
{3}
     redo ;
{4}
```

where x is an integer variable which takes values between b and $(-b+1)$. We will assume that b and $(-b+1)$ denote the largest and smallest integer values that can be represented in a computer and that $b > 1000$. For the sake of simplicity, we will assume that α is an integer constant between 0 and b.

Here is the system of backward semantic equations with four unknown variables $X_1, \ldots, X_4$ corresponding to this program:

$$\left\{ \begin{aligned} X_1 &= \underline{\text{test}}(\boldsymbol{\lambda}\, x \bullet [x \geq 1000])(X_2)\ \underline{\text{or}}\ \underline{\text{test}}(\boldsymbol{\lambda}\, x \bullet [x < 1000])(X_4) \\ X_2 &= \underline{\text{assign-1}}(\boldsymbol{\lambda}\, x \bullet [x + \alpha])(X_3) \\ X_3 &= \underline{\text{test}}(\boldsymbol{\lambda}\, x \bullet [x \geq 1000])(X_2)\ \underline{\text{or}}\ \underline{\text{test}}(\boldsymbol{\lambda}\, x \bullet [x < 1000])(X_4) \\ X_4 &= \boldsymbol{\lambda}\, x \bullet [x = \underline{\text{true}}] \end{aligned} \right.$$

We can simplify this system as follows:

$$\begin{aligned}
X_1 &= \underline{\text{test}}(\boldsymbol{\lambda}\, x \bullet [x \geq 1000])(X_2) \ \underline{\text{or}}\ \underline{\text{test}}(\boldsymbol{\lambda}\, x \bullet [x < 1000])(\boldsymbol{\lambda}\, x \bullet [x = \underline{\text{true}}]) \\
&= \boldsymbol{\lambda}\, x \bullet \{[X_2(x)\ \underline{\text{and}}\ (-b+1 \leq x \leq b)\ \underline{\text{and}}\ (x \geq 1000)]\ \underline{\text{or}}\ [\underline{\text{true}}\ \underline{\text{and}}\ (-b+1 \leq \\
&\quad x \leq b)\ \underline{\text{and}}\ (x < 1000)]\} \\
&= \boldsymbol{\lambda}\, x \bullet \{(-b+1 \leq x \leq b)\ \underline{\text{and}}\ (X_2(x)\ \underline{\text{or}}\ (x < 1000))\} \\
X_2 &= \underline{\text{assign-1}}(\boldsymbol{\lambda}\, x \bullet [x + \alpha])(X_1) \\
&= \boldsymbol{\lambda}\, x \bullet \{(-b+1 \leq x+\alpha \leq b)\ \underline{\text{and}}\ (X_2(x+\alpha)\ \underline{\text{or}}\ (x+\alpha < 1000))\ \underline{\text{and}}\ (-b+1 \leq \\
&\quad x \leq b)\} \\
&= \boldsymbol{\lambda}\, x \bullet \{(-b+1 \leq x \leq b-\alpha)\ \underline{\text{and}}\ (X_2(x+\alpha)\ \underline{\text{or}}\ (x < 1000-\alpha))\}
\end{aligned}$$

Thanks to Theorem 2.8.0.2, we can compute the least solution of the equation which defines X_2 by successive approximations as follows:

$$\begin{aligned}
P_2^0 &= \boldsymbol{\lambda}\, x \bullet \{\underline{\text{false}}\} \\
P_2^1 &= \boldsymbol{\lambda}\, x \bullet \{(-b+1 \leq x \leq b-\alpha)\ \underline{\text{and}}\ (\underline{\text{false}}\ \underline{\text{or}}\ (x < 1000-\alpha))\} \\
&= \boldsymbol{\lambda}\, x \bullet \{-b+1 \leq x \leq 1000-\alpha\} \qquad \text{as } (b > 1000) \\
P_2^2 &= \boldsymbol{\lambda}\, x \bullet \{(-b+1 \leq x \leq b-\alpha)\ \underline{\text{and}}\ ((-b+1 \leq x+\alpha < 1000-\alpha)\ \underline{\text{or}}\ (x < \\
&\quad 1000-\alpha))\} \\
&= \boldsymbol{\lambda}\, x \bullet \{(-b+1 \leq x < 1000-2\alpha)\ \underline{\text{or}}\ (-b+1 \leq x < 1000-\alpha)\} \\
&= \boldsymbol{\lambda}\, x \bullet \{-b+1 \leq x < 1000-\alpha\}
\end{aligned}$$

As the iterations have converged, we have obtained the solution, so that:

$$\begin{aligned}
P_1 &= \boldsymbol{\lambda}\, x \bullet \{(-b+1 \leq x \leq b)\ \underline{\text{and}}\ ((-b+1 \leq x < 1000-\alpha)\ \underline{\text{or}}\ (x < 1000))\} \\
&= \boldsymbol{\lambda}\, x \bullet \{-b+1 \leq x < 1000\}
\end{aligned}$$

Let us now compute the greatest solution of the equation which defines X_2 by successive approximations:

$$\begin{aligned}
Q_2^0 &= \boldsymbol{\lambda}\, x \bullet \{\underline{\text{true}}\} \\
Q_2^1 &= \boldsymbol{\lambda}\, x \bullet \{(-b+1 \leq x \leq b-\alpha)\ \underline{\text{and}}\ (\underline{\text{true}}\ \underline{\text{or}}\ (x < 1000-\alpha))\}
\end{aligned}$$

$$
\begin{aligned}
&= \boldsymbol{\lambda} x \bullet \{-b+1 \le x \le b-\alpha\} \\
Q_2^2 &= \boldsymbol{\lambda} x \bullet \{(-b+1 \le x \le b-\alpha)\,\underline{\text{and}}\,((-b+1 \le x+\alpha \le b-\alpha)\,\underline{\text{or}}\,(x < 1000-\alpha))\} \\
&= \boldsymbol{\lambda} x \bullet \{(-b+1 \le x \le b-2\alpha)\ \underline{\text{or}}\ (-b+1 \le x \le 999-\alpha)\} \\
&= \boldsymbol{\lambda} x \bullet \{-b+1 \le x \le \underline{\text{max}}(b-2\alpha, 999-\alpha)\}
\end{aligned}
$$

Let us perform a proof by induction on the assumption that:

$$
Q_2^k = \boldsymbol{\lambda} x \bullet \{-b+1 \le x \le \underline{\text{max}}(b-k\alpha, 999-\alpha)\}
$$

then, we get:

$$
\begin{aligned}
Q_2^{k+1} &= \boldsymbol{\lambda} x \bullet \{(-b+1 \le x \le b-\alpha)\ \underline{\text{and}}\ ((-b+1 \le x+\alpha \le \underline{\text{max}}(b-k\alpha, 999- \\
&\quad \alpha))\ \underline{\text{or}}\ (x < 1000-\alpha))\} \\
&= \boldsymbol{\lambda} x \bullet \{-b+1 \le x \le \underline{\text{max}}(b-(k+1)\alpha, 999-\alpha)\}
\end{aligned}
$$

and, by passing to the limit, we obtain the greatest fixpoint:

$$
\begin{aligned}
Q_2 &= \boldsymbol{\lambda} x \bullet \{\forall k \ge 1, -b+1 \le x \le \underline{\text{max}}(b-k\alpha, 999-\alpha)\} \\
&= \boldsymbol{\lambda} x \bullet \{(-b+1 \le x \le 1000-\alpha)\ \underline{\text{or}}\ [\forall k \ge 1, (-b+1 \le x \le b-k\alpha)]\} \\
&= \boldsymbol{\lambda} x \bullet \{(-b+1 \le x \le 1000-\alpha)\ \underline{\text{or}}\ ((-b+1 \le x \le b)\ \underline{\text{and}}\ (\alpha = 0))\}
\end{aligned}
$$

After computing this solution, we can deduce that:

$$
\begin{aligned}
Q_1 &= \boldsymbol{\lambda} x \bullet \{[-b+1 \le x \le b]\ \underline{\text{and}}\ [(-b+1 \le x+\alpha < 1000-\alpha)\ \underline{\text{or}}\ ((-b+1 \le \\
&\quad x+\alpha \le b)\ \underline{\text{and}}\ (\alpha = 0))\ \underline{\text{or}}\ (x < 1000)]\} \\
&= \boldsymbol{\lambda} x \bullet \{(-b+1 \le x < 1000)\ \underline{\text{or}}\ ((-b+1 \le x \le b)\ \underline{\text{and}}\ (\alpha = 0))\} \\
&= P_1\ \underline{\text{or}}\ \boldsymbol{\lambda} x \bullet \{(-b+1 \le x \le b)\ \underline{\text{and}}\ (\alpha = 0)\}
\end{aligned}
$$

In the end, Proposition 3.5.0.5 allows us to deduce that, if the variable x has the initial value m, then the program under consideration:

- terminates with no semantic error if and only if $P_1(m)$ is true, which is equivalent to $(-b+1 \le m < 1000)$,

- does not terminate if and only if $(Q_1(m)\,\underline{\text{and}}\,\underline{\text{not}}\,(P_1(m)))$ is true, which is equivalent to $\{[(-b+1 \leq m \leq b)\ \underline{\text{and}}\ (\alpha = 0)]\ \underline{\text{and}}\ [(m < -b+1)\ \underline{\text{or}}\ (m \geq 1000)]\} = \{(1000 \leq m \leq b)\ \underline{\text{and}}\ (\alpha = 0)\}$,
- produces a semantic error if and only if $\underline{\text{not}}\,(Q_1(m))$ is true, which is equivalent to $\{(m < -b+1)\ \underline{\text{or}}\ (b < m)\ \underline{\text{or}}\ ((1000 \leq m)\ \underline{\text{and}}\ (\alpha \neq 0))\}$.

This result does indeed match our intuition since the program is defined only if $-b+1 \leq m \leq b$. Then, if $m < 1000$, the loop is not executed and the program terminates. However, if $m \geq 1000$, the loop is executed infinitely many times when $\alpha = 0$, and terminates with a semantic error due to an overflow when computing the addition $x+\alpha$ when $\alpha \neq 0$.

3.6.4 Use of the backward deductive semantics in order to characterize, for each program point, the set of descendants of the initial states which satisfy an entry specification

Let π be a program with n variables which take values in $\mathcal{U}$ and α program points $a_1, \ldots, a_\alpha$, the entry point of which is a_ε. Let $\phi \in \mathcal{P}_n$ be an entry specification. We wish to determine, for all $i \in [1, \alpha]$,

$$\boldsymbol{\lambda}\, m_2 \bullet \{\exists m_1 \in \mathcal{U}^n : \phi(m_1)\ \underline{\text{and}}\ \tau^\star(\langle m_1,\ a_\varepsilon\rangle, \langle m_2,\ a_i\rangle)\}$$

that is:

$$\boldsymbol{\lambda}\, e_2 \bullet \{\exists e_1 \in \mathcal{S} : \nu_\varepsilon(e_1)\ \underline{\text{and}}\ \bar{\phi}(e_1)\ \underline{\text{and}}\ \tau^\star(e_1, e_2)\ \underline{\text{and}}\ \nu_i(e_2)\}$$

where:

$$\bar{\phi} \quad = \quad \iota^{-1}(\phi)$$

Propositions 3.1.3.0.8 and 3.1.4.0.3 show that we can use a system of backward semantic equations, and compute:

$$\boldsymbol{\lambda}\, \bar{e} \bullet \{\exists e_1 \in \mathcal{S} : \nu_\varepsilon(e_1)\ \underline{\text{and}}\ \bar{\phi}(e_1)\ \underline{\text{and}}\ \mathit{lfp}(\boldsymbol{\lambda}\, \theta \bullet [\boldsymbol{\lambda}\, e \bullet (e = \bar{e})\ \underline{\text{or}}\ \mathit{wp}(\tau)(\theta)])(e_1)\ \underline{\text{and}}\ \nu_i(\bar{e})\}$$

that is:

$$\begin{aligned}&\boldsymbol{\lambda}\,\bar{m}_i \bullet \{\exists(\bar{m}_1,\ldots,\bar{m}_{i-1},\bar{m}_{i+1},\ldots,\bar{m}_\alpha,m_\varepsilon)\in(\mathcal{U}^n)^\alpha : \phi(m_\varepsilon)\ \underline{\text{and}}\ [\mathit{lfp}(\boldsymbol{\lambda}\,X\bullet\\ &\qquad [\boldsymbol{\lambda}\,(m_1,\ldots,m_\alpha)\bullet(m_1=\bar{m}_1,\ldots,m_\alpha=\bar{m}_\alpha)\ \underline{\text{or}}\ B_\pi(\boldsymbol{\lambda}\,x\bullet\underline{\text{true}})(X)])]_\varepsilon(m_\varepsilon)\}\end{aligned}$$

In order to exemplify this on a simple case, let us consider the program below (where a denotes an integer constant):

