

UNIVERSITÀ DEGLI STUDI DI VERONA

Dipartimento di Informatica

Dottorato di Ricerca in Informatica


Ciclo XXII

Titolo della tesi di dottorato:

Exploiting Loop Transformations for the Protection of Software

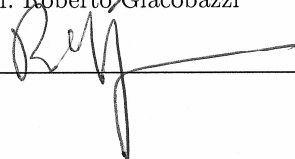
S.S.D: INF/01

Coordinatore: Prof. Luca Viganò



Tutor:

Prof. Roberto Giacobazzi



Dottorando: Dott. Enrico Visentini



Enrico Visentini

Exploiting Loop Transformations for the Protection of Software

May 20, 2010

Università degli Studi di Verona
Dipartimento di Informatica

Advisor:
prof. Roberto Giacobazzi

Series N°: **TD-09-10**

Università di Verona
Dipartimento di Informatica
Strada le Grazie 15, 37134 Verona
Italy

L'IMPERATORE

Non perderti! Non perderti, straniero!

LA FOLLA

È per la vita!

TIMUR (*disperatamente*)

È per la vita! Parla!

GIUSEPPE ADAMI and RENATO SIMONI, *Turandot*,
libretto for a music drama in three acts with a score by
Giacomo Puccini, premiered on April 25, 1926 at the
Teatro Alla Scala in Milan (Italy).

Summary. Software retains most of the know-how required for its development. Because nowadays software can be easily cloned and spread worldwide, the risk of intellectual property infringement on a global scale is high. One of the most viable solutions to this problem is to endow software with a watermark. Good watermarks are required not only to state unambiguously the owner of software, but also to be resilient and pervasive. In this thesis we base resiliency and pervasiveness on trace semantics. We point out loops as pervasive programming constructs and we introduce loop transformations as the basic block of pervasive watermarking schemes. We survey several loop transformations, outlining their underlying principles. Then we exploit these principles to build some pervasive watermarking techniques. Resiliency still remains a big and challenging open issue.

Contents

1	Introduction	1
	1.1 Securing Software	2
	1.2 This Thesis	3
2	Preliminaries	5
	2.1 Primitive Objects	5
	2.2 Logic	5
	2.3 Sets	6
	2.4 Relations	7
	2.5 Natural Numbers	8
	2.6 Integer Numbers	8
	2.7 Posets	8
	2.8 Functions	9
	2.9 Operations	10
	2.10 Strings	12
	2.11 Deterministic Automata	12
	2.12 Fixpoints	13
	2.13 Transition Systems	14
	2.14 Programming Language	14
	2.15 Abstract Interpretation	37
	2.16 A Framework for Program Transformations	40
	2.17 Deriving Semantic Transformers	43
3	Software Steganography	51
	3.1 A Grab of Software	51
	3.2 Protection of Software	52
	3.3 Code Obfuscation	53
	3.4 Software Watermarking	58
	3.5 Software Steganography by Abstract Interpretation	63
4	Loop Affine Transformations	67
	4.1 Loop Bumping	67
	4.2 Loop Reversal	68

VI Contents

4.3	Principle of Loop Affine Transformations	69
4.4	Loop Skewing	72
4.5	Normalization	74
4.6	Syntactic Transformer	74
4.7	Semantic Transformer	75
4.8	Local Commutation Condition	76
5	Loop Unrolling	81
5.1	Loop Peeling	81
5.2	Loop Splitting	83
5.3	Loop Unrolling	84
5.4	Principle of Loop Unrolling	88
5.5	Syntactic Transformer	98
5.6	Semantic Transformer	98
5.7	Local Commutation Condition	100
5.8	Folding	106
6	Hiding Software Watermarks in Looping Constructs	109
6.1	Steganographic Approach to Software Watermarking	109
6.2	Exposing Iterations for the sake of Watermarking	113
6.3	Encoding the Signature	114
6.4	Tracing Appropriate Locations for the Watermark	115
6.5	Embedding	115
6.6	Locating the Watermark	118
6.7	Extracting and Decoding the Watermark	118
6.8	Determining the Signature	120
6.9	Quantitative Evaluation	121
6.10	Attacks and Countermeasures	123
A	Tools	125
	References	131

Introduction

Hide and seek are not just the basis of child play, but also serious aspects of computer security and forensics. Making hard the access to information is a key aspect of most methods for information hiding since the beginning of human history. In a world where information is ubiquitous, there is an urgent need to prevent undesired observers from taking advantage of confidential data, as well as to sight malicious or even criminal data notwithstanding to which extent they are concealed.

Since the existence of information in the physical world involves a content, such as a message or a signature, spread on a physical support acting as a container or carrier [75], methods for information hiding accordingly group into two disciplines: *cryptology* addresses the content, making it meaningless to any observer unprovided of the suitable decipherment key; *steganography* addresses the observer's aptitude for the container, hiding the content inside the container in such a way that the observer, without a clue, is not able to notice it.

Regardless of their targets, both strategies attempt to overcome the natural skills of the observer, driving the complexity of the observation up so high that its cost is too high to be worth performing, even though it could eventually be successful. In the field of software protection, mainly in the early literature [15], this principle has been named *security through obscurity*: the observer succeeds in retrieving the concealed data only if it really knows, or it manages to guess by an unsystematic kind of intuition, the key piece of information useful at improving its observation ability or restricting the region it observes.

Almost surprisingly, in recent years the phrase itself has taken on a quite different meaning, as now it mainly refers to the design of systems whose security relies on secrecy of design details [116]. Here what is concealed is how contents and containers are dealt with. Even though this practice may act as a temporary "speed bump" for undesired spies, for instance while a resolution to a known security issue is implemented, it is severely frowned up. Design details invariably become known, especially in the case of commercially successful systems [44] or military systems.

Actually, this has been apparent since 1883, when Auguste Kerckhoffs [68] devised the well known security principle stipulating that the security of a system should not suffer if an adversary knows all the details of the system aside from

the secret key/clue.¹ Clearly security through obscurity, when intended in its later meaning, violates Kerckhoffs' principle. However, if stated in its original definition, not only it complies with the principle, but also it provides a solid foundation to cryptography, whose success is confirmed by its widespread use in modern communications, and to steganography, whose capabilities deserve deeper investigations.

1.1 Securing Software

High quality software is the result of an intellectual effort. Plenty of users can benefit from the result, but only if some producers make the effort. Producers – not users – are entitled to choose a business model for software. Unfortunately software products, or *programs* for short, have some features that issue several challenges to this assumption.

1. It is extremely easy to clone programs and make thousands of illegal copies out of one legally purchased program.
2. Because of a more and more interconnected global network, it is easier and easier to exchange copies of a program.
3. Programs – unlike manufactured goods – mostly retain in themselves the know-how required for their development.
4. Inspecting the content of a program can be simple, especially if the program is expressed in widespread programming languages, like Java or .NET.
5. It is easy to transform a program and embed (part of) it into another program.

It is well understood that the interaction of these five points opens the door to piracy and copyright infringement on a global scale. Yet it is not clear how to tackle them. The first two points seem to be connected to the technological infrastructure we exploit nowadays for storing and exchanging programs. The third point is inherent to the very nature of software. The last point stems from the fourth one, assuming that one cannot take advantage of what they cannot inspect. Therefore, to thwart piracy, we must try to make inspection unfeasible.

Cryptography can be a powerful tool in the protection of software, as encrypted programs are difficult, if not impossible, to inspect. But they can be executed only in decrypted form. At runtime encryption thereby fails in protecting programs. Steganography instead is not prone to this inconvenience. Consider for instance obfuscation, which is a typical application of steganography to software: *obfuscation* makes the control flow or the data structures of a program harder to analyze, but it preserves both functionality and executability [23]. So far, however, very few provable secure obfuscation schemes are known [120]. If a copyright infringement is likely to take place, *watermarking* can be of help in its detection. A watermark is additional information one embeds in a program to prove ownership [22]. A comment with a copyright notice is a first example of watermark. Unfortunately this kind of watermarks, being totally taken apart from functionality, are easy to distort or remove. A strong connection should exist between a watermark and the

¹ Originally from [68]: « *Il mafaut qu'il [= le système] n'exige pas le secret, et qu'il puisse sans inconvénient tomber entre les mains de l'ennemi.* »

functionality of a program, so that you cannot damage one while preserving the other.

1.2 This Thesis

We use sets of traces to detail the functionality of a program [28]. A program P is a set of statements which can either test or change the value of a set of variables. Values are stored in environments and commands can be thought as transformers of environments. A trace is just a sequence of environments punctuated by statements. The functionality, or *semantics*, of P is the set $\mathcal{S}[[P]]$ of all traces whose statements are the statements of P .

The fact that comments play no role in the determination of a trace accounts for the weakness of comment-based watermarks. In order to contribute to the definition and the implementation of good watermarks, let us make some claims.

1. Good watermarks should be pervasive: the more pervasive a watermark, the higher the number of environments in the traces of $\mathcal{S}[[P]]$ that should be affected by the embedding of the watermark.
2. Good watermarks should be resilient. In our view, resilience implies that you cannot derive the original semantics of P from the watermarked program. We believe that resilience requires the embedding of the watermark to entail a loss of information. Such information is called the *key*. Because of the loss, when you try to characterize the functionality of the watermarked program, you no longer get its semantics, but an over-approximation describing behaviors and properties that the watermarked program actually has not. This thwarts the recognition of the original functionality and, consequently, the detection of the watermark.
3. The key should carry the information that tells good traces from misleading ones. Also, the key should enable the detection of the watermark within the good traces. Of course, the key should be held only by the people who are entitled to extract the watermark.
4. Resilience, in its full meaning, entails that if one tries to distort the watermark or remove it without knowing the key, they should obtain a new program whose functionality is radically different from the functionality of the original program.

A good pervasive watermarking scheme is the dynamic path-based one [16], which takes advantage of branching constructs. Branching not only is ubiquitous in non-trivial programs, but it strongly affects evolution of traces. Also looping constructs can be of interest. A widely held rule of thumb is that a program spends 90% of its execution time in only 10% of the code [60]. This critical 10% of the code frequently consists of loops [41]. We thereby expect 90% of a trace to be determined by commands involved in loops. Hence, we also expect loop transformations to be the basic blocks for the design of pervasive watermarking schemes.

In this thesis, we first provide the preliminary details for the comprehension of our work (Chapter 2). After introducing the mathematical background, we take a long time (Section 2.14) describing the syntax and the formal trace-based semantics

of an imperative programming language introduced by Cousot and Cousot [33]. As a novel contribution, we enrich such language with subroutines and for-loops; we also introduce a semantics-based definition of iteration and several related notions. Then we present the Cousots' abstract interpretation theory (Section 2.15) and the Cousots' proposal for the systematic design of program transformation in a framework based on abstract interpretation (Section 2.16). We derive a novel variant to that framework (Section 2.17) which we exploit later to model loop transformations.

Next, we survey obfuscation and watermarking concepts, techniques and terminology (Chapter 3).

After that, we explore several loop transformations which are well known from the literature. By recasting each transformation in our trace-based semantic framework, we point out that they ultimately shift information from environments to statements. Depending on the way they implement such shift, we partition them into two classes. We call *loop affine transformations* the members of first class (Chapter 4); we show that they are based on affinities; since affinities are reversible, they can be easily undone by means of themselves. The members of the second class (Chapter 5) are loop peeling, loop splitting and loop unrolling; we show that peeling and splitting are instances of unrolling; we prove that unrolling provides an effective way to make semantic information explicit (Section 5.8).

Finally, we take advantage of the latter class of transformations to introduce (Chapter 6) our novel watermarking technique [38], which we presented at the 15th Static Analysis Symposium (SAS), held in Valencia (Spain) on July 2009. Our proposal is still at an embryonic stage and is meant to be a first attempt at the instantiation of our claims. Unfortunately, resilience proved to be a very hard issue to deal with; thus we failed in properly address our last claim. What we gained however is a comprehension of loops as media which can hide and disclose semantic information.

Preliminaries

This Chapter introduces the basics of logic, the notions of set theory and, more in general, the formal concepts we take advantage later to express and clarify our thesis. We just extend the natural language to deal with mathematical objects without ambiguities. The survey we present here can be used for reference in the following Chapters.

2.1 Primitive Objects

We assume the existence of primitive objects as *elements* and *collections* of elements. Examples of elements are **true**, **false**, the natural numbers (0, 1, 2...), the integer numbers (0, 1, -1, 2, -2...), the letters of the alphabet (a, b, c...) and so on. If a and b are the same element, we write $a = b$.

By the *axiom of extensionality*, a collection is determined only through its elements and only after its elements have been determined.