```
{1}
          x := a ;
{2}
     L :
{3}
          x := x + 1 ;
{4}
          goto L ;
```

Since the variable x takes integer values, we need to solve the following system of backward equations:

$$\left\{\begin{array}{lcl} P_1 & = & \boldsymbol{\lambda}\,x\bullet[(x=\bar{m}_1)\ \underline{\text{or}}\ P_2(a)] \\ P_2 & = & \boldsymbol{\lambda}\,x\bullet[(x=\bar{m}_2)\ \underline{\text{or}}\ P_3(x)] \\ P_3 & = & \boldsymbol{\lambda}\,x\bullet[(x=\bar{m}_3)\ \underline{\text{or}}\ P_4(x+1)] \\ P_4 & = & \boldsymbol{\lambda}\,x\bullet[(x=\bar{m}_4)\ \underline{\text{or}}\ P_3(x)] \end{array}\right.$$

The least solution is:

$$\left[\begin{array}{lcl} P_1 & = & \boldsymbol{\lambda}\,x\bullet[(x=\bar{m}_1)\ \underline{\text{or}}\ (a=\bar{m}_2)\ \underline{\text{or}}\ (\bar{m}_3\geq a)\ \underline{\text{or}}\ (\bar{m}_4\geq a+1)] \\ P_2 & = & \boldsymbol{\lambda}\,x\bullet[(x=\bar{m}_2)\ \underline{\text{or}}\ (\bar{m}_3\geq x)\ \underline{\text{or}}\ (\bar{m}_4\geq x+1)] \\ P_3 & = & \boldsymbol{\lambda}\,x\bullet[(\bar{m}_3\geq x)\ \underline{\text{or}}\ (\bar{m}_4\geq x+1)] \\ P_4 & = & \boldsymbol{\lambda}\,x\bullet[(\bar{m}_3\geq x)\ \underline{\text{or}}\ (\bar{m}_4\geq x)] \end{array}\right.$$

Thus, the set of descendants, at program point $\{i\}$, of the entry states which satisfy the entry condition ϕ is characterized by:

$$\begin{aligned}&\boldsymbol{\lambda}\,\bar{m}_i\bullet\{\exists(\bar{m}_1,\ldots,\bar{m}_{i-1},\bar{m}_{i+1},\ldots,\bar{m}_4,m_\varepsilon)\in\mathcal{N}:\\ &\qquad\qquad \phi(m_\varepsilon)\ \underline{\text{and}}\ [(m_\varepsilon=\bar{m}_1)\ \underline{\text{or}}\ (a=\bar{m}_2)\ \underline{\text{or}}\ (\bar{m}_3\geq a)\ \underline{\text{or}}\ (\bar{m}_4\geq a+1)]\}\end{aligned}$$

so, for $\phi = \boldsymbol{\lambda} x \bullet \underline{\text{true}}$

$$\left[\begin{array}{l} \boldsymbol{\lambda} \bar{m}_1 \bullet \{\underline{\text{true}}\} \\ \boldsymbol{\lambda} \bar{m}_2 \bullet \{\bar{m}_2 = a\} \\ \boldsymbol{\lambda} \bar{m}_3 \bullet \{\bar{m}_3 \geq a\} \\ \boldsymbol{\lambda} \bar{m}_4 \bullet \{\bar{m}_4 \geq a+1\} \end{array}\right.$$

3.7 COMBINATION OF FORWARD AND BACKWARD SEMANTIC ANALYSES OF A PROGRAM

Let $F_\pi(\phi)$ be the system of forward semantic equations associated with a program π and $B_\pi(\psi)$ be the system of backward semantic equations associated with π; let ϕ and ψ denote an entry specification and an exit specification. In Chapter 5, we will look for a characterization of the set of descendants at any point a_i of the entry states which satisfy the entry condition ϕ, and which are also ancestors of the exit states which satisfy the exit condition ψ; thus, we will have to characterize:

$$\boldsymbol{\lambda} e \bullet \{\exists e_1, e_2 \in \mathcal{S} : \nu_\varepsilon(e_1) \ \underline{\text{and}}\ \bar{\phi}(e_1) \ \underline{\text{and}}\ \tau^\star(e_1, e) \ \underline{\text{and}}\ \nu_i(e) \ \underline{\text{and}}\ \tau^\star(e, e_2) \ \underline{\text{and}}\ \nu_\sigma(e_2) \ \underline{\text{and}}\ \bar{\psi}(e_2)\}$$

Following from Propositions 3.3.0.2 and 3.5.0.2, this is equivalent to determining (up to isomorphism):

$$[\mathit{lfp}(F_\pi(\phi)) \ \underline{\text{and}}\ \mathit{lfp}(B_\pi(\psi))]_i$$

Since the least fixpoint of a system of semantic equations may not be computable (Theorem 2.5.6.0.1), we shall simply try to approximate these fixpoints using constructive techniques for approximation of fixpoints that will be described in Chapter 4. In order to apply these techniques in Chapter 5, we will need the proposition below which expresses some properties of $\{\mathit{lfp}(F_\pi(\phi)) \ \underline{\text{and}}\ \mathit{lfp}(B_\pi(\psi))\}$. These properties are immediate consequences of Propositions 3.1.5.0.3, 3.1.4.0.2, and 3.1.4.0.3.

PROPOSITION 3.7.0.1

Let π be a program with n variables with values in $\mathcal{U}$ and α program points. Let $\mathcal{P}_n \in (\mathcal{U}^n \to \mathcal{B})$ and $\phi, \psi \in \mathcal{P}_n$.

(a) - $\{\forall P \in (\mathcal{P}_n), B_\pi(\psi)(P) \ \underline{\text{and}}\ \mathit{lfp}(F_\pi(\phi))\} \Rightarrow B_\pi(\psi)(P \ \underline{\text{and}}\ \mathit{lfp}(F_\pi(\phi)))\}$

$\{\mathit{lfp}(F_\pi(\phi)) \ \underline{\text{and}}\ \mathit{lfp}(B_\pi(\psi))\}$

(b) - $= \mathit{lfp}(F_\pi\{[\mathit{lfp}(B_\pi(\psi))]_\varepsilon \ \underline{\text{and}}\ \phi\})$

(c) - $= \mathit{lfp}(\boldsymbol{\lambda} X \bullet [\mathit{lfp}(F_\pi(\phi)) \ \underline{\text{and}}\ B_\pi(\psi)(X)])$

(d) - $= \mathit{lfp}(\boldsymbol{\lambda} X \bullet [\mathit{lfp}(B_\pi(\phi)) \ \underline{\text{and}}\ F_\pi(\phi)(X)])$

(e) - $= \mathit{lfp}(\boldsymbol{\lambda} X \bullet [\mathit{lfp}(F_\pi(\phi)) \ \underline{\text{and}}\ \mathit{lfp}(B_\pi(\psi)) \ \underline{\text{and}}\ F_\pi(\phi)(X)])$

(f) - $= \mathit{lfp}(\boldsymbol{\lambda} X \bullet [\mathit{lfp}(F_\pi(\phi)) \ \underline{\text{and}}\ \mathit{lfp}(B_\pi(\psi)) \ \underline{\text{and}}\ B_\pi(\psi)(X)])$

3.8 BIBLIOGRAPHY

Many algorithms for analyzing simple properties of programs are based on an operational semantics (for instance Kildall [1973], Wegbreit [1975], Cousot & Cousot [1975b]). It is interesting to understand that these algorithms in fact consist in solving a system of equations associated with the program, and that these equations are obtained by simplifying the semantic equations (Cousot & Cousot [1977a]). Switching from an operational semantics to a deductive semantics is advantageous since it allows reasoning on systems of semantic equations associated with the program instead of reasoning on the program itself, so that we can reduce the semantic analysis of programs to the classic mathematical problem of computing a solution or an approximation of a solution of a system of equations. For instance, there are very few analysis algorithms for recursive procedures. A possible reason for this is that the definition of a semantics for recursive procedures based on the inlining of the body of the procedure at each call (Wegbreit

[1975]) or on the transformation of the program into an iterative program using a recursion stack (Karr [1975]) does not help the intuition. On the contrary, reasoning on systems of equations (Cousot & Cousot [1977d]) allowed us to obtain better results. The difficulties which arise when extending the axiomatic semantics of Hoare [1969] to instructions with a non-contextual syntax (Clint & Hoare [1972]) is another example. In the deductive semantics, the syntactic process to build the system of equations is under context, but the justification of Hoare's method (3.4.1) does not depend on syntactic issues, since it is only based on the properties of the system of semantic equations.

The language we consider to illustrate our method for semantic analysis of programs is simple, but it is general enough to illustrate our results. In fact, Paragraph 3.1 shows well that the methods for semantic analysis of programs can be studied independently from any specific programming language. C. Pair gave us the idea to use the notion of discrete dynamic system. It was also used by Pnueli [1977] in order to formalize the temporal reasonings which arise in the verification of parallel programs.

In order to extend our study to richer programming languages, we would need to consider more complex data-structures (see, for instance, de Bakker [1977a], Finance [1976], Luckham & Suzuki [1976], Pair [1974], Rémy [1974]).

When considering a richer language, defining the semantics of this language would be harder, and the operational method would probably be too cumbersome to allow deriving a deductive semantics in a straightforward way. There exist too many techniques that could replace the operational semantics to list them all here (see for instance Bjørner [1977a,1977b]). The computational semantics would most likely be well-fitted (Finance [1976]); as would the denotational semantics (Tennent [1976] provides an introduction, which can be extended by Stoy [1977]; Milne & Strachey [1976] is the main reference whereas Scott [1976] provides a full bibliography). However, please note that using "continuation" techniques (Strachey & Wadsworth [1974], Milne [1977]) to handle unconditional branching is not mandatory for the denotational semantics, provided that the language does not feature label variables (nor passing functions as parameters).

Following Kleene [1952], the Oxford group introduced the use of fixpoints in order to define the denotational semantics of programming languages. The application to proofs of recursive procedures was immediate (see, for instance, Manna, Ness & Vuillemin [1973]). Even though this is not mandatory (Cousot & Cousot [1977e], Milne [1977]), the techniques to verify iterative programs are most often justified by considering equivalent recursive programs obtained by MacCarthy's transformation (Bird [1976], Clarke [1977], Manna [1974], Vuillemin [1973]) which is sometimes implicitly applied (de Bakker [1977a]). We believe that the use of the deductive semantics is the best one, not only to justify the methods to verify the partial correctness of programs, but also to extend these to verify the total correctness (which was already done for Hoare's method by Dijkstra [1976] and also by Basu & Yeh [1975], de Bakker [1976], Hehner [1976]), for the analysis of incorrect programs (in particular regarding non-termination, which was not studied much (Katz & Manna [1976], Sintzoff [1976a], van Lamsweerde [1977])), and also to justify or discover methods for the approximate semantic analysis of programs (Chapter 5).

CHAPTER 4.

CONSTRUCTIVE METHODS TO APPROXIMATE FIXPOINTS OF MONOTONE OPERATORS ON A COMPLETE LATTICE

4. CONSTRUCTIVE METHODS TO APPROXIMATE FIXPOINTS OF MONOTONE OPERATORS ON A COMPLETE LATTICE

4. CONSTRUCTIVE METHODS TO APPROXIMATE FIXPOINTS OF MONOTONE OPERATORS ON A COMPLETE LATTICE

Let $L(\sqsubseteq)$ be a set that is partially ordered by an ordering relation $\sqsubseteq$ and $x, y \in L$. We say that x is an *under-approximation* of y if and only if $x \sqsubseteq y$. Dually, we say that x is an *over-approximation* of y if and only if $y \sqsubseteq x$.

In the preceding chapter, we have shown that the semantic analysis of a program boils down to computing the extreme fixpoints of systems of equations associated with this program. Since the exact computation of these fixpoints cannot be automatized, we are going to bound these fixpoints between an under- and an over-approximation. This is why we design, in the present chapter, methods to effectively compute under- and over-approximations of extreme fixpoints of monotone operators on a complete lattice. In Paragraphs 4.1 and 4.3, we describe two kinds of approximation methods which will be used together in practice. The methods to approximate the fixpoints of equation systems described in Paragraph 4.3 are based on the idea of simplifying the equations we want to solve. The terms to be neglected cannot be simplified according to numerical criteria, but according to only some algebraic criteria based on the notion of closure operators which are described in Paragraph 4.2. The methods to approximate the fixpoints described in Paragraph 4.1 are based on the idea of accelerating the convergence of the exact iterative methods described in Chapter 2. We will extrapolate the terms of the iteration sequences in order to get an approximation of their limit within a finite amount of steps.

4.1 ITERATIVE ALGORITHMS TO APPROXIMATE FIXPOINTS BY ACCELERATING THE CONVERGENCE BY EXTRAPOLATION

We consider, in Paragraph 4.1.1, some algorithms to approximate the extreme fixpoints of monotone operators on a complete lattice. Then, in Paragraph 4.1.2, we will focus on the particular case of systems of monotone equations.

4.1.1 Approximation of the fixpoints of monotone operators

In order to bound a fixpoint P of an operator f on a complete lattice L, we use the iterative methods from Chapter 2, while accelerating the convergence by extrapolating the terms of the sequence of iterates. In order to avoid iterating forever along cycles of incomparable elements, we use the main idea of Chapter 2, that is to construct a sequence of iterates which is either increasing or decreasing. Moreover, we consider an extrapolation that can be either an over-approximation or an under-approximation of the terms of the sequence, which gives four approximate iterative methods. Then, we show how these methods can be used in order to bound the extreme fixpoints of monotone operators on complete lattices.

Increasing over-approximated iteration sequence

DEFINITION 4.1.1.0.1

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $f \in mon(L \to L)$, then an *increasing over-approximated iteration sequence for f starting from $d \in L$* is a sequence $\langle x^\delta : \delta \in \mu(L)\rangle$ of elements in L such that:

$$\begin{array}{llll}
\text{(a) -} & x^0 & = & d \\
\text{(b) -} & x^\delta & \sqsupseteq & x^{\delta-1} \sqcup f(x^{\delta-1}) \quad \text{if } \delta \text{ is a successor ordinal such that } x^{\delta-1} \notin postfp(f) \\
\text{(c) -} & x^\delta & = & x^{\delta-1} \quad \text{if } \delta \text{ is a successor ordinal such that } x^{\delta-1} \in postfp(f) \\
\text{(d) -} & x^\delta & \sqsupseteq & \bigsqcup_{\alpha<\delta} x^\alpha \quad \text{if } \delta \text{ is a limit ordinal}
\end{array}$$

└

THEOREM 4.1.1.0.2

┌ Let $d \in L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ and $f \in mon(L \to L)$, then an increasing over-approximated iteration sequence for f starting from $d \in L$ is a stationary ascending chain the limit of which is a post-fixpoint of f and an over-approximation of $luis(\boldsymbol{\lambda} x \bullet x \sqcup f(x))(d)$.
└

Proof: Let $\langle x^\delta : \delta \in \mu(L)\rangle$ be an increasing over-approximated iteration sequence for f starting from d. It is an ascending chain, that is, $\{\forall \delta \in \mu(L), \forall \beta \in \mu(L), \{\delta \leq \beta\} \Rightarrow \{x^\delta \sqsubseteq x^\beta\}\}$. Let $\delta \in \mu(L)$ be a given element. Whenever $\beta = \delta$, this lemma holds because $\sqsubseteq$ is reflexive. Let us now assume that the lemma holds for any β such that $\delta \leq \beta < \gamma < \mu(L)$. For any successor ordinal γ, we have $x^\delta \sqsubseteq x^{\gamma-1}$ thanks to the induction hypothesis. If $x^{\gamma-1} \in postfp(f)$, then $x^\delta \sqsubseteq x^{\gamma-1} = x^\gamma$, otherwise $x^\delta \sqsubseteq x^{\gamma-1} \sqsubseteq x^{\gamma-1} \sqcup f(x^{\gamma-1}) \sqsubseteq x^\gamma$. For any limit ordinal γ, we have $x^\delta \sqsubseteq \bigsqcup_{\beta<\gamma} x^\beta \sqsubseteq x^\gamma$. By transfinite induction, we proved that $\langle x^\delta : \delta \in \mu(L)\rangle$ is an ascending chain, which, by definition of $\mu(L)$, cannot be strictly increasing: $\{\exists \varepsilon \in \mu(L) : (\varepsilon + 1) \in \mu(L)$ <u>and</u> $x^\varepsilon = x^{\varepsilon+1}\}$. Since $\varepsilon + 1$ is a successor ordinal, $x^\varepsilon = x^{\varepsilon+1}$ is a post-fixpoint of f. Indeed, $x^{\varepsilon+1}$ could not have been computed by applying the rule 4.1.1.0.1.(b) because $x^\varepsilon = x^{\varepsilon+1} = x^\varepsilon \sqcup f(x^\varepsilon)$ implies $x^\varepsilon = f(x^\varepsilon)$ which would contradict $x^\varepsilon \notin postfp(f)$. So, the rule 4.1.1.0.1.(c) implies that $\langle x^\delta : \delta \in \mu(L)\rangle$ is stationary and there exists an ordinal ε such that $x^{\delta} = x^{\varepsilon}$ for any ordinal $\delta \in \mu(L)$ such that $\delta \geq \varepsilon$. Theorem 2.5.3.0.1 implies that $luis(\boldsymbol{\lambda}\, x \bullet x \sqcup f(x))(d) \sqsubseteq x^{\varepsilon}$ because x^{ε} is a post-fixpoint of f that is greater than d.

End of proof.

Decreasing under-approximated iteration sequence

By the duality principle, we get:

DEFINITION 4.1.1.0.3

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $f \in mon(L \to L)$, then a *decreasing under-approximated iteration sequence for f starting from $d \in L$* is a sequence $\langle x^{\delta} : \delta \in \mu(L)\rangle$ of elements in L such that:

(a) - $x^0 = d$

(b) - $x^{\delta} \sqsubseteq x^{\delta-1} \sqcap f(x^{\delta-1})$ if δ is a successor ordinal such that $x^{\delta-1} \notin prefp(f)$

(c) - $x^{\delta} = x^{\delta-1}$ if δ is a successor ordinal such that $x^{\delta-1} \in prefp(f)$

(d) - $x^{\delta} \sqsubseteq \bigsqcap_{\alpha<\delta} x^{\alpha}$ if δ is a limit ordinal

THEOREM 4.1.1.0.4

Let $d \in L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ and $f \in mon(L \to L)$, then a decreasing under-approximated iteration sequence for f starting from $d \in L$ is a stationary descending chain the limit of which is a pre-fixpoint of f and an under-approximation of $llis(\boldsymbol{\lambda}\, x \bullet x \sqcap f(x))(d)$.

Increasing under-approximated iteration sequence

DEFINITION 4.1.1.0.5

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice, $f \in mon(L \to L)$, then an *increasing under-approximated iteration sequence for f starting from $d \in L$* is a sequence $\langle x^\delta : \delta \in \mu(L)\rangle$ of elements in L such that:

$$\begin{array}{llll} \text{(a) -} & x^0 & = & d & \\ \text{(b) -} & x^{\delta-1} & \sqsubseteq & x^\delta \sqsubseteq f(x^{\delta-1}) \sqcup x^{\delta-1} & \text{if } \delta \text{ is a successor ordinal} \\ \text{(c) -} & x^\delta & = & \bigsqcup\limits_{\alpha<\delta} x^\alpha & \text{if } \delta \text{ is a limit ordinal} \end{array}$$

└

THEOREM 4.1.1.0.6

┌ Let $d \in L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ and $f \in mon(L \to L)$, then an increasing under-approximated iteration sequence for f starting from d is an ascending chain and all terms are under-approximations of $luis(\boldsymbol{\lambda} x \bullet x \sqcup f(x))(d)$. └

Proof: Let $\langle x^\delta : \delta \in \mu(L)\rangle$ be an increasing under-approximated iteration sequence for f starting from d. Let us prove that $\{\forall \delta \in \mu(L), \forall \beta \in \mu(L), \{\delta \leq \beta\} \Rightarrow \{x^\delta \sqsubseteq x^\beta\}\}$. Assume that δ is given, the proof is done by transfinite induction on β. If $\beta = \delta$, then the lemma holds since $\sqsubseteq$ is reflexive. Assume that the lemma holds for any β such that $\delta \leq \beta < \gamma < \mu(L)$. If γ is a successor ordinal, then $x^\delta \sqsubseteq x^{\gamma-1}$ by induction hypothesis and $x^{\gamma-1} \sqsubseteq x^\gamma$ by 4.1.1.0.5.(b). Likewise, if γ is a limit ordinal, then $x^\delta \sqsubseteq \bigsqcup\limits_{\beta<\gamma} x^\beta = x^\gamma$. We proved, by transfinite induction, that $\langle x^\delta : \delta \in \mu(L)\rangle$ is an ascending chain.

Let $\langle y^\delta : \delta \in \mu(L)\rangle$ be the increasing under-approximated iteration sequence starting from d and defined by $\boldsymbol{\lambda} x \bullet x \sqcup f(x)$. We know that $x^0 = y^0 = d$. Assume that for any $\gamma < \delta$ we have $x^\gamma \sqsubseteq y^\gamma$. If δ is a successor ordinal, then, in particular, $x^{\delta-1} \sqsubseteq y^{\delta-1}$. So, by 4.1.1.0.5.(b) and monotonicity, $x^\delta \sqsubseteq f(x^{\delta-1}) \sqcup x^{\delta-1} \sqsubseteq f(y^{\delta-1}) \sqcup y^{\delta-1} = y^\delta$. If δ is a limit ordinal, then we have $x^\delta = \bigsqcup\limits_{\gamma<\delta} x^\gamma \sqsubseteq \bigsqcup\limits_{\gamma<\delta} y^\gamma = y^\delta$. By transfinite induction and Theorem 2.5.3.0.1, we proved that $\forall \delta \in \mu(L), x^\delta \sqsubseteq y^\delta \sqsubseteq luis(\boldsymbol{\lambda} x \bullet x \sqcup f(x))(d)$.
End of proof.

Decreasing over-approximated iteration sequence

By the duality principle, we get:

DEFINITION 4.1.1.0.7

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $f \in mon(L \to L)$, then a *decreasing over-approximated iteration sequence for* f *starting from* $d \in L$ is a sequence $\langle x^\delta : \delta \in \mu(L)\rangle$ of elements in L such that:

$$\begin{array}{llcll}
\text{(a) -} & x^0 & = & d & \\
\text{(b) -} & f(x^{\delta-1}) \sqcap x^{\delta-1} & \sqsubseteq & x^\delta \sqsubseteq x^{\delta-1} & \text{if } \delta \text{ is a successor ordinal} \\
\text{(c) -} & x^\delta & = & \displaystyle\bigsqcap_{\alpha<\delta} x^\alpha & \text{if } \delta \text{ is a limit ordinal}
\end{array}$$

THEOREM 4.1.1.0.8

Let $d \in L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ and $f \in mon(L \to L)$, then a decreasing over-approximated iteration sequence for f starting from d is a descending chain and all terms are over-approximations of $llis(\boldsymbol{\lambda} x \bullet x \sqcap f(x))(d)$.

Chapter 2 has shown that the fixpoints of a monotone operator f on a complete lattice can be computed as the limits of increasing or decreasing iteration sequences for $\boldsymbol{\lambda} x \bullet x \sqcup f(x)$ and $\boldsymbol{\lambda} x \bullet x \sqcap f(x)$ (Theorem 2.5.5.0.2). Having defined in a very general framework some iterative methods with accelerated convergence that enable under- and over- approximations of these limits, we can now describe some fixpoint approximation methods for a monotone operator on a complete lattice.

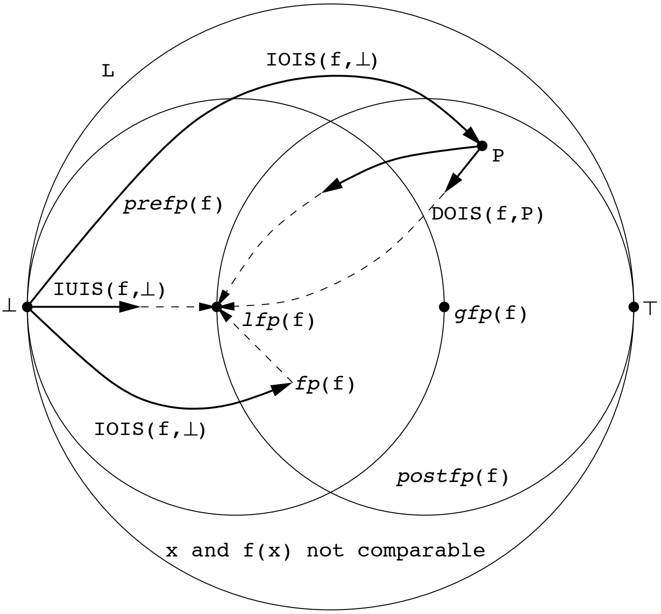
Remark 4.1.1.0.9 Approximation of the extreme fixpoints of a monotone operator on a complete lattice

We want to bound the least fixpoint $lfp(f)$ of a monotone operator $f \in mon(L \to L)$ on the complete lattice $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$. It is always possible to first find d and $D \in L$ that bound $lfp(f)$, since we can choose for instance $d = \bot \sqsubseteq lfp(f) \sqsubseteq \top = D$, and then, to improve this result since $d \sqsubseteq \mathit{luis}(\boldsymbol{\lambda} x \bullet x \sqcup f(x))(d) = \mathit{lfp}(f) \sqsubseteq \mathit{llis}(\boldsymbol{\lambda} x \bullet x \sqcap f(x))(D) \sqsubseteq D$. In practice, we can approach these limits by any term of an increasing under-approximated iteration sequence for f starting from d ($IUIS(f, d)$) and of a decreasing over-approximated iteration sequence for f starting from D ($DOIS(f, D)$), since Theorems 4.1.1.0.6 and 4.1.1.0.8 imply that $d \sqsubseteq IUIS(f, d) \sqsubseteq \mathit{luis}(\boldsymbol{\lambda} x \bullet x \sqcup f(x))(d) = \mathit{lfp}(f) \sqsubseteq \mathit{llis}(\boldsymbol{\lambda} x \bullet x \sqcap f(x))(D) \sqsubseteq DOIS(f, D) \sqsubseteq D$. By stopping after a given iteration rank, the convergence of approximation algorithms can always be enforced.

In order to improve the initial over-approximation D of $\mathit{lfp}(f)$ by the terms of $DOIS(f, D)$, no fixpoint of f shall be jumped over (that is to say the choice of a term x in $DOIS(f, D)$ such that $x \sqsubseteq P = f(P) \sqsubseteq D$ shall not be allowed) for the good reason that it would then be possible to jump below $\mathit{lfp}(f)$ in particular, which leads to an unsound over-approximation. So, if D is greater than another fixpoint P of f, all the terms of $DOIS(f, D)$ are greater than P, which does not lead to an accurate over-approximation of $\mathit{lfp}(f)$. It is better to use the limit of an increasing over-approximated iteration sequence for f starting from an under-approximation d ($IOIS(f, d)$). Indeed, Theorem 4.1.1.0.2 implies that $\mathit{lfp}(f) = \mathit{luis}(\boldsymbol{\lambda} x \bullet x \sqcup f(x))(d) \sqsubseteq IOIS(f, d)$ and it is always possible to enforce the convergence by choosing approximate terms that are large enough, but smaller than D, which ensure that $\mathit{lfp}(f) \sqsubseteq IOIS(f, d) \sqsubseteq D$. If the limit of $IOIS(f, d)$ is not a fixpoint of f, it is possible to improve it by applying, as previously, Theorem 4.1.1.0.8. To sum up, having under- and over-approximations d and D of $\mathit{lfp}(f)$, we propose to improve them as follows:

$$d \sqsubseteq IUIS(f, d) \sqsubseteq \mathit{lfp}(f) \sqsubseteq DOIS(f, D \sqcap IOIS(f, d)) \sqsubseteq D$$

It is always possible to choose $d = \bot$ and $D = \top$, which gives graphically:



Legend:

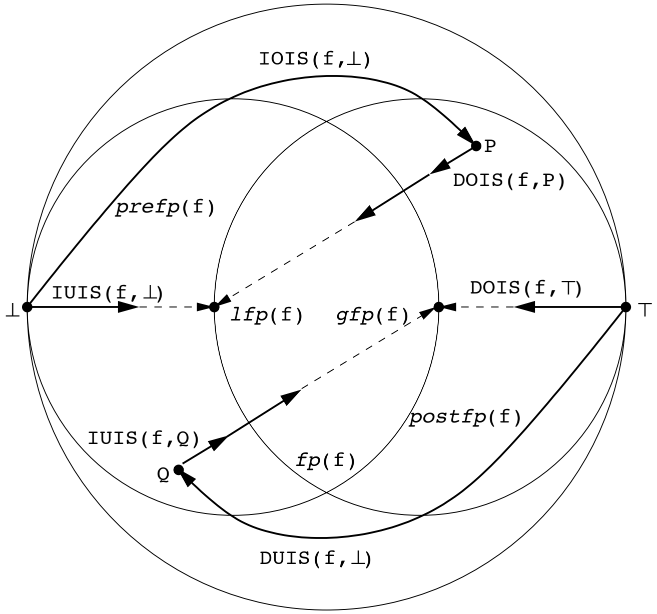
f	:	monotone operator on the complete lattice $L(\sqsubseteq, \perp, \top, \sqcup, \sqcap)$
$\mathit{lfp}(f)$	:	least fixpoint of f
$\mathit{gfp}(f)$	:	greatest fixpoint f
$\mathit{prefp}(f)$	:	$\{x \in L : x \sqsubseteq f(x)\}$
$\mathit{postfp}(f)$	:	$\{x \in L : f(x) \sqsubseteq x\}$

Increasing over-approximated iteration sequence	:	IOIS
Increasing under-approximated iteration sequence	:	IUIS
Decreasing over-approximated iteration sequence	:	DOIS
Decreasing under-approximated iteration sequence	:	DUIS

Dually, given am approximation $d \sqsubseteq \mathit{gfp}(f) \sqsubseteq D$ of the greatest fixpoint of f, we will get a better over-approximation of $\mathit{gfp}(f)$ by any term of a decreasing over-approximated iteration sequence for f starting from D ($DOIS(f, D)$) (Theorem 4.1.1.0.8). In order to get a better under-approximation of $gfp(f)$, we will compute the limit of a decreasing under-approximated iteration sequence for f starting from D ($DUIS(f, D)$) by choosing all terms greater than d (Theorem 4.1.1.0.4). If the result is not a fixpoint, we will improve it by an increasing under-approximated iteration sequence for f. As a summary, we will have:

$$d \sqsubseteq IUIS(f, d \sqcup DUIS(f, D)) \sqsubseteq gfp(f) \sqsubseteq DOIS(f, D) \sqsubseteq D$$

which gives graphically, for the choice $d = \bot$, $D = \top$:



End of remark.

4.1.2 Approximation of the solution of a system of equations

We complete the preceding study of the iterative methods to approximate the fixpoints of monotone operators, based on convergence acceleration by extrapolation, with the case of a system of equations. Our goal is to show how the use of extrapolations can be applied to the methods of chaotic iterations (Theorem 2.9.1.0.2) and, more particularly, to take into account the structure of the equations so that only a minimal amount of extrapolation is performed while ensuring the convergence.

DEFINITION 4.1.2.0.1 *Dependency graph of a system of equations*

Let $X = F(X)$ be a system of equations where $F \in L^n(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ has the following form:

$$\begin{cases} X_1 & = & F_1(X_1, \ldots, X_n) \\ \ldots & & \\ X_n & = & F_n(X_1, \ldots, X_n) \end{cases}$$

A dependency graph associated with this system of equations satisfies the following conditions:

- The graph contains n edges numbered $1, \ldots, n$. Some of these edges are labeled "simple", whereas some others are labeled "head of circuit."

- We say that $f \in (L^n \to L)$ "depends on the i-th component" if and only if $\{\exists x_1, \ldots, x_i, x'_i, \ldots, x_n \in L^{n+1} : f(x_1, \ldots, x_i, \ldots, x_n) \neq f(x_1, \ldots, x'_i, \ldots, x_n)\}$. Then, for all $i, j = 1, \ldots, n$, the target of the edge numbered i is the source of the edge numbered j if F_j depends on the i-th component.

- Any circuit (Berge [1973, p. 8]) of the graph passes through an edge labelled "head of circuit" and the number of edges labelled "head of circuits" is minimal.

Remark 4.1.2.0.2 Choosing the heads of circuits

Let us consider the system of forward semantic equations associated with a program π. Then, the graph of the program π is a dependency graph of this system of equations. We have shown that we can choose as set of head circuits any minimal

set (with respect to set inclusion) in the family of the grids of the hypergraph (Berge [1973, p. 404–405]) such that each edge is the set of outgoing edges from the junction nodes belonging to an elementary circuit of the program graph. As this choice is not unique, various heuristics have been proposed in Cousot & Cousot [1975b, p. 40–46]. In particular, when the graph is reducible as defined in Allen & Cocke [1972] (and according to Knuth [1971], 95% of FORTRAN programs satisfy this property), the edges labeled "head of circuit" are the outgoing edges from the junction nodes that are the heads of the intervals of the program graph.
End of remark.

LEMMA 4.1.2.0.3 *Necessary and sufficient condition to ensure the convergence of a chaotic iteration*

Let $\langle x^\delta : \delta \in \mathrm{Ord} \rangle$ be a chaotic iteration starting from D and defined by $F \in (L^n \to L^n)$ and $\langle J^\delta : \delta \in \mathrm{Ord} \rangle$ (satisfying the assumption 2.9.2.0.1.(a)). Let G be a dependency graph of the system of equations $X = F(X)$.

The sequence $\langle x^\delta : \delta \in \mathrm{Ord} \rangle$ is stationary if and only if the sequence $\langle X_i^\delta : \delta \in \mathrm{Ord} \rangle$ is stationary for all i in $1, \ldots, n$ such that the edge numbered i in G is labeled "head of circuit."

Lemma 4.1.2.0.3 shows that an arbitrary chaotic iteration defined by an arbitrary operator on L^n is stationary if and only if all the "head of circuit" components are stable after a finite number of iterations. So, we see that, in order to accelerate the convergence of a chaotic iteration, it is sufficient to only enforce the convergence of the "head of circuit" components. During the iteration, we will use only one extrapolation method that is formalized by "widening" and "narrowing" operators. Again, we consider increasing and decreasing iterations with over- and under-extrapolation.

Chaotic increasing iteration sequence with upper widening

DEFINITION 4.1.2.0.4 *Upper widening*

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice, then $\bar{\nabla} \in (L \times L \to L)$ is called an *upper widening* if and only if it satisfies the following conditions:

(a) - $\{\forall x, y \in L, x \sqcup y \sqsubseteq x \mathbin{\bar{\nabla}} y\}$

(b) - For any ascending chain $\langle x^\delta : \delta \in \omega \rangle$ of elements in L, the sequence $\langle y^\delta : \delta \in \omega \rangle$ defined as $y^0 = x^0$ and $y^{\delta+1} = y^\delta \mathbin{\bar{\nabla}} x^{\delta+1}$ is a non-strictly ascending chain.

DEFINITION 4.1.2.0.5 *Chaotic increasing iteration sequence with upper widening*

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $F \in mon(L^n \to L)$, then a *chaotic increasing iteration sequence with upper widening starting from $D \in L^n$ and defined by F, the upper widening $\bar{\nabla}$, the dependency graph G of the system of equations $X = F(X)$, and the sequence* $\langle J^\delta : \delta \in \omega \rangle$ (that satisfies the condition 2.9.2.0.1.(a)) is a sequence $\langle X^\delta : \delta \in \omega \rangle$ of elements in L^n defined as:

(a) - $X^0 = D$

(b) - $X_i^\delta = X_i^{\delta-1} \sqcup F_i(X^{\delta-1})$ whenever $\delta > 0, i \in J^\delta$, the edge numbered i in G is "simple" and $\underline{\text{not}}\,(F_i(X^{\delta-1}) \sqsubseteq X_i^{\delta-1})$

(c) - $X_i^\delta = X_i^{\delta-1} \mathbin{\bar{\nabla}} F_i(X^{\delta-1})$ whenever $\delta > 0, i \in J^\delta$, the edge numbered i in G is "head of circuit" and $\underline{\text{not}}\,(F_i(X^{\delta-1}) \sqsubseteq X_i^{\delta-1})$

(d) - $X_i^\delta = X_i^{\delta-1}$ whenever $\delta > 0$ and $((i \notin J^\delta)$ $\underline{\text{or}}$ $(F_i(X^{\delta-1}) \sqsubseteq X_i^{\delta-1}))$

THEOREM 4.1.2.0.6 *Convergence of a chaotic increasing iteration sequence with upper widening*

Let $D \in L^n(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ and $F \in mon(L^n \to L^n)$, then a chaotic increasing iteration sequence $\langle X^\delta : \delta \in \omega \rangle$ with upper widening for F and D is an ascending chain that is stationary after a finite number of steps, and its limit is a post-fixpoint of F. Moreover, this limit is an over-approximation of $luis(\boldsymbol{\lambda} X \cdot X \sqcup F(X))(D)$.

Proof: The sequence $\langle X^\delta : \delta \in \omega \rangle$ is an ascending chain since, for any $i = 1, \ldots, n$, we have either $X_i^\delta = X_i^{\delta-1} \sqcup F_i(X_1^{\delta-1}, \ldots, X_n^{\delta-1}) \sqsupseteq X_i^{\delta-1}$, or $X_i^\delta = X_i^{\delta-1} \bar{\nabla} F_i(X_1^{\delta-1}, \ldots, X_n^{\delta-1}) \sqsupseteq X_i^{\delta-1} \sqcup F_i(X_1^{\delta-1}, \ldots, X_n^{\delta-1}) \sqsupseteq X_i^{\delta-1}$, or lastly $X_i^\delta = X_i^{\delta-1}$.

Let i be an arbitrary element in $\{1, \ldots, n\}$ such that the edge numbered i in G is labeled "head of circuit." Let us consider the sequence $\delta^1, \ldots, \delta^k, \ldots$ such that $\delta^0 = 0$ and such that, for any $k \geq 1$, we have $i \in J^{\delta^k}$ whereas, for any δ satisfying $\delta^{k-1} < \delta < \delta^k$, we have $i \notin J^\delta$. According to Definition 4.1.2.0.5, we know that $X_i^{\delta^{k-1}} = X_i^{\delta^{k-1}+1} = \ldots = X_i^{\delta^k-1}$ and $X_i^{\delta^k} = X_i^{\delta^k-1} \bar{\nabla} F_i(X^{\delta^k-1})$ which implies that $X_i^{\delta^k} = X_i^{\delta^{k-1}} \bar{\nabla} F_i(X^{\delta^{k-1}})$. Since $\langle X^\delta : \delta \in \omega \rangle$ is an ascending chain and F_i is monotone, Definition 4.1.2.0.4.(b) implies that the sequence $X_i^{\delta^0}, \ldots, X_i^{\delta^k}, \ldots$ is an ascending chain which is stationary after a finite number of steps, and so is the sequence $\langle X_i^\delta : \delta \in \omega \rangle$.

According to Lemma 4.1.2.0.3, the ascending chain $\langle X^\delta : \delta \in \omega \rangle$ is stationary after a finite number of steps.

Since the ascending chain $\langle X^\delta : \delta \in \omega \rangle$ is stationary after a finite number ε of steps, we know that for any $i = 1, \ldots, n$, there exists, according to the definition of $\langle J^\delta : \delta \in \omega \rangle$, an integer $\delta \in \omega$ such that $\delta > \varepsilon$ and $i \in J^\delta$, where $X^\varepsilon = X^{\delta-1} = X^\delta$. The computation of X_i^δ did not involve the rule 4.1.2.0.5.(b) nor the rule 4.1.2.0.5.(c) because $X_i^\delta = X_i^{\delta-1} \sqcup F_i(X^{\delta-1})$ and $X_i^\delta = X_i^{\delta-1}$ imply that $F_i(X^{\delta-1}) \sqsubseteq X_i^{\delta-1}$, which is in contradiction with $\underline{\text{not}}(F_i(X^{\delta-1}) \sqsubseteq X_i^{\delta-1})$. Likewise, $X_i^\delta = X_i^{\delta-1} \bar{\nabla} F_i(X^{\delta-1})$ and $X^\delta = X^{\delta-1}$ imply that $X_i^{\delta-1} = X_i^{\delta-1} \bar{\nabla} F_i(X^{\delta-1}) \sqsupseteq X_i^{\delta-1} \sqcup F_i(X^{\delta-1})$, and so, $X_i^{\delta-1} = X_i^{\delta-1} \sqcup F_i(X^{\delta-1})$, which is in contradiction with $\underline{\text{not}}(F_i(X^{\delta-1}) \sqsubseteq X_i^{\delta-1})$. So, we have applied the rule 4.1.2.0.4.(d) in order to get X_i^δ and, since $i \in J^\delta$, we have $F_i(X^{\delta-1}) \sqsubseteq X_i^{\delta-1}$. We deduce that X^ε is a post-fixpoint of F greater than D, so, it is an over-approximation of *luis*$(\boldsymbol{\lambda} X \bullet X \sqcup F(X))(D)$.

End of proof.

Remark 4.1.2.0.7 On the convergence speed and the accuracy of the results

(a) - When the starting point D of the chaotic iteration is a pre-fixpoint of F, then the rule 4.1.2.0.5.(b) can be simplified into $X_i^\delta = F_i(X^{\delta-1})$.

(b) - The convergence does not depend on the iteration order (that is to say on the choice of $\langle J^\delta : \delta \in \omega\rangle$) but the convergence speed does. Moreover, when the widening is not monotone, the accuracy of the result depends on the iteration order as well.

(c) - We can, in the rule 4.1.2.0.5.(c), choose to use a different upper widening $\bar{\nabla}^\delta$ at each application of this rule, provided that $\{\forall x, y \in L, \forall \delta \in \omega, x \sqcup y \sqsubseteq x \mathrel{\bar{\nabla}^\delta} y\}$ and for any sequence $\langle x^\delta : \delta \in \omega\rangle$ of elements in L, the sequence $\langle y^\delta : \delta \in \omega\rangle$ defined as $y^0 = x^0$, $y^{\delta+1} = y^\delta \mathrel{\bar{\nabla}^{\delta+1}} x^{\delta+1}$ is a non-strictly ascending chain.

(d) - In the rule 4.1.2.0.5.(c) the widening enables an extrapolation that is based on two consecutive iterates, so, we have chosen an iterative method with separated steps. It is also possible to base the extrapolation on all the iterates at rank strictly smaller than δ or to choose a method based on related steps using the iterates at rank $\delta - 1, \ldots, \delta - p$.

(e) - Let $M \subseteq L$ such that $F(M^n) \subseteq M^n$. Whenever $D \in M$ but $\mathit{luis}(\boldsymbol{\lambda} X \bullet X \sqcup F(X))(D) \notin M$, it is possible to find an over-approximation of $\mathit{luis}(\boldsymbol{\lambda} X \bullet X \sqcup F(X))(D)$ in M by using a chaotic and increasing iteration with an upper widening $\bar{\nabla}$ such that $\forall x, y \in M, (x \mathrel{\bar{\nabla}} y) \in M$.

End of remark.

Chaotic decreasing iteration sequence with lower widening

By the duality principle, we get:

<u>DEFINITION</u> 4.1.2.0.8 *Lower widening*

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice, then $\underline{\nabla} \in (L \times L \rightarrow L)$ is called a *lower widening* operator if and only if:

(a) - $\{\forall x, y \in L, x \mathbin{\underline{\nabla}} y \sqsubseteq x \sqcap y\}$

(b) - For any descending chain $\langle x^\delta : \delta \in \omega \rangle$ of elements in L, the sequence $\langle y^\delta : \delta \in \omega \rangle$ defined as $y^0 = x^0$ and $y^{\delta+1} = y^\delta \mathbin{\underline{\nabla}} x^{\delta+1}$ is a non-strictly descending chain.

<u>DEFINITION</u> 4.1.2.0.9 *Chaotic decreasing iteration sequence with lower widening*

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $F \in mon(L^n \rightarrow L^n)$, then a *chaotic decreasing iteration sequence with lower widening starting from $D \in L^n$ and defined by F, the lower widening $\underline{\nabla}$, the dependency graph G of the system of equations $X = F(X)$, and the sequence* $\langle J^\delta : \delta \in \omega \rangle$ (that satisfies the condition 2.9.2.0.1.(a)) is a sequence $\langle X^\delta : \delta \in \omega \rangle$ of elements in L^n defined as follows:

(a) - $X^0 = D$

(b) - $X_i^\delta = X_i^{\delta-1} \sqcap F_i(X^{\delta-1})$ whenever $\delta > 0, i \in J^\delta$ and the edge numbered i in G is "simple" and $\underline{\text{not}}\,(X_i^{\delta-1} \sqsubseteq F_i(X^{\delta-1}))$

(c) - $X_i^\delta = X_i^{\delta-1} \mathbin{\underline{\nabla}} F_i(X^{\delta-1})$ whenever $\delta > 0, i \in J^\delta$ and the edge numbered i in G is "head of circuit" and $\underline{\text{not}}\,(X_i^{\delta-1} \sqsubseteq F_i(X^{\delta-1}))$

(d) - $X_i^\delta = X_i^{\delta-1}$ whenever $\delta > 0$ and $((i \notin J^\delta)$ $\underline{\text{or}}$ $(X_i^{\delta-1} \sqsubseteq F_i(X^{\delta-1}))$

<u>THEOREM</u> 4.1.2.0.10 *Convergence of a chaotic decreasing iteration sequence with lower widening*

Let $D \in L^n(\sqsubseteq, \perp, \top, \sqcup, \sqcap)$ and $F \in mon(L^n \rightarrow L^n)$, then a chaotic decreasing iteration sequence with lower widening for F and D is a descending chain that is stationary after a finite number of steps, and its limit is a pre-fixpoint of F which is an under-approximation of $llis(\boldsymbol{\lambda} X \bullet X \sqcap F(X))(D)$.

Chaotic increasing iteration sequence with upper narrowing

DEFINITION 4.1.2.0.11 *Upper narrowing*

Let $L(\sqsubseteq, \perp, \top, \sqcup, \sqcap)$ be a complete lattice, then $\bar{\Delta} \in (L \times L \rightarrow L)$ is called an upper narrowing operator if and only if:

(a) - $\{\forall x, y \in L, x \sqsubseteq x \bar{\Delta} y \sqsubseteq x \sqcup y\}$

(b) - For any ascending chain $\langle x^\delta : \delta \in \omega \rangle$ of elements in L, the sequence $\langle y^\delta : \delta \in \omega \rangle$ defined as $y^0 = x^0$ and $y^{\delta+1} = y^\delta \bar{\Delta} x^{\delta+1}$ is a non-strictly ascending chain.

DEFINITION 4.1.2.0.12 *Chaotic increasing iteration sequence with upper narrowing*

Let $L^n(\sqsubseteq, \perp, \top, \sqcup, \sqcap)$ be a complete lattice and $F \in mon(L^n \rightarrow L^n)$, then a *chaotic increasing iteration sequence with upper narrowing starting from $D \in L^n$ and defined by F, the upper narrowing $\bar{\Delta}$, the dependency graph G of the system of equations $X = F(X)$, and the sequence $\langle J^\delta : \delta \in \omega \rangle$* (that satisfies the condition 2.9.2.0.1.(a)) is a sequence $\langle X^\delta : \delta \in \omega \rangle$ of elements in L^n defined as follows:

(a) - $X^0 = D$

(b) - $X_i^\delta = X_i^{\delta-1} \sqcup F_i(X^{\delta-1})$ whenever $\delta > 0, i \in J^\delta$, and the edge numbered i in G is "simple"

(c) - $X_i^\delta = X_i^{\delta-1} \ \bar{\Delta} \ F_i(X^{\delta-1})$ whenever $\delta > 0, i \in J^\delta$, and the edge numbered i in G is "head of circuit"

(d) - $X_i^\delta = X_i^{\delta-1}$ whenever $\delta > 0$ and $i \notin J^\delta$

THEOREM 4.1.2.0.13 *Convergence of a chaotic increasing iteration sequence with upper narrowing*

Let $D \in L^n(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ and $F \in \mathit{mon}(L^n \rightarrow L^n)$, then a chaotic increasing iteration sequence $\langle X^\delta : \delta \in \omega \rangle$ with upper narrowing for F and D is an ascending chain which is stationary after a finite number of steps, and each term is an underapproximation of $\mathit{luis}(\boldsymbol{\lambda} X \bullet X \sqcup F(X))(D)$.

Proof: The proof that $\langle X^\delta : \delta \in \omega \rangle$ is an ascending chain is completely similar to the proof that was given for Theorem 4.1.2.0.6.

Each term of the increasing chaotic iteration $\langle Y^\delta : \delta \in \omega \rangle$ starting from D and defined by $\boldsymbol{\lambda} X \bullet X \sqcup F(X)$ and $\langle J^\delta : \delta \in \omega \rangle$ is greater than or equal to the corresponding term in $\langle X^\delta : \delta \in \omega \rangle$. Indeed, $X^0 = Y^0 = D$. Let us assume that $X^{\delta-1} \sqsubseteq Y^{\delta-1}$. If $i \notin J^\delta$, then $X^\delta = X^{\delta-1} \sqsubseteq Y^{\delta-1} = Y^\delta$. If $i \in J^\delta$, then, whenever the edge numbered i in G is "simple", we have $X_i^\delta = X_i^{\delta-1} \sqcup F_i(X^{\delta-1}) \sqsubseteq Y_i^{\delta-1} \sqcup F_i(Y^{\delta-1})$ because F_i is monotone; otherwise, the edge numbered i in G is "head of circuit" and then $X_i^\delta = X_i^{\delta-1} \bar{\Delta} F_i(X^{\delta-1}) \sqsubseteq X_i^{\delta-1} \sqcup F_i(X^{\delta-1}) \sqsubseteq Y^{\delta-1} \sqcup F_i(Y^{\delta-1})$ by the condition 4.1.2.0.11.(a), the induction hypothesis, and monotonicity. By induction on δ, we get that $X^\delta \sqsubseteq Y^\delta$ for any $\delta \in \omega$. Moreover, we know that $\langle Y^\delta : \delta \in \omega \rangle$ is an ascending chain and that $Y^\omega \sqsubseteq \mathit{luis}(\boldsymbol{\lambda} X \bullet X \sqcup F(X))(D)$. As a consequence, for any $\delta \in \omega$, we have $X^\delta \sqsubseteq \mathit{luis}(\boldsymbol{\lambda} X \bullet X \sqcup F(X))(D)$.

End of proof.

Remark 4.1.2.0.14

Remark 4.1.2.0.7 holds for chaotic iterations with upper narrowing. In particular, when D is a pre-fixpoint of F, the condition 4.1.2.0.11.(a) can be replaced with $\{\forall x, y \in L, \{x \sqsubseteq y\} \Rightarrow \{x \sqsubseteq x \mathbin{\bar{\Delta}} y \sqsubseteq y\}\}$ whereas the rule 4.1.2.0.12.(b) can be simplified into $X_i^\delta = F_i(X^{\delta-1})$. We rediscover the (dual) conditions of Cousot & Cousot [1977a].

End of remark.

Chaotic decreasing iteration sequence with lower narrowing

By the duality principle, we get:

DEFINITION 4.1.2.0.15 *Lower narrowing*

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice, then $\underline{\Delta} \in (L \times L \rightarrow L)$ is called a lower narrowing operator if and only if:

(a) - $\{\forall x, y \in L, x \sqcap y \sqsubseteq x \underline{\Delta} y \sqsubseteq x\}$

(b) - For any descending chain $\langle x^\delta : \delta \in \omega \rangle$ of elements in L, the sequence $\langle y^\delta : \delta \in \omega \rangle$ defined as $y^0 = x^0$ and $y^{\delta+1} = y^\delta \underline{\Delta} x^{\delta+1}$ is a non-strictly descending chain.

DEFINITION 4.1.2.0.16 *Chaotic decreasing iteration sequence with lower narrowing*

Let $L^n(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $F \in mon(L^n \rightarrow L^n)$, then a *chaotic decreasing iteration sequence with lower narrowing starting from* $D \in L^n$ *and defined by* F*, the lower narrowing* $\underline{\Delta}$*, the dependency graph* G *of the system of equations* $X = F(X)$*, and the sequence* $\langle J^\delta : \delta \in \omega \rangle$ (that satisfies the condition 2.9.2.0.1.(a)) is a sequence $\langle X^\delta : \delta \in \omega \rangle$ of elements in L^n defined as follows:

(a) - $X^0 = D$

(b) - $X_i^\delta = X_i^{\delta-1} \sqcap F_i(X^{\delta-1})$ whenever $\delta > 0, i \in J^\delta$, and the edge numbered i in G is "simple"

(c) - $X_i^\delta = X_i^{\delta-1} \underline{\Delta} F_i(X^{\delta-1})$ whenever $\delta > 0, i \in J^\delta$, and the edge numbered i in G is "head of circuit"

(d) - $X_i^\delta = X_i^{\delta-1}$ whenever $\delta > 0$ and $i \notin J^\delta$

THEOREM 4.1.2.0.17 *Convergence of a chaotic decreasing iteration sequence with lower narrowing*

Let $D \in L^n(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ and $F \in mon(L^n \to L^n)$, then a chaotic decreasing iteration sequence with lower narrowing for F and D is a descending chain that is stationary after a finite number of steps, and each term is an over-approximation of
$\llcorner$ $\mathit{llis}(\boldsymbol{\lambda} X \bullet X \sqcap F(X))(D)$.

Now we can use these different results together with Remark 4.1.1.0.9 in order to under- and over-approximate the extreme solutions of a system of monotone equations on a complete lattice.

We have decided to formulate the iterative methods of fixpoint approximation based on convergence acceleration within a very general context. In particular, the notion of extrapolation has been formalized while keeping only the minimal assumptions, and this provides no conceptual answer to the issue of choosing the best widening and narrowing operations with respect to both the efficiency of the computation and the accuracy of the approximation. Indeed, a good choice of extrapolation shall take into account the particular properties of the lattices and of the systems of equations that we consider, as shown in Chapter 5 on some practical examples.

4.2 CLOSURE OPERATORS ON A COMPLETE LATTICE

The algorithms to approximate the solutions of a system of monotone equations on a complete lattice, based on a simplification of the equations, rely on the fact that, in order to over- (respectively under-)approximate the least fixpoint of $f \in mon(L \to L)$, it is sufficient to find g such that $f \sqsubseteq g$ (respectively $g \sqsubseteq f$) and such that the least fixpoint of g can be computed or over- (respectively under-)approximated because $\mathit{lfp}(f) \sqsubseteq \mathit{lfp}(g)$ (respectively $\mathit{lfp}(g) \sqsubseteq \mathit{lfp}(f)$). In particular, we can build an approximation g of f by using a closure operator on L, which enables the restriction of the space of program properties to a subspace modeling the information we want to collect about programs and abstracting away the information we have *a priori* decided to ignore.

4.2.1 Definition, characterizations, and properties of closure operators

Recall the definitions of upper and lower closure operators given in Paragraph 2.3:

DEFINITION 4.2.1.0.1 *Upper closure operator (Moore [1910])*

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice. A function $\bar{\rho}$ on L is an upper closure operator if and only if:

(a) - $\bar{\rho}$ is monotone $\{\forall x, y \in L, \{x \sqsubseteq y\} \Rightarrow \{\bar{\rho}(x) \sqsubseteq \bar{\rho}(y)\}\}$

(b) - $\bar{\rho}$ is extensive $\{\forall x \in L, x \sqsubseteq \bar{\rho}(x)\}$

(d) - $\bar{\rho}$ is idempotent $\{\forall x \in L, \bar{\rho}(x) = \bar{\rho}(\bar{\rho}(x))\}$

DEFINITION 4.2.1.0.2 *Lower closure operator*

A function $\underline{\rho}$ on $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ is a lower closure operator if and only if $\underline{\rho}$ is monotone, reductive $\{\forall x \in L, \underline{\rho}(x) \sqsubseteq x\}$, and idempotent.

As both notions are dual, we will study in particular upper closure operators. We start by restating some definitions of upper closure operators that are equivalent to the classic Definition 4.2.1.0.1.

CHARACTERIZATION 4.2.1.0.3 *(Monteiro [1945])*

$\rho \in (L \to L)$ is an upper closure operator if and only if $\{\forall x, y \in L, y \sqcup \rho(y) \sqcup \rho(\rho(x)) \sqsubseteq \rho(x \sqcup y)\}$.

CHARACTERIZATION 4.2.1.0.4 *(Iseki [1951])*

$\rho \in (L \to L)$ is an upper closure operator if and only if $\{\forall x, y \in L, x \sqcup \rho(\rho(x)) \sqsubseteq \rho(x \sqcup y)\}$.

CHARACTERIZATION 4.2.1.0.5 *(Morgado [1962b])*

$\rho \in (L \to L)$ is an upper closure operator if and only if $\{\forall x, y \in L, \{\{x \sqsubseteq \rho(y)\} \Leftrightarrow \{\rho(x) \sqsubseteq \rho(y)\}\}\}$.

CHARACTERIZATION 4.2.1.0.6 *(Morgado [1965b])*

$\rho \in (L \rightarrow L)$ is an upper closure operator if and only if $\{\forall f \in (L \rightarrow L), \{\{\boldsymbol{\lambda} x \bullet x \sqsubseteq f\} \Rightarrow \{\boldsymbol{\lambda} x \bullet x \sqsubseteq \rho \circ \rho \sqsubseteq \rho \circ f\}\}\}$.

CHARACTERIZATION 4.2.1.0.7 *(Morgado [1965b])*

$\rho \in (L \rightarrow L)$ is an upper closure operator if and only if $\{\forall f \in (L \rightarrow L), \{\{\boldsymbol{\lambda} x \bullet x \sqsubseteq \rho \circ f\} \Leftrightarrow \{\rho \sqsubseteq \rho \circ f\}\}\}$.

PROPOSITION 4.2.1.0.8 *(Ward [1942])*

An upper closure operator ρ on $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ is a *complete upper quasi-morphism*, that is to say, $\{\forall S \subseteq L, \rho(\sqcup S) = \rho(\sqcup \rho(S))$ <u>and</u> $\sqcap \rho(S) = \rho(\sqcap \rho(S))\}$.

4.2.2 Characterization of a subset of a complete lattice as the image of this lattice by an upper closure operator

Theorems 2.3.0.1 and 2.3.0.3 that we owe to Ward show that the image of a complete lattice by an upper closure operator is a complete lattice. Conversely, Monteiro and Ribeiro have studied the properties of the subsets of a complete lattice that can be described as the image of this complete lattice by an upper closure operator. We recall these results and complete them.

THEOREM 4.2.2.0.1 *(Monteiro & Ribeiro [1942, Thm. 5.2])*

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice. An extensive operator ρ on L is uniquely defined by the set of its fixpoints if and only if ρ is an upper closure operator on L.

DEFINITION 4.2.2.0.2 *Lower Moore family*

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice, we will say that $M \subseteq L$ is a *lower Moore family* of L if and only if, for any x in L, the set $\{y \in M : x \sqsubseteq y\}$ is not empty and has a least element.

LEMMA 4.2.2.0.3 *Characterization of a lower Moore family*

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice, then $M \subseteq L$ is a lower Moore family of L if and only if:

(a) - $\{\top \in L\}$

(b) - $\{\forall S \subseteq M, (\sqcap S) \in M\}$

Proof: Let $M \subseteq L$ be such that $\{\top \in L\}$ and $\{\forall S \subseteq M, (\sqcap S) \in M\}$, then for any x in L, the set $M' = \{y \in M : x \sqsubseteq y\}$ is not empty (since $\top \in M$) and has a least element $\sqcap M'$ (for any y in M', $(\sqcap M') \sqsubseteq y$ and $(\sqcap M') \in M'$).

Conversely, let M be a lower Moore family. For $\top \in L$, the set $\{y \in M : \top \sqsubseteq y\}$ is not empty, so, we have $\top \in M$. Let $S \subseteq M$ such that $(\sqcap S) \notin M$. We consider $M' = \{y \in M : (\sqcap S) \sqsubseteq y\}$ and y_0 the least element of M'. We have $y_0 \in M'$, and so, $(\sqcap S) \sqsubseteq y_0$. But $S \subseteq M'$, so, $y_0 \sqsubseteq (\sqcap S)$. Then, by antisymmetry, $y_0 = (\sqcap S) \in M' \subseteq M$, which contradicts $(\sqcap S) \notin M$. By contradiction, we have $\{\forall S \subseteq M, (\sqcap S) \in M\}$.

End of proof.

THEOREM 4.2.2.0.4 *(Monteiro & Ribeiro [1942, Thm. 5.3])*

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $M \subseteq L$. Then, there exists an upper closure operator ρ on L such that $\rho(L) = M$ if and only if M is a lower Moore family of L (in such a case $\rho = \boldsymbol{\lambda} x \bullet \sqcap \{y \in M : x \sqsubseteq y\}$).

THEOREM 4.2.2.0.5

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $S \subseteq L$. The upper closure operator ρ on L such that $\rho(L)$ is the least lower Moore family that contains S is $\rho = \boldsymbol{\lambda} x \bullet \sqcap \{y \in (S \cup \{\top\}) : x \sqsubseteq y\}$.

Proof: Let us first prove that ρ is an upper closure operator:

- $\forall x \in L, x \sqsubseteq \sqcap\{y \in (S \cup \{\top\}) : x \sqsubseteq y\}$, so, ρ is extensive.

- If $x \sqsubseteq z$ then $\forall y \in (S \cup \{\top\}), \{z \sqsubseteq y\}$ implies $\{x \sqsubseteq y\}$, so, $\sqcap\{y \in (S \cup \{\top\}) : x \sqsubseteq y\} \sqsubseteq \sqcap\{y \in (S \cup \{\top\}) : z \sqsubseteq y\}$, which proves that ρ is monotone.

- Let $y \in (S \cup \{\top\})$ such that $x \sqsubseteq y$. Then, $(\sqcap\{z \in (S \cup \{\top\}) : x \sqsubseteq z\}) \sqsubseteq y$, and so, $\sqcap\{y \in (S \cup \{\top\}) : (\sqcap\{z \in (S \cup \{\top\}) : x \sqsubseteq z\}) \sqsubseteq y \sqsubseteq \sqcap\{y \in (S \cup \{\top\}) : x \sqsubseteq y\}$, thus, $\rho(\rho(x)) \sqsubseteq \rho(x)$, moreover $\rho(x) \sqsubseteq \rho(\rho(x))$, and so, by antisymmetry, ρ is idempotent.

Now:

- For any $x \in S, \rho(x) = x$, so, $S \subseteq \rho(L)$.

- Let θ be an upper closure operator such that $S \subseteq \theta(L)$. $\forall z \in \rho(L), \exists y \in L$ such that $z = \rho(y) = \sqcap\{x \in (S \cup \{\top\}) : y \sqsubseteq x\}$. Let $R = \{t \in L : \theta(t) \in (S \cup \{\top\})\}$. We have $z = \sqcap\{\theta(t) : y \sqsubseteq \theta(t) \ \underline{\text{and}}\ t \in R\}$ which is the meet of some elements of $\theta(L)$, and so, according to 2.3.0.1, $z \in \theta(L)$. We conclude that $\rho(L) \subseteq \theta(L)$.

End of proof.

4.2.3 Lattice of the upper closure operators on a complete lattice and lattice of the induced spaces

The upper closure operators on a complete lattice L are partially ordered by the pointwise ordering on L, that is to say $\{\rho \sqsubseteq \eta\} \Leftrightarrow \{\forall x \in L, \rho(x) \sqsubseteq \eta(x)\}$.

PROPOSITION 4.2.3.0.1 *Characterization of the order on upper closure operators (Ore [1943a,1943b])*

Let ρ and η be two upper closure operators on L, then:

(a) - $\{\rho_1 \sqsubseteq \rho_2\} \Leftrightarrow \{\forall x \in L, \{\rho_2(x) = x\} \Rightarrow \{\rho_1(x) = x\}\}$

(b) - $\{\rho_1 \sqsubseteq \rho_2\} \Leftrightarrow \{\rho_2(L) \subseteq \rho_1(L)\}$

(c) - $\{\rho_1 \sqsubseteq \rho_2\} \Leftrightarrow \{\rho_1 \circ \rho_2 = \rho_2\} \Leftrightarrow \{\rho_2 \circ \rho_1 = \rho_2\}$

It is well-known that the set of all upper closure operators on a complete lattice is a complete lattice for the order $\sqsubseteq$ defined above:

PROPOSITION 4.2.3.0.2 *The complete lattice of the upper closure operators on a complete lattice (Ward [1942])*

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice. The set R of the upper closure operators on L is a complete lattice $(\sqsubseteq, \boldsymbol{\lambda}\, x \bullet x, \boldsymbol{\lambda}\, x \bullet \top, \boldsymbol{\lambda}\, S \bullet \sqcap \{\eta \in R : \{\forall \rho \in S, \rho \sqsubseteq \eta\}\}, \sqcap)$.

We now give a constructive version of this theorem using our constructive version of Tarski's theorem (2.5.5.0.1).

LEMMA 4.2.3.0.3 *Complete lattice of the extensive operators on a complete lattice*

Let $ext = \boldsymbol{\lambda}\, f \bullet (\boldsymbol{\lambda}\, x \bullet x \sqcup f(x))$. Then, ext is an upper closure operator on $L \rightarrow L$ and, for any $f \in (L \rightarrow L)$, $ext(f)$ is the least extensive operator on L that is greater than or equal to f. The set $ext(L \rightarrow L)$ of all extensive operators on L is a complete lattice $(\sqsubseteq, \boldsymbol{\lambda}\, x \bullet x, \boldsymbol{\lambda}\, x \bullet \top, \sqcup, \sqcap)$.

Proof: An operator on L is extensive if and only if $ext(f) = f$. We check that ext is an upper closure operator on $L \rightarrow L$. Moreover, ext is a complete join-morphism, and so, $ext(L \rightarrow L)$ is a complete sub-lattice of $L \rightarrow L$ (Theorem 2.3.0.3). The least element of this lattice is $ext(\boldsymbol{\lambda}\, x \bullet \bot) = \boldsymbol{\lambda}\, x \bullet x \sqcup \bot = \boldsymbol{\lambda}\, x \bullet x$.

End of proof.

LEMMA 4.2.3.0.4 *Complete lattice of the monotone and extensive operators on a complete lattice*

- $mon \circ ext = ext \circ mon$ is an upper closure operator on $L \rightarrow L$
- $mon(L \rightarrow L) \cap ext(L \rightarrow L) = mon \circ ext(L \rightarrow L) = ext \circ mon(L \rightarrow L)$ is a complete sub-lattice $(\sqsubseteq, \boldsymbol{\lambda}\, x \bullet x, \top, \sqcup, \sqcap)$ of $(L \rightarrow L)$

Proof: Since $\sqsubseteq$ is reflexive, for any $f \in (L \rightarrow L)$ and $x \in L$, we have $x \sqsubseteq \sqcup\{y \sqcup f(y) : y \sqsubseteq x\}$ and, as a consequence, $mon(ext(f))(x) = \sqcup\{y \sqcup f(y) : y \sqsubseteq x\} = x \sqcup (\sqcup\{y \sqcup f(y) : y \sqsubseteq x\}) = \sqcup\{x \sqcup y \sqcup f(y) : y \sqsubseteq x\} = \sqcup\{x \sqcup f(y) : y \sqsubseteq x\} = x \sqcup (\sqcup\{f(y) : y \sqsubseteq x\}) = ext(mon(f))(x)$. $mon \circ ext$ is a composition of monotone and extensive operators (2.4.0.3, 4.2.3.0.3), and so, it is monotone and extensive. Since mon and ext commute and are idempotent, $mon \circ ext$ is idempotent. $mon \circ ext$ is an upper closure operator on $(L \rightarrow L)$ and is a complete join-morphism, and so, $mon \circ ext(L \rightarrow L)$ is a complete sub-lattice $(\sqsubseteq, mon \circ ext(\bot) = \boldsymbol{\lambda} x \bullet x, \top, \sqcup, \sqcap)$ on $(L \rightarrow L)$ (Theorem 2.3.0.3). Thus, $\forall f \in (L \rightarrow L), (f \in mon(L \rightarrow L) \cap ext(L \rightarrow L)) \Leftrightarrow (mon(f) = f \ \underline{\text{and}}\ ext(f) = f) \Leftrightarrow (f = mon(ext(f))) \Leftrightarrow (f \in mon \circ ext(L \rightarrow L))$.

End of proof.

THEOREM 4.2.3.0.5 *Complete lattice of the upper closure operators on a complete lattice*

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice, $idem = \boldsymbol{\lambda} f \bullet\ luis(\boldsymbol{\lambda} g \bullet g \circ g)(f)$ and $clos = idem \circ ext \circ mon = idem \circ mon \circ ext$. $clos$ is an upper closure operator on $(L \rightarrow L)$ and, for any $f \in (L \rightarrow L)$, the least upper closure operator on L greater than or equal to f is $clos(f)$. The set $clos(L \rightarrow L)$ of all upper closure operators on L is a complete lattice $(\sqsubseteq, \boldsymbol{\lambda} x \bullet x, \top, \boldsymbol{\lambda} S \bullet\ clos(\sqcup S) = \boldsymbol{\lambda} S \bullet\ luis(\boldsymbol{\lambda} g \bullet g \circ g)(\sqcup S), \sqcap)$.

Proof: According to Definition 4.2.1.0.1, the set of all upper closure operators on the complete lattice L is the set of elements of $(mon(L \rightarrow L) \cap ext(L \rightarrow L))$ that are idempotent, that is to say, fixpoints of $\boldsymbol{\lambda} g \bullet g \circ g$. $\boldsymbol{\lambda} g \bullet g \circ g$ is a monotone operator on $(mon(L \rightarrow L) \cap ext(L \rightarrow L))$. Indeed, let us take f,g such that $f, g \in (mon(L \rightarrow L) \cap ext(L \rightarrow L))$ and $f \sqsubseteq g$. Since $f \in mon(L \rightarrow L)$, we have $f \circ f \sqsubseteq f \circ g$. Since $f \circ g \sqsubseteq g \circ g$, we have $f \circ f \sqsubseteq g \circ g$. Moreover, $fp(\boldsymbol{\lambda} g \bullet g \circ g) = postfp(\boldsymbol{\lambda} g \bullet g \circ g)$ because, if $f \in (mon(L \rightarrow L) \cap ext(L \rightarrow L))$ such that $f \in postfp(\boldsymbol{\lambda} g \bullet g \circ g)$, then $f \circ f \sqsubseteq f$ and $f \sqsubseteq f \circ f$ (since $\boldsymbol{\lambda} x \bullet x \sqsubseteq f$ by extensivity, then $f \sqsubseteq f \circ f$ by monotonicity). Thus, by antisymmetry, $f = f \circ f$. According to Theorem 2.5.3.0.2, the set of all upper closure operators on L is then a complete lattice $\mathit{postfp}(\boldsymbol{\lambda} g \bullet g \circ g)(\sqsubseteq, \mathit{luis}(\boldsymbol{\lambda} g \bullet g \circ g)(\boldsymbol{\lambda} x \bullet x) = \boldsymbol{\lambda} x \bullet x, \top, \boldsymbol{\lambda} S \bullet\ \mathit{luis}(\boldsymbol{\lambda} g \bullet g \circ g)(\sqcup S), \sqcap)$ which is the image of the complete lattice $(\mathit{mon}(L \rightarrow L) \cap \mathit{ext}(L \rightarrow L))(\sqsubseteq, \boldsymbol{\lambda} x \bullet x, \top, \sqcup, \sqcap)$ by the upper closure operator $\mathit{idem} = \boldsymbol{\lambda} f \bullet\ \mathit{luis}(\boldsymbol{\lambda} g \bullet g \sqcup g \circ g)(f)$. But, for any, $f \in \mathit{ext}(L \rightarrow L)$ we have $f \in \mathit{prefp}(\boldsymbol{\lambda} g \bullet g \circ g)$, and so, $\mathit{idem} = \boldsymbol{\lambda} f \bullet\ \mathit{luis}(\boldsymbol{\lambda} g \bullet g \circ g)(f)$.

$\mathit{clos} = \mathit{idem} \circ \mathit{ext} \circ \mathit{mon}$ is monotone and extensive as a composition of monotone and extensive operators. For any $f \in (L \rightarrow L)$, $\mathit{clos}(\mathit{clos}(f)) = \mathit{idem} \circ \mathit{ext} \circ \mathit{mon}(\mathit{clos}(f)) = \mathit{idem} \circ \mathit{ext}(\mathit{clos}(f)) = \mathit{idem}(\mathit{clos}(f)) = \mathit{clos}(f)$ because $\mathit{clos}(f)$ is monotone, extensive, and idempotent. So, clos is idempotent, which finishes the proof that clos is an upper closure operator.

According to Proposition 2.3.0.4.(a), for any $f \in (L \rightarrow L)$, the set $\{\rho \in \mathit{clos}(L \rightarrow L) : f \sqsubseteq \rho\}$ of upper closure operators on L that are greater than or equal to f is not empty and has a least element which is equal to $\mathit{clos}(f)$.

End of proof.

Let us note that an upper closure operator ρ on a complete lattice $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ is monotone (so, $\rho = \mathit{mon}(\rho)$, Theorem 2.4.0.2), extensive (so, $\rho = \mathit{ext}(\rho)$, Theorem 4.2.3.0.3), and idempotent (so, $\rho = \rho \circ \rho$), which implies that $\rho = \mathit{mon}(\mathit{ext}(\rho \circ \rho))$. Conversely, if $\rho = \mathit{mon}(\mathit{ext}(\rho \circ \rho))$, then ρ is monotone, and so, $\mathit{ext}(\rho \circ \rho)$ is monotone as well, which implies that $\rho = \mathit{ext}(\rho \circ \rho)$. Since ρ is extensive, $\rho \circ \rho$ which is extensive as well is a fixpoint of ext, which implies that ρ is idempotent. So, we have the characterization:

If $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ is a complete lattice and $\rho \in (L \rightarrow L)$, then $\{\rho \in \mathit{clos}(L \rightarrow L)\} \Leftrightarrow \{\rho = \boldsymbol{\lambda} x \bullet\ \sqcup \{y \sqcup \rho(\rho(y)) : (y \in L)\ \underline{\text{and}}\ (y \sqsubseteq x)\}\}$

In the following, we give several useful characterizations of clos:

<u>PROPOSITION</u> 4.2.3.0.6

$$clos \quad = \quad \boldsymbol{\lambda} f \bullet (\boldsymbol{\lambda} x \bullet \; luis(\boldsymbol{\lambda} y \bullet \; ext(mon(f))(y))(x))$$

Proof: According to Theorems 2.5.2.0.5 and 2.5.3.0.1, we know that for any $f \in (L \to L)$, $\boldsymbol{\lambda} x \bullet \; luis(\boldsymbol{\lambda} y \bullet y \sqcup mon(f)(y))(x)$ is an upper closure operator on L that is greater than f. Let ρ be an upper closure operator on L such that $f \sqsubseteq \rho$. Then, $ext(mon(f)) \sqsubseteq ext(mon(\rho)) = \rho$. So, $\boldsymbol{\lambda} x \bullet \; luis(\boldsymbol{\lambda} y \bullet \; ext(mon(f))(y))(x) \sqsubseteq \boldsymbol{\lambda} x \bullet \; luis(\rho)(x) = \rho$ since ρ is idempotent. We conclude that for any $f \in (L \to L)$, $\boldsymbol{\lambda} x \bullet \; luis(\boldsymbol{\lambda} y \bullet y \sqcup mon(f)(y))(x)$ is the least upper closure operator on L that is greater than or equal to f, so, it is $clos(f)$.
End of proof.

COROLLARY 4.2.3.0.7

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice, then the set $clos(L \to L)$ of the upper closure operators on L is a complete lattice $(\sqsubseteq, \boldsymbol{\lambda} x \bullet x, \boldsymbol{\lambda} x \bullet \top, \boldsymbol{\lambda} S \bullet \; luis(\sqcup S), \sqcap)$.

We know (Devidé [1964]) that $\boldsymbol{\lambda} x \bullet \; lfp(\boldsymbol{\lambda} y \bullet x \sqcup f(y))$ is an upper closure operator on L when f is monotone. Then, we remark that, according to Theorem 2.5.3.0.1, $\boldsymbol{\lambda} x \bullet \; luis(\boldsymbol{\lambda} y \bullet y \sqcup f(y))(x) = \boldsymbol{\lambda} x \bullet \; luis(\boldsymbol{\lambda} y \bullet x \sqcup f(y))(x) = \boldsymbol{\lambda} x \bullet \; lfp(\boldsymbol{\lambda} y \bullet x \sqcup f(y))$, which gives:

PROPOSITION 4.2.3.0.8

$$clos \quad = \quad \boldsymbol{\lambda} f \bullet (\boldsymbol{\lambda} x \bullet \; lfp(\boldsymbol{\lambda} y \bullet x \sqcup mon(f)(y))(x))$$

COROLLARY 4.2.3.0.9

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice, then the set $clos(L \to L)$ of the upper closure operators on L is a complete lattice $(\sqsubseteq, \boldsymbol{\lambda} x \bullet x, \boldsymbol{\lambda} x \bullet \top, \boldsymbol{\lambda} S \bullet (\boldsymbol{\lambda} x \bullet \; lfp(\boldsymbol{\lambda} y \bullet x \sqcup (\sqcup S)(y))), \sqcap)$.

We deduce immediately from 4.2.3.0.1.(b) the following theorem:

THEOREM 4.2.3.0.10 *(Ward [1942])*

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice, then $clos(L \rightarrow L)$ is isomorphic to the dual of the complete lattice of the lower Moore families of L for the order $\subseteq$ (set inclusion), least element $\{\top\}$, greatest element L, meet $\cap$ (set meet), and join $\boldsymbol{\lambda} S \cdot \{\sqcap P : P \subseteq \cup S\}$.

4.2.4 Composition of upper closure operators on a complete lattice

The composition $\rho_1 \circ \rho_2$ of two upper closure operators ρ_1 and ρ_2 on $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ is monotone and extensive but not necessarily idempotent, so that $\rho_1 \circ \rho_2$ is not necessarily an upper closure operator. So, we give necessary and sufficient conditions in order for the composition of two upper closure operators to be an upper closure operator. After proving:

PROPOSITION 4.2.4.0.1

$$\forall \rho_1, \rho_2 \in clos(L \rightarrow L), \quad clos(\rho_1 \sqcup \rho_2) = clos(\rho_1 \circ \rho_2) = clos(\rho_2 \circ \rho_1)$$

Proof: $\rho_1 \sqsubseteq clos(\rho_1 \sqcup \rho_2)$ and $\rho_2 \sqsubseteq clos(\rho_1 \sqcup \rho_2)$ imply, by monotonicity, that $\rho_1 \circ \rho_2 \sqsubseteq clos(\rho_1 \sqcup \rho_2) \circ clos(\rho_1 \sqcup \rho_2) = clos(\rho_1 \sqcup \rho_2)$. Since $\boldsymbol{\lambda} x \cdot x \sqsubseteq \rho_2$, we get that $\rho_1 \sqsubseteq \rho_1 \circ \rho_2$ and $\rho_2 \sqsubseteq \rho_1 \circ \rho_2$, since ρ_1 is extensive, and so, $\rho_1 \sqcup \rho_2 \sqsubseteq \rho_1 \circ \rho_2 \sqsubseteq clos(\rho_1 \sqcup \rho_2)$. So, $clos(\rho_1 \sqcup \rho_2) \sqsubseteq clos(\rho_1 \circ \rho_2) \sqsubseteq clos(clos(\rho_1 \circ \rho_2)) = clos(\rho_1 \sqcup \rho_2)$.
End of proof.

The following propositions which we owe to Ore [1943b, p. 524–526] are immediate consequences of this proposition:

PROPOSITION 4.2.4.0.2 *(Ore [1943b])*

If ρ_1 and ρ_2 are two upper closure operators on the complete lattice $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$, then $clos(\rho_1 \sqcup \rho_2)$ is the least upper closure operator on L that is greater than or equal to $\rho_1 \circ \rho_2$.

PROPOSITION 4.2.4.0.3 *(Ore [1943b])*

If ρ_1 and ρ_2 are two upper closure operators on the complete lattice $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$, then $\rho_1 \circ \rho_2$ is an upper closure operator on L if and only if $\rho_1 \circ \rho_2 = clos(\rho_1 \sqcup \rho_2)$.

PROPOSITION 4.2.4.0.4 *(Ore [1943b])*

A necessary and sufficient condition in order to ensure that the composition $\rho_1 \circ \rho_2$ of two upper closure operators ρ_1 and ρ_2 on a complete lattice L is an upper closure operator on L is that $\rho_2 \circ \rho_1 \circ \rho_2 = \rho_1 \circ \rho_2$.

THEOREM 4.2.4.0.5 *(Ore [1943b])*

A necessary and sufficient condition in order to ensure that both compositions $\rho_1 \circ \rho_2$ and $\rho_2 \circ \rho_1$ of two upper closure operators ρ_1 and ρ_2 on a complete lattice L are upper closure operators is that ρ_1 and ρ_2 commute (i.e., $\rho_1 \circ \rho_2 = \rho_2 \circ \rho_1$).

In order to define upper closure operators on L, we will often proceed in two steps: first, we will define $\rho \in clos(L \to L)$, then $\eta \in clos(\rho(L) \to \rho(L))$, which gives $\eta \circ \rho \in clos(L \to L)$. This process can be repeated in a cascade.

LEMMA 4.2.4.0.6

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $\rho \in clos(L \to L)$. If $\eta \in clos(\rho(L) \to \rho(L))$, then $\eta \circ \rho \in clos(L \to L)$ and $\rho \sqsubseteq \eta \circ \rho$.

Proof: Since η is extensive, we have $\rho \sqsubseteq \eta \circ \rho$, so, according to 4.2.3.0.1.(c), $\rho \circ \eta \circ \rho = \eta \circ \rho$, then, according to 4.2.4.0.4, we get that $\eta \circ \rho \in clos(L \to L)$.
End of proof.

THEOREM 4.2.4.0.7

Let $L(\sqsubseteq, \perp, \top, \sqcup, \sqcap)$ be a complete lattice and $\rho \in \mathit{clos}(L \to L)$. Then, the complete lattice $\mathit{clos}(\rho(L) \to \rho(L))$ is isomorphic to the complete lattice $\{\theta \in \mathit{clos}(L \to L) : \rho \sqsubseteq \theta\}$ by the complete morphism $\boldsymbol{\lambda}\eta \bullet \eta \circ \rho$, and its inverse morphism is $\boldsymbol{\lambda}\theta \bullet \theta \circ \rho$.

Proof: Let $\eta \in \mathit{clos}(\rho(L) \to \rho(L))$, we have $\eta \circ \rho \circ \rho = \eta \circ \rho$, but, whenever $\eta \circ \rho$ is applied to an element x in $\rho(L)$, we have $\eta(\rho(x)) = \eta(x)$. Conversely, let $\theta \in \mathit{clos}(L \to L)$ such that $\rho \sqsubseteq \theta$, we have $(\theta \circ \rho) \circ \rho = \theta \circ \rho = \theta$, since $\rho \sqsubseteq \theta$ and by Theorem 4.2.3.0.1.(c). Since $(\boldsymbol{\lambda}\eta \bullet \eta \circ \rho) \circ (\boldsymbol{\lambda}\theta \bullet \theta \circ \rho)$ and $(\boldsymbol{\lambda}\theta \bullet \theta \circ \rho) \circ (\boldsymbol{\lambda}\eta \bullet \eta \circ \rho)$ are the identity function, we know that $\boldsymbol{\lambda}\eta \bullet \eta \circ \rho$ is bijective and that $\boldsymbol{\lambda}\theta \bullet \theta \circ \rho$ is its inverse.

Let $\{\theta_i : i \in I\}$ be a family of elements of $\{\theta \in \mathit{clos}(L \to L) : \rho \sqsubseteq \theta\}$, then $(\bigsqcap_{i \in I} \theta_i) \circ \rho = \bigsqcap_{i \in I}(\theta_i \circ \rho)$. The same way, $\mathit{clos}(\bigsqcup_{i \in I} \theta_i) \circ \rho = \mathit{clos}(\bigsqcup_{i \in I} \theta_i \circ \rho)$ (since $\theta_i \circ \rho = \theta_i$ whenever $\rho \sqsubseteq \theta_i$) and $\mathit{clos}(\bigsqcup_{i \in I} \theta_i \circ \rho) \in \mathit{clos}(\rho(L) \to \rho(L))$. Indeed, $\mathit{clos}(\bigsqcup_{i \in I} \theta_i \circ \rho) = \mathit{clos}((\bigsqcup_{i \in I} \theta_i) \circ \rho) = \mathit{clos}((\bigsqcup_{i \in I} \theta_i) \sqcup \rho) \sqsupseteq \mathit{clos}(\rho) = \rho$ and, since $\rho \sqsubseteq \mathit{clos}(\bigsqcup_{i \in I} \theta_i \circ \rho)$, we have $\mathit{clos}(\bigsqcup_{i \in I} \theta_i \circ \rho)(L) \sqsubseteq \rho(L)$. So, it comes that $\mathit{clos}(\bigsqcup_{i \in I} \theta_i \circ \rho) = \mathit{clos}(\bigsqcup_{i \in I} \theta_i \circ \rho) \circ \rho$ and, by transitivity, $\mathit{clos}(\bigsqcup_{i \in I} \theta_i) \circ \rho = \mathit{clos}(\bigsqcup_{i \in I} \theta_i \circ \rho)$. We proved that $\boldsymbol{\lambda}\theta \bullet \theta \circ \rho$ is a bijective function from $\{\theta \in \mathit{clos}(L \to L) : \rho \sqsubseteq \theta\}$ to $\mathit{clos}(\rho(L) \to \rho(L))$ and also a complete morphism. Since the composition of $\boldsymbol{\lambda}\theta \bullet \theta \circ \rho$ and $\boldsymbol{\lambda}\eta \bullet \eta \circ \rho$ is the identity function, we can deduce immediately that $\boldsymbol{\lambda}\eta \bullet \eta \circ \rho$ is a complete isomorphism from $\mathit{clos}(\rho(L) \to \rho(L))$ to $\{\theta \in \mathit{clos}(L \to L) : \rho \sqsubseteq \theta\}$.

End of proof.

4.2.5 Definition of an upper closure operator by a family of principal ideals

PROPOSITION 4.2.5.0.1

Let $L(\sqsubseteq, \perp, \top, \sqcup, \sqcap)$ be a complete lattice and $\rho \in \mathit{clos}(L \to L)$. The set of elements of L that have the same closure by ρ is a complete convex sub-join-semi-lattice of L. (Recall that $S \subseteq L$ is convex if and only if $\{\forall x, y \in S, \forall t \in L, \{x \sqsubseteq t \sqsubseteq y\} \Rightarrow \{t \in S\}\}$.)

Proof: Let $a \in \rho(L)$ and $S_a = \{x \in L : \rho(x) = a\}$. S_a is not empty because $\rho(a) = a$. Let S be a non-empty subset of S_a. Then, $\forall x \in S, x \sqsubseteq \rho(x) = a$, and so, $(\sqcup S) \sqsubseteq a$. This gives, by monotonicity, $\rho(\sqcup S) \sqsubseteq \rho(a) = a$. Then, by monotonicity again, $\rho(\sqcup S) \sqsupseteq \sqcup \rho(S) = a$, and so, $\rho(\sqcup S) = a$. This proves that $(\sqcup S) \in S_a$ and implies that S_a is a complete sub-join-semi-lattice of L. Now, if $x, y \in S_a$ and $x \sqsubseteq t \sqsubseteq y$, we have $a = \rho(x) \sqsubseteq \rho(t) \sqsubseteq \rho(y) = a$. Thus, $\rho(t) = a$ which implies that $t \in S_a$ and proves that S_a is convex.
End of proof.

An *ideal* J of a complete lattice L is a non-empty subset of L such that (i) : $\{\{a \in J, x \in L, x \sqsubseteq a\} \Rightarrow \{x \in J\}\}$, (ii) : $\{\{a \in J, b \in J\} \Rightarrow \{(a \sqcup b) \in J\}\}$. An equivalent characterization of an ideal J of L is that $\{J \subseteq L, J \neq \emptyset, \{\{a \in J, b \in J\} \Leftrightarrow \{(a \sqcup b) \in J\}\}\}$. The meet of an infinite family of ideals of L is an ideal of L.

The *principal ideals* of L are the subsets $\{x \in L, x \sqsubseteq a\}$ for any a in L. The meet of an infinite family of principal ideals of L is again a principal ideal of L. In particular, in a lattice that satisfies the ascending chain condition, any ideal is principal.

A *semi-ideal* I of a complete lattice L is a subset of L such that $\{\forall a \in I, \{x \in L \text{ \underline{and} } x \sqsubseteq a\} \Rightarrow \{x \in I\}\}$. A *dual semi-ideal* J of L is such that $\{\forall a \in J, \{x \in L \text{ \underline{and} } a \sqsubseteq x\} \Rightarrow \{x \in J\}\}$.

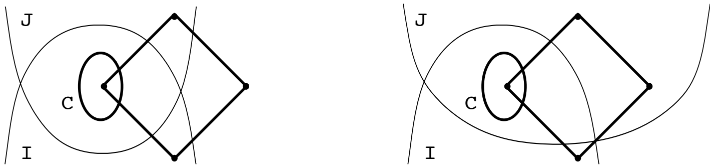
PROPOSITION 4.2.5.0.2

Let I be a principal ideal of a complete lattice $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ and J be a dual semi-ideal of L. Whenever $(I \cap J) \neq \emptyset$, then $I \cap J$ is a complete convex sub-join-semi-lattice of L. Moreover, any complete convex sub-join-semi-lattice C of L can be written as $I \cap J$ where $I = \{x \in L : x \sqsubseteq (\sqcup C)\}$ and $\{x \in L : \{\exists y \in C : y \sqsubseteq x\}\} \subseteq J$.

Proof: Let us set $D = I \cap J$ and let $S \subseteq D$ be such that $S \neq \emptyset$. Then, $S \subseteq I$, and so, $(\sqcup S) \sqsubseteq (\sqcup I)$. This ensures that $(\sqcup S) \in I$. Since S is not empty, $\{\exists x \in S : x \in J$ <u>and</u> $x \sqsubseteq (\sqcup S)\}$, and so, $(\sqcup S) \in J$. Thus, $(\sqcup S) \in D$. So, D is a complete sub-join-semi-lattice of L. If $x, y \in D$ and $t \in L$ are such that $x \sqsubseteq t \sqsubseteq y$, then $t \sqsubseteq y$ and $y \in I$ imply $t \in I$. $x \sqsubseteq t$ and $x \in I$ also imply $t \in J$, thus, $t \in D$. This ensures that D is convex.

Let C be a complete convex sub-join-semi-lattice of L. Let us set $I = \{x \in L : x \sqsubseteq (\sqcup C)\}$. The set I is a principal ideal of L. Let us set $J = \{x \in L : \{\exists y \in C : y \sqsubseteq x\}\}$. The set J is a dual semi-ideal of L. Then, $C \subseteq (I \cap J)$ since $\forall x \in C, x \sqsubseteq x \sqsubseteq (\sqcup C)$. If $t \in (I \cap J)$, then $t \in I$, and so, $t \sqsubseteq (\sqcup C)$ with $(\sqcup C) \in C$. Moreover, $t \in J$, so that $\exists c \in C$ such that $c \sqsubseteq t$. Since C is convex, $t \in C$, which ensures that $C = I \cap J$.

Let us assume now that C can be written as $C = I_1 \cap J_1$. Since $C \subseteq I_1$, we can deduce that $\{x \in L : x \sqsubseteq (\sqcup C)\} \subseteq I_1$. Let $a \in I_1$ and x be an arbitrary element in C. Then, $(a \sqcup x) \in I_1$. Moreover, $(a \sqcup x) \sqsupseteq x$ and $x \in J_1$ imply that $(a \sqcup x) \in J_1$ and $(a \sqcup x) \in (I_1 \cap J_1) = C$. So, $a \sqsubseteq (a \sqcup x) \sqsubseteq (\sqcup C) \in \{x \in L : x \sqsubseteq (\sqcup C)\}$. Thus, $I_1 \subseteq \{x \in L : x \sqsubseteq (\sqcup C)\}$ and, by antisymmetry, $I_1 = \{x \in L : x \sqsubseteq (\sqcup C)\}$. Likewise, since $C \subseteq J_1$, we get that $J = \{x \in L : \{\exists y \in C : y \sqsubseteq x\}\} \subseteq J_1$. However, the choice of J_1 is not unique, as shown in the following counter-example:



End of proof.

THEOREM 4.2.5.0.3

Let $\{I_i : i \in \Delta\}$ be a family of principal ideals of the complete lattice $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$. Then, $\boldsymbol{\lambda} x \bullet \sqcup (\cap\{J \in (\{L\} \cup \{I_i : i \in \Delta\}) : x \in J\})$ is the upper closure operator that is generated by $\{I_i : i \in \Delta\}$.

Proof: For $x \in L$, let us set $S_x = \cap\{J \in (\{L\} \cup \{I_i : i \in \Delta\}) : x \in J\}$ and $\rho(x) = (\sqcup S_x)$. Since $x \in L$, we have $x \in S_x$ and $x \sqsubseteq (\sqcup S_x)$. This ensures that ρ is extensive. If $x \sqsubseteq y$ then $S_x \subseteq S_y$ since $y \in J, x \in L$, and $x \sqsubseteq y$ imply $x \in J$ for any ideal J of L. So, $\rho(x) = (\sqcup S_x) \sqsubseteq (\sqcup S_y) = \rho(y)$, which ensures that ρ is monotone. For any $J \in (\{L\} \cup \{I_i : i \in \Delta\})$, $x \in J$ implies that $(\sqcup S_x) \in J$, since $x \in J$ implies that $S_x \subseteq J$. So, $\cap\{J \in (\{L\} \cup \{I_i : i \in \Delta\}) : (\sqcup S_x) \in J\}$ is a subset of $\cap\{J \in (\{L\} \cup \{I_i : i \in \Delta\}) : x \in J\}$, and so, $\rho(\rho(x)) \sqsubseteq \rho(x)$. Moreover, $\rho(x) \sqsubseteq \rho(\rho(x))$ since ρ is extensive and monotone, and, by antisymmetry, we conclude that ρ is idempotent.

End of proof.

COROLLARY 4.2.5.0.4

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $\rho \in \mathit{clos}(L \to L)$, then ρ is equal to the upper closure operator on L that is generated by $\{\{y \in L : y \sqsubseteq a\} : a \in \rho(L)\}$.

Proof: Since $\top \in \rho(L)$, we have $\boldsymbol{\lambda} x \bullet \sqcup (\cap\{J \in \{\{y \in L : y \sqsubseteq a\} : a \in \rho(L)\} : x \in$

$J\}) = \boldsymbol{\lambda} x \bullet \sqcup (\cap\{\{y \in L : y \sqsubseteq a\} : a \in \rho(L)$ $\underline{\text{and}}$ $x \sqsubseteq a\}) = \boldsymbol{\lambda} x \bullet \sqcap \{a \in \rho(L)\} : x \sqsubseteq a\} = \rho(x)$.

End of proof.

4.2.6 Definition of an upper closure operator by a join-complete congruence relation

DEFINITION 4.2.6.0.1 *Join-complete congruence relation*

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice, then a *join-complete congruence relation* on L is an equivalence relation θ that satisfies both the *substitution property for the join* $\{\forall x, y, u \in L, \{\{x \equiv y(\theta)\} \Rightarrow \{(x \sqcup u) \equiv (y \sqcup u)(\theta)\}\}\}$ and the *completeness property* $\{\forall x \in L, x \equiv \sqcup([x]\theta)(\theta)\}$ where $[x]\theta = \{y \in L : x \equiv y(\theta)\}$.

The substitution property for the join can be written equivalently as $\{\forall x_1, y_1, x_2, y_2 \in L, \{\{x_1 \equiv y_1(\theta)\}$ $\underline{\text{and}}$ $\{x_2 \equiv y_2(\theta)\}\} \ \Rightarrow \{(x_1 \sqcup x_2) \equiv (y_1 \sqcup y_2)(\theta)\}\}$.

PROPOSITION 4.2.6.0.2

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $\rho \in \mathit{clos}(L \to L)$, then the relation (ρ) defined as $\{x \equiv y(\rho)\} \Leftrightarrow \{\rho(x) = \rho(y)\}$ is a join-complete congruence relation.

Proof: For any $\rho \in (L \to L)$, it is well-known that (ρ) is an equivalence relation. Let us assume that $\rho \in \mathit{clos}(L \to L)$ and $x, y, u \in L$, then $\rho(x) = \rho(y)$ implies that $\rho(x \sqcup u) = \rho(\rho(x) \sqcup \rho(u)) = \rho(\rho(y) \sqcup \rho(u)) = \rho(y \sqcup u)$ since ρ is a complete upper quasi-morphism (4.2.1.0.8). As a consequence, $x \equiv y(\rho)$ implies that $(x \sqcup u) \equiv (y \sqcup u)(\rho)$. Moreover, $\forall y \in [x](\rho), \rho(y) = \rho(x)$, so that $\sqcup\{\rho(y) : y \in [x](\rho)\} = \rho(x)$. Now, $\rho(\sqcup([x](\rho))) = \rho(\sqcup\rho([x](\rho))) = \rho(\sqcup\{\rho(y) : y \in [x](\rho)\}) = \rho(\rho(x)) = \rho(x)$, which gives $x \equiv \sqcup([x](\rho))(\rho)$.

End of proof.

COROLLARY 4.2.6.0.3

Let $\rho \in clos(L \to L)$, then $\rho = \boldsymbol{\lambda} x \bullet \sqcup ([x](\rho))$.

THEOREM 4.2.6.0.4

Let θ be a join-complete congruence relation on L, then, for any x in L, $[x]\theta$ is a complete convex sub-join-semi-lattice. Moreover, $\boldsymbol{\lambda} x \bullet \sqcup ([x]\theta) \in clos(L \to L)$.

Proof: Let $x \in L$ and $y, z \in ([x]\theta)$. Let $t \in L$ such that $y \sqsubseteq t \sqsubseteq z$. Since $y \equiv z(\theta)$, we have $(y \sqcup t) \equiv (z \sqcup t)(\theta)$. Moreover, $t = y \sqcup t$ and $z \sqcup t = z$, so, $t \equiv z(\theta)$ and $z \equiv x(\theta)$. This implies that $t \equiv x(\theta)$ and $t \in ([x]\theta)$, so, $[x]\theta$ is a convex subset of L.

Let $S \subseteq ([x]\theta)$ be such that $S \neq \emptyset$, $\exists y \in S$ such that $y \equiv x(\theta)$. Moreover, $\sqcup([x]\theta) \equiv x(\theta)$ and $y \sqsubseteq \sqcup S \sqsubseteq \sqcup([x]\theta)$. So, $(\sqcup S) \in ([x]\theta)$. This ensures that $[x]\theta$ is a complete sub-join-semi-lattice of L.

Let us set $\rho = \boldsymbol{\lambda} x \bullet \sqcup ([x]\theta)$. ρ is extensive, since $x \in ([x]\theta)$ implies that $x \sqsubseteq \sqcup([x]\theta) = \rho(x)$. Moreover, $\sqcup([x]\theta) \in ([x]\theta)$, so that $\rho(\rho(x)) = \rho(\sqcup([x]\theta)) = \sqcup[\sqcup([x]\theta)]\theta = \sqcup([x]\theta) = \rho(x)$. Thus, ρ is idempotent. If $x \sqsubseteq y$, then $y = x \sqcup y \equiv (\rho(x) \sqcup \rho(y))(\theta)$ by the substitution property for the join. So, $y \equiv \rho(y)(\theta)$ and, by transitivity, we have that $\rho(x) \sqcup \rho(y)$ belongs to $[\rho(y)]\theta$. Thus, as a consequence, $(\rho(x) \sqcup \rho(y)) \sqsubseteq \sqcup([\rho(y)]\theta) = \rho(y)$. Since $\rho(y) \sqsubseteq \rho(x) \sqcup \rho(y)$, we conclude by antisymmetry that $\rho(x) \sqsubseteq \rho(y)$. Thus, ρ is monotone.

End of proof.

Tedious computations are sometimes necessary to prove that a given binary relation is a join-complete congruence relation. Computations are often made easier thanks to the following theorem (that we can compare to the theorem that has been stated on congruences in Grätzer & Schmidt [1958]).

PROPOSITION 4.2.6.0.5

A reflexive and symmetric binary relation θ on a complete lattice $L(\sqsubseteq, \perp, \top, \sqcup, \sqcap)$ is a join-complete congruence relation if and only if the following three properties are satisfied for any $x, y, z, t \in L$ and $S \subseteq L$:

(a) - $\{x \equiv y(\theta)\} \;\Leftrightarrow\; \{\exists u \in L : (x \sqcup y) \sqsubseteq u \; \underline{\text{and}} \; u \equiv x(\theta) \; \underline{\text{and}} \; u \equiv y(\theta)\}$

(b) - $\{x \sqsubseteq y \sqsubseteq z \; \underline{\text{and}} \; x \equiv y(\theta) \; \underline{\text{and}} \; y \equiv z(\theta)\} \;\Rightarrow\; \{x \equiv z(\theta)\}$

(c) - $\{x \sqsubseteq y \; \underline{\text{and}} \; x \equiv y(\theta)\} \;\Rightarrow\; \{(x \sqcup t) \equiv (y \sqcup t)(\theta)\}$

$\llcorner$

Proof: A join-complete congruence relation satisfies (a) since $\{x \equiv y(\theta)\} \Rightarrow \{(x \sqcup y) \equiv x(\theta) \; \underline{\text{and}} \; (x \sqcup y) \equiv y(\theta)\}$ and $\{\exists u \in L : u \equiv x(\theta) \; \underline{\text{and}} \; u \equiv (\sqcup S)(\theta)\} \Rightarrow \{x \equiv (\sqcup S)(\theta)\}$. (b) holds by transitivity and (c) is true because of the reflexivity and the substitution property for $\sqcup$.

Conversely, let θ be a reflexive and symmetric binary relation that satisfies (a), (b), and (c).

- θ is transitive, since $x \equiv y(\theta)$ and $y \equiv z(\theta)$ and (a) imply that $\{\exists u, u' \in L : x \sqsubseteq u, y \sqsubseteq u, z \sqsubseteq u', y \sqsubseteq u', x \equiv u(\theta), y \equiv u(\theta), y \equiv u'(\theta), z \equiv u'(\theta)\}$. Then, according to (c), we have that $u' = (y \sqcup u') \equiv (u \sqcup u')(\theta)$ and $u = (y \sqcup u) \equiv (u \sqcup u')(\theta)$. According to (b), $x \sqsubseteq u \sqsubseteq (u \sqcup u')$, so, $x \equiv (u \sqcup u')(\theta)$. Likewise, $z \equiv (u \sqcup u')(\theta)$. So, $(x \sqcup z) \equiv (u \sqcup u')(\theta)$ and, thus, according to (a), we conclude that $x \equiv z(\theta)$.

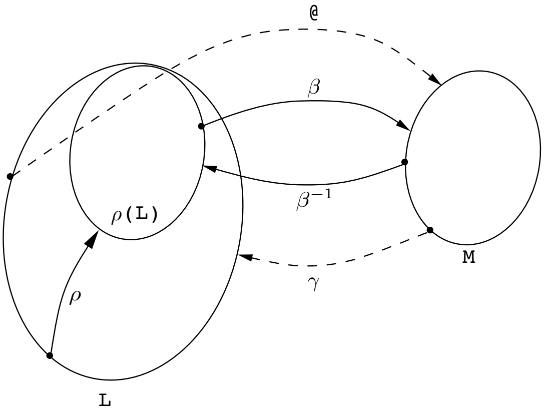
- θ satisfies the substitution property for $\sqcup$. We have to prove that $x \equiv y(\theta)$ implies $(x \sqcup t) \equiv (y \sqcup t)(\theta)$. According to (a), $\{\exists u \in L : x \sqsubseteq u, y \sqsubseteq u, x \equiv u(\theta), y \equiv u(\theta)\}$, so, according to (c), $(x \sqcup t) \equiv (u \sqcup t)(\theta)$ and $(y \sqcup t) \equiv (u \sqcup t)(\theta)$. The transitivity property that we have just shown implies that $(x \sqcup t) \equiv (y \sqcup t)(\theta)$.

End of proof.

In order to check that a congruence relation is join-complete, it is sufficient to prove the additional property $\{\forall x \in L, x \equiv \sqcup([x]\theta)(\theta)\}$.

4.2.7 Definition of an upper closure operator by a pair of adjoint functions

Let L be a complete lattice and $\rho \in \mathit{clos}(L \to L)$. In order to describe the elements of $\rho(L)$, we will use a complete lattice $M(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ that is isomorphic to $\rho(L)$ by a complete isomorphism $\beta \in (\rho(L) \to M)$. Let @ be equal to $\beta \circ \rho$ and let γ be the extension of β^{-1} to L:



In reference to our first work (Cousot & Cousot [1975a,1975b], Cousot & Cousot [1976]), we call @ an *abstraction* operator and γ a *concretization* operator. It is easy to check that @ is monotone and surjective, γ is monotone and injective, $@ \circ \gamma = \boldsymbol{\lambda} x \bullet x$, and $\gamma \circ @ \sqsupseteq \boldsymbol{\lambda} x \bullet x$, which immediately imply that $\{\forall x \in L, \forall y \in M, \{x \sqsubseteq \gamma(y)\} \Leftrightarrow \{@(x) \sqsubseteq y\}\}$. Moreover, γ is a complete morphism and @ is a complete join-morphism. Indeed, let $\langle x_i : i \in \Delta\rangle$ be a family of elements in L, then, by monotonicity $@(\bigsqcup_{i\in\Delta} x_i) \sqsupseteq \bigsqcup_{i\in\Delta} @(x_i)$. Moreover, $@(\bigsqcup_{i\in\Delta} x_i) \sqsubseteq @(\bigsqcup_{i\in\Delta} \gamma(@(x_i))) = @(\gamma(\bigsqcup_{i\in\Delta} @(x_i)) = \bigsqcup_{i\in\Delta} @(x_i)$. By antisymmetry, we have $@(\bigsqcup_{i\in\Delta} x_i) = \bigsqcup_{i\in\Delta} @(x_i)$.

DEFINITION 4.2.7.0.1 *Pair of upper adjoint functions*

Let $L(\sqsubseteq, \perp, \top, \sqcup, \sqcap)$ and $M(\sqsubseteq, \perp, \top, \sqcup, \sqcap)$ be two complete lattices. A pair of monotone functions $@ \in (L \rightarrow M)$ and $\gamma \in (M \rightarrow L)$ is called a pair of upper adjoint functions if and only if $\{\forall x \in L, \forall y \in M, \{\{x \sqsubseteq \gamma(y)\} \Leftrightarrow \{@(x) \sqsubseteq y\}\}\}$.

The notion of pair of adjoint functions appeared already in Cousot & Cousot [1975a,1975b] with the same assumptions. We have borrowed the term "pair of adjoint functions" from Scott [1976] who is using the dual notion with the additional hypotheses that $@$ and γ are upper continuous and that L and M are "continuous lattices." Theorem 4.2.7.0.3 generalizes Scott's result and shows that these additional assumptions are useless.

PROPOSITION 4.2.7.0.2

Let $L(\sqsubseteq, \perp, \top, \sqcup, \sqcap)$ and $M(\sqsubseteq, \perp, \top, \sqcup, \sqcap)$ be two complete lattices and $\rho \in$ *clos*$(L \rightarrow L)$ such that $\beta \in (\rho(L) \rightarrow M)$ is a complete isomorphism. Then, $(@, \gamma)$ is a pair of upper adjoint functions, $@$ is surjective, γ is injective, $@$ is a complete join-morphism, and γ is a complete morphism.

Conversely, we use pairs of adjoint functions in order to define upper closure operators:

THEOREM 4.2.7.0.3

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ and $M(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be two complete lattices and $(@, \gamma)$ with $@ \in (L \to M)$ and $\gamma \in (M \to L)$ be a pair of upper adjoint functions.

(a) - In a pair $(@, \gamma)$ of upper adjoint functions, each function fully defines the other in a unique way and $\boldsymbol{\lambda} x \bullet x \sqsubseteq \gamma \circ @$ and $@ \circ \gamma \sqsubseteq \boldsymbol{\lambda} y \bullet y$.

(b) - $@$ is surjective if and only if γ is injective.

(c) - Whenever $@$ is surjective or γ is injective, then:

- $\gamma \circ @ \in \mathit{clos}(L \to L)$ and $@ \circ \gamma = \boldsymbol{\lambda} y \bullet y$,
- $@$ is a complete join-morphism and γ is a complete meet-morphism,
- $@ = \boldsymbol{\lambda} x \bullet \sqcap \{y \in M : x \sqsubseteq \gamma(y)\}$ and $\gamma = \boldsymbol{\lambda} y \bullet \sqcup \{x \in L : @(x) \sqsubseteq y\}$,
- $\gamma \circ @(L)$ and M are isomorphic by the complete isomorphism $(@ \mid \gamma \circ @(L))$ the inverse of which is γ.

└

Proof:

(a) - $\forall x \in L, @(x) \sqsubseteq @(x)$ and, according to 4.2.7.0.1, this implies that $x \sqsubseteq \gamma(@(x))$. Likewise, $\forall y \in M, \gamma(y) \sqsubseteq \gamma(y)$ implies that $@(\gamma(y)) \sqsubseteq y$.

Let f be such that (f, γ) is a pair of upper adjoint functions. Then, $\forall x \in L, x \sqsubseteq \gamma(f(x))$ and, since $@$ is monotone, we have $@(x) \sqsubseteq @(\gamma(f(x))) \sqsubseteq f(x)$. Moreover, $\forall x \in L, x \sqsubseteq \gamma(@(x))$ and, since f is monotone, we have $f(x) \sqsubseteq f(\gamma(@(x))) \sqsubseteq @(x)$. By antisymmetry, we conclude that $f = @$.

Let f be such that $(@, f)$ is a pair of upper adjoint functions. Then, $\forall y \in L, @(f(y)) \sqsubseteq y$, so, since γ is monotone, we have that $\gamma(y) \sqsupseteq \gamma(@(f(y))) \sqsupseteq f(y)$. Likewise, $f(y) \sqsupseteq f(@(\gamma(y))) \sqsupseteq \gamma(y)$ and, by antisymmetry, we conclude that $f = \gamma$.

(b) - Let us assume that γ is injective and let us prove that $@$ is surjective. For any y in M, we have $@(\gamma(y)) \sqsubseteq y$ and, by monotonicity, $\gamma(@(\gamma(y))) \sqsubseteq \gamma(y)$. Moreover, $\gamma(y) \sqsubseteq \gamma(@(\gamma(y)))$, so, by antisymmetry, $\gamma(y) = \gamma(@(\gamma(y)))$. Since γ is injective, this implies that $@(\gamma(y)) = y$. So, for any y in M, there exists $x = \gamma(y) \in L$ such that $y = @(x)$, and so, @ is surjective.

Let us assume that @ is surjective and monotone, and let us prove that γ is injective. Indeed, $\forall y \in M, \exists x \in L : y = @(x)$. So, $\gamma(y) = \gamma(@(x)) \sqsupseteq x$. The monotonicity of @ implies that $@(\gamma(y)) \sqsupseteq @(x) = y$. Moreover, we always have $@(\gamma(y)) \sqsubseteq y$, and so, by antisymmetry, $@(\gamma(y)) = y$. Let $y_1, y_2 \in M$, then $\gamma(y_1) = \gamma(y_2)$ implies that $@(\gamma(y_1)) = @(\gamma(y_2))$, which implies that $y_1 = y_2$, that is to say that γ is injective.

(c) - Note that if γ is injective or if @ is surjective, then we have $@ \circ \gamma = \boldsymbol{\lambda} y \bullet y$. Let us prove that $\gamma \circ @$ is an upper closure operator on L. $\forall x \in L, x \sqsubseteq \gamma(@(x))$, so that $\gamma \circ @$ is extensive. $\gamma \circ @$ is a composition of monotone functions, so, it is monotone. Since $@ \circ \gamma = \boldsymbol{\lambda} y \bullet y$, we have $(\gamma \circ @) \circ (\gamma \circ @) = \gamma \circ @$, which proves that $\gamma \circ @$ is idempotent.

- Let $\{x_i : i \in \Delta\}$ be a family of elements in L. Then, by monotonicity, $@(\bigsqcup_{i\in\Delta} x_i) \sqsupseteq \bigsqcup_{i\in\Delta} @(x_i)$. Moreover, $@(\bigsqcup_{i\in\Delta} x_i) \sqsubseteq @(\bigsqcup_{i\in\Delta} \gamma(@(x_i))) \sqsubseteq @(\gamma(\bigsqcup_{i\in\Delta} @(x_i))) = \bigsqcup_{i\in\Delta} @(x_i)$, by monotonicity of γ, @, and $@ \circ \gamma = \boldsymbol{\lambda} y \bullet y$. By antisymmetry, @ is a complete join-morphism. Likewise, by monotonicity, $\gamma(\bigsqcap_{i\in\Delta} x_i) \sqsubseteq \bigsqcap_{i\in\Delta} \gamma(x_i)$ and $\bigsqcap_{i\in\Delta} \gamma(x_i) \sqsubseteq \gamma(@(\bigsqcap_{i\in\Delta} \gamma(x_i))) \sqsubseteq \gamma(\bigsqcap_{i\in\Delta} @(\gamma(x_i))) = \gamma(\bigsqcap_{i\in\Delta} x_i)$. Thus, we conclude, by antisymmetry, that γ is a complete meet-morphism.

- Let $f = \boldsymbol{\lambda} y \bullet \sqcup \{x \in L : @(x) \sqsubseteq y\}$. Since @ is surjective for any y in M, there exists $x \in L$ such that $@(x) = y$, so, $\{x \in L : @(x) \sqsubseteq y\}$ is not empty and, since L is a complete lattice, $\sqcup\{x \in L : @(x) \sqsubseteq y\}$ exists, which ensures that f is a total function from M to L.

Let t, u be two elements in M such that $t \sqsubseteq u$. Then, $\forall x \in L, \{@(x) \sqsubseteq t\} \Rightarrow \{@(x) \sqsubseteq u\}$, so, $\sqcup\{x \in L : @(x) \sqsubseteq t\} \sqsubseteq \sqcup\{x \in L : @(x) \sqsubseteq u\}$, which is equivalent to $f(t) \sqsubseteq f(u)$. This ensures that f is monotone.

$\forall x \in L, \forall y \in M, \{x \sqsubseteq f(y)\} \Rightarrow \{@(x) \sqsubseteq @(f(y))\}$ since @ is monotone. $@(f(y)) = @(\sqcup\{z \in L : @(z) \sqsubseteq y\}) = \sqcup\{@(z) : @(z) \sqsubseteq y\}$ since @ is a complete join-morphism. So, $@(f(y)) \sqsubseteq y$. By transitivity, $\{x \sqsubseteq f(y)\} \Rightarrow \{@(x) \sqsubseteq y\}$. Let us now assume that $@(x) \sqsubseteq y$, then $f(@(x)) \sqsubseteq f(y)$ since f is monotone. So, $f(@(x)) = \sqcup\{z \in L : @(z) \sqsubseteq @(x)\} \sqsupseteq \sqcup\{z \in L : z \sqsubseteq x\} = x$. We get, by transitivity, that $\{@(x) \sqsubseteq y\} \Rightarrow \{x \sqsubseteq f(y)\}$. So, $(@, f)$ is a pair of upper adjoint functions and, as one defines the other, we conclude that $f = \gamma$.

- Since @ is monotone and surjective, we have $@(\top) = \top$. So, $\top \sqsubseteq \gamma \circ @(\top) = \gamma(\top) \sqsubseteq \top$, which is equivalent to $\gamma(\top) = \top$. So, let γ be an injective complete meet-morphism from M to L such that $\gamma(\top) = \top$. Let us set $f = \boldsymbol{\lambda} x \bullet \sqcap \{y \in M : x \sqsubseteq \gamma(y)\}$. For any x in L, we have $x \sqsubseteq \top = \gamma(\top)$, so, $\{y \in M : x \sqsubseteq \gamma(y)\}$ is not empty and, since M is a complete lattice, f is a total function from L to M.

Let $t \sqsubseteq u$, then $\forall y \in M, \{u \sqsubseteq \gamma(y)\} \Rightarrow \{t \sqsubseteq \gamma(y)\}$, so, $\sqcap\{y \in M : t \sqsubseteq \gamma(y)\} \sqsubseteq \sqcap\{y \in M : u \sqsubseteq \gamma(y)\}$ and f is monotone.

$\forall x \in L, \forall y \in M, \{x \sqsubseteq \gamma(y)\} \Rightarrow \{f(x) \sqsubseteq f(\gamma(y))\}$ by monotonicity. $f(\gamma(y)) = \sqcap\{z \in M : \gamma(y) \sqsubseteq \gamma(z)\} \sqsubseteq \sqcap\{z \in M : y \sqsubseteq z\} = y$. By transitivity, $\{x \sqsubseteq \gamma(y)\} \Rightarrow \{f(x) \sqsubseteq y\}$. Let us now assume that $f(x) \sqsubseteq y$, then $\gamma(f(x)) \sqsubseteq \gamma(y)$. We have $\gamma(f(x)) = \gamma(\sqcap\{z \in M : x \sqsubseteq \gamma(z)\}) = \sqcap\{\gamma(z) : x \sqsubseteq \gamma(z)\} \sqsupseteq x$ since γ is a complete meet-morphism. So, $\{f(x) \sqsubseteq y)\} \Rightarrow \{x \sqsubseteq \gamma(y)\}$ and (f, γ) is a pair of upper adjoint functions and, since one defines the other, we conclude that $f = @$.

- Let us prove that $\gamma \circ @(L)$ and M are isomorphic by the complete isomorphism $(@ \mid \gamma \circ @(L))$. Let us prove that the restriction $(@ \mid \gamma \circ @(L))$ of $@$ to $\gamma \circ @(L)$ is bijective. $\forall y \in M, y \in @(L)$, since $@$ is surjective. So, $\gamma(y) \in (\gamma \circ @(L))$ and $@ \circ \gamma(y) = y$. Thus, $\forall y \in M, \exists x \in (\gamma \circ @(L))$ such that $@(x) = y$ and $(@ \mid \gamma \circ @(L))$ is surjective. Now, $\gamma \circ @(L) = \gamma(M)$ and γ is surjective from M to $\gamma(M)$, so, $\forall x \in (\gamma \circ @(L)), \exists y \in M$ such that $x = \gamma(y)$. Then, $(@ \mid \gamma \circ @(L))(x) = (@ \mid \gamma \circ @(L)) \circ \gamma(y) = y$, so, $\gamma \circ (@ \mid \gamma \circ @(L))(x) = \gamma(y) = x$. Now, let $x_1, x_2 \in \gamma \circ @(L)$ be such that $(@ \mid \gamma \circ @(L))(x_1) = (@ \mid \gamma \circ @(L))(x_2)$. Then, $x_1 = \gamma((@ \mid \gamma \circ @(L))(x_1)) = \gamma((@ \mid \gamma \circ @(L))(x_2)) = x_2$, which ensures that $(@ \mid \gamma \circ @(L))$ is injective. Moreover, its inverse is γ.

We are left to prove that $(@ \mid \gamma \circ @(L))$ is a complete isomorphism. Since $(\gamma \circ @) \in \mathit{clos}(L \to L)$, we know that $\gamma \circ @(L)$ is a complete lattice $(\sqsubseteq, \gamma \circ @(\bot), \top, \boldsymbol{\lambda} S \bullet \gamma \circ @(\sqcup S), \sqcap)$. Let $S \subseteq \gamma \circ @(L)$. The least upper bound of S in $\gamma \circ @(L)$ is $\gamma \circ @(\sqcup S)$, and $@(\gamma \circ @(\sqcup S)) = @(\sqcup S) = \sqcup @(S)$, so, $(@ \mid \gamma \circ @(L))(\gamma \circ @(\sqcup S)) = \sqcup(@ \mid \gamma \circ @(L))(S)$. Moreover, $\gamma(\sqcap @(S)) = \sqcap(\gamma(@(S))) = \gamma \circ @(\sqcap(\gamma \circ @(S))) = \gamma \circ @(\sqcap S)$ since $\gamma \circ @(S)$ is a set of fixpoints of the upper closure operator $\gamma \circ @$, and so, $\sqcap(\gamma \circ @(S)) = (\sqcap S) \in (\gamma \circ @(L))$. Since γ is injective, we have $\sqcap @(S) = @(\sqcap S)$ and $\sqcap(@ \mid \gamma \circ @(L))(S) = (@ \mid \gamma \circ @(L))(\sqcap S)$.

End of proof.

COROLLARY 4.2.7.0.4

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ and $M(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be two complete lattices and $@$ be a complete surjective join-morphism from L to M. Then, $(@, \boldsymbol{\lambda} y \bullet \sqcup \{x \in L : @(x) \sqsubseteq y\})$ is a pair of upper adjoint functions.

COROLLARY 4.2.7.0.5

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ and $M(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be two complete lattices and γ be a complete injective meet-morphism from M to L such that $\gamma(\top) = \top$. Then, $(\boldsymbol{\lambda} x \bullet \sqcap \{y \in M : x \sqsubseteq \gamma(y)\}, \gamma)$ is a pair of upper adjoint functions.

DEFINITION 4.2.7.0.6 *Over-approximated image of a complete lattice*

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ and $M(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be two complete lattices. We will say that M is the over-approximated image of L (which we shall write as $L\bar{\rhd}M$) if and only if there exists $\rho \in \mathit{clos}(L \to L)$ such that $\rho(L)(\sqsubseteq, \rho(\bot), \top, \boldsymbol{\lambda} S \bullet \rho(\sqcup S), \sqcap)$ and $M(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ are completely isomorphic.

COROLLARY 4.2.7.0.7

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ and $M(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be two complete lattices.

$\{L\bar{\rhd}M\}$ $\Leftrightarrow$ $\{\exists @ \in (L \to M)$ surjective, $\exists \gamma \in (M \to L)$ injective: $(@, \gamma)$ is a pair of upper adjoint functions$\}$

$\{L\bar{\rhd}M\}$ $\Leftrightarrow$ $\{\exists \gamma \in (M \to L)$: injective, complete meet-morphism such that $\gamma(\top) = \top\}$

$\{L\bar{\rhd}M\}$ $\Leftrightarrow$ $\{\exists @ \in (L \to M)$: surjective, complete join-morphism$\}$

4.2.8 Induced closure operator on the space of monotone operators on a complete lattice L by a closure operator on L

THEOREM 4.2.8.0.1

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $\rho \in \mathit{clos}(L \to L)$, then

$$\boldsymbol{\lambda} f \bullet \rho \circ f \circ \rho \quad \in \quad \mathit{clos}(\mathit{mon}(L \to L) \to \mathit{mon}(L \to L))$$

Proof: Let $f, g \in \mathit{mon}(L \to L)$ be such that $f \sqsubseteq g$, then $\rho \circ f \circ \rho \sqsubseteq \rho \circ g \circ \rho$, since ρ is monotone. So, $\boldsymbol{\lambda} f \bullet \rho \circ f \circ \rho$ is monotone. Moreover, $f \sqsubseteq \rho \circ f \circ \rho$, since $\forall x \in L, f(x) \sqsubseteq \rho(f(\rho(x)))$ because ρ is extensive and both f and ρ are monotone. Finally, $\rho \circ \rho \circ f \circ \rho \circ \rho = \rho \circ f \circ \rho$, which ensures that $\boldsymbol{\lambda} f \bullet \rho \circ f \circ \rho$ is idempotent.

End of proof.

DEFINITION 4.2.8.0.2

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and ρ be an upper closure operator on L. We denote by $\bar{\rho} = \boldsymbol{\lambda} f \bullet \rho \circ f \circ \rho$ the *upper closure operator that is induced on the space of monotone operators on L by ρ.*

We shall note that $\bar{\rho}(f)$ is the least monotone function on $(\rho(L) \rightarrow \rho(L))$ that is greater than the restriction of f to $\rho(L)$. (Indeed, let $g \in \mathit{mon}(\rho(L) \rightarrow \rho(L))$ be such that $(f \mid \rho(L)) \sqsubseteq g$, then $\rho \circ f \circ \rho \sqsubseteq \rho \circ g \circ \rho = g$.)

THEOREM 4.2.8.0.3

$$\{L \bar{\triangleright}(@, \gamma) M\}$$
$$\Rightarrow \quad \{\mathit{mon}(L \rightarrow L) \bar{\triangleright}(\boldsymbol{\lambda} f \bullet @ \circ f \circ \gamma, \boldsymbol{\lambda} f \bullet \gamma \circ f \circ @)\, \mathit{mon}(M \rightarrow M)\}$$

COROLLARY 4.2.8.0.4

Let L, M be two complete lattices such that $L \bar{\triangleright}(@, \gamma) M$ and $\mathit{mon}(L \rightarrow L) \bar{\triangleright}(\bar{@}, \bar{\gamma})\, \mathit{mon}(M \rightarrow M)$ with $\bar{@} = \boldsymbol{\lambda} f \bullet @ \circ f \circ \gamma$ and $\bar{\gamma} = \boldsymbol{\lambda} f \bullet \gamma \circ f \circ @$. Let $F, G \in \mathit{mon}(L \rightarrow L)$, then:

(a) - $\{\mathit{lfp}(F) \sqsubseteq \gamma(\mathit{lfp}(\bar{@}(F)))\}$

(b) - $\{\bar{F} \in \mathit{mon}(M \rightarrow M)$ <u>and</u> $\bar{@}(F) \sqsubseteq \bar{F}\} \Rightarrow \{\mathit{lfp}(F) \sqsubseteq \gamma(\mathit{lfp}(\bar{F}))\}$

(c) - $\{\mathit{lfp}(F \circ G) \sqsubseteq \gamma(\mathit{lfp}(\bar{@}(F) \circ \bar{@}(G)))\}$

(d) - $\{\bar{F}, \bar{G} \in \mathit{mon}(M \rightarrow M)$ <u>and</u> $\bar{@}(F) \sqsubseteq \bar{F}$ <u>and</u> $\bar{@}(G) \sqsubseteq \bar{G}\}$
$\Rightarrow \quad \{\mathit{lfp}(F \circ G) \sqsubseteq \gamma(\mathit{lfp}(\bar{F} \circ \bar{G}))\}$

(the same holds for *gfp*).

PROPOSITION 4.2.8.0.5

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice, $\rho_1, \rho_2 \in \mathit{clos}(L \to L)$, and $f \in \mathit{mon}(L \to L)$, then:

$$\mathit{lfp}((\rho_1 \sqcap \rho_2) \circ f \circ (\rho_1 \sqcap \rho_2)) \quad \sqsubseteq \quad \mathit{lfp}(\rho_1 \circ f \circ \rho_1) \sqcap \mathit{lfp}(\rho_2 \circ f \circ \rho_2)$$

Proof: $\rho_1 \sqcap \rho_2 \sqsubseteq \rho_1$ and f is monotone imply that $(\rho_1 \sqcap \rho_2) \circ f \circ (\rho_1 \sqcap \rho_2) \sqsubseteq (\rho_1 \circ f \circ \rho_1)$. Since *lfp* is monotone, we have $\mathit{lfp}((\rho_1 \sqcap \rho_2) \circ f \circ (\rho_1 \sqcap \rho_2)) \sqsubseteq \mathit{lfp}(\rho_1 \circ f \circ \rho_1)$. The same holds for ρ_2.
End of proof.

4.3 APPROXIMATION OF THE FIXPOINTS OF AN OPERATOR BY APPROXIMATION OF THE OPERATOR

The complex computation of the extreme fixpoints of an operator F can be replaced with the easier computation of the fixpoints of an operator $\bar{F}$ that approximates F as follows:

THEOREM 4.3.0.1

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $F, G \in \mathit{mon}(L \to L)$, then

$$\{\{F \ \sqsubseteq G\} \quad \Rightarrow \quad \{\{\ \mathit{lfp}(F) \ \sqsubseteq \mathit{lfp}(G)\} \ \underline{\text{and}} \ \{\mathit{gfp}(F) \ \sqsubseteq \mathit{gfp}(G)\}\}\}$$

Proof: Recall that $\mathit{lfp}(F) = \sqcap\{X \in L : F(X) \sqsubseteq X\}$ and $\mathit{lfp}(G) = \sqcap\{X \in L : G(X) \sqsubseteq X\}$. Since $F \sqsubseteq G$, we have $\{\forall X \in L : \{G(X) \sqsubseteq X\} \Rightarrow \{F(X) \sqsubseteq X\}\}$, which implies $\mathit{lfp}(F) \sqsubseteq \mathit{lfp}(G)$ and, by duality, $\{\{F \sqsupseteq G\} \Rightarrow \{\mathit{gfp}(F) \sqsupseteq \mathit{gfp}(G)\}\}$.
End of proof.

4.3.1 Induced approximation of an operator on a complete lattice by an approximated image of the lattice

In order to approximate the fixpoints of a monotone operator F on a complete lattice $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$, we will compute the fixpoints of a simpler operator $\bar{F}$ that will be chosen so that the algorithm that computes $\bar{F}$ is less complex than the algorithm that computes F. Let us consider the case of an over-approximation (the case of an under-approximation is dual), that is to say that we have to choose $\bar{F}$ such that $F \sqsubseteq \bar{F}$ (Theorem 4.3.0.1). The issue is to find a means to derive $\bar{F}$ from F.

We propose to define a subset M of L in which will be chosen the approximation of the fixpoints of F. To achieve that goal, having fixed M, we will choose $\bar{F}$ as the best over-approximation of F that belongs to $(L \rightarrow M)$.

We also propose to choose M independently from F and, as a consequence, we will choose a subset M of L such that each element x in L has an over-approximation in M. Let $\rho \in (L \rightarrow M)$ be an operator which maps each element x in L to an over-approximation $\rho(x)$ of x in M. Since ρ is extensive, it is uniquely defined by the choice of M if and only if ρ is an upper closure operator on L (4.2.2.0.1) and M is a lower Moore family of L (4.2.2.0.4). Indeed, all the elements y in M that are greater than x are an over-approximation of x. In particular, the supremum $\top$ of L can be over-approximated only by $\top$, and so, $\top \in M$. Among the over-approximations of x in M, some are better than the others. Since the comparison criterion is the order $\sqsubseteq$ on L, $a \in M$ is a better approximation of $x \in L$ than $b \in M$ only if $x \sqsubseteq a \sqsubseteq b$. The best approximations of x in M are the minimal elements of $\{y \in M : x \sqsubseteq y\}$. There exists a unique best approximation of an arbitrary element x of L in M if $\{y \in M : x \sqsubseteq y\}$ has a least element, that is to say, if M is a lower Moore family (4.2.2.0.2).

Let us assume that we have chosen a subset M of L that is not a lower Moore family. In such a case, some elements of L have several minimal over-approximations in M and it is impossible to choose a best one independently from a given F. In this case, Theorem 4.2.2.0.5 gives the minimal set of elements of L to add to M in order to avoid the ambiguity.

A lower Moore family M induces a unique upper closure operator ρ such that $\rho(L) = M$ (4.2.2.0.5) and, conversely, an upper closure operator ρ uniquely defines a lower Moore family (2.3.0.1), so that the choice of a set M can be replaced with the choice of an upper closure operator ρ by defining $M = \rho(L)$. Thus, we have restated several characterizations of upper closure operators (4.2.1).

If we choose an operator ρ that is not an upper closure operator, Theorem 4.2.3.0.5 shows that $clos(\rho)$ is the least upper closure operator on L such that any x in L has a best over-approximation $clos(\rho)(x)$ greater than $\rho(x)$. We have thus given several equivalent definitions of *clos* (4.2.3.0.5, 4.2.3.0.6, 4.2.3.0.8).

We have studied several ways to build upper closure operators (4.2.1, 4.2.5, 4.2.6, 4.2.7) and to compose them (4.2.3, 4.2.4). In particular, the definition of an upper closure operator by a family of principal ideals (4.2.5.0.3) or, better, the use of a join-complete congruence relation (4.2.6.0.4) emphasizes the idea of not distinguishing the elements in an equivalence class, which are all approximated by the supremum of this class.

We will often use combinations of closure operators in order to strengthen an approximation (4.2.3.0.5, 4.2.4.0.7) or, contrariwise, to refine it (meet of closure operators 4.2.4.0.5). (Note that Propositions 4.2.4.0.2 and 4.2.4.0.3 show that it is always better to use the join $clos(\rho_1 \sqcup \rho_2)$ rather than the composition $\rho_1 \circ \rho_2$ of two closures ρ_1 and ρ_2, since $clos(\rho_1 \sqcup \rho_2)$ is always a closure operator, whereas $\rho_1 \circ \rho_2$ is not necessarily a closure operator and, if $\rho_1 \circ \rho_2$ is a closure operator, then $\rho_1 \circ \rho_2 = clos(\rho_1 \sqcup \rho_2)$.)

The choice of the space M of approximated values or of the induced upper closure operator depends on the problem to be solved, on the desired accuracy in the approximation, and of the cost that we can afford in order to solve the problem. This will be largely illustrated in Chapter 5. Theorems 4.2.4.0.5 and 4.2.3.0.10 enable the partial ordering of approximations according to their accuracy, thanks to the partial order defined on closure operators or the corresponding Moore families.

Having chosen an upper closure operator ρ, the best over-approximation of $F \in mon(L \rightarrow L)$ in $mon(L \rightarrow \rho(L))$ is $\rho \circ F$. Nevertheless, in order to restrict the computation domain to $\rho(L)$, we will choose $\bar{F} = \rho \circ F \circ \rho$, which is the least monotone function on $(\rho(L) \rightarrow \rho(L))$ that is greater than the restriction of F to $\rho(L)$ (4.2.8). Then, $lfp(F) \sqsubseteq lfp(\bar{F}) = luis(\rho \circ F \circ \rho)(\bot) = luis(\rho \circ F)(\rho(\bot)) = lfp((\rho \circ F \mid \rho(L)))$. Likewise, $gfp(F)$ is over-approximated by $gfp((\rho \circ F \mid \rho(L)))$.

When the elements of $\rho(L)$ are not computer-representable, we will use a space of approximated values M such that $L\bar{\triangleright}(@, \gamma)M$ with $\rho = \gamma \circ @$, and choose M so that its elements can be easily represented in a computer. Then, the best approximation of $F \in mon(L \rightarrow L)$ in $mon(M \rightarrow M)$ is $@ \circ F \circ \gamma$ (4.2.8.0.3). When it is difficult to write an algorithm that would compute $@ \circ F \circ \gamma$, we will choose to implement $\bar{F} \in mon(M \rightarrow M)$ such that $@ \circ F \circ \gamma \sqsubseteq \bar{F}$. Then, $lfp(F) \sqsubseteq \gamma(lfp(\bar{F}))$ and $gfp(F) \sqsubseteq \gamma(gfp(\bar{F}))$ (4.2.8.0.4.(b)). Usually, F is a composition of elementary functions, which enables a systematic derivation of $\bar{F}$ from F by over-approximating each elementary function $f \in mon(L \rightarrow L)$ by an elementary approximate function $\bar{f} \in mon(M \rightarrow M)$ such that $\bar{f} \sqsubseteq \gamma \circ f \circ @$ (4.2.8.0.4.(d)).

Note that instead of approximating the monotone operator F by a monotone operator $\bar{F}$, we could also use an extensive operator (or, dually, reductive):

PROPOSITION 4.3.1.0.1

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $f \in ext(L \rightarrow L)$, then $fp(f) = luis(f)(L)$.

PROPOSITION 4.3.1.0.2

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $f \in mon(L \rightarrow L)$, $\bar{f} \in (L \rightarrow L)$ such that $f \sqsubseteq \bar{f}$, then $lfp(f) \sqsubseteq luis(ext(\bar{f}))(\bot)$.

4.3.2 Improving the approximation of a fixpoint of a monotone operator

The results of Paragraph 4.1.1 enable the improvement of the approximation of an extreme fixpoint of a monotone operator f on a complete lattice L. We now specialize these results to the case when we have an approximation g of f, ($g \sqsubseteq f$ or $f \sqsubseteq g$).

<u>THEOREM</u> 4.3.2.0.1

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $f, g \in mon(L \to L)$,

$$\{\{\exists y \in L : y \sqsubseteq \mathit{lfp}(f)\} \ \underline{\text{and}}\ \{g \sqsubseteq f\}\}$$
$$\Rightarrow \quad \{y \sqsubseteq \mathit{lfp}(\boldsymbol{\lambda}\, x \bullet y \sqcup g(x)) \sqsubseteq \mathit{luis}(\boldsymbol{\lambda}\, x \bullet x \sqcup g(x))(y) \sqsubseteq \mathit{lfp}(f)\}$$

Proof: Let $y \sqsubseteq \mathit{lfp}(f)$ and $g \sqsubseteq f$. Let us consider the increasing iteration sequence $\langle x^\delta : \delta \in \mu(L)\rangle$ starting from y and defined by $\boldsymbol{\lambda}\, x \bullet x \sqcup g(x)$ (2.5.1.0.1). This is an increasing under-approximated iteration sequence for f starting from y (4.1.1.0.5). Indeed, $x^0 = y$, whenever δ is a successor ordinal, then $x^{\delta-1} \sqsubseteq x^\delta = x^{\delta-1} \sqcup g(x^{\delta-1}) \sqsubseteq x^{\delta-1} \sqcup f(x^{\delta-1})$ since $g \sqsubseteq f$, and, whenever δ is a limit ordinal, $x^\delta = \bigsqcup\limits_{\alpha<\delta} x^\alpha$. As a consequence, Theorem 4.1.1.0.6 implies that the limit $\mathit{luis}(\boldsymbol{\lambda}\, x \bullet x \sqcup g(x))(y)$ of $\langle x^\delta : \delta \in \mu(L)\rangle$ is an under-approximation of $\mathit{luis}(\boldsymbol{\lambda}\, x \bullet x \sqcup f(x))(y)$. But, according to Theorems 2.5.2.0.5 and 2.5.3.0.1, $\mathit{luis}(\boldsymbol{\lambda}\, x \bullet x \sqcup g(x))(y) = \mathit{luis}(\boldsymbol{\lambda}\, x \bullet y \sqcup g(x))(y) \sqsupseteq \mathit{luis}(\boldsymbol{\lambda}\, x \bullet y \sqcup g(x))(\bot) = \mathit{lfp}(\boldsymbol{\lambda}\, x \bullet y \sqcup g(x)) \sqsupseteq y$. Moreover, $\bot \sqsubseteq y \sqsubseteq \mathit{lfp}(f)$ implies $\mathit{lfp}(f) = \mathit{luis}(\boldsymbol{\lambda}\, x \bullet x \sqcup f(x))(\bot) \sqsubseteq \mathit{luis}(\boldsymbol{\lambda}\, x \bullet x \sqcup f(x))(y) \sqsubseteq \mathit{luis}(\boldsymbol{\lambda}\, x \bullet x \sqcup f(x))(\mathit{lfp}(f)) = \mathit{lfp}(f)$, which enables us to conclude that $y \sqsubseteq \mathit{lfp}(\boldsymbol{\lambda}\, x \bullet y \sqcup g(x)) \sqsubseteq \mathit{luis}(\boldsymbol{\lambda}\, x \bullet x \sqcup g(x))(y) \sqsubseteq \mathit{lfp}(f)$
End of proof.

Remark 4.3.2.0.2

Let y be an under-approximation of $\mathit{lfp}(f)$. Given $g \sqsubseteq f$, we can improve y because $y \sqsubseteq \mathit{luis}(\boldsymbol{\lambda}\, x \bullet x \sqcup g(x))(y) \sqsubseteq \mathit{lfp}(f)$. It is impossible to improve $\mathit{luis}(\boldsymbol{\lambda}\, x \bullet x \sqcup$

$g(x))(y)$ by iterating this process because $\mathit{luis}(\boldsymbol{\lambda} x \cdot x \sqcup g(x))$ is idempotent. Moreover, Theorem 2.5.4.0.2 shows that $\mathit{luis}(\boldsymbol{\lambda} x \cdot x \sqcup g(x))(y)$ is the greatest value of L which can possibly be obtained from y by using $\sqcup, \sqcap$, and g. So, $\mathit{luis}(\boldsymbol{\lambda} x \cdot x \sqcup g(x))(y)$ is the best under-approximation of $\mathit{lfp}(f)$ that can be obtained by using only these elements.

End of remark.

By the duality principle, we get:

THEOREM 4.3.2.0.3

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $f, g \in \mathit{mon}(L \to L)$,

$$\{\{\exists y \in L : \mathit{gfp}(f) \sqsubseteq y\} \ \underline{\text{and}} \ \{f \sqsubseteq g\}\}$$
$$\Rightarrow \quad \{\mathit{gfp}(f) \sqsubseteq \mathit{llis}(\boldsymbol{\lambda} x \cdot x \sqcap g(x))(y) \sqsubseteq \mathit{gfp}(\boldsymbol{\lambda} x \cdot y \sqcap g(x)) \sqsubseteq y\}$$

THEOREM 4.3.2.0.4

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $f, g \in \mathit{mon}(L \to L)$,

$$\{\{\exists y \in L : \mathit{lfp}(f) \sqsubseteq y\} \ \underline{\text{and}} \ \{f \sqsubseteq g\}\}$$
$$\Rightarrow \quad \{\mathit{lfp}(f) \sqsubseteq \mathit{lfp}(\boldsymbol{\lambda} x \cdot y \sqcap g(x)) \sqsubseteq \mathit{llis}(\boldsymbol{\lambda} x \cdot x \sqcap g(x))(y) \sqsubseteq y\}$$

Proof: Let $\mathit{lfp}(f) \sqsubseteq y$ and $f \sqsubseteq g$. We consider the increasing iteration sequence $\langle x^\delta : \delta \in \mu(L)\rangle$ starting from $\bot$ and defined by $\boldsymbol{\lambda} x \cdot y \sqcap g(x)$ (2.5.1.0.1). It is an ascending chain with limit $P = \mathit{lfp}(\boldsymbol{\lambda} x \cdot y \sqcap g(x))$ (2.5.3.0.1 and 2.5.3.0.2). Since $P \sqsubset y \sqcap g(P) \sqsubseteq y$, we have $x^\delta \sqsubseteq P \sqsubseteq y$ for any $\delta \in \mu(L)$. For any successor ordinal $\delta \in \mu(L)$, we have $x^{\delta-1} \sqsubseteq x^\delta = (y \sqcap g(x^{\delta-1})) \sqsubseteq g(x^{\delta-1})$. Let us prove that $\langle x^\delta : \delta \in \mu(L)\rangle$ is an increasing over-approximated iteration sequence for $\boldsymbol{\lambda} x \cdot y \sqcap f(x)$ starting from $\bot$ (4.1.1.0.1). Indeed, $x^0 = \bot$. For any successor ordinal, we have $x^{\delta-1} \sqcup (y \sqcap f(x^{\delta-1})) \sqsubseteq x^\delta = y \sqcap g(x^{\delta-1})$ because $x^{\delta-1} \sqsubseteq y$, $y \sqcap f(x^{\delta-1}) \sqsubseteq y$, $x^{\delta-1} \sqsubseteq$

$g(x^{\delta-1})$ and $y \sqcap f(x^{\delta-1}) \sqsubseteq g(x^{\delta-1})$, since $y \sqcap f(x^{\delta-1}) \sqsubseteq f(x^{\delta-1}) \sqsubseteq g(x^{\delta-1})$, as $f \sqsubseteq g$. Theorem 4.1.1.0.2 implies that $\mathit{lfp}(\boldsymbol{\lambda}\, x \bullet y \sqcap g(x))$ is an over-approximation of $\mathit{luis}(\boldsymbol{\lambda}\, x \bullet x \sqcup (y \sqcap f(x)))(\bot)$. Let $\langle t^{\delta} : \delta \in \mu(L)\rangle$ be the increasing iteration sequence starting from $\bot$ and defined by f and let $\langle z^{\delta} : \delta \in \mu(L)\rangle$ be the increasing iteration sequence starting from $\bot$ and defined by $\boldsymbol{\lambda}\, x \bullet x \sqcup (y \sqcap f(x))$. We have $t^0 = z^0 = \bot$. Let us assume that $t^{\alpha} = z^{\alpha}$ for all $\alpha < \delta$. Whenever δ is a limit ordinal, we have $t^{\delta} = \bigsqcup\limits_{\alpha<\delta} t^{\alpha} = \bigsqcup\limits_{\alpha<\delta} z^{\alpha} = z^{\delta}$. Whenever δ is a successor ordinal, we have, in particular, $t^{\delta-1} = z^{\delta-1}$. Since $t^{\delta-1} \sqsubseteq t^{\delta} = f(t^{\delta-1}) \sqsubseteq \mathit{lfp}(f) \sqsubseteq y$, we have $y \sqcap f(z^{\delta-1}) = y \sqcap f(t^{\delta-1}) = f(t^{\delta-1}) = f(z^{\delta-1})$. Since $z^{\delta-1} \sqsubseteq f(z^{\delta-1})$, we have that $t^{\delta} = f(t^{\delta-1}) = f(z^{\delta-1}) = z^{\delta-1} \sqcup (y \sqcap f(z^{\delta-1}))$. By transfinite induction, we conclude that $\{\forall \delta \in \mu(L), t^{\delta} = z^{\delta}\}$ and, in particular, $\mathit{luis}(\boldsymbol{\lambda}\, x \bullet x \sqcup (y \sqcap f(x)))(\bot) = \mathit{luis}(f)(\bot) = \mathit{lfp}(f)$. Moreover, it is easy to prove that each term of $\langle x^{\delta} : \delta \in \mu(L)\rangle$ is smaller than the corresponding term in the decreasing iteration sequence starting from y and defined by $\boldsymbol{\lambda}\, x \bullet x \sqcap g(x)$ (2.5.1.0.2), which enables us to conclude that $\mathit{lfp}(f) \sqsubseteq \mathit{lfp}(\boldsymbol{\lambda}\, x \bullet y \sqcap g(x)) \sqsubseteq \mathit{llis}(\boldsymbol{\lambda}\, x \bullet x \sqcap g(x))(y) \sqsubseteq y$.

End of proof.

THEOREM 4.3.2.0.5

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $g \in \mathit{mon}(L \to L)$, then $\boldsymbol{\lambda}\, y \bullet \mathit{lfp}(\boldsymbol{\lambda}\, x \bullet y \sqcap g(x))$ is a lower closure operator on L that is smaller than g.

Proof: Let us set $h = \boldsymbol{\lambda}\, y \bullet \mathit{lfp}(\boldsymbol{\lambda}\, x \bullet y \sqcap g(x))$. Theorem 4.3.2.0.5 stated that h is reductive. Moreover, $y \sqsubseteq z$ implies that $\boldsymbol{\lambda}\, x \bullet y \sqcap g(x) \sqsubseteq \boldsymbol{\lambda}\, x \bullet z \sqcap g(x)$ and $\mathit{lfp}(\boldsymbol{\lambda}\, x \bullet y \sqcap g(x)) \sqsubseteq \mathit{lfp}(\boldsymbol{\lambda}\, x \bullet z \sqcap g(x))$ (4.3.0.1), which proves that h is monotone. Let $\langle x^{\delta} : \delta \in \mu(L)\rangle$ and $\langle y^{\delta} : \delta \in \mu(L)\rangle$ be the increasing iteration sequences starting from $\bot$ and defined respectively by $\boldsymbol{\lambda}\, x \bullet y \sqcap g(x)$ and $\boldsymbol{\lambda}\, x \bullet h(y) \sqcap g(x)$ (2.5.1.0.1). Since $\langle x^{\delta} : \delta \in \mu(L)\rangle$ is a stationary chain with limit $h(y)$, we have, for any $\delta \in \mu(L), x^{\delta} \sqsubseteq h(y)$. For $\delta = 0$, we have $x^0 = y^0 = \bot$. Let us assume that, for any $\alpha < \delta$, we have $x^{\alpha} = y^{\alpha}$. If δ is a limit ordinal, then we have $x^{\delta} = \bigsqcup\limits_{\alpha<\delta} x^{\alpha} = \bigsqcup\limits_{\alpha<\delta} y^{\alpha} = y^{\delta}$. If δ is

a successor ordinal, then $x^{\delta-1} = y^{\delta-1}$ and $y^{\delta} = (h(y) \sqcap g(y^{\delta-1})) = h(y) \sqcap g(x^{\delta-1}) = y \sqcap g(h(y)) \sqcap g(x^{\delta-1})$. Since $x^{\delta-1} \sqsubseteq h(y)$ and g is monotone, we have $g(x^{\delta-1}) \sqsubseteq g(h(y))$, so, $g(h(y)) \sqcap g(x^{\delta-1}) = g(x^{\delta-1})$, which ensures that $y^{\delta} = y \sqcap g(x^{\delta-1}) = x^{\delta}$. By transfinite induction, we have in particular $h(y) = \mathit{luis}(\boldsymbol{\lambda}\, x \bullet y \sqcap g(x))(\bot) = \mathit{luis}(\boldsymbol{\lambda}\, x \bullet h(y) \sqcap g(x))(\bot) = h(h(y))$. Finally, $\forall y \in L, h(y) = y \sqcap g(h(y)) \sqsubseteq g(h(y)) \sqsubseteq g(y)$.
End of proof.

Remark 4.3.2.0.6

Given $g \sqsubseteq f$, an over-approximation y of $\mathit{lfp}(f)$ can, as previously, be improved as $\mathit{lfp}(\boldsymbol{\lambda}\, x \bullet y \sqcap g(x))$. This process cannot be iterated to further improve the approximation since $\boldsymbol{\lambda}\, y \bullet \mathit{lfp}(\boldsymbol{\lambda}\, x \bullet y \sqcap g(x))$ is idempotent. Let us note that $\mathit{lfp}(\boldsymbol{\lambda}\, x \bullet y \sqcap g(x))$ is a better approximation of $\mathit{lfp}(f)$ than $\mathit{llis}(\boldsymbol{\lambda}\, x \bullet x \sqcap g(x))(y)$ which, according to the dual of Theorem 2.5.4.0.2, is the least value in L that can be computed by using $\sqcup$, $\sqcap$, g, and y. As observed in Remark 4.1.1.0.9, we have achieved a better improvement of the over-approximation y of $\mathit{lfp}(f)$ by using 4.1.1.0.1 with $\boldsymbol{\lambda}\, x \bullet y \sqcap f(x)$ than we would have by using 4.1.1.0.7 starting from y. This leads us to come back to Remark 4.1.1.0.9 in order to note that, if $y = DOIS(f, IOIS(f, \bot))$, we have $IOIS(\boldsymbol{\lambda}\, x \bullet y \sqcap f(x), \bot) = y$ whenever the same extrapolation method is used when computing $IOIS(f, \bot)$ and $IOIS(\boldsymbol{\lambda}\, x \bullet y \sqcap f(x), \bot)$ (for instance, by choosing the same $\bar{\nabla}$ in 4.1.2.0.5). So, it is again impossible to improve y.
End of remark.

<u>THEOREM</u> 4.3.2.0.7

Let $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $f, g \in \mathit{mon}(L \to L)$,

$$\{\{\exists y \in L : y \sqsubseteq \mathit{gfp}(f)\} \ \underline{\text{and}}\ \{g \sqsubseteq f\}\}$$

$$\Rightarrow \quad \{y \sqsubseteq \mathit{luis}(\boldsymbol{\lambda}\, x \bullet x \sqcup g(x))(y) \sqsubseteq \mathit{gfp}(\boldsymbol{\lambda}\, x \bullet y \sqcup g(x)) \sqsubseteq \mathit{gfp}(f)\}$$

4.4 BIBLIOGRAPHIC NOTES

Approximation methods of fixpoints that are known in numerical analysis (for instance Amann [1976], Kuhn & MacKinnon [1975], Rockafellar [1976], Scarf [1967], Todd [1976], etc.) are not very useful to us because we are working on non-numerical spaces for which we have no notion of distance. In particular, we do not know how to measure the accuracy of approximations.

The idea of under- and over-approximating a fixpoint is classic in functional analysis (Krasnosel'skii [1964]) and is also seen, with a distinct perspective, in Cousot & Cousot [1977a]. However, examples were only provided for over-approximations, which did not emphasize under-approximations. This is why we have systematically stated the dual of all methods of approximation in Paragraph 4.1. However, in Paragraph 4.2, the dual results were kept implicit.

Paragraph 4.1 on iterative methods of approximation of fixpoints based on convergence acceleration by extrapolation is a formalization of the algorithms used in Cousot & Cousot [1975a, 1976], even though the fact that these algorithms are based on the approximation of the solutions of the equation systems associated with programs is implicit in those works. This idea is explicitly stated in Cousot & Cousot [1977a]. At the same time, M. Sintzoff and A. van Lamsweerde have also designed methods to compute or approximate fixpoints in order to solve their problems of construction or improvement of non-deterministic programs (Sintzoff [1977a, 1977b], van Lamsweerde [1977]) or parallel programs (van Lamsweerde & Sintzoff [1976]).

The formalization of the idea of simplification of equation systems (Paragraph 4.3) by the means of a closure operator (Paragraph 4.2) initially comes from Cousot & Cousot [1975a, 1975b,1976, 1977a] where we were using a pair of upper adjoint functions. The relation between the notion of closure operator is presented in Cousot & Cousot [1977d], Cousot [1977b, 1977c]. In addition to the known results on closure operators we restated in Paragraph 4.2 (also providing references but not proofs), we can quote Achache [1969], Dubreil-Jacotin, Lesieur & Croisot [1963], Dwinger [1954, 1955], Iseki [1951], Ladegaillerie [1973], Monteiro [1945], Monteiro & Ribeiro [1942], Morgado [1960, 1961, 1962a, 1962b, 1963, 1964, 1965a, 1965b, 1966], Ore [1943a, 1943b], Ward [1942]. The only use (apart from ours) of the notion of closure operator that we have found in Computer Science is Scott [1976].

CHAPTER 5.

APPROXIMATE SEMANTIC ANALYSIS OF PROGRAMS AND APPLICATIONS

5. APPROXIMATE SEMANTIC ANALYSIS OF PROGRAMS AND APPLICATIONS

5. APPROXIMATE SEMANTIC ANALYSIS OF PROGRAMS AND APPLICATIONS

In Chapter 3, we reduced the problem of the static analysis of programs to the problem of solving systems of forward and backward semantic equations associated with the program. Some problems of semantic analysis of programs, such as the termination problem (Manna [1974, §4.2]) or the seemingly simpler problem of discovering constant program expressions (Rief & Lewis [1977]) being undecidable, solving these semantic equations cannot be performed automatically. Even solving them manually is often extremely difficult (consider, for instance, the open problem of proving that the following program terminates for any positive integer initial value of n:

```
while  $n \neq 1$  do
    $n :=$ if even($n$) then $n/2$ else $3n + 1$ endif ;
redo.)
```

In practice, one must be content with partial answers to questions about the semantics of a program. For instance, it is easy to prove that the above program terminates for any positive initial value of n which is a power of two, and this provides a partial answer to the problem of the termination of the program. In another application field, a compiler need only answer with "yes" or "don't know" to the question of whether some expression is constant at a given program point. Indeed, if the compiler can prove that the expression is constant, it will compute its value once and for all before the execution but, if it cannot, it will produce machine code to compute the value at execution time.

To perform the approximate semantic analysis of a program, we propose to compute an approximate solution to the forward and backward semantic equations associated with the program. This consists in applying the approximate equation solving methods developed in Chapter 4 to the semantic equations considered in Chapter 3. We show how, given a class of program properties answering a specific problem, the results of Chapter 4 can be used to construct an algorithm to automatically analyze an arbitrary program for this class of properties. We demonstrate the method by constructing a list of example algorithms that perform approximate semantic analyses of programs. This list is in no way exhaustive as we chose to develop a methodology of approximate semantic analysis of programs instead of solving specific problems and providing hasty implementations from which one cannot learn any lesson — which is often the case in computer science research. Moreover, we focus our examples on compilation problems for which no satisfying solution exists yet, as solving these problems has a considerable economical impact.

5.1 BUILDING AN APPROXIMATE PROGRAM ANALYSIS TECHNIQUE FOR A GIVEN CLASS OF SEMANTIC PROPERTIES

In order to build a method for the semantic analysis of programs, we must first state which questions to ask about programs. We then explain how, using the results of Chapter 3, an over- or under-approximation of the solutions of the systems of semantic equations associated with a program can answer these questions. Finally, Chapter 4 provides a panel of methods to solve approximately these semantic equations.

Example

To solve the problem "find the domain of termination of a program π", we can compute $lfp(B_\pi(\boldsymbol{\lambda}\, x \cdot \underline{\text{true}}))_\varepsilon$ (Definition 3.5.0.1 and Theorem 3.5.0.5). A partial answer may consist in discovering subsets of the domain of termination. This kind of approximate answers is then obtained by characterising a sub-domain of termination by a predicate R such that $R \Rightarrow \mathit{lfp}(B_\pi(\boldsymbol{\lambda}\, x \bullet \underline{\text{true}}))_\varepsilon$. Thus, to compute an under-approximation of the least fixpoint of $\boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\, n \bullet [((n \neq 1)\ \underline{\text{and}}\ \underline{\text{even}}(n)\ \underline{\text{and}}\ P(n/2))\ \underline{\text{or}}$ $((n \neq 1)\ \underline{\text{and}}\ \underline{\text{odd}}(n)\ \underline{\text{and}}\ P(3n{+}1))\ \underline{\text{or}}\ (n = 1)]\}$, we first simplify this operator into $\boldsymbol{\lambda}\, P \bullet$ $\{\boldsymbol{\lambda}\, n \bullet [(\underline{\text{even}}(n)\ \underline{\text{and}}\ P(n/2))\ \underline{\text{or}}\ (n = 1)]\}$, which gives $R = \boldsymbol{\lambda}\, n \bullet \{\exists k \geq 0 : n = 2^k\}$. *End of example.*

Although it is sufficient, when considering a manual analysis, to state some general principles to solve approximately semantic equations — which leaves us free to choose the best method according to each individual program — it is necessary, in the case of automatic analysis, to provide one algorithm to solve approximately the semantic equations permitting the analysis of any program. In this case, we plan to choose, for a given problem, a specific kind of answers to this problem. Chapter 3 then provides a way to design an approximate analysis method for this kind of semantic properties.

For instance, let π be a program with n variables $x_1, \ldots, x_n$ with value in $\mathcal{U}$ and α program points $a_1, \ldots, a_\alpha$ where a_ε and a_σ denote respectively the entry and exit program points. To answer the question "what is the domain of possible values of the variables of the program during any execution of the program", we may compute $\mathit{lfp}(F_\pi(\phi))$ (Definition 3.3.0.1 and Theorem 3.3.0.2). Actually, any $P \in (\mathcal{P}_n)^\alpha = (\mathcal{U}^n \to \{\underline{\text{true}}, \underline{\text{false}}\})^\alpha$ such that $\mathit{lfp}(F_\pi(\phi)) \Rightarrow P$ provides a partial answer to the problem by characterising an over-approximation of the domain of possible values at each program point. In general, there exists infinitely many P that are over-approximations of $\mathit{lfp}(F_\pi(\phi))$, but some are more enlightening than others for a given problem. When performing a manual analysis, one guesses intuitively the optimal shape P should take while performing the computation itself. On the contrary, in an automatic analysis, we cannot rely on intuition to drive the computation of the approximate solution. Thus, we suggest to fix, before starting the analysis, the shape that approximate solutions should take. This amounts to choosing *a priori* a subset $\mathcal{R}$ of $(\mathcal{P}_n)^\alpha$ such that an over-approximation P of $\mathit{lfp}(F_\pi(\phi))$ in $\mathcal{R}$ answers partially the given problem in an intuitively satisfying way. In order for any predicate P on $(\mathcal{P}_n)^\alpha$ to enjoy a best over-approximation $\bar{\rho}(P)$ in $\mathcal{R}$, it is necessary and sufficient for $\mathcal{R}$ to be a lower Moore family, i.e., $\bar{\rho}$ should be an upper closure operator on $(\mathcal{P}_n)^\alpha$ such that $\bar{\rho}(\mathcal{P}_n)^\alpha = R$ (Theorems 4.2.2.0.1 and 4.2.2.0.4). When $\mathcal{R}$ is not a lower Moore family, we can construct the least lower Moore family $\bar{\mathcal{R}}$ containing $\mathcal{R}$ (Theorem 4.2.2.0.5). The least monotone operator on $\bar{\mathcal{R}}$ greater than or equal to $F_\pi(\phi)$ restricted to $\bar{\mathcal{R}}$ is then $\bar{\rho} \circ F_\pi(\phi) \circ \bar{\rho}$ (Paragraph 4.2.8) and, according to Theorem 4.3.1.0.1, $\mathit{lfp}(F_\pi(\phi)) \Rightarrow \mathit{lfp}(\bar{\rho} \circ (F_\pi(\phi)) \circ \bar{\rho})$ holds. In a way, choosing a space $\bar{\mathcal{R}}$ of approximate properties simplifies the system of equations to solve as computing $\mathit{lfp}(\bar{\rho} \circ F_\pi(\phi) \circ \bar{\rho})$ reduces to computing the least solution of the system of approximate equations $X = \bar{\rho}(F_\pi(\phi))(X)$ defined on $\bar{\mathcal{R}}$. This system of approximate equations is simpler than $X = F_\pi(\phi)(X)$ as it is defined on a space $\bar{\mathcal{R}}$ of properties simpler than $(\mathcal{P}_n)^\alpha$. Specifically, $\bar{\mathcal{R}}$ will be chosen so that its elements have a simple machine representation. However, the machine evaluation of $\bar{\rho} \circ F_\pi(\phi)$ poses the same difficulty as that of evaluating $F_\pi(\phi)$ because the elements in $(\mathcal{P}_n)^\alpha$ do not have a canonical representation, which makes the computation hard to automatise due to simplification issues. Thus, we suggest to evaluate $\bar{\rho} \circ F_\pi(\phi)$ manually or, more precisely, to design by hand an algorithm to compute $\bar{\rho} \circ F_\pi(\phi)$ or, when this is too difficult, an algorithm to compute $\bar{F}_\pi \in \mathit{mon}(\bar{\mathcal{R}} \to \bar{\mathcal{R}})$ where $\bar{\rho} \circ F_\pi(\phi) \Rightarrow \bar{F}_\pi$ so that $\mathit{lfp}(\bar{\rho} \circ F_\pi(\phi)) \Rightarrow \mathit{lfp}(\bar{F}_\pi)$ (see Theorem 4.3.0.1 and also 4.3.1.0.1–2). To avoid redoing this work for every program π, we will design an algorithm to directly construct an approximate system of equations $X = \bar{F}_\pi(X)$ associated with an arbitrary program π. Most often, all the approximate properties to compute at each program point will have the same shape, which amounts to choosing $\bar{\rho} \in ((\mathcal{P}_n)^\alpha \to (\mathcal{P}_n)^\alpha)$ of the form $\bar{\rho} = (\bar{\rho}_1, \ldots, \bar{\rho}_\alpha)$ where $\bar{\rho}_1 = \bar{\rho}_2 = \ldots = \bar{\rho}_\alpha = \rho$ and $\rho \in (\mathcal{P}_n \to \mathcal{P}_n)$. In this case, to each rule $X_i = F_i(X_1, \ldots, X_\alpha)$ from Definition 3.3.0.1 corresponds an approximate rule $X_i = \bar{F}_i(X_1, \ldots, X_n)$ where $X_j \in \bar{\mathcal{R}}$ and $\bar{\rho} \circ F_i \Rightarrow \bar{F}_i$. The algorithm to associate with each program π the approximate system of equations $X = \bar{F}(X)$ can be written once and for all using as parameter the machine representation of elements in $\bar{\mathcal{R}}$ and the functions corresponding to elementary rules. Likewise, algorithms to solve approximate systems of equations $X = \bar{F}_\pi(X)$ can be written once and for all. We distinguish the case where $\bar{\mathcal{R}}$ satisfies the ascending chain condition, in which case either Theorem 2.9.1.0.2, 2.9.2.0.2, or 2.9.3.0.9 may be applied to compute iteratively, in a finite number of steps, the least solution of $X = \bar{F}_\pi(X)$ starting from the infimum of $\bar{\mathcal{R}}$. In the case where the convergence does not hold naturally, Remark 4.1.1.0.9 suggests we use an increasing iteration sequence with over-approximation (4.1.1.0.1) followed with a decreasing iteration sequence with over-approximation (4.1.1.0.7). This algorithm is parametrized by an upper widening (4.1.2.0.5) and a lower narrowing (4.1.2.0.16) that must be chosen for each application.

To sum up, in order to define an *algorithm for the approximate analysis of programs for a given class of semantic properties* we suggest to:

1 - choose a closure operator (parametrized by the number n of variables) on $\mathcal{P}_n$, which defines a sub-space $\bar{\mathcal{R}}$ of approximate properties,

2 - use this closure operator and the rules defining the systems of forward or backward semantic equations to design, by hand, the rules defining the approximate systems of equations,

3 - write the algorithm that associates with each program the approximate systems of (forward and/or backward) equations,

4 - write an algorithm to solve these systems (maybe using convergence acceleration methods, when the convergence is not guaranteed to occur naturally in a finite number of steps).

We first provide very simple examples to illustrate the method in detail. We then provide more complex examples related to concrete problems. Specifically, Paragraphs 5.6 and 5.7 develop and demonstrate the backward semantic analysis method as well as the method of combining forward and backward semantic analyses.

5.2 EXAMPLE IN AUTOMATICALLY DISCOVERING THE SIGN OF THE NUMERIC VARIABLES OF A PROGRAM USING A FORWARD SEMANTIC ANALYSIS WITH OVER-APPROXIMATION

The goal of this very simple example is to demonstrate in an intuitive way our technique of approximate analysis of the semantic properties of programs.

5.2.1 Defining a space of approximate properties by an upper closure operator

To study the sign of the values of the variables of a program π, it is necessary to discover, at each program point, invariants of the form $\underset{i=1}{\overset{n}{\underline{\text{AND}}}}(x_i \ r_i \ 0)$, where r_i is an inequality relation $\geq$ or $\leq$. Given this specific class of semantic properties, we now construct the corresponding upper closure operator.

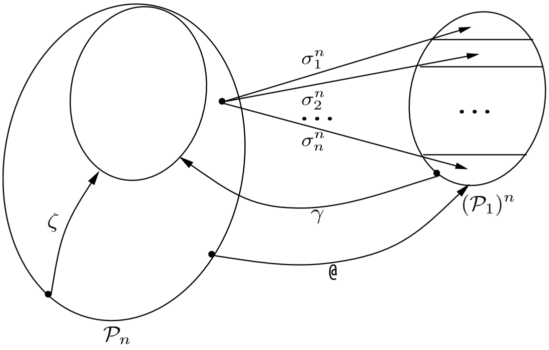
We will often encounter the case where the space of approximate properties does not expose the relationships between variables $x_1, \ldots, x_n$. For instance, an over-approximation of $P = \boldsymbol{\lambda}\,(x, y) \cdot [(x \geq y > 0) \ \underline{\text{or}} \ (x = y = 0)]$ exposing only relations on a single variable may be $\boldsymbol{\lambda}\,(x, y) \cdot (x \geq 0 \ \underline{\text{and}} \ y \geq 0)$. To define such an approximation by an upper closure operator, we proceed as follows:

For all $j = 1, \ldots, n$, let:

$$\sigma_j^n \quad \in \quad \mathcal{P}_n \to \mathcal{P}_1 \qquad \text{where } \mathcal{P}_n = (\mathcal{U}^n \to \mathcal{B}) \text{ and } \mathcal{B} = \{\underline{\text{true}}, \underline{\text{false}}\}$$

$$
\begin{aligned}
\sigma_j^n &= \boldsymbol{\lambda} P \bullet [\boldsymbol{\lambda} x \bullet \exists (v_1, \ldots, v_{j-1}, v_{j+1}, \ldots, v_n) \in \mathcal{U}^{n-1} : \\
&\qquad\qquad\qquad\qquad P(v_1, \ldots, v_{j-1}, x, v_{j+1}, \ldots, v_n)] \\
@ &\in \mathcal{P}_n \to (\mathcal{P}_1)^n \\
@ &= \boldsymbol{\lambda} P \bullet (\sigma_1^n(P), \sigma_2^n(P), \ldots, \sigma_n^n(P)) \\
\gamma &\in (\mathcal{P}_1)^n \to \mathcal{P}_n \\
\gamma &= \boldsymbol{\lambda} (P_1, \ldots, P_n) \bullet [\boldsymbol{\lambda} (x_1, \ldots, x_n) \bullet (\underline{\mathrm{AND}}_{j=1}^{n} P_j(x_j))] \\
\zeta &\in \gamma \circ @ \in (\mathcal{P}_n \to \mathcal{P}_n) \\
\zeta &= \boldsymbol{\lambda} P \bullet [\boldsymbol{\lambda} (x_1, \ldots, x_n) \bullet (\underline{\mathrm{AND}}_{j=1}^{n} \sigma_j^n(P)(x_j))]
\end{aligned}
$$

Graphically, we get:



We check that ζ is an upper closure operator on $\mathcal{P}_n$ (Definition 4.2.1.0.1) and that $\zeta(\mathcal{P}_n)$ is the set of predicates on $x_1, \ldots, x_n$ that do not express any relationship between these variables, that is, $\zeta(\mathcal{P}_n)$ is fully isomorphic to $(\mathcal{P}_1)^n = (\mathcal{U} \to \{\underline{\mathrm{true}}, \underline{\mathrm{false}}\})^n$ by the complete isomorphism γ with inverse @.

As even predicates on a single variable can still be very complex, we introduce an upper closure operator η on $\mathcal{P}_1$ so as to keep only the properties that are of interest for a given problem. For instance, to study the sign of the values of the variables, we may choose predicates of the form $\boldsymbol{\lambda} x \bullet (x = \Omega)$, $\boldsymbol{\lambda} x \bullet (x \geq 0 \ \underline{\text{or}}\ x = \Omega)$, $\boldsymbol{\lambda} x \bullet (x > 0 \ \underline{\text{or}}\ x = \Omega)$, $\boldsymbol{\lambda} x \bullet (x \leq 0 \ \underline{\text{or}}\ x = \Omega)$, $\boldsymbol{\lambda} x \bullet (x < 0 \ \underline{\text{or}}\ x = \Omega)$. We assume that $\Omega \in \mathcal{U}$ is the value of uninitialized variables. We chose predicates that are in disjunction with $\boldsymbol{\lambda} x \bullet (x = \Omega)$ as it seems difficult to study the proper initialization of variables based only on their sign. Also note that this set is not a lower Moore family as it does not contain $\boldsymbol{\lambda} x \bullet (x \geq 0 \ \underline{\text{or}}\ x = \Omega) \ \underline{\text{and}}\ \boldsymbol{\lambda} x \bullet (x \leq 0 \ \underline{\text{or}}\ x = \Omega) = \boldsymbol{\lambda} x \bullet (x = 0 \ \underline{\text{or}}\ x = \Omega)$. In this case, the predicate $\boldsymbol{\lambda} x \bullet (x = 0)$ has two possible approximations, i.e., either $\boldsymbol{\lambda} x \bullet (x \geq 0 \ \underline{\text{or}}\ x = \Omega)$ or $\boldsymbol{\lambda} x \bullet (x \leq 0 \ \underline{\text{or}}\ x = \Omega)$. There is no best approximation for the predicate $\boldsymbol{\lambda} x \bullet (x = 0)$ independently from the choice of a program as, for instance, using the approximation $\boldsymbol{\lambda} x \bullet (x \geq 0 \ \underline{\text{or}}\ x = \Omega)$ in:

$$\mathrm{x} := 0 \quad \{\boldsymbol{\lambda} x \bullet (x \geq 0 \ \underline{\text{or}}\ x = \Omega)\} \ ; \quad \mathrm{x} := \mathrm{x} + 1 \quad \{\boldsymbol{\lambda} x \bullet (x \geq 0 \ \underline{\text{or}}\ x = \Omega)\} \ ;$$

results in a finer analysis than using the approximation $\boldsymbol{\lambda} x \bullet (x \leq 0 \ \underline{\text{or}}\ x = \Omega)$:

$$\mathrm{x} := 0 \quad \{\boldsymbol{\lambda} x \bullet (x \leq 0 \ \underline{\text{or}}\ x = \Omega)\} \ ; \quad \mathrm{x} := \mathrm{x} + 1 \quad \{\boldsymbol{\lambda} x \bullet \underline{\text{true}}\} \ ;$$

whereas

$$\mathrm{x} := 0 \quad \{\boldsymbol{\lambda} x \bullet (x \leq 0 \ \underline{\text{or}}\ x = \Omega)\} \ ; \quad \mathrm{x} := \mathrm{x} - 1 \quad \{\boldsymbol{\lambda} x \bullet (x \leq 0 \ \underline{\text{or}}\ x = \Omega)\} \ ;$$

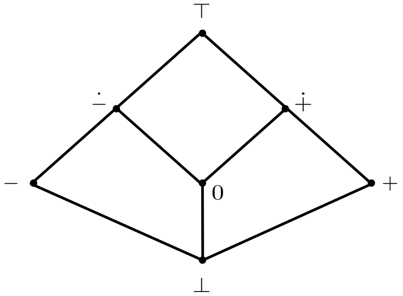
is better than:

$$\mathrm{x} := 0 \quad \{\boldsymbol{\lambda} x \bullet (x \geq 0 \ \underline{\text{or}}\ x = \Omega)\} \ ; \quad \mathrm{x} := \mathrm{x} - 1 \quad \{\boldsymbol{\lambda} x \bullet \underline{\text{true}}\} \ ;$$

To define an approximation of a predicate that would be the best approximation on all programs to be analyzed, there should exist a best choice with respect to the order in the lattice $\mathcal{P}_1$ as this is our only “metric” for the quality of an approximation. This requires the existence of an over-approximation which is smaller than all others, which amounts to choosing a set of approximate properties that forms a lower Moore family (this is always possible according to Theorem 4.2.2.0.5). In our example, we will then define:

$$
\begin{array}{lll}
\eta & = & \boldsymbol{\lambda} P \bullet \underline{\text{case}} \\
 & & \quad \begin{array}{llll}
P & \Rightarrow & \boldsymbol{\lambda} x \bullet (x = \Omega) & \rightarrow \ \boldsymbol{\lambda} x \bullet (x = \Omega) \ ; \\
P & \Rightarrow & \boldsymbol{\lambda} x \bullet (x = 0 \ \underline{\text{or}} \ x = \Omega) & \rightarrow \ \boldsymbol{\lambda} x \bullet (x = 0 \ \underline{\text{or}} \ x = \Omega) \ ; \\
P & \Rightarrow & \boldsymbol{\lambda} x \bullet (x > 0 \ \underline{\text{or}} \ x = \Omega) & \rightarrow \ \boldsymbol{\lambda} x \bullet (x > 0 \ \underline{\text{or}} \ x = \Omega) \ ; \\
P & \Rightarrow & \boldsymbol{\lambda} x \bullet (x < 0 \ \underline{\text{or}} \ x = \Omega) & \rightarrow \ \boldsymbol{\lambda} x \bullet (x < 0 \ \underline{\text{or}} \ x = \Omega) \ ; \\
P & \Rightarrow & \boldsymbol{\lambda} x \bullet (x \geq 0 \ \underline{\text{or}} \ x = \Omega) & \rightarrow \ \boldsymbol{\lambda} x \bullet (x \geq 0 \ \underline{\text{or}} \ x = \Omega) \ ; \\
P & \Rightarrow & \boldsymbol{\lambda} x \bullet (x \leq 0 \ \underline{\text{or}} \ x = \Omega) & \rightarrow \ \boldsymbol{\lambda} x \bullet (x \leq 0 \ \underline{\text{or}} \ x = \Omega) \ ; \\
P & \Rightarrow & \boldsymbol{\lambda} x \bullet \underline{\text{true}} & \rightarrow \ \boldsymbol{\lambda} x \bullet \underline{\text{true}} \ ;
\end{array} \\
 & & \underline{\text{endcase}} \ ;
\end{array}
$$

To represent the elements in $\eta(\mathcal{P}_1)$ in a computer, we use the following lattice:

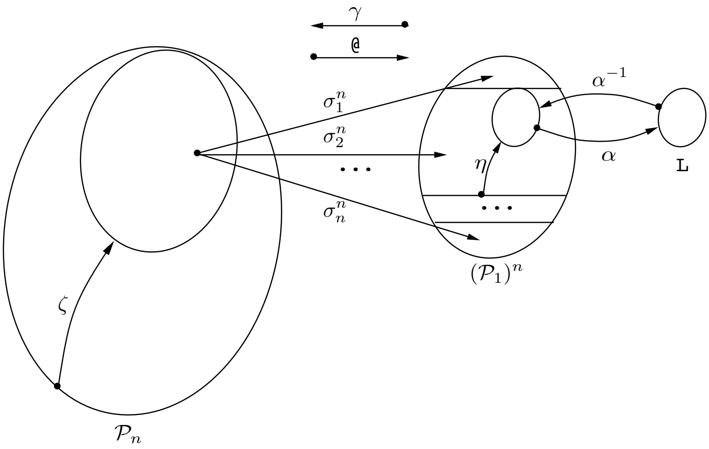


which is isomorphic to the lattice $\eta(\mathcal{P}_1)$ by the α isomorphism:

$$
\begin{array}{lll}
\alpha & = & \boldsymbol{\lambda} P \bullet \underline{\text{case}} \, P \, \underline{\text{in}} \\
 & & \quad \begin{array}{lll}
\boldsymbol{\lambda} x \bullet (x = \Omega) & \rightarrow & \perp \ ; \\
\boldsymbol{\lambda} x \bullet (x = 0 \ \underline{\text{or}} \ x = \Omega) & \rightarrow & 0 \ ; \\
\boldsymbol{\lambda} x \bullet (x > 0 \ \underline{\text{or}} \ x = \Omega) & \rightarrow & + \ ; \\
\boldsymbol{\lambda} x \bullet (x < 0 \ \underline{\text{or}} \ x = \Omega) & \rightarrow & - \ ; \\
\boldsymbol{\lambda} x \bullet (x \geq 0 \ \underline{\text{or}} \ x = \Omega) & \rightarrow & \dot{+} \ ;
\end{array}
\end{array}
$$