Let a be an element and U be a collection. If a is an element collected in U we write $a \in U$, otherwise we write $a \notin U$. If both a and b are elements of U , we can write $a, b \in U$. If all the elements of U are at the same time elements of another collection V , we write $U \subseteq V$. Obviously $U \subseteq U$. We have that U and V are the same collection if $U \subseteq V$ and $V \subseteq U$; in such case we write $U = V$.

2.2 Logic

A *logic formula* is an assertion about elements or collections. A formula can be either **true** or **false**. A formula A can include *variables*, which are placeholders for elements. If A includes variable x , we can write $A(x)$. Then $A(x)$ is **true** or **false** depending on the element which x is standing for.

If φ and ψ are logic formulas, then from φ and ψ we can derive other logic formulas, namely: the *negation* of φ , noted $\neg\varphi$; the *conjunction* of φ and ψ , noted $\varphi \wedge \psi$; the *disjunction* of φ and ψ , noted $\varphi \vee \psi$; the *logic implication* between φ and ψ , noted $\varphi \implies \psi$; the *logic equivalence* of φ and ψ , noted $\varphi \iff \psi$. Symbol \neg has the highest priority, whereas \iff has the lowest. We use parentheses to

establish absolute priorities. We note the *universal quantifier* as \forall ; if $\forall x. \varphi(x)$ is true, then $\varphi(x)$ is true whichever is the element x stands for. The *existential quantifier* is noted \exists ; if $\exists x. \varphi(x)$ is true, then $\varphi(x)$ is true if and only if there is some element c such that $\varphi(c)$ is true.

2.3 Sets

We call *set* any collection that can be treated as an element. We assume that the collection which does not include any element is a set, called the *empty set* and noted \emptyset ; moreover, any finite collection is a (finite) set. Notice that $\emptyset \subseteq U$ for all collections U .

We write $\{a, b, c\}$ to note a set with three elements a , b and c . When we use the brace notation, we are not concerned with the ordering of the elements, nor with the number of occurrences of an element within the braces. Thus, for instance, $\{a, b, c\}$ or $\{b, c, a\}$ or $\{a, b, c, a\}$ denote the same set. If we want U to be the set including just a , b and c , we write $U \stackrel{\text{def}}{=} \{a, b, c\}$. An *ordered pair* with left element a and right element b is noted $(a, b) \stackrel{\text{def}}{=} \{\{a\}, \{a, b\}\}$. Pairs can be easily generalized to so-called *tuples*, which have more than two elements.

Let A , B and C sets.

- Any collection U such that $U \subseteq A$ is a set; in particular, U is termed as a *subset of A* .
- If $\varphi(x)$ a formula, the collection of all $a \in A$ such that $\varphi(a)$ is true is a set, noted either $\{a \mid a \in A \wedge \varphi(a)\}$ or $\{a \in A \mid \varphi(a)\}$.
- The collection including all the subsets of A is a set, noted $\wp(A) \stackrel{\text{def}}{=} \{U \mid U \subseteq A\}$.
- The collection including all the finite subsets of A is a set, noted $\wp_{\text{finite}}(A) \stackrel{\text{def}}{=} \{U \mid U \subseteq A\}$.
- If for each $a \in A$ there is an element noted b_a , then the collection of all such elements is a set, noted $\{b_a \mid a \in A\}$.
- The collection encompassing all the elements included in A but not in B is a set, noted $A \setminus B \stackrel{\text{def}}{=} \{a \mid a \in A \wedge a \notin B\}$.
- The collection encompassing all the elements included in A and in B is a set, noted $A \cap B \stackrel{\text{def}}{=} \{a \mid a \in A \wedge a \in B\}$.
- The collection encompassing all the elements included in every set $B \in A$ is a set, noted $\bigcap A \stackrel{\text{def}}{=} \{b \mid \forall B \in A. b \in B\}$; in case $A = \{B_c \mid c \in C\}$, then $\bigcap A$ is also noted $\bigcap_{c \in C} B_c$.
- The collection encompassing all the elements included in A or in B is a set, noted $A \cup B \stackrel{\text{def}}{=} \{a \mid a \in A \vee a \in B\}$.
- The collection encompassing all the elements included in some set $B \in A$ is a set, noted $\bigcup A \stackrel{\text{def}}{=} \{b \mid \exists B \in A. b \in B\}$; in case $A = \{B_c \mid c \in C\}$, then $\bigcup A$ is also noted $\bigcup_{c \in C} B_c$.
- The collection including every ordered pair with left element $a \in A$ and right element $b \in B$ is a set, noted $A \times B \stackrel{\text{def}}{=} \{(a, b) \mid a \in A \wedge b \in B\}$.
- A set $P \subseteq \wp(A)$ is a *partition of A* if and only if:
 - (i) $\emptyset \notin P$;
 - (ii) $\bigcup P = A$;

$$(iii) \forall B \in P. \forall C \in P. B \neq C \implies B \cap C \neq \emptyset.$$

2.4 Relations

An element $R \in \wp(A \times B)$ is a *binary relation between A and B*. Thus $R \subseteq A \times B$. We write $a R b$ if and only if $(a, b) \in R$, otherwise we write either $\neg(a R b)$ or $a \not R b$. If $R \in \wp(A \times A)$ then we say R is a *binary relation on A*.

A relation R on A is:

- *reflexive* if and only if

$$\forall a \in A. a R a ;$$

- *symmetric* if and only if

$$\forall a \in A. \forall b \in A. a R b \implies b R a ;$$

- *antisymmetric* if and only if

$$\forall a \in A. \forall b \in A. a R b \wedge b R a \implies a = b ;$$

- *transitive* if and only if

$$\forall a \in A. \forall b \in A. \forall c \in A. a R b \wedge b R c \implies a R c .$$

Equivalence Relations

A reflexive, symmetric and transitive relation is an *equivalence relation*, noted \equiv or \sim or \simeq . If \equiv is an equivalence relation on A , any set

$$[b]_{\equiv} \stackrel{\text{def}}{=} \{ a \in A \mid a \equiv b \}$$

is an *equivalence class of b with respect to \equiv* . It has been proved that

$$A/\equiv \stackrel{\text{def}}{=} \{ [b]_{\equiv} \mid b \in A \}$$

is a partition on A ; on the other side, any partition P on A is related to an equivalence relation whose classes are the elements of P .

Let \equiv and \equiv' be two equivalence relations on A . We say that \equiv is a *refinement of \equiv'* if and only if every class in A/\equiv is a subset of some class in A/\equiv' .

Order Relations

A reflexive, antisymmetric and transitive relation is an *order relation*, noted \leq or \sqsubseteq or \preceq . An order relation \leq on A establishes a *partial ordering* on A . Such ordering is *total* if and only if $\forall a \in A. \forall b \in A. a \leq b \vee b \leq a$.

Let $a, b \in A$. If $a \leq b$ and $a \neq b$, we can write $a < b$.

2.5 Natural Numbers

The collection of all natural numbers is assumed to be a set, noted \mathbb{N} . A *natural number* is either the empty set \emptyset or a set $\text{succ}(A)$ such that $A \in \mathbb{N}$ and $\text{succ}(A) \stackrel{\text{def}}{=} A \cup \{A\}$. Each element in \mathbb{N} has its own notation. Here below we report some elements of \mathbb{N} and their correspondent notation:

\emptyset	0
$\{\emptyset\}$	1
$\{\emptyset, \{\emptyset\}\}$	2
$\{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}$	3
\vdots	\vdots

A generic element of \mathbb{N} is usually noted n .

An order relation \leq is defined on \mathbb{N} such that, given $n, n' \in \mathbb{N}$, $n \leq n'$ if and only if $n \subseteq n'$. We introduce special notations for some subsets of \mathbb{N} , namely:

$$\begin{aligned} (n, n') &\stackrel{\text{def}}{=} \{n'' \mid n < n'' < n'\} \\ [n, n') &\stackrel{\text{def}}{=} (n, n') \cup \{n\} \\ (n, n'] &\stackrel{\text{def}}{=} (n, n') \cup \{n'\} \\ [n, n'] &\stackrel{\text{def}}{=} (n, n') \cup \{n, n'\} . \end{aligned}$$

2.6 Integer Numbers

Let \sim be a relation on $\mathbb{N} \times \mathbb{N}$ such that, given $(a, b) \in \mathbb{N} \times \mathbb{N}$ and $(c, d) \in \mathbb{N} \times \mathbb{N}$, we have that $(a, b) \sim (c, d)$ if and only if $a + d = b + c$. It can be easily proved that \sim is an equivalence relation. An element in $\mathbb{N} \times \mathbb{N} / \sim$ is called an *integer number*. It can be proved that $\mathbb{N} \times \mathbb{N} / \sim = \{[(0, 0)]_{\sim}\} \cup \{[(0, n)]_{\sim} \mid n \in \mathbb{N}\} \cup \{[(n, 0)]_{\sim} \mid n \in \mathbb{N}\}$. We note $[(0, 0)]_{\sim}$ with 0 and, given $n \in \mathbb{N}$, we note $[(0, n)]_{\sim}$ with $-n$ and $[(n, 0)]_{\sim}$ with $+n$ or n . We refer to $\mathbb{N} \times \mathbb{N} / \sim$ as the *set of the integer numbers*, noted \mathbb{Z} . An element in \mathbb{Z} is noted z .

An order relation \leq is defined on \mathbb{Z} such that, given $[(a, b)]_{\sim}, [(a, b)]_{\sim} \in \mathbb{Z}$, $[(a, b)]_{\sim} \leq [(a, b)]_{\sim} \in \mathbb{Z}$ if and only if $a + d \leq b + c$, where \leq in the latter assertion is the order relation on \mathbb{N} .

2.7 Posets

A set A endowed with a partial ordering \leq is a *partial ordered set*, or *poset* for short, and is noted $\langle A, \leq \rangle$, or $\langle A, \leq_A \rangle$ to stress the fact that \leq is an ordering on A . An example of poset is the set of natural numbers endowed with its canonical order, which in fact is a total order; such poset is noted $\langle \mathbb{N}, \leq \rangle$.

Let $\langle A, \sqsubseteq \rangle$ be a poset and let $B \subseteq A$ and $a \in A$.

- a is an *upper bound* of B if and only if $\forall b \in B. b \sqsubseteq a$;
- a is the *maximum* of B , noted \top_B or $\max B$, if and only if a is an upper bound of B and $a \in B$;
- a is the *least upper bound* of B , noted $\bigsqcup B$ (or $b \sqcup b'$ when $B = \{b, b'\}$), if and only if it is the minimum among the upper bounds of B ;
- a is a *lower bound* of B if and only if $\forall b \in B. a \sqsubseteq b$;
- a is the *minimum* of B , noted \perp_B or $\min B$, if and only if a is a lower bound of B and $a \in B$;
- a is the *greatest lower bound* of B , noted $\bigsqcap B$ (or $b \sqcap b'$ when $B = \{b, b'\}$), if and only if it is the maximum among the lower bounds of B .

A function $f : \mathbb{N} \rightarrow A, n \mapsto a_n$ is a ω -*chain* of A if and only if $\forall n, n' \in \mathbb{N}. n \leq n' \implies a_n \sqsubseteq a_{n'}$.

A poset $\langle A, \sqsubseteq \rangle$ is:

- a *lattice* if and only if $a \sqcup a' \in A$ and $a \sqcap a' \in A$ for all $a, a' \in A$;

Moreover it is:

- a *complete partial order* if and only if the range of each ω -chain in A has a least upper bound, noted $\bigsqcup_{n \in \mathbb{N}} a_n$.
- a *complete lattice* if and only if $\forall B \subseteq A. \bigsqcup B \in A$ if and only if $\forall B \subseteq A. \bigsqcap B \in A$.

It has been proved that a complete lattice has both minimum and maximum; moreover, any complete lattice is a complete partial order, but not vice versa. Given a set B , an example of complete lattice is $\langle \wp(B), \subseteq \rangle$, whose minimum is $\bigcap \wp(B) = \emptyset$ and maximum is $\bigcup \wp(B) = B$.

2.8 Functions

A relation $f \subseteq A \times B$ is a *function* from A to B , noted $f: A \rightarrow B$, if and only if $a f b$ and $a f b'$ entail $b = b'$. We can note the function as $\lambda a \in A. f(a)$, when B is clear from the context. If A is clear as well, we can even write $\lambda a. f(a)$. We usually note $a f b$ as either $f: a \mapsto b$ or $b = f(a)$ and we say that f is *defined* in a . If there is no $b \in B$ such that $f(a) = b$, we say that f is *undefined* in a . Because f is allowed to be undefined for some elements, it establishes a *partial* mapping between A , its *domain*, and B , its *range*.

A function $f: A \rightarrow B$ is:

- *total* if and only if

$$\forall a \in A. \exists b \in B. f(a) = b ;$$

- *surjective* if and only if

$$\forall b \in B. \exists a \in A. f(a) = b ;$$

- *injective* if and only if

$$\forall a, a' \in A. f(a) = f(a') \implies a = a' ;$$

- *bijjective* if and only if it is surjective and injective.

An example of bijective function is the *identity on A*, noted $\text{id}_A: A \rightarrow A$, which maps each $a \in A$ to itself. If U is a collection and there exists a bijection between U and \mathbb{N} then U is a set.

If $f: A \rightarrow B$ and $g: B \rightarrow C$ are functions, then their *composition* is a function $g \circ f: A \rightarrow C$ which maps $a \in A$ to $c \in C$ if and only if some $b \in B$ exists such that $f(a) = b$ and $g(b) = c$. Then we can write either $(g \circ f)(a) = c$ or $g(f(a)) = c$.

If $f: A \rightarrow B$ is total and bijective, then there exists a function $g: B \rightarrow A$ such that $g \circ f = \text{id}_A$. Function g is the *inverse of f* and is noted f^{-1} .

2.9 Operations

Let A be a nonempty set. A function f from $A \times A$ to A is an *operation on A*. An operation on A mapping (a, a') to a'' is customarily written with *infix notation* $a f a' = a''$ rather than *prefix notation* $f(a, a') = a''$. Moreover:

- f is *associative* if and only if $(a f a') a'' = a f (a' f a'')$;
- f is *commutative* if and only if $a f a' = a' f a$;
- f has an *identity element* or *neutral element* e if and only if for all $a \in A$, $a f e = e f a = a$.

If A is a set and f an associative operation on A , then A, f is a *semigroup*.

Arithmetic Operations

Let $n, n', n'' \in \mathbb{N}$. We have two basic operations on \mathbb{N} :

- *sum* or *addition*, noted $+$, is such that:

$$\begin{aligned} n + 0 &\stackrel{\text{def}}{=} n \\ n + \text{succ}(n') &\stackrel{\text{def}}{=} \text{succ}(n + n') \quad ; \end{aligned}$$

- *product* or *multiplication*, noted \cdot or \times , is such that:

$$\begin{aligned} n \cdot 0 &\stackrel{\text{def}}{=} 0 \\ n \cdot \text{succ}(n') &\stackrel{\text{def}}{=} n + (n \cdot n') \quad . \end{aligned}$$

Notice that \cdot can be omitted if no ambiguities arise. It can be proved that $n' \leq n$ if and only if there exists an only natural number n'' such that $n' + n'' = n$. Since n'' is unique, we can define

- *subtraction* or *difference*, noted $-$, which is such that:

$$n - n' \stackrel{\text{def}}{=} n'' \text{ if and only if } n' + n'' = n \quad .$$

Then we can note n'' as $n - n'$. We also define:

- *corrected subtraction*, noted \ominus , which is such that:

$$n \ominus n' \stackrel{\text{def}}{=} \begin{cases} n - n' & \text{if } n' \leq n \\ 0 & \text{otherwise.} \end{cases} \quad (2.1)$$

It can be proved that if $n, n' \in \mathbb{N}$ and $n' > 0$, then there exists an only pair (q, r) of natural numbers such that $n = n'q + r$ and $0 \leq r < n'$. Since (q, r) is unique, we can define

- *division* or *quotient*, noted div , which is such that:

$$n \text{ div } n' \stackrel{\text{def}}{=} q ;$$

- *modulo* or *remainder*, noted mod , which is such that:

$$n \text{ mod } n' \stackrel{\text{def}}{=} r ;$$

- a special function, noted \cdot , which is such that:

$$n \cdot n' \stackrel{\text{def}}{=} (n \text{ div } n') \cdot n' .$$

We can easily prove that

$$n \cdot n' + n \text{ mod } n' = n . \quad (2.2)$$

Integer Operations

Given $[(a, b)]_{\sim}, [(c, d)]_{\sim} \in \mathbb{Z}$, we have two basic operations on \mathbb{Z} :

- *sum* or *addition*, noted $+$, such that:

$$[(a, b)]_{\sim} + [(c, d)]_{\sim} \stackrel{\text{def}}{=} [(a + c, b + d)]_{\sim} ,$$

where symbol $+$ on the left hand side stands for the addition in \mathbb{Z} , whereas symbol $+$ on the right hand side stands for the addition in \mathbb{N} ;

- *product* or *multiplication*, noted \cdot or \times , such that:

$$[(a, b)]_{\sim} \cdot [(c, d)]_{\sim} \stackrel{\text{def}}{=} [(ac + bd, ad + bc)]_{\sim} .$$

We can omit \cdot if no ambiguities arise. We can also define

- *subtraction* or *difference*, noted $-$, which is a function such that:

$$[(a, b)]_{\sim} - [(c, d)]_{\sim} \stackrel{\text{def}}{=} [(a, b)]_{\sim} + [(d, c)]_{\sim} .$$

2.10 Strings

An *alphabet* is a set Σ whose elements are named *symbols*. A *string* or *sequence* σ on Σ is a function $\lambda n \in [0, N)$. σ_n where $N \in \mathbb{N}$ and, for all $n \in [0, N)$, $\sigma_n \in \Sigma$. We usually consider σ just a train of symbols, rather than a function. For instance, if $a, b, c \in \Sigma$, then $abac$ is a string. If $\sigma = cba$, then $\sigma_0 = c$, $\sigma_1 = b$ and $\sigma_2 = a$. The *length* of σ is $|\sigma| \stackrel{\text{def}}{=} N$. The *empty string* ε is such that $|\varepsilon| = 0$.

The set of all strings on Σ is

$$\Sigma^* \stackrel{\text{def}}{=} \{ \sigma \mid \forall n \in [0, |\sigma|). \sigma_n \in \Sigma \wedge |\sigma| \in \mathbb{N} \} .$$

An operation is defined on Σ^* that maps (σ, σ') to $\sigma\sigma'$ such that

$$\sigma\sigma' \stackrel{\text{def}}{=} \lambda n \in [0, |\sigma| + |\sigma'|). \begin{cases} \sigma_n & \text{if } n \in [0, |\sigma|) \\ \sigma'_{n-|\sigma|} & \text{if } n \in [|\sigma|, |\sigma| + |\sigma'|) \end{cases}$$

Such operation is named *concatenation*. If $\sigma = a\sigma' \in \Sigma^*$, then its first symbol is a , noted $\vdash\sigma$. If $\sigma = \sigma'b \in \Sigma^*$, then its last symbol is b , noted $\dashv\sigma$. More formally:

$$\vdash: \Sigma^* \rightarrow \Sigma, a\sigma' \mapsto a \quad (2.3)$$

$$\dashv: \Sigma^* \rightarrow \Sigma, \sigma'b \mapsto b . \quad (2.4)$$

If $\sigma\sigma' \in \Sigma^*$, then σ is a *prefix* of $\sigma\sigma'$, whereas σ' is a *suffix* of $\sigma\sigma'$. It is easy to prove that the neutral element of concatenation is ε . Furthermore Σ^* endowed with concatenation is a semigroup.

Any $L \subseteq \Sigma^*$ is a *formal language*, or *language* for short.

2.11 Deterministic Automata

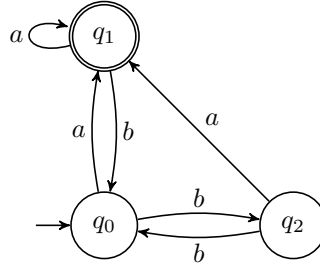
A *deterministic finite state automaton* is defined by a tuple $M = (Q, \Sigma, \delta, q_0, F)$, where:

- Q is a finite set of *states*;
- Σ is an alphabet;
- $\delta: Q \times \Sigma \rightarrow Q$ is the *transition function*;
- q_0 is the *initial state*;
- $F \subseteq Q$ is the set of *final* or *accepting states*.

We use graphs to represent automata. For instance, suppose $Q \stackrel{\text{def}}{=} \{ q_0, q_1, q_2 \}$, $\Sigma \stackrel{\text{def}}{=} \{ a, b \}$, $F \stackrel{\text{def}}{=} \{ q_1 \}$ and that δ is defined according to the following mapping:

$$\begin{aligned} (q_0, a) &\mapsto q_1 \\ (q_0, b) &\mapsto q_2 \\ (q_1, a) &\mapsto q_1 \\ (q_1, b) &\mapsto q_0 \\ (q_2, a) &\mapsto q_1 \\ (q_2, b) &\mapsto q_0 . \end{aligned}$$

If the initial state is q_0 , we can illustrate this automaton as



From δ we can derive a function $\hat{\delta}: Q \times \Sigma^* \rightarrow Q$ such that:

$$\begin{aligned} \hat{\delta}(q, \varepsilon) &\stackrel{\text{def}}{=} q \\ \hat{\delta}(q, \sigma a) &\stackrel{\text{def}}{=} \delta(\hat{\delta}(q, \sigma), a) . \end{aligned}$$

A string σ is *accepted by M* if and only if $\hat{\delta}(q_0, \sigma) \in F$. The *language accepted by M* is thereby

$$L(M) \stackrel{\text{def}}{=} \left\{ \sigma \in \Sigma^* \mid \hat{\delta}(q_0, \sigma) \in F \right\} .$$

2.12 Fixpoints

A poset $\langle A, \leq \rangle$ is sometimes thought as collecting data ordered by their degree of informativeness. Thus $a \leq b$ implies an informational gap between a and b . In some contexts it means that a is less informative than b , so \perp_A carries the smallest amount of information. In other contexts it means that b is an approximation of a , so \perp_A is the most informative element in A .

Functions between posets can be thought as information transformers. A function $f: \langle A, \leq \rangle \rightarrow \langle B, \sqsubseteq \rangle$ is:

- \perp_A -*strict* if and only if

$$f(\perp_A) = \perp_B ;$$

- an *order morphism* or, in other words, *monotonic* if and only if

$$\forall a, a' \in A. a \leq a' \implies f(a) \sqsubseteq f(a') ;$$

- *Scott-continuous* if and only if it is monotonic and for all ω -chains $\lambda n \in \mathbb{N}. a_n$,

$$f\left(\bigvee_{n \in \mathbb{N}} a_n\right) = \bigvee_{n \in \mathbb{N}} f(a_n) ;$$

- *additive* if and only if it is monotonic and for all $C \subseteq A$,

$$f\left(\bigvee C\right) = \bigsqcup \{ f(c) \mid c \in C \} .$$

Let $\langle A, \sqsubseteq \rangle$ be a poset. If f is a function on A , it is also called an *operator on A*. Suppose $\langle A, \sqsubseteq \rangle$ is a complete partial order with minimum \perp and f a monotonic operator on A . If $b \in A$ is such that $f(b) = b$, then b is a *fixpoint* of f . The minimum

among the fixpoints of f is called *the least fixpoint of f* and is noted $\text{lfp}_{\perp}^{\square} f$, or $\text{lfp} f$ for short. Given $a \in A$, the *transfinite iterates of f from \perp* are defined as:

$$\begin{aligned} f^0(a) &\stackrel{\text{def}}{=} \perp \\ f^{n+1}(a) &\stackrel{\text{def}}{=} f(f^n(a)) \\ f^{\omega}(a) &\stackrel{\text{def}}{=} \bigsqcup_{n \in \mathbb{N}} f^n(a) . \end{aligned}$$

Through his famous fixpoint theorem (reported in [124]), Alfred Tarski proved that $\text{lfp}_{\perp}^{\square} f = f^{\omega}(\perp)$, that is,

$$\text{lfp}_{\perp}^{\square} f = \bigsqcup_{n \in \mathbb{N}} f^n(\perp) .$$

2.13 Transition Systems

A *transition system* [58] is a triple $\langle \Sigma, \mathcal{J}, \rightarrow \rangle$, where:

- Σ is a set of *states*;
- $\mathcal{J} \subseteq \Sigma$ is a set of *initial states*;
- $\rightarrow \subseteq \Sigma \times \Sigma$ is a *transition relation*, where $s \rightarrow s'$ asserts that state s may transition to state s' .

A finite transition sequence, or *finite trace*, is a sequence of states s_0, s_1, \dots, s_n , such that $s_0 \in \mathcal{J}$ and $s_i \rightarrow s_{i+1}$ for every $0 \leq i < n$. The set of all finite traces of a transition system is

$$\Sigma^{\rightarrow} \stackrel{\text{def}}{=} \text{lfp}_{\emptyset}^{\subseteq} F ,$$

where F is an operator on Σ^* that make sequences grow one state longer. More formally:

$$F(\mathcal{J}) \stackrel{\text{def}}{=} \mathcal{J} \cup \{ \sigma s s' \mid \sigma s \in \mathcal{J} \wedge s \rightarrow s' \} .$$

All finite traces in $\Sigma^{\rightarrow} \subseteq \Sigma^*$ are said to be *partial*. A finite trace is *maximal* if and only if $\sigma_n \not\rightarrow s$ for all $s \in \Sigma$.