```
    $\boldsymbol{\lambda} x \bullet (x \leq 0$ or $x = \Omega)$   $\rightarrow$   $\dot{-}$ ;
    $\boldsymbol{\lambda} x \bullet$ true                $\rightarrow$   $\top$ ;
  endcase ;
```

the inverse of which will be denoted as α^{-1}. At this stage, the situation looks like this:



The simplest way to go is to perform the same approximation for all program variables. We check easily that:

$$\bar{\eta} \quad = \quad \boldsymbol{\lambda}(P_1, \ldots, P_n) \bullet (\eta(P_1), \eta(P_2), \ldots, \eta(P_n))$$

is an upper closure operator on $(\mathcal{P}_1)^n$ and that $\bar{\eta}(\mathcal{P}_1^n)$ is fully isomorphic to L^n by the following complete isomorphism:

$$\bar{\alpha} \quad = \quad \boldsymbol{\lambda}(P_1, \ldots, P_n) \bullet (\alpha(P_1), \alpha(P_2), \ldots, \alpha(P_n))$$

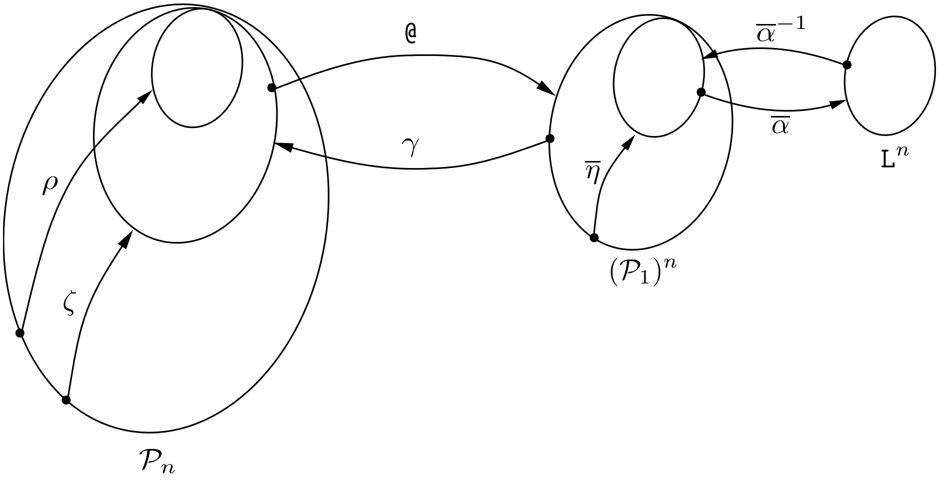
with inverse:

$$\bar{\alpha}^{-1} \quad = \quad \boldsymbol{\lambda}(v_1, \ldots, v_n) \bullet (\alpha^{-1}(v_1), \alpha^{-1}(v_2), \ldots, \alpha^{-1}(v_n))$$

It follows from Theorem 4.2.4.0.6 that we can define a closure operator ρ on $\mathcal{P}_n$ as:

$$\rho \;=\; \gamma \circ \bar{\eta} \circ @ \circ \zeta$$

and $\rho(\mathcal{P}_n)$ is isomorphic to L^n by $\bar{\alpha} \circ @$ with inverse $\gamma \circ \bar{\alpha}^{-1}$. Thus, we obtain an over-approximation of $\mathcal{P}_n$ which is easily represented in a computer:



Given our closure operator ρ on $\mathcal{P}_n$, it is natural to use the same approximation at all program points (although this is in no way mandatory) and we obtain a closure operator $\bar{\rho}$ on $(\mathcal{P}_n)^\alpha$ defined as:

$$\bar{\rho} \;=\; \boldsymbol{\lambda}(P_1, \ldots, P_\alpha) \boldsymbol{\cdot} (\rho(P_1), \rho(P_2), \ldots, \rho(P_\alpha))$$

5.2.2 Rules to construct the approximate system of forward equations associated with a program

It follows from Theorem 3.3.0.2 that the sign of the values of the variables during the execution of a program π starting in a state satisfying ϕ is $\bar{\rho}(\mathit{lfp}(F_\pi(\phi)))$, which we will over-approximate as $\mathit{lfp}(\bar{\rho} \circ F_\pi(\phi) \mid \bar{\rho}((\mathcal{P}_n)^\alpha))$. Given our choice of representing the elements of $\bar{\rho}((\mathcal{P}_n)^\alpha)$ by the elements of $(L^n)^\alpha$ in a computer, we now define by hand a computer representation for $(\bar{\rho} \circ F_\pi(\phi) \mid \bar{\rho}((\mathcal{P}_n)^\alpha))$. To do this independently from the choice of a specific program π, we need to state rules to construct an approximate system of equations $X = \bar{F}_\pi(\bar{\phi})(X)$ on $(L^n)^\alpha$. These rules are justified by proving that $\{(\rho \circ F_\pi(\phi)_j \circ \rho) \Rightarrow (\gamma \circ \bar{\alpha}^{-1} \circ \bar{F}_\pi(\bar{\phi})_j \circ \alpha \circ @)\}$ as, according to Theorem 4.3.0.1, this implies that, for all $j = 1, \ldots, \alpha$, $\{\mathit{lfp}(F_\pi(\phi))_j \Rightarrow \mathit{lfp}(\bar{\rho} \circ F_\pi(\phi) \circ \bar{\rho})_j \Rightarrow \gamma \circ \alpha^{-1}(\mathit{lfp}(\bar{F}_\pi(\bar{\phi}))_j)\}$. We state and justify these rules as follows:

Program entry:

If a_j is the entry point of the program, then $X_j = \bar{\alpha} \circ @ \circ \rho(\phi)$ and this computation is performed manually by the user that directly inputs an element of L^n instead of an element of $\mathcal{P}_n$ as entry condition. In most cases, the entry specification is standard (the simplest cases being the case where no variable is initialized $(\bot, \bot, \ldots, \bot)$ and the case where all variables are initialized with unknown values $(\top, \top, \ldots, \top)$).

Path junction:

If a_j is a program point following a label following the program points $a_{i_1}, \ldots, a_{i_k}$, then $X_j = \bigsqcup_{l=1}^{k} X_{i_l}$ where $\sqcup$ is the join in the lattice $L^n(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$.
To justify this rule, we must prove that (Theorem 4.3.0.1):

$$\forall (X_{i_1}, \ldots, X_{i_k}) \in (L^n)^k, \rho(\underset{l=1}{\overset{k}{\underline{\mathrm{OR}}}} \gamma \circ \bar{\alpha}^{-1}(X_{i_l})) \quad \Rightarrow \quad \gamma \circ \bar{\alpha}^{-1}(\bigsqcup_{l=1}^{k} X_{i_l})$$

(Intuitively, the computations in L^n should be over-approximations of those in $\rho(\mathcal{P}_n)$. Ideally, one could wish them to be equal, which is the case in this example.)
As $\rho = \gamma \circ \bar{\eta} \circ @ \circ \zeta = \gamma \circ \bar{\eta} \circ @ \circ \gamma \circ @ = \gamma \circ \bar{\eta} \circ @$, it is sufficient to prove that:

$$\bar{\eta} \circ @(\underset{l=1}{\overset{k}{\underline{\mathrm{OR}}}} \gamma \circ \alpha^{-1}(X_{i_l})) \quad = \quad \bar{\alpha}^{-1}(\bigsqcup_{l=1}^{k} X_{i_l})$$

This is done one component at a time, which reduces to proving that $\forall m = 1, \ldots, n$ the following holds:

$$\eta \circ \sigma^n_m(\underset{l=1}{\overset{k}{\underline{\mathrm{OR}}}}\, \gamma \circ \alpha^{-1}(X_{i_l})) \quad = \quad \alpha^{-1}(\bigsqcup_{l=1}^{k}(X_{i_l})_m)$$

that is:

$$\eta \circ \sigma^n_m(\underset{l=1}{\overset{k}{\underline{\mathrm{OR}}}}(\boldsymbol{\lambda}\,(x_1, \ldots, x_n) \bullet (\underset{j=1}{\overset{n}{\underline{\mathrm{AND}}}}\, \alpha^{-1}(X_{i_l})_j(x_j)))) \quad = \quad \alpha^{-1}(\bigsqcup_{l=1}^{k}(X_{i_l})_m)$$

as α^{-1} is a complete isomorphism from L to $\eta(\mathcal{P}_1)$ and the join in $\eta(\mathcal{P}_1)$ is given by Theorem 2.3.0.1. We are left to prove that:

$$\eta \circ \sigma^n_m(\boldsymbol{\lambda}\,(x_1, \ldots, x_n) \bullet (\underset{l=1}{\overset{k}{\underline{\mathrm{OR}}}}(\underset{j=1}{\overset{n}{\underline{\mathrm{AND}}}}\, \alpha^{-1}(X_{i_l})_j(x_j)))) \quad = \quad \eta(\underset{l=1}{\overset{k}{\underline{\mathrm{OR}}}}\, \alpha^{-1}((X_{i_l})_m))$$

By eliminating η and replacing σ^n_m with its definition, we get:

$$\begin{aligned} &\boldsymbol{\lambda}\, x \bullet \{\exists(v_1, \ldots, v_{m-1}, v_{m+1}, \ldots, v_n) \in \mathcal{U}^{n-1} : \\ &\qquad\qquad \underset{l=1}{\overset{k}{\underline{\mathrm{OR}}}}((\underset{j=1, j\neq m}{\overset{n}{\underline{\mathrm{AND}}}}\ \alpha^{-1}(X_{i_l})_j(v_j))\ \underline{\text{and}}\ (\alpha^{-1}(X_{i_l})_m)(x))\} \end{aligned}$$

Given $j \in \{1, \ldots, n\} - \{m\}$ and choosing $v_j = \Omega$, $\alpha^{-1}(X_{i_l})_j(v_j)$ always holds as $\alpha^{-1}(X_{i_l})$ has the form $\boldsymbol{\lambda}\, x \bullet ((x = \Omega)\ \underline{\text{or}}\ \ldots)$, so that we get:

$$\underset{l=1}{\overset{k}{\underline{\mathrm{OR}}}}(\alpha^{-1}(X_{i_l})_m)$$

which concludes the proof.

Test:

We consider the case where a_j is the <u>true</u> exit point from a test p. To simplify, we assume that p has the form $\boldsymbol{\lambda}\,(x, y) \bullet (x = y)$, $\boldsymbol{\lambda}\,(x, y) \bullet (x \neq y)$, $\boldsymbol{\lambda}\,(x, y) \bullet (x > y)$, or $\boldsymbol{\lambda}\,(x, y) \bullet (x \geq y)$. (This is in no way a theoretical limitation as all other cases can be handled through composition using Theorem 4.2.6.0.4.(c)–(d). Thus, ($\underline{\text{if}}\ x < y\ \underline{\text{then}}\ I_1\ \underline{\text{else}}\ I_2\ \underline{\text{endif}}$) is equivalent to ($\underline{\text{if}}\ y > x\ \underline{\text{then}}\ I_2\ \underline{\text{else}}\ I_1\ \underline{\text{endif}}$); moreover,

($\underline{\text{if}}\ x = 0\ \underline{\text{then}}\ \ldots$) is equivalent to ($z := 0\ ; \underline{\text{if}}\ x = z\ \underline{\text{then}}\ \ldots$), ($\underline{\text{if}}\ (x = z)\ \underline{\text{and}}\ (y > z)\ \underline{\text{then}}\ \ldots$) is equivalent to ($\underline{\text{if}}\ x = y\ \underline{\text{then}}\ (\underline{\text{if}}\ y > z\ \underline{\text{then}}\ \ldots)\ \underline{\text{endif}}\ ;$), etc. Of course, it is more efficient in practice to consider the tests in their most general form as specified in the language. However, the computations are too tedious to be presented here and do not contribute further to the understanding of the topic.)

Let X_j and $X_i \in L^n$ be associated respectively with the true exit point a_j and to the entry point a_i of a test with condition p. Given the value of X_i, we must compute a value for X_j that is equal to or greater than:

$$\bar{\alpha} \circ @ \circ \rho(\underline{\text{test}}(p)(\gamma \circ \bar{\alpha}^{-1}(X_i)))$$

As $\bar{\alpha} \circ @ \circ \rho = \bar{\alpha} \circ @ \circ \gamma \circ \bar{\eta} \circ @ \circ \zeta = \bar{\alpha} \circ \bar{\eta} \circ @ \circ \gamma \circ @ = \bar{\alpha} \circ \bar{\eta} \circ @$ and according to Definition 3.3.0.1, we must construct an over-approximation of:

$$\begin{array}{ll} & \bar{\alpha} \circ \bar{\eta} \circ @(\boldsymbol{\lambda}\, (x_1, \ldots, x_n) \bullet \{\gamma \circ \bar{\alpha}^{-1}(X_i)(x_1, \ldots, x_n)\ \underline{\text{and}}\ (x_1, \ldots, x_n) \in dom(p)\ \underline{\text{and}} \\ & p(x_1, \ldots, x_n)\}) \\ \Rightarrow & \bar{\alpha} \circ \bar{\eta} \circ @(\boldsymbol{\lambda}\, (x_1, \ldots, x_n) \bullet \{\gamma \circ \bar{\alpha}^{-1}(X_i)(x_1, \ldots, x_n)\ \underline{\text{and}}\ p(x_1, \ldots, x_n)\}) \end{array}$$

because $\bar{\alpha}$, $\bar{\eta}$, and $@$ are monotone. We ignore here the cases where the test is incorrect. Continuing the computation component-wise, we must find, for $q = 1, \ldots, n$, an over-approximation of:

$$\begin{array}{ll} & \alpha \circ \eta \circ \sigma_q^n(\boldsymbol{\lambda}\, (x_1, \ldots, x_n) \bullet \{\gamma \circ \bar{\alpha}^{-1}(X_i)(x_1, \ldots, x_n)\ \underline{\text{and}}\ p(x_1, \ldots, x_n)\}) \\ = & \alpha \circ \eta(\boldsymbol{\lambda}\, x \bullet \{\exists (v_1, \ldots, v_{q-1}, v_{q+1}, \ldots, v_n) \in \mathcal{U}^{n-1} : \\ & \qquad \gamma \circ \bar{\alpha}^{-1}(X_i)(v_1, \ldots, x, \ldots, v_n)\ \underline{\text{and}}\ p(v_1, \ldots, x, \ldots, v_n)\}) \\ = & \alpha \circ \eta(\boldsymbol{\lambda}\, x \bullet \{\exists (v_1, \ldots, v_n) \in \mathcal{U}^{n-1} : \\ & \qquad \underset{l=1, l \neq q}{\overset{n}{\underline{\text{AND}}}}\ (\alpha^{-1}(X_i)_l)(v_l)\ \underline{\text{and}}\ (\alpha^{-1}(X_i)_q)(x)\ \underline{\text{and}}\ p(v_1, \ldots, x, \ldots, v_n)\}) \end{array}$$

- When the variable x_q does not occur in the test, we get that for every $x \in \mathcal{U}^n$, $p(v_1, \ldots, x, \ldots, v_n) = p(v_1, \ldots, \Omega, \ldots, v_n)$. So, putting $v_1 = \ldots = v_{q-1} = v_{q+1} = \ldots = v_n = \Omega$, we can simplify into:

$$= \quad \alpha \circ \eta(\alpha^{-1}(X_i)_q) = (X_i)_q$$

For all the variables that do not occur in the test, we will then pick $(X_j)_q = (X_i)_q$. Obviously, the outcome for the other variables depends on the test.

- When $p = \boldsymbol{\lambda}\,(x_1, \ldots, x_n) \bullet (x_q = x_r)$, we must construct an over-approximation of:

$$\begin{aligned}
&\alpha \circ \eta(\boldsymbol{\lambda}\, x \bullet \{\exists (v_1, \ldots, v_n) \in \mathcal{U}^{n-1} : \underline{\overset{n}{\underset{l=1, l \neq q}{\text{AND}}}}\, (\alpha^{-1}(X_i)_l)(v_l) \ \underline{\text{and}}\ (\alpha^{-1}(X_i)_q)(x) \ \underline{\text{and}} \\
&x = v_r\}) \\
= \quad &\alpha \circ \eta(\boldsymbol{\lambda}\, x \bullet \{\exists v_r \in \mathcal{U} : (\alpha^{-1}(X_i)_q)(x) \ \underline{\text{and}}\ (\alpha^{-1}(X_i)_r)(v_r) \ \underline{\text{and}}\ x = v_r\}) \\
= \quad &\alpha \circ \eta\{\alpha^{-1}(X_i)_q \ \underline{\text{and}}\ \alpha^{-1}(X_i)_r\} \\
= \quad &\{\alpha \circ \eta \circ \alpha^{-1}(X_i)_q \sqcap \alpha \circ \eta \circ \alpha^{-1}(X_i)_r\}
\end{aligned}$$

as η is a closure operator, by Theorem 2.3.0.1, and by the fact that α is a complete isomorphism from $\eta(\mathcal{P}_1)$ to L

$$= \quad (X_i)_q \sqcap (X_i)_r$$

We conclude that if X_i is associated with the entry point a_i of a test $x_q = x_r$, we get $X_j = X_i(x_q \leftarrow x_q \sqcap x_r, x_r \leftarrow x_q \sqcap x_r)$ on the <u>true</u> exit arc, where this notation means that $(X_j)_m = (X_i)_m$ for $m \in \{1, \ldots, n\} - \{q, r\}$, $(X_j)_q = (X_i)_q \sqcap (X_i)_r$ and $(X_j)_r = (X_i)_q \sqcap (X_i)_r$.

- When $p = \boldsymbol{\lambda}\,(x_1, \ldots, x_n) \bullet (x_q \geq x_r)$, we must construct an over-approximation of:

$$\begin{aligned}
&\alpha \circ \eta(\boldsymbol{\lambda}\, x \bullet \{\exists (v_1, \ldots, v_n) \in \mathcal{U}^{n-1} : \\
&\qquad \underline{\overset{n}{\underset{l=1, l \neq q, l \neq r}{\text{AND}}}}\ (\alpha^{-1}(X_i)_l)(v_l) \ \underline{\text{and}}\ (\alpha^{-1}(X_i)_q)(x) \ \underline{\text{and}}\ (\alpha^{-1}(X_i)_r)(v_r) \ \underline{\text{and}}\ x \geq v_r\}) \\
= \quad &\alpha \circ \eta(\boldsymbol{\lambda}\, x \bullet \{\exists v_r : (\alpha^{-1}(X_i)_q)(x) \ \underline{\text{and}}\ (\alpha^{-1}(X_i)_r)(v_r) \ \underline{\text{and}}\ x \geq v_r\})
\end{aligned}$$

We now distinguish several cases:

- If $(\alpha^{-1}(X_i)_r)(v_r) = (v_r = \Omega)$, then $x \geq \Omega$ is undefined and we may consider that the semantics of the language states that neither exit branch from the test is taken. This amounts to saying that the test is <u>false</u> for the <u>true</u> exit branch (and <u>true</u> for the <u>false</u> exit branch):

$$\alpha \circ \eta(\boldsymbol{\lambda}\, x \bullet \underline{\text{false}}) = \alpha(\boldsymbol{\lambda}\, x \bullet (x = \Omega)) = \bot$$

- If $(\alpha^{-1}(X_i)_r)(v_r) = (v_r = 0 \ \underline{\text{or}}\ v_r = \Omega)$, then:

  $$\begin{aligned}
  &= \alpha \circ \eta(\boldsymbol{\lambda}\, x \bullet \{\exists v_r : (\alpha^{-1}(X_i)_q)(x) \ \underline{\text{and}}\ ((v_r = 0 \ \underline{\text{or}}\ v_r = \Omega) \ \underline{\text{and}}\ x \geq v_r)\})\\
  &= \alpha \circ \eta(\boldsymbol{\lambda}\, x \bullet \{\exists v_r : (\alpha^{-1}(X_i)_q)(x) \ \underline{\text{and}}\ ((v_r = 0 \ \underline{\text{and}}\ x \geq v_r) \ \underline{\text{or}}\ (v_r = \Omega \ \underline{\text{and}}\ x \geq v_r))\})
  \end{aligned}$$

  As $x \geq \Omega$ is false, we get:

  $$\begin{aligned}
  &= \alpha \circ \eta(\boldsymbol{\lambda}\, x \bullet \{(\alpha^{-1}(X_i)_q)(x) \ \underline{\text{and}}\ (x \geq 0)\}\\
  &\Rightarrow \alpha \circ \eta(\boldsymbol{\lambda}\, x \bullet \{(\alpha^{-1}(X_i)_q)(x) \ \underline{\text{and}}\ (x \geq 0 \ \underline{\text{or}}\ x = \Omega)\}\\
  &= \alpha \circ \eta \circ \alpha^{-1}(X_i)_q \sqcap \alpha \circ \eta \circ \alpha^{-1}(\dot{+})\\
  &= (X_i)_q \sqcap \dot{+}
  \end{aligned}$$

  We see that we obtain the same result when $(\alpha^{-1}(X_i)_r)(v_r) = (v_r \geq 0 \ \underline{\text{or}}\ v_r = \Omega)$ while, when $(\alpha^{-1}(X_i)_r)(v_r) = (v_r > 0 \ \underline{\text{or}}\ v_r = \Omega)$, we obtain:

  $$(X_i)_q \sqcap +$$

- If $(\alpha^{-1}(X_i)_r)(v_r) = (v_r \leq 0 \ \underline{\text{or}}\ v_r = \Omega)$, we must find an over-approximation of:

  $$\alpha \circ \eta(\boldsymbol{\lambda}\, x \bullet \{\exists v_r : (\alpha^{-1}(X_i)_q)(x) \ \underline{\text{and}}\ (v_r \leq 0 \ \underline{\text{and}}\ x \geq v_r)\}) = (X_i)_q$$

  The result is similar when $(X_i)_r = \top$ or $-$.

We could keep on studying all the different p and provide composition rules to handle tests with $\underline{\text{and}}$, $\underline{\text{or}}$, and $\underline{\text{not}}$. This is unnecessary as, in this example, we can follow our intuition. Nevertheless, we will keep in mind that the theory provides an excellent guide to devise construction rules for approximate systems of equations. Moreover, it provides a proof of their correctness, which in turn proves that the approximate analysis of any program will also be correct.

Assignment:

Let us nevertheless briefly state the rule for the assignment instruction. It is sufficient to replace, in the right-hand side of the assignment, all variables with their sign, do the same for constants with absolute value not equal to 1, and then apply the rule of signs, as in $\bot + \dot{+} = \bot$, $+ + \dot{+} = +$, $+ - 1 = \dot{+}$, $\dot{+} + \dot{-} = \top$, etc. We obtain this way the sign of the expression on the right-hand side of the assignment, which can be assigned to the variable on the left-hand side. To sum up, the equation associated with the assignment:

“$\mathrm{a_j}$: $\mathrm{x_q} := \mathrm{f}(\mathrm{x_1}, \ldots, \mathrm{x_n})$; $\mathrm{a_i}$: …″

is:

$$X_j \quad = \quad X_i(x_q \leftarrow \bar{f}((X_i)_1, \ldots, (X_i)_n))$$

where $\bar{f}$ is constructed from f by applying the rule of signs.

5.2.3 Iterative solving of an approximate system of equations when the convergence occurs naturally with example

Let us now consider an example program (Manna [1974, p. 185]). This program computes $\lceil \sqrt{x} \rceil$ assuming that $x \geq 0$:

```
{1}
          $(y_1, y_2, y_3) := (0, 0, 1)$ ;
{2}
     L:
{3}
          $y_2 := y_2 + y_3$ ;
{4}
          if $y_2 \leq x$ then
{5}
               $(y_1, y_3) := (y_1 + 1, y_3 + 2)$ ;
{6}
               goto L ;
           endif ;
{7}
```

The rules we just provided can be used to associate automatically the following approximate system of equations with this program:

$$\left\{\begin{array}{lcl}
P_1 & = & \langle (x, \dot{+}),\ (y_1, \bot),\ (y_2, \bot),\ (y_3, \bot)\rangle \\
P_2 & = & P_1(y_1 \leftarrow 0, y_2 \leftarrow 0, y_3 \leftarrow +) \\
P_3 & = & P_2 \sqcup P_6 \\
P_4 & = & P_3(y_2 \leftarrow y_2 + y_3) \\
P_5 & = & P_4(y_2 \leftarrow \underline{\text{if}}\ x\ = \bot\ \underline{\text{then}}\ \bot\ \underline{\text{else}}\ y_2 \sqcap (x \sqcup -)\ \underline{\text{endif}}, \\
 & & \qquad x\ \leftarrow \underline{\text{if}}\ y_2 = \bot\ \underline{\text{then}}\ \bot\ \underline{\text{else}}\ x \sqcap (y_2 \sqcup +)\ \underline{\text{endif}}) \\
P_6 & = & P_5(y_1 \leftarrow y_1 + 1, y_3 \leftarrow y_3 + +) \\
P_7 & = & P_4(y_2 \leftarrow \underline{\text{if}}\ x\ = \bot\ \underline{\text{then}} \bot\ \underline{\text{elsif}}\ (x = 0)\ \underline{\text{or}}\ (x = +)\ \underline{\text{or}}\ (x = \dot{+}) \\
 & & \qquad\qquad\qquad \underline{\text{then}}\ y_2 \sqcap +\ \underline{\text{else}}\ y_2\ \underline{\text{endif}}, \\
 & & \qquad x\ \leftarrow \underline{\text{if}}\ y_2 = \bot\ \underline{\text{then}} \bot\ \underline{\text{elsif}}\ (y_2 = 0)\ \underline{\text{or}}\ (y_2 = -)\ \underline{\text{or}}\ (y_2 = \dot{-}) \\
 & & \qquad\qquad\qquad \underline{\text{then}}\ x \sqcap -\ \underline{\text{else}}\ x\ \underline{\text{endif}})
\end{array}\right.$$

Given that L is finite, $(L^n)^\alpha$ is finite and the computation of $\mathit{lfp}(\bar{F}_\pi(\bar{\alpha}))$ through successive approximations converges in a finite number of steps.
This computation gives the following result:

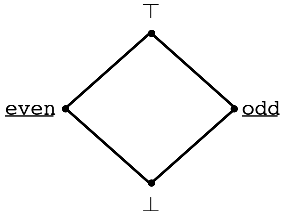
	x	y_1	y_2	y_3
$\{1\}$	$\dot{+}$	$\bot$	$\bot$	$\bot$
$\{2\}$	$\dot{+}$	0	0	$+$
$\{3\}$	$\dot{+}$	$\dot{+}$	$\dot{+}$	$+$
$\{4\}$	$\dot{+}$	$\dot{+}$	$+$	$+$
$\{5\}$	$+$	$\dot{+}$	$+$	$+$
$\{6\}$	$+$	$+$	$+$	$+$
$\{7\}$	$\dot{+}$	$\dot{+}$	$+$	$+$

From this example, we learn the following lesson. When developing an algorithm for the approximate semantic analysis of programs, the choice of a space of approximate properties adapted to solving a given problem is left to the human being. Once this space is chosen, our theory provides the rules to construct an approximate system of equations. As shown in the example, this simply amounts to simplifying the approximate predicate transformers that may occur at the various program points. We left this simplification task to the human being, although one may consider computer assistance to perform these symbolic computations. Once this task is completed once and for all, the computer will be able to provide an approximate system of equations to each particular program and solve it using an iterative method. When the convergence is not guaranteed naturally, human intervention is required to devise once and for all a method of dynamic approximation, which mainly amounts to choosing a widening and a narrowing (see Paragraph 4.1.2).

5.3 EXAMPLE IN AUTOMATICALLY AND APPROXIMATELY DISCOVERING THE PARITY OF THE INTEGER VARIABLES OF A PROGRAM

Another very simple example is the discovery of the parity of the variables of a program. Let n be the number of integer variables $x_1, \ldots, x_n$ in the program. We consider an over-approximation L^n of the complete lattice $\mathcal{P}_n$ of predicates on these variables defined as follows:

L =



$$\gamma \quad = \quad \boldsymbol{\lambda}\,(v_1, \ldots, v_n) \bullet [\boldsymbol{\lambda}\,(x_1, \ldots, x_n) \bullet \underset{j=1}{\overset{n}{\underline{\mathrm{AND}}}}\, \gamma_1(v_j)(x_j)]$$

$$\begin{array}{lllll}
\gamma_1 & = & \boldsymbol{\lambda}\,v \bullet \underline{\mathrm{case}}\,v\,\underline{\mathrm{in}} & & \\
 & & \quad \bot & \rightarrow & \boldsymbol{\lambda}\,x \bullet (x = \Omega)\ ; \\
 & & \quad \underline{\mathrm{even}} & \rightarrow & \boldsymbol{\lambda}\,x \bullet ((x\ \underline{\mathrm{modulo}}\ 2 = 0)\ \underline{\mathrm{or}}\ (x = \Omega))\ ; \\
 & & \quad \underline{\mathrm{odd}} & \rightarrow & \boldsymbol{\lambda}\,x \bullet ((x\ \underline{\mathrm{modulo}}\ 2 = 1)\ \underline{\mathrm{or}}\ (x = \Omega))\ ; \\
 & & \quad \top & \rightarrow & \boldsymbol{\lambda}\,x \bullet \underline{\mathrm{true}}\ ; \\
 & & \underline{\mathrm{endcase}}\ ; & &
\end{array}$$

Remark that γ is an injective complete meet-morphism, and so, Theorem 4.2.7.0.5 provides the matching abstraction function. We do not provide in detail the method to associate an approximate system of equations with a program (5.2.2) but simply give an example (the product of two integers a and b):

```
        r := 0 ;   i := 1 ;
{1}
        until   i = abs(b)   do
{2}
        r := r + a ;   i := i + 1 ;
{3}
        redo ;
{4}
        if b < 0  then  r := −r;  endif ;
{5}
```

Let α, β be the parity of the initial values of variables a and b. The approximate system of equations associated with this program is:

$$\begin{cases} X_1 & = & \langle (a,\alpha),\ (b,\beta),\ (r,\underline{\text{even}}),\ (i,\underline{\text{odd}})\rangle \\ X_2 & = & X_1 \sqcup X_3 \\ X_3 & = & X_2(r \leftarrow r + a, i \leftarrow i + \underline{\text{odd}}) \\ X_4 & = & (X_1 \sqcup X_3)(i \leftarrow i \sqcap b, b \leftarrow i \sqcap b) \\ X_5 & = & X_4 \end{cases}$$

Note that the test $i = \underline{\text{abs}}(b)$ provides some information, i.e., that i and $\underline{\text{abs}}(b)$, and so i and b, have the same parity when the test holds. On the contrary, the tests $i \neq \underline{\text{abs}}(b)$ and $b < 0$ do not provide any parity information. Arithmetic expressions are handled using the rules $\underline{\text{even}} + \underline{\text{even}} \rightarrow \underline{\text{even}}$, $\underline{\text{even}} + \underline{\text{odd}} \rightarrow \underline{\text{odd}}$, $\underline{\text{odd}} + \underline{\text{odd}} \rightarrow \underline{\text{even}}$, etc. As the lattice L^n is finite, the least solution of these equations can be obtained by a finite iteration.

If a is initially $\underline{\text{even}}$ and b is $\underline{\text{odd}}$, we get:

$$\left[\begin{array}{lcl} X_1 & = & \langle (a,\underline{\text{even}}),\ (b,\underline{\text{odd}}),\ (r,\underline{\text{even}}),\ (i,\underline{\text{odd}})\rangle \\ X_2, X_3 & = & \langle (a,\underline{\text{even}}),\ (b,\underline{\text{odd}}),\ (r,\underline{\text{even}}),\ (i,\top)\rangle \\ X_4, X_5 & = & \langle (a,\underline{\text{even}}),\ (b,\underline{\text{odd}}),\ (r,\underline{\text{even}}),\ (i,\underline{\text{odd}})\rangle \end{array}\right.$$

The loss of precision due to the approximation appears in the case where a is initially $\underline{\text{odd}}$ and b is $\underline{\text{even}}$. Indeed we get:

$$\left[\begin{array}{rcl} X_1 & = & \langle (a,\underline{\text{odd}}),\ (b,\underline{\text{even}}),\ (r,\underline{\text{even}}),\ (i,\underline{\text{odd}})\rangle \\ X_2, X_3 & = & \langle (a,\underline{\text{odd}}),\ (b,\underline{\text{even}}),\ (r,\top),\ (i,\top)\rangle \\ X_4, X_5 & = & \langle (a,\underline{\text{odd}}),\ (b,\underline{\text{even}}),\ (r,\top),\ (i,\underline{\text{even}})\rangle \end{array}\right.$$

Actually, when the initial value of a is odd and that of b is even, the final value of r when the program exits is even. To prove this, we would use the fact that when adding an odd number to itself an even number of times, the result is an even number. As this proof explicitly uses the value of b, it cannot be performed when using an approximation that forgets this value.

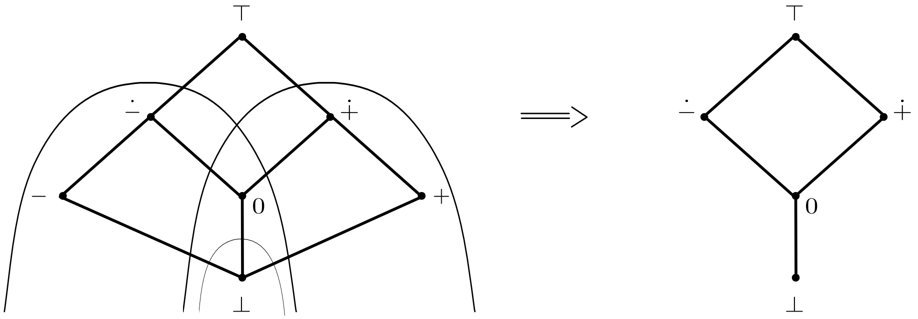
5.4 COMBINING APPROXIMATE ANALYSES: SIGN AND PARITY OF THE INTEGER VARIABLES OF A PROGRAM

The goal of this example is to demonstrate Paragraph 4.2.3 on the combination of approximate analyses of a program by combination of the corresponding closure operators.

For the rule of signs (not considering properties about the strict positivity or negativity) the upper closure operator is:

$$\begin{array}{lll}
\rho_1 & = & \boldsymbol{\lambda} P \bullet \underline{\text{case}} \\
 & & \qquad \begin{array}{lcl}
P \Rightarrow \boldsymbol{\lambda} x \bullet (x = \Omega) & \rightarrow & \bot\ ; \\
P \Rightarrow \boldsymbol{\lambda} x \bullet (x = 0 \ \underline{\text{or}}\ x = \Omega) & \rightarrow & 0\ ; \\
P \Rightarrow \boldsymbol{\lambda} x \bullet (x \geq 0 \ \underline{\text{or}}\ x = \Omega) & \rightarrow & \dot{+}\ ; \\
P \Rightarrow \boldsymbol{\lambda} x \bullet (x \leq 0 \ \underline{\text{or}}\ x = \Omega) & \rightarrow & \dot{-}\ ; \\
P \Rightarrow \boldsymbol{\lambda} x \bullet \underline{\text{true}} & \rightarrow & \top\ ;
\end{array} \\
 & & \quad \underline{\text{endcase}}\ ;
\end{array}$$

Note incidentally that ρ_1 is constructed from η (5.2.1) as the family of principal ideals (4.2.5) generated by the suprema $\dot{+}$, $\dot{-}$, and $\bot$:



To study the parity, we used:

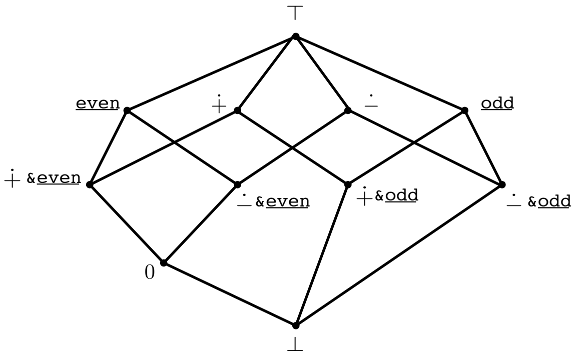
$$
\begin{array}{lll}
\rho_2 & = & \boldsymbol{\lambda} P \bullet \underline{\text{case}} \\
& & \quad \begin{array}{lll}
P \Rightarrow \boldsymbol{\lambda} x \bullet (x = \Omega) & \rightarrow & \bot \ ; \\
P \Rightarrow \boldsymbol{\lambda} x \bullet (x \ \underline{\text{modulo}} \ 2 = 0 \ \underline{\text{or}} \ x = \Omega) & \rightarrow & \underline{\text{even}} \ ; \\
P \Rightarrow \boldsymbol{\lambda} x \bullet (x \ \underline{\text{modulo}} \ 2 = 1 \ \underline{\text{or}} \ x = \Omega) & \rightarrow & \underline{\text{odd}} \ ; \\
P \Rightarrow \boldsymbol{\lambda} x \bullet \underline{\text{true}} & \rightarrow & \top \ ;
\end{array} \\
& & \underline{\text{endcase}} \ ;
\end{array}
$$

It follows from Theorem 4.2.3.0.10 that, to compute $\rho_1 \sqcup \rho_2$, it is sufficient to compute the intersection of the sets $\{\bot, 0, \dot{+}, \dot{-}, \top\}$ and $\{\bot, \underline{\text{even}}, \underline{\text{odd}}, \top\}$, which gives $\{\bot, \top\}$ and is a Moore family corresponding to following the closure operator:

$$
\begin{array}{lll}
\rho_1 \sqcup \rho_2 & = & \boldsymbol{\lambda} P \bullet \underline{\text{case}} \\
& & \quad \begin{array}{lll}
P \Rightarrow \boldsymbol{\lambda} x \bullet (x = \Omega) & \rightarrow & \bot \ ; \\
P \Rightarrow \boldsymbol{\lambda} x \bullet \underline{\text{true}} & \rightarrow & \top \ ;
\end{array} \\
& & \underline{\text{endcase}} \ ;
\end{array}
$$

In this example, the approximation we obtain is so coarse as to seem useless but this is not the case as it allows proving that the program graph is connected!

It follows also from Theorem 4.2.3.0.10 that, in order to obtain $\rho_1 \sqcap \rho_2$, it is sufficient to construct the least Moore family containing $\{\bot, 0, \dot{+}, \dot{-}, \top\}$ and $\{\bot, \underline{\text{even}}, \underline{\text{odd}}, \top\}$. It is obtained by adding all the meets that are not in this set:



After performing a topological sort, we obtain the corresponding closure operator:

$$
\begin{array}{llll}
\rho_1 \sqcap \rho_2 \; = \; \boldsymbol{\lambda} P \bullet & \underline{\text{case}} & & \\
& P \Rightarrow \boldsymbol{\lambda} x \bullet (x = \Omega) & \rightarrow & \bot \, ; \\
& P \Rightarrow \boldsymbol{\lambda} x \bullet (x = 0 \; \underline{\text{or}} \; x = \Omega) & \rightarrow & 0 \, ; \\
& P \Rightarrow \boldsymbol{\lambda} x \bullet (((x \geq 0) \; \underline{\text{and}} \; (x \; \underline{\text{modulo}} \; 2 = 0)) \; \underline{\text{or}} \; (x = \Omega)) & \rightarrow & \dot{+}\& \, \underline{\text{even}} \, ; \\
& P \Rightarrow \boldsymbol{\lambda} x \bullet (((x \leq 0) \; \underline{\text{and}} \; (x \; \underline{\text{modulo}} \; 2 = 0)) \; \underline{\text{or}} \; (x = \Omega)) & \rightarrow & \dot{-}\& \, \underline{\text{even}} \, ; \\
& P \Rightarrow \boldsymbol{\lambda} x \bullet (((x \geq 0) \; \underline{\text{and}} \; (x \; \underline{\text{modulo}} \; 2 = 1)) \; \underline{\text{or}} \; (x = \Omega)) & \rightarrow & \dot{+}\& \, \underline{\text{odd}} \, ; \\
& P \Rightarrow \boldsymbol{\lambda} x \bullet (((x \leq 0) \; \underline{\text{and}} \; (x \; \underline{\text{modulo}} \; 2 = 1)) \; \underline{\text{or}} \; (x = \Omega)) & \rightarrow & \dot{-}\& \, \underline{\text{odd}} \, ; \\
& P \Rightarrow \boldsymbol{\lambda} x \bullet ((x \; \underline{\text{modulo}} \; 2 = 0) \; \underline{\text{or}} \; (x = \Omega)) & \rightarrow & \underline{\text{even}} \, ; \\
& P \Rightarrow \boldsymbol{\lambda} x \bullet (x \geq 0 \; \underline{\text{or}} \; x = \Omega) & \rightarrow & \dot{+} \, ; \\
& P \Rightarrow \boldsymbol{\lambda} x \bullet (x \leq 0 \; \underline{\text{or}} \; x = \Omega) & \rightarrow & \dot{-} \, ; \\
& P \Rightarrow \boldsymbol{\lambda} x \bullet ((x \; \underline{\text{modulo}} \; 2 = 1) \; \underline{\text{or}} \; (x = \Omega)) & \rightarrow & \underline{\text{odd}} \, ; \\
& P \Rightarrow \boldsymbol{\lambda} x \bullet \underline{\text{true}} & \rightarrow & \top \, ; \\
& \underline{\text{endcase}} \, ; & &
\end{array}
$$

The benefit of this combination of two analyses is that it gives better results than each analysis performed separately. Consider, for instance, the following program (Manna

[1974, p. 179]) to compute $y^3 = x_1^{x_2}$ (using the convention $0^0 = 1$) for any integer x_1 and any natural number x_2:

```
        $\langle y_1, y_2, y_3\rangle := \langle x_1, x_2, 1\rangle$ ;
{1}
        until  $y_2 = 0$  do
{2}
            if odd($y_2$) then
{3}
                $\langle y_2, y_3\rangle := \langle y_2 - 1, y_1 * y_3\rangle$ ;
{4}
            else
{5}
                $\langle y_1, y_2\rangle := \langle y_1 * y_1, y_2/2\rangle$ ;
{6}
            endif ;
{7}
        redo ;
{8}
```

If we perform only a sign analysis (corresponding to the closure operator ρ_1), we obtain for the variable y_2:

	{1}	{2}	{3}	{4}	{5}	{6}	{7}	{8}
y_2	$\dot{+}$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	0

If we perform only a parity analysis (corresponding to the closure operator ρ_2), we get:

	{1}	{2}	{3}	{4}	{5}	{6}	{7}	{8}
y_2	$\top$	$\top$	odd	even	even	$\top$	$\top$	even

Intersecting the results of these two separate analyses does not provide any further information on the sign of y_2. On the contrary, combining these two analyses into an analysis corresponding to the closure operator $\rho_1 \sqcap \rho_2$ enables the rule $(\dot{+} \,\&\, \underline{\text{odd}}) - 1 =$

$(\dot{+}\ \&\ \underline{\text{even}})$, which improves the result on the sign of y_2:

	{1}	{2}	{3}	{4}	{5}	{6}	{7}	{8}
y_2	$\dot{+}$	$\dot{+}$	$\dot{+}\ \&\ \underline{\text{odd}}$	$\dot{+}\ \&\ \underline{\text{even}}$	$\dot{+}\ \&\ \underline{\text{even}}$	$\dot{+}$	$\dot{+}$	0

This example teaches us the following lesson: when planning to analyse several disjoint properties on programs — that is, properties corresponding to closure operators ρ_1 and ρ_2 that cannot be compared — it is always better — for both speed and precision — to combine them using the methods of Paragraph 4.2.3. In all generality, Theorem 4.2.8.0.5 proves that the analysis corresponding to $\rho_1 \sqcap \rho_2$ gives a better result than intersecting the results of the separate analyses corresponding to ρ_1 and ρ_2.

5.5 CLASSIC PROGRAM OPTIMISATION TECHNIQUES

A large amount of techniques to optimise programs, such as determining which variables are live at each program point, are purely syntactic. They correspond to checking program properties that are not related to the values of the variables. Others, such as constant propagation, are semantic. In both cases, they can be reduced to the problem of solving a system of equations associated with the program.

5.5.1 Boolean techniques to optimise programs

5.5.1.1 Live variables of a program

A variable is “live” at a program point (Aho & Ullman [1977], Hecht & Ullman [1973], Kennedy [1971], Kennedy [1976], Ullman [1975]) if its value is used along some execution path of the program starting from this point. (For instance, in $\underline{\text{begin}}\ x := 1\ ; y := 2\ ; \{1\} y := x + 1\ \underline{\text{end}}$, x is live at point $\{1\}$ while y is not. This information is obviously useful to perform register allocation.) Let us denote by $\underline{\text{used}}(b)$, for each program node b, the set of variables the value of which is used in this node, and by $\underline{\text{transparent}}(b)$ the set of variables the value of which is not modified in this node. Then, the set $\underline{\text{live}}(b)$ of live variables at the entry of program node b can be obtained by computing the least solution (with respect to set inclusion $\subseteq$) of the system of fixpoint equations associated with the program by the following backward rule:

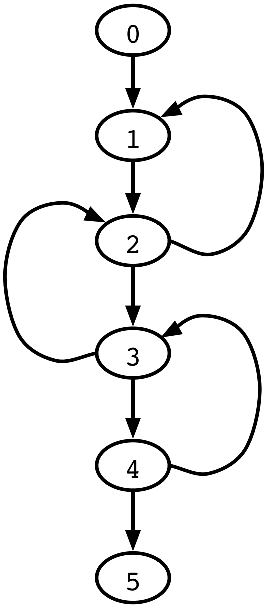
$$\underline{\text{live}}(b) \quad = \quad \underline{\text{used}}(b) \cup \bigcup_{x \in \underline{\text{succ}}(b)} (\underline{\text{transparent}}(b) \cap \underline{\text{live}}(x))$$

and, at exit nodes:

$$\underline{\text{live}}(b) \quad = \quad \emptyset$$

(A variable is live at the entry of a block b if it is used in this block, or if its value is not changed in block b and the variable is used in an execution path starting from the block exit, that is, if it is live at the block exit.)

Consider the following example with two variables α and β, and vectors used and transparent defined syntactically as follows:



node	0	1	2	3	4
used	$\emptyset$	$\{\beta\}$	$\emptyset$	$\emptyset$	$\{\alpha\}$
transparent	$\emptyset$	$\{\alpha,\beta\}$	$\{\alpha,\beta\}$	$\{\alpha,\beta\}$	$\{\alpha\}$

The least solution of the above system of equations is:

node	0	1	2	3	4	5
live	$\emptyset$	$\{\alpha,\beta\}$	$\{\alpha,\beta\}$	$\{\alpha,\beta\}$	$\{\alpha\}$	$\emptyset$

In the iterative computation, live is initialized to used, which is smaller than the least fixpoint. In a computer, we would represent these sets as boolean vectors with length the number of program variables.

5.5.1.2 Available expressions

An expression is available at a program point (Cocke [1970], Hecht & Ullman [1973], Morel & Renvoise [1974], Schaefer [1973], Ullman [1974],Urschler [1974]) if the value of the expression has been computed in the past and the value of no argument in the expression has been modified since the expression has been evaluated last. If the value of an available expression has been saved, it does not need to be recomputed.

We associate with each program node b the set $\underline{\text{transparent}}(b)$ of elementary program expressions such that the value of none of their arguments is modified in this node. Let $\underline{\text{loc–available}}(b)$ be the set of elementary program expressions that are locally available at node b, that is, are evaluated in the node while none of their arguments are modified. This information can be computed at each program point using simple syntactic criteria. The set $\underline{\text{available}}(b)$ of expressions that are available at the exit of every program point is then given by the greatest solution (with respect to set inclusion $\subseteq$) of the system of equations associated with a program using the following forward rule:

$$\underline{\text{available}}(b) \quad = \quad \underline{\text{loc–available}}(b) \cup \bigcap_{x \in \underline{\text{pred}}(b)} (\underline{\text{transparent}}(b) \cap \underline{\text{available}}(x))$$

while, at the entry nodes, we have:

$$\underline{\text{available}}(b) \quad = \quad \emptyset$$

(An expression is available at a block exit if its value is computed within the block, or if it is available at the block entry and none of its arguments is modified within the block.)

Consider, for instance, the following program template:

```
1:   begin
2:        I := . . . ;   J := . . . ;
3:        K := I + J ;
4:        . . .
5:        if . . . then
6:             I := . . . ;
7:             K := I + J ;
               goto 4 ;
          else
8:             . . .
9:             if . . . K . . . then
                    goto 10 ;
               endif ;
               goto 4 ;
          endif ;
10:  end .
```

We get:

b	1	2	3	4	5	6	7	8	9
transparent(b)		$\emptyset$	$\{I+J\}$	$\{I+J\}$	$\{I+J\}$	$\emptyset$	$\{I+J\}$	$\{I+J\}$	$\{I+J\}$
loc–available(b)		$\emptyset$	$\{I+J\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{I+J\}$	$\emptyset$	$\emptyset$
available(b)	$\emptyset$	$\emptyset$	$\{I+J\}$	$\{I+J\}$	$\{I+J\}$	$\emptyset$	$\{I+J\}$	$\{I+J\}$	$\{I+J\}$

5.5.2 Non-boolean techniques to optimise programs: the example of constant propagation

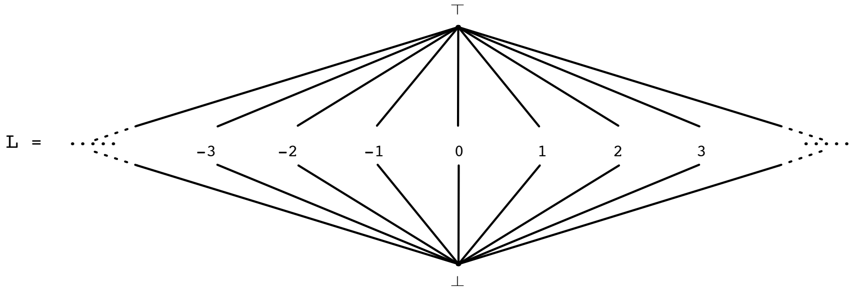
Classic non-boolean techniques are rather scarce (Aho & Ullman [1977]), the most widespread being certainly “constant propagation” (**?** [1976], Kam & Ullman [1977], Kildall [1973], Rief & Lewis [1977]). It is presented as a symbolic execution of the program where a variable can optionally take the symbolic value $\top$ (meaning: not constant) when, at the junction of several execution paths, the variable might not have the same constant value on all these paths. On the following example:

```
        a := 1 ;   b := 2 ;   c := 3 ;   d := 3 ;   e := 0 ;
{1}
        while  ...  do
{2}
            b := 2 * a ;   d := d + 1 ;   e := e − a ;
{3}
            a := b − a ;   c := e + d ;
{4}
        redo ;
```

this symbolic execution gives:

	a	b	c	d	e	
{1}	1	2	3	3	0	
{2}	1	2	3	3	0	
{3}	1	2	3	4	-1	
{4}	1	2	3	4	-1	
{2}	1	2	3	$\top$	$\top$	← As the values of d and e changed at this point.
{3}	1	2	3	$\top$	$\top$	
{4}	1	2	$\top$	$\top$	$\top$	← As c is the sum of two non-constant values.
{2}	1	2	$\top$	$\top$	$\top$	
{3}	1	2	$\top$	$\top$	$\top$	
{4}	1	2	$\top$	$\top$	$\top$	← The symbolic execution stops when the values of the variables at all program points do not change anymore.

We may view this symbolic execution as solving by chaotic iterations an approximate system of equations constructed from the upper closure operator ρ defined as follows (see 5.2.1):



We do not present in detail the rules to construct the approximate equations, but only give the system associated with our running example:

$$\left\{ \begin{array}{lcl} P_1 & = & (a = 1, b = 2, c = 3, d = 3, e = 0) \\ P_2 & = & P_1 \sqcup P_4 \\ P_3 & = & P_2(b \leftarrow 2 * a, d \leftarrow d + 1, e \leftarrow e - a) \\ P_4 & = & P_3(a \leftarrow b - a, c \leftarrow e + d) \end{array} \right.$$

The lattice $(L^5)^4$ is infinite but it satisfies the ascending chain condition, so, we can solve the approximate system of equations by iteration in a finite number of steps. If we choose a chaotic strategy corresponding to the program graph, we retrieve the classic presentation in terms of symbolic execution, which gives:

$$\left[\begin{array}{lcl} P_1 & = & (a = 1, b = 2, c = 3, d = 3, e = 0) \\ P_2, P_3, P_4 & = & (a = 1, b = 2, c = \top, d = \top, e = \top) \end{array} \right.$$

A compiler may optimise the program by replacing a and b with the values 1 and 2, and by eliminating the variables a and b from the object program. Note that, due to the static approximation we chose, we cannot discover that, at line $\{3\}$, $e+d=3$ holds, and so, that c is constant and equals 3 in the program. We will present in Paragraph 5.8 a less coarse but more costly approximation that can discover automatically the loop invariant $\{a=1, b=2, c=3, e \leq 0, d+e=3\}$.

Note that we discover at each program point invariants of the form $\{(a=i)$ or $(a=\Omega)\}$. If at the corresponding point in the object program the variable a is replaced with the value i, the source and object programs will not be equivalent in the case where a may not be initialized at this point in the source program. To avoid this behavior, we would choose instead a closure operator of the following kind:

```
$\boldsymbol{\lambda} P \bullet$ case
        $P = \boldsymbol{\lambda} x \bullet$ false $\rightarrow \bot$ ;
        $P = \boldsymbol{\lambda} x \bullet (x = \Omega) \rightarrow \Omega$ ;
        $P \Rightarrow \boldsymbol{\lambda} x \bullet (x = i) \rightarrow i$ ;
        $P \Rightarrow \boldsymbol{\lambda} x \bullet$ true $\rightarrow \top$ ;
    endcase ;
```

(defining this closure operator is sufficient to fully define the corresponding analysis method).

5.6 AUTOMATICALLY DISCOVERING THE TYPE OF THE VARIABLES OF A PROGRAM

In classic programming languages, the type of a variable at a given program point is often a sub-type of the declared type (even though this sub-type may not exist explicitly in the language definition). For instance, a variable with global type "pointer to a record of type R" may have locally the sub-type "non-nil pointer to a record of type *R*." This information can be critical in the code generation phase, for instance in order to avoid testing dynamically that an indirect access to the record is valid. Likewise, in other languages such as APL (Iverson [1962]) or SETL (Schwartz [1973]), there is no declaration and the type of the objects manipulated by the program must be computed at execution time. The language implementation is based on interpretation and, to design a compiler, one must first and foremost solve the problem of statically determining the type of the objects manipulated by the programs.

5.6.1 Handling pointers

In programming languages such as PASCAL (Jensen & Wirth [1975]) or LIS (Ichbiah, Rissen, Héliard & Cousot [1974]), using variables with a pointer type to access indirectly to a memory block requires the machine to check that the pointer is not nil beforehand. Most compilers delay this check until the program is executed. These tests have a negligible cost at run-time when implemented using memory protection mechanisms. However, this solution suffers from the fact that programming errors are discovered very late, and it cannot be used in system implementation programming languages such as LIS as programs may run in supervisor mode. In this case, performing tests at run-time is costly and a static analysis of the program is mandatory to reduce this cost.

As pointers give access to dynamically allocated memory, this poses the problem for the compiler of discovering which objects can be accessed or modified through them. The following example illustrates this problem:

```
          type E = record A, B : integer  end ;
          var P, Q : ref E ;
          var X, Y : integer ;
          ...
    {1}   X := P.A + P.B ;
    {2}   Q.A := O ;
```

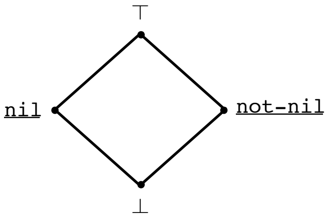
{3} $Y := P.A + P.B$;

If the compiler can prove statically that P and Q do not reference the same record at line {2}, then the expression $P.A + P.B$ is available at line {3} and does not need to be recomputed. If, however, the compiler cannot prove that P and Q point to distinct records, it must assume that assigning the value O to $Q.A$ may also modify $P.A$, and so, the expression $P.A + P.B$ must be recomputed at line {3}. To solve this problem, we suggest in Paragraph 5.6.1.2 an analysis to compute a partition of pointer variables into groups such that two variables in distinct groups can never reference indirectly the same object.

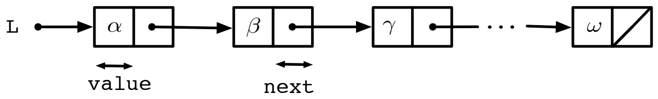
(Note that this issue is not specific to the use of pointers as it also arises with array indexes, which also permit dynamic memory accesses. In that case, the problem is even more complex (see 5.8) as arrays can implement explicit computations of memory addresses that are forbidden for pointers, at least in high-level languages.)

5.6.1.1 Nil and non-nil pointers

This example is similar to the applications of Paragraphs 5.2 and 5.3, and so, we will not present it in detail. We choose as approximate properties of pointers the following lattice:



The following program (Jensen & Wirth [1975]) looks for integer n in a linked list of integers:



```
        pt := L ;    b := true ;
{1}
        while   (pt ≠ nil and b)   do
{2}
             if pt ↑ . value = n then
                  b := false ;
{3}
              else
{4}
                  pt := pt ↑ . next ;
{5}
              endif ;
        redo ;
```

The associated equation system is:

$$\begin{cases} P_1 = \top & \text{(list } L \text{ may be empty or non-empty)} \\ P_2 = (P_1 \sqcup P_5) \sqcap \underline{\text{non–nil}} & \\ P_3 = P_2 & \\ P_4 = P_2 & \\ P_5 = P_4 \sqcup \top & \end{cases}$$

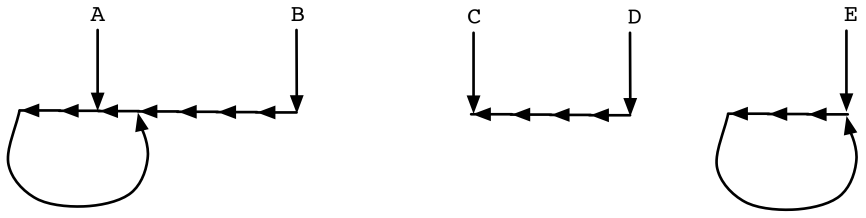
and its solution is:

	{1}	{2}	{3}	{4}	{5}
pt	$\top$	$\underline{\text{non–nil}}$	$\underline{\text{non–nil}}$	$\underline{\text{non–nil}}$	$\top$

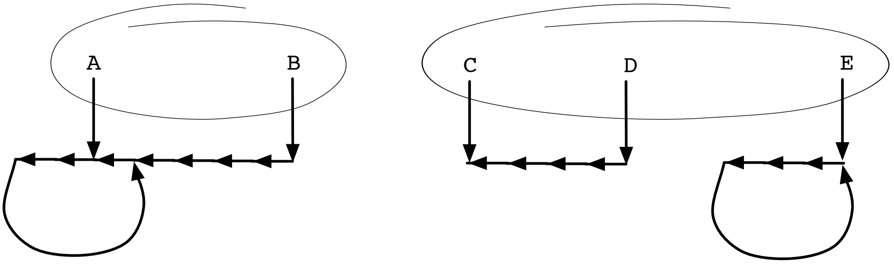
When using pt to access a record, it is never nil, and so, all the pointer uses in this program are correct.

5.6.1.2 Pointers referencing distinct records

In classic programming languages, the only answer to the question "which objects can be referenced by a pointer variable?" is that a given pointer can only reference records of a same type. The notion of "domain" in LIS (Ichbiah, Rissen, Héliard & Cousot [1974]) that also inspired that of "collection" in EUCLID (Lampson et al. [1976]) allows a finer partitioning of the set of pointer variables referencing records of the same type. This improvement is not satisfactory because global declarations are not precise enough to describe the specific situation at each program point. We thus propose to construct at each program point a partition of the set of pointer variables referencing records of a given type. Two pointer variables will be equivalent if they can reference the same record. Two pointer variables belonging to distinct equivalence classes may not reference, even indirectly, the same object. For instance, one partition that describes the following situation:



may be:



which is denoted as $/A, B/C, D, E/$.

We now briefly present the rules to construct a system of equations associated with a program:

Pointer that is either nil or the only pointer to reference a newly allocated record:

$$P \quad = \quad \{/X, X_1, \ldots, X_p/Y_1, \ldots, Y_q/ \ldots /Z_1, \ldots, Z_r/\}$$

(partition before instruction)

X := nil ;
X := Y ;   (where Y is nil thanks to 5.6.1.1)
if X = nil then ...
allocate(X) ;

$$\varepsilon(X, P) \quad = \quad \{/X/X_1, \ldots, X_p/Y_1, \ldots, Y_q/ \ldots /Z_1, \ldots, Z_r/\}$$

(partition after instruction)

Pointer assignment:

$$P \quad = \quad \{/X, X_1, \ldots, X_p/Y, Y_1, \ldots, Y_q/ \ldots /Z_1, \ldots, Z_r/\}$$

$$\text{X} \uparrow .\text{A} \underbrace{\ldots \uparrow .B}_{\text{optional}} \text{:=Y} \underbrace{\uparrow .C \ldots \uparrow .D}_{\text{optional}} \;;$$

$$P \sqcup \{/X, Y/X_1/\ldots/X_p/Y_1/\ldots/Y_q/\ldots/Z_1/\ldots/Z_r/\} \qquad \text{(partition join)}$$

$$= \{/X, X_1, \ldots, X_p, Y, Y_1, \ldots, Y_q/\ldots/Z_1, \ldots, Z_r/\}$$

(As X and Y reference the same object, they are put in the same partition. As we do not know the precise organisation of data, we ignore the possibility that the instruction may split a partition into two disjoint partitions.)

$$P = \{/X, X_1, \ldots, X_p/Y_1, \ldots, Y_q/\ldots/Z_1, \ldots, Z_r/\}$$

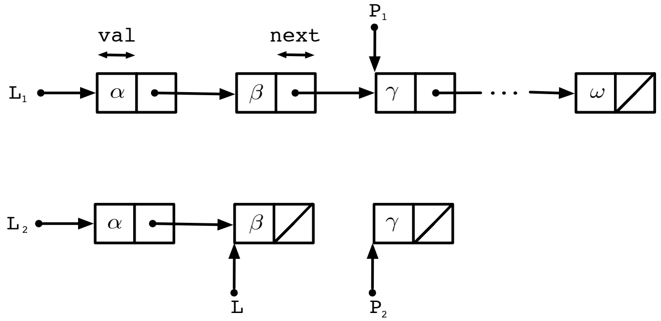
$$\text{X} := \text{Y}\underbrace{\uparrow .C \ldots \uparrow .D}_{\text{optimal}} ;$$

$$\varepsilon(X, P) \sqcup \{/X, Y/X_1/\ldots/Z_r/\}$$

$$= \{/X_1, \ldots, X_p/X, Y, Y_1, \ldots, Y_q/\ldots/Z_1, \ldots, Z_r/\}$$

(After the assignment, X cannot reference, even indirectly, an object referenced by $X_1, \ldots, X_p$.)

Let us consider, as an example, the following program that duplicates a linked list:



```
          procedure copy($L_1$ : list ; var $L_2$ : list) ;
              var $P_1, P_2, L$ : list ;
          begin
{1}
              $P_1 := L_1$ ; $L_2 :=$ nil ; $L :=$ nil ;
{2}
              while  $P_1 \neq$ nil  do
{3}
                  allocate($P_2$) ; $P_2 \uparrow .$value$:= P_1 \uparrow .$value ; $P_2 \uparrow .$next$:=$ nil ;
{4}
                  if $L =$ nil then
{5}
                      $L_2 := P_2$ ;
{6}
                  else
{7}
                      $L \uparrow .$next$:= P_2$ ;
{8}
                  endif ;
{9}
                  $L := P_2$ ; $P_1 := P_1 \uparrow .$next ;
{10}
              redo ;
{11}
          end ;
```

The corresponding approximate system of equations is:

$$\left\{\begin{array}{rcl}
S_1 & = & \{/L_1, L_2/P_1, P_2, L/\} \\
S_2 & = & \varepsilon(L, \varepsilon(L_2, \varepsilon(P_1, S_1) \sqcup \{/P_1, L_1/L/L_2/P_2/\})) \\
S_3 & = & S_2 \sqcup S_{10} \\
S_4 & = & \varepsilon(P_2, S_3) \\
S_5 & = & \varepsilon(L, S_4) \\
S_6 & = & \varepsilon(L_2, S_5) \sqcup \{/L_2, P_2/L/L_1/P_1/\} \\
S_7 & = & S_4 \\
S_8 & = & S_7 \sqcup \{/L, P_2/L_1/L_2/P_1/\} \\
S_9 & = & S_6 \sqcup S_8 \\
S_{10} & = & \varepsilon(L, S_9) \sqcup \{/L, P_2/L_1/L_2/P_1/\} \\
S_{11} & = & \varepsilon(P_1, S_2 \sqcup S_{10})
\end{array}\right.$$

At line {1}, the parameters and the local variables belong to disjoint partitions. At lines {3} and {7}, the tests $P_1 \neq \underline{\text{nil}}$ and $L \neq \underline{\text{nil}}$ do not provide any information on the partitioning of pointer variables. At line {4}, the assignment $P_2 \uparrow .\text{value}{:=}P_1 \uparrow .\text{value}$ of a non-pointer value and the deep modification $P_2 \uparrow .\text{next} := \underline{\text{nil}}$ in the partition of P_2 are ignored. At line {10}, $P_1 := P_1 \uparrow .\text{next}$ is ignored as the instruction cannot make P_1 reference indirectly any record that was not already reachable from P_1.

As the number of pointer variables is finite, the lattice of partitions of their names is finite and the system can be solved by iteration in a finite number of steps, starting from the infimum $\{/L_1/L_2/L/P_1/P_2/\}$. We obtain:

$$\left[\begin{array}{rcl}
S_1 & = & \{/L_1, L_2/P_1, P_2, L/\} \\
S_2 & = & \{/L_1, P_1/L_2/P_2/L/\} \\
S_3 & = & \{/L_1, P_1/L_2, P_2, L/\} \\
S_4 & = & \{/L_1, P_1/L_2, L/P_2/\} \\
S_5 & = & \{/L_1, P_1/L_2/P_2/L/\} \\
S_6 & = & \{/L_1, P_1/L_2, P_2/L/\} \\
S_7 & = & \{/L_1, P_1/L_2, L/P_2/\} \\
S_8 & = & \{/L_1, P_1/L_2, L, P_2/\} \\
S_9 & = & \{/L_1, P_1/L_2, P_2, L/\} \\
S_{10} & = & \{/L_1, P_1/L_2, P_2, L/\} \\
S_{11} & = & \{/L_1/P_1/L_2, P_2, L/\}
\end{array}\right.$$

Thus, we have proved automatically by simple means that, although L_1 and L_2 can reference the same record before calling the procedure *copy*(L_1, L_2), after this call, they cannot reference, even indirectly, the same object.

More details on applications to pointer handling (in particular in the case of records with variants) can be found in Cousot & Cousot [1977b]. Note that such information can also be useful to garbage collectors (in particular when it is explicit, in which case it is necessary to check that the freed memory is not referenced).

5.6.2 Discovering the type of objects in a program in a very high-level language without declarations

We select here an example which is too simple to demonstrate the power of a language such as SETL (Schwartz [1973]) but is adequate to present an application of the results from Paragraph 3.6.

Consider a language with base types integers (int), reals (real), character strings (char), and no declaration. Now, consider the following program:

```
{1}
          s := 0 ;
{2}
     L:
{3}
          read(x, y) ;
{4}
          if x ≥ 0 then
{5}
               x := (x modulo 2) + y ;
{6}
               s := s + x ;
{7}
           endif ;
{8}
          if (y modulo x) ≠ 0 then
{9}
               goto L ;
           endif ;
{10}
```

In this program, the type of a variable can be any element in the set L of subsets of $\mathcal{T} = \{\underline{\mathrm{int}}, \underline{\mathrm{real}}, \underline{\mathrm{char}}\}$, with the convention that, at execution time, an object of type $t \subseteq \mathcal{T}$ must have one of the elementary types in t. The language operators are polymorphic, but may not be defined for all types of arguments. The read procedure is defined for all types of arguments. The comparison $x \geq 0$ is only defined if x has numeric value. The operation x modulo y is only defined if x and y are integers. The + operation is defined for all types of arguments: it may denote the concatenation of two character strings, the concatenation of a string with a numeric value converted to a string, or the sum of two numeric values. If x has type α and y has type β, then $x + y$ has type $\alpha \bar{+} \beta$ defined as:

$$\begin{array}{lll} \bar{+} & = & \boldsymbol{\lambda}\, \alpha, \beta \bullet \underline{\text{case}}\, \alpha, \beta\, \underline{\text{in}} \\ & & \qquad \begin{array}{ccccl} \underline{\text{char}} & , & ? & \rightarrow & \underline{\text{char}}\ ; \\ ? & , & \underline{\text{char}} & \rightarrow & \underline{\text{char}}\ ; \\ \underline{\text{real}} & , & ? & \rightarrow & \underline{\text{real}}\ ; \\ ? & , & \underline{\text{real}} & \rightarrow & \underline{\text{real}}\ ; \\ ? & , & ? & \rightarrow & \underline{\text{int}}\ ; \end{array} \\ & & \quad \underline{\text{endcase}}\ ; \end{array}$$

5.6.2.1 Approximate system of forward equations

The approximate system of forward equations $P = \bar{F}(\bar{\phi})(P)$, where $\bar{F}(\bar{\phi}) \in mon((L^3)^{10} \rightarrow (L^3)^{10})$, associated with the program is:

$$\left\{ \begin{array}{lll} P_1 & = & \bar{\phi} \\ P_2 & = & P_1(s \leftarrow \{\underline{\text{int}}\}) \\ P_3 & = & P_2 \cup P_9 \\ P_4 & = & P_3(x \leftarrow \top, y \leftarrow \top) \\ P_5 & = & P_4(x \leftarrow x \cap \{\underline{\text{int}}, \underline{\text{real}}\}) \\ P_6 & = & P_5(x \leftarrow \{\underline{\text{int}}\} \bar{\bar{+}} y) \\ P_7 & = & P_6(s \leftarrow s \bar{\bar{+}} x) \\ P_8 & = & P_7 \cup P_4(x \leftarrow x \cap \{\underline{\text{int}}, \underline{\text{real}}\}) \\ P_9 & = & P_8(x \leftarrow x \cap \{\underline{\text{int}}\}, y \leftarrow y \cap \{\underline{\text{int}}\}) \\ P_{10} & = & P_8(x \leftarrow x \cap \{\underline{\text{int}}\}, y \leftarrow y \cap \{\underline{\text{int}}\}) \end{array} \right.$$

To establish P_4, we took into account the ability of read to store into x and y values of arbitrary elementary type. In P_5 and P_8, we take into account the fact that the test $x \geq 0$ is only defined for numeric values of x. Likewise, in P_{10} and P_9, the test $(y \ \underline{\text{modulo}}\ x) \neq 0$ requires that x and y are integers, otherwise the program terminates with an error. Finally, $\bar{\bar{+}} = \boldsymbol{\lambda}(t_1, t_2) \bullet \{\alpha \bar{+} \beta : \alpha \in t_1 \ \underline{\text{and}}\ \beta \in t_2\}$ provides the set of possible types for the result of $+$ given the set of possible types for its parameters.

5.6.2.2 Approximate system of backward equations

The approximate system of backward equations $P = \bar{B}(\bar{\psi})(P)$, where $\bar{B}(\bar{\psi}) \in mon((L^3)^{10} \rightarrow (L^3)^{10})$, associated with the program is:

$$\left\{\begin{array}{lcl}
P_1 & = & P_2(s \leftarrow \top) \\
P_2 & = & P_3 \\
P_3 & = & P_4(x \leftarrow \top, y \leftarrow \top) \\
P_4 & = & (P_5 \cup P_8)(x \leftarrow x \cap \{\underline{\text{int}}, \underline{\text{real}}\}) \\
P_5 & = & P_6(x \leftarrow x \cap \{\underline{\text{int}}\}, y \leftarrow y \cap \underline{\text{plus2}}(x, \underline{\text{int}}, y) \\
P_6 & = & P_7(x \leftarrow x \cap \underline{\text{plus2}}(s, \top, x), s \leftarrow s \cap \underline{\text{plus1}}(s, \top, x)) \\
P_7 & = & P_8 \\
P_8 & = & (P_9 \cup P_{10})(x \leftarrow x \cap \{\underline{\text{int}}\}, y \leftarrow y \cap \{\underline{\text{int}}\}) \\
P_9 & = & P_3 \\
P_{10} & = & \bar{\psi}
\end{array}\right.$$

To establish P_4 and P_8, we take into account the fact that the tests must be well-defined. In P_6, we take into account the fact that $(x \ \underline{\text{modulo}}\ 2)$ is well-defined only if x is integer, in which case the result is an integer. Moreover, we use:

$$\begin{array}{lcl}
\underline{\text{plus1}}(t_0, t_1, t_2) & = & \{\beta \in t_1 : \{\exists \alpha \in t_0, \exists \gamma \in t_2 : \alpha = \beta + \gamma\}\} \\
\underline{\text{plus2}}(t_0, t_1, t_2) & = & \{\gamma \in t_2 : \{\exists \alpha \in t_0, \exists \beta \in t_1 : \alpha = \beta + \gamma\}\}
\end{array}$$

Indeed, given the possible types t_0 of the result and the possible types $t_1, \ldots, t_n$ of the n arguments before an operation $f(x_1, \ldots, x_n)$ has been applied (and these are the same as the types after the operation has been carried out for the arguments that are not modified by the instruction), we can deduce that only some of the combinations could actually have occurred before the operation was applied for the type of the result to have the expected type. For instance, for $1 + y$ to have $\underline{\text{char}}$ type, it is mandatory that y has type $\underline{\text{plus2}}(\underline{\text{char}}, \underline{\text{int}}, \top) = \underline{\text{char}}$.

5.6.2.3 Gist of the solving method

Let F_π and B_π be the systems of semantic equations associated with the program π above, and ϕ and ψ be the entry and exit specifications. (In our example, we will take $\phi = \boldsymbol{\lambda}(x, y, s) \cdot (x = y = s = \Omega)$ and $\psi = \boldsymbol{\lambda}(x, y, s) \cdot \underline{\text{true}}$.) We wish to find an over-approximation of $\mathit{lfp}(F_\pi(\phi))$ $\underline{\text{and}}$ $\mathit{lfp}(B_\pi(\psi))$, that is, to find for each program point a super-set of the values of the variables that can be encountered on an arbitrary execution path that starts at the entry point in a state satisfying the entry condition ϕ, then reaches this point (Proposition 3.3.0.2), and finally finishes in a state satisfying the exit specification ψ (Proposition 3.5.0.2). We are equipped with $\bar{\phi} \sqsupseteq \phi$, $\bar{\psi} \sqsupseteq \psi$, $\bar{F}(\bar{\phi}) \sqsupseteq F_\pi(\phi)$, and $\bar{B}(\bar{\psi}) \sqsupseteq B_\pi(\psi)$.

We propose to compute the limit of the decreasing sequence $P^1 = \mathit{lfp}(\bar{F}_\pi(\bar{\phi}))$, $P^2 = \mathit{lfp}(\boldsymbol{\lambda} X \cdot P^1 \ \underline{\text{and}}\ \bar{B}(\bar{\psi})(X))$, …, $P^{2k+1} = \mathit{lfp}(\boldsymbol{\lambda} X \cdot P^{2k} \ \underline{\text{and}}\ \bar{F}(\bar{\phi})(X))$, $P^{2k+2} = \mathit{lfp}(\boldsymbol{\lambda} X \cdot P^{2k+1} \ \underline{\text{and}}\ \bar{B}(\bar{\psi})(X))$, …. It follows from Theorem 4.3.2.0.5 that this sequence is decreasing and, as in our case the space $(L^3)^{10}$ of approximate properties satisfies the descending chain condition, this sequence is stationary after a finite rank. To prove that this choice is indeed valid, it is sufficient to prove that every term is greater than $(\mathit{lfp}(F_\pi(\phi))$ $\underline{\text{and}}$ $\mathit{lfp}(B_\pi(\psi)))$:

We get $P^1 = \mathit{lfp}(\bar{F}(\bar{\phi})) \sqsupseteq \mathit{lfp}(F_\pi(\phi)) \sqsupseteq (\mathit{lfp}(F_\pi(\phi)) \ \underline{\text{and}}\ \mathit{lfp}(B_\pi(\psi)))$. As $\boldsymbol{\lambda} X \cdot [P^1 \ \underline{\text{and}}\ \bar{B}(\bar{\psi})(X)] \sqsupseteq \boldsymbol{\lambda} X \cdot [\mathit{lfp}(F_\pi(\phi)) \ \underline{\text{and}}\ B_\pi(\psi)(X)]$, we deduce by monotonicity that $P^2 = \mathit{lfp}(\boldsymbol{\lambda} X \cdot [P^1 \ \underline{\text{and}}\ \bar{B}(\bar{\psi})(X)]) \sqsupseteq \mathit{lfp}(\boldsymbol{\lambda} X \cdot [\mathit{lfp}(F_\pi(\phi)) \ \underline{\text{and}}\ B_\pi(\psi)(X)])$, which is equal, according to Proposition 3.7.0.1.(c), to $(\mathit{lfp}(F_\pi(\phi)) \ \underline{\text{and}}\ \mathit{lfp}(B_\pi(\psi)))$. The induction step is similar and uses Proposition 3.7.0.1.(e)–(f).

5.6.2.4 Example

On our example, we get:

$$\begin{aligned}\bar{\phi} &= ((x = \emptyset), (y = \emptyset), (s = \emptyset))\\ \bar{\psi} &= ((x = \top), (y = \top), (s = \top))\end{aligned}$$

$$P^1 = \mathit{lfp}(\bar{F}(\bar{\phi}))$$

	$\{1\}$	$\{2\}$	$\{3\}$	$\{4\}$	$\{5\}$	$\{6\}$	$\{7\}$	$\{8\}$	$\{9\}$	$\{10\}$
x	$\emptyset$	$\emptyset$	$\{\underline{\text{int}}\}$	$\top$	$\{\underline{\text{int}}, \underline{\text{real}}\}$	$\top$	$\top$	$\{\underline{\text{int}}, \underline{\text{real}}\}$	$\{\underline{\text{int}}\}$	$\{\underline{\text{int}}\}$
y	$\emptyset$	$\emptyset$	$\{\underline{\text{int}}\}$	$\top$	$\top$	$\top$	$\top$	$\top$	$\{\underline{\text{int}}\}$	$\{\underline{\text{int}}\}$
s	$\emptyset$	$\{\underline{\text{int}}\}$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

$$P^2 = \mathit{lfp}(\boldsymbol{\lambda} X \bullet P^1 \cap \bar{B}(\bar{\psi}))$$

	$\{1\}$	$\{2\}$	$\{3\}$	$\{4\}$	$\{5\}$	$\{6\}$	$\{7\}$	$\{8\}$	$\{9\}$	$\{10\}$
x	$\emptyset$	$\emptyset$	$\{\underline{\text{int}}\}$	$\{\underline{\text{int}}\}$	$\{\underline{\text{int}}\}$	$\{\underline{\text{int}}\}$	$\{\underline{\text{int}}\}$	$\{\underline{\text{int}}\}$	$\{\underline{\text{int}}\}$	$\{\underline{\text{int}}\}$
y	$\emptyset$	$\emptyset$	$\{\underline{\text{int}}\}$	$\{\underline{\text{int}}\}$	$\{\underline{\text{int}}\}$	$\{\underline{\text{int}}\}$	$\{\underline{\text{int}}\}$	$\{\underline{\text{int}}\}$	$\{\underline{\text{int}}\}$	$\{\underline{\text{int}}\}$
s	$\emptyset$	$\{\underline{\text{int}}\}$	$\top$	$\{\underline{\text{int}}\}$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

$$P^3 = \mathit{lfp}(\boldsymbol{\lambda} X \bullet P^2 \cap \bar{F}(\bar{\phi}))$$

	$\{1\}$	$\{2\}$	$\{3\}$	$\{4\}$	$\{5\}$	$\{6\}$	$\{7\}$	$\{8\}$	$\{9\}$	$\{10\}$

x	∅	∅	{int}	{int}	{int}	{int}	{int}	{int}	{int}	{int}
y	∅	∅	{int}	{int}	{int}	{int}	{int}	{int}	{int}	{int}
s	∅	{int}	{int}	{int}	{int}	{int}	{int}	{int}	{int}	{int}

Finally, $P^4 = lfp(\boldsymbol{\lambda} X \cdot P^3 \cap \bar{B}(\bar{\psi}))$ gives the same result, which proves that x, y, and s must be declared as integers in order for the program to terminate correctly and not crash with an error. An intuitive proof of this result lies in remarking that, if the value of x or y is not integer, then the test $(y\ \underline{\text{modulo}}\ x) \neq 0$ is eventually evaluated, which leads to an error for real or string arguments. As x and y must be integers, it follows that s must also be an integer as it is initialized and incremented with an integer value. Indeed, these two proof steps appear clearly in P^2 and P^3.

5.7 APPROXIMATION TECHNIQUES FOR INFINITE SPACES OF APPROXIMATE PROPERTIES: EXAMPLE IN AUTOMATICALLY DISCOVERING AN INTERVAL OF VALUES FOR THE NUMERIC VARIABLES OF A PROGRAM

In a PASCAL program (Jensen & Wirth [1975]), it is possible to declare an integer that lies between two numeric bounds. In addition to the benefit of more precise declarations, this also allows a compiler to optimise memory allocation (in particular for arrays of words, half-words, and bytes that can be easily handled on most machines). A drawback is the prohibitive cost of checking the consistency of programs at run-time, as the hardware seldom provides hardwired overflow checking for non word or double-word arithmetic.

To highlight the cost of run-time checks, we review some figures (on a CDC 6600 computer running the SCOPE 3.4 operating system) given by Wirth [1976] for the following programs: (1) Computing 2^k and $1/2^k$ for $k = 1..90$, (2) Finding all numbers between 1 and 1000 the square of which are palindromic, (3) Quicksort (Hoare [1962]), (4) Solving the 8-queens problem (Wirth [1971]), and (5) Computing the first 1000 prime numbers. For each program, we provide the number "n" of program lines in PASCAL, the number "a" of array accesses in the program source, and the execution times "t_a" and "t_s" in *ms* when running the program respectively with and without dynamic array index bound checking:

Program	n	a	t_a	t_s
(1)	34	9	916	813
(2)	16	2	3466	2695
(3)	26	7	4098	2861
(4)	34	15	1017	679
(5)	30	8	1347	1061

This time gain (around 20% on average, but sometimes much higher) causes programmers to almost exclusively use the compiler option to remove run-time checks. This can have surprising results, as shown in the following example:

```
begin
    x : 0..255 ;
    x := 1 ;
    while   x ≠ 0   do
        x := x + 1 ;
    redo ;
 end.
```

Using a compiler that does not generate checks that, in the loop body, the value of x must lie between 0 and 255, we observed the following behavior: as x is declared with type 0..255, the compiler allocates one byte of memory to x. During program execution, x is naturally incremented by 1 until it reaches the value 255. At this point, during the evaluation of $x := x + 1$, the value 255 in x is loaded into a register. The register value is incremented by 1, which results in the value 256 and does not cause any overflow, as the register is $4-$byte wide. Then, the least significant byte of the register, which is zero, is stored into x. As x has reached the value zero, the loop terminates successfully despite the fact that the program is obviously incorrect.

Such an example clearly shows that checking interval declarations is mandatory. Run-time tests being very costly — especially for programs that are run many times — we would like to eliminate most useless tests using a precise analysis at compilation-time. We now develop a useful model to solve this problem in the case of programming languages, such as PASCAL, where the declarations of integer variables and arrays contain numeric bounds.

5.7.1 Space of approximate properties

Let $\mathcal{Z}$ be the set of positive or negative integers ordered by the natural order $\leq$ and $\mathcal{Z}^\star = \mathcal{Z} \cup \{-\infty, +\infty\}$, where $-\infty \leq -\infty \leq i \leq +\infty \leq +\infty$ for any i in $\mathcal{Z}$. Let L be the complete lattice with infimum $\perp$ and, as other elements, pairs $[a, b]$ where $a, b \in \mathcal{Z}^\star$ and $a \leq b$. The partial order $\sqsubseteq$ on L is defined as $\perp \sqsubseteq x$ for every x in L and $\{[a, b] \sqsubseteq [c, d]\} \Leftrightarrow \{c \leq a \leq b \leq d\}$. The join is defined as $[a, b] \sqcup [c, d] = [\underline{\min}(a, c), \underline{\max}(b, d)]$, $\perp$ is an identity element, and $\top = [-\infty, +\infty]$ is an absorbing element. The meet is defined as $[a, b] \sqcap [c, d] = \underline{\text{if}}\ \underline{\max}(a, c) \leq \underline{\min}(b, d)\ \underline{\text{then}}\ [\underline{\max}(a, c), \underline{\min}(b, d)]\ \underline{\text{else}}\ \perp\ \underline{\text{endif}}$, $\top$ is an identity element, and $\perp$ is an absorbing element. We define the concretization function (4.2.7) as follows:

$$\begin{array}{rcl}
\gamma & \in & L \rightarrow (2^{\mathcal{Z} \cup \{\Omega\}} \rightarrow \{\underline{\text{true}}, \underline{\text{false}}\}) = \mathcal{P}_1 \\
& = & \boldsymbol{\lambda} x \bullet \underline{\text{case}}\, x\, \underline{\text{in}} \\
& & \qquad\qquad \perp \qquad \rightarrow \quad \boldsymbol{\lambda} x \bullet (x = \Omega)\ ;
\end{array}$$

$$[a, b] \quad \rightarrow \quad \boldsymbol{\lambda}\, x \bullet ((a \leq x \leq b) \ \underline{\text{or}}\ (x = \Omega)) \ ;$$

$\underline{\text{endcase}}$;

Let n be the number of program variables, then L^n is an over-approximated image of $\mathcal{P}_n = (2^{\mathcal{Z} \cup \{\Omega\}})^n \rightarrow \{\underline{\text{true}}, \underline{\text{false}}\}$ by the following injective concretization function:

$$\bar{\gamma} \quad = \quad \boldsymbol{\lambda}\, (v_1, \ldots, v_n) \bullet (\boldsymbol{\lambda}\, (x_1, \ldots, x_n) \bullet (\underset{j=1}{\overset{n}{\underline{\text{AND}}}}\, \gamma(v_j)(x_j)))$$

The corresponding abstraction function is defined as:

$$\begin{array}{lll} @ & \in & \mathcal{P}_1 \rightarrow L \\ & = & \boldsymbol{\lambda}\, P \bullet \underline{\text{if}}\ P \Rightarrow \boldsymbol{\lambda}\, x \bullet (x = \Omega) \ \ \underline{\text{then}} \\ & & \qquad\qquad \perp \\ & & \qquad \underline{\text{else}} \\ & & \qquad\qquad [\underline{\min}\{x \in \mathcal{Z} : P(x)\}, \underline{\max}\{x \in \mathcal{Z} : P(x)\}] \\ & & \qquad \underline{\text{endif}} \ ; \end{array}$$

Note that this definition of @ is valid since $\underline{\text{not}}\,(P \Rightarrow \boldsymbol{\lambda}\, x \bullet (x = \Omega))$ implies $\{\exists x \in \mathcal{Z} : P(x)\}$ and since $\underline{\min}(\mathcal{Z}) = -\infty$ and $\underline{\max}(\mathcal{Z}) = +\infty$. Moreover, @ is surjective.

For all $j = 1, \ldots, n$, we define:

$$\begin{array}{lll} \sigma_j^n & \in & \mathcal{P}_n \rightarrow \mathcal{P}_1 \\ & = & \boldsymbol{\lambda}\, P \bullet [\boldsymbol{\lambda}\, x \bullet \{\exists (v_1, \ldots, v_{j-1}, v_{j+1}, \ldots, v_n) \in (\mathcal{Z} \cup \{\Omega\})^n : \\ & & \qquad\qquad P(v_1, \ldots, v_{j-1}, x, v_{j+1}, \ldots, v_n)\}] \end{array}$$

$$\begin{array}{lll} \bar{@} & \in & \mathcal{P}_n \rightarrow L^n \\ & = & \boldsymbol{\lambda}\, P \bullet (@(\sigma_1^n(P)), @(\sigma_2^n(P)), \ldots, @(\sigma_n^n(P))) \end{array}$$

We now prove that $(\bar{\gamma}, \bar{@})$ forms a pair of upper adjoint functions (Definition 4.2.7.0.1).

- To prove that the concretization $\bar{\gamma}$ and the abstraction $\bar{@}$ are monotone, it is sufficient to prove that γ and $@$ are monotone, which is easy as $([a, b] \sqsubseteq [c, d]) \Rightarrow (c \leq a \leq b \leq d) \Rightarrow \boldsymbol{\lambda} x \bullet ((a \leq x \leq b) \ \underline{\text{or}}\ (x = \Omega)) \Rightarrow \boldsymbol{\lambda} x \bullet ((c \leq x \leq d) \ \underline{\text{or}}\ (x = \Omega))$ and, on the other hand, $\underline{\text{not}}\,(P \Rightarrow \boldsymbol{\lambda} x \bullet (x = \Omega))$, $\underline{\text{not}}\,(Q \Rightarrow \boldsymbol{\lambda} x \bullet (x = \Omega))$, and $(P \Rightarrow Q)$ imply $\underline{\min}\{x \in \mathcal{Z} : Q(x)\} \leq \underline{\min}\{x \in \mathcal{Z} : P(x)\} \leq \underline{\max}\{x \in \mathcal{Z} : P(x)\} \leq \underline{\max}\{x \in \mathcal{Z} : Q(x)\}$.
- Let us now prove that $\forall Q \in \mathcal{P}_1, Q \Rightarrow \gamma \circ @(Q)$.
  - If $Q = \boldsymbol{\lambda} x \bullet \underline{\text{false}}$ or $Q = \boldsymbol{\lambda} x \bullet (x = \Omega)$, then $\gamma \circ @(Q) = \gamma(\bot) = \boldsymbol{\lambda} x \bullet (x = \Omega)$, in which case $Q \Rightarrow \boldsymbol{\lambda} x \bullet (x = \Omega)$.
  - Otherwise, we get $\underline{\text{not}}\,(Q \Rightarrow \boldsymbol{\lambda} x \bullet (x = \Omega))$ and

$$\begin{aligned} \gamma \circ @(Q) &= \gamma([\underline{\min}\{x \in \mathcal{Z} : Q(x)\}, \underline{\max}\{x \in \mathcal{Z} : Q(x)\}]) \\ &= \boldsymbol{\lambda} z \bullet ((\underline{\min}\{x \in \mathcal{Z} : Q(x)\} \leq z \leq \underline{\max}\{x \in \mathcal{Z} : Q(x)\}) \ \underline{\text{or}}\ (z = \Omega)) \end{aligned}$$

    and so, $\forall z \in (\mathcal{Z} \cup \{\Omega\})$ and we get $Q(z) \Rightarrow \gamma \circ @(Q)(z)$.
- Let us now prove that $\forall v \in L, v = @ \circ \gamma(v)$.
  - If $v = \bot$, then $@ \circ \gamma(\bot) = @(\boldsymbol{\lambda} x \bullet (x = \Omega)) = \bot$
  - If $v = [a, b]$ where $a, b \in \mathcal{Z}^\star$ and $a \leq b$, then

$$\begin{aligned} @ \circ \gamma([a, b]) &= @(\boldsymbol{\lambda} x \bullet (a \leq x \leq b) \ \underline{\text{or}}\ (x = \Omega)) \\ &= [\underline{\min}\{x \in \mathcal{Z} : a \leq x \leq b\}, \underline{\max}\{x \in \mathcal{Z} : a \leq x \leq b\}] \\ &= [a, b] \end{aligned}$$

- Let $P \in \mathcal{P}_1$ and $v \in L$ then, on the one hand, $(P \Rightarrow \gamma(v))$ implies by monotonicity $@(P) \sqsubseteq @ \circ \gamma(v) = v$ and, on the other hand, $(@(P) \sqsubseteq v)$ implies $P \Rightarrow \gamma \circ @(P) \Rightarrow \gamma(v)$, thus, $(@, \gamma)$ is a pair of upper adjoint functions.
- Let us now prove that $\forall Q \in \mathcal{P}_n, Q \Rightarrow \bar{\gamma} \circ \bar{@}(Q)$.

$$\begin{aligned}\bar{\gamma} \circ \bar{@}(Q) &= \bar{\gamma}(@(\sigma_1^n(Q)), \ldots, @(\sigma_n^n(Q))) \\ &= \boldsymbol{\lambda}\,(x_1, \ldots, x_n) \bullet \underset{j=1}{\overset{n}{\underline{\text{AND}}}}\, \gamma(@(\sigma_j^n(Q)))(x_j)\end{aligned}$$

For any $j, \sigma_j^n(Q) \Rightarrow \gamma \circ @(\sigma_j^n(Q))$, so, $\bar{\gamma} \circ \bar{@}(Q) \Leftarrow \boldsymbol{\lambda}\,(x_1, \ldots, x_n) \bullet (\underset{j=1}{\overset{n}{\underline{\text{AND}}}}\, \sigma_j^n(Q)(x_j)) \Leftarrow$ Q as $Q(x_1, \ldots, x_n) \Rightarrow \underset{j=1}{\overset{n}{\underline{\text{AND}}}}\{\exists(x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n \in (\mathcal{Z} \cup \{\Omega\}) : Q(x_1, \ldots, x_j, \ldots, x_n)\}$

- If $v \in L^n$, then $v = \bar{@} \circ \bar{\gamma}(v)$, indeed:

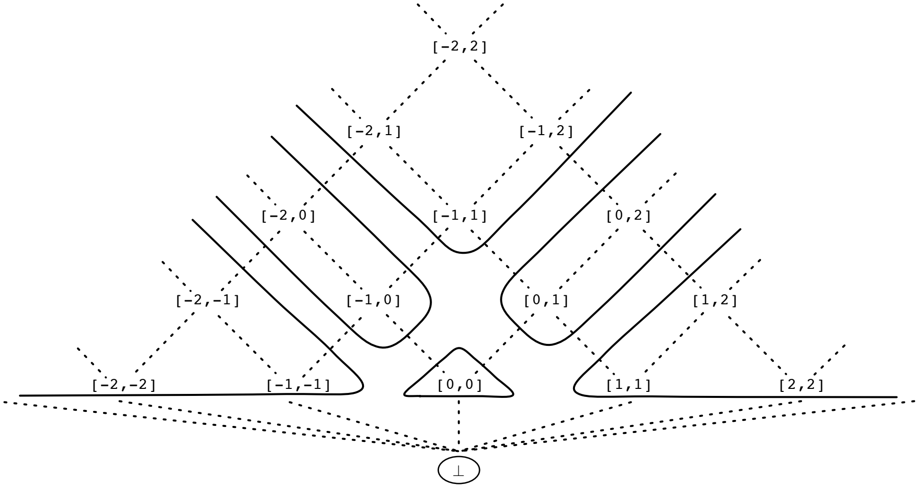
$$\bar{@} \circ \bar{\gamma}(v) \quad = \quad \bar{@}(\boldsymbol{\lambda}\,(x_1, \ldots, x_n) \bullet (\underset{j=1}{\overset{n}{\underline{\text{AND}}}}\, \gamma(v_j)(x_j)))$$

and, for the $j-$th component:

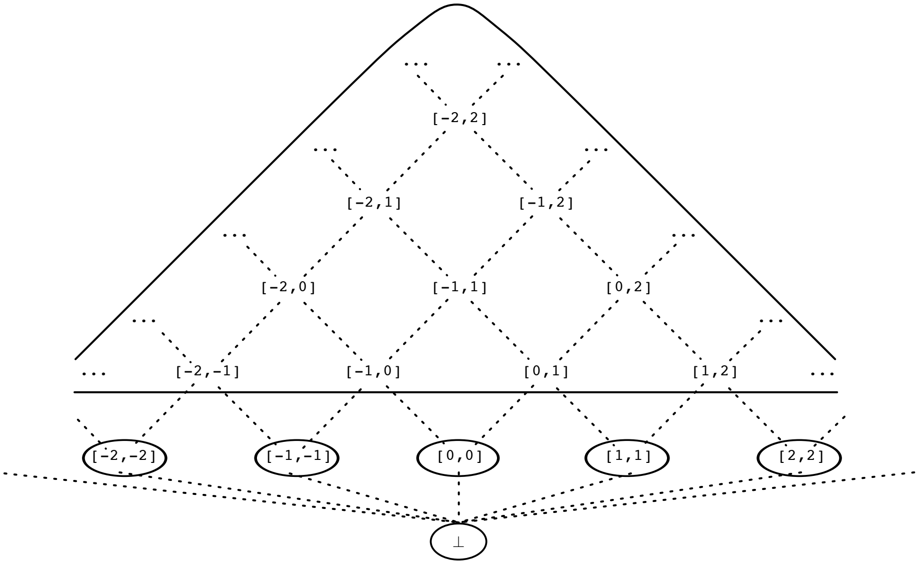
$$(\bar{@} \circ \bar{\gamma}(v))_j \quad = \quad @ \circ \sigma_j^n(\boldsymbol{\lambda}\,(x_1, \ldots, x_n) \bullet (\underset{j=1}{\overset{n}{\underline{\text{AND}}}}\, \gamma(v_j)(x_j))) = @ \circ \gamma(v_j) = v_j$$

- As before, $(\bar{@}, \bar{\gamma})$ is a pair of upper adjoint functions and it follows from 4.2.7.0.3 that $\bar{\gamma} \circ \bar{@}$ is an upper closure operator on $\mathcal{P}_n$, $@$ is a complete join-morphism, and $\bar{\gamma}$ is a complete meet-morphism.

Note that the upper closure operator η used in Paragraph 5.2.1 to discover the sign of the numeric variables of a program can be derived from the upper closure operator $\gamma \circ @$ we just defined by using the join-complete congruence relation (4.2.6) the equivalence classes of which are defined as follows (using the image L of $\gamma \circ @(\mathcal{P}_1)$):



while the upper closure operator used in the constant propagation (5.5.2) can be derived from the join-complete congruence relation defined by the following equivalence classes:



5.7.2 Rules to construct the approximate system of forward equations associated with a program

We present a few rules without giving much detail (these can be found in Cousot & Cousot [1975a]).

Path junction:

$$X_j \quad = \quad \bigsqcup_{i \in \underline{\mathrm{pred}}(j)} X_i$$

Assignment:

We evaluate the right-hand side using interval arithmetic and assign the resulting value to the left-hand side:

$$\begin{array}{l} \{((x = [1, 10]), (y = [-2, 3]))\} \\ \qquad \mathrm{x} := \mathrm{x} + \mathrm{y} + 1 \ ; \\ \{((x = [0, 14]), (y = [-2, 3]))\} \\ \qquad \text{as } [1, 10] + [-2, 3] + [1, 1] = [1 - 2 + 1, 10 + 3 + 1] = [0, 14] \end{array}$$

$$\begin{array}{l} \{((x = [1, +\infty]), (y = [-\infty, 10]))\} \\ \qquad \mathrm{x} := \mathrm{x} + \mathrm{y} + 1 \ ; \\ \{((x = [-\infty, +\infty]), (y = [-\infty, 10]))\} \\ \qquad \text{as } [1, +\infty] + [-\infty, 10] + [1, 1] = [-\infty + 2, +\infty + 11] = [-\infty, +\infty] \end{array}$$

Test:

$$\begin{array}{l} \{((x = [a, b]), (y = [c, d]))\} \\ \underline{\text{if}}\ \mathrm{x} \leq \mathrm{y}\ \underline{\text{then}} \\ \qquad \{((x = [a, b] \sqcap [-\infty, d]), (y = [c, d] \sqcap [a, +\infty]))\} \\ \qquad \ldots \\ \ \underline{\text{else}} \\ \qquad \{((x = [a, b] \sqcap [c + 1, +\infty]), (y = [c, d] \sqcap [-\infty, b - 1]))\} \\ \qquad \ldots \end{array}$$

```
  endif ;

 $\{((x=[a,b]),(y=[c,d]))\}$
 if x = y then
      $\{((x=[a,b]\sqcap[c,d]),(y=[a,b]\sqcap[c,d]))\}$
      ...
  else
      {if $a=b=c$ then $((x=[a,b]),(y=[c+1,d]))$
       elsif $a=b=d$ then $((x=[a,b]),(y=[c,d-1]))$
       elsif $a=c=d$ then $((x=[a+1,b]),(y=[c,d]))$
       elsif $b=c=d$ then $((x=[a,b-1]),(y=[c,d]))$
       else $((x=[a,b]),(y=[c,d]))$
       endif}
      ...
  endif ;
```

(Using the convention: $[a,b]=\perp$ when $b<a$.)

Example 5.7.2.0.1

The equation system associated with the program:

```
           begin  n : 0..1000 ; n : integer ;
                read(n) ;   {$0 \le n \le 1000$ must be checked at run-time}
 {1}
                x := 0 ;
 {2}
      L:
 {3}
                if x ≤ n then
 {4}
                     x := x + 2 ;
 {5}
                     goto L ;
                 endif ;
 {6}
           end.
```

is (denoting $\underline{\mathrm{ub}}(\bot) = +\infty$, $\underline{\mathrm{ub}}([a, b]) = b$, $\underline{\mathrm{lb}}(\bot) = -\infty$, and $\underline{\mathrm{lb}}([a, b]) = a$):

$$\begin{cases} P_1 &=& ((x = \bot), (n = [0, 1000])) \\ P_2 &=& ((x = [0, 0]), (n = P_1(n))) \\ P_3 &=& P_2 \sqcup P_5 \\ P_4 &=& ((x = P_3(x) \sqcap [-\infty, \underline{\mathrm{ub}}(P_3(n))]), (n = P_3(n) \sqcap [\underline{\mathrm{lb}}(P_3(x)), +\infty])) \\ P_5 &=& ((x = P_4(x) + [2, 2]), (n = P_4(n))) \\ P_6 &=& ((x = P_3(x) \sqcap [\underline{\mathrm{lb}}(P_3(n)) + 1, +\infty]), (n = P_3(n) \sqcap [-\infty, \underline{\mathrm{ub}}(P_3(x)) - 1])) \end{cases}$$

End of example.

5.7.3 Solving the approximate system of equations by dynamic approximation

As the lattice L is infinite, it is easy to prove that solving the equations iteratively does not converge in a finite number of steps in the general case. Thus, we use the dynamic approximation techniques introduced in Paragraph 4.1.2.

5.7.3.1 Approximating the least solution using an increasing chaotic iteration sequence with upper widening

To define once and for all an approximation method to make an increasing iteration sequence converge in a finite number of steps, we introduce an upper widening $\bar{\nabla}$ on L (4.1.2.0.4) defined as:

- $\forall x \in L, \bot \mathbin{\bar{\nabla}} x \;=\; x \mathbin{\bar{\nabla}} \bot \;=\; x$
- $[a, b] \mathbin{\bar{\nabla}} [c, d] \;=\; [\underline{\mathrm{if}}\ c < a\ \underline{\mathrm{then}}\ -\infty\ \underline{\mathrm{else}}\ a\ \underline{\mathrm{endif}},$

  $\underline{\mathrm{if}}\ d > b\ \underline{\mathrm{then}}\ +\infty\ \underline{\mathrm{else}}\ b\ \underline{\mathrm{endif}}]$

We can check that this upper widening satisfies the hypotheses from Definition 4.1.2.0.4. Note that $\bar{\nabla}$ is not monotone (see 4.1.2.0.7.(b)). The widening $\bar{\bar{\nabla}}$ on L^n is obtained by computing $\bar{\nabla}$ component-wise. Definition 4.1.2.0.5 can then be applied so that Theorem 4.1.2.0.6 guarantees the convergence of the iterates and the correctness of the approximation.

We demonstrate our method on Example 5.7.2.0.1. The graph of the program being reducible in the sense of Allen and Cocke, we choose $\{3\}$ as head of the circuit (4.1.2.0.2). By convention, $((x = \alpha), (n = \beta))$ will be abbreviated as (α, β). Moreover, we use Remark 4.1.2.0.7.(a):

$$
\begin{array}{lcl}
P_j^0 & = & (\bot, \bot) \qquad \text{for } j = 1, \ldots, 6 \\
P_1^1 & = & (\bot, [0, 1000]) \\
P_2^1 & = & ([0, 0], P_1^1(n)) = ([0, 0], [0, 1000]) \\
P_3^1 & = & P_3^0 \overset{\equiv}{\nabla} (P_2^1 \sqcup P_5^0) = (\bot, \bot) \overset{\equiv}{\nabla} (P_2^1 \sqcup (\bot, \bot)) = ([0, 0], [0, 1000]) \\
P_4^1 & = & (P_3^1(x) \sqcap [-\infty, \underline{\text{ub}}(P_3^1(n))], P_3^1(n) \sqcap [\underline{\text{lb}}(P_3^1(x)), +\infty]) \\
 & = & ([0, 0] \sqcap [-\infty, 1000], [0, 1000] \sqcap [0, +\infty]) = ([0, 0], [0, 1000]) \\
P_5^1 & = & (P_4^1(x) + [2, 2], P_4^1(n)) = ([0, 0] + [2, 2], [0, 1000]) = ([2, 2], [0, 1000]) \\
P_3^2 & = & P_3^1 \overset{\equiv}{\nabla} (P_2^1 \sqcup P_5^1) = ([0, 0] \bar{\nabla} ([0, 0] \sqcup [2, 2]), [0, 1000] \bar{\nabla} ([0, 1000] \sqcup [0, 1000])) \\
 & = & ([0, 0] \bar{\nabla} [0, 2], [0, 1000] \bar{\nabla} [0, 1000]) = ([0, +\infty], [0, 1000]) \\
P_4^2 & = & (P_3^2(x) \sqcap [-\infty, \underline{\text{ub}}(P_3^2(n))], P_3^2(n) \sqcap [\underline{\text{lb}}(P_3^2(x)), +\infty]) \\
 & = & ([0, +\infty] \sqcap [-\infty, 1000], [0, 1000] \sqcap [0, +\infty]) = ([0, 1000], [0, 1000]) \\
P_5^2 & = & (P_4^2(x) + [2, 2], P_4^2(n)) = ([2, 1002], [0, 1000])
\end{array}
$$

As $P_5^2 \sqcup P_2^1 \sqsubseteq P_3^2$, the head of the circuit is stable and we are left to compute:

$$\begin{array}{rcl} P_6^2 & = & (P_3^2(x) \sqcap [\underline{\text{lb}}(P_3^2(n)) + 1, +\infty], P_3^2(n) \sqcap [-\infty, \underline{\text{ub}}(P_3^2(x)) - 1]) \\ & = & ([0, +\infty] \sqcap [1, +\infty], [0, 1000] \sqcap [-\infty, +\infty]) = ([1, +\infty], [0, 1000]) \end{array}$$

Thus, we obtain the following over-approximated solution:

$$\left[\begin{array}{rcl} P_1 & = & (\bot, [0, 1000]) \\ P_2 & = & ([0, 0], [0, 1000]) \\ P_3 & = & ([0, +\infty], [0, 1000]) \\ P_4 & = & ([0, 1000], [0, 1000]) \\ P_5 & = & ([2, 1002], [0, 1000]) \\ P_6 & = & ([1, +\infty], [0, 1000]) \end{array}\right.$$

A chaotic iteration sequence without widening would stabilize after 500 computation steps, whereas the widening makes the computation converge in 2 steps. Obviously, the result is less precise. However, it follows from Remark 4.1.1.0.9 that our approximate solution can be improved.

5.7.3.2 Improving the approximate solution using a decreasing chaotic iteration sequence with lower narrowing

Given our post-fixpoint P of $X = F(X)$, $lfp(F) \sqsubseteq llis(\boldsymbol{\lambda} X \cdot X \sqcap F(X))(P)$ holds. As the computation of $llis(\boldsymbol{\lambda} X \cdot X \sqcap F(X))(P)$ may not converge in a finite number of steps, we suggest to compute an over-approximation of it by exploiting Definition 4.1.2.0.16 and Theorem 4.1.2.0.17.

Let us define a lower narrowing $\underline{\Delta}$ on L as:

- $\forall x \in L, \bot \,\underline{\Delta}\, x = x \,\underline{\Delta}\, \bot = \bot$
- $[a, b] \,\underline{\Delta}\, [c, d] \quad = \quad [\underline{\text{if}}\ a = -\infty\ \ \underline{\text{then}}\ \ c\ \ \underline{\text{else}}\ \ \underline{\min}(a, c)\ \underline{\text{endif}},$

  $\underline{\text{if}}\ b = +\infty\ \ \underline{\text{then}}\ \ d\ \ \underline{\text{else}}\ \ \underline{\max}(b, d)\ \underline{\text{endif}}]$

The hypotheses from Definition 4.1.2.0.15 are satisfied (see the dual of Remark 4.1.2.0.14). As before, $\underline{\underline{\Delta}}$ is defined on L^n by component-wise application of $\underline{\Delta}$. Coming back to Example 5.7.2.0.1, we compute:

$$
\begin{array}{rcl}
P_1^0 & = & (\bot, [0, 1000]) \\
P_2^0 & = & ([0, 0], [0, 1000]) \\
P_3^0 & = & ([0, +\infty], [0, 1000]) \\
P_4^0 & = & ([0, 1000], [0, 1000]) \\
P_5^0 & = & ([2, 1002], [0, 1000]) \\
P_6^0 & = & ([1, +\infty], [0, 1000]) \\
P_3^1 & = & (P_3^0 \,\underline{\underline{\Delta}}\, (P_2^0 \sqcup P_5^0) \\
 & = & ([0, +\infty]) \,\underline{\Delta}\, ([0, 0] \sqcup [2, 1002]), [0, 1000] \,\underline{\Delta}\, ([0, 1000] \sqcup [0, 1000])) \\
 & = & ([0, +\infty]) \,\underline{\Delta}\, [0, 1002], [0, 1000] \,\underline{\Delta}\, [0, 1000]) = ([0, 1002], [0, 1000]) \\
P_4^1 & = & ([0, 1002] \sqcap [-\infty, 1000], [0, 1000] \sqcap [0, +\infty]) = ([0, 1000], [0, 1000]) \\
P_5^1 & = & ([0, 1000] + [2, 2], [0, 1000]) = ([2, 1002], [0, 1000]) \\
P_6^1 & = & ([0, 1002] \sqcap [1, +\infty], [0, 1000] \sqcap [-\infty, 1001]) = ([1, 1002], [0, 1000])
\end{array}
$$

Now that the computation has reached a fixpoint, the final result is:

$$
\left[
\begin{array}{rcl}
P_1 & = & (\bot, [0, 1000]) \\
P_2 & = & ([0, 0], [0, 1000]) \\
P_3 & = & ([0, 1002], [0, 1000]) \\
P_4 & = & ([0, 1000], [0, 1000]) \\
P_5 & = & ([2, 1002], [0, 1000]) \\
P_6 & = & ([1, 1002], [0, 1000])
\end{array}
\right.
$$

In an actual implementation, arithmetic overflows may occur in the interval computations. It is thus necessary to catch this interrupt and return the infinite result $-\infty$ or $+\infty$. Moreover, it is possible to take declarations into account by inserting tests on the bounds at the intermediate language level, then applying our method, and finally eliminating dead branches before code generation.

5.7.4 Example in eliminating run-time bound checks

Our first example is a binary search of a key k in a table R of 100 elements sorted in increasing order of keys. The analysis result is given directly in the program source as comments:

```
type table = array[1, 100] of integers ;
procedure binary–search(var R : table ; k : value integer ; m : result integer) ;
    var lb, ub : integer ;
begin
    lb := lb(table) ;    ub := ub(table) ;
    ${((lb = [1, 1]), (ub = [100, 100]), (m = \bot))}$
    while  lb ≤ ub  do
        ${((lb = [1, 100]), (ub = [1, 100]), (m = [1, 100]))}$
        m := (lb + ub) div 2 ;
        ${((lb = [1, 100]), (ub = [1, 100]), (m = [1, 100]))}$
        if k = R(m) then
            lb := ub + 1 ;
            ${((lb = [2, 101]), (ub = [1, 100]), (m = [1, 100]))}$
        elsif k < R(m) then
            ub := m − 1 ;
            ${((lb = [1, 100]), (ub = [0, 99]), (m = [1, 100]))}$
        else
```

```
            lb := m + 1 ;
            $\{((lb = [2, 101]), (ub = [1, 100]), (m = [1, 100]))\}$
        endif ;
        $\{((lb = [1, 101]), (ub = [0, 100]), (m = [1, 100]))\}$
    redo
    $\{((lb = [1, 101]), (ub = [0, 100]), (m = [1, 100]))\}$
    if R(m) $\neq$ k then m := lb(table) $-$ 1 endif ;
    $\{((lb = [1, 101]), (ub = [0, 100]), (m = [0, 100]))\}$
end.
```

The compiler can prove that all the accesses to the array R are correct (as $1 \leq m \leq 100$ always holds) and that there is no overflow in the additions and subtractions (thus, it is useless to insert code to warn the programmer of run-time errors). Moreover, by taking the union of intervals obtained on all program points for each variable, the compiler can deduce that a valid declaration could have been m : result 0..100, var lb : 1..101, var ub : 0..100; in particular, the knowledge that the result m is an integer between 0 and 100 can be propagated to all the calls of the procedure. Finally, in $m := (lb + ub)$ div 2, the analysis has proved that $(lb + ub) \in [2, 200]$, and so, $(lb + ub) \geq 0$. As a consequence, the division can be implemented as a right shift — which would not be possible if we had $(lb{+}ub) < 0$. According to Welsh [1977], inserting run-time tests and not optimizing the division causes an increase of approximately 50% in the size of the generated code, and 60% in average execution time.

It is enlightening to compare our method to *discover* intervals of values of integer variables to the *verification* method of Welsh [1977]. Welsh's method is a classic compilation technique consisting in assuming that, whenever a variable is accessed, its value is in the interval specified by the user in the declaration and checking, whenever a variable is assigned, that the assigned value — using the same interval arithmetic that we used to compute the right-hand side of assignments — is in the declared interval — and generate a test if this is not the case. This one-pass method is cheap but not very effective. One can observe that programmers seldom — or incorrectly — exercise the option to declare integer variables in a numeric interval in PASCAL as introduced by Wirth. (It is sufficient to look at the examples given by Wirth himself in Wirth [1976] where this option is *never* exercised.) The reason for it is that, as bounds are compelled to be numeric constants — and not symbolic constants that are evaluated at compile-time as in LIS (Ichbiah, Rissen, Héliard & Cousot [1974]) — it is often required to change all the bounds when changing a declaration. In our procedure *binary-search*, we could have declared m : 0..100, lb : 1..101, ub : 0..100 but, if the number of elements in the *table* is changed, then all bounds in m, lb, ub as well as the initialization of lb and ub must be changed by hand. On the contrary, in LIS, we could declare $m : \underline{\text{lb}}(table) - 1..\,\underline{\text{ub}}(table)$, $lb : \underline{\text{lb}}(table)..\,\underline{\text{ub}}(table) + 1$, $ub : \underline{\text{lb}}(table) - 1..\,\underline{\text{ub}}(table)$ so that, when changing a bound of *table*, the compiler can take this change into account automatically. Coming back to Walsh's method, if the programmer declares $m, lb, ub : \underline{\text{integer}}$, then 3 tests are required on array bounds as well as 4 overflow tests. If the programmer declares m : 0..100, lb : 1..101, ub : 0..100 (which is the declaration we found automatically), then all overflow tests are eliminated and the division can be optimised, but 3 tests remain on the lower bounds of the array (as the access $R(m)$ requires that $1 \leq m \leq 100$, while we found $0 \leq m \leq 100$). This results in an increase of 19% in code size and of 17% in average execution time with respect to our optimal solution — figures courtesy of Welsh. A verification method based solely on global declarations is necessarily incomplete as the type of a variable at a given program point is almost always a sub-type of the globally declared type (Meertens [1975]). In our example, a programmer knowledgeable in compilation methods could make explicit the fact that the type of m is not the same within the loop and when exiting the loop by introducing a new variable p and writing:

```
procedure binary–search(var R : table ; k : value integer ; p : result 0..100) ;
      var lb : 1..101 ; ub : 0..100 ; m : 1..100 ;
begin
      lb := 1 ;    ub := 100 ;
      while   $lb \leq ub$   do
            m := (lb + ub) div 2 ;
            if R(m) = k then
                  lb := ub + 1 ;
             elsif R(m) > k then
                  ub := m − 1 ;
             else
                  lb := m + 1 ;
             endif ;
      redo ;
      p := if R(m) = k  then  m  else  0 endif ;
 end.
```

However, this solution is still not satisfactory as, in the instruction $m := (lb + ub)$ div 2, the type of the right-hand side is $([1, 101] + [0, 100])$ div $[2, 2] = [0, 100]$ and a test must be inserted to check that $(lb + ub)$ div $2 \geq 1$! The problem is now to compare the cost of this run-time test with the cost of an analysis that can remove it. A more decisive argument is that our method allows discovering many more programming errors *before* program execution. Finally, our approach also works in the case of symbolic bounds (see 5.8).

5.7.5 Combining forward and backward approximate analyses

We demonstrate the technique presented in Paragraph 5.6.2.3 when the convergence does not hold naturally. Given the over-approximations $\bar{F}(\bar{\phi})$ and $\bar{B}(\bar{\psi})$ of the systems of forward and backward semantic equations $F_\pi(\phi)$ and $B_\pi(\psi)$, we compute:

$$P^1 \quad \sqsupseteq \quad lfp(\bar{F}(\bar{\phi}))$$

$$\begin{array}{llllll}
P^2 & = & P^1 \,\underline{\Delta}\, X^2 & \text{where} & X^2 & \sqsupseteq \; \mathit{lfp}(\boldsymbol{\lambda} X \cdot P^1 \sqcap \bar{B}(\bar{\psi})(X)) \\
\ldots & & & & & \\
P^{2k+1} & = & P^{2k} \,\underline{\Delta}\, X^{2k+1} & \text{where} & X^{2k+1} & \sqsupseteq \; \mathit{lfp}(\boldsymbol{\lambda} X \cdot P^{2k} \sqcap \bar{F}(\bar{\phi})(X)) \\
P^{2k+2} & = & P^{2k+1} \,\underline{\Delta}\, X^{2k+2} & \text{where} & X^{2k+2} & \sqsupseteq \; \mathit{lfp}(\boldsymbol{\lambda} X \cdot P^{2k+1} \sqcap \bar{B}(\bar{\psi})(X)) \\
\ldots & & & & &
\end{array}$$

It follows from Definition 4.1.2.0.15 of the lower narrowing $\underline{\Delta}$ that the sequence is finite and every term is greater than $\mathit{lfp}(F_\pi(\phi))$ $\underline{\text{and}}$ $\mathit{lfp}(B_\pi(\psi))$. To compute X^k, we perform an increasing chaotic iteration sequence with upper widening and, if the solution we obtain is not a fixpoint, we improve on it using a decreasing chaotic iteration sequence with lower narrowing.

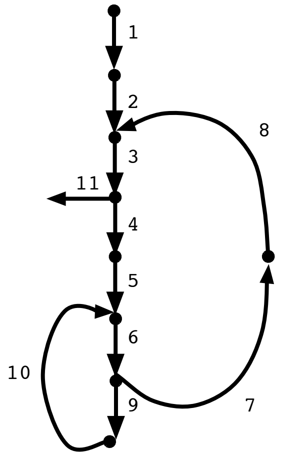
We demonstrate this technique on an array sorting program from Manna [1974, p. 191] using the "bubble sort" method. After removing all the instructions pertaining to the array to sort (which is ignored in the approximate analysis), the program becomes:

```
{1}
          i := n ;
{2}
     L:
{3}
          if i ≠ 0 then
{4}
               j := 0 ;
{5}
     M:
{6}
               if j = i then
{7}
                    i := i − 1 ;
{8}
                    goto L ;
                endif ;
{9}
```

```
                  j := j + 1 ;
{10}
                  goto M ;
              endif ;
{11}
```

The approximate system of forward equations associated with the program is as follows:

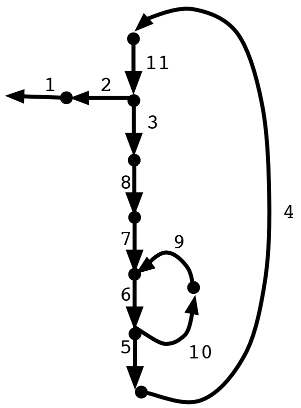
$$
\left\{
\begin{array}{rcl}
P_1 & = & \bar{\phi} \\
P_2 & = & P_1(i \leftarrow n) \\
P_3 & = & P_2 \sqcup P_8 \\
P_4 & = & P_3(i \leftarrow i \neq 0) \\
P_5 & = & P_4(j \leftarrow [0,0]) \\
P_6 & = & P_5 \sqcup P_{10} \\
P_7 & = & P_6(i \leftarrow i \sqcap j, j \leftarrow i \sqcap j) \\
P_8 & = & P_7(i \leftarrow i - 1) \\
P_9 & = & P_6(i \leftarrow i \neq j, j \leftarrow i \neq j) \\
P_{10} & = & P_9(j \leftarrow j + 1) \\
P_{11} & = & P_3(i \leftarrow i \sqcap [0,0])
\end{array}
\right.
$$



Graph of dependences
(loop test: 6)

The approximate system of backward equations associated with the program is as follows:

$$\left\{\begin{array}{lcl}
P_1 & = & P_2(i \leftarrow [-\infty,+\infty], n \leftarrow i) \\
P_2 & = & P_3 \\
P_3 & = & P_{11} \sqcup P_4 \\
P_4 & = & P_5(j \leftarrow [-\infty,+\infty]) \\
P_5 & = & P_6 \\
P_6 & = & P_7 \sqcup P_9 \\
P_7 & = & P_8(i \leftarrow i+1) \\
P_8 & = & P_3 \\
P_9 & = & P_{10}(j \leftarrow j-1) \\
P_{10} & = & P_6 \\
P_{11} & = & \bar{\psi}
\end{array}\right.$$



Graph of dependences
(loop test: 6)

We analyze the program with respect to the following specifications:

$$\bar{\phi} \quad = \quad ((i = \bot), (j = \bot), (n = [-\infty,+\infty]))$$

$$\bar{\psi} \quad = \quad ((i = [-\infty,+\infty]), (j = [-\infty,+\infty]), (n = [-\infty,+\infty]))$$

Without giving the details of the computation, we get:

$$P^1 \quad \sqsupseteq \quad lfp(\bar{F}(\bar{\phi}))$$

	$\{1\}$	$\{2\}$	$\{3\}$	$\{4\}$	$\{5\}$	$\{6\}$	$\{7\}$	
i	$\bot$	$[-\infty,+\infty]$	$[-\infty,+\infty]$	$[-\infty,+\infty]$	$[-\infty,+\infty]$	$[-\infty,+\infty]$	$[0,+\infty]$	$[-1$
j	$\bot$	$\bot$	$[0,+\infty]$	$[0,+\infty]$	$[0,0]$	$[0,+\infty]$	$[0,+\infty]$	$[0,$
n	$[-\infty,+\infty]$	$[-\infty,+\infty]$	$[-\infty,+\infty]$	$[-\infty,+\infty]$	$[-\infty,+\infty]$	$[-\infty,+\infty]$	$[-\infty,+\infty]$	$[-\infty$

This result is very disappointing, except for j which is found to be positive as it is initialized to zero and incremented by one. The backward analysis provides some improvement:

$$P^2 \quad = \quad P^1 \mathrel{\underline{\underline{\Delta}}} X^2 \qquad \text{where } X^2 \sqsupseteq \mathit{lfp}(\boldsymbol{\lambda}\, X \cdot P^1 \sqcap \bar{B}(\bar{\psi})(X))$$

	$\{1\}$	$\{2\}$	$\{3\}$	$\{4\}$	$\{5\}$	$\{6\}$	$\{7\}$	$\{8$
i	$\perp$	$[0, +\infty]$	$[0, +\infty]$	$[1, +\infty]$	$[1, +\infty]$	$[1, +\infty]$	$[1, +\infty]$	$[0, +$
j	$\perp$	$\perp$	$[0, +\infty]$	$[0, +\infty]$	$[0, +\infty]$	$[0, +\infty]$	$[0, +\infty]$	$[0, +$
n	$[0, +\infty]$	$[-\infty, +\infty]$	$[-\infty, +\infty]$	$[-\infty, +\infty]$	$[-\infty, +\infty]$	$[-\infty, +\infty]$	$[-\infty, +\infty]$	$[-\infty,$

The backward analysis was precise enough to discover that, as i is decremented by one in a loop ending with $i = 0$, i is necessarily positive before entering the loop for the loop to terminate. The information we just found can be propagated forward:

$$P^3 \quad = \quad P^2 \mathrel{\underline{\underline{\Delta}}} X^3 \qquad \text{where } X^3 \sqsupseteq \mathit{lfp}(\boldsymbol{\lambda}\, X \cdot P^2 \sqcap \bar{F}(\bar{\phi})(X))$$

	$\{1\}$	$\{2\}$	$\{3\}$	$\{4\}$	$\{5\}$	$\{6\}$	$\{7\}$	$\{8\}$	$\{9\}$	$\{10$
i	$\perp$	$[0, +\infty]$	$[0, +\infty]$	$[1, +\infty]$	$[1, +\infty]$	$[1, +\infty]$	$[1, +\infty]$	$[0, +\infty]$	$[1, +\infty]$	$[1, +$
j	$\perp$	$\perp$	$[1, +\infty]$	$[1, +\infty]$	$[0, 0]$	$[0, +\infty]$	$[1, +\infty]$	$[1, +\infty]$	$[0, +\infty]$	$[1, +$
n	$[0, +\infty]$	$[0, +\infty]$	$[0, +\infty]$	$[0, +\infty]$	$[0, +\infty]$	$[0, +\infty]$	$[0, +\infty]$	$[0, +\infty]$	$[0, +\infty]$	$[0, +$

A subsequent step would show that these results are stable. Indeed, we have found the best result we could obtain with the considered approximate system of equations. In particular, we proved that, if $n \in [-\infty, -1]$, then the program either does not terminate or results in an error. The entry specification $n \geq 0$ is indeed provided by Manna [1974, p. 191], but it is valuable to note that, if it is not provided by the programmer, we can prove easily and automatically that there is an error.

Now that we discovered the entry condition:

$$((i = \perp), (j = \perp), (n = [0, +\infty]))$$

as well as the declarations that must appear in the program:

$$D \quad = \quad ((i = [0, +\infty]), (j = [0, +\infty]), (n = [0, +\infty]))$$

it is left to us to discover where run-time tests are to be inserted. Given our knowledge of the properties P_k^3 of the variables at the entry a_k of an instruction I, it is sufficient to check, using the system of forward equations, that the image P of P_k^3 by I at the exit point a_l of I is included in D. When this is not the case, a run-time test is required, that is:

- At the entry point, we must check that $n \geq 0$.
- When decrementing i, as $[1, +\infty] - [1, 1] \sqsubseteq [0, +\infty]$, no test is required.
- When incrementing j, as $[0, +\infty] + [1, 1] = [0, +\infty + 1]$, the compiler must foresee a possible detection by the hardware of some overflow during the incrementation. Actually, this is useless as we can see that $j \leq i$. Due to the approximation chosen in 5.6.1, it is not possible to express the relationships between variables, and so, to discover this inequality. We now present a less coarse approximation method that can discover relationships between the numeric variables of a program.

5.8 AUTOMATICALLY DISCOVERING LINEAR EQUALITY OR INEQUALITY RELATIONS BETWEEN THE NUMERIC VARIABLES OF A PROGRAM

It is not often that a reasoning on a program does not require discovering invariant relations between the variables $x_1, \ldots, x_n$ of the program — see for instance 5.6.1.2. In the following application, we plan to discover linear equality ($ax_1 + bx_2 + \ldots + wx_n = \alpha$ where a, b, ..., w, and α are constants) or inequality ($ax_1 + bx_2 + \ldots + wx_n \leq \alpha$) relations between the numeric variables $x_1, \ldots, x_n$ of a program. As an introductory example, we present the analysis of the sorting algorithm "bubble sort" from Knuth [1973, p. 107]:

```
        procedure sort(N : value integer ; K : array[1, N] of reals) ;
            var B, J, T : integer ;
        begin
            B := N ;
  {1}
            while   B ≥ 1   do
  {2}
                J := 1 ;    T := 0 ;
  {3}
                while   J ≤ (B − 1)   do
  {4}
                    if K[J] ≥ K[J + 1] then
  {5}
                        exchange(J, J + 1) ;    {no side-effect on N, B, J, T}
  {6}
                        T := J ;
  {7}
                    endif ;
  {8}
                    J := J + 1 ;
  {9}
                redo ;
  {10}
                if T = 0  then  return endif ;
  {11}
                B := T ;
```

```
{12}
          redo ;
{13}
      end ;
```

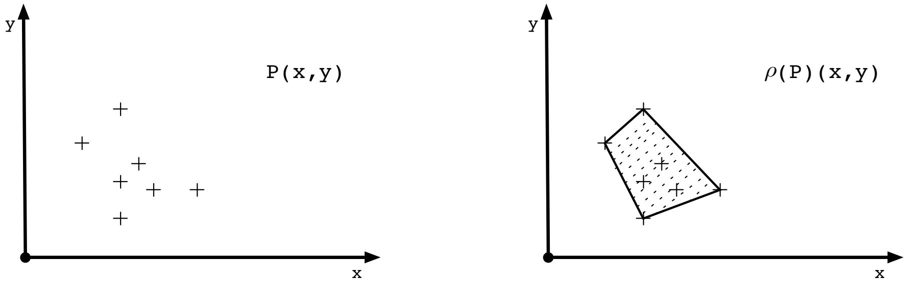
The automatic analysis of this procedure — performed using the experimental implementation by N. Halbwachs — provides the following invariants:

$$\left[\begin{array}{lcl}
P_1 & = & \{B = N\} \\
P_2 & = & \{1 \leq B \leq N\} \\
P_3 & = & \{1 \leq B \leq N, J = 1, T = 0\} \\
P_4, P_5, P_6 & = & \{B \leq N, T \geq 0, T+1 \leq J, J+1 \leq B\} \\
P_7 & = & \{B \leq N, J \geq 1, J+1 \leq B, J = T\} \\
P_8 & = & \{B \leq N, J+1 \leq B, J \geq 1, T \geq 0, T \leq J\} \\
P_9 & = & \{B \leq N, J \leq B, J \geq 2, T \geq 0, T+1 \leq J\} \\
P_{10}, P_{11} & = & \{B \leq N, J \leq B, T \geq 0, T+1 \leq J, B \leq J+1\} \\
P_{12} & = & \{J \leq N, T \geq 0, T+1 \leq J, T = B\} \\
P_{13} & = & \{B \leq N, B \leq 1\}
\end{array}\right.$$

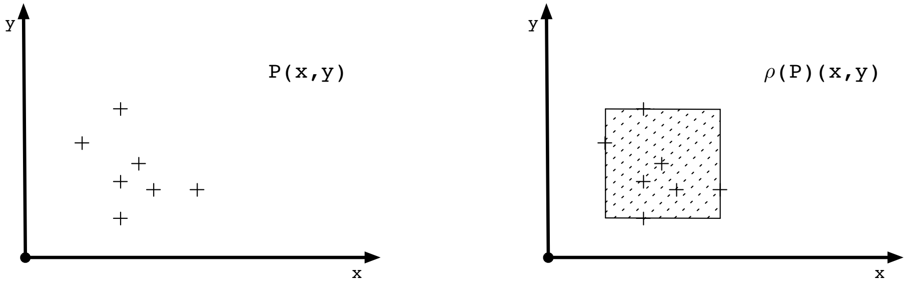
This information is useful to check that all array indexes lie within array bounds, especially when these bounds are provided as symbolic constants — while the application from 5.7 is better suited to the case of numeric bounds. Moreover, these invariants are required to validate the program and complement the specifications provided by the programmer, that often do not feature this level of detail.

5.8.1 Space of approximate properties

Let $P \in \mathcal{P}_n = \mathcal{U}^n \to \{\underline{\text{true}}, \underline{\text{false}}\}$ be a property over n variables with values in the set $\mathcal{U}$ of rationals. The space of approximate properties is defined by the upper closure operator ρ, where $\rho(P)$ is the characteristic property of the set of points in the convex hull of the set characterised by P. The geometrical interpretation is thus:



To compare this application to the previous one concerning the discovery of an interval of values for each numeric variable, we must observe that the closure operator used in Paragraph 5.7 was simply:



For the application at hand, we must study the set $\rho(\mathcal{P}_n)$. Let $P \in \mathcal{P}_n$. As $\mathcal{U}$ is the set of integers, rationals, or reals that can be represented in a computer, it is finite. Thus, the number of values in $\mathcal{U}^n$ satisfying P is finite, and so, $\rho(P)$ is a convex polyhedron, as it is the convex hull of a finite number of points in $\mathcal{U}^n$. This implies that we can represent $\rho(P)$ by a finite number of linear equality or inequality relations between program variables. On a more abstract level, we can forget about the traditional limitations of programming languages and consider that program variables have values in the set $\mathcal{R}$ of reals. In this case, $\rho(\mathcal{P}_n)$ denotes the convex subsets of $\mathcal{R}^n$, some of which cannot be described by a finite system of linear equality or inequality relations. Then, the space of approximate properties corresponding to convex polyhedra does not form a complete lattice. Nevertheless, when using the dynamic approximation algorithms introduced in Paragraph 4.1, solving iteratively the systems of equations associated with programs always takes a finite number of steps so that the limit of the sequence of iterates is a convex polyhedron (see Remark 4.1.2.0.7.(e)).

5.8.2 Rules to construct the approximate system of forward equations associated with a program

Assignment:

Let X be the column vector of the variables of the program and $AX \leq B$ be the constraint system before an assignment instruction (for the sake of simplicity, equality relations are represented as pairs of opposite inequality relations):

- If the assignment is not linear (for instance $\mathrm{x := x^2 + yz - t}$), then the constraint system after the assignment is constructed by eliminating x from the system $AX \leq B$.

- Otherwise, the assignment is linear and has the form $\mathrm{x_k := a_1x_1 + \ldots + a_nx_n + b}$ where $a_1, \ldots, a_n, b$ are rational coefficients.

  - If a_k is not zero, then the assignment is invertible, that is, it is possible to compute the former value of x_k satisfying a given constraint system given its new value. The constraint system after assignment is thus constructed by replacing x_k with $(x_k - a_1x_1 - \ldots - a_{k-1}x_{k-1} - a_{k+1}x_{k+1} - \ldots - a_nx_n - b)/a_k$ in $AX \leq B$.

- If a_k is zero, then the former value of x_k is lost, so, we eliminate x_k from the constraint system $AX \leq B$. We then add the constraint $x_k = a_1x_1 + \ldots + a_{k-1}x_{k-1} + a_{k+1}x_{k+1} + \ldots + a_nx_n + b$.

Example

Constraint system before assignment:

$$\left[\begin{array}{ccccc} & & x_2 & \geq & 1 \\ x_1 & + & x_2 & \geq & 5 \\ x_1 & - & x_2 & \geq & -1 \end{array}\right.$$

Non-linear assignment:	Non-invertible linear assignment:	Invertible linear assignment:
$\mathrm{x}_2 := (\mathrm{x}_1)^2 - (\mathrm{x}_2)^2$	$\mathrm{x}_2 := \mathrm{x}_1 + 1$	$\mathrm{x}_2 := \mathrm{x}_1 + \mathrm{x}_2/2 + 1$

Output constraint system:

$$\left[\begin{array}{ccc} x_1 & \geq & 2 \end{array}\right.$$

$$\left[\begin{array}{cccccc} & x_1 & & & \geq & 2 \\ - & x_1 & + & x_2 & = & 1 \end{array}\right.$$

$$\left[\begin{array}{cccccc} - & 2x_1 & + & 2x_2 & \geq & 3 \\ - & x_1 & + & 2x_2 & \geq & 7 \\ + & 3x_1 & - & 2x_2 & \geq & -3 \end{array}\right.$$

End of example.

Test:

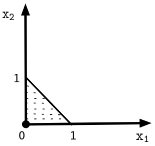
If the test is linear, then the output constraints are constructed by adding the test constraint to the input constraint system. If the test is not linear, then it is simply ignored and the input and output constraints are equal.

Path junction:

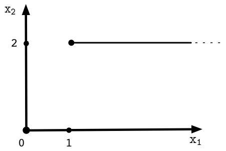
At a path junction $a_1, \ldots, a_k$ where the respective constraint systems are $A_1X \leq B_1, \ldots, A_kX \leq B_k$, we compute the constraint system corresponding to the convex hull of the convex polyhedra defined by $A_1X \leq B_1, \ldots, A_kX \leq B_k$. To partly avoid this costly computation, we keep in the implementation a double representation of the abstract properties: a constraint system and a system of generators corresponding to the polyhedron (Lanery [1966]).

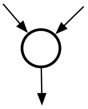
Example

$$\left[\begin{array}{ccccc} x_1 & & & \geq & 0 \\ & & x_2 & \geq & 0 \\ x_1 & + & x_2 & \leq & 1 \end{array}\right.$$

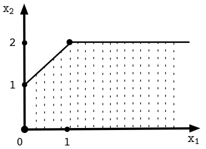


$$\left[\begin{array}{ccc} x_1 & \geq & 1 \\ x_2 & = & 2 \end{array}\right.$$





$$\left[\begin{array}{cccccc} & x_1 & & & \geq & 0 \\ & & & x_2 & \geq & 0 \\ & & & x_2 & \leq & 2 \\ - & x_1 & + & x_2 & \leq & 1 \end{array}\right.$$



End of example.

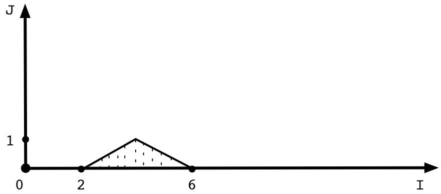
5.8.3 Approximate solving of the system of equations by increasing chaotic iteration sequences with upper widening

The iterative solving of the system of equations associated with a program does not converge in a finite number of steps in general, so, we employ an increasing chaotic iteration sequence with upper widening (Definition 4.1.2.0.5).

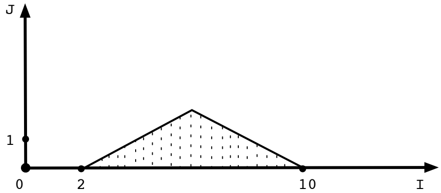
The upper widening $P_1 \bar{\nabla} P_2$ is defined as the set of constraints from P_1 that are satisfied by the elements in the system of generators of P_2. As the number of constraints can only decrease, the hypotheses in Definition 4.1.2.0.4 hold.

Example

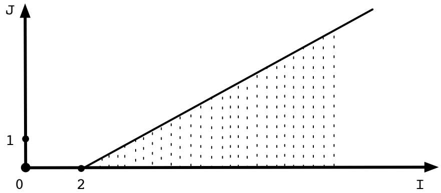
$$Q_1 \left[\begin{array}{rcr} -I + 2J & \leq & -2 \\ I + 2J & \leq & 6 \\ J & \geq & 0 \end{array}\right.$$



$$Q_2 \left[\begin{array}{rcr} -I + 2J & \leq & -2 \\ I + 2J & \leq & 10 \\ J & \geq & 0 \end{array}\right.$$



$$Q_1 \bar{\nabla} Q_2 \left[\begin{array}{rcr} -I + 2J & \leq & -2 \\ J & \geq & 0 \end{array}\right.$$



End of example.

We demonstrate how to solve an approximate system of equations associated with a program on the following example:

```
{1}
          I := 2 ;   J := 0 ;
{2}
     L:
{3}
          if ... then
```

```
{4}
            I := I + 4 ;
{5}
        else
{6}
            I := I + 2 ;    J := J + 1 ;
{7}
        endif ;
{8}
       goto L ;
```

(The test of a non-linear condition is ignored.)

The rules stated in Paragraph 5.8.2 give the following approximate system of forward equations associated with the program:

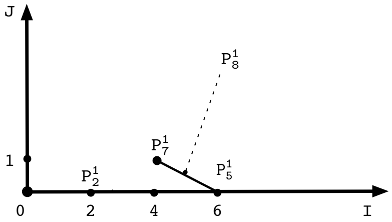
$$\left\{\begin{array}{lcl}
P_1 & = & \top \\
P_2 & = & \underline{\text{assign}}(I := 2) \circ \underline{\text{assign}}(J := 0)(P_1) \\
P_3 & = & \underline{\text{convex–hull}}(P_2, P_8) \\
P_4 & = & P_3 \\
P_5 & = & \underline{\text{assign}}(I := I + 4)(P_4) \\
P_6 & = & P_3 \\
P_7 & = & \underline{\text{assign}}(J := J + 1) \circ \underline{\text{assign}}(I := I + 2)(P_6) \\
P_8 & = & \underline{\text{convex–hull}}(P_5, P_7)
\end{array}\right.$$

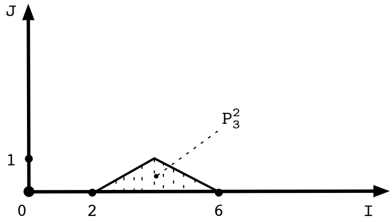
As this example is very simple, we can now demonstrate how to compute an approximate solution of this system of equations. As permitted by Remark 4.1.2.0.7.(c), the widening is only used when a sufficient amount of information is gathered at each program point.

$$P_i^0 \quad = \quad \bot \qquad \text{for } i = 1, \ldots, 8 \qquad \text{(empty polyhedron)}$$

$$P_1^1 \quad = \quad \top$$

$$\begin{aligned}
P_2^1 &= \underline{\text{assign}}(I := 2) \circ \underline{\text{assign}}(J := 0)(P_1^1) \\
&= \{I = 2, J = 0\} \\
P_3^1 &= \underline{\text{convex–hull}}(P_2^1, P_8^0) = \underline{\text{convex–hull}}(P_2^1, \perp) = P_2^1 \\
P_4^1 &= P_6^1 = P_3^1 \\
P_5^1 &= \underline{\text{assign}}(I := I + 4)(P_3^1) \\
&= \{I = 6, J = 0\} \\
P_7^1 &= \underline{\text{assign}}(J := J + 1) \circ \underline{\text{assign}}(I := I + 2)(P_6^1) \\
&= \{I = 4, J = 1\} \\
P_8^1 &= \underline{\text{convex–hull}}(P_5^1, P_7^1) \\
&= \{I + 2J = 6, 4 \leq I \leq 6\}
\end{aligned}$$





$$\begin{aligned}
P_3^2 &= \underline{\text{convex–hull}}(P_2^1, P_8^1) \\
&= \{2J + 2 \leq I, I + 2J \leq 6, 0 \leq J\} \\
P_4^2 &= P_6^2 = P_3^2 \\
P_5^2 &= \underline{\text{assign}}(I := I + 4)(P_4^2)
\end{aligned}$$

$$= \{2J+6 \le I, I+2J \le 10, 0 \le J\}$$

(I is replaced with $I-4$ in P_4^2 as the assignment is invertible)

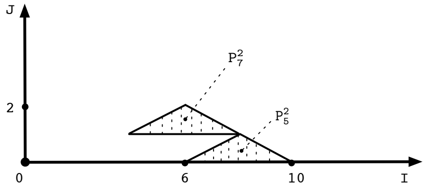
$$P_7^2 = \underline{\text{assign}}(J := J+1) \circ \underline{\text{assign}}(I := I+2)(P_6^2)$$

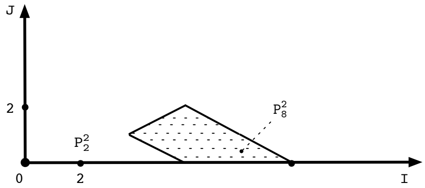
$$= \{2J+2 \le I, I+2J \le 10, 1 \le J\}$$

(I and J are replaced with, respectively, $I-2$ and $J-1$ in P_6^2)

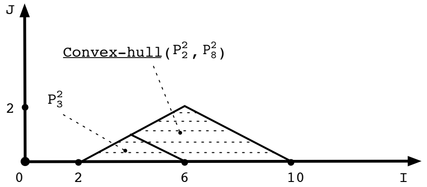
$$P_8^2 = \underline{\text{convex–hull}}(P_5^2, P_7^2)$$

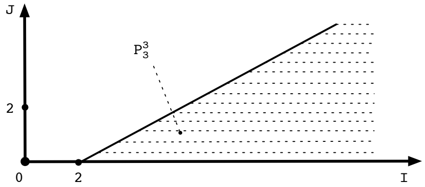
$$= \{2J+2 \le I, 6 \le I+2J \le 10, 0 \le J\}$$





$$P_3^3 = P_3^2 \,\bar{\nabla}\, \underline{\text{convex–hull}}(P_2^2, P_8^2)$$





$$P_3^3 = \{2J+2 \le I, 0 \le J\}$$

$$P_4^3 = P_6^3 = P_3^3$$

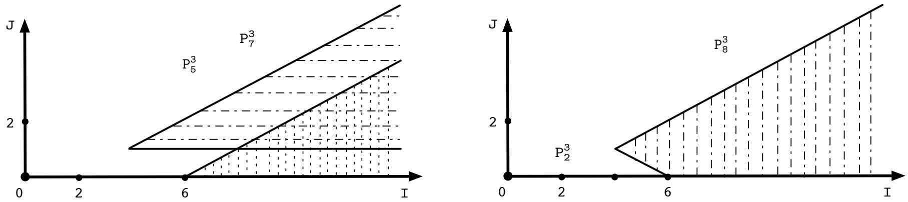
$$P_5^3 = \underline{\text{assign}}(I := I+4)(P_4^3)$$

$$= \{2J+6 \le I, 0 \le J\}$$

$$P_7^3 = \underline{\text{assign}}(J := J+1) \circ \underline{\text{assign}}(I := I+2)(P_6^3)$$

$$= \{2J+2 \le I, 1 \le J\}$$

$$P_8^3 = \underline{\text{convex–hull}}(P_5^3, P_7^3)$$



$$P_8^3 = \{2J+2 \le I, 6 \le I+2J, 0 \le J\}$$

Then, $\underline{\text{convex–hull}}(P_2^3, P_8^3) = P_6^3$, so, we reached a fixpoint of the system of equations and the final result is:

```
{1}
            I := 2 ;    J := 0 ;
{2}         $\{I = 2, J = 0\}$
      L:
{3}         $\{2J+2 \le I, 0 \le J\}$
            if ... then
{4}               $\{2J+2 \le I, 0 \le J\}$
                  I := I + 4 ;
{5}               $\{2J+6 \le I, 0 \le J\}$
             else
{6}               $\{2J+2 \le I, 0 \le J\}$
```

```
                    I := I + 2 ;    J := J + 1 ;
{7}                 $\{2J + 2 \leq I, 1 \leq J\}$
                endif ;
{8}             $\{2J + 2 \leq I, 6 \leq I + 2J, 0 \leq J\}$
                goto L ;
```

In particular, we found a loop invariant $\{2J + 2 \leq I, 0 \leq J\}$ expressing an invariant relation between the program variables I and J, although this relation does not appear explicitly in any program instruction.

More information on this application can be found in Cousot & Halbwachs [1978]. To complete this reference, we add that the results obtained after an increasing iteration sequence with upper widening can be improved using a decreasing iteration sequence with lower narrowing (as in Paragraph 5.7.3.2). Moreover, the technique in 5.7.5 can be used to obtain necessary constraints for the program to terminate without any error.

5.8.4 Example

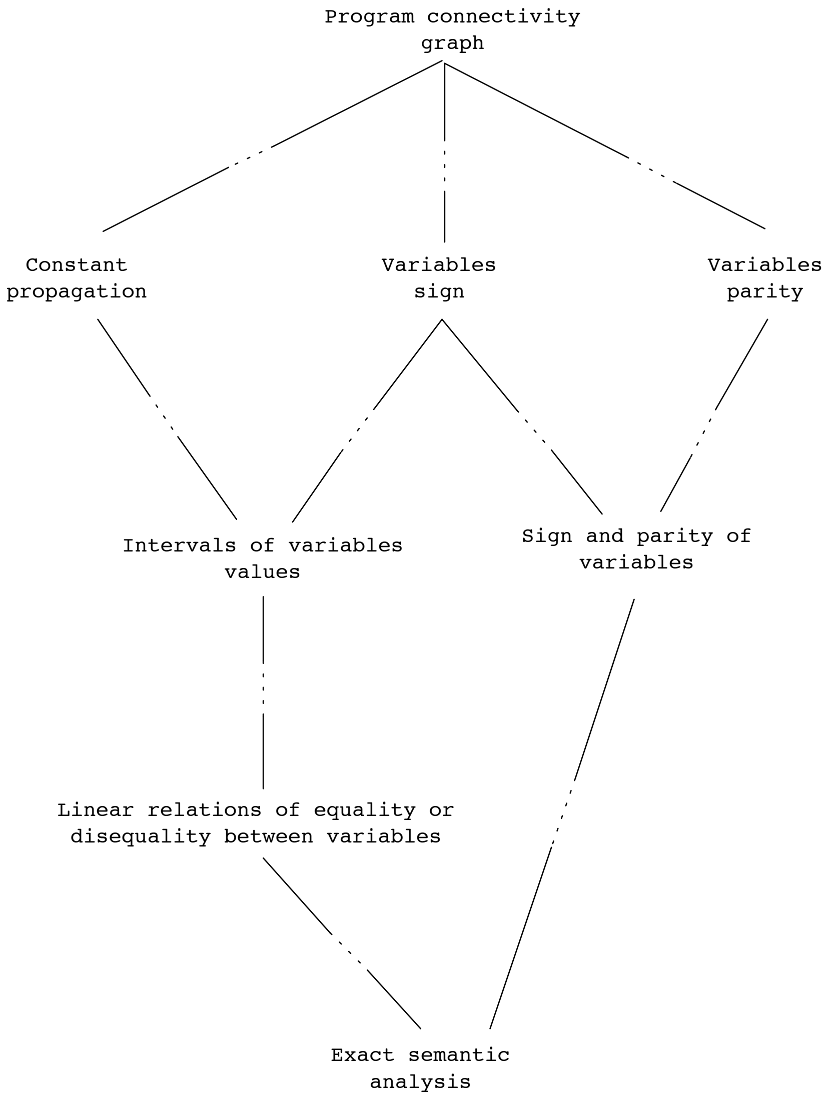
To conclude, let us come back to Example 5.7.4 on the binary search of a key k in a table R containing n elements sorted in increasing order of keys. Note that the number of elements in the array is now a symbolic constant with a fixed but unknown value and no longer a numeric constant. We provide in comments in the procedure source the results of an analysis of the linear equality or inequality relations between the variables of the procedure. This analysis has been performed automatically on a computer. The results can be favorably compared to those obtained by the verification methods in Suzuki & Ishihata [1977], the heuristic methods in German [1978] based on trials and errors, and the data-flow analysis methods in Gillett [1977].