An infinite transition sequence, or *infinite trace*, is a sequence of states $s_0, s_1, s_2 \dots$ such that $s_0 \in \mathcal{J}$ and for every $i \in \mathbb{N}$ there exists $s_{i+1} \in \Sigma$ such that $s_i \rightarrow s_{i+1}$.

2.14 Programming Language

We consider a simple imperative language, whose syntax is presented in Table 2.1. Such language is certainly not standard and may be not immediately intuitive. It has been introduced by Cousot and Cousot [33] to outline in the abstract interpretation theory (see Section 2.15) a framework for the systematic development of program transformations (see Section 2.16). Since we derive a variant to that framework (see Section 2.17) in order to model loop transformations in the next chapters, Cousot's language shows to us as the most natural choice.

Integers	$z \in \mathbb{Z}$
Variables	$x \in \mathbb{X}$
Arithmetic Expressions	$A \in \mathbb{A}, A ::= z \mid x \mid A \cdot A$
Boolean Expressions	$B \in \mathbb{B}, B ::= \text{true} \mid A \approx A \mid \neg B \mid B \star B$
Commands	$C \in \mathbb{C}, C ::= B \mid x := A \mid x := ?$
Labels	$\ell \in \mathbb{L} \stackrel{\text{def}}{=} \{a, b, \dots, z, 0, 1, \dots, 9\}^*$
Statements	$S \in \mathbb{S}, S ::= \ell: C \rightarrow \ell$
Programs	$P \in \mathbb{P} \stackrel{\text{def}}{=} \wp_{\text{finite}}(\mathbb{S})$

Table 2.1. Syntax.

Integer numbers, variables, and arithmetic¹ and boolean expressions are defined as it is customary for common programming languages. Notice the symbols we use in compound expressions:

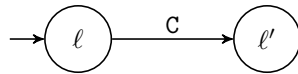
- symbol \cdot stands for any binary arithmetic operation;
- symbol \approx stands for any binary relation between integers;
- symbol \neg is the customary boolean negation;
- symbol \star stands for any binary boolean connective.

Operations, relations and connectives whatsoever follow the customary precedence rules. Parentheses may be used to avoid confusion.

A command $C \in \mathbb{C}$ is a boolean test (B), a deterministic assignment ($x := A$) or a random assignment ($x := ?$).

A label $\ell \in \mathbb{L}$ is a string of symbols chosen from the English alphabet or the Arabic numerals. The set \mathbb{L} of all labels is closed by concatenation.

Commands are combined with labels to build up statements. In particular, given a command $C \in \mathbb{C}$ and two labels $\ell, \ell' \in \mathbb{L}$, a statement is a transition from ℓ to ℓ' through C , noted $\ell: C \rightarrow \ell'$. A statement can also be thought as a small automaton



which transitions from initial state ℓ to state ℓ' by reading symbol C . Any command C' referenced by ℓ' is candidate for being read after C . In the automaton representation, this corresponds to let ℓ' be the initial state of a transition $\ell': C' \rightarrow \ell''$, where $\ell'' \in \mathbb{L}$.

Programs are finite sets of statements. Any variable appearing in a program is global. By kind of example, we consider a program P that sets an integer variable x to an arbitrary value, and reduces it to zero in case it is positive. If we used Java [54], we could write this program:

```
class P {
    public static int x;

    public static void main(String args[]) {
```

¹ With a slight abuse of language, we adopt the word *arithmetic* even if we deal with integer numbers. In fact arithmetic is concerned with natural numbers.

```

    java.util.Random r = new java.util.Random();
    for (x = r.nextInt(); x > 0; x--) {}
  }
}

```

In our imperative programming language, instead, we provide a set P of four statements acting on (global) variable x :

```

a:  $x := ? \rightarrow b$ 
b:  $x > 0 \rightarrow c$ 
b:  $x \leq 0 \rightarrow d$ 
c:  $x := x - 1 \rightarrow b$ 

```

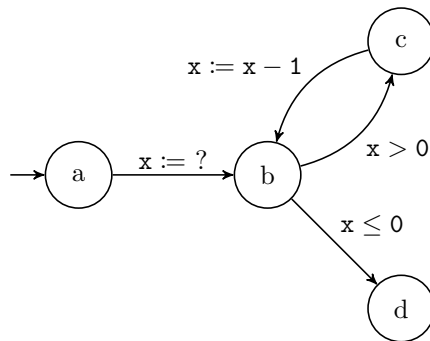
Execution starts at initial label ‘a’. The initial labels of a program P are called its *entry points* and are collected in a set $\mathcal{L}[P]$; hence in our example we have $\mathcal{L}[P] = \{a\}$. Notice that we allow $\mathcal{L}[P] \cap \text{lab}[P] = \emptyset$. Moreover, if $P = \emptyset$ then $\mathcal{L}[P] = \emptyset$.

If execution is at some label ℓ , then one of the transitions $\ell: C \rightarrow \ell'$ labelled with ℓ is crossed; if C is not blocking, the execution can go on from ℓ' . Programs are possibly non-deterministic, since several commands can be referenced by the same label. If the execution is at $\ell: C \rightarrow \ell'$ and there is no command labelled with ℓ' , then the execution is blocked at ℓ ; in our example, the execution is blocked at d , since in P there are no commands referenced by label d .

We can more easily deal with the chaining of statements by representing P as an automaton with

- set of states $\mathcal{L}[P] \cup \text{lab}[P] \cup \text{suc}[P]$;
- set of initial states $\mathcal{L}[P]$;
- set of accepting states equal to the set of states;
- alphabet $\{C \mid \exists \ell, \ell'. \ell: C \rightarrow \ell' \in P\}$;
- transition function P .

For instance, if P is the example program introduced above, we get:



Then all and only the correct chains of statements are captured by $L[P]$.

Syntactic Abstractions

Sometimes we would like to refer to a selected component of a syntactic construct. We thereby introduce some syntactic functions that take a construct and get rid of any but the component of interest. We call these functions *syntactic abstractions*, because they retain only what is relevant. Syntactic abstractions, presented in Table 2.2, are given names that are suggestive of their targets.

$\text{var}[[z]]$	$\stackrel{\text{def}}{=} \emptyset$		
$\text{var}[[x]]$	$\stackrel{\text{def}}{=} \{x\}$	$\text{com}[[\ell: C \rightarrow \ell']]$	$\stackrel{\text{def}}{=} C$
$\text{var}[[A_1 \cdot A_2]]$	$\stackrel{\text{def}}{=} \text{var}[[A_1]] \cup \text{var}[[A_2]]$		
$\text{var}[[\text{true}]]$	$\stackrel{\text{def}}{=} \emptyset$		
$\text{var}[[A_1 \approx A_2]]$	$\stackrel{\text{def}}{=} \text{var}[[A_1]] \cup \text{var}[[A_2]]$	$\text{lab}[[\ell: C \rightarrow \ell']]$	$\stackrel{\text{def}}{=} \ell$
$\text{var}[[\neg B]]$	$\stackrel{\text{def}}{=} \text{var}[[B]]$	$\text{lab}[[P]]$	$\stackrel{\text{def}}{=} \{ \text{lab}[[S] \mid S \in P \}$
$\text{var}[[B_1 \star B_2]]$	$\stackrel{\text{def}}{=} \text{var}[[B_1]] \cup \text{var}[[B_2]]$		
$\text{var}[[x := A]]$	$\stackrel{\text{def}}{=} \{x\} \cup \text{var}[[A]]$		
$\text{var}[[x := ?]]$	$\stackrel{\text{def}}{=} \{x\}$	$\text{suc}[[\ell: C \rightarrow \ell']]$	$\stackrel{\text{def}}{=} \ell'$
$\text{var}[[\ell: C \rightarrow \ell']]$	$\stackrel{\text{def}}{=} \text{var}[[C]]$	$\text{suc}[[P]]$	$\stackrel{\text{def}}{=} \{ \text{suc}[[S] \mid S \in P \}$
$\text{var}[[P]]$	$\stackrel{\text{def}}{=} \{ \text{var}[[S] \mid S \in P \}$		

Table 2.2. Syntactic abstractions.

Freshness

We take advantage of syntactic abstractions to introduce the notion of freshness. In particular:

- a variable x is *fresh* with respect to P if and only if $x \notin \text{var}[[P]]$;
- a label ℓ is *fresh* with respect to P if and only if $\ell \notin \text{lab}[[P]] \cup \text{suc}[[P]]$.

Replacing Variables

A variable y that is found inside a syntactic construct acts as placeholder for integer values. Sometimes we would like to replace y with some arithmetic expression A . In Table 2.3 a replacement function is defined. Rather than a formal name, this function is given peculiar notation. Double brackets around each of its three syntactic arguments have been dropped for the sake of simplicity.

Replacing Commands

A function we introduce that replaces whichever command with **true**:

$$\begin{aligned} \text{true}[[P]] &\stackrel{\text{def}}{=} \{ \text{true}[[S] \mid S \in P \} \\ \text{true}[[\ell: C \rightarrow \ell']] &\stackrel{\text{def}}{=} \ell: \text{true}[[C]] \rightarrow \ell' \\ \text{true}[[C]] &\stackrel{\text{def}}{=} \text{true} \end{aligned}$$

$z [A/y]$	$\stackrel{\text{def}}{=}$	z
$x [A/y]$	$\stackrel{\text{def}}{=}$	$\begin{cases} A & \text{if } y = x \\ x & \text{otherwise} \end{cases}$
$(A_1 \cdot A_2) [A/y]$	$\stackrel{\text{def}}{=}$	$A_1 [A/y] \cdot A_2 [A/y]$
true $[A/y]$	$\stackrel{\text{def}}{=}$	true
$(A_1 \approx A_2) [A/y]$	$\stackrel{\text{def}}{=}$	$A_1 [A/y] \approx A_2 [A/y]$
$\neg B [A/y]$	$\stackrel{\text{def}}{=}$	$B [A/y]$
$(B_1 \star B_2) [A/y]$	$\stackrel{\text{def}}{=}$	$B_1 [A/y] \star B_2 [A/y]$
$(x := A) [A/y]$	$\stackrel{\text{def}}{=}$	$x := A [A/y]$
$(x := ?) [A/y]$	$\stackrel{\text{def}}{=}$	$x := ?$
$(\ell: C \rightarrow \ell') [A/y]$	$\stackrel{\text{def}}{=}$	$C [A/y]$
$P [A/y]$	$\stackrel{\text{def}}{=}$	$\{ S [A/y] \mid S \in P \}$

Table 2.3. Replacement function.

Subroutines

In programming languages, a subroutine is customarily a small program which is used as one step in a larger program. It is accessed (called) by the program through a unique entry point, it performs some task and, upon completion, it branches back (returns) to the program. Subroutines are often collected into libraries, to enhance reusability. In this thesis, we expand the Cousot's language [33] to include subroutines. If P and H are programs and H can be used as a subroutine of P , we write $H \in \text{rou}[P]$. Thus $\text{rou}[P]$ is the library of the programs that may act as a subroutine for P .

Whenever we enrich a program P with a statement S that calls a subroutine $H \in \text{rou}[P]$, we are just considering a new program $P' = P \cup \{ S \} \cup H$. Not every $H \in \mathbb{P}$ can be a subroutine of P ; for instance, $P \notin \text{rou}[P]$. In fact, H can be a subroutine of P if and only if there are two labels ℓ and ℓ' such that:

- (a) the only entry point of H is ℓ ;
- (b) P can call H only through ℓ ;
- (c) H and P do not share labels, except for ℓ ;
- (d) H is allowed to return to P only through ℓ' .

Properties (a), (b) and (d) describe how subroutine H is expected to interface with program P ; we name ℓ and ℓ' respectively the *entry point* and the *exit point* of H ; sometimes, to stress this fact, we abuse notation and we designate H with $\ell: H \rightarrow \ell'$. Property (c) says that H and P have no statements in common; we allow ℓ to be shared so that P may non-deterministically call H and, at the same time, perform any statement $S' \in P$ such that $\text{lab}[S'] = \ell$.

Formally, given $\ell, \ell' \in \mathbb{L}$, we say that $H \in \mathbb{P}$ is can be used *subroutine of P with entry point ℓ and exit point ℓ'* and we write $\ell: H \rightarrow \ell' \in \text{rou}[P]$ if and only if:

$$\text{lab}[\mathbb{H}] \cap (\mathfrak{L}[\mathbb{P}] \cup \text{suc}[\mathbb{P}]) \subseteq \mathfrak{L}[\mathbb{H}] \quad (2.5)$$

$$\text{lab}[\mathbb{H}] \cap (\mathfrak{L}[\mathbb{P}] \cup \text{suc}[\mathbb{P}]) \subseteq \{ \ell \} \quad (2.6)$$

$$\text{lab}[\mathbb{H}] \cap \text{lab}[\mathbb{P}] \subseteq \{ \ell \} \quad (2.7)$$

$$\text{suc}[\mathbb{H}] \cap \text{lab}[\mathbb{P}] \subseteq \{ \ell' \} \quad (2.8)$$

$$\text{suc}[\mathbb{H}] \cap \text{suc}[\mathbb{P}] \subseteq \{ \ell' \} . \quad (2.9)$$

Notice that these constraints precisely captures the four properties enumerated above. Furthermore, they only concerns labels and not variables, which by definition are global; therefore \mathbb{P} can fully access the variables of \mathbb{H} and vice versa.

We can recast the definition of subroutine in an equivalent form.

Theorem 2.1. \mathbb{H} is a subroutine of \mathbb{P} with entry point ℓ and exit point ℓ' if and only if:

- (i) $\ell'' = \ell \in \mathfrak{L}[\mathbb{H}]$ for all $\ell'' \in \text{lab}[\mathbb{H}] \cap (\mathfrak{L}[\mathbb{P}] \cup \text{suc}[\mathbb{P}])$;
- (ii) if $\ell \neq \ell'$ and ℓ is not fresh with respect to \mathbb{P} , then $\ell \notin \text{suc}[\mathbb{H}]$;
- (iii) if $\ell \neq \ell'$ and ℓ' is not fresh with respect to \mathbb{P} , then $\ell' \notin \text{lab}[\mathbb{H}]$;
- (iv) any $\ell'' \in \text{lab}[\mathbb{H}] \cup \text{suc}[\mathbb{H}]$ such that $\ell'' \neq \ell$ and $\ell'' \neq \ell'$ is fresh with respect to \mathbb{P} .

Proof. It is trivial to show that (2.5) \wedge (2.6) \iff (i).

Let us deal with what else must be proven.

(\Rightarrow)

Suppose ℓ is not fresh with respect to \mathbb{P} , that is $\ell \in \text{lab}[\mathbb{P}] \cup \text{suc}[\mathbb{P}]$. Moreover let $\ell \neq \ell'$. From respectively (2.8) and (2.9), we get:

$$\ell \notin \text{suc}[\mathbb{H}] \cap \text{lab}[\mathbb{P}] \quad (2.10)$$

$$\ell \notin \text{suc}[\mathbb{H}] \cap \text{suc}[\mathbb{P}] . \quad (2.11)$$

From our initial assumption $\ell \in \text{lab}[\mathbb{P}] \cup \text{suc}[\mathbb{P}]$, we get $\ell \in \text{lab}[\mathbb{P}]$ or $\ell \in \text{suc}[\mathbb{P}]$. Then by (2.10) and (2.11), we have that $\ell \notin \text{suc}[\mathbb{H}]$. Thus we proved (ii).

Suppose ℓ' is not fresh with respect to \mathbb{P} , that is $\ell' \in \text{lab}[\mathbb{P}] \cup \text{suc}[\mathbb{P}]$. Moreover let $\ell \neq \ell'$. Then by (2.5) we have $\ell' \notin \mathfrak{L}[\mathbb{P}]$. From respectively (2.6) and (2.7) we get:

$$\ell' \notin \text{lab}[\mathbb{H}] \cap \text{suc}[\mathbb{P}] \quad (2.12)$$

$$\ell' \notin \text{lab}[\mathbb{H}] \cap \text{lab}[\mathbb{P}] . \quad (2.13)$$

From our initial assumption $\ell' \in \text{lab}[\mathbb{P}] \cup \text{suc}[\mathbb{P}]$, we get $\ell' \in \text{lab}[\mathbb{P}]$ or $\ell' \in \text{suc}[\mathbb{P}]$. Then by (2.12) and (2.13) we have that $\ell' \notin \text{lab}[\mathbb{H}]$. Thus we proved (iii).

Finally, suppose $\ell'' \in \text{lab}[\mathbb{H}] \cup \text{suc}[\mathbb{H}]$, $\ell'' \neq \ell$ and $\ell'' \neq \ell'$. Then by (2.5) we have $\ell'' \notin \mathfrak{L}[\mathbb{P}]$. Moreover, from respectively (2.6), (2.7), (2.8) and (2.9), we get:

$$\ell'' \notin \text{lab}[\mathbb{H}] \cap \text{suc}[\mathbb{P}] \quad (2.14)$$

$$\ell'' \notin \text{lab}[\mathbb{H}] \cap \text{lab}[\mathbb{P}] \quad (2.15)$$

$$\ell'' \notin \text{suc}[\mathbb{H}] \cap \text{lab}[\mathbb{P}] \quad (2.16)$$

$$\ell'' \notin \text{suc}[\mathbb{H}] \cap \text{suc}[\mathbb{P}] . \quad (2.17)$$

From our initial assumption $\ell'' \in \text{lab}[\mathbb{H}] \cup \text{suc}[\mathbb{H}]$, we get $\ell'' \in \text{lab}[\mathbb{H}]$ or $\ell'' \in \text{suc}[\mathbb{H}]$. In the former case, by (2.14) and (2.15), we have that $\ell'' \notin \text{lab}[\mathbb{P}] \cup \text{suc}[\mathbb{P}]$, that is,

ℓ'' is fresh with respect to P . In the latter case, we derive the same result by (2.16) and (2.17). Thus we proved (iv).

(\Leftarrow)

Let $\ell \neq \ell'$. Then by (i), $\ell' \notin \mathfrak{L}[P]$. We have the following points.

- If ℓ is fresh with respect to P , then $\ell \notin \text{lab}[P] \cup \text{suc}[P]$; if ℓ is not fresh with respect to P , then by (ii) we have $\ell \notin \text{suc}[H]$.
- If ℓ' is fresh with respect to P , then $\ell' \notin \text{lab}[P] \cup \text{suc}[P]$; if ℓ' is not fresh with respect to P , then by (iii) we have $\ell' \notin \text{lab}[H]$.

Consequently we obtain:

$$\ell' \notin \text{lab}[H] \cap (\mathfrak{L}[P] \cup \text{suc}[P]) \quad (2.18)$$

$$\ell' \notin \text{lab}[H] \cap \text{lab}[P] \quad (2.19)$$

$$\ell \notin \text{suc}[H] \cap \text{lab}[P] \quad (2.20)$$

$$\ell \notin \text{suc}[H] \cap \text{suc}[P] . \quad (2.21)$$

Moreover, let $\ell'' \neq \ell$ and $\ell'' \neq \ell'$. We consider two alternatives:

- $\ell'' \notin \text{lab}[H] \cup \text{suc}[H]$, whence we have $\ell'' \notin \text{lab}[H] \wedge \ell'' \notin \text{suc}[H]$;
- $\ell'' \in \text{lab}[H] \cup \text{suc}[H]$ whence, by (iv), we have $\ell'' \notin \text{lab}[P] \wedge \ell'' \notin \text{suc}[P]$.

Regardless of alternatives, we obtain:

$$\ell'' \notin \text{lab}[H] \cap (\mathfrak{L}[P] \cup \text{suc}[P]) \quad (2.22)$$

$$\ell'' \notin \text{lab}[H] \cap \text{lab}[P] \quad (2.23)$$

$$\ell'' \notin \text{suc}[H] \cap \text{lab}[P] \quad (2.24)$$

$$\ell'' \notin \text{suc}[H] \cap \text{suc}[P] . \quad (2.25)$$

We now gather the results we obtained. In particular:

- (2.18) and (2.22) yield (2.6);
- (2.19) and (2.23) yield (2.7);
- (2.20) and (2.24) yield (2.8);
- (2.21) and (2.25) yield (2.9).

□

Corollary 2.2. *For any program P and any statement $\ell: C \rightarrow \ell'$, $\{\ell: C \rightarrow \ell'\}$ is a subroutine of P with entry point ℓ and exit point ℓ' .*

Proof. Trivial by Theorem 2.1. □

With a slight abuse of notation, instead of $\ell: \{ \ell: C \rightarrow \ell' \} \rightarrow \ell' \in \text{rou}[P]$, we just write $\ell: C \rightarrow \ell' \in \text{rou}[P]$.

A program $P' = P \cup H$ is usually easier to be analyzed and maintained if $H \in \text{rou}[P]$ is a large subroutine. Indeed H narrows the range of possible interactions within the elements of P' by allowing very few of its own statements to be shared with P . We prove this fact in the following Proposition, but only for a restricted class of programs, namely the subroutines of H ; thus, instead of $P \in \mathbb{P}$ and $H \in \text{rou}[P]$, we are going to consider $H' \in \text{rou}[H]$ and $H \in \text{rou}[H']$. In particular, H and H' can share a statement S only if they have the same entry point ℓ and exit point ℓ' ; in such case $\text{lab}[S] = \ell$ and $\text{suc}[S] = \ell'$.

Proposition 2.3. *Let ℓ , ℓ' , ℓ'' and ℓ''' be labels. Let H and H' be programs. Let $\ell: H \rightarrow \ell' \in \text{rou}[H']$ and $\ell'': H' \rightarrow \ell''' \in \text{rou}[H]$. Then, for any statement $S \in H \cap H'$, $\text{lab}[S] = \ell = \ell''$ and $\text{suc}[S] = \ell' = \ell'''$.*

Proof. Let $S = \underline{\ell}: C \rightarrow \underline{\ell}'$ and $S \in H \cap H'$. The latter assumption entails $S \in H$ and $S \in H'$, whence:

$$\underline{\ell} \in \text{lab}[H] \quad (2.26)$$

$$\underline{\ell}' \in \text{suc}[H] \quad (2.27)$$

$$\underline{\ell} \in \text{lab}[H'] \quad (2.28)$$

$$\underline{\ell}' \in \text{suc}[H'] . \quad (2.29)$$

From (2.26) and (2.27) we get

$$\underline{\ell}, \underline{\ell}' \in \text{lab}[H] \cup \text{suc}[H] , \quad (2.30)$$

whereas from (2.28) and (2.29) we get

$$\underline{\ell}, \underline{\ell}' \in \text{lab}[H'] \cup \text{suc}[H'] . \quad (2.31)$$

As $\ell: H \rightarrow \ell'$ is a subroutine of H' , it enjoys Theorem 2.1.

- First, we consider property (ii)

$$\ell \neq \ell' \wedge \ell \in \text{lab}[H'] \cup \text{suc}[H'] \implies \ell \notin \text{suc}[H] ,$$

whose contrapositive form is

$$\ell \in \text{suc}[H] \implies \ell = \ell' \vee \ell \notin \text{lab}[H'] \cup \text{suc}[H'] . \quad (2.32)$$

- We now consider property (iii)

$$\ell \neq \ell' \wedge \ell' \in \text{lab}[H'] \cup \text{suc}[H'] \implies \ell' \notin \text{lab}[H] ,$$

whose contrapositive form is

$$\ell' \in \text{lab}[H] \implies \ell = \ell' \vee \ell' \notin \text{lab}[H'] \cup \text{suc}[H'] . \quad (2.33)$$

- Finally, let us consider property (iv):

$$\underline{\ell} \in \text{lab}[H] \cup \text{suc}[H] \wedge \underline{\ell} \neq \ell \wedge \underline{\ell} \neq \ell' \implies \underline{\ell} \notin \text{lab}[H'] \cup \text{suc}[H'] .$$

The contrapositive form of this property is

$$\underline{\ell} \in \text{lab}[H'] \cup \text{suc}[H'] \implies \underline{\ell} \notin \text{lab}[H] \cup \text{suc}[H] \vee \underline{\ell} = \ell \vee \underline{\ell} = \ell' ,$$

which, by (2.30) and (2.31), is equivalent to:

$$\underline{\ell} = \ell \vee \underline{\ell} = \ell' . \quad (2.34)$$

We can make the same argument for $\underline{\ell}'$; hence we obtain:

$$\underline{\ell}' = \ell \vee \underline{\ell}' = \ell' . \quad (2.35)$$

Let us summarize the results we obtained just now:

$$\ell \in \text{suc}[\mathbb{H}] \implies \ell = \ell' \vee \ell \notin \text{lab}[\mathbb{H}'] \cup \text{suc}[\mathbb{H}'] \quad (2.32)$$

$$\ell' \in \text{lab}[\mathbb{H}] \implies \ell' = \ell \vee \ell' \notin \text{lab}[\mathbb{H}'] \cup \text{suc}[\mathbb{H}'] \quad (2.33)$$

$$\underline{\ell} = \ell \vee \underline{\ell} = \ell' \quad (2.34)$$

$$\underline{\ell}' = \ell \vee \underline{\ell}' = \ell' . \quad (2.35)$$

Consider now that $\ell'' : \mathbb{H}' \rightarrow \ell'''$ is a subroutine of \mathbb{H} and enjoys Theorem 2.1 as well. By the same arguments above, we derive:

$$\ell'' \in \text{suc}[\mathbb{H}'] \implies \ell'' = \ell''' \vee \ell'' \notin \text{lab}[\mathbb{H}] \cup \text{suc}[\mathbb{H}] \quad (2.36)$$

$$\ell''' \in \text{lab}[\mathbb{H}'] \implies \ell''' = \ell'' \vee \ell''' \notin \text{lab}[\mathbb{H}] \cup \text{suc}[\mathbb{H}] \quad (2.37)$$

$$\underline{\ell} = \ell'' \vee \underline{\ell} = \ell''' \quad (2.38)$$

$$\underline{\ell}' = \ell'' \vee \underline{\ell}' = \ell''' . \quad (2.39)$$

We now explicit the properties of $\underline{\ell}$ and $\underline{\ell}'$.