```
type table = array[1, n] of integers ;
procedure binary–search(var R : table ; k : value integer ; m : result integer) ;
    var lb, ub : integer ;
begin
```

```
    lb := lb(table) ;    ub := ub(table) ;
    $\{lb = 1, ub = n\}$
    while  lb $\leq$ ub  do
        $\{1 \leq lb \leq ub \leq n\}$
        m := (lb + ub) div 2 ;
        $\{1 \leq lb \leq ub \leq n, 2m \leq lb + ub \leq 2m + 1$ (and so, $1 \leq m \leq n$, as $m$ is an integer)$\}$
        if k = R(m) then
            lb := ub + 1 ;
            $\{1 \leq ub \leq n, ub \leq 2m \leq 2ub, lb = ub + 1\}$
        elsif k < R(m) then
            ub := m − 1 ;
            $\{1 \leq lb \leq n, 2lb - 1 \leq 2m \leq lb + n, m = ub + 1\}$
        else
            lb := m + 1 ;
            $\{1 \leq ub \leq n, ub \leq 2m \leq 2ub, lb = m + 1\}$
        endif ;
        $\{m \leq ub + 1, 3lb \leq 2ub + n + 3, 3lb \leq 2ub + 2m + 3, 2lb \leq 2ub + 3,$
         $ub + m + 1 \leq lb + n, lb + m \leq ub + n + 1, 1 \leq n, ub \leq n, ub \leq 2m,$
         $ub + 4 \leq 3lb + m, 1 \leq 2m, ub + 1 \leq lb + m\}$
    redo ;
    $\{1 \leq lb, ub \leq n, ub < lb$   (note that when $n < 1, m$ is not initialized)$\}$
    if R(m) $\neq$ k  then  m := lb(table) − 1 endif ;
 end.
```

5.9 HIERARCHY OF APPLICATIONS

To conclude this chapter, we provide a partial order of the few approximate analyses that have been demonstrated here — using the intuitive notion of precision of the approximation corresponding to the order $\sqsubseteq$ in the lattice of closure operators:


Program connectivity graph
Constant propagation
Variables sign
Variables parity
Intervals of variables values
Sign and parity of variables
Linear relations of equality or disequality between variables
Exact semantic analysis

A task left to be done is to complete this lattice with other interesting applications. It might seem difficult to envision models that are approximate and useful, but each example given in this chapter demonstrates that the theoretical scheme we propose provides a very thorough guide highlighting the problems to be solved for each specific application and offering general methods to solve them.

Finally, the method of program analysis we provide is the same, whether it is to perform an exact semantic analysis (§3) or to perform an automatic but approximate semantic analysis (§5). Between these extremes, the same model could be used to perform by hand analyses that are approximate but too complex to be fully automated. Although we did not provide such examples, this is a field open to stimulating applications.

5.10 BIBLIOGRAPHIC NOTES

The boolean program optimisation techniques date back to Vyssotsky and Wegner who applied them in a FORTRAN compiler. They used an iterative solving method. Later, Allen [1970], Allen [1971], and Cocke [1970] introduced a solving method similar to Gauss elimination and based on the intervals of the program graph. As this method requires the equations to be boolean and the program graph to be "reducible", it is not general (Earnest [1974], Graham & Wegman [1976], Hecht & Ullman [1972], Hecht & Ullman [1974], Kasvanov [1973], Kennedy [1972], Schaefer [1973], Tarjan [1974]). Iterative solving of boolean equations has been used independently by Ichbiah, Morel & Renvoise [1972] and Hecht & Ullman [1973]. It has obviously no restriction. Attempts at comparing the direct and iterative methods (Hecht & Ullman [1975], Kennedy [1976], Tarjan [1975]) are not very conclusive because the hypotheses required to use direct methods generally also guarantee a fast convergence of the iterative methods. For instance, in the case of live variable analysis (5.5.1.1), we can prove that there exists an optimal order to traverse the program graph (Kennedy [1975], Tarjan [1976]) and, when the program is reducible, there exists an algorithm to construct this order (Aho & Ullman [1976]). This proves, in particular, that the theoretical search for optimal chaotic strategies may be successful in some interesting special cases. Spillman [1971], Boom

[1974], and Aho & Ullman [1977, Chap. 14.7] discuss how to use classic optimisation techniques in the presence of pointers.

Our method to analyze programs generalizes the idea, now quite old, of performing a symbolic execution using abstract values that characterise the properties to discover (Jensen [1965], Naur [1965], Sintzoff [1972], Kildall [1973], Karr [1975], Schwartz [1975], Wegbreit [1975]).

Example techniques to discover at compile-time the type of the objects manipulated by a high-level language can be found in Bauer & Saal [1974], Tenenbaum [1974], Jones & Müchnich [1976], and Kaplan & Ullman [1978].

Our application to discover an interval of values for the numeric variables of a program (§5.7) can be compared to the empirical method of Harrison [1977]. For the sake of completeness, let us also cite the verification methods used by Welsh [1977] and Suzuki & Ishihata [1977].

Our application to discover linear equality or inequality relations between the variables of a program (§5.8) improves on the results by Karr [1976] on the simpler problem of determining linear equality constraints. Other approaches exist that give partial answers to the problem of discovering invariant relations between the variables of a program: Cooper [1971], Caplain [1975], Elspas [1974], German & Wegbreit [1975], Katz & Manna [1976], Wegbreit [1974], Wegbreit [1977].

To conclude, let us note that our application to discover automatically an interval of values for the numeric variables of a program has been implemented by a student of J. Cohen at the University Brandeis following Cousot & Cousot [1976]. Curry [1977] applies the abstract interpretation idea from Cousot & Cousot [1977a] to a graphical programming language.

CHAPTER 6.

SEMANTIC ANALYSIS OF RECURSIVE PROCEDURES

6. SEMANTIC ANALYSIS OF RECURSIVE PROCEDURES

6. SEMANTIC ANALYSIS OF RECURSIVE PROCEDURES

In this chapter, we consider a more general programming language than before, with assignments, conditional instructions, unconditional branching, block structures, and possibly recursive procedures (with value-result parameter passing). Studying those concepts is necessary to take into account widely used programming language features. Also, it shows that the reasoning we developed in the preceding chapters to design a method of semantic program analysis is indeed general. We repeat the outline we followed for sequential iterative programs, except that, instead of considering systems of equations $X = F(X)$, we have to consider systems of the form $f(X) = F(f)(X)$. Indeed, in the preceding chapter, we observed that a sequential iterative program π could be analyzed by associating with each program point α a predicate $P_\alpha(\varphi)$ depending implicitly on an entry specification φ. These predicates were obtained as the solution of a system of equations $P(\varphi) = G(P)(\varphi)$. Observing that the system of equations can be written as $P(\varphi) = F_\pi(\varphi)(P(\varphi))$ and analyzing the program for a given specification φ, it was not necessary to consider a system of functional equations, since we could write $X_\alpha = P_\alpha(\varphi)$ and solve $X = F(X)$ with $F = F_\pi(\varphi)$. In the case of a recursive procedure, the system of equations $P(\varphi) = G(P)(\varphi)$ can be written as $P(\varphi) = H(P(g(\varphi)))$ since, for each recursive procedure call, there is a different entry specification which depends on the entry specification of the main call. To solve $P(\varphi) = G(P)(\varphi)$ exactly, it is not possible to simplify and consider only the entry specification of a given main call, as this leads to considering the entry specifications corresponding to each possible recursive procedure call, meaning generally an infinite number of equations. Thus, it is necessary to associate with each procedure point a predicate function (or predicate transformer). This does not raise any new theoretical difficulty, as the system of equations $P(\varphi) = G(P)(\varphi)$ can be solved using the results obtained in the preceding chapters (writing $P = F(P)$ with $F = \boldsymbol{\lambda} P \bullet \{\boldsymbol{\lambda} \varphi \bullet [G(P)(\varphi)]\}$). However, in the case of an approximate analysis, it is possible to specialize the automatic methods of approximate resolution to the case of systems of functional equations, while avoiding the difficult issue of the machine representation of functions. The concepts of given entry specifications and approximation are sufficient to avoid considering an infinite number of entry specifications.

6.1 FORWARD DEDUCTIVE SEMANTICS OF RECURSIVE PROCEDURES

Since the language we consider now contains block structures, not all program variables are visible at every program point. It follows that we have to define an environment at each program point $(x_1 : \mathcal{U}_1, \ldots, x_n : \mathcal{U}_n)$, in short $(\bar{x} : \bar{\mathcal{U}})$, which gives the visible program variables at that point, and for each variable, its domain of values. This environment is defined statically by the syntax of the language.

Let f be a procedure with α program points $a_1, \ldots, a_\alpha$ and with parameters passed by value $\bar{v} = (v_1, \ldots, v_p)$ and by result $\bar{r} = (r_1, \ldots, r_q)$ that take their values in $\bar{\mathcal{T}} = (\mathcal{T}_1, \ldots, \mathcal{T}_p)$ and $\bar{\mathcal{T}}' = (\mathcal{T}'_1, \ldots, \mathcal{T}'_q)$. Each program point a_i of the procedure f is associated with a predicate transformer $\phi_i \in ((\bar{\mathcal{T}} \to \mathcal{B}) \to (\bar{\mathcal{U}} \to \mathcal{B}))$ where $\mathcal{B} = \{\underline{\text{true}}, \underline{\text{false}}\}$, which is obtained as the least solution of a system of forward semantic equations associated with procedure f. Let $\varphi \in (\bar{\mathcal{T}} \to \mathcal{B})$ be an entry specification of procedure f, then the set of possible values for variables $\bar{x}$ at point a_i during the execution of f (called with input parameters with values $\bar{v}$ satisfying $\varphi(\bar{v})$) is characterized by $\phi_i(\varphi)$.

Now, we give the rules to build the system of forward equations associated with a procedure. The rules for an assignment, a test, or an unconditional branching are the same as in Chapter 3, except that they take into account the scope of identifiers resulting from the block structure. The main difficulty consists in stating the rules for declarations and procedure calls.

Assignments:

Let $(\bar{x} : \bar{\mathcal{U}})$ and ϕ_i be the environment and predicate transformer associated with point a_i of a procedure, before the assignment $\mathrm{\bar{x} := e(\bar{x})}$, where $e \in (\bar{\mathcal{U}} \rightarrow \bar{\mathcal{U}})$. Then, the environment corresponding to point a_j after the assignment is $(\bar{x} : \bar{\mathcal{U}})$ and $\phi_j = \underline{\mathrm{assign}}(e) \circ \phi_i$. To summarize, we write:

```
    $\langle \phi_i,\ (\bar{x} : \bar{\mathcal{U}}) \rangle$
    $\mathrm{\bar{x} := e(\bar{x})}$ ;
    $\langle \phi_j = \underline{\mathrm{assign}}(e) \circ \phi_i,\ (\bar{x} : \bar{\mathcal{U}}) \rangle$
```

Conditional instruction:

```
    $\langle \phi_i,\ (\bar{x} : \bar{\mathcal{U}}) \rangle$
    $\underline{\mathrm{if}}$ $\mathrm{p(\bar{x})}$ $\underline{\mathrm{then}}$        (where $p \in (\bar{\mathcal{U}} \rightarrow \mathcal{B})$)
        $\langle \phi_j = \underline{\mathrm{test}}(p) \circ \phi_i,\ (\bar{x} : \bar{\mathcal{U}}) \rangle$
        ...
     $\underline{\mathrm{else}}$
        $\langle \phi_k = \underline{\mathrm{test}}(\underline{\mathrm{not}}\,(p)) \circ \phi_i,\ (\bar{x} : \bar{\mathcal{U}}) \rangle$
        ...
     $\underline{\mathrm{endif}}$ ;
```

Block:

Unlike ALGOL 60 (Naur [1963]), we consider that only one declaration can apply to an identifier at each program point, that is to say that, as in LIS, an identifier I declared in a block A cannot be redeclared in a block B inside A (which would mean that, inside B, the declaration of I from B hides the declaration of I from A). This is not a semantic restriction since it is always possible to modify syntactically the identifiers of the inner block. Besides, this language feature has been adopted by some programing languages (Ichbiah, Rissen, Héliard & Cousot [1974], Lampson et al. [1976]).

```
⟨$\phi_i$, $(\bar{x} : \bar{\mathcal{U}})$⟩
    begin
        $\mathrm{y_1 : t_1, \ldots, y_m : t_m}$ ;
        ⟨$\phi_j = \underline{\text{begin}}\ (\bar{\mathcal{U}}, \bar{\mathcal{T}}) \circ \phi_i$, $(\bar{x} : \bar{\mathcal{U}}, \bar{y} : \bar{\mathcal{T}})$⟩
        ...
        ⟨$\phi_k$, $(\bar{x} : \bar{\mathcal{U}}, \bar{y} : \bar{\mathcal{T}})$⟩
    end ;
⟨$\phi_l = \underline{\text{end}}(\bar{\mathcal{U}}, \bar{\mathcal{T}}) \circ \phi_k$, $(\bar{x} : \bar{\mathcal{U}})$⟩
```

where

- the identifiers $\bar{y} = y_1, \ldots, y_m$ differ syntactically from $\bar{x} = x_1, \ldots, x_n$.

- $\bar{\mathcal{T}} = \mathcal{T}_1 \times \ldots \times \mathcal{T}_m$ and the $\mathcal{T}_1, \ldots, \mathcal{T}_m$ are the domains of values for the variables of type $t_1, \ldots, t_m$.

- $\underline{\text{begin}}\ (\bar{\mathcal{U}}, \bar{\mathcal{T}})$

$$= \underline{\text{begin}}\ ((\mathcal{U}_1 \times \ldots \times \mathcal{U}_n), (\mathcal{T}_1 \times \ldots \times \mathcal{T}_m))$$

$$\in ((\bar{\mathcal{U}} \to \mathcal{B}) \to (\bar{\mathcal{U}} \times \bar{\mathcal{T}} \to \mathcal{B}))$$

$$\in ((\mathcal{U}_1 \times \ldots \times \mathcal{U}_n \to \mathcal{B}) \to (\mathcal{U}_1 \times \ldots \times \mathcal{U}_n \times \mathcal{T}_1 \times \ldots \times \mathcal{T}_m \to \mathcal{B}))$$

$$= \boldsymbol{\lambda} P \bullet \{\boldsymbol{\lambda} (\bar{x}, \bar{y}) \bullet [P(\bar{x})\ \underline{\text{and}}\ \bar{y} = \bar{\Omega}]\}$$

$$= \boldsymbol{\lambda} P \bullet \{\boldsymbol{\lambda} (x_1, \ldots, x_n, y_1, \ldots, y_m) \bullet [P(x_1, \ldots, x_n)\ \underline{\text{and}}\ (\underset{j=1}{\overset{m}{\underline{\text{AND}}}}(y_j = \Omega_j))]\}$$

where Ω_j is the "non-initialized" value in $\mathcal{T}_j$

- $\underline{\text{end}}(\bar{\mathcal{U}}, \bar{\mathcal{T}})$

$$\begin{aligned}
&\in \quad ((\mathcal{U} \times \bar{\mathcal{T}} \to \mathcal{B}) \to (\bar{\mathcal{U}} \to \mathcal{B}))\\
&= \quad \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\,(\bar{x}) \bullet [\exists \bar{v} \in \bar{\mathcal{T}} : P(\bar{x}, \bar{v})]\}\\
&= \quad \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\,(x_1, \ldots, x_n) \bullet [\exists (v_1, \ldots, v_m) \in \mathcal{T}_1 \times \ldots \times \mathcal{T}_m :\\
&\qquad\qquad P(x_1, \ldots, x_n, v_1, \ldots, v_m)]\}
\end{aligned}$$

We simply express that, in the block, the variables $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$ are visible. At the block entry, we know that variables $x_1, \ldots, x_n$ have the same value as outside the block and that variables $y_1, \ldots, y_m$ are not initialized. At the block exit, variables $y_1, \ldots, y_m$, local to the block, are eliminated.

Let $P, i, n, i_1, \ldots, i_q$ be such that $P \in (\mathcal{U}_1 \times \ldots \times \mathcal{U}_n \to \mathcal{B})$, $(1 \leq i, q \leq n)$, $(\forall k \in [1, q], (1 \leq i_k \leq n))$ and $(\forall k, l \in [1, q], (k \neq l) \Rightarrow (i_k \neq i_l))$. In the following we will use the notations:

$$\begin{aligned}
\bar{\sigma}_i^n(P) \quad &= \quad \boldsymbol{\lambda}\,(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n) \bullet [\exists a \in \mathcal{U}_i :\\
&\qquad P(v_1, \ldots, v_{i-1}, a, v_{i+1}, \ldots, v_n)]\\
\bar{\sigma}_{i_1, \ldots, i_q}^n \quad &= \quad \bar{\sigma}_{i_1}^{n-q+1} \circ \bar{\sigma}_{i_2}^{n-q+2} \circ \ldots \circ \bar{\sigma}_{i_{q-1}}^{n-1} \circ \bar{\sigma}_{i_q}^n\\
&\qquad (\text{note that: } \underline{\text{end}}((\mathcal{U}_1 \times \ldots \times \mathcal{U}_n), (\mathcal{T}_1 \times \ldots \times \mathcal{T}_m)) = \bar{\sigma}_{n+1, \ldots, n+m}^{n+m})\\
\sigma_i^n(P) \quad &= \quad \boldsymbol{\lambda}\, x \bullet [\exists (a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n) \in (\mathcal{U}_1 \times \ldots \times \mathcal{U}_{i-1} \times \mathcal{U}_{i+1} \times \ldots \times \mathcal{U}_n) :\\
&\qquad P(a_1, \ldots, a_{i-1}, x, a_{i+1}, \ldots, a_n)]\\
\sigma_{i_1, \ldots, i_q}^n(P) \quad &= \quad \boldsymbol{\lambda}\,(x_1, \ldots, x_q) \bullet [(\forall k \in [1, n] - \{i_1, \ldots, i_q\}, \exists a_k \in \mathcal{U}_k : P(v_1, \ldots, v_n)]\\
&\qquad \text{where } \forall l \in [1, n], v_l = \underline{\text{if}}\ (\exists j \in [1, q] : l = i_j)\ \underline{\text{then}}\ x_j\ \underline{\text{else}}\ a_l\ \underline{\text{endif}}\ ;
\end{aligned}$$

Unconditional branching and labels:

We assume that the values of all labels are statically determined and that an unconditional branching always occurs within the same procedure. (Jumping outside a procedure, which would lead to more complex equations, is thus excluded.)

The predicate transformer ϕ_i associated with the program point a_i following a label L is the disjunction of the predicate transformers ϕ_j associated with the program points a_j preceding an instruction goto L or the label L. In the case of an unconditional branching outside a block, the block exit must be considered. Since we want to avoid an heavy syntactic formalism to describe that rule, an example will be sufficient:

```
            $\langle(\bar{x} : \bar{\mathcal{U}})\rangle$
            ...
{1}
            goto L ;
            ...
{2}
      L:
{3}
            ...
              begin
                $y_1 : t_1$ ;
                ...
                begin
                  $y_2 : t_2$ ;
                  ...
{4}
                  goto L ;
                  ...
                 end ;
                ...
{5}
                goto L ;
                ...
               end ;
            ...
{6}
            goto L ;
```

We have:

$$\phi_3 = [\phi_1]\,\underline{\text{or}}\,[\phi_2]\,\underline{\text{or}}\,[\ \underline{\text{end}}((\bar{\mathcal{U}}\times\mathcal{T}_1),(\mathcal{T}_2))\circ\underline{\text{end}}((\bar{\mathcal{U}}),(\mathcal{T}_1))\circ\phi_4]\,\underline{\text{or}}\,[\ \underline{\text{end}}((\bar{\mathcal{U}}),(\mathcal{T}_1))\circ \phi_5]\ \underline{\text{or}}\ [\phi_6]$$

Body of a procedure:

We consider that parameters are passed either by value (the value of the actual parameter is copied into the formal parameter before the procedure call) or by result (the value of the formal parameter is copied into the actual parameter after the procedure call). To simplify the rules to construct the system of forward semantic equations:

- We consider that, for a parameter passed by value, the actual parameter is a variable. (We can derive the rule corresponding to the general case by considering that the call $f(e)$ is equivalent to $\underline{\text{begin}}$ z : t ; z := e ; f(z) ; $\underline{\text{end}}$; .)

- We will not consider parameter passing by value–result (since the call $f(x)$ is equivalent to $\underline{\text{begin}}$ z : t ; f′(x, z) ; x := z ; $\underline{\text{end}}$; where the body of procedure f' is the same as f except that it ends with the assignment of x to z).

- We consider also that no global variable is visible inside the procedure (since the mechanism of parameter passing can be used to access or modify a global variable, provided that the global variables which are accessible in the body of the procedure are also accessible at each call point).

- We leave out functions, which are a mere syntactic convenience that can always be replaced with a procedure introducing an additional result parameter.

All the above restrictions in fact concern only writing conveniences and the only semantic restriction is the following:

- We suppose that we can always associate statically a procedure body with a procedure call (which excludes passing functions or procedures as procedure parameters).

We can now state the construction rule for the equations associated with the body of a procedure:

```
procedure f($\bar{\mathrm{v}}$ : $\bar{\mathrm{t}}$ value ; $\bar{\mathrm{r}}$ : $\bar{\mathrm{t}}'$ result) =
    $\langle \phi_i = \underline{\text{input}}(\bar{\mathcal{T}}, \bar{\mathcal{T}}'),\ (\bar{v}' : \bar{\mathcal{T}}, \bar{v} : \bar{\mathcal{T}}, \bar{r} : \bar{\mathcal{T}}') \rangle$
    ...
    $\langle \phi_j \rangle$
end–proc ;
$\langle \phi_k = \underline{\text{output}}(\bar{\mathcal{T}}, \bar{\mathcal{T}}') \circ \phi_j \rangle$
```

where

- The variables $\bar{v}'$ do not appear in the body of the procedure,

- $\bar{\mathcal{T}}$ and $\bar{\mathcal{T}}'$ are the domains of values of variables $\bar{v}$ and $\bar{r}$ of types $\bar{t}$ and $\bar{t}'$,

- $\underline{\text{input}}(\bar{\mathcal{T}}, \bar{\mathcal{T}}')$

$$\in \quad ((\bar{\mathcal{T}} \to \mathcal{B}) \to (\bar{\mathcal{T}} \times \bar{\mathcal{T}}' \times \bar{\mathcal{T}} \to \mathcal{B}))$$

$$= \quad \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\, (\bar{v}', \bar{r}, \bar{v}) \bullet [P(\bar{v}')\ \underline{\text{and}}\ (\bar{v} = \bar{v}')\ \underline{\text{and}}\ (\bar{r} = \bar{\Omega})]\}$$

- $\underline{\text{output}}(\bar{\mathcal{T}}, \bar{\mathcal{T}}')$

$$\in \quad ((\bar{\mathcal{T}} \times \bar{\mathcal{T}}' \times \bar{\mathcal{T}} \to \mathcal{B}) \to (\bar{\mathcal{T}} \times \bar{\mathcal{T}}' \to \mathcal{B}))$$

$$= \quad \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\, (\bar{v}', \bar{r}) \bullet [\exists \bar{v} \in \bar{\mathcal{T}} : P(\bar{v}', \bar{v}, \bar{r})]\}$$

We note that, if $\bar{\mathcal{T}} = \mathcal{T}_1 \times \ldots \times \mathcal{T}_p$ and $\bar{\mathcal{T}}' = \mathcal{T}_{p+1} \times \ldots \times \mathcal{T}_q$, then

$$\underline{\text{output}}(\bar{\mathcal{T}}, \bar{\mathcal{T}}') \quad = \quad \sigma^{p+q}_{1,\ldots,q}$$

Intuitively, a specification of the parameters passed by value $\bar{v}$ is known, whereas the parameters passed by result $\bar{r}$ are not initialized. The initial value of the input parameters $\bar{v}$ is copied into ancillary variables $\bar{v}'$ so that, at the procedure exit, we know a predicate, on the initial value of the input parameters and the final value of the input and output parameters, that expresses the termination condition for the procedure as well as the final value of the parameters. Since the final value of the input parameters is of no interest in the call context, the variables $\bar{v}$ are eliminated at the exit of the procedure.

Procedure call:

To avoid choosing an order of parameters passed by result, we suppose that all the actual parameters are syntactically different variables.

Let $P(\bar{x}, \bar{v}, \bar{r})$ be a predicate characterizing the domain of the variables $\bar{x}$, $\bar{v}$, and $\bar{r}$ before the call to procedure $f(\bar{v}, \bar{r})$ which takes place in the syntactic environment $(\bar{x} : \bar{\mathcal{U}}, \bar{v} : \bar{\mathcal{T}}, \bar{r} : \bar{\mathcal{T}}')$. The variables $\bar{x}$ do not occur in the call and the parameters $\bar{v}$ passed by value are not modified, so that $Q(\bar{x}, \bar{v}) = \{\exists \bar{r} \in \bar{\mathcal{T}}' : P(\bar{x}, \bar{v}, \bar{r})\}$ is true after the call. The specification of the input parameters $\bar{v}$ before the call is $\varphi = \boldsymbol{\lambda}\, \bar{v} \bullet \{\exists \bar{x} \in \bar{\mathcal{U}}, \exists \bar{r} \in \bar{\mathcal{T}}' : P(\bar{x}, \bar{v}, \bar{r})\}$. Let $\bar{f}$ be the predicate transformer associated with procedure f, we saw that $\bar{f}(\varphi)(\bar{v}, \bar{r})$ expresses a condition on the values of the input parameters $\bar{v}$ such that the execution of f terminates and characterizes the domain of the results $\bar{r}$ of the procedure. So, we have:

$\langle P(\bar{x}, \bar{v}, \bar{r}),\ (\bar{x} : \bar{\mathcal{U}}, \bar{v} : \bar{\mathcal{T}}, \bar{r} : \bar{\mathcal{T}}')\rangle$

$\mathrm{f}(\bar{\mathrm{v}}; \bar{\mathrm{r}})$;

$\langle (\bar{f}(\varphi)(\bar{v}, \bar{r})\ \underline{\text{and}}\ Q(\bar{x}, \bar{v})),\ (\bar{x} : \bar{\mathcal{U}}, \bar{v} : \bar{\mathcal{T}}, \bar{r} : \bar{\mathcal{T}}')\rangle$

where

- $\varphi \quad = \quad \boldsymbol{\lambda}\, \bar{v} \bullet \{\exists \bar{x} \in \bar{\mathcal{U}}, \exists \bar{r} \in \bar{\mathcal{T}}' : P(\bar{x}, \bar{v}, \bar{r})\}$
- $Q \quad = \quad \boldsymbol{\lambda}\, (\bar{x}, \bar{v}) \bullet \{\exists \bar{r} \in \bar{\mathcal{T}}' : P(\bar{x}, \bar{v}, \bar{r})\}$

To be more precise, we can detail this rule as follows:

$\langle \phi_i,\ (x_1 : \mathcal{U}_1, \ldots, x_n : \mathcal{U}_n)\rangle$

$\mathrm{f}(\mathrm{x}_{\mathrm{i}_1}, \ldots, \mathrm{x}_{\mathrm{i}_\mathrm{p}}; \mathrm{x}_{\mathrm{i}_{\mathrm{p}+1}}, \ldots, \mathrm{x}_{\mathrm{i}_\mathrm{q}})$;

$\langle \phi_j = \underline{\mathrm{call}}(f, (\mathcal{U}_1 \times \ldots \times \mathcal{U}_n), (i_1, \ldots, i_p), (i_{p+1}, \ldots, i_q)) \circ \phi_i, (x_1 : \mathcal{U}_1, \ldots, x_n : \mathcal{U}_n)\rangle$

where

- $\underline{\mathrm{call}}(f, (\mathcal{U}_1 \times \ldots \times \mathcal{U}_n), (i_1, \ldots, i_p), (i_{p+1}, \ldots, i_q))$

$$\begin{aligned} &\in \quad ((\mathcal{U}_1 \times \ldots \times \mathcal{U}_n \rightarrow \mathcal{B}) \rightarrow (\mathcal{U}_1 \times \ldots \times \mathcal{U}_n \rightarrow \mathcal{B})) \\ &= \quad \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\, (x_1, \ldots, x_n) \bullet [\phi_k(\sigma^n_{i_1,\ldots,i_p}(P))(x_{i_1}, \ldots, x_{i_p}, x_{i_{p+1}}, \ldots, x_{i_q}) \quad \underline{\text{and}} \\ &\qquad \sum_{i_{p+1},\ldots,i_q}^{n} (\bar{\sigma}^n_{i_{p+1},\ldots,i_q}(P))(x_1, \ldots, x_n)]\} \end{aligned}$$

where

- ϕ_k is associated with the program point following the <u>end–proc</u> of the body of procedure f,
- If $(r \leq n), (\forall k \in [1, r], 1 \leq i_k \leq n), (\forall k, l \in [1, r], (k \neq l) \Rightarrow (i_k \neq i_l))$ <u>and</u> $P \in (\mathcal{U}_1 \times \ldots \times \mathcal{U}_{n-r} \rightarrow \mathcal{B})$, then $\sum\limits_{i_1,\ldots,i_r}^{n} (P) = \boldsymbol{\lambda}\, (x_1, \ldots, x_n) \bullet [P(v_{j_1}, \ldots, v_{j_{n-r}})]$ where $\forall k \in [1, n-r], j_k = min\{i : (i > j_{k-1})$ <u>and</u> $(\forall l \in [1, r], i \neq i_l)\}$ with $j_0 = 0$.

Example 6.1.0.1

We illustrate the association of a system of forward semantic equations with a recursive procedure call on the classic factorial example:

```
        procedure f(x : integer value ; y : integer result) =
{1}
            if x = 0 then
{2}
                y := 1 ;
{3}
            else
{4}
                begin z : integer ;
{5}
                    z := x − 1 ;
{6}
                    f(z; y) ;
{7}
                    y := x ∗ y ;
{8}
                 end ;
{9}
            endif ;
{10}
        end–proc ;
{11}
```

The system of forward semantic equations associated with this procedure is:

$$\begin{aligned}
\phi_1 &\in (\underline{\text{integer}} \to \mathcal{B}) \to (\underline{\text{integer}}^3 \to \mathcal{B}) \\
&= \underline{\text{input}}[(\underline{\text{integer}}), (\underline{\text{integer}})] \\
&= \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\,(a, y, x) \bullet [P(a) \ \underline{\text{and}}\ (x = a)\ \underline{\text{and}}\ (y = \Omega)]\} \\
\phi_2 &\in (\underline{\text{integer}} \to \mathcal{B}) \to (\underline{\text{integer}}^3 \to \mathcal{B}) \\
&= \underline{\text{test}}(\boldsymbol{\lambda}\,(a, y, x) \bullet (x = 0)) \circ \phi_1 \\
&= \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\,(a, y, x) \bullet [\phi_1(P)(a, y, x)\ \underline{\text{and}}\ (x = 0)]\} \\
\phi_3 &\in (\underline{\text{integer}} \to \mathcal{B}) \to (\underline{\text{integer}}^3 \to \mathcal{B}) \\
&= \underline{\text{assign}}(\boldsymbol{\lambda}\,(a, y, x) \bullet (a, 1, x)) \circ \phi_2 \\
&= \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\,(a, y, x) \bullet [\exists m : \phi_2(P)(a, m, x)\ \underline{\text{and}}\ (y = 1)]\}
\end{aligned}$$

$$\begin{aligned}
\phi_4 &\in (\underline{\text{integer}} \rightarrow \mathcal{B}) \rightarrow (\underline{\text{integer}}^3 \rightarrow \mathcal{B}) \\
&= \underline{\text{test}}(\boldsymbol{\lambda}\,(a,y,x) \bullet (x \neq 0)) \circ \phi_1 \\
&= \boldsymbol{\lambda}\,P \bullet \{\boldsymbol{\lambda}\,(a,y,x) \bullet [\phi_1(P)(a,y,x)\ \underline{\text{and}}\ (x \neq 0)]\}
\end{aligned}$$

$$\begin{aligned}
\phi_5 &\in (\underline{\text{integer}} \rightarrow \mathcal{B}) \rightarrow (\underline{\text{integer}}^4 \rightarrow \mathcal{B}) \\
&= \underline{\text{begin}}\ (\underline{\text{integer}}^3, \underline{\text{integer}}) \circ \phi_4 \\
&= \boldsymbol{\lambda}\,P \bullet \{\boldsymbol{\lambda}\,(a,y,x,z) \bullet [\phi_4(P)(a,y,x)\ \underline{\text{and}}\ (z = \Omega)]\}
\end{aligned}$$

$$\begin{aligned}
\phi_6 &\in (\underline{\text{integer}} \rightarrow \mathcal{B}) \rightarrow (\underline{\text{integer}}^4 \rightarrow \mathcal{B}) \\
&= \underline{\text{assign}}(\boldsymbol{\lambda}\,(a,y,x,z) \bullet (a,y,x,x-1)) \circ \phi_5 \\
&= \boldsymbol{\lambda}\,P \bullet \{\boldsymbol{\lambda}\,(a,y,x,z) \bullet [\exists m : \phi_5(P)(a,y,x,m)\ \underline{\text{and}}\ (z = x-1)]\}
\end{aligned}$$

$$\begin{aligned}
\phi_7 &\in (\underline{\text{integer}} \rightarrow \mathcal{B}) \rightarrow (\underline{\text{integer}}^4 \rightarrow \mathcal{B}) \\
&= \underline{\text{call}}(f, \underline{\text{integer}}^4, (4), (2)) \circ \phi_6 \\
&= \boldsymbol{\lambda}\,P \bullet \{\boldsymbol{\lambda}\,(a,y,x,z) \bullet [\phi_{11}(\sigma_4^4(P))(z,y)\ \underline{\text{and}}\ \bar{\sigma}_2^4(P)(a,x,z)]\} \circ \phi_6 \\
&= \boldsymbol{\lambda}\,P \bullet \{\boldsymbol{\lambda}\,(a,y,x,z) \bullet [\phi_{11}(\sigma_4^4(\phi_6(P)))(z,y)\ \underline{\text{and}}\ \bar{\sigma}_2^4(\phi_6(P))(a,x,z)]\}
\end{aligned}$$

$$\begin{aligned}
\phi_8 &\in (\underline{\text{integer}} \rightarrow \mathcal{B}) \rightarrow (\underline{\text{integer}}^4 \rightarrow \mathcal{B}) \\
&= \underline{\text{assign}}(\boldsymbol{\lambda}\,(a,y,x,z) \bullet (a, x*y, x, z)) \circ \phi_7 \\
&= \boldsymbol{\lambda}\,P \bullet \{\boldsymbol{\lambda}\,(a,y,x,z) \bullet [\exists n : \phi_7(P)(a,n,x,z)\ \underline{\text{and}}\ (y = x*n)]\}
\end{aligned}$$

$$\begin{aligned}
\phi_9 &\in (\underline{\text{integer}} \rightarrow \mathcal{B}) \rightarrow (\underline{\text{integer}}^4 \rightarrow \mathcal{B}) \\
&= \underline{\text{end}}(\underline{\text{integer}}^3, \underline{\text{integer}}) \circ \phi_8 \\
&= \bar{\sigma}_4^4 \circ \phi_8
\end{aligned}$$

$$\phi_{10} \in (\underline{\text{integer}} \rightarrow \mathcal{B}) \rightarrow (\underline{\text{integer}}^3 \rightarrow \mathcal{B})$$

$$
\begin{array}{lcl}
 & = & \phi_3 \ \underline{\text{or}}\ \phi_9 \\
\phi_{11} & \in & (\underline{\text{integer}} \rightarrow \mathcal{B}) \rightarrow (\underline{\text{integer}}^2 \rightarrow \mathcal{B}) \\
 & = & \underline{\text{output}}[(\underline{\text{integer}}), (\underline{\text{integer}})] \circ \phi_{10} \\
 & = & \bar{\sigma}_3^3 \circ \phi_{10}
\end{array}
$$

End of example.

The construction rules of the system of forward semantic equations associated with a program including recursive procedures can be justified with respect to an operational or denotational semantics. Except for the complexity introduced by the recursion, the process if fundamentally the same as in Chapter 3. In particular, the least solution (for implication) of the system of equations

$$
\left\{ \ \phi \ = \ \boldsymbol{\lambda}\,\psi \bullet [\![\boldsymbol{\lambda}\,P \bullet \{F_\pi(\psi)(P)\}]\!][\![\phi]\!] \right.
$$

associated with program π is obtained by Theorem 2.7.0.1, since $\boldsymbol{\lambda}\,\psi \bullet [\![\boldsymbol{\lambda}\,P \bullet \{F_\pi(\psi)(P)\}]\!]$ is a complete join-morphism.

Example 6.1.0.2

We solve the system of equations associated with the factorial procedure of Example 6.1.0.1. To perform manual computations, it is easier to simplify the system of equations, e.g., by eliminating $\phi_1, \ldots, \phi_{10}$, which gives:

$$
\left\{ \begin{array}{lcl}
\phi_{11} & = & \boldsymbol{\lambda}\,\psi \bullet [\![\boldsymbol{\lambda}\,P \bullet \{\boldsymbol{\lambda}\,(a,y) \bullet [(P(a)\ \underline{\text{and}}\ (a=0)\ \underline{\text{and}}\ (y=1))\ \underline{\text{or}}\ (\exists n : \psi(\boldsymbol{\lambda}\,z \bullet \\
 & & \quad [P(z+1)\ \underline{\text{and}}\ (z+1 \neq 0)])(a-1,n)\ \underline{\text{and}}\ P(a)\ \underline{\text{and}}\ (a \neq 0)\ \underline{\text{and}}\ (y = \\
 & & a * n))]\}]\!][\![\phi_{11}]\!]
\end{array} \right.
$$

Solving this equation iteratively starting from the infimum of $((\underline{\text{integer}} \rightarrow \mathcal{B}) \rightarrow (\underline{\text{integer}}^2 \rightarrow \mathcal{B}))$, we get for the first terms:

$$\left[\;\phi^{0}_{11} \;=\; \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\,(a,y) \bullet [\underline{\text{false}}]\}\right.$$

$$\left[\;\phi^{1}_{11} \;=\; \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\,(a,y) \bullet [P(a)\ \underline{\text{and}}\ (a=0)\ \underline{\text{and}}\ (y=a!)]\}\right.$$

$$\left[\;\phi^{2}_{11} \;=\; \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\,(a,y) \bullet [P(a)\ \underline{\text{and}}\ (0 \le a \le 1)\ \underline{\text{and}}\ (y=a!)]\}\right.$$

The iteration is infinite, so, to get to the limit, we guess the general term ϕ^{k}_{11}, prove that it is correct by finite induction, and go to the limit by extending k to infinity. Theorem 2.7.0.1 shows that this limit is the least solution (for implication) of the functional equation. For the induction step, suppose we have:

$$\left[\;\phi^{k}_{11} \;=\; \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\,(a,y) \bullet [P(a)\ \underline{\text{and}}\ (0 \le a \le (k-1))\ \underline{\text{and}}\ (y=a!)]\}\right.$$

then we get:

- $\phi^{k}_{11}(\boldsymbol{\lambda}\, z \bullet [P(z+1)\ \underline{\text{and}}\ (z+1 \neq 0)])$

  $= \;\; \boldsymbol{\lambda}\,(a,y) \bullet [P(a+1)\ \underline{\text{and}}\ (a+1 \neq 0)\ \underline{\text{and}}\ (0 \le a \le (k-1))\ \underline{\text{and}}\ (y=a!)]$

- $\phi^{k}_{11}(\boldsymbol{\lambda}\, z \bullet [P(z+1)\ \underline{\text{and}}\ (z+1 \neq 0)])(a-1,n)$

  $= \;\; \{P(a)\ \underline{\text{and}}\ (1 \le a \le k)\ \underline{\text{and}}\ (n=(a-1)!)\}$

- $(\exists n : \phi_{11}(\boldsymbol{\lambda}\, z \bullet [P(z{+}1)\,\underline{\text{and}}\,(z{+}1 \neq 0)])(a{-}1,n)\,\underline{\text{and}}\,P(a)\,\underline{\text{and}}\,(a \neq 0)\,\underline{\text{and}}\,(y = a{*}n))$

  $= \;\; \{P(a)\ \underline{\text{and}}\ (1 \le a \le k)\ \underline{\text{and}}\ (y=a!)\}$

$$\left[\;\phi^{k+1}_{11} \;=\; \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\,(a,y) \bullet [P(a)\ \underline{\text{and}}\ (0 \le a \le k)\ \underline{\text{and}}\ (y=a!)]\}\right.$$

Going to the limit, we obtain:

$$\phi^{\omega}_{11} \;=\; \underset{k \in \omega}{\underline{\text{OR}}}(\boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\,(a,y) \bullet [P(a)\ \underline{\text{and}}\ (0 \le a \le k)\ \underline{\text{and}}\ (y=a!)]\})$$

$$\left[\ \phi_{11}^{\omega} \quad = \quad \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\,(a, y) \bullet [P(a) \ \underline{\text{and}}\ (0 \leq a) \ \underline{\text{and}}\ (y = a!)]\}\right.$$

In particular, note that factorial, as we wrote it, does not terminate for negative values of its input parameter.

End of example.

Example 6.1.0.3

MacCarthy's 91 function

$$f(x) \quad = \quad \underline{\text{if}}\ x > 100\ \ \underline{\text{then}}\ \ x - 10\ \ \underline{\text{else}}\ \ f(f(x + 11))\ \underline{\text{endif}}$$

which computes:

$$\underline{\text{if}}\ x > 100\ \ \underline{\text{then}}\ \ x - 10\ \ \underline{\text{else}}\ \ 91\ \underline{\text{endif}}$$

illustrates the case of nested recursive calls. In our language, it can be written:

```
      procedure f(x : integer value ; y : integer result) =
{1}
          if x > 100 then
{2}
              y := x − 10 ;
{3}
           else
{4}
              begin z : integer ;
{5}
                  z := x + 11 ;
{6}
                  begin t : integer ;
{7}
                      f(z; t) ;
{8}
                      f(t; y) ;
{9}
                   end ;
{10}
               end ;
```

```
{11}
        endif ;
{12}
    end–proc ;
{13}
```

The system of forward semantic equations associated with this procedure is the following:

$$\begin{aligned}
\phi_1 &\in (\underline{\text{integer}} \rightarrow \mathcal{B}) \rightarrow (\underline{\text{integer}}^3 \rightarrow \mathcal{B}) \\
&= \underline{\text{input}}[(\underline{\text{integer}}), (\underline{\text{integer}})] \\
&= \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\, (a, y, x) \bullet [P(a) \ \underline{\text{and}}\ (x = a) \ \underline{\text{and}}\ (y = \Omega)]\} \\
\phi_2 &\in (\underline{\text{integer}} \rightarrow \mathcal{B}) \rightarrow (\underline{\text{integer}}^3 \rightarrow \mathcal{B}) \\
&= \underline{\text{test}}(\boldsymbol{\lambda}\, (a, y, x) \bullet (x > 100)) \circ \phi_1 \\
&= \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\, (a, y, x) \bullet [\phi_1(P)(a, y, x) \ \underline{\text{and}}\ (x > 100)]\} \\
\phi_3 &\in (\underline{\text{integer}} \rightarrow \mathcal{B}) \rightarrow (\underline{\text{integer}}^3 \rightarrow \mathcal{B}) \\
&= \underline{\text{assign}}(\boldsymbol{\lambda}\, (a, y, x) \bullet (a, x - 10, x)) \circ \phi_2 \\
&= \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\, (a, y, x) \bullet [\exists m : \phi_2(P)(a, m, x) \ \underline{\text{and}}\ (y = x - 10)]\} \\
\phi_4 &\in (\underline{\text{integer}} \rightarrow \mathcal{B}) \rightarrow (\underline{\text{integer}}^3 \rightarrow \mathcal{B}) \\
&= \underline{\text{test}}(\boldsymbol{\lambda}\, (a, y, x) \bullet (x \leq 100)) \circ \phi_1 \\
&= \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\, (a, y, x) \bullet [\phi_1(P)(a, y, x) \ \underline{\text{and}}\ (x \leq 100)]\} \\
\phi_5 &\in (\underline{\text{integer}} \rightarrow \mathcal{B}) \rightarrow (\underline{\text{integer}}^4 \rightarrow \mathcal{B}) \\
&= \underline{\text{begin}}\ ((\underline{\text{integer}}^3), (\underline{\text{integer}})) \circ \phi_4 \\
&= \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\, (a, y, x, z) \bullet [\phi_4(P)(a, y, x) \ \underline{\text{and}}\ (z = \Omega)]\} \\
\phi_6 &\in (\underline{\text{integer}} \rightarrow \mathcal{B}) \rightarrow (\underline{\text{integer}}^4 \rightarrow \mathcal{B})
\end{aligned}$$

$$
\begin{aligned}
&= \underline{\text{assign}}(\boldsymbol{\lambda}\,(a,y,x,z)\bullet(a,y,x,x+11))\circ\phi_5\\
&= \boldsymbol{\lambda}\,P\bullet\{\boldsymbol{\lambda}\,(a,y,x,z)\bullet[\exists m:\phi_5(P)(a,y,x,m)\ \underline{\text{and}}\ (z=x+11)]\}\\
\phi_7 &\in (\underline{\text{integer}}\rightarrow\mathcal{B})\rightarrow(\underline{\text{integer}}^5\rightarrow\mathcal{B})\\
&= \underline{\text{begin}}\ ((\underline{\text{integer}}^4),(\underline{\text{integer}}))\circ\phi_6\\
&= \boldsymbol{\lambda}\,P\bullet\{\boldsymbol{\lambda}\,(a,y,x,z,t)\bullet[\phi_6(P)(a,y,x,z)\ \underline{\text{and}}\ (t=\Omega)]\}\\
\phi_8 &\in (\underline{\text{integer}}\rightarrow\mathcal{B})\rightarrow(\underline{\text{integer}}^5\rightarrow\mathcal{B})\\
&= \underline{\text{call}}(f,\underline{\text{integer}}^5,(4),(5))\circ\phi_7\\
&= \boldsymbol{\lambda}\,P\bullet\{\boldsymbol{\lambda}\,(a,y,x,z,t)\bullet[\phi_{13}(\sigma^5_4(\phi_7(P)))(z,t)\ \underline{\text{and}}\ \bar{\sigma}^5_5(\phi_7(P))(a,y,x,z)]\}\\
\phi_9 &\in (\underline{\text{integer}}\rightarrow\mathcal{B})\rightarrow(\underline{\text{integer}}^5\rightarrow\mathcal{B})\\
&= \underline{\text{call}}(f,\underline{\text{integer}}^5,(5),(2))\circ\phi_8\\
&= \boldsymbol{\lambda}\,P\bullet\{\boldsymbol{\lambda}\,(a,y,x,z,t)\bullet[\phi_{13}(\sigma^5_5(\phi_8(P)))(z,t)\ \underline{\text{and}}\ \bar{\sigma}^5_2(\phi_8(P))(a,y,x,z)]\}\\
\phi_{10} &\in (\underline{\text{integer}}\rightarrow\mathcal{B})\rightarrow(\underline{\text{integer}}^4\rightarrow\mathcal{B})\\
&= \underline{\text{end}}((\underline{\text{integer}}^4),(\underline{\text{integer}}))\circ\phi_9\\
&= \bar{\sigma}^5_5\circ\phi_9\\
\phi_{11} &\in (\underline{\text{integer}}\rightarrow\mathcal{B})\rightarrow(\underline{\text{integer}}^3\rightarrow\mathcal{B})\\
&= \underline{\text{end}}((\underline{\text{integer}}^3),(\underline{\text{integer}}))\circ\phi_{10}\\
&= \bar{\sigma}^4_4\circ\phi_{10}\\
\phi_{12} &\in (\underline{\text{integer}}\rightarrow\mathcal{B})\rightarrow(\underline{\text{integer}}^3\rightarrow\mathcal{B})\\
&= \phi_3\ \underline{\text{or}}\ \phi_{11}\\
\phi_{13} &\in (\underline{\text{integer}}\rightarrow\mathcal{B})\rightarrow(\underline{\text{integer}}^2\rightarrow\mathcal{B})
\end{aligned}
$$

$$\begin{array}{cl} = & \underline{\text{output}}[(\underline{\text{integer}}), (\underline{\text{integer}})] \circ \phi_{12} \\ = & \bar{\sigma}_3^3 \circ \phi_{12} \end{array}$$

This system of equations can be simplified as follows:

$$\left\{ \begin{array}{lcl} \phi_8 & = & \boldsymbol{\lambda} P \bullet \{\boldsymbol{\lambda}(a,y,x,z,t) \bullet [\phi_{13}(\boldsymbol{\lambda} z \bullet [P(z-11) \ \underline{\text{and}} \ (z \leq 111)])(z,t) \ \underline{\text{and}} \\ & & P(a) \ \underline{\text{and}} \ (x = a) \ \underline{\text{and}} \ (y = \Omega) \ \underline{\text{and}} \ (x \leq 100) \ \underline{\text{and}} \ (z = x + 11)]\} \\ \phi_9 & = & \boldsymbol{\lambda} P \bullet \{\boldsymbol{\lambda}(a,y,x,z,t) \bullet [\phi_{13}(\sigma_5^5(P))(t,y) \ \underline{\text{and}} \ \bar{\sigma}_2^5(P)(a,x,z,t)]\} \circ \phi_8 \\ \phi_{13} & = & \boldsymbol{\lambda} P \bullet \{\boldsymbol{\lambda}(a,y) \bullet [P(a) \ \underline{\text{and}} \ (a > 100) \ \underline{\text{and}} \ (y = a - 10)]\}) \ \underline{\text{or}} \ (\sigma_{1,2}^5 \circ \phi_9) \end{array} \right.$$

which can be written:

$$\left\{ \begin{array}{lcl} \phi & = & \boldsymbol{\lambda} P \bullet \{\boldsymbol{\lambda}(a,y) \bullet [P(a) \ \underline{\text{and}} \ (a > 100) \ \underline{\text{and}} \ (y = a - 10)] \ \underline{\text{or}} \\ & & \sigma_{1,2}^5(\psi(\boldsymbol{\lambda}(a,y,x,z,t) \bullet [\phi(\boldsymbol{\lambda} z \bullet [P(z-11) \ \underline{\text{and}} \ (z \leq 111)])(z,t) \ \underline{\text{and}} \\ & & P(a) \ \underline{\text{and}} \ (a \leq 100) \ \underline{\text{and}} \ (x = a) \ \underline{\text{and}} \ (y = \Omega) \ \underline{\text{and}} \ (z = x + 11)])))\} \\ \psi & = & \boldsymbol{\lambda} P \bullet \{\boldsymbol{\lambda}(a,y,x,z,t) \bullet [\phi(\sigma_5^5(P))(t,y) \ \underline{\text{and}} \ \bar{\sigma}_2^5(P)(a,x,z,t)]\} \end{array} \right.$$

The least fixpoint is computed by successive approximations starting form the infimum:

$$\left[\begin{array}{lcl} \phi^0 & = & \boldsymbol{\lambda} P \bullet \{\boldsymbol{\lambda}(a,y) \bullet [\underline{\text{false}}]\} \\ \psi^0 & = & \boldsymbol{\lambda} P \bullet \{\boldsymbol{\lambda}(a,y,x,z,t) \bullet [\underline{\text{false}}]\} \end{array} \right.$$

$$\left[\begin{array}{lcl} \phi^1 & = & \boldsymbol{\lambda} P \bullet \{\boldsymbol{\lambda}(a,y) \bullet [P(a) \ \underline{\text{and}} \ (a > 100) \ \underline{\text{and}} \ (y = a - 10)]\} \end{array} \right.$$

$$\left[\begin{array}{lcl} \phi^2 & = & \boldsymbol{\lambda} P \bullet \{\boldsymbol{\lambda}(a,y) \bullet [P(a) \ \underline{\text{and}} \ (((a > 100) \ \underline{\text{and}} \ (y = a - 10)) \ \underline{\text{or}} \ ((a = 100) \ \underline{\text{and}} \\ & & (y = 91)))]\} \end{array} \right.$$

With a little imagination we foresee the general term of the sequence as follows:

$$\left[\begin{array}{lcl} \phi^k & = & \boldsymbol{\lambda} P \bullet \{\boldsymbol{\lambda}(a,y) \bullet [P(a) \ \underline{\text{and}} \ \{((a > 100) \ \underline{\text{and}} \ (y = a - 10)) \ \underline{\text{or}} \ ((k < 11) \ \underline{\text{and}} \\ & & (102 - k \leq a \leq 100) \ \underline{\text{and}} \ (y = 91)) \ \underline{\text{or}} \ ((k \geq 11) \ \underline{\text{and}} \ (91 - 11 * (k - 11) \leq \\ & & a \leq 100) \ \underline{\text{and}} \ (y = 91))\}]\} \\ \psi^k & = & \boldsymbol{\lambda} P \bullet \{\boldsymbol{\lambda}(a,y,x,z,t) \bullet [\phi^k(\sigma_5^5(P))(t,y) \ \underline{\text{and}} \ \bar{\sigma}_2^5(P)(a,x,z,t)]\} \end{array} \right.$$

The induction step ($k \geq 2$) shows that this hypothesis is correct:

$$\begin{aligned}
&\phi^k(\boldsymbol{\lambda}\, z \bullet [P(z-11)\ \underline{\text{and}}\ (z \le 111)])(z,t)\\
&\quad = [P(z-11)\ \underline{\text{and}}\ (z \le 111)\ \underline{\text{and}}\ \{((z > 100)\ \underline{\text{and}}\ (t = z-10))\ \underline{\text{or}}\ ((k \le 11)\ \underline{\text{and}}\ (102-k \le z \le 100)\ \underline{\text{and}}\ (t = 91))\ \underline{\text{or}}\ ((k \ge 11)\ \underline{\text{and}}\ (91 - 11 * (k-11) \le z \le 100)\ \underline{\text{and}}\ (t = 91))\}]
\end{aligned}$$

Let

$$Q = \boldsymbol{\lambda}\,(a,y,x,z,t) \bullet [\phi^k(\boldsymbol{\lambda}\, z \bullet [P(z-11)\ \underline{\text{and}}\ (z \le 111)])(z,t)\ \underline{\text{and}}\ P(a)\ \underline{\text{and}}\ (a \le 100)\ \underline{\text{and}}\ (x = a)\ \underline{\text{and}}\ (y = \Omega)\ \underline{\text{and}}\ (z = x+11)]$$

After substitution and simplification, we get:

$$Q = \boldsymbol{\lambda}\,(a,y,x,z,t) \bullet [P(a)\ \underline{\text{and}}\ (x = a)\ \underline{\text{and}}\ (y = \Omega)\ \underline{\text{and}}\ (z = a+11)\ \underline{\text{and}}\ \{((90 \le a \le 100)\ \underline{\text{and}}\ (t = a+1))\ \underline{\text{or}}\ ((k \le 11)\ \underline{\text{and}}\ (91-k \le a \le 89)\ \underline{\text{and}}\ (t = 91))\ \underline{\text{or}}\ ((k \ge 11)\ \underline{\text{and}}\ (91 - 11 * ((k+1)-11) \le a \le 89)\ \underline{\text{and}}\ (t = 91))\}]$$

It follows that $\sigma_5^5(Q) = \boldsymbol{\lambda}\, t \bullet [((91 \le t \le 100)\ \underline{\text{and}}\ P(t-1))\ \underline{\text{or}}\ (t = 91)]$, which allows the evaluation of

$$\begin{aligned}
&\phi^k(\sigma_5^5(Q))(t,y)\\
&\quad = ((y = 91)\ \underline{\text{and}}\ (\{P(t-1)\ \underline{\text{and}}\ \{(t = 101)\ \underline{\text{or}}\ ((k \le 11)\ \underline{\text{and}}\ (102-k \le t \le 100))\ \underline{\text{or}}\ ((k \ge 11)\ \underline{\text{and}}\ (91 \le t \le 100))\}\}\ \underline{\text{or}}\ \{(k \ge 11)\ \underline{\text{and}}\ (t = 91)\}))
\end{aligned}$$

as

$$\begin{aligned}
&\sigma_{1,2}^5(\psi^k(Q))\\
&\quad = \boldsymbol{\lambda}\,(a,y) \bullet [P(a)\ \underline{\text{and}}\ (y = 91)\ \underline{\text{and}}\ \{(a = 100)\ \underline{\text{or}}\ ((k \le 11)\ \underline{\text{and}}\ (102-(k+1) \le a \le 99))\ \underline{\text{or}}\ ((k \ge 11)\ \underline{\text{and}}\ (90 \le a \le 99))\ \underline{\text{or}}\ ((k \ge 11)\ \underline{\text{and}}\ (a = 90))\ \underline{\text{or}}\ ((k = 11)\ \underline{\text{and}}\ (91-k \le a \le 89))\ \underline{\text{or}}\ ((k \ge 11)\ \underline{\text{and}}\ (91 - 11 * (k+1) - 11 \le a \le 89))\}]
\end{aligned}$$

we obtain:

$$\left[\ \phi^{k+1} = \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\,(a,y) \bullet [P(a)\ \underline{\text{and}}\ \{((a > 100)\ \underline{\text{and}}\ (y = a-10))\ \underline{\text{or}}\ ((k+1 \le 11)\ \underline{\text{and}}\ (102-(k+1) \le a \le 100)\ \underline{\text{and}}\ (y = 91))\ \underline{\text{or}}\ ((k+1 \ge 11)\ \underline{\text{and}}\ (91 - 11 * ((k+1)-11) \le a \le 100)\ \underline{\text{and}}\ (y = 91))\}]\}\right.$$

Going to the limit, the least fixpoint is:

$$\left[\begin{array}{lcl} \phi^{\omega} & = & \boldsymbol{\lambda} P \bullet \{\boldsymbol{\lambda}(a,y)\bullet[P(a)\ \underline{\text{and}}\ \{((a>100)\ \underline{\text{and}}\ (y=a-10))\ \underline{\text{or}}\ ((a\leq \\ & & 100)\ \underline{\text{and}}\ (y=91))\}]\} \end{array}\right.$$

End of example.

Example 6.1.0.4

This last example (computing powers of two) illustrates the use of Theorem 4.1.1.0.2 to compute, by hand, an over-approximation of the least fixpoint of the system of equations associated with a procedure, which can provide a proof of partial correctness.