- By combining (2.34) and (2.38), we get:

$$\begin{aligned} \underline{\ell} &= \ell = \ell'' \\ \vee \underline{\ell} &= \ell = \ell''' \\ \vee \underline{\ell} &= \ell' = \ell'' \\ \vee \underline{\ell} &= \ell' = \ell''' . \end{aligned} \quad (2.40)$$

Because by (2.26) we have $\underline{\ell} \in \text{lab}[\mathbb{H}]$ and by (2.31) we have $\underline{\ell} \in \text{lab}[\mathbb{H}'] \cup \text{suc}[\mathbb{H}']$, whenever $\underline{\ell} = \ell'$ from (2.33) we obtain $\underline{\ell} = \ell$.

Likewise, because by (2.28) we have $\underline{\ell} \in \text{lab}[\mathbb{H}']$ and by (2.30) we have $\underline{\ell} \in \text{lab}[\mathbb{H}] \cup \text{suc}[\mathbb{H}]$, whenever $\underline{\ell} = \ell'''$ from (2.37) we obtain $\underline{\ell} = \ell''$.

Thus we can extend (2.40) as follows:

$$\begin{aligned} \underline{\ell} &= \ell = \ell'' \\ \vee \underline{\ell} &= \ell = \ell''' = \ell'' \\ \vee \underline{\ell} &= \ell' = \ell'' = \ell \\ \vee \underline{\ell} &= \ell' = \ell''' = \ell = \ell'' . \end{aligned} \quad (2.41)$$

- By combining (2.35) and (2.39), we get:

$$\begin{aligned} \underline{\ell}' &= \ell = \ell'' \\ \vee \underline{\ell}' &= \ell = \ell''' \\ \vee \underline{\ell}' &= \ell' = \ell'' \\ \vee \underline{\ell}' &= \ell' = \ell''' . \end{aligned} \quad (2.42)$$

Because by (2.27) we have $\underline{\ell}' \in \text{suc}[\mathbb{H}]$ and by (2.31) we have $\underline{\ell}' \in \text{lab}[\mathbb{H}'] \cup \text{suc}[\mathbb{H}']$, whenever $\underline{\ell}' = \ell$ from (2.32) we obtain $\underline{\ell}' = \ell'$.

Likewise, because by (2.29) we have $\underline{\ell}' \in \text{suc}[\mathbb{H}']$ and by (2.30) we have $\underline{\ell}' \in \text{lab}[\mathbb{H}] \cup \text{suc}[\mathbb{H}]$, whenever $\underline{\ell}' = \ell''$ from (2.36) we obtain $\underline{\ell}' = \ell'''$.

Thus we can extend (2.42) as follows:

$$\begin{aligned}
 \underline{\ell}' &= \ell = \ell'' = \ell' = \ell''' \\
 \vee \underline{\ell}' &= \ell = \ell''' = \ell' \\
 \vee \underline{\ell}' &= \ell' = \ell'' = \ell''' \\
 \vee \underline{\ell}' &= \ell' = \ell''' .
 \end{aligned} \tag{2.43}$$

From (2.41) and (2.43) we conclude that $\underline{\ell} = \ell = \ell''$ and $\underline{\ell}' = \ell' = \ell'''$. \square

Both Lemma 2.2 and this Proposition tell us that statements are ultimate subroutines, that is, subroutines with respect to any program. We observe that there exists only one other program with this property: it is the empty program \emptyset . It is easy indeed to prove that $\ell: \emptyset \rightarrow \ell' \in \text{rou}[\mathbb{P}]$ for all $\ell, \ell' \in \mathbb{L}$ and for all $\mathbb{P} \in \mathbb{P}$.

Conversely, if a program \mathbb{H} is neither the empty set nor a singleton, it cannot be said to be a subroutine unconditionally, but only with respect to some program \mathbb{P}' . This compels us to always introduce such \mathbb{P}' even if we are only interested in \mathbb{H} . Notice that \mathbb{P}' collects commands that supposedly capture the context in which the subroutine is to be introduced. If we have no context, we can let $\mathbb{P}' = \emptyset$. Then it is trivial to verify that $\ell: \mathbb{H} \rightarrow \ell' \in \text{rou}[\emptyset]$ holds for any \mathbb{H} and for all $\ell, \ell' \in \mathbb{L}$. Thus, when the context is empty, $\ell: \mathbb{H} \rightarrow \ell'$ is always recognized as a subroutine. In the following, whenever we introduce a subroutine $\ell: \mathbb{H} \rightarrow \ell'$ without specifying its context \mathbb{P}' , we implicitly assume that $\mathbb{P}' = \emptyset$.

Environments

Program variables take their values in the domain of integer numbers, enriched with a special value \mathcal{U} , called the *undefined value*, with its obvious meaning. An environment ρ is a partial mapping from variables to the domain of values:

$$\rho: \mathbb{X} \rightarrow \mathbb{Z} \cup \{ \mathcal{U} \} .$$

The set of all environments is \mathfrak{E} . Given a program \mathbb{P} , we define its related set of environments as

$$\mathfrak{E}[\mathbb{P}] \stackrel{\text{def}}{=} \{ \rho \in \mathfrak{E} \mid \text{dom}[\rho] = \mathbb{X} \wedge \forall x \notin \text{var}[\mathbb{P}]. \rho(x) = \mathcal{U} \} ,$$

in which we assume that all the variables that are not found in \mathbb{P} denote the undefined value. We can obtain new environments out of old ones. Let $n, n' \in \mathbb{N}$, $\rho \in \mathfrak{E}$, $z_1, \dots, z_n, z'_1, \dots, z'_{n'} \in \mathbb{Z}$, $\mathbf{x}, \mathbf{x}_1, \dots, \mathbf{x}_n, \mathbf{x}'_1, \dots, \mathbf{x}'_{n'} \in \mathbb{X}$ and $\mathcal{X} \subseteq \mathbb{X}$. Then:

- $\rho \begin{bmatrix} \mathbf{x}_1 := z_1 \\ \vdots \\ \mathbf{x}_n := z_n \end{bmatrix}$ maps \mathbf{x}_i to z_i , and any other $\mathbf{x}' \neq \mathbf{x}_i$ to $\rho(\mathbf{x}')$, for all $1 \leq i \leq n$;
- $\rho \begin{bmatrix} \mathbf{x}_1 := z_1 \\ \vdots \\ \mathbf{x}_n := z_n \end{bmatrix} \begin{bmatrix} \mathbf{x}'_1 := z'_1 \\ \vdots \\ \mathbf{x}'_{n'} := z'_{n'} \end{bmatrix}$ is a shorthand for $\left(\rho \begin{bmatrix} \mathbf{x}_1 := z_1 \\ \vdots \\ \mathbf{x}_n := z_n \end{bmatrix} \right) \begin{bmatrix} \mathbf{x}'_1 := z'_1 \\ \vdots \\ \mathbf{x}'_{n'} := z'_{n'} \end{bmatrix}$,
where $n, n' \in \mathbb{N}$;

- $\rho|_{\mathcal{X}}$ is a shorthand for $\lambda x \in \text{dom}[\rho] \cap \mathcal{X}. \rho(x)$;
- $\rho \setminus \mathcal{X}$ is a shorthand for $\lambda x \in \text{dom}[\rho] \setminus \mathcal{X}. \rho(x)$;
- $\rho \setminus \mathbf{x}$ is a shorthand for $\rho \setminus \{ \mathbf{x} \}$.

$A[z] \rho$	$\stackrel{\text{def}}{=}$	z
$A[\mathbf{x}] \rho$	$\stackrel{\text{def}}{=}$	$\rho(\mathbf{x})$
$A[A_1 \cdot A_2] \rho$	$\stackrel{\text{def}}{=}$	$A[A_1] \rho \cdot A[A_2] \rho$
$B[\text{true}] \rho$	$\stackrel{\text{def}}{=}$	true
$B[A_1 \approx A_2] \rho$	$\stackrel{\text{def}}{=}$	$A[A_1] \rho \approx A[A_2] \rho$
$B[\neg B] \rho$	$\stackrel{\text{def}}{=}$	$\neg B[B] \rho$
$B[B_1 \star B_2] \rho$	$\stackrel{\text{def}}{=}$	$B[B_1] \rho \star B[B_2] \rho$
$C[B] \rho$	$\stackrel{\text{def}}{=}$	$\{ \rho' \mid B[B] \rho' = \text{true} \wedge \rho' = \rho \}$
$C[\mathbf{x} := A] \rho$	$\stackrel{\text{def}}{=}$	$\{ \rho[\mathbf{x} := A[A] \rho] \}$
$C[\mathbf{x} := ?] \rho$	$\stackrel{\text{def}}{=}$	$\{ \rho[\mathbf{x} := z] \mid z \in \mathbb{Z} \}$

Table 2.4. Semantics.

The Semantics of Expressions

The semantics of arithmetic and boolean expressions is specified through a couple of mappings

$$\begin{aligned} A: \mathbb{A} &\rightarrow (\mathfrak{E}[\mathbb{P}] \rightarrow \mathbb{Z} \cup \{ \mathcal{U} \}) \\ B: \mathbb{B} &\rightarrow (\mathfrak{E}[\mathbb{P}] \rightarrow \{ \text{true}, \text{false}, \mathcal{U} \}) . \end{aligned}$$

The inductive definition of both A and B is illustrated by Table 2.4. For the sake of simplicity, we make the following assumptions:

- we fail in telling syntactic representations of integer numbers, unary operators and binary operators from their semantic denotations;
- we implicitly assume that \mathcal{U} propagates from subexpressions to super-expressions, that is:

$$\begin{aligned} \text{if } B[B] \rho = \mathcal{U} & \quad \text{then } \neg B[B] \rho = \mathcal{U}; \\ \text{if } A[A_1] \rho = \mathcal{U} \text{ or } A[A_2] \rho = \mathcal{U} & \quad \text{then } \begin{cases} A[A_1] \rho \cdot A[A_2] \rho = \mathcal{U} \\ A[A_1] \rho \approx A[A_2] \rho = \mathcal{U}; \end{cases} \\ \text{if } B[B_1] \rho = \mathcal{U} \text{ or } B[B_2] \rho = \mathcal{U} & \quad \text{then } B[B_1] \rho \star B[B_2] \rho = \mathcal{U}. \end{aligned}$$

Once we have given these premises, we can easily acknowledge that:

- $A[A] \rho$ is well defined for $A \in \mathbb{A}$ and $\rho \in \mathfrak{E}$ if and only if $\text{var}[A] \subseteq \text{dom}[\rho]$;
- $B[B] \rho$ is well defined for $B \in \mathbb{B}$ and $\rho \in \mathfrak{E}$ if and only if $\text{var}[B] \subseteq \text{dom}[\rho]$.

The Semantics of Commands

The semantics of commands is specified through a mapping

$$\mathbf{C}: \mathbb{C} \rightarrow (\mathfrak{E} \rightarrow \wp(\mathfrak{E})) ,$$

meaning that the execution of $\mathbf{C} \in \mathbb{C}$ in $\rho \in \mathfrak{E}$ results in a set of environments. This is typical in imperative languages, whose commands are expected to make new environments out of the current one.