```
         procedure f(x : integer value ; y : integer result) =
{1}
              y := 1 ;
{2}
         L:
{3}
              if x > 0 then
{4}
                   x := x − 1 ;
{5}
                   begin z : integer ;
{6}
                        f(x; z) ;
{7}
                        y := y + z ;
{8}
                        goto L ;
                    end ;
               endif ;
{9}
         end–proc ;
{10}
```

The system of forward semantic equations associated with this procedure is the following:

$$
\begin{array}{lcl}
\phi_1 & \in & (\underline{\text{integer}} \rightarrow \mathcal{B}) \rightarrow (\underline{\text{integer}}^3 \rightarrow \mathcal{B}) \\
 & = & \underline{\text{input}}[(\underline{\text{integer}}), (\underline{\text{integer}})] \\
 & = & \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\, (a, y, x) \bullet [P(a) \ \underline{\text{and}} \ (x = a) \ \underline{\text{and}} \ (y = \Omega)]\} \\
\phi_2 & \in & (\underline{\text{integer}} \rightarrow \mathcal{B}) \rightarrow (\underline{\text{integer}}^3 \rightarrow \mathcal{B}) \\
 & = & \underline{\text{assign}}(\boldsymbol{\lambda}\, (a, y, x) \bullet (a, 1, x)) \circ \phi_1 \\
 & = & \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\, (a, y, x) \bullet [\exists m : \phi_1(P)(a, m, x) \ \underline{\text{and}} \ (y = 1)]\} \\
\phi_3 & \in & (\underline{\text{integer}} \rightarrow \mathcal{B}) \rightarrow (\underline{\text{integer}}^3 \rightarrow \mathcal{B}) \\
 & = & \phi_2 \, \underline{\text{or}} \ \ \underline{\text{end}}((\underline{\text{integer}}^4), (\underline{\text{integer}})) \circ \phi_8 \\
 & = & \phi_2 \ \underline{\text{or}} \ \bar{\sigma}_4^4 \circ \phi_8 \\
\phi_4 & \in & (\underline{\text{integer}} \rightarrow \mathcal{B}) \rightarrow (\underline{\text{integer}}^3 \rightarrow \mathcal{B}) \\
 & = & \underline{\text{test}}(\boldsymbol{\lambda}\, (a, y, x) \bullet (x > 0)) \circ \phi_3 \\
 & = & \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\, (a, y, x) \bullet [\phi_3(P)(a, y, x) \ \underline{\text{and}} \ (x > 0)]\} \\
\phi_5 & \in & (\underline{\text{integer}} \rightarrow \mathcal{B}) \rightarrow (\underline{\text{integer}}^3 \rightarrow \mathcal{B}) \\
 & = & \underline{\text{assign}}(\boldsymbol{\lambda}\, (a, y, x) \bullet [a, y, x - 1]) \circ \phi_4 \\
 & = & \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\, (a, y, x) \bullet [\phi_4(P)(a, y, x + 1)]\} \\
\phi_6 & \in & (\underline{\text{integer}} \rightarrow \mathcal{B}) \rightarrow (\underline{\text{integer}}^4 \rightarrow \mathcal{B}) \\
 & = & \underline{\text{begin}} \ ((\underline{\text{integer}}^3), (\underline{\text{integer}})) \circ \phi_5 \\
 & = & \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\, (a, y, x, z) \bullet [\phi_5(P)(a, y, x) \ \underline{\text{and}} \ (z = \Omega)]\} \\
\phi_7 & \in & (\underline{\text{integer}} \rightarrow \mathcal{B}) \rightarrow (\underline{\text{integer}}^4 \rightarrow \mathcal{B})
\end{array}
$$

$$
\begin{array}{rcl}
 & = & \underline{\text{call}}(f, \underline{\text{integer}}^4, 3, 4) \circ \phi_6 \\
 & = & \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\, (a, y, x, z) \bullet [\phi_{10}(\sigma_3^4(\phi_6(P)))(x, z) \ \underline{\text{and}}\ \bar{\sigma}_4^4(\phi_6(P))(a, y, x)]\} \\
\phi_8 & \in & (\underline{\text{integer}} \rightarrow \mathcal{B}) \rightarrow (\underline{\text{integer}}^4 \rightarrow \mathcal{B}) \\
 & = & \underline{\text{assign}}(\boldsymbol{\lambda}\, (a, y, x, z) \bullet (a, y + z, x, z)) \circ \phi_7 \\
 & = & \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\, (a, y, x, z) \bullet [\exists m : \phi_7(P)(a, m, x, z) \ \underline{\text{and}}\ (y = m + z)]\} \\
\phi_9 & \in & (\underline{\text{integer}} \rightarrow \mathcal{B}) \rightarrow (\underline{\text{integer}}^3 \rightarrow \mathcal{B}) \\
 & = & \underline{\text{test}}(\boldsymbol{\lambda}\, (a, y, x) \bullet (x \leq 0)) \circ \phi_3 \\
 & = & \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\, (a, y, x) \bullet [\phi_3(P)(a, y, x) \ \underline{\text{and}}\ (x \leq 0)]\} \\
\phi_{10} & \in & (\underline{\text{integer}} \rightarrow \mathcal{B}) \rightarrow (\underline{\text{integer}}^2 \rightarrow \mathcal{B}) \\
 & = & \underline{\text{output}}[(\underline{\text{integer}}), (\underline{\text{integer}})] \circ \phi_9 \\
 & = & \bar{\sigma}_3^3 \circ \phi_9
\end{array}
$$

This system of equations can be substantially simplified into:

$$
\left\{
\begin{array}{rcl}
\phi_3 & = & F_1(\phi_3, \phi_{10}) \\
 & = & \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\, (a, y, x) \bullet [(P(a) \ \underline{\text{and}}\ (x = a) \ \underline{\text{and}}\ (y = 1)) \ \underline{\text{or}}\ (\exists m : \\
 & & \phi_{10}(\sigma_3^3(\boldsymbol{\lambda}\, (a, y, x) \bullet [\phi_3(P)(a, y, x + 1) \ \underline{\text{and}}\ (x \geq 0)]))(x, y - m) \ \underline{\text{and}} \\
 & & \phi_3(P)(a, m, x + 1) \ \underline{\text{and}}\ (x \geq 0))]\} \\
\phi_{10} & = & F_2(\phi_3, \phi_{10}) \\
 & = & \boldsymbol{\lambda}\, P \bullet \{\sigma_3^3(\boldsymbol{\lambda}\, (a, y, x) \bullet [\phi_3(P)(a, y, x) \ \underline{\text{and}}\ (x \leq 0)])\}
\end{array}
\right.
$$

Solving iteratively starting from the infimum:

$$
\left[
\begin{array}{rcl}
\phi_3^0 & = & \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\, (a, y, x) \bullet [\underline{\text{false}}]\} \\
\phi_{10}^0 & = & \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\, (a, y) \bullet [\underline{\text{false}}]\}
\end{array}
\right.
$$

using Gauss–Seidel's chaotic strategy:

$$\left[\begin{array}{lcl} \phi_3^{k+1} & = & F_1(\phi_3^k, \phi_{10}^k) \\ \phi_{10}^{k+1} & = & F_2(\phi_3^{k+1}, \phi_{10}^k) \end{array}\right.$$

we obtain for the first terms:

$$\left[\begin{array}{rcl} \phi_3^1 & = & \boldsymbol{\lambda} P \bullet \{\boldsymbol{\lambda}(a,y,x) \bullet [P(a)\ \underline{\text{and}}\ (x=a)\ \underline{\text{and}}\ (y=1)]\} \\ \phi_{10}^1 & = & \boldsymbol{\lambda} P \bullet \{\boldsymbol{\lambda}(a,y) \bullet [P(a)\ \underline{\text{and}}\ (a \le 0)\ \underline{\text{and}}\ (y=1)]\} \end{array}\right.$$

$$\left[\begin{array}{rcl} \phi_3^2 & = & \boldsymbol{\lambda} P \bullet \{\boldsymbol{\lambda}(a,y,x) \bullet ([P(a)\ \underline{\text{and}}\ (x=a)\ \underline{\text{and}}\ (y=1)]\ \underline{\text{or}}\ [P(a)\ \underline{\text{and}}\ (x= \\ & & 0)\ \underline{\text{and}}\ (a=1)\ \underline{\text{and}}\ (y=2)])\} \\ \phi_{10}^2 & = & \boldsymbol{\lambda} P \bullet \{\boldsymbol{\lambda}(a,y) \bullet ([P(a)\ \underline{\text{and}}\ (a \le 0)\ \underline{\text{and}}\ (y=1)]\ \underline{\text{or}}\ [P(a)\ \underline{\text{and}}\ (a= \\ & & 1)\ \underline{\text{and}}\ (y=2)])\} \end{array}\right.$$

$$\left[\begin{array}{rcl} \phi_3^3 & = & \boldsymbol{\lambda} P \bullet \{\boldsymbol{\lambda}(a,y,x) \bullet ([P(a)\ \underline{\text{and}}\ (x=a)\ \underline{\text{and}}\ (y=1)]\ \underline{\text{or}}\ [P(a)\ \underline{\text{and}}\ (x= \\ & & 0)\ \underline{\text{and}}\ (a=1)\ \underline{\text{and}}\ (y=2)]\ \underline{\text{or}}\ [P(a)\ \underline{\text{and}}\ (x=1)\ \underline{\text{and}}\ (a=2)\ \underline{\text{and}}\ (y= \\ & & 3)])\} \\ \phi_{10}^3 & = & \boldsymbol{\lambda} P \bullet \{\boldsymbol{\lambda}(a,y) \bullet ([P(a)\ \underline{\text{and}}\ (a \le 0)\ \underline{\text{and}}\ (y=1)]\ \underline{\text{or}}\ [P(a)\ \underline{\text{and}}\ (a= \\ & & 1)\ \underline{\text{and}}\ (y=2)])\} \end{array}\right.$$

$$\left[\begin{array}{rcl} \phi_3^4 & = & \boldsymbol{\lambda} P \bullet \{\boldsymbol{\lambda}(a,y,x) \bullet ([P(a)\ \underline{\text{and}}\ (x=a)\ \underline{\text{and}}\ (y=1)]\ \underline{\text{or}}\ [P(a)\ \underline{\text{and}}\ (x= \\ & & 0)\ \underline{\text{and}}\ (a=1)\ \underline{\text{and}}\ (y=2)]\ \underline{\text{or}}\ [P(a)\ \underline{\text{and}}\ (x=0)\ \underline{\text{and}}\ (a=2)\ \underline{\text{and}}\ (y= \\ & & 4)]\ \underline{\text{or}}\ [P(a)\ \underline{\text{and}}\ (x=1)\ \underline{\text{and}}\ (a=2)\ \underline{\text{and}}\ (y=3)])\} \\ \phi_{10}^4 & = & \boldsymbol{\lambda} P \bullet \{\boldsymbol{\lambda}(a,y) \bullet ([P(a)\ \underline{\text{and}}\ (a \le 0)\ \underline{\text{and}}\ (y=1)]\ \underline{\text{or}}\ [P(a)\ \underline{\text{and}}\ (a= \\ & & 1)\ \underline{\text{and}}\ (y=2)]\ \underline{\text{or}}\ [P(a)\ \underline{\text{and}}\ (a=2)\ \underline{\text{and}}\ (y=4)])\} \end{array}\right.$$

$$\left[\begin{array}{rcl} \phi_3^5 & = & \boldsymbol{\lambda} P \bullet \{\boldsymbol{\lambda}(a,y,x) \bullet ([P(a)\ \underline{\text{and}}\ (x=a)\ \underline{\text{and}}\ (y=1)]\ \underline{\text{or}}\ [P(a)\ \underline{\text{and}}\ (x= \\ & & 0)\ \underline{\text{and}}\ (a=1)\ \underline{\text{and}}\ (y=2)]\ \underline{\text{or}}\ [P(a)\ \underline{\text{and}}\ (x=0)\ \underline{\text{and}}\ (a=2)\ \underline{\text{and}}\ (y= \\ & & 4)]\ \underline{\text{or}}\ [P(a)\ \underline{\text{and}}\ (x=1)\ \underline{\text{and}}\ (a=2)\ \underline{\text{and}}\ (y=3)]\ \underline{\text{or}}\ [P(a)\ \underline{\text{and}}\ (x= \\ & & 2)\ \underline{\text{and}}\ (a=3)\ \underline{\text{and}}\ (y=5)])\} \\ \phi_{10}^5 & = & \boldsymbol{\lambda} P \bullet \{\boldsymbol{\lambda}(a,y) \bullet ([P(a)\ \underline{\text{and}}\ (a \le 0)\ \underline{\text{and}}\ (y=1)]\ \underline{\text{or}}\ [P(a)\ \underline{\text{and}}\ (a= \\ & & 1)\ \underline{\text{and}}\ (y=2)]\ \underline{\text{or}}\ [P(a)\ \underline{\text{and}}\ (a=2)\ \underline{\text{and}}\ (y=4)])\} \end{array}\right.$$

With a little intuition, the computation of these first terms allows us to foresee the following form for the general term:

$$\phi_3^k = \boldsymbol{\lambda} P \bullet \{\boldsymbol{\lambda}(a,y,x) \bullet ([P(a) \ \underline{\text{and}} \ (x = a) \ \underline{\text{and}} \ (y = 1)] \ \underline{\text{or}} \ [P(a) \ \underline{\text{and}} \ (0 \leq x < a) \ \underline{\text{and}} \ (y = 1 + \sum_{i=x}^{a-1} 2^i) \ \underline{\text{and}} \ ((a * (a+1))/2 - x < k)])\}$$

We deduce:

$$\phi_{10}^k = \boldsymbol{\lambda} P \bullet \{\boldsymbol{\lambda}(a,y) \bullet ([P(a) \ \underline{\text{and}} \ (a \leq 0) \ \underline{\text{and}} \ (y = 1)] \ \underline{\text{or}} \ [P(a) \ \underline{\text{and}} \ (0 < a) \ \underline{\text{and}} \ (y = 2^a) \ \underline{\text{and}} \ ((a * (a+1))/2 < k)])\}$$

It is easy to check that the first terms are of the above form. For the induction step, we must prove that $\phi_3^{k+1} = F_1(\phi_3^k, \phi_{10}^k)$. The computation being complex, we will settle for a simple check that $F_1(\phi_3^k, \phi_{10}^k) \Rightarrow \phi_3^{k+1}$. As it is obvious that $\phi_3^k \Rightarrow \phi_3^{k+1}$, we will have $(\phi_3^k \ \underline{\text{or}} \ F_1(\phi_3^k, \phi_{10}^k)) \Rightarrow \phi_3^{k+1}$, which shows that ϕ_3^k is the general term of an increasing over-approximated iteration sequence (4.1.1.0.1).

$$\begin{aligned}
&\phi_{10}^k(\sigma_3^3(\boldsymbol{\lambda}(a,y,x) \bullet [\phi_3^k(P)(a,y,x+1) \ \underline{\text{and}} \ (x \geq 0)]))(x, y-m) \\
&\quad \Rightarrow \ \phi_{10}^k(\boldsymbol{\lambda} x \bullet x \geq 0)(x, y-m) \\
&\quad \Rightarrow \ [(x = 0) \ \underline{\text{and}} \ (y - m = 1)] \ \underline{\text{or}} \ [(0 < x) \ \underline{\text{and}} \ (y - m = 2^x) \ \underline{\text{and}} \ ((x * (x+1))/2 < k)]
\end{aligned}$$

$$\begin{aligned}
&\phi_3^k(P)(a,m,x+1) \ \underline{\text{and}} \ (x \geq 0) \\
&\quad = \ [P(a) \ \underline{\text{and}} \ (x+1 = a) \ \underline{\text{and}} \ (m = 1) \ \underline{\text{and}} \ (x \geq 0)] \ \underline{\text{or}} \ [P(a) \ \underline{\text{and}} \ (0 \leq x < a - 1) \ \underline{\text{and}} \ (m = 1 + \sum_{i=x+1}^{a-1} 2^i) \ \underline{\text{and}} \ (((a * (a+1))/2 - x) < k+1)]
\end{aligned}$$

so

$$F_1(\phi_3^k, \phi_{10}^k)$$

$$\begin{array}{ll}\Rightarrow & \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\,(a,y,x) \bullet ([P(a)\ \underline{\text{and}}\ (x = a)\ \underline{\text{and}}\ (y = 1)]\ \underline{\text{or}}\ [P(a)\ \underline{\text{and}}\ (x = 0)\ \underline{\text{and}}\ (a = 1)\ \underline{\text{and}}\ (y = 2)]\ \underline{\text{or}}\ [P(a)\ \underline{\text{and}}\ (x = 0 < a - 1)\ \underline{\text{and}}\ (y = 1 + \sum_{i=x}^{a-1} 2^i)\ \underline{\text{and}}\ (((a * (a+1))/2 - x) < k+1)]\ \underline{\text{or}}\ [P(a)\ \underline{\text{and}}\ (0 < x = a - 1)\ \underline{\text{and}}\ (y = 1 + 2^{a-1})\ \underline{\text{and}}\ ((a * (a-1))/2 < k)]\ \underline{\text{or}}\ [P(a)\ \underline{\text{and}}\ (0 < x < a - 1)\ \underline{\text{and}}\ (y = 1 + \sum_{i=x}^{a-1} 2^i)\ \underline{\text{and}}\ (((a * (a+1))/2 - x) < k+1)])\} \\ \Rightarrow & \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\,(a,y,x) \bullet ([P(a)\ \underline{\text{and}}\ (x = a)\ \underline{\text{and}}\ (y = 1)]\ \underline{\text{or}}\ [P(a)\ \underline{\text{and}}\ (0 \leq x < a)\ \underline{\text{and}}\ (y = 1 + \sum_{i=x}^{a-1} 2^i)\ \underline{\text{and}}\ (((a * (a+1))/2 - x) < k+1)])\} \\ \Rightarrow & \phi_3^{k+1}\end{array}$$

The term at rank ω of the chaotic over-approximated increasing iteration sequence is thus $\phi_3^\omega = \bigsqcup_{k<\omega} \phi_3^k$, that is:

$$\left[\begin{array}{lcl}\phi_3^\omega & = & \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\,(a,y,x) \bullet [P(a)\ \underline{\text{and}}\ (((x = a)\ \underline{\text{and}}\ (y = 1))\ \underline{\text{or}}\ ((0 \leq x < a)\ \underline{\text{and}}\ (y = 1 + \sum_{i=x}^{a-1} 2^i)))]\}\end{array}\right.$$

we deduce:

$$\left[\begin{array}{lcl}\phi_{10}^\omega & = & \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\,(a,y) \bullet [P(a)\ \underline{\text{and}}\ (((a \leq 0)\ \underline{\text{and}}\ (y = 1))\ \underline{\text{or}}\ ((a > 0)\ \underline{\text{and}}\ (y = 1 + \sum_{i=0}^{a-1} 2^i)))]\} \\ & = & \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\,(a,y) \bullet [P(a)\ \underline{\text{and}}\ (((a < 0)\ \underline{\text{and}}\ (y = 1))\ \underline{\text{or}}\ ((a \geq 0)\ \underline{\text{and}}\ (y = 2^a)))]\}\end{array}\right.$$

Shortening the system of equations into $\phi = F(\phi)$, we get:

$$\left[\begin{array}{lllllllll}\phi^\omega & = & \underset{k<\omega}{\underline{\text{OR}}}\, \phi^k & = & \underset{k<\omega}{\underline{\text{OR}}}\, \phi^{k+1} & \Leftarrow & \underset{k<\omega}{\underline{\text{OR}}}\, F(\phi^k) & = & F(\underset{k<\omega}{\underline{\text{OR}}}\, \phi^k) \quad = \quad F(\phi^\omega)\end{array}\right.$$

because we showed that $(\forall k < \omega, F(\phi^k) \Rightarrow \phi^{k+1})$ and F is a complete join-morphism. Since ϕ^ω is a post-fixpoint of F, it is the limit of the iteration (4.1.1.0.1.(c)) and

Theorem 4.1.1.0.2 shows that $\mathit{lfp}(F) \Rightarrow \phi^{\omega}$. We deduce that procedure f is partially correct: if x is a positive of null integer and $f(x, y)$ terminates, then $y = 2^x$. The proof of termination is obvious by finite induction.
End of example.

6.2 CONSTRUCTIVE METHODS TO APPROXIMATE SOLUTIONS OF A SYSTEM OF FUNCTIONAL EQUATIONS

The techniques developed in Chapter 4 are directly applicable to approach the least fixpoint of the system of forward semantic equations:

$$\phi \quad = \quad F(\phi) \quad \text{where} \quad \phi \in (\prod_{i=1}^{n}((\mathcal{U}^m \to \mathcal{B}) \to (\mathcal{U}^{m_i} \to \mathcal{B})))$$

associated with a recursive procedure (with m input parameters with value in $\mathcal{U}$ and with n program points). For example, for an over-approximation, we will define an over-approximated image L_k of $(\mathcal{U}^k \to \mathcal{B})$ for all $k \geq 0$ (Definition 4.2.7.0.6):

$$L_k \quad \bar{\triangleright}(@_k, \gamma_k) \quad (\mathcal{U}^k \to \mathcal{B})$$

so that Paragraph 4.2.8 indicates how to build a system of approximate equations by defining (thanks to Theorem 4.2.8.0.3):

$$(L_m \to L_{m_i}) \quad \bar{\triangleright}(@_{m_i,m}, \gamma_{m_i,m}) \quad ((\mathcal{U}^m \to \mathcal{B}) \to (\mathcal{U}^{m_i} \to \mathcal{B}))$$

with

$$@_{m_i,m} \quad = \quad \boldsymbol{\lambda}\phi \bullet (@_{m_i} \circ \phi \circ \gamma_m)$$

$$\gamma_{m_i,m} \quad = \quad \boldsymbol{\lambda}\phi \bullet (\gamma_{m_i} \circ \phi \circ @_m)$$

and

$$\prod_{i=1}^{n}(L_m \to L_{m_i}) \quad \bar{\triangleright}(\bar{@}_n, \bar{\gamma}_n) \quad (\prod_{i=1}^{n}((\mathcal{U}^m \to \mathcal{B}) \to (\mathcal{U}^{m_i} \to \mathcal{B})))$$

with

$$\begin{array}{rcl} \bar{@}_n & = & \boldsymbol{\lambda}(\phi_1, \ldots, \phi_n) \bullet (@_{m_1,m}(\phi_1), \ldots, @_{m_n,m}(\phi_n)) \\ \bar{\gamma}_n & = & \boldsymbol{\lambda}(\phi_1, \ldots, \phi_n) \bullet (\gamma_{m_1,m}(\phi_1), \ldots, \gamma_{m_n,m}(\phi_n)) \end{array}$$

which gives a system of approximate equations of the form:

$$\phi \quad = \quad F(\phi) \quad \text{where} \quad \phi \in (\prod_{i=1}^{n}(L_m \to L_{m_i}))$$

where the rules of construction for $\bar{F}$ ensure that:

$$\bar{\bar{@}}_n(F) \quad \sqsubseteq \quad \bar{F}$$

with

$$\begin{array}{rcl} \bar{\bar{@}}_n & = & \boldsymbol{\lambda} F \bullet \bar{@}_n \circ F \circ \bar{\gamma}_n \\ \bar{\bar{\gamma}}_n & = & \boldsymbol{\lambda} F \bullet \bar{\gamma}_n \circ F \circ \bar{@}_n \end{array}$$

After computing $\mathit{lfp}(\bar{F})$, using Theorem 2.9.1.0.2, or an over-approximation $\bar{\phi}$ of $\mathit{lfp}(\bar{F})$, using the fixpoint approximation techniques based on convergence acceleration by extrapolation, as in Paragraph 4.1, Theorem 4.2.8.0.4 ensures that:

$$\mathit{lfp}(F) \quad \Rightarrow \quad \bar{\gamma}_n(\bar{\phi})$$

For example, let us perform an approximate analysis of the sign of the function:

$$f(x) \quad = \quad \underline{\text{if}}\ x \geq 1000\ \ \underline{\text{then}}\ \ x\ \ \underline{\text{else}}\ \ -f(-2 * x)\ \underline{\text{endif}}\ ;$$

using the approximation defined in Paragraph 5.3. The function which associates the sign of $f(x)$ with the sign of x is the least fixpoint of the function:

$$\left\{\ F \quad = \quad \boldsymbol{\lambda}\,\phi \centerdot \{\boldsymbol{\lambda}\,P \centerdot [(P \sqcap \dot{+}) \sqcup -\phi(-P)]\}\right.$$

By iterative solving, we obtain:

$$\left[\ \phi^0 \quad = \quad \boldsymbol{\lambda}\,P \centerdot \bot\right.$$

$$\left[\ \phi^1 \quad = \quad F(\phi^0) \quad = \quad \boldsymbol{\lambda}\,P \centerdot \{(P \sqcap \dot{+}) \sqcup -\bot\} \quad = \quad \boldsymbol{\lambda}\,P \centerdot (P \sqcap \dot{+})\right.$$

$$\left[\begin{aligned} \phi^2 \quad = \quad F(\phi^1) \quad &= \quad \boldsymbol{\lambda}\,P \centerdot \{(P \sqcap \dot{+}) \sqcup -((-P) \sqcap \dot{+})\} \\ &= \quad \boldsymbol{\lambda}\,P \centerdot \{(P \sqcap \dot{+}) \sqcup (P \sqcap \dot{-})\} \\ & \qquad \text{as } (-P) \sqcap \dot{+} = -(P \sqcap \dot{-}) \text{ and } - - P = P \\ &= \quad \boldsymbol{\lambda}\,P \centerdot \{P \sqcap (\dot{+} \sqcup \dot{-})\} \\ &= \quad \boldsymbol{\lambda}\,P \centerdot \{P \sqcap \top\} \\ &= \quad \boldsymbol{\lambda}\,P \centerdot P \end{aligned}\right.$$

$$\left[\begin{aligned} \phi^3 \quad = \quad F(\phi^2) \quad &= \quad \boldsymbol{\lambda}\,P \centerdot \{(P \sqcap \dot{+}) \sqcup - - P\} \\ &= \quad \boldsymbol{\lambda}\,P \centerdot P \end{aligned}\right.$$

We just showed that the sign of $f(x)$ is the same as the sign of x. Our reasoning is exactly the same as in Chapter 5, except that the unknowns ϕ_i of the system of equations $\phi = \bar{F}(\phi)$ are functions $\phi_i \in (L_m \rightarrow L_{m_i})$ whereas, in Chapter 5, we considered a system of equations $X = \bar{F}(X)$ where $X_i \in L_k$. Thus, in the above example, the computation was performed on functions represented using symbolic lambda-expressions and not on elements of the lattice $L_1 = \{\bot, 0, \dot{+}, \dot{-}, \top\}$. This representation of functions is not practical in a computer because symbolic computations are hard to automate and a representation of the elements of $L_1 \rightarrow L_1$ by a table with five entries will be more convenient. However, in the general case, the cardinal of L_1 is high or infinite and no finite representation of the elements of $L_1 \rightarrow L_1$ is convenient for a computer.

However, in practice, it is not necessary to know the function $\mathit{lfp}(F)$ but, much more simply, the predicate $\mathit{lfp}(F)(P)$ for a number of entry conditions P. It means that we will only compute the over-approximation $\bar{\phi} \in (L_1 \rightarrow L_1)$ of $\bar{@}_1(\mathit{lfp}(F))$ for a finite number of arguments, which will allow the representation of $\bar{\phi}$ by a table. Among possible ideas, we can consider approximating $\phi(x)$ by $\phi(\top)$ for all x of L_1, because $x \sqsubseteq \top$ implies by monotonicity that $\phi(x) \sqsubseteq \phi(\top)$. Better still, we can compute $\bar{\phi}(P)$ where P is given by the various call contexts. Rather than analyzing the procedure at declaration time, this amounts to performing the analysis for each call. For example, to compute $\mathit{lfp}(\bar{F})(\dot{+})$, we will get:

$$\begin{array}{lclclclclcl}
\phi^0(P) & = & \bot \quad \forall P = \{\bot, 0, \dot{+}, \dot{-}, \top\} \\
\phi^1(\dot{+}) & = & (\dot{+} \sqcap \dot{+}) \sqcup -\phi^0(-\dot{+}) & = & \dot{+} \sqcup -\phi^0(\dot{-}) & = & \dot{+} \sqcup \bot & = & \dot{+} \\
\phi^1(\dot{-}) & = & (\dot{-} \sqcap \dot{+}) \sqcup -\phi^1(-\dot{-}) & = & 0 \sqcup -\phi^1(\dot{+}) & = & 0 \sqcup -\dot{+} & = & 0 \sqcup \dot{-} & = & \dot{-} \\
\phi^2(\dot{+}) & = & (\dot{+} \sqcap \dot{+}) \sqcup -\phi^1(-\dot{+}) & = & \dot{+} \sqcup -\phi^1(\dot{-}) & = & \dot{+} \sqcup -\dot{-} & = & \dot{+} \sqcup \dot{+} & = & \dot{+} \\
\phi^2(\dot{-}) & = & (\dot{-} \sqcap \dot{+}) \sqcup -\phi^2(-\dot{-}) & = & 0 \sqcup -\phi^2(\dot{+}) & = & 0 \sqcup -\dot{+} & = & 0 \sqcup \dot{-} & = & \dot{-}
\end{array}$$

The computation of $\mathit{lfp}(\bar{F})(\dot{+})$ does not need the complete knowledge of the function $\mathit{lfp}(\bar{F})$, but only of $\mathit{lfp}(\bar{F})(\dot{-})$ and $\mathit{lfp}(\bar{F})(\dot{+})$. Now, we formalize the ideas presented above (6.2.1) and then we solve (6.2.2) the problems arising when considering spaces of infinite cardinal, drawing our inspirations from 4.1.

6.2.1 Resolution of a system of functional fixpoint equations in a finite space by chaotic iteration

DEFINITION 6.2.1.0.1 *Finite chaotic iteration for a system of functional fixpoint equations*

For all $i = 1, \ldots, n$, let $D_i, D'_i, L = \prod_{i=1}^{n}(D_i \rightarrow D'_i)$ be complete lattices. Let $F \in \mathit{ucont}(L \rightarrow L)$. We consider the system of functional equations $\phi = F(\phi)$ which can be expressed as:

$$\left\{\begin{array}{rcl} \phi_i & = & F_i(\phi) \quad \text{where } F_i \in (L \rightarrow (D_i \rightarrow D'_i)) \\ & = & \boldsymbol{\lambda}\,(\psi_1, \ldots, \psi_n) \bullet \{\boldsymbol{\lambda}\,P \bullet [f_i(P, \psi_1, \ldots, \psi_n)]\}(\phi_1, \ldots, \phi_n) \\ i & = & 1, \ldots, n \end{array}\right.$$

- An *index* is a vector $(J_1, \ldots, J_n)$ such that $\{\forall i \in [1,n], J_i \subseteq D_i\}$
- Given an index $J \subseteq \prod_{i=1}^{n} D_i$, we define F_j by:

$$\forall \phi \in L, F_j(\phi) = \psi$$

where

$$\forall i \in [1,n], \psi_i = \boldsymbol{\lambda}\,X \bullet \underline{\text{if}}\ X \in J_i\ \underline{\text{then}}\ F_i(\phi)(X)\ \underline{\text{else}}\ \phi_i(X)\ \underline{\text{endif}}\ ;$$

- Let $\phi \in L$, $Y \in D_j$. We will say that the *evaluation of* $F_j(\phi)(Y)$ *requires the evaluation of* $\phi_i(X)$, and write it $F_j(\phi)(Y) \hookrightarrow \phi_i(X)$, if and only if F_j can be written in the form $\boldsymbol{\lambda}\,\psi \bullet \{\boldsymbol{\lambda}\,P \bullet [\ldots \psi_i(f(P, \psi)) \ldots]\}$ and $f(Y, \phi) = X$.
- Let $V \subseteq \prod_{i=1}^{n} D_i$, we will denote by $\text{F}-\mathit{closure}(\phi, V)$ the least vector (for set inclusion $\subseteq$) $\bar{V} \subseteq \prod_{i=1}^{n} D_i$ such that:

$$\{\forall i \in [1,n], \bar{V}_i = V_i \cup \{X \in D_i : \{\exists k \in [1,n], \exists Y \in \bar{V}_k : F_k(\phi)(Y) \longleftrightarrow \phi_i(X)\}\}\}$$

- *A finite chaotic iteration sequence starting from* $\phi^0 \in L$, *for* $V \subseteq \prod_{i=1}^{n} D_i$, *and defined by* F *and the sequence of eligible indexes* $J^0, J^1, \ldots, J^k, \ldots$ is a sequence $\phi^0, \phi^1, \ldots, \phi^k, \ldots$ of elements of L such that:
  - $\{\forall k \geq 1, \phi^k = F_{J^{k-1}}(\phi^{k-1})\}$
  - $\{(\forall i \in [1,n]), (\forall X \in V_i), (\forall k \geq 0), (\exists l \geq 1) : (X \in J_i^{k+l})\ \underline{\text{and}}\ (\forall j \in [1,n], \bar{W}_j \subseteq \bigcup_{p=0}^{l-1} J_j^{k+p})$

  where $(W_i = \{X\}), (\forall j \in [1,n] : (j \neq i), (W_j = \emptyset)), (\bar{W} = \text{F}-\mathit{closure}(\phi^{k+1}, W))\}$

LEMMA 6.2.1.0.2

A finite chaotic iteration sequence defined by the functional F and starting from $\phi^0 \in \mathit{prefp}(F)$ is an ascending chain:

$$\{\forall k \geq 0, \phi^k \sqsubseteq \phi^{k+1} \sqsubseteq F(\phi^k) \sqsubseteq \mathit{luis}(F)(\phi^0)\}$$

Proof: As $\phi^0 \sqsubseteq F(\phi^0) \sqsubseteq \mathit{luis}(F)(\phi^0)$, for all $i = 1, \ldots, n$ and $X \in D_i$, we have $\phi^0_i(X) \sqsubseteq F_i(\phi^0)(X) \sqsubseteq (\mathit{luis}(F)(\phi^0))_i(X)$. If $X \in J^0_1$, then $\phi^0_i(X) \sqsubseteq \phi^1_i(X) = F_i(\phi^0)(X)$, otherwise $X \notin J^0_1$, in which case $\phi^0_i(X) = \phi^1_i(X) \sqsubseteq F_i(\phi^0)(X)$.

For the induction step, we suppose that, for $k > 0$, we have $\phi^{k-1} \sqsubseteq \phi^k \sqsubseteq F(\phi^{k-1}) \sqsubseteq \mathit{luis}(F)(\phi^0)$. For all $i = 1, \ldots, n$ and all $X \in D_i$, we have either $X \in J^{k-1}_i$, in which case $\phi^k_i(X) = F_i(\phi^{k-1})(X) \sqsubseteq F_i(\phi^k)(X) \sqsubseteq (\mathit{luis}(F)(\phi^0))_i(X)$ because F_i is monotone, or $X \notin J^{k-1}$ and, in that case, $\phi^k_i(X) = \phi^{k-1}_i(X) \sqsubseteq F_i(\phi^{k-1}_i)(X) \sqsubseteq F_i(\phi^k)(X) \sqsubseteq (\mathit{luis}(F)(\phi^0))_i(X)$, which proves that $\phi^k_i \sqsubseteq F_i(\phi^k) \sqsubseteq (\mathit{luis}(F)(\phi^0))_i$. Now, for all $i = 1, \ldots, n$ and all X in D_i, we have either $X \in J^k_i$ and $\phi^k_i(X) \sqsubseteq F_i(\phi^k)(X) = \phi^{k+1}_i(X)$, or $X \notin J^k_i$ and $\phi^k_i(X) = \phi^{k+1}_i(X) \sqsubseteq F_i(\phi^k)(X)$, which proves that $\phi^k \sqsubseteq \phi^{k+1} \sqsubseteq F(\phi^k) \sqsubseteq \mathit{luis}(F)(\phi^0)$. By finite induction on k, the lemma is proved.

End of proof.

THEOREM 6.2.1.0.3

Let $\phi^0 \in \mathit{prefp}(F)$ be such that $\phi^0 \sqsubseteq \mathit{lfp}(F)$, then a finite chaotic iteration sequence defined for the functional F starting from ϕ^0 for $V \subseteq \prod_{i=1}^{n} D_i$ and stable after ε steps is such that:

$$\{(\forall i \in [1, n]), (\forall X \in V_i), \phi^\varepsilon_i(X) = (\mathit{lfp}(F))_i(X)\}$$

Proof: Let $W^\varepsilon = \text{F}-\mathit{closure}(\phi^\varepsilon, V)$. Let $i \in [1, n]$ and $X \in V_i$. By definition of eligible sequences of indexes, $\exists l \geq 1$ such that $X \in J^{\varepsilon+l}_i$, and so, $\phi^{\varepsilon+l+1}_i(X) = F_i(\phi^{\varepsilon+l})(X)$

and, as $\phi_i^{\varepsilon+l+1} = \phi_i^{\varepsilon+l} = \phi_i^{\varepsilon}$, we have $\phi_i^{\varepsilon} = F_i(\phi_i^{\varepsilon})$. Now, let $X \in (W_i^{\varepsilon} - V_i)$, then $\exists k \in [1,n], \exists Y \in V_k$ such that $F_k(\phi^{\varepsilon})(Y) \bullet\!\!\longrightarrow \phi_j^{\varepsilon}(Z)$ and $F_j(\phi^{\varepsilon})(Z) \bullet\!\!\longrightarrow \ldots \bullet\!\!\longrightarrow \phi_i^{\varepsilon}(X)$, which implies $\exists l \geq 1$ such that $Y \in J_k^{\varepsilon+l}$ and $X \in \mathrm{F}{-}\mathit{closure}(\phi^{\varepsilon+1}, (\emptyset, \emptyset, \ldots, \{Y\}, \ldots, \emptyset))_i$, where $\{Y\}$ is in k^{th} position. Therefore, $\exists p \in [0, l-1]$ such that $X \in J_i^{\varepsilon+p}$, in which case, $\phi_i^{\varepsilon+p+1}(X) = F_i(\phi^{\varepsilon+p})(X)$, and so, $\phi_i^{\varepsilon}(X) = F_i(\phi^{\varepsilon})(X)$. As $V_i \subseteq W_i^{\varepsilon}$, we have shown that $\{\forall X \in W_i^{\varepsilon}, \phi_i^{\varepsilon}(X) = F_i(\phi^{\varepsilon})(X)\}$.

Let $\psi^0 \in L$ such that, for all $i = 1, \ldots, n$ and all $X \in D_i$, we have $\psi_i^0 = \underline{\text{if}}\ X \in W_i\ \underline{\text{then}}\ \phi_i^{\varepsilon}(X)\ \underline{\text{else}}\ \bot\ \underline{\text{endif}}$. Then, $\psi^0 \in \mathit{prefp}(F)$ and, following 6.2.1.0.2, $\psi^0 \sqsubseteq \mathit{luis}(F)(\phi^0) = \mathit{lfp}(F)$ and, following Theorem 2.7.0.1, we have $\mathit{lfp}(F) = \bigsqcup_{k \in \omega} \psi^k$, where $\psi^k = F(\psi^{k-1})$. Let us prove that, for all $k \in \omega$, we have $\mathrm{F}{-}\mathit{closure}(\psi^0, W^{\varepsilon}) = \mathrm{F}{-}\mathit{closure}(\psi^k, W^{\varepsilon}) = W^{\varepsilon}$ and $\forall i \in [1,n], \forall X \in W_i^{\varepsilon}, \psi_i^k(X) = \psi_i^0(X)$. For $k = 0$, we have $\mathrm{F}{-}\mathit{closure}(\psi^0, W^{\varepsilon}) = \mathrm{F}{-}\mathit{closure}(\phi^0, W^{\varepsilon}) = \mathrm{F}{-}\mathit{closure}(\phi^0, V) = W^{\varepsilon}$. Suppose that the lemma is true up to k. We have $\psi_i^{k+1}(X) = F_i(\psi^k)(X)$. For all $Z \in D_j$ such that $F_i(\psi^k)(X) \bullet\!\!\longrightarrow \psi_j^k(Z)$, we know by induction hypothesis that $Z \in W_j^{\varepsilon}$ and, thus, $\psi_i^k(Z) = \psi_i^0(Z)$, which implies $\psi_j^k(Z) = \psi_j^0(Z)$ and $\psi_i^{k+1}(X) = F_i(\psi^k)(X) = F_i(\psi^0)(X) = \psi_i^0(X)$. Moreover, $\mathrm{F}{-}\mathit{closure}(\psi^{k+1}, W^{\varepsilon}) = W^{\varepsilon}$ because, for all $Y \in W_j^{\varepsilon}, F_j(\psi^{k+1})(Y) \bullet\!\!\longrightarrow \phi_i(X)$ implies that $F_j(\psi^0)(Y) \bullet\!\!\longrightarrow \phi_i(X)$, and so, $X \in \mathrm{F}{-}\mathit{closure}(\psi^0, W^{\varepsilon}) = W^{\varepsilon}$. By induction on k, we get $\forall X \in W_i^{\varepsilon}, \psi_i^k(X) = \phi_i^{\varepsilon}(X)$, and so, $(\mathit{lfp}(F))_i(X) = \bigsqcup_{k \in \omega} \psi_i^k(X) = \phi_i^{\varepsilon}(X)$.

End of proof.

In the case where the $D_i', i = 1, \ldots, n$ are lattices satisfying the ascending chain condition, then the chaotic iteration sequence is stationary but it is possible that, to satisfy Definition 6.2.1.0.1, any eligible sequence of indexes must contain an index with an infinite component. For example, it is the case of the equation $\phi = \boldsymbol{\lambda}\, \psi \bullet \{\boldsymbol{\lambda}\, X \bullet [\psi(X+1)]\}(\phi)$ where $\phi \in (L \to L)$, with $L = \mathcal{Z} \cup \{\bot, \top\}$ ordered by $\bot \sqsubseteq \bot \sqsubseteq X \sqsubseteq X \sqsubseteq \top \sqsubseteq \top$ for all $X \in \mathcal{Z}$, and $\bot + 1 = \bot$ and $\top + 1 = \top$. So, in practice, Definition 6.2.1.0.1 is applicable only to indexes of finite components, which is the case, for example, if the $D_1, \ldots, D_n$ are finite lattices.

Now, we tackle the general case by leveraging the algorithms for fixpoint approximation based on convergence acceleration by extrapolation from Paragraph 4.1 that we adapt to the case of functionals.

6.2.2 Increasing chaotic iteration sequence with upper widening to approximate the solution of a system of functional equations

<u>DEFINITION</u> 6.2.2.0.1 *Increasing chaotic iteration sequence with upper widening for a system of functional fixpoint equations*

For $i = 1, \ldots, n$, let $D_i, D'_i, L = \prod_{i=1}^{n}(D_i \to D'_i)$ be complete lattices and $F \in mon(L \to L)$. Let $\tilde{F} \in (L \to L)$ be such that $F \sqsubseteq \tilde{F}$ and, for $i = 1, \ldots, n$, the upper widenings $\bar{\nabla}_i \in (D'_i \times D'_i \to D'_i)$ satisfying Definition 4.1.2.0.4.

- Given an index $J \subseteq \prod_{i=1}^{n} D_i$, we define $\tilde{F}_J$ as $\forall \phi \in L, \tilde{F}_J(\phi) = \psi$ with $\forall i \in [1,n], \psi_i = \boldsymbol{\lambda} X \bullet \underline{\text{if}}\ X \in J_i\ \underline{\text{then}}\ \phi_i(X)\ \bar{\nabla}_i\ \tilde{F}_i(\phi)(X)\ \underline{\text{else}}\ \phi_i(X)\ \underline{\text{endif}}$.

- An *increasing chaotic iteration sequence starting from $\phi^0 \in L$, for $V \subseteq \prod_{i=1}^{n} D_i$, and defined by $\tilde{F}$ and the eligible sequence of indexes $J^0, J^1, \ldots, J^k, \ldots$* is a sequence $\phi^0, \phi^1, \ldots, \phi^k, \ldots$ of elements in L such that:

  - $\{\forall k \geq 1, \phi^k = \tilde{F}_{J^{k-1}}(\phi^{k-1})\}$
  - $\{(\forall i \in [1,n]), (\forall X \in V_i), (\forall k \geq 0), (\exists l \geq 1) : (X \in J_i^{k+l})\ \underline{\text{and}}\ (\forall j \in [1,n], \bar{W}_j \subseteq \bigcup_{p=0}^{l-1} J_j^{k+p})$

    where $(W_i = \{X\}), (\forall j \in [1,n] : (j \neq i), (W_j = \emptyset)), (\bar{W} = \tilde{\mathrm{F}}\text{–}closure(\phi^{k+1}, W))\}$

<u>THEOREM</u> 6.2.2.0.2

A chaotic iteration sequence starting from $\phi^0 \in L$, for $V \subseteq \prod_{i=1}^{n} D_i$, and defined by F and $\tilde{F}$ is a stationary ascending chain with limit ϕ^ε such that:

$$\{(\forall i \in [1,n]), (\forall X \in V_i), (\mathit{lfp}(F))_i(X) \sqsubseteq \phi_i^\varepsilon(X)\}$$

⌞

Proof: According to 4.1.2.0.4.(a), we have $\forall X \in D_i, \phi_i(X) \sqsubseteq \phi_i(X)\,\bar{\nabla}_i\,\bar{F}_i(\phi)(X)$, which proves that $\tilde{F}_{J_k}$ is extensive for all $k \geq 0$ and, consequently, the sequence $\phi^0, \phi^1, \ldots, \phi^k, \ldots$ is an ascending chain. As X appears an infinite number of times in the sequence $J^0, \ldots, J^k, \ldots$, there exists a sequence $i_0, i_1, \ldots, i_k$ such that $\forall j \in [1,n], \phi_j^{i_k+1}(X) = \phi_j^{i_k}(X)\bar{\nabla}_j C_j$ where $C_j \in D'_j$, which proves, according to 4.1.2.0.4.(b), that the sequence $\phi^0, \ldots, \phi^k, \ldots$ is stationary.

Let $W^\varepsilon = \tilde{\mathrm{F}}\text{–}\mathit{closure}(\phi^\varepsilon, V)$. For all $i = 1, \ldots, n$ and all $X \in V_i$, we have $\exists l \geq 1$ such that $X \in J_i^{\varepsilon+l}$ and, thus, $\phi_i^{\varepsilon+l+1}(X) = \phi_i^{\varepsilon+l}(X)\,\bar{\nabla}_i\,\tilde{F}_i(\phi^{\varepsilon+l})(X)$, and so, $\phi_i^\varepsilon(X) = \phi_i^\varepsilon(X)\,\bar{\nabla}_i\,\tilde{F}_i(\phi^\varepsilon)(X) \sqsupseteq \phi_i^\varepsilon(X) \sqcup \tilde{F}_i(\phi^\varepsilon)(X) \sqsupseteq \phi_i^\varepsilon(X)$ and, thus, $\tilde{F}_i(\phi^\varepsilon)(X) \sqsubseteq \phi_i^\varepsilon(X)$. Suppose now that $X \in (W_i^\varepsilon - V_i)$, then $\exists k \in [1,n], \exists Y \in V_k$ such that $F_k(\phi^\varepsilon)(Y) \bullet\!\!\longrightarrow \phi_j^\varepsilon(Z)$ and $F_j(\phi^\varepsilon)(Z) \bullet\!\!\longrightarrow \ldots \bullet\!\!\longrightarrow \phi_i^\varepsilon(X)$, which implies $\exists l \geq 1 : Y \in J_k^{\varepsilon+l}$ and $X \in \mathrm{F}\text{–}\mathit{closure}(\phi^{\varepsilon+1}, (\emptyset, \emptyset, \ldots, \{Y\}, \ldots, \emptyset))_i$ where $\{Y\}$ is in k^{th} position. Therefore, $\exists p \in [0, l-1]$ such that $X \in J_i^{\varepsilon+p}$ and, thus, $\phi_i^{k+p+1}(X) = \phi_i^{\varepsilon+p}(X)\,\bar{\nabla}_i\,\tilde{F}_i(\phi^\varepsilon)(X)$ which, as previously, implies that $\tilde{F}_i(\phi^\varepsilon)(X) \sqsubseteq \phi_i^\varepsilon(X)$.

Let us define ψ such that $\forall i \in [1,n], \psi_i = \boldsymbol{\lambda} X \bullet \underline{\text{if}}\ X \in W_i^\varepsilon\ \underline{\text{then}}\ \phi_i^\varepsilon(X)\ \underline{\text{else}}\ \top\ \underline{\text{endif}}$; ψ is a post-fixpoint of $\tilde{F}$ and, as $F \sqsubseteq \tilde{F}$, ψ is a post-fixpoint of F, so that Theorem 2.5.5.0.1 implies $\mathit{lfp}(F) \sqsubseteq \psi$, which proves $\forall i \in [1,n], \forall X \in V_i \subseteq W_i^\varepsilon, (\mathit{lfp}(F))_i(X) \sqsubseteq \phi_i(X)$.

End of proof.

6.3 EXAMPLES OF APPROXIMATE FORWARD SEMANTIC ANALYSIS OF RECURSIVE PROCEDURES

This paragraph reviews some of the applications described in Chapter 5 to illustrate the over-approximation of the least solution of a system of forward semantic functional equations associated with a recursive procedure.

6.3.1 Case of a finite space of approximate properties

6.3.1.1 Sign of the variables of a procedure

Example 6.3.1.1.0.1

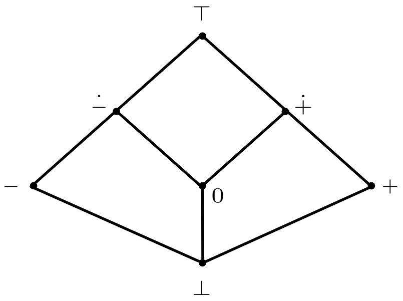
Let us consider the following procedure:

```
       procedure f(x : integer value ; y : integer result) =
{1}
           if x ≥ 1000 then
{2}
               y := x ;
{3}
            else
{4}
               begin z : integer ;
{5}
                   z := −2 ∗ x ;
{6}
                   f(z; y) ;
{7}
                   y := −y ;
{8}
                end ;
{9}
            endif ;
{10}
       end–proc ;
{11}
```

The system of forward semantic equations associated with this procedure is the following:

$$\left\{\begin{array}{lcl}
\phi_1 & = & \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\,(a,y,x) \bullet [P(a) \ \underline{\text{and}}\ (x=a)\ \underline{\text{and}}\ (y=\Omega)]\} \\
\phi_2 & = & \underline{\text{test}}(\boldsymbol{\lambda}\,(a,y,x) \bullet (x \geq 1000)) \circ \phi_1 \\
\phi_3 & = & \underline{\text{assign}}(\boldsymbol{\lambda}\,(a,y,x) \bullet (a,x,x)) \circ \phi_2 \\
\phi_4 & = & \underline{\text{test}}(\boldsymbol{\lambda}\,(a,y,x) \bullet (x < 1000)) \circ \phi_1 \\
\phi_5 & = & \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\,(a,y,x,z) \bullet [P(a,y,x)\ \underline{\text{and}}\ (z=\Omega)]\} \\
\phi_6 & = & \underline{\text{assign}}(\boldsymbol{\lambda}\,(a,y,x,z) \bullet (a,y,x,-2*x)) \circ \phi_5 \\
\phi_7 & = & \boldsymbol{\lambda}\, P \bullet \{\boldsymbol{\lambda}\,(a,y,x,z) \bullet [\phi_{11}(\sigma_4^4(P))(z,y)\ \underline{\text{and}}\ \bar{\sigma}_2^4(P)(a,x,z)]\} \circ \phi_6 \\
\phi_8 & = & \underline{\text{assign}}(\boldsymbol{\lambda}\,(a,y,x,z) \bullet (a,-y,x,z)) \circ \phi_7 \\
\phi_9 & = & \bar{\sigma}_4^4 \circ \phi_8 \\
\phi_{10} & = & \phi_3\ \underline{\text{or}}\ \phi_9 \\
\phi_{11} & = & \sigma_{1,2}^3 \circ \phi_{10}
\end{array}\right.$$

Choosing the closure operator defined in Paragraph 5.2.1, a predicate P over n variables is approximated by a vector of n elements of the lattice:



The corresponding system of approximate equations is the following (we will write $P = (P(1), \ldots, P(n))$ when $P \in L^n$):

$$\begin{aligned}
\phi_1 &\in (L^1 \to L^3)\\
&= \boldsymbol{\lambda} P \bullet (P(1), \bot, P(1))\\
\phi_2 &\in (L^1 \to L^3)\\
&= \boldsymbol{\lambda} P \bullet \{\phi_1(P) \sqcap (\top, \top, +)\}\\
\phi_3 &\in (L^1 \to L^3)\\
&= \boldsymbol{\lambda} P \bullet \{\phi_2(P)(y \leftarrow x)\}\\
\phi_4 &\in (L^1 \to L^3)\\
&= \phi_1\\
\phi_5 &\in (L^1 \to L^4)\\
&= \boldsymbol{\lambda} P \bullet (P(1), P(2), P(3), \bot) \circ \phi_4\\
\phi_6 &\in (L^1 \to L^4)\\
&= \boldsymbol{\lambda} P \bullet \{\phi_5(P)(z \leftarrow -x)\}\\
\phi_7 &\in (L^1 \to L^4)\\
&= \boldsymbol{\lambda} P \bullet \{(\top, \phi_{11}(P(4))(2), \top, \phi_{11}(P(4))(1)) \sqcap (P(1), \top, P(3), P(4))\} \circ \phi_6\\
\phi_8 &\in (L^1 \to L^4)\\
&= \boldsymbol{\lambda} P \bullet \{\phi_7(P)(y \leftarrow -y)\}\\
\phi_9 &\in (L^1 \to L^3)\\
&= \boldsymbol{\lambda} P \bullet (P(1), P(2), P(3)) \circ \phi_8\\
\phi_{10} &\in (L^1 \to L^3)\\
&= \boldsymbol{\lambda} P \bullet \{\phi_3(P) \sqcup \phi_9(P)\}
\end{aligned}$$

$$\begin{aligned} \phi_{11} &\in (L^1 \to L^2) \\ &= \boldsymbol{\lambda}\, P \bullet (P(1), (P(2)) \circ \phi_{10} \end{aligned}$$

This system of equations can be simplified as follows:

$$\begin{aligned} \phi_{11} &\in (L^1 \to L^2) \\ &= \boldsymbol{\lambda}\, P \bullet (P(1), \{(P(1) \sqcap +) \sqcup -\phi_{11}(-P(1))(1)\}) \end{aligned}$$

or, more simply:

$$\left\{ \begin{aligned} \phi_{11} &\in (L \to L) \\ &= \boldsymbol{\lambda}\, x \bullet \{(x \sqcap +) \sqcup -\phi_{11}(-x)\} \end{aligned} \right.$$

We have to compute $\phi_{11}(\dot{+})$ using Definition 6.2.1.0.1. We have:

Step 0 :

$$\phi_{11}^0 = \boldsymbol{\lambda}\, x \bullet \bot$$

Step 1 : $J_0 = (\{\dot{+}\})$

$$\begin{aligned} \phi_{11}^1(\dot{+}) &= (\dot{+} \sqcap +) \sqcup -\phi_{11}^0(-(\dot{+})) \\ &= + \sqcup \bot \\ &= + \end{aligned}$$

Step 2 : $J_1 = (\{\dot{-}\})$

$$\begin{aligned} \phi_{11}^2(\dot{-}) &= (\dot{-} \sqcap +) \sqcup -\phi_{11}^1(-(\dot{-})) \\ &= \bot \sqcup -\phi_{11}^1(\dot{+}) \\ &= - \end{aligned}$$

Step 3 : $J_2 = (\{\dot{+}\})$

$$
\begin{aligned}
\phi_{11}^{3}(\dot{+}) &= (\dot{+} \sqcap +) \sqcup -\phi_{11}^{2}(-(\dot{+})) \\
&= + \sqcup -\phi_{11}^{2}(\dot{-}) \\
&= + \sqcup -(-) \\
&= + \sqcup + \\
&= +
\end{aligned}
$$

Step 4 : $J_3 = (\{\dot{-}\})$

$$
\begin{aligned}
\phi_{11}^{4}(\dot{-}) &= (\dot{-} \sqcap +) \sqcup -\phi_{11}^{3}(-(\dot{-})) \\
&= \perp \sqcup -\phi_{11}^{3}(\dot{+}) \\
&= -
\end{aligned}
$$

The computation converges, and so, $\phi_{11}(\dot{+}) = +$ and $\phi_{11}(\dot{-}) = -$.

End of example.

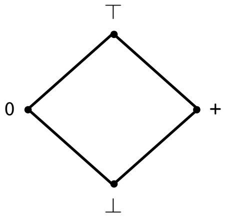
Example 6.3.1.1.0.2

Ackermann's function defined on natural numbers provides a more complex example:

```
$f(x, y)$  =  if $x = 0$ then
                  $y + 1$
              elsif $y = 0$ then
                  $f(x - 1, 1)$
              else
                  $f(x - 1, f(x, y - 1))$
              endif ;
```

Choosing the following space of approximate properties:

L =



we must solve:

$$\begin{cases} \phi_1 & = & \boldsymbol{\lambda}\,(x,y) \cdot [\mathit{incr}(y) \sqcup \phi_1(\mathit{decr}(x \sqcap +), +) \sqcup \phi_2(x \sqcap +, \phi_1(x \sqcap +, \mathit{decr}(y \sqcap +)))] \\ \phi_2 & = & \boldsymbol{\lambda}\,(x,y) \cdot [\phi_1(\mathit{decr}(x), y)] \end{cases}$$

where

$$\mathit{incr} \;=\; \boldsymbol{\lambda}\,x \cdot \underline{\text{case}} \quad x \quad \underline{\text{in}} \quad \bot \to \bot\,; \quad 0 \to +\,; \quad + \to +\,; \quad \top \to + \quad \underline{\text{endcase}}\,;$$

$$\mathit{decr} \;=\; \boldsymbol{\lambda}\,x \cdot \underline{\text{case}} \quad x \quad \underline{\text{in}} \quad \bot \to \bot\,; \quad 0 \to \bot\,; \quad + \to \top\,; \quad \top \to \top \quad \underline{\text{endcase}}\,;$$

The value of $\phi_1(\top,\top)$ can be computed by a finite chaotic iteration sequence for $V = (\{(\top,\top)\}, \emptyset)$ and starting from:

$$\left[\begin{array}{lcl} \phi_1^0 & = & \boldsymbol{\lambda}\,(x,y) \cdot \bot \\ \phi_2^0 & = & \boldsymbol{\lambda}\,(x,y) \cdot \bot \end{array}\right.$$

Step 1 : $\qquad J_0 = (\{(\top,\top), (+,\top), (\top,+)\}, \emptyset)$

$$\begin{array}{lcll} \phi_1^1(\top,\top) & = & + \sqcup \phi_1^0(\top,+) \sqcup \phi_2^0(+, \phi_1^0(+,\top)) & = \;\; + \\ \phi_1^1(+,\top) & = & + \sqcup \phi_1^0(\top,+) \sqcup \phi_2^0(+, \phi_1^0(+,\top)) & = \;\; + \\ \phi_1^1(\top,+) & = & + \sqcup \phi_1^0(\top,+) \sqcup \phi_2^0(+, \phi_1^0(+,\top)) & = \;\; + \\ \text{F}{-}\mathit{closure}(\phi^1, (\{(\top,\top)\}, \emptyset)) & = & (\{(\top,\top), (\top,+), (+,\top)\}, \{(+,\bot)\}) & \end{array}$$

Step 2 : $\qquad J_1 = (\emptyset, \{(+,+)\})$

$$\begin{array}{llll}
\qquad \phi_2^2(+,+) & = & \phi_1^1(\top,+) & = \quad + \\[2ex]
\text{Step } 3: & & J_2 = (\{(\top,\top),(+,\top),(\top,+)\},\emptyset) & \\[2ex]
\qquad \phi_1^3(\top,\top) & = & + \sqcup \phi_1^2(\top,+) \sqcup \phi_2^2(+,\phi_1^2(+,\top)) & \\[2ex]
 & = & + \sqcup \phi_1^1(\top,+) \sqcup \phi_2^2(+,\phi_1^1(+,\top)) & = \quad + \\[2ex]
\qquad \phi_1^3(+,\top) & = & + \sqcup \phi_1^2(\top,+) \sqcup \phi_2^2(+,\phi_1^2(+,\top)) & = \quad + \\[2ex]
\qquad \phi_1^3(\top,+) & = & + \sqcup \phi_1^1(\top,+) \sqcup \phi_2^2(+,\phi_1^1(+,\top)) & = \quad + \\[2ex]
\qquad \text{F}-\mathit{closure}(\phi^3,(\{(\top,\top)\},\emptyset)) & = & (\{(\top,\top),(\top,+),(+,\top)\},\{(+,+)\}) & \\[2ex]
\text{Step } 4: & & J_3 = (\emptyset,\{(+,+)\}) & \\[2ex]
\qquad \phi_2^4(+,+) & = & \phi_1^3(\top,+) & = \quad + \\[2ex]
\text{Step } 5: & & J_4 = (\{(\top,\top)\}) & \\[2ex]
\qquad \phi_1^5(\top,\top) & = & + \sqcup \phi_1^4(\top,+) \sqcup \phi_2^4(+,\phi_1^4(+,\top)) & \\[2ex]
 & = & + \sqcup \phi_1^3(\top,+) \sqcup \phi_2^4(+,\phi_1^3(+,\top)) & = \quad + \\[2ex]
\qquad \text{F}-\mathit{closure}(\phi^5,(\{(\top,\top)\},\emptyset)) & = & (\{(\top,\top),(\top,+),(+,\top)\},\{(+,+)\}) &
\end{array}$$

We have proven automatically that, if Ackermann's function is called with natural number arguments, then so are the subsequent recursive calls, and the result is a strictly positive integer.
End of example.

Remark 6.3.1.1.0.3 Determination of an eligible sequence of indexes

In practice, the eligible sequence of indexes is determined during the computation using, for example, the following algorithm:

At each step k, we evaluate $\phi_i(X)$ for all $X \in V_i$ and, when $F_i(X,\phi) \overset{\star}{\bullet\!\!\!\longrightarrow} \phi_j(Y)$, we determine the value Z of $\phi_j(Y)$ as follows:

- If $\phi_j(Y)$ has already been evaluated at step k, Z is the corresponding value;
- Else, if $\phi_j(Y)$ is being evaluated (that is to say $F_j(Y,\phi) \overset{\star}{\bullet\!\!\!\longrightarrow} \phi_j(Y)$), then, if $k > 1$, the value of Z is the same as $\phi_j(Y)$ at step $k-1$, otherwise $k = 1$ and Z is the infimum $\perp$ of D_j;
- Else, Z is the value of $F_j(Y,\phi)$.

End of remark.

6.3.1.2 Nil pointers and non-nil pointers

Let us consider the procedure *reverse*$(L, \underline{\text{nil}}, L')$ which returns a copy L' of the inverse image of a linked list L:

```
type node = record
                  val : integer ;
                  next : ↑node ;
            end ;
procedure reverse(x, y : ↑node value ; z : ↑node result) =
     if x = nil then
          z := y ;
      else
          begin t : ↑node ;
               t := allocate(node) ;
               t. val :=x.val ;
               t. next :=y ;
               reverse(x.next, t; z) ;
           end ;
      endif ;
end–proc ;
```

According to 5.6.1.1, the system of approximate equations associated with *reverse* is:

$$\left\{\begin{array}{lcl} \phi_1 & = & \boldsymbol{\lambda}\,(x,y) \bullet [\underline{\text{case}}\, x\, \underline{\text{in}} \perp \to \perp \; ; nil \to y \; ; \neg nil \to \sigma_3^{\star^3}(\phi_2(\neg nil, y, \perp)) \; ; \top \to \\ & & y \sqcup \sigma_3^{\star^3}(\phi_2(\top, y, \perp))\, \underline{\text{endcase}} \; ;] \\ \phi_2 & = & \boldsymbol{\lambda}\,(x,y,z) \bullet [\bar{\sigma}_4^{\star^4}(\phi_3(x,y,z,\neg nil))] \\ \phi_3 & = & \boldsymbol{\lambda}\,(x,y,z,t) \bullet [(x,y,\phi_1(next(x),t),t)] \end{array}\right.$$

where

$$\begin{array}{lcl} \sigma_i^{\star^n}\{(x_1,\ldots,x_i,\ldots,x_n)\} & = & x_i \\ \bar{\sigma}_i^{\star^n}\{(x_1,\ldots,x_i,\ldots,x_n)\} & = & (x_1,\ldots,x_{i-1},x_{i+1},\ldots,x_n) \end{array}$$

$$next \;=\; \boldsymbol{\lambda}\, p \bullet [\underline{\text{case}}\, p\, \underline{\text{in}} \perp \to \perp \; ; nil \to \perp \; ; \neg nil \to \top \; ; \top \to \top\, \underline{\text{endcase}} \; ;]$$

Evaluating $\phi_1(\neg nil, nil)$ while determining an eligible sequence of indexes by Algorithm 6.3.1.1.0.3, we obtain:

Step 1 :

$$\begin{array}{lcl} \phi_1(\neg nil, nil) & = & \sigma_3^{\star^3}(\phi_2(\neg nil, nil, \perp)) \\ & = & \sigma_3^{\star^3}(\bar{\sigma}_4^{\star^4}(\phi_3(\neg nil, nil, \perp, nil))) \\ & = & \sigma_3^{\star^3}(\bar{\sigma}_4^{\star^4}(\neg nil, nil, \phi_1(\perp, \neg nil), \neg nil)) \\ & = & \sigma_3^{\star^3}(\bar{\sigma}_4^{\star^4}(\neg nil, nil, (\neg nil \sqcup \sigma_3^{\star^3}(\phi_2(\top, \neg nil, \perp))), \neg nil)) \\ & = & \sigma_3^{\star^3}(\bar{\sigma}_4^{\star^4}(\neg nil, nil, (\neg nil \sqcup \sigma_3^{\star^3}(\bar{\sigma}_4^{\star^4}(\phi_3(\top, \neg nil, \perp, \neg nil)))), \neg nil)) \\ & = & \sigma_3^{\star^3}(\bar{\sigma}_4^{\star^4}(\neg nil, nil, (\neg nil \sqcup \sigma_3^{\star^3}(\bar{\sigma}_4^{\star^4}(\top, \neg nil, \phi_1(\top, \neg nil), \neg nil))), \neg nil)) \end{array}$$

As $\phi_1(\top, \neg nil)$ is under evaluation, it is approximated as $\perp$:

$$\begin{array}{lcl} & = & \sigma_3^{\star^3}(\bar{\sigma}_4^{\star^4}(\neg nil, nil, (\neg nil \sqcup \sigma_3^{\star^3}(\bar{\sigma}_4^{\star^4}(\top, \neg nil, \perp, \neg nil))), \neg nil)) \\ & = & \neg nil \end{array}$$

Step 2 :

$$
\begin{aligned}
\phi_1(\neg nil, nil) &= \sigma_3^{\star^3}(\bar\sigma_4^{\star^4}(\neg nil, nil, \phi_1(\bot, \neg nil), \neg nil)) \\
&= \sigma_3^{\star^3}(\bar\sigma_4^{\star^4}(\neg nil, nil, (\neg nil \sqcup \sigma_3^{\star^3}(\bar\sigma_4^{\star^4}(\top, \neg nil, \phi_1(\top, \neg nil), \neg nil))), \neg nil))
\end{aligned}
$$

As $\phi_1(\top, \neg nil)$ is under evaluation, it is approximated as the value $\neg nil$ obtained at the previous step:

$$
\begin{aligned}
&= \sigma_3^{\star^3}(\bar\sigma_4^{\star^4}(\neg nil, nil, (\neg nil \sqcup \sigma_3^{\star^3}(\bar\sigma_4^{\star^4}(\top, \neg nil, \neg nil, \neg nil))), \neg nil)) \\
&= \neg nil
\end{aligned}
$$

Note that the computation can be reordered to correspond to Definition 6.2.1.0.1. Concerning the example, we discovered automatically that $reverse(L, \underline{\text{nil}}; L')$ returns L' different from nil when L is not nil.

6.3.1.3 Pointers pointing to different records

Let us review the example of the *reverse* procedure:

```
          procedure reverse(x, y : ↑node value ; z : ↑node result) =
  {1}
              if x = nil  then
  {2}
                  z := y ;
  {3}
               else
  {4}
                  begin t : ↑node ;
  {5}
                      t := allocate(node) ; t. val :=x. val ;
  {6}
                      t. next :=y ;
  {7}
                      reverse(x.next, t; z) ;
  {8}
                   end ;
```