The inductive definition of \mathbf{C} is found in Table 2.4. In particular:

- a boolean test \mathbf{B} yields at most one environment – indeed, by definition, we have $\mathbf{C}[\mathbf{B}] \rho = \{ \rho \}$ if and only if $\mathbf{B}[\mathbf{B}] \rho = \mathbf{true}$, otherwise $\mathbf{C}[\mathbf{B}] \rho = \emptyset$;
- a deterministic assignment yields just one environment, obtained from the original environment by updating the assigned variable;
- a random assignment yields an infinite number of environments, as defined by the correspondent entry of Table 2.4.

We know that if we replace a variable $\mathbf{y} \in \mathbf{var}[\mathbb{C}]$ with $\mathbf{A} \in \mathbb{A}$ in a command \mathbf{C} , we obtain a new command $\mathbf{C}[\mathbf{A}/\mathbf{y}]$. The following result relates the semantic of \mathbf{C} to the semantics of $\mathbf{C}[\mathbf{A}/\mathbf{y}]$ in case \mathbf{A} denotes a total bijective function φ on \mathbb{Z} .

Lemma 2.4. *Let y, y' be integer numbers, ρ, ρ' be environments and \mathbf{y} be a variable such that $\mathbf{A}[\mathbf{y}] \rho = y$ and $\mathbf{A}[\mathbf{y}] \rho' = y'$. Let \mathbf{C} be a command such that \mathbf{C} is not an assignment on \mathbf{y} . Let φ denote a total bijective function on \mathbb{Z} . Then:*

- $\rho' \in \mathbf{C}[\mathbf{C}[\varphi(\mathbf{y})/\mathbf{y}]] \rho[\mathbf{y} := \varphi^{-1}(y)]$ if and only if $\rho'[\mathbf{y} := \varphi(y')] \in \mathbf{C}[\mathbf{C}] \rho$;
- $\rho' \in \mathbf{C}[\mathbf{C}] \rho[\mathbf{y} := \varphi^{-1}(y)]$ if and only if $\rho'[\mathbf{y} := \varphi(y')] \in \mathbf{C}[\mathbf{C}[\varphi(\mathbf{y})/\mathbf{y}]] \rho$.

Proof. We prove only the first point (the second one being similar) by induction on \mathbf{C} . We first show that, for any arithmetic expression \mathbf{A} , we have $\mathbf{A}[\mathbf{A}[\varphi(\mathbf{y})/\mathbf{y}]] \rho[\mathbf{y} := \varphi^{-1}(y)] = \mathbf{A}[\mathbf{A}] \rho$.

- $\mathbf{A}[\mathbf{z}[\varphi(\mathbf{y})/\mathbf{y}]] \rho[\mathbf{y} := \varphi^{-1}(y)] = \mathbf{A}[\mathbf{z}] \rho[\mathbf{y} := \varphi^{-1}(y)]$
 $= z$
 $= \mathbf{A}[\mathbf{z}] \rho .$
- Let $\mathbf{y} = \mathbf{x}$. Then $\mathbf{A}[\mathbf{x}[\varphi(\mathbf{y})/\mathbf{y}]] \rho[\mathbf{y} := \varphi^{-1}(y)] = \mathbf{A}[\varphi(y)] \rho[\mathbf{y} := \varphi^{-1}(y)]$
 $= \varphi(\mathbf{A}[\mathbf{y}] \rho[\mathbf{y} := \varphi^{-1}(y)])$
 $= \varphi(\varphi^{-1}(y))$
 $= y$
 $= \mathbf{A}[\mathbf{y}] \rho .$
- Let $\mathbf{y} \neq \mathbf{x}$. Then $\mathbf{A}[\mathbf{x}[\varphi(\mathbf{y})/\mathbf{y}]] \rho[\mathbf{y} := \varphi^{-1}(y)] = \mathbf{A}[\mathbf{x}] \rho[\mathbf{y} := \varphi^{-1}(y)]$
 $= \rho(\mathbf{x})$
 $= \mathbf{A}[\mathbf{x}] \rho .$
- By inductive hypothesis we have $\mathbf{A}[\mathbf{A}_1[\varphi(\mathbf{y})/\mathbf{y}]] \rho[\mathbf{y} := \varphi^{-1}(y)] = \mathbf{A}[\mathbf{A}_1] \rho$ and $\mathbf{A}[\mathbf{A}_2[\varphi(\mathbf{y})/\mathbf{y}]] \rho[\mathbf{y} := \varphi^{-1}(y)] = \mathbf{A}[\mathbf{A}_2] \rho$.

$$\begin{aligned}
& \mathbf{A}[\mathbf{A}_1 \cdot \mathbf{A}_2] [\varphi(y)/y] \rho [y := \varphi^{-1}(y)] \\
&= \mathbf{A}[\mathbf{A}_1 [\varphi(y)/y] \cdot \mathbf{A}_2 [\varphi(y)/y]] \rho [y := \varphi^{-1}(y)] \\
&= \mathbf{A}[\mathbf{A}_1 [\varphi(y)/y]] \rho [y := \varphi^{-1}(y)] \cdot \mathbf{A}[\mathbf{A}_2 [\varphi(y)/y]] \rho [y := \varphi^{-1}(y)] \\
&= \mathbf{A}[\mathbf{A}_1] \rho \cdot \mathbf{A}[\mathbf{A}_2] \rho \\
&= \mathbf{A}[\mathbf{A}_1 \cdot \mathbf{A}_2] \rho .
\end{aligned}$$

Next we show that, for any boolean expression \mathbf{B} , we have $\mathbf{B}[\mathbf{B} [\varphi(y)/y]] \rho [y := \varphi^{-1}(y)] = \mathbf{B}[\mathbf{B}] \rho$.

- $\mathbf{B}[\mathbf{true} [\varphi(y)/y]] \rho [y := \varphi^{-1}(y)] = \mathbf{B}[\mathbf{true}] \rho [y := \varphi^{-1}(y)]$
 $= \mathbf{true}$
 $= \mathbf{B}[\mathbf{true}] \rho .$
- We already know that $\mathbf{A}[\mathbf{A}_1 [\varphi(y)/y]] \rho [y := \varphi^{-1}(y)] = \mathbf{A}[\mathbf{A}_1] \rho$ and, on the other side, that $\mathbf{A}[\mathbf{A}_2 [\varphi(y)/y]] \rho [y := \varphi^{-1}(y)] = \mathbf{A}[\mathbf{A}_2] \rho$.

$$\begin{aligned}
& \mathbf{B}[(\mathbf{A}_1 \simeq \mathbf{A}_2) [\varphi(y)/y]] \rho [y := \varphi^{-1}(y)] \\
&= \mathbf{A}[\mathbf{A}_1 [\varphi(y)/y] \simeq \mathbf{A}_2 [\varphi(y)/y]] \rho [y := \varphi^{-1}(y)] \\
&= \mathbf{A}[\mathbf{A}_1 [\varphi(y)/y]] \rho [y := \varphi^{-1}(y)] \simeq \mathbf{A}[\mathbf{A}_2 [\varphi(y)/y]] \rho [y := \varphi^{-1}(y)] \\
&= \mathbf{A}[\mathbf{A}_1] \rho \simeq \mathbf{A}[\mathbf{A}_2] \rho \\
&= \mathbf{B}[\mathbf{A}_1 \simeq \mathbf{A}_2] \rho .
\end{aligned}$$

- By inductive hypothesis we have that $\mathbf{B}[\mathbf{B} [\varphi(y)/y]] \rho [y := \varphi^{-1}(y)] = \mathbf{B}[\mathbf{B}] \rho$.

$$\begin{aligned}
& \mathbf{B}[(\neg \mathbf{B}) [\varphi(y)/y]] \rho [y := \varphi^{-1}(y)] \\
&= \mathbf{B}[\neg \mathbf{B} [\varphi(y)/y]] \rho [y := \varphi^{-1}(y)] \\
&= \neg \mathbf{B}[\mathbf{B} [\varphi(y)/y]] \rho [y := \varphi^{-1}(y)] \\
&= \neg \mathbf{B}[\mathbf{B}] \rho \\
&= \mathbf{B}[\neg \mathbf{B}] \rho .
\end{aligned}$$

- By inductive hypothesis we have that $\mathbf{B}[\mathbf{B}_1 [\varphi(y)/y]] \rho [y := \varphi^{-1}(y)] = \mathbf{B}[\mathbf{B}_1] \rho$ and $\mathbf{B}[\mathbf{B}_2 [\varphi(y)/y]] \rho [y := \varphi^{-1}(y)] = \mathbf{B}[\mathbf{B}_2] \rho$.

$$\begin{aligned}
& \mathbf{B}[(\mathbf{B}_1 \star \mathbf{B}_2) [\varphi(y)/y]] \rho [y := \varphi^{-1}(y)] \\
&= \mathbf{B}[\mathbf{B}_1 [\varphi(y)/y] \star \mathbf{B}_2 [\varphi(y)/y]] \rho [y := \varphi^{-1}(y)] \\
&= \mathbf{B}[\mathbf{B}_1 [\varphi(y)/y]] \rho [y := \varphi^{-1}(y)] \star \mathbf{B}[\mathbf{B}_2 [\varphi(y)/y]] \rho [y := \varphi^{-1}(y)] \\
&= \mathbf{B}[\mathbf{B}_1] \rho \star \mathbf{B}[\mathbf{B}_2] \rho \\
&= \mathbf{B}[\mathbf{B}_1 \star \mathbf{B}_2] \rho .
\end{aligned}$$

Finally, we prove our thesis.

- We already know that $\mathbf{B}[\mathbf{B} [\varphi(y)/y]] \rho [y := \varphi^{-1}(y)] = \mathbf{B}[\mathbf{B}] \rho$.

$$\begin{aligned}
& \rho' \in \mathbf{C}[\mathbf{B} [\varphi(y)/y]] \rho [y := \varphi^{-1}(y)] \\
& \iff \rho' \in \{ \rho' \mid \mathbf{B}[\mathbf{B} [\varphi(y)/y]] \rho [y := \varphi^{-1}(y)] = \mathbf{true} \wedge \rho' = \rho [y := \varphi^{-1}(y)] \} \\
& \iff \rho' \in \{ \rho' \mid \mathbf{B}[\mathbf{B}] \rho = \mathbf{true} \wedge \rho' = \rho [y := \varphi^{-1}(y)] \} \\
& \iff \rho' [y := \varphi(y')] \in \{ \rho' [y := \varphi(y')] \mid \mathbf{B}[\mathbf{B}] \rho = \mathbf{true} \\
& \quad \wedge \rho' [y := \varphi(y')] = \rho [y := \varphi^{-1}(y)] [y := \varphi(y')] \} \\
& \iff \rho' [y := \varphi(y')] \in \{ \rho' [y := \varphi(y')] \mid \mathbf{B}[\mathbf{B}] \rho = \mathbf{true} \\
& \quad \wedge \rho' [y := \varphi(y')] = \rho [y := \varphi(\varphi^{-1}(y))] \} \\
& \iff \rho' [y := \varphi(y')] \in \{ \rho' [y := \varphi(y')] \mid \mathbf{B}[\mathbf{B}] \rho = \mathbf{true} \\
& \quad \wedge \rho' [y := \varphi(y')] = \rho [y := y] \} \\
& \iff \rho' [y := \varphi(y')] \in \{ \rho' [y := \varphi(y')] \mid \mathbf{B}[\mathbf{B}] \rho = \mathbf{true} \wedge \rho' [y := \varphi(y')] = \rho \} \\
& \iff \rho' [y := \varphi(y')] \in \mathbf{C}[\mathbf{B}] \rho .
\end{aligned}$$

- We already know that $\mathbf{A}[\mathbf{A} [\varphi(y)/y]] \rho [y := \varphi^{-1}(y)] = \mathbf{A}[\mathbf{A}] \rho$.