```
{9}
            endif ;
{10}
        end–proc ;
{11}
```

and apply to it the approximate analysis from Paragraph 5.6.1.2. The approximate image of a predicate over n variables of pointer types $x_1, \ldots, x_n$ is an application of the form $\boldsymbol{\lambda}\,(x_1, \ldots, x_n) \bullet P$, where P is a partition of $\{x_1, \ldots, x_n\}$. Recall the convention that, if x_i and y_j are in distinct partitions, then they cannot point to the same records, even indirectly. We will write:

$$\begin{aligned}
\{/X_1, \ldots, X_n/ \ldots /Y_1, \ldots, Y_m/\} + \{Z\} &= \{/X_1, \ldots, X_n/ \ldots /Y_1, \ldots, Y_m/Z/\} \\
\{/X, X_1, \ldots, X_n/ \ldots /Y_1, \ldots, Y_m/\} - \{X\} &= \{/X_1, \ldots, X_n/ \ldots /Y_1, \ldots, Y_m/\}
\end{aligned}$$

Then, the system of approximate equations associated with *reverse* is:

$$\left\{\begin{array}{lcl}
\phi_1 & = & \boldsymbol{\lambda}\, C \bullet (\boldsymbol{\lambda}\,(a,b,z,x,y) \bullet [(C(x,y) + \{a,b,z\}) \sqcup \{/a,x/b,y/z/\}]) \\
\phi_2 & = & \boldsymbol{\lambda}\, C \bullet (\boldsymbol{\lambda}\,(a,b,z,x,y) \bullet [\varepsilon(x, C(a,b,z,x,y))]) \circ \phi_1 \\
\phi_3 & = & \boldsymbol{\lambda}\, C \bullet (\boldsymbol{\lambda}\,(a,b,z,x,y) \bullet [\varepsilon(z, C(a,b,z,x,y)) \sqcup \{/a/b/x/y,z/\}]) \circ \phi_2 \\
\phi_4 & = & \phi_1 \\
\phi_5 & = & \boldsymbol{\lambda}\, C \bullet (\boldsymbol{\lambda}\,(a,b,z,x,y,t) \bullet [C(a,b,x,y,z) + \{t\}]) \circ \phi_4 \\
\phi_6 & = & \boldsymbol{\lambda}\, C \bullet (\boldsymbol{\lambda}\,(a,b,z,x,y,t) \bullet [\varepsilon(t, C(a,b,z,x,y,t))]) \circ \phi_5 \\
\phi_7 & = & \boldsymbol{\lambda}\, C \bullet (\boldsymbol{\lambda}\,(a,b,z,x,y,t) \bullet [C(a,b,z,x,y,t) \sqcup \{/a/b/z/x/t,y/\}]) \circ \phi_6 \\
\phi_8 & = & \boldsymbol{\lambda}\, C \bullet (\boldsymbol{\lambda}\,(a,b,z,x,y,t) \bullet [\{\phi_{11}(\boldsymbol{\lambda}\,(x,t) \bullet [C(a,b,z,x,y,t) \quad - \quad \{a,b,z,y\}]) \\
 & & (x,t,z) + \{a,b,y\}\} \sqcup \varepsilon(z, C(a,b,z,x,y,t))]) \circ \phi_7 \\
\phi_9 & = & \boldsymbol{\lambda}\, C \bullet (\boldsymbol{\lambda}\,(a,b,z,x,y) \bullet [C(a,b,z,x,y,t) - \{t\}]) \circ \phi_8 \\
\phi_{10} & = & \phi_3 \sqcup \phi_9 \\
\phi_{11} & = & \boldsymbol{\lambda}\, C \bullet (\boldsymbol{\lambda}\,(a,b,z) \bullet [C(a,b,z,x,y) - \{x,y\}]) \circ \phi_{10}
\end{array}\right.$$

Let us solve this system of equations for the specification $C = \boldsymbol{\lambda}\,(x,y) \bullet \{/x/y/\}$ corresponding to the call *reverse*$(L, \underline{\text{nil}}; L')$.

Step 1 :

$$\begin{aligned}
\phi_1(C) &= \boldsymbol{\lambda}\,(a,b,z,x,y) \bullet [(\{/x/y/\} + \{a,b,z\}) \sqcup \{/a,x/b,y/z/\}] \\
&= \boldsymbol{\lambda}\,(a,b,z,x,y) \bullet \{/a,x/b,y/z/\} \\
\phi_2(C) &= \boldsymbol{\lambda}\,(a,b,z,x,y) \bullet [\varepsilon(x,\{/a,x/b,y/z/\})] \\
&= \boldsymbol{\lambda}\,(a,b,z,x,y) \bullet \{/a/x/b,y/z/\} \\
\phi_3(C) &= \boldsymbol{\lambda}\,(a,b,z,x,y) \bullet [\varepsilon(z,\{/a/x/b,y/z/\}) \sqcup \{/a/b/x/y,z/\}]) \\
&= \boldsymbol{\lambda}\,(a,b,z,x,y) \bullet \{/a/x/b,y,z/\} \\
\phi_4(C) &= \boldsymbol{\lambda}\,(a,b,z,x,y) \bullet \{/a,x/b,y/z/\} \\
\phi_5(C) &= \phi_6(C) \\
&= \boldsymbol{\lambda}\,(a,b,z,x,y,t) \bullet \{/a,x/b,y/z/t/\} \\
\phi_7(C) &= \boldsymbol{\lambda}\,(a,b,z,x,y,t) \bullet (\{/a,x/b,y/z/t/\}) \sqcup \{/a/b/z/x/t,y/\}) \\
&= \boldsymbol{\lambda}\,(a,b,z,x,y,t) \bullet \{/a,x/b,y,t/z/\}
\end{aligned}$$

As $\boldsymbol{\lambda}\,(x,t) \bullet [\phi_7(C)(a,b,z,x,y,t) - \{a,b,z,y\}]$

$$\begin{aligned}
&= \boldsymbol{\lambda}\,(x,t) \bullet [/a,x/b,y,t/z/ - \{a,b,z,y\}] \\
&= \boldsymbol{\lambda}\,(x,t) \bullet \{/x/t/\} = C
\end{aligned}$$

and the first approximation of $\phi_{11}(C)$ is the infimum $\boldsymbol{\lambda}\,(a,b,z) \bullet [\{/a/b/z/\}]$, we have:

$$\begin{aligned}
\phi_8(C) &= \boldsymbol{\lambda}\,(a,b,z,x,y,t) \bullet [\boldsymbol{\lambda}\,(a,b,z) \bullet [\{/a/b/z/\}](x,t,z) + \{a,b,y\}) \sqcup \varepsilon(z,\{/a,x/b,y,t/z/\})] \\
&= \boldsymbol{\lambda}\,(a,b,z,x,y,t) \bullet \{/a,x/b,y,t/z/\} \\
\phi_9(C) &= \boldsymbol{\lambda}\,(a,b,z,x,y) \bullet [\{/a,x/b,y,t/z/\} - \{t\}]
\end{aligned}$$

$$\begin{aligned}
&= \boldsymbol{\lambda}\,(a,b,z,x,y)\bullet\{/a,x/b,y/z/\}\\
\phi_{10}(C) &= \phi_3(C)\sqcup\phi_9(C)\\
&= \boldsymbol{\lambda}\,(a,b,z,x,y)\bullet[\{/a/x/b,y,z/\}\sqcup\{/a,x/b,y/z/\}]\\
&= \boldsymbol{\lambda}\,(a,b,z,x,y)\bullet\{/a,x/b,y,z/\}\\
\phi_{11}(C) &= \boldsymbol{\lambda}\,(a,b,z)\bullet[\{/a,x/b,y,z/\}-\{x,y\}]\\
&= \boldsymbol{\lambda}\,(a,b,z)\bullet\{/a/b,z/\}
\end{aligned}$$

Step 2 :

$$\begin{aligned}
\phi_1(C) &= \phi_4(C)=\boldsymbol{\lambda}\,(a,b,z,x,y)\bullet\{/a,x/b,y/z/\}\\
\phi_2(C) &= \boldsymbol{\lambda}\,(a,b,z,x,y)\bullet\{/a/x/b,y/z/\}\\
\phi_3(C) &= \boldsymbol{\lambda}\,(a,b,z,x,y)\bullet\{/a/x/b,y,z/\}\\
\phi_5(C) &= \phi_6(C)=\boldsymbol{\lambda}\,(a,b,z,x,y,t)\bullet\{/a,x/b,y/z/t/\}\\
\phi_7(C) &= \boldsymbol{\lambda}\,(a,b,z,x,y,t)\bullet\{/a,x/b,y,t/z/\}
\end{aligned}$$

Once again $\boldsymbol{\lambda}\,(x,t)\bullet[\phi_7(C)(a,b,z,x,y,t)-\{a,b,z,y\}]$

$$=\boldsymbol{\lambda}\,(x,t)\bullet\{/x/t/\}=C$$

and the value of $\phi_{11}(C)$ being $\boldsymbol{\lambda}\,(a,b,z)\bullet\{/a/b,z/\}$ at the previous step, we have:

$$\begin{aligned}
\phi_8(C) &= \boldsymbol{\lambda}\,(a,b,z,x,y,t)\bullet\{/a,x/b,y,t/z/\}\\
\phi_9(C) &= \boldsymbol{\lambda}\,(a,b,z,x,y)\bullet\{/a,x/b,y/z/\}\\
\phi_{10}(C) &= \boldsymbol{\lambda}\,(a,b,z,x,y)\bullet\{/a,x/b,y,z/\}\\
\phi_{11}(C) &= \boldsymbol{\lambda}\,(a,b,z)\bullet\{/a/b,z/\}
\end{aligned}$$

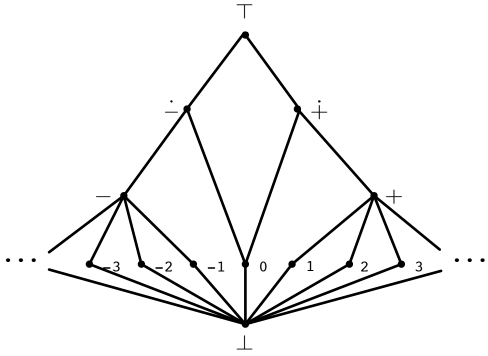
So, we have discovered automatically that, after the call to $\mathit{reverse}(L, \underline{\text{nil}}; L')$, the references L and L' cannot point to the same records, even indirectly. Similar information is available at each point in the procedure for the entry specification $\boldsymbol{\lambda}(x, y) \bullet \{/x/y/\}$.

6.3.2 Case of a space of approximate properties satisfying the ascending chain condition

Let us consider the procedure:

```
procedure factorial(x, y, z : integer value ; f : integer result) =
    if x = y then
        f := z ;
    else
        factorial(x, y + 1, z * (y + 1); f) ;
    endif ;
end–proc ;
```

such that $\mathit{factorial}(n, 0, 1; f)$ returns $f = n!$ when $n \geq 0$. We propose to analyze it on the following space of approximate properties:



Ignoring the test $(x = y)$, we have to solve:

$$\phi \quad = \quad F(\phi) \quad = \quad \boldsymbol{\lambda}\,(x, y, z) \bullet [z \sqcup \phi(x, y+1, z*(y+1))]$$

It is clear that $\phi(7,0,1) \bullet\!\!\longrightarrow \phi(7,1,1) \bullet\!\!\longrightarrow \phi(7,2,4) \bullet\!\!\longrightarrow \ldots \bullet\!\!\longrightarrow \phi(7,k,k!) \bullet\!\!\longrightarrow \ldots$ so that any eligible sequence of indexes must contain an index with a component of infinite cardinal.

So, we propose to approximate F as $\tilde{F}$ such that $F \sqsubseteq \tilde{F}$, so that $\mathit{lfp}(F) \sqsubseteq \mathit{lfp}(\tilde{F})$ (Theorem 4.3.0.1). Choosing:

$$\tilde{\phi} \quad = \quad \tilde{F}(\tilde{\phi}) \quad = \quad \boldsymbol{\lambda}\,(x, y, z) \bullet [z \sqcup \tilde{\phi}(x \sqcup x, y \sqcup (y+1), z \sqcup (z*(y+1))]$$

each time $\tilde{\phi}(x,y,z) \bullet\!\!\longrightarrow \tilde{\phi}(x',y',z')$, we have $(x,y,z) \sqsubseteq (x',y',z')$. As the lattice of approximate properties satisfies the ascending chain condition, such a derivation is necessarily finite.

$$\begin{array}{rclcl} \tilde{\phi}(7,0,1) & = & 1 \sqcup \tilde{\phi}(7, 0 \sqcup 1, 1 \sqcup 1) & = & 1 \sqcup \tilde{\phi}(7, \dot{+}, 1) \\ & = & 1 \sqcup (1 \sqcup \tilde{\phi}(7, \dot{+}, +)) & & \\ & = & 1 \sqcup (1 \sqcup (+ \sqcup \tilde{\phi}(7, \dot{+}, +))) & = & 1 \sqcup (1 \sqcup (+ \sqcup \bot)) \quad = \quad + \\ \tilde{\phi}(7,0,1) & = & 1 \sqcup (1 \sqcup \tilde{\phi}(7, \dot{+}, +)) & & \\ & = & 1 \sqcup (1 \sqcup (+ \sqcup \tilde{\phi}(7, \dot{+}, +))) & = & 1 \sqcup (1 \sqcup (+ \sqcup +)) \quad = \quad + \end{array}$$

and, more generally, $\tilde{\phi}(\top, 0, 1) = +$, which shows that $\mathit{factorial}(n, 0, 1; f)$ returns a strictly positive integer f when it terminates.

6.3.3 General case of an infinite space of approximate properties not satisfying the ascending chain condition

As example of space of approximate properties not satisfying the ascending chain condition, we consider the lattice of integer intervals from Paragraph 5.7.1. Consider the following functional equation to solve:

$$\left\{ \begin{array}{lcl} \phi_1 & = & \boldsymbol{\lambda}\, x \bullet \{\phi_1(x + [1,1])\} \end{array} \right.$$

It illustrates the problem, already encountered in the preceding paragraph, of a domain of non-converging parameters, for example $\phi_1([0,255]) \bullet\!\!\!\longrightarrow \phi_1([1,256]) \bullet\!\!\!\longrightarrow \phi_1([2,257]) \bullet\!\!\!\longrightarrow \ldots$ which we handle by over-approximating ϕ_1 by $\tilde{\phi}$ defined as:

$$\left\{ \begin{array}{lcl} \tilde{\phi}_1 & = & \boldsymbol{\lambda}\, x \bullet \{\phi_1(x \;\bar{\nabla}\; (x + [1,1]))\} \end{array} \right.$$

that we solve using Theorem 6.2.1.0.3. We obtain:

Step 1 :

$$\begin{array}{rcl} \tilde{\phi}_1^1([0,255]) & = & \tilde{\phi}_1([0,255] \;\bar{\nabla}\; [1,256]) \\ & = & \tilde{\phi}_1^1([0,+\infty]) \\ & = & \tilde{\phi}_1([0,+\infty] \;\bar{\nabla}\; [2,+\infty]) = \tilde{\phi}_1^0([0,+\infty]) = \bot \end{array}$$

Step 2 :

$$\begin{array}{rcl} \tilde{\phi}_1^2([0,255]) & = & \tilde{\phi}_1^2([0,+\infty]) = \tilde{\phi}_1^1([0,+\infty]) = \bot \end{array}$$

Now, let us consider the functional equation:

$$\left\{ \begin{array}{lcl} \phi_2 & = & \boldsymbol{\lambda}\, x \bullet \{[0,0] \sqcup ([1,1] + \phi_2(x))\} \end{array} \right.$$

which illustrates the problem, already encountered in Paragraph 5.7.3, of a non-converging iteration sequence. Indeed, the computation of $\phi_2([0,255])$ consists in solving $x = [0,0] \sqcup ([1,1] + x)$ where $x = \phi_2([0,255])$. We handle that problem by solving:

$$\left\{ \begin{array}{lcl} \tilde{\phi}_2 & = & \boldsymbol{\lambda}\, x \bullet \{\tilde{\phi}_2(x) \;\bar{\nabla}\; ([0,0] \sqcup ([1,1] + \tilde{\phi}_2(x)))\} \end{array} \right.$$

which gives:

Step 1 :

$$\begin{array}{rcl}\tilde{\phi}_2^1([0,255]) & = & \tilde{\phi}_2^0([0,255])\ \bar{\nabla}\ ([0,0] \sqcup ([1,1] + \tilde{\phi}_2^0([0,255]))) \\ & = & \bot\ \bar{\nabla}\ ([0,0] \sqcup \bot) = [0,0]\end{array}$$

Step 2 :

$$\begin{array}{rcl}\tilde{\phi}_2^2([0,255]) & = & \tilde{\phi}_2^1([0,255])\ \bar{\nabla}\ ([0,0] \sqcup ([1,1] + \tilde{\phi}_2^1([0,255]))) \\ & = & [0,0]\ \bar{\nabla}\ ([0,0] \sqcup [1,1]) = [0,+\infty]\end{array}$$

Step 3 :

$$\begin{array}{rcl}\tilde{\phi}_2^3([0,255]) & = & \tilde{\phi}_2^2([0,255])\ \bar{\nabla}\ ([0,0] \sqcup ([1,1] + \tilde{\phi}_2^2([0,255]))) \\ & = & [0,+\infty]\ \bar{\nabla}\ ([0,0] \sqcup [1,+\infty]) = [0,+\infty] \\ \tilde{\phi}_2([0,255]) & \sqsubseteq & \tilde{\phi}_2^2([0,255]) = [0,+\infty]\end{array}$$

In the general case where the two issues occur at the same time, they can be solved as previously in the framework of Theorem 6.2.2.0.2. For example, the analysis of MacCarthy's 91 function (6.1.0.3) consists in solving:

$$\left\{\ \phi_1 \quad = \quad \boldsymbol{\lambda}\, x \bullet [((x \sqcap [101,+\infty]) - [10,10]) \sqcup \phi_1(\phi_1((x \sqcap [-\infty,100]) + [11,11]))]\right.$$

which we over-approximate as:

$$\left\{\begin{array}{rcl}\tilde{\phi}_1 & = & \boldsymbol{\lambda}\, x \bullet [\tilde{\phi}_1(x)\ \bar{\nabla}\ \{((x \sqcap [101,+\infty]) - [10,10]) \sqcup \tilde{\phi}_1(x\ \bar{\nabla}\ \tilde{\phi}_1(x\ \bar{\nabla}\ ((x \sqcap \\ & & [-\infty,100]) + [11,11]))))\}]\end{array}\right.$$

and allows us to discover that MacCarthy's function returns a result greater than or equal to 91:

$$\left[\ \tilde{\phi}_1([-\infty,+\infty]) \quad = \quad [91,+\infty]\right.$$

6.4 BIBLIOGRAPHIC NOTES

The results in this chapter improve Cousot & Cousot [1977d], in particular concerning the support for unconditional branching (a system of equations is now associated with a program using the notion of program point and not anymore by induction on the syntactic structure of the program). The Example 6.3.1.3 is also treated in a more rigorous way.

In Paragraph 6.1, the usage of ancillary variables (that store initial values) to express intermediate predicates in recursive procedures seems essential *a posteriori*, as it is in the rules of procedure proofs introduced by Hoare [1971] and generalized by Igarashi, London & Luckham [1975], Ernst [1977], Apt & de Bakker [1977], Guttag, Horning & London [1977] for the axiomatic system to be complete (de Bakker & Merteens [1975], Cook [1975], Gorelick [1975], Apt & Meertens [1977], Apt, Bergstra & Meertens [1977], Clarke [1977]). It allows also defining the meaning of a recursive procedure regardless of a particular call. Note that, to define the semantics of procedures, we did not use the technique of syntactic substitutions (which seems of limited power, de Bakker [1977b]) and not even resort to the concept of "continuation", as in Milne [1977]. That is possible as long as a procedure body can be statically associated with every procedure name and excludes call by name, passing procedures and functions as parameters, coroutines etc. Another element of comparison with Milne [1977] is that we associate a system of equations, not with a language, but with a program, which is useful to reason on approximation techniques.

The automatic analysis of semantic properties of recursive procedures was very little studied. We can cite Sintzoff [1972] who studies the manual verification of properties with the help of symbolic execution, Wegbreit [1975] who deals with procedures by expanding the body of the procedure at each call, and Karr [1976] who proposes a symbolic execution of the iterative program corresponding to a compilation of the procedure with a recursivity stack.

In the particular case of classic boolean program optimization techniques, we can cite Spillman [1971], Allen [1974], Lomet [1975], Rosen [1975], and Barth [1977].

CHAPTER 7.

CONCLUSIONS

7. CONCLUSIONS

Concluding remarks having been given in each chapter, now we discuss some directions for future work.

The fixpoint approach to study the behavior of discrete dynamic systems (Paragraph 3.1) as well as the study of methods of fixpoint approximation (Chapter 4) show quite clearly that we can study the techniques of semantic analysis of programs regardless of the languages used for programming. However, we devoted the best part of our study to the case of deterministic discrete dynamic systems because deterministic programs or more frequent in practice in Computer Science. The case of non-deterministic and parallel programs, which are worth studying too, is apparently more complex. Discrete dynamic systems most probably offer a framework that is abstract enough to allow for a study that would be independent of language issues (which, in general, make the understanding of non-deterministic programming issues more complicated than easier.)

The language issues do not really appear before we have to make the semantic analysis of programs written in a particular language. On that point, our work must be completed to take into account on one hand the problems coming from complex data structures and on the other hand the problems coming from dynamic control structures (which do not allow a static partitioning of the set of states). Sometimes it is difficult to derive a deductive semantics from a low level or informal semantics. This work could be undertaken for the most commonly used programming languages. Indeed it seems that it is not possible to do rigorous reasonings on programs written in a language if it is not possible to define the deductive semantics of that language.

Thanks to the notions of closure and extrapolation, Chapter 4 potentially defines all approximate and possibly automatable analyses we can consider for programs.

However, this point of view is theoretical and in practical applications an important work is left which is necessary to find the good level of approximation, that is offering a good cost/precision ratio for the analysis. It is certain that many applications can be developed for classic languages, such as Algol 68 which features most of the difficulties we can meet in other programming languages.

Perhaps is it rather preferable to envisage new languages or language features designed towards the resolution of exact or approximate semantic analysis of programs. For example, our study brings out that local declarations are more useful than global declarations, that the analysis of properties of objects manipulated by a program depends on the operations made on those objects (point of view of the 'abstract types') but also and most of all on the context in which those operations are made, that the extension of a programming language should come with the information necessary for the analysis of the features in the extension, that the type of objects can be analyzed with more or less refinement and that there exists a hierarchy between type notions and assertions. Those ideas, that need further elaboration, could guide the conception of a programming language, that would be a point of view often neglected in the development of very high level languages.

In Chapters 2 and 4, we studied the construction and approximation of fixpoints of monotonic operators on a lattice, from a purely algebraic point of view. The fact remains that the analogy with numeric analysis was useful and could certainly be successfully exploited again. It is possible to improve the efficiency of the iterative methods and to go into our notion of approximation in more depth. New methods (not necessarily iterative) of resolution of approximate semantic equations can be devised. Our hypothesis of monotone equations over a complete lattice was well suited to our problems. It is a little too strong for some problems which involve a non monotonic negation. So we have to think about hypotheses weaker than monotonicity. It is certain that we also have to envisage stronger hypotheses, for we have observed in applications that some hypotheses specific to particular applications are nonetheless useful to devise methods of resolution.

CHAPTER 8.

Bibliography

8. Bibliography

ABIAN S. & BROWN A.B. [1971], *A theorem on partially ordered set with application to fixed point theorems*, Canad. J. Math., 13(1961), 78–82.

ABTROUN A. [1977], *Recherche d'une permutation optimale des variables dans la méthode itérative de Gauss-Seidel*, Thèse de 3ème cycle, Université Scientifique et Médicale de Grenoble, (mai 1977).

ACHACHE A. [1969], *Structure de l'ensemble de fermetures d'un treillis complet*, Portugal. Math., 28(1969), 111–119.

AHO A.V. & ULLMAN J.D. [1976], *Node listings for reducible flow graphs*, J. Computer and Systems Sciences 13, 3(1976). 286–299.

AHO A.V. & ULLMAN J.D. [1977], *Principles of compiler design*, Addison Wesley Pub. Co., (1977).

ALLEN F.E. [1970], *Control flow analysis*, SIGPLAN Notices 5. 7(1970). 1–19.

ALLEN F.E. [1971], *A basis for program optimization*, Proc. IFIP Congress 71, Vol.1. North-Holland Pub. Co., Amsterdam, (1971), 385–390.

ALLEN F.E. [1974], *Interprocedural data flow analysis*, Proc. IFIP Congress 74, North-Holland Pub. Co., Amsterdam, (1974), 398–402.

ALLEN F.E. & COCKE J. [1972], *Graph theoretic constructs for program flow analysis*, IBM Res. Rep. RC-3923. T.J. Watson Research Center, Yorktown Heights. N.Y., U.S.A., (July 1972).

AMANN H. [1976], *Fixed point equations and non-linear eigenvalue problems in ordered Banach spaces*, SIAM Review, 18(Oct. 1976), 620–709.

APT K.R. & de BAKKER J.W. [1977], *Semantics and proof theory of PASCAL procedures*, Tech. Rep., Stichting Mathematisch Centrum, Amsterdam, (1977).

APT K.R. & MEERTENS L. [1977], *Completeness with finite systems of intermediate assertions for recursive program schemes*, Report IW-84/77, Mathematisch Centrum, Amsterdam, (1977)[1].

APT K.R., BERGSTRA J.A. & MEERTENS L.G. [1977], *Recursive assertions are not enough or are they?*, Report IW-92/77. Mathematisch Centrum, Amsterdam, (1977)[2].

BASU S.K. & YEH R.T. [1975], *Strong verification of programs*, Res. Rep. SESLTR-13. Soft. Eng. and Syst. Lab., Univ. of Texas at Austin, (June 1975)[3].

BARTH J.M. [1977], *An interprocedural data flow analysis algorithm*, Proc. of the Fourth ACM Symp. on Principles of Programming Languages, Los Angeles, Calif., U.S.A., (Jan. 1977), 119–131.

BAUDET G. [1976], *Asynchronous iterative methods for multiprocessors*, Research Report. Carnegie Mellon Univ., Pittsburgh. PA., (Nov. 1976). (à paraître dans JACM[4]).

BAUER A. & SAAL H. [1974], *Does APL really need run-time checking*, Software Practice and Experience 4. 2(1974).

BEKIĆ H. [1969], *Definable operations in generaL algebras and the theory of automata and flowcharts*, Manuscript. IBM Lab., Vienne, (1969)[5].

BERGE C. [1973], *Graphes et hypergraphes*, Dunod, Paris (1973).

BIRD R. [1976], *Programs and machines: an introduction to the theory of computation*, Wiley & Sons. London, (1976).

BIRKHOFF G. [1967], *Lattice theory*, AMS Colloquim Publications. XXV, Third edition. Providence, R.I., U.S.A., (1967).

BJØRNER O. [1977a], *Programming languages: formal development of interpreters and*

[1] SIAM J. on Computing, Vol. 9, Issue 4, 665–671, SIAM 1980

[2] Theoretical Computer Science, Vol. 8, 73–87, 1979.

[3] IEEE Trans. Software Eng. 1(3): 339-346 (1975)

[4] Journal of the ACM (JACM), Volume 25, Issue 2 (April 1978), 226–244.

[5] Lecture Notes In Computer Science; Vol. 177, Programming Languages and Their Definition, 30–55, 1984

compilers, Int. Comp. Symp., North-Holland Pub. Co., (1977).

BJØRNER O. [1977b], *Programming languages: linguistics and semantics*, Int. Comp. Symp., North-Holland Pub. Co., (1977).

BOOM H. [1974], *Optimization analysis of programs in languages with pointer variables*, Ph.D. Thesis, Dept. Appl. Math. and Comp. Science. University of Waterloo. U.S.A., (1974).

BOURBAKI N. [1967], *Théorie des ensembles*, Livre I, Chap. III, Fas. XX, Ed. Hermann, 2ème édition, Paris, (1967).

BURSTALL R.M. [1969], *Proving properties of programs by structural induction*, Computer Journal 12, (1969), 41–48.

BURSTALL R.M. [1974], *Program proving as hand simulation with a little induction*, Proc. IFIP Congress 74, Software, North-Holland Pub. Co., Amsterdam, (1974), 308–312.

CAPLAIN M. [1975], *Finding invariant assertion for proving programs*, Proc. Int. Conf. on Reliable Software, Los Angeles, Calif., U.S.A., (April 1975), 165–171.

CARTWRIGHT R. & OPPEN D.C. [1978], *Unrestricted procedure calls in Hoare's logic*, Conf. Rec. of the 5th ACM Symp. on Principles of Programming Languages, Tucson, Ariz., U.S.A., (Jan. 1978), 131–140.

CHARNAY M. [1975], *Itérations chaotiques sur un produit d'espaces métriques*, Thèse de 3ème cycle, Lyon, (1975).

CHAZAN O. & MIRANKER W. [1969], *Chaotic relaxation*, Linear Algebra and its Appl., 2(1969), 199–222.

CHURCH A. [1951], *The calculi of lambda-conversion*, Annals of Math. Studies. 6(1951).

CLARKE E.M. Jr. [1977], *Program invariants as fixed points*, Proc. 18th Annual Symp. on Foundations of Computer Science, Providence, R.I., U.S.A., (Oct.31–Nov.2 1977). 18–29.

CLINT M. & HOARE C.A.R. [1972], *Program proving jumps and functions*, Acta Informatica, 1(1972), 214–224.

COCKE J. [1970], *Global common subexpressions elimination*, SIGPLAN Notices 5, 7(1970), 20–24.

COMTE P. [1976], *Algorithmes de relaxation-décentralisation*, Thèse de 3ème cycle, Besançon, (1976).

COOK S.A. [1975], *Axiomatic and interpretative semantics for an ALGOL fragment*, Tech. Rep. 79, Dept. of Comp. Science, U. of Toronto, Canada (1975).

COOPER D.C. [1971], *Programs for mechanical program verification*, Machine Intelligence 6, American Elsevier, New York, (1971), 43–59.

COUSOT P. [1974], *Définition interprétative et implantation de langages de programmation*, Thèse Docteur-Ingénieur, Université Scientifique et Médicale de Grenoble, (Déc. 1974).

COUSOT P. [1976], *The system implementation language LIS, an introduction*, édité par IRIA. Rocquencourt, (Juin 1976).

COUSOT P. [1977b], A mathematical theory of global program analysis, Présenté au "Panel: Mathematical Theory of Data Flow Analysis". Chairman: ULLMAN J.D. (U.S.A.), Panelists: COUSOT P. (F.); KENNEDY K. (U.S.A.), ROSEN B. (U.S.A.), TARJAN R. (U.S.A.). IFIP Congress 1977, Toronto, Canada, (Aug. 1977).

COUSOT P. [1977c], *Analysis of programs properties*, Présenté au "Panel : Use and Benefit of Formal Description Techniques, Report of W.G.2.2.". Chairman: NEUHOLD E. (D.), Panelists: BLUM E. (U.S.A.), BOEHM B. (I.), COUSOT P. (F.), De BAKKER J. (N.L.), IGARASHI S. (J.), NIVAT M. (F.), OWICKI S. (U.S.A.), TENNENT R. (C.D.N.), IFIP Congress 1977, Toronto, Canada, (Aug. 1977).

COUSOT P. [1977d], *Iterative and approximate methods for compile time analysis of programs*, IBM Seminar, T.J. Watson Research Center, Computer Sciences Dept., Yorktown Heights, N.Y., U.S.A., (August 18, 1977).

COUSOT P. [1977e], *Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice*, Rapport de Recherche n° 88, Laboratoire IMAG, Grenoble, (Sept. 1977).

COUSOT P. [1978], *Chaotic and asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice*, IBM Seminar, T.J. Watson Research Center, Mathematical Sciences Dept., Yorktown Heights, N.Y., U.S.A., (January 26, 1978).

COUSOT P. & COUSOT R. [1975a], *Vérification statique de la cohérence dynamique des programmes*, Rapport du contrat IRIA-SESDRI 75-035, (Sept. 1975).

COUSOT P. & COUSOT R. [1975b], *Static verification of dynamic type properties of variables*, Rapport de Recherche n° 25, Laboratoire IMAG, Grenoble, (Nov. 1975).

COUSOT P. & COUSOT R. [1976], *Static determination of dynamic properties of programs*, Proc. 2nd Int. Symp. on Programming, Dunod. Paris, (Avril 1976), 106–130. [Aussi dans MOL Bulletin 5, Cousot P. (Ed.), IRIA Rocquencourt, (Sept. 1976), 27–52].

COUSOT P. & COUSOT R. [1977a], *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, Conf. Rec. of the 4th ACM Symp. on Principles of Programming Languages, Los Angeles, Calif., U.S.A., (Janv. 1977), 238–252.

COUSOT P. & COUSOT R. [1977b], *Static determination of dynamic properties of generalized type unions*, ACM Conf. on Language Design for Reliable Software, Raleigh, N.C., U.S.A., (March 1977). SIGPLAN Notices 12, 3(March 1977), 77–94.

COUSOT P. & COUSOT R. [1977c],. *Fixed point approach to the approximate semantic analysis of programs*, (Juin 1977).

COUSOT P. & COUSOT R. [1977d], *Static determination of dynamic properties of recursive procedures*, IFIP W.G.2.2. Working Conf. on Formal Description of Programming Concepts, St-Andrews. N.B., Canada, North-Holland Pub. Co. (Aug. 1977).

COUSOT P. & COUSOT R. [1977e], *Automatic synthesis of optimal invariant assertions: mathematical foundations*, Proc. of the ACM Symp. on Artificial Intelligence & Programming Languages, Rochester, New York, SIGPLAN Notices 12, 8(Aug. 1977), 1–12.

COUSOT P. & COUSOT R. [1977f], *Constructive versions of Tarski's fixed points theorems*, Rapport de Recherche n° 85. Laboratoire IMAG, Grenoble, (Sept. 1977)[6].

COUSOT P. & HALBWACHS N. [1978], *Automatic discovery of linear restraints among variables of a program*, Conf. Rec. of the 5th ACM Symp. on Principles of Programming Languages, Tucson, Ariz., U.S.A., (Jan. 1978), 84–97.

CURRY G. [1977], *Programming by abstract demonstration*, Technical Report n° 77-08-02. Computer Sci. Department, University of Washington, U.S.A., (Aug. 1977).

De BAKKER J.W. [1976], *Semantics and termination of non deterministic recursive programs*, 3rd Int. ColI. Automata. Languages and Programming, University Press, Edinburgh, (1976), 435–477.

De BAKKER J.W. [1977a]. *Topics in denotational semantics*, Lecture Notes for the Advanced Summer School on Math. Foundations of Computer Science, Turku, (June 1977).

De BAKKER J.W. [1977b], *Recursive programs as predicate transformers*, IFIP Working Conf. on Formal Description of Programming Concepts, St.Andrews, Canada, North-Holland Pub. Co., (Aug. 1977).

De BAKKER J.W. & MEERTENS L.G. [1975], *On the completeness of the inductive assertion method*, Journal of Computer and System Sciences 11, 3(1975), 323–357.

De BAKKER J.W. & SCOTT D. [1969]. *A theory of programs*, IBM Seminar, Vienna, (1969).

De MARR R. [1964], *Common fixed points for isotone mappings*, Colloquium Math.

[6] Pacific Journal of Mathematics, Vol. 82, No. 1, 1979, pp. 43–57.

13(1964), 45–48.

DEVIDE V. [1964], *On monotonous mappings of complete lattices*, Fundamenta Mathematicae, LIII(2), (1964), 147–154.

DIJKSTRA E.W. [1968], *GO TO Statement considered harmful*, Letter to the editor, CACM 11, 3(March 1968).

DIJKSTRA E.W. [1975], *Guarded commands, non determinacy and formal derivation of programs*, CACM 18, 8(Aug. 1975). 453–457.

DIJKSTRA E.W. [1976], *A discipline of programming*, Prentice-Hall, Englewood Cliffs, N.J., U.S.A., (1976).

DIJKSTRA E.W. [1977], *On making solutions more and more fine grained*, Note EW0622 `www.cs.utexas.edu/users/EWD/ewd06xx/EWD622.PDF`.

DUBREIL-JACOTIN M.L., LESIEUR L. & CROISOT R. [1963], *Leçons sur la théorie des treillis des structures algébriques ordonnées et des treillis géométriques*, Gauthier-Villars, Paris, (1963).

DWINGER Ph. [1954], *On the closure operators of a complete lattice*, Nederl. Akad. Wetench. Proc. Ser. A, 57(1954), 560–563.

DWINGER Ph. [1955], *The closure operators of the cardinal and ordinal sums and products of partially ordered sets and closed lattices*, Nederl. Akad. Wetensch. Proc. Ser. A, 17(1955), 341–351.

EARNEST C. [1974], *Some topics in code optimization*, JACM 21, 1(1974), 76–102.

ELSPAS B. [1974], *The semi-automatic generation of inductive assertions for proving program correctness*, Research Rep., SRI, Menlo Park, Calif., U.S.A., (July 1974).

EL TARAZI M.N. [1976], *Sur des algorithmes mixtes par blocs de type Newton-Relaxation chaotique à retards*, CRAS Paris, t. 283, Série A, (Oct. 1976), 721–724.

ERNST G.W. [1977], *Rules of inference for procedure calls*, Acta Informatica 8, (1977), 145–152.

FINANCE J.P. [1976], *Une formalisation de la sémantique des langages de programmation*, RAIRO 2(Août 1976), 10(Dec. 1976).

FLOYD R.W. [1967], *Assigning meaning to programs*, Proc. Symp. in Applied Math., Vo1. 19. AMS. Providence, R.I., U.S.A., (1967), 19–32.

GERMAN S. [1978], *Automating proofs of the absence of common runtime errors*, Conf. Rec. of the 5th ACM Symp. on Principles of Programming Languages, Tucson, Ariz., U.S.A., (Jan. 1978), 105–118.

GERMAN S. & WEGBREIT B. [1975], *A synthesizer of inductive assertions*, IEEE Trans. Software Eng., SE-1, 1(March 1975), 68–75.

GILLETT W.O. [1977], *Iterative global flow techniques for detecting program anomalies*, Ph.D Thesis. UIUCOCS-R-77-848, U. of Illinois at Urbana Champaign, (Jan. 1977).

GORELICK G.A. [1975], *A complete axiomatic system for proving assertions about recursive and non-recursive programs*, Tech. Rep. 75, Dept. of Comp. Science, U. of Toronto, Canada, (Jan. 1975).

GRAHAM S.L. & WEGMAN M. [1976], *A fast and usually linear algorithm for global flow analysis*, JACM 23, 1(1976), 172–202.

GRÄTZER G. [1971], Lattice theory first concepts and distributive lattices, W.H. Freeman and Co., San Francisco, Calif., U.S.A., (1971).

GRÄTZER G. & SCHMIDT E.T. [1958], Ideals and congruence relations in lattices, Acta Mathematica Hungarica, Volume 9, Numbers 1-2, (1958), 137-175.

GUTTAG J.V., HORNING J.J. & LONDON R.L. [1977], *A proof rule for EUCLID procedures*, IFIP W.G.2.2 Working Conf. on Formal Description of Programming Concepts, St-Andrews, N.B., Canada, North-Holland Pub. Co., (Aug. 1977).

HANTLER S.L. & KING J.C. [1976], *An introduction to proving the correctness of programs*, Comp. Surveys 8, 3(Sept. 1976), 331–353.

HARRISON W. [1977], *Compiler analysis of the value ranges of variables*, IEEE Trans. on Software Engineering, 3(1977), 243–250.

HECHT M.S. & ULLMAN J.D. [1972], *Flow graph reducibility*, SIAM J. Computing 1. 2(1972), 188–202.

HECHT M.S. & ULLMAN J.D. [1973], *A simple algorithm for global data flow analysis*, Conf. Rec. of the ACM Symp. on Principles of Programming Languages, Boston, Mass., U.S.A., (Oct. 1973), 207–217. [Aussi SIAM J. Computing 4,4(1975), 519–532].

HECHT M.S. & ULLMAN J.D. [1974], *Characterizations of reducible flow graphs*, JACM 21, 3(1974), 367–375.

HECHT M.S. & ULLMAN J.D. [1975], *A simple algorithm for global data flow analysis of programs*, SIAM J. Computing 4, 4(1975), 519–532.

HEHNER E. C.R. [1976], *DO considered OD : a contribution to the programming calculus*, Tech. Rep. CSRG-75, Comp. Syst. Res. Group, U. of Toronto, Canada, (Nov. 1976)[7].

HITCHCOCK P. & PARK D. [1973], *Induction rules and proofs of termination*, Proc. Symp. on Automata, Languages and Programming, North-Holland Pub. Co., (1973), 225–251.

HOARE C.A.R. [1962], *Quicksort*, Computer Journal 5, 1(1962), 10–15.

HOARE C.A.R. [1969], *An axiomatic approach to computer programming*, CACM 12, 10(Oct. 1969), 576–580,583.

HOARE C.A. R. [1971], *Procedures and parameters: an axiomatic approach*, Lect. Notes in Math. 188, Springer-Verlag, Berlin, (1971), 102–116.

HÖFT H. & HÖFT M. [1976], *Some fixed point theorems for partially ordered sets*, Canad. J. Math., 5(1976), 992–997.

HOPCROFT J.E. & ULLMAN J.D. [1969], *Formal languages and their relation to*

[7] *DO considered OD: A contribution to the programming calculus*, Acta Informatica, Vol. 11, Nb. 4, Dec. 1979, 287–304

automata, Addison-Wesley. Reading, Mass., (1969).

ICHBIAH J.D., MOREL E. & RENVOISE C. [1972], *Une méthode directe de résolution des systèmes d'équations de redondance*, Note de programmation CII-OSB/AS/72/023, (Juin 1972).

ICHBIAH J.D., RISSEN J.P., HÉLLlARD J.C. & COUSOT P. [1974], *The system implementation language LIS, Reference Manual*, Rapport CII-4549-E1/EN. (Déc. 1974), Révisé (Jan. 1976).

IGARASHI S., LONDON R.L. & LUCKHAM D.C. [1975], *Automatic program verification. I. A logical basis and its implementation*, Acta Informatica 4, (1975), 145–182.

ISEKI K. [1951], *On closure operation in lattice theory*, Nederl. Akad. Wetensch. Proc. Ser. A, 54, Indag. Math., 13(1951), 318–320.

IVERSON K. E. [1962], *A Programming Language*, J. Wiley & Sons Inc., (1962).

JACQUEMARD C. [1977], *Contribution à l'étude d'algorithmes de relaxation à convergence monotone*, Thèse de 3ème cycle, Besançon, (Mai 1977).

JENSEN J. [1965], *Generation of machine code in ALGOL compilers*, BIT 5, (1965), 235–245.

JENSEN K. & WIRTH N. [1976], *PASCAL user manual and report*, Second edition, Springer-Verlag, Heidelberg, (1975).

JONES N.D. & MUCHNICK S.S. [1976], *Binding time optimization in programming languages: some thoughts toward the design of an ideal language*, Conf. Rec. of the 3rd ACM Symp. on Principles of Programming Languages, Atlanta, G.A., U.S.A., (Jan. 1976), 77–91.

KAM J.B. & ULLMAN J.D. [1977], *Monotone data flow analysis frameworks*, Acta Informatica, 7:3(Sep. 1977), 305–317.

KAPLAN M.A. & ULLMAN J.D. [1978], *A general scheme for the automatic inference of variable types*, Conf. Rec. of the 5th ACM symp. on Principes of Programming Languages, Tucson, Ariz., U.S.A., (Jan. 1978), 60–75.

KARR M. [1975], *Gathering information about programs*, Mass. Compo Associates Inc., CA-7507-1411, (July 1975).

KARR M. [1976], *Affine relationships among variables of a program*, Acta Informatica 6 (April 1976), 188–206.

KASVANOV V.N. [1973], *Some properties of fully reducible graphs*, Information Processing Letters 2, 4(1974), 113–117.

KATZ S. & MANNA Z. [1976], *Logical analysis of program*, CACM 19, 4(Avril 1976), 188–206.

KENNEDY K. [1971], *A global flow analysis algorithm*, Int. J. Computer Math. 3, (1971), 5–15.

KENNEDY K. [1972], *Index register allocation in straight line code and simple loops*, in: Rustin R., Design and Optimization of Compilers, Prentice-Hall, Englewood Cliffs, N.J., U.S.A., (1972).

KENNEDY K. [1975], *Node listings applied to dataflow analysis*, Conf. Rec. of the 2nd ACM Symp. on Principles of Programming Languages, Palo Alto, Calif., U.S.A., (Jan. 1975), 10–21.

KENNEDY K. [1976], *A comparison of two algorithms for global data flow analysis*, SIAM J. Computing 1, (Mar. 1976), 158–180.

KILDALL G. [1973], *A unified approach to global program optimization*, Conf. Rec. of the ACM Symp. on Principles of Programming Languages, Boston, Mass. (Oct. 1973), 194–206.

KLEENE S.C. [1952], *Introduction to metamathematics*, North-Holland. Pub. Co., Amsterdam, (1952).

KNASTER B. [1928], *Un théorème sur les fonctions d'ensembles*, Ann. Soc. Polon. Math., 5(1928), 133–134.

KNUTH D.E. [1971], *An empirical study of FORTRAN Programs*, Software Practice and Experience 1, 2(1971), 105–134.

KNUTH D.E. [1973], *The art of computer programming, vol. 3, sorting and searching*, Addison-Wesley Pub. Co., Reading, Mass., U.S.A., (1973).

KNUTH D.E. [1974], *Structured programming with GOTO statements*, Computing Surveys 6, 4(Dec. 1974).

KOLODNER I.I. [1968], *On completeness of partially ordered sets and fix-points theorems for isotone mappings*, Amer. Math. Monthly, 75(1968), 48–49.

KRASNOSEL'SKII M.A. [1964], *Positive solutions of operator equations*, P. Noordhoff Ltd, Groningen, The Netherlands, (1964).

KUHN H.W. & MacKINNON J. [1975], *Sandwich method for finding fixpoints*, J. Optimiz. Th. and Applications, 17(1975), 189–204.

LADEGAILLERIE Y. [1973], *Préfermeture sur un ensemble ordonné*, RAIRO 1(1973), 35–43.

LAMPSON B.W., HORNING J.J., LONDON R.L., MITCHELL J.G. & POPEK G.J. [1976], *Report on the programming language EUCLID*, MOL Bulletin 5. Cousot P. (Ed.), (Sept. 1976), 92–172., SIGPLAN Notices 12, 2(Feb. 1977).

LANERY E. [1966], *Recherche d'un système générateur minimal d'un polyèdre convexe*, Thèse de 3ème cycle, Caen, (1966).

LESZCZYLOWSKI J. [1971], *A theorem on resolving equations in the space of languages*, Bull. Acad. Polon. Sci., Ser. Sci. Math. Astronom. Phys., 12(1971), 967–970.

LOMET D.B. [1975], *Data flow analysis in the presence of procedure calls*, IBM Report RC-5728, T.J. Watson Research Center, Yorktown Heights, N.Y., U.S.A., (1975)[8].

LORHO B. [1974], *De la définition à la traduction des langages de programmation : méthode des attributs sémantiques*, Thèse d'Etat, Université Paul Sabatier de Toulouse, (Déc. 1974).

LUCKHAM D. & SUZUKI N. [1976], *Automatic program verification V : verification-*

[8] IBM J. of Research and Development, Vol. 21, Nb. 6, 559–571, 1977.

oriented proof rules for arrays, records and pointers, Report STAN-CS-76-549, Comp. Sci. Dept., Stanford Univ.,Calif., U.S.A., (March 1976)[9].

LUONG N.X. [1975], *Algorithmes de relaxation conduits par l'algorithme secondaire*, Thèse de 3ème cycle, Besançon, (1975).

MAHJOUB Z. [1977], *Expérimentation de stratégies itératives chaotiques sur les prob-Lèmes de point fixe à grand nombre de variables*, Thèse de Docteur-Ingénieur. Université Scientifique et Médicale de Grenoble. (Mai 1977).

MANNA Z. [1974], *Mathematical theory of computation*, Mac-Graw Hill Book Co., New York, U.S.A., (1974).

MANNA Z., NESS Z. & VUILLEMIN J. [1973], *Inductive methods for proving properties of programs*, CACM 16, 8(Aug. 1973), 491–502.

MANNA Z. & SHAMIR A. [1977], *The convergence of functions to fixed points of recursive definitions*, Report STAN-CS-77-614. Stanford Univ., Calif., U.S.A., (May 1977)[10].

MARKOWSKY G. [1976], *Chain-complete posets and directed sets with applications*, Algebra Univ., 6(1976), 53–58.

MEERTENS L. [1975], *Mode and meaning*, dans Schuman S. (Ed.), New directions in algorithmic languages 1975, IFIP W.G. 2.1., IRIA Pub., (1975).

MIELLOU J.C. [1975a], *Algorithmes de relaxation chaotique à retards*, RAIRO, Revue Rouge. AFCET RI, (1975), 55–82.

MIELLOU J.C. [1975b], *Itérations chaotiques à retards ; étude de la convergence dans le cas d'espaces partiellement ordonnés*, CRAS Paris, t. 280, Série A, (Jan. 1975), 233–236.

MIELLOU J.C. [1977], *Algorithmes de relaxation : propriétés de convergence monotone*, Séminaire d'Analyse Numérique n° 278, Laboratoire IMAG, Grenoble, (Juin 1977).

[9] ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 1, Issue 2 (October 1979), 226–244, 1979.

[10] Theor. Comput. Sci. 6, 109–141 (1978).

MILNE R. [1977], *Transforming predicate transformer*, IFIP W.G.2.2. Working Conf. an Formal Description of Programming Concepts, St-Andrews, N.B., Canada, (Aug. 1977).

MILNE R. & STRACHEY C. [1976], *A theory of programming language semantics*, Chapman and Hall (Londres) & Wiley (New York), (1976).

MIRANKER W.L. [1977], *Parallel methods for solving equations*, IBM Research Report RC-6545 (# 28250), Mathematical Sciences Dept.,

MONK O. [1969], *Introduction to set theory*, Int. Series in Pure and Applied Mathematics, Mac-Graw Hill Book Co., N.Y., U.S.A., (1969).

MONTEIRO A. [1945], *Caractérisation de l'opération de fermeture par un seul axiome*, Portugal. Math. 4(1945), 158–160.

MONTEIRO A. & RIBEIRO H. [1942], *L'opération de fermeture et ses invariants dans les systèmes partiellement ordonnés*, Portugal. Math. 3(1942), 171–184.

MOORE E.H. [1910], *Introduction to a form of general analysis*, New Haven Colloquium, (1910).

MOREL É. & RENVOISE C. [1974], *Étude et réalisation d'un optimiseur global*, Thèse de 3ème cycle, Univ. de Paris VI, (Juin 1974).

MORGADO J. [1960]b *Some results on closure operations of partially ordered sets*, Portugal. Math. 19(1960), 101 -139.

MORGADO J. [1961] *On the closure operators of the ordinal sum of partially ordered sets*, Nederl. Akad. Wetench. Proc. Ser. 1, 23(1961), 546–550.

MORGADO J. [1962a], *Note on complemented closure operators of complete lattices*, Portugal. Math. 21, 3(1962), 135–142.

MORGADO J. [1962b], *A characterization of the closure operators by means of a single axiom*, Portugal. Math., 21(1962), 155–156.

MORGADO J. [1963], *On the closure operators of the cardinal product of partially ordered sets*, Nederl. Akad. Wetench. Proc. Ser. A, 25(1963), 65–75.

MORGADO J. [1964], *Note on the distributive closure operators of a complete lattice*, Portugal. Math. 23, 1(1964), 11–25.

MORGADO J. [1965a], *Note on the system of closure operators of the ordinal product of partially ordered sets*, Portugal. Math. 24, 4(1965), 189–220.

MORGADO J. [1965b], *A single axiom for closure operators of partially ordered sets*, Gazeta de Mathematica, 100(1965), 57–58.

MORGADO J. [1966], *Factorization of the lattice of closure operators of a complete lattice*, Portugal. Math. 25, 1(1966), 181–185.

NAUR P. [1963], (Ed.). *"Revised report on the algorithmic language ALGOL 60"*, CACM 6, 1(Jan. 1963), 1–17.

NAUR P. [1965], *Checking of operand types in ALGOL compilers*, BIT 5, (1965), 151–163.

NAUR P. [1966], *Proof of algorithms by general snapshots*, BIT 6, (1966). 310–316.

NIVAT M. [1972], *Langages algébriques sur la magma libre et sémantique des schémas de programmes*, Proc. Coll. an Automata. Formal languages and Programming, North-Holland Pub. Co., Amsterdam, (1972).

ORE O. [1943a], *Some studies on closure relations*, Duke Math. Journal 10, (1943), 761–785.

ORE O. [1943b], *Combinations of closure relations*, Ann. of Math., 44(1943), 514–533.

OSTROWSKI A. [1955], *Determination mit überwiegender haupt diagonale und die absolute konvergenz von linearen iterations prozessen*, Comm. Math. Helv. 30, (1955), 175–210.

PARK D. [1969], *Fixpoint induction and proofs of program properties*, Machine Intelligence, 5(1969), 59–78.

PAIR C. [1974], *Formalization of the notion of data, information and information structure*, Data base management, North-Holland Pub. Co., Amsterdam, (1974).

PASINI A. [1974], *Some fixed point theorems of the mappings of partially ordered*, Rend.

Sem. Mat. Univ. Padova., 51(1974), 167–177.

PELCZAR A. [1961], *On the invariant points of a transformation*, Ann. Polan. Math., 11(1961), 199–202.

PELCZAR A. [1971], *Remarks on commuting mappings in partially ordered spaces*, Zeszyty Nauk. Univ. Jagiello., Prace Mat., Zeszyt, 15(1971), 131–133.

PNUELI A. [1977], *The temporal logic of programs*, Proc. 18th Annual Symp. an Foundations of Computer Science, Providence, R.I., U.S.A., (Oct. 31–Nov. 2 1977), 46–57.

REMY J.L. [1974], *Structure d'information, formalisation des notions d'accès et de modification d'une donnée*, Thèse de 3ème cycle, Univ. de Nancy I, (1974).

RIEF J.H. & LEWIS H.R. [1977], *Symbolic evaluation and the global value graph*, Conf. Rec. of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles, Calif., U.S.A., (Jan. 1977), 104–118.

ROBERT F. [1974], *Contraction en norme vectorielle. Convergence d'itérations chaotiques pour des équations de point fixe à plusieurs variables*, Gatlingburgh VI Symp. on Num. Alg., (Dec. 1974). Linear Algebra and its Appl. 13, (1976). 19–35.

ROBERT F. [1976a], *Sur la transformation de Gauss-Seidel*, Séminaire d'Analyse Numérique n° 255, Laboratoire IMAG, Grenoble, (Nov. 1976).

ROBERT F. [1976b], *Convergence locale d'itérations chaotiques non linéaires*, Rapport de Recherche n° 58. Laboratoire IMAG, Grenoble, (Déc. 1976).

ROCKAFELLAR R.T. [1976], *Monotone operators and the proximal point algorithm*, SIAM J. Control and Optimization, 14(1976), 877–898.

ROSEN B.K. [1975], *Data flow analysis for procedural languages*, IBM Report RC-5211, T.J. Watson Research Center, Yorktown Heights, N.Y., U.S.A., (1975)[11].

ROSEN B.K. [1978], *Monoids for rapid data analysis*, Conf. Rec. of the 5th ACM Symp. on Principles of Programming Languages, Tucson, Ariz., U.S.A., (Jan.

[11] Journal of the ACM (JACM), Vol. 26 , Issue 2 (April 1979), 322–344, 1979

1978), 47–59.

SCARF H. [1967], *The approximation of fixed points of a continuous mapping*, SIAM J. Appl. Math., 15(1967), 1328–1343.

SCHAEFER M. [1973], *A mathematical theory of global program optimization*, Prentice Hall, Englewood Cliffs, N.J., U.S.A., (1973).

SCHWARTZ J.T. [1973], *On programming: an interim report on the SETL project installment 1. Generalities, installment 2. The SETL language and examples of its use*, New York Univ., (1973).

SCHWARTZ J.T. [1975], *Automatic data structure choice in a language of very high level*, CACM 18, 12(Dec. 1975), 722–728.

SCOTT D. [1972], *Continuous lattices*, Proc. 1971 Dalhousie Conf., Lect. Notes in Math. 274, Springer-Verlag, New York, 97–136.

SCOTT D. [1976], *Data types as lattices*, SIAM J. on Computing 5, 3(Sept. 1976), 522–587.

SCOTT D. [1977a], *1976 ACM Turing award lecture : logic and programming languages*, CACM 20, 9(Sept. 1977).

SCOTT D. [1977b], *Retracts*, Notes de cours, École IRIA "Séminaire Avancé de Sémantique", Sophia-Antipolis, (Oct. 1977).

SCOTT D. & STRACHEY C. [1971], *Toward a mathematical semantics for computer languages*, Proc. Symp. on Computers and Automata. Polytechnic Inst. of Brooklyn, Vol. 21, (1971), 19–46.

SINTZOFF M. [1972], *Calculating properties of programs by valuations on specific models*, Proc. ACM Conf. an Proving Assertions about Programs, SIGPLAN Notices 7, 1(1972), 203–207.

SINTZOFF M. [1975], *Vérification d'assertions pour des fonctions utilisables comme valeurs et affectant des variables extérieures*, Proc. Int. Symp. on Proving and Improving Programs, Arcs et Senans, (Juillet 1975), 11–27.

SINTZOFF M. [1976a]. *Eliminating blind alleys from backtrack programs.* 3rd Int. Coll. an Automata Languages and Programming, Edinburgh, (July 1976).

SINTZOFF M. [1976b]. *Iterative methods for the generation of successful programs.* Notes de travail, Lab. MBLE, Bruxelles, (Déc. 1976).

SINTZOFF M. [1977a]. *Inventing program construction rules.* IFIP W.G.2.4. Working Conf. on Constructing Quality Software. Novosibirsk. North-Holland Pub. Co., (May 1977).

SINTZOFF M. [1977b]. *Some logical construction rules for programs.* Notes de travail, CRI Nancy, (1977).

SINTZOFF M. & VAN LAMSWEERDE A. [1975], *Constructing correct and efficient concurrent programs,*. Proc. Int. Conf. an Reliable Software, SIGPLAN Notices, 10(1975), 319–326.

SMITHSON R.E. [1973], *Fixed points in partially ordered sets*, Pacific J. Math., 1(1973), 363–367.

SOUTHWELL R.V. [1955], *Relaxation methods in theoretical physics*, Clarendon Press, Oxford, (1946).

SPILLMAN T.C. [1971], *Exposing side effects in a PL/I optimizing compiler.* Proc. IFIP Congress 71, North-Holland Pub. Co, Amsterdam, (1971). 376–361.

STEIN P. & ROSENBERG R.L. [1946], *On the solution of linear simultaneous equations by iterations*, J. London Math. Soc., 1948 s1-23(2):111–118.

STOY J. [1977], *Denotational semantics*, MIT Press. (1977).

STRACHEY C. & WADSWORTH C. [1974], *Continuations: a mathematical semantics for handling full jumps.* Tech. Monograph PRG-11. Oxford U., Camp. Lab., Programming Research Group, (1974)[12].

SUZUKI N. & ISHIHATA K. [1977], *Implementation of an array bound checker*, Conf. Rec. of the 4th ACM Symp. an Principles of Programming Languages, Los Angeles, Calif., U.S.A., (Jan. 1977), 132–143.

[12] Higher-Order and Symbolic Computation, Vol. 13, Nb. 1/2, April 2000, 135–152

SZÁSZ G. [1971], *Théorie des treillis*, Dunod, Paris, (1971).

TARJAN R.E. [1974], *Testing flow graph reducibility*, J. Computer and Systems Sciences 9, 3(1974), 355–365.

TARJAN R.E. [1975], *Solving path problems on directed graphs*, Research Report STAN-CS-75-526. Computer Sci. Dept., Stanford U., Calif., U.S.A., (1975)[13].

TARJAN R.E. [1976], *Iterative algorithmes for global flow analysis*, Research Report STAN-CS-75-545, Computer Sci. Dept., Stanford U., Calif., U.S.A., (Feb. 1976)[14].

TARSKI A. [1955], *A lattice theoretical fixpoint theorem and its applications*, Pacific J. Math., 5(1955), 285–310.

TENENBAUM A.M. [1974], *Type determination for very high level languages*, Report NSD-3, Computer Sci. Dept., New York U., U.S.A., (Oct. 1974).

TENNENT R.D. [1976], *The denotational semantics of programming languages*, CACM 19, 8(1976), 437–453.

TODD M. [1976], *The computation of fixed points and applications*, Lecture Notes in Economics and Math. Systems 114, Springer-Verlag, Berlin, (1976).

TRAUB J.F. [1964], *Iterative methods for solutions of equations*, Prentice Hall, (1964).

ULLMAN J.D. [1974], *Fast algorithms for the elimination of common subexpressions*, Acta Informatica 2, 3(1974), 191–213.

ULLMAN J.D. [1975], *Data flow analysis*, Proc. 2nd USA-Japan Computer Conf., AFIPS Press, Montwale, N.J. (1975), 335–342.

URSCHLER G. [1974], *Complete redundant expression elimination in flow diagrams*, IBM Research Report RC-4965, T.J. Watson Research Center, Yorktown Heights, N.Y. (1974).

[13] See *Fast Algorithms for Solving Path Problems*, Journal of the ACM (JACM), Vol. 28 , Issue 3 (July 1981), 594–614.

[14] Algorithms and Complexity: New Directions and Recent Results, J.F. Traub, ed., Academic Press, New York, (1976), 71–102.

VAN LAMSWEERDE A. [1977], *From verifying termination to guaranteeing it : a case study*, IFIP Working Conf. on Formal Description of Programming concepts, St-Andrews, N.B., Canada, North-Holland Pub. Co., (Aug. 1977).

VAN LAMSWEERDE A. & SINTZDFF M. [1976], *Formal derivation of strongly correct parallel programs*, rapport R338, Lab. MBLE, Bruxelles, (1976)[15].

VUILLEMIN J.E. [1973], *Proof techniques for recursive programs*, Ph.D. Thesis, STAN-CS-73-393, Stanford U., Calif. (Oct. 1973).

WARD M. [1942], *The closure operators of a lattice*, Annals Math., 43(1942), 191–196.

WARD L.E. Jr. [1957], *Completeness in semi-lattices*, Canad. J. Math., 9(1957), 578–582.

WEGBREIT B. [1974], *The synthesis of loop predicates*, CACM 17, 2(Feb. 1974), 102–112.

WEGBREIT B. [1975], *Property extraction in well-founded property sets*, IEEE Trans. on Soft. Eng., SE-1, 3(Sept. 1975), 270–285.

WEGBREIT B. [1977], *Complexity of synthesizing inductive assertion*, JACM 24, 3(July 1977), 504–512.

WELSH J. [1977], *Economic range checking in PASCAL*, Dept. of Comp. Science, Queen's University, Belfast, Northern Ireland, (Oct. 1977)[16].

WIRTH N. [1971], *Programm development by stepwise refinement*, CACM 14, 4(April 1971), 221–227.

WIRTH N. [1976], *Programming languages: what to demand and how to assess them*, Symp. on Software Engineering, Belfast, (April 1976), [Aussi Rep. 17. Eidgenössische Technische Hochschule Zürich, Institut für Informatik].

WOLK E.S. [1957], *Dedekind completeness and a fixed point theorem*, Canad. J. Math., 9(1957), 400–405.

[15] Acta Informatica, Vol. 12, Nb 1, June, 1979, 1–31.

[16] Software: Practice and Experience, Vol.8, Issue 1, 85–97, 1978.

WONG J.S.W. [1967], *Common fixed point of commuting monotone mappings*, Canad. J. Math., 19(1967), 617–620.

8.1 BIBLIOGRAPHIC ADDENDUM

[1] R.P. Agarwal, M. Meehan, and D. O'Regan. Fixed point theory and applications. In *Cambridge Tracts in Mathematics*, volume 141. Cambridge University Press, Cambridge, United Kingdom, 2001.

[2] K.R. Apt and E.-R. Olderog. Proof rules dealing with fairness. In *Logic of Programs, Workshop, Lecture Notes in Computer Science 131*, pages 1–8. Springer, Berlin, Germany, 1981.

[3] J.M. Bahi, S. Contassot-Vivier, and R. Couturier. Evaluation of the asynchronous iterative algorithms in the context of distant heterogeneous clusters. *Parallel Comput.*, 31(5):439–461, May 2005.

[4] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, invited chapter. In T. Mogensen, D.A. Schmidt, and I.H. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, Lecture Notes in Computer Science 2566, pages 85–108. Springer, Berlin, Germany, 2002.

[5] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN '2003 Conference on Programming Language Design and Implementation (PLDI)*, pages 196–207, San Diego, California, United States, 7–14 June 2003. ACM Press, New York, New York, United States.

[6] T.S. Blyth and M.F. Janowitz. *Residuation Theory.* Pergammon Press, Oxford, United Kingdom, 1972.

[7] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The Astrée analyser. In M. Sagiv, editor, *Proceedings of the Fourteenth European Symposium on Programming Languages and Systems, ESOP '2005, Edinburg, Scotland*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, Berlin, Germany, 2–10 April 2005.

[8] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Varieties of static analyzers: A comparison with ASTRÉE, invited paper. In M. Hinchey, He Jifeng, and J. Sanders, editors, *Proceedings of the First IEEE & IFIP International Symposium on Theoretical Aspects of Software Engineering, TASE '07*, pages 3–17, Shanghai, China, 6–8 June 2007. IEEE Computer Society Press, Los Alamitos, California, United States.

[9] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the ASTRÉE static analyzer. In M. Okada and I. Satoh, editors, *Eleventh Annual Asian Computing Science Conference, ASIAN 06*, pages 272–300, Tokyo, Japan, 6–8 December 2006, 2008. Lecture Notes in Computer Science 4435, Springer, Berlin, Germany.

[10] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order, Second Edition.* Cambridge University Press, Cambridge, United Kingdom, 2002.

[11] J. Dugundji and A. Granas. *Fixed point theory.* Springer, Berlin, Germany, 2003.

[12] N. Francez. *Fairness.* Springer, Berlin, Germany, 1986.

[13] G.A. Gratzer, B.A. Davey, R. Freese, B. Ganter, M. Greferath, P. Jipsen, H.A. Priestley, H. Rose, E. T. Schmidt, S.E. Schmidt, F. Wehrung, and R. Wille. *General Lattice Theory.* Springer, Berlin, Germany, second edition, January 2003.

[14] D. Kozen K.R. Apt. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6):307–309, 1986.

[15] W.A. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, December 1992.

[16] Z. Li and M. Parashar. A decentralized computational infrastructure for grid-based parallel asynchronous iterative applications. *J. Grid Computing*, 4(4):355–372, December 2006.

[17] L. Mauborgne. ASTRÉE: Verification of absence of run-time error. In P. Jacquart, editor, *Building the Information Society*, chapter 4, pages 385–392. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2004.

[18] D. Monniaux. The parallel implementation of the ASTRÉE static analyzer. In *Proceedings of the Third Asian Symposium on Programming Languages and Systems, APLAS '2005*, pages 86–96, Tsukuba, Japan, 3–5 November 2005. Lecture Notes in Computer Science 3780, Springer, Berlin, Germany.

[19] D. Pataria. A constructive proof of Tarski's fixed-point theorem for DCPO's. *65th Peripatetic Seminar on Sheaves and Logic, Århus, Denmark*, November 1997. Reported by M.H. Escardó in "Joins in the frame of nuclei", *Applied Categorical Structures*, Vol. 11, Issue 2, Springer, Berlin, Germany, pp. 117-124, April 2003.

[20] J. Wei. Parallel asynchronous iterations of least fixed points. *Parallel Comput.*, 19(8):887–895, 1993.

CHAPTER 9.

INDEX

9. INDEX

CHAPTER 10.

TABLE OF CONTENTS

10. TABLE OF CONTENTS

6. SEMANTIC ANALYSIS OF RECURSIVE PROCEDURES