$$\begin{aligned}
& \rho' \in \mathbf{C}[(x := \mathbf{A}) [\varphi(y)/y]] \rho [y := \varphi^{-1}(y)] \\
& \iff \rho' \in \mathbf{C}[(x := \mathbf{A} [\varphi(y)/y])] \rho [y := \varphi^{-1}(y)] \\
& \iff \rho' \in \{ \rho [y := \varphi^{-1}(y)] [x := \mathbf{A}[\mathbf{A} [\varphi(y)/y]] \rho [y := \varphi^{-1}(y)]] \} \\
& \iff \rho' \in \{ \rho [y := \varphi^{-1}(y)] [x := \mathbf{A}[\mathbf{A}] \rho] \} \\
& \iff \rho' \in \left\{ \rho \left[\begin{array}{l} y := \varphi^{-1}(y) \\ x := \mathbf{A}[\mathbf{A}] \rho \end{array} \right] \right\} \\
& \iff \rho' [y := \varphi(y')] \in \left\{ \rho \left[\begin{array}{l} y := \varphi^{-1}(y) \\ x := \mathbf{A}[\mathbf{A}] \rho \end{array} \right] \left[y := \varphi \left(\mathbf{A}[\mathbf{y}] \rho \left[\begin{array}{l} y := \varphi^{-1}(y) \\ x := \mathbf{A}[\mathbf{A}] \rho \end{array} \right] \right) \right] \right\} \\
& \iff \rho' [y := \varphi(y')] \in \left\{ \rho \left[\begin{array}{l} y := \varphi^{-1}(y) \\ x := \mathbf{A}[\mathbf{A}] \rho \end{array} \right] [y := \varphi(\varphi^{-1}(y))] \right\} \\
& \iff \rho' [y := \varphi(y')] \in \left\{ \rho \left[\begin{array}{l} y := \varphi^{-1}(y) \\ x := \mathbf{A}[\mathbf{A}] \rho \end{array} \right] [y := y] \right\} \\
& \iff \rho' [y := \varphi(y')] \in \{ \rho [x := \mathbf{A}[\mathbf{A}] \rho] \} \\
& \iff \rho' [y := \varphi(y')] \in \mathbf{C}[x := \mathbf{A}] \rho .
\end{aligned}$$

- $\rho' \in \mathbf{C}[\![\mathbf{x} := ?]\!] [\mathcal{A}(y)/y] \rho [\mathbf{y} := \varphi^{-1}(y)]$
 $\iff \rho' \in \mathbf{C}[\![\mathbf{x} := ?]\!] \rho [\mathbf{y} := \varphi^{-1}(y)]$
 $\iff \rho' \in \{ \rho [\mathbf{y} := \varphi^{-1}(y)] [\mathbf{x} := z] \mid z \in \mathbb{Z} \}$
 $\iff \rho' \in \left\{ \rho \left[\begin{array}{l} \mathbf{y} := \varphi^{-1}(y) \\ \mathbf{x} := z \end{array} \right] \mid z \in \mathbb{Z} \right\}$
 $\iff \rho' [\mathbf{y} := \varphi(y')] \in \left\{ \rho \left[\begin{array}{l} \mathbf{y} := \varphi^{-1}(y) \\ \mathbf{x} := z \end{array} \right] \left[\mathbf{y} := \varphi \left(\mathbf{A}[\![\mathbf{y}]\!] \rho \left[\begin{array}{l} \mathbf{y} := \varphi^{-1}(y) \\ \mathbf{x} := z \end{array} \right] \right) \right] \mid z \in \mathbb{Z} \right\}$
 $\iff \rho' [\mathbf{y} := \varphi(y')] \in \left\{ \rho \left[\begin{array}{l} \mathbf{y} := \varphi^{-1}(y) \\ \mathbf{x} := z \end{array} \right] [\mathbf{y} := \varphi(\varphi^{-1}(y))] \mid z \in \mathbb{Z} \right\}$
 $\iff \rho' [\mathbf{y} := \varphi(y')] \in \left\{ \rho \left[\begin{array}{l} \mathbf{y} := \varphi^{-1}(y) \\ \mathbf{x} := z \end{array} \right] [\mathbf{y} := y] \mid z \in \mathbb{Z} \right\}$
 $\iff \rho' [\mathbf{y} := \varphi(y')] \in \{ \rho [\mathbf{x} := z] \mid z \in \mathbb{Z} \}$
 $\iff \rho' [\mathbf{y} := \varphi(y')] \in \mathbf{C}[\![\mathbf{x} := ?]\!] \rho .$
-

The Semantics of Statements

The semantics of a statement is specified through a transition system (see Section 2.13).

In our framework, a state s is a pair $\langle \rho, \mathbf{S} \rangle$, where environment $\rho \in \mathfrak{E}$ records the value of variables and $\mathbf{S} \in \mathfrak{S}$ is the next statement to be executed. The set of all states is noted Σ , where

$$\Sigma \stackrel{\text{def}}{=} \mathfrak{E} \times \mathfrak{S} .$$

All the states in Σ can be initial states; therefore we let

$$\mathcal{J} \stackrel{\text{def}}{=} \Sigma .$$

Finally, instead of a transition relation, we provide a function $\mathbf{S}: \Sigma \rightarrow \wp(\Sigma)$ which takes a state s and returns the set of all its possible successors:

$$\mathbf{S} \langle \rho, \mathbf{S} \rangle \stackrel{\text{def}}{=} \{ \langle \rho', \mathbf{S}' \rangle \in \Sigma \mid \rho' \in \mathbf{C}[\![\text{com}[\![\mathbf{S}]\!]] \rho \wedge \text{suc}[\![\mathbf{S}]\!] = \text{lab}[\![\mathbf{S}']]\!] \} . \quad (2.44)$$

The set of all finite partial traces is $\Sigma^\rightarrow \subseteq \Sigma^*$, where

$$\Sigma^\rightarrow \stackrel{\text{def}}{=} \text{lfp}_{\subseteq}^{\mathfrak{C}} \mathbf{F} .$$

The fixpoint operator $\mathbf{F}: \Sigma^* \rightarrow \Sigma^*$, used to make traces grow one state longer, is

$$\mathbf{F}(\mathcal{J}) \stackrel{\text{def}}{=} \mathcal{J} \cup \{ \sigma s s' \mid \sigma s \in \mathcal{J} \wedge s' \in \mathbf{S}(s) \} .$$

We take advantage of this transition system to expand the semantic description of our language. In particular, the semantics of a statement \mathbf{S} executed in an environment ρ is $\mathbf{S} \langle \rho, \mathbf{S} \rangle$. Therefore we can say that the execution of a statement in an environment yields a set of states.

The Semantics of Programs

In the transition system defined above, states are made out of all possible environments and statements. In order however to define the semantics of a program P , we need a transition system whose states involve only environments in $\mathfrak{E}[P]$ and statements in P . The set that collects all such states is

$$\Sigma[P] \stackrel{\text{def}}{=} \mathfrak{E}[P] \times P .$$

The elements of $\Sigma[P]$ are called the states of P . The function used to specify the transition relation is $S[P]: \Sigma \rightarrow \wp(\Sigma[P])$, where

$$S[P] \langle \rho, S \rangle \stackrel{\text{def}}{=} S \langle \rho, S \rangle \cap \Sigma[P] .$$

A state $\langle \rho, S \rangle \in \Sigma[P]$ is initial as long as $\text{lab}[S]$ is an entry point of P . Thus we let the set of the initial states of P be

$$\mathcal{J}[P] \stackrel{\text{def}}{=} \{ \langle \rho, S \rangle \in \Sigma[P] \mid \text{lab}[S] \in \mathcal{L}[P] \} , \quad (2.45)$$

where $\mathcal{L}[P] \subseteq \text{lab}[P]$ is the set of the entry points of P . This completes the definition of a transition system restricted to the states of P .

To compute the finite partial traces of this new system, we introduce a fixpoint operator $F[P]: \Sigma^* \rightarrow \Sigma^*$ which makes traces grow longer by appending to them only states that are in $\Sigma[P]$:

$$F[P]\mathcal{J} \stackrel{\text{def}}{=} \mathcal{J}[P] \cup \{ \sigma s s' \mid \sigma s \in \mathcal{J} \wedge s' \in S[P] s \} . \quad (2.46)$$

Accordingly, the set of finite partial traces, defined in the customary least fixpoint form, is

$$\mathcal{S}[P] \stackrel{\text{def}}{=} \text{lfp}_{\subseteq}^{\mathcal{S}} F[P] . \quad (2.47)$$

It is easy to verify that $\mathcal{S}[P] \subseteq \Sigma^+$ for any program $P \in \mathbb{P}$. Moreover, if $\sigma \in \mathcal{S}[P]$ and $\sigma = \lambda i. \langle \rho_i, S_i \rangle$, where $\rho_i \in \mathfrak{E}[P]$ and $S_i \in P$, then $(\lambda i. S_i) \in \mathcal{L}[P]$. This means that each trace in $\mathcal{S}[P]$ is finite and captures a partial execution of P .

We let the semantics of P be $\mathcal{S}[P]$. Notice that $\mathcal{S}[P] = \text{lfp}_{\subseteq}^{\mathcal{S}} F[P]$ is undefined in case P denotes an infinite trace. Indeed, since $F[P]$ makes traces grow only one state longer, it would require an infinite number of applications to build up an infinite trace, whereas well-defined least fix points are expected to be reached in an unbound, but finite, number of steps.

In this thesis we consider only programs that denote finite traces. \mathcal{S} then can be seen as a mapping from programs to sets of finite traces. Programs are collected by \mathbb{P} . Sets of finite traces are collected by $\wp(\Sigma^+)$. So we can say that the semantics of programs is specified through a mapping

$$\mathcal{S}: \mathbb{P} \rightarrow \wp(\Sigma^+) .$$

Deriving Programs from Sequences of States

A set $\mathcal{J} \in \wp(\Sigma^*)$ of sequences can be mapped to a program by collecting the statements executed along the sequences:

$$\mathbb{P}(\mathcal{T}) \stackrel{\text{def}}{=} \{ \mathbf{S} \mid \exists \sigma \in \mathcal{T}. \exists j \in [0, |\sigma|). \exists \rho \in \mathfrak{C}. \sigma_j = \langle \rho, \mathbf{S} \rangle \} .$$

So we can say that \mathbb{P} is a mapping

$$\mathbb{P}: \wp(\Sigma^*) \rightarrow \mathbb{P} .$$

Syntactic Equivalence

Let $A_1, A_2 \in \mathbb{A}$ be two arithmetic expressions. We say that A_1 is *syntactically equivalent* to A_2 , noted $A_1 \equiv A_2$, if and only if $A[[A_1]] = A[[A_2]]$.

$A_1 \equiv A_2$	if and only if	$A[[A_1]] = A[[A_2]]$
$B_1 \equiv B_2$	if and only if	$B[[B_1]] = B[[B_2]]$
$C_1 \equiv C_2$	if and only if	$C[[C_1]] = C[[C_2]]$
$S \equiv S'$	if and only if	$S[[\text{com}[[S]]]] = S[[\text{com}[[S']]]]$

Table 2.5. Syntactic equivalence.

In general, given two constructs from the same syntactic class, we have syntactic equivalence whenever both constructs have the same denotation. This principle instantiates to the entries of Table 2.5. In each entry we exploited the same symbol \equiv , relating upon the context for disambiguation. The fact that \equiv is an equivalence relation follows from the use of $=$ in the definition of each relation.

Equivalence between Programs

In order to define a notion of syntactic equivalence between programs, let us first introduce equivalence relations between states, sequences and set of sequences. Again, in each entry of Table 2.6 we use the same symbol \equiv , relating upon the

$\langle \rho, \mathbf{S} \rangle \equiv \langle \rho', \mathbf{S}' \rangle$	if and only if	$\rho = \rho'$ and $\mathbf{S} \equiv \mathbf{S}'$
$\sigma \equiv \sigma'$	if and only if	$ \sigma = \sigma' $ and $\forall i. 0 \leq i < \sigma \implies \sigma_i \equiv \sigma'_i$
$\mathcal{T} \equiv \mathcal{T}'$	if and only if	$\forall \sigma \in \mathcal{T}. \exists \sigma' \in \mathcal{T}'. \sigma \equiv \sigma'$ and $\forall \sigma' \in \mathcal{T}'. \exists \sigma \in \mathcal{T}. \sigma' \equiv \sigma$

Table 2.6. Equivalence relations.

context for disambiguation.

A program P is equivalent to another program P' if and only if they have equivalent set of traces:

$$P \equiv P' \text{ if and only if } \mathcal{S}[[P]] \equiv \mathcal{S}[[P']] .$$

Dependence Relations

A *dependence* is a binary relation on P that constraints the execution order of statements [80].

Let $S, S' \in P$ be two statements. Assume there exists $\rho, \rho' \in \mathfrak{C}[P]$ and σ, σ' and σ'' such that $\sigma \langle \rho, S \rangle \sigma' \langle \rho', S' \rangle \sigma'' \in \mathfrak{S}[P]$. Thus we are supposing that S comes before S' in the execution of P tracked by σ . Statement S' depends on statement S if and only if we cannot swap them without affecting σ'' .

In particular:

- if S is a boolean expression, we have a *control dependence*;

Furthermore, we have four kinds of *data dependences*:

- if S' reads from a variable S assigns, we have a *flow or true dependence*;
- if S' assigns a variable that S reads from, we have an *anti-dependence*;
- if both S and S' assigns the same variable, we have an *output dependence*;
- if both S and S' reads from the same variable, we have an *input dependence*.

Actually, the last dependence does not constraint the execution order, but it can be useful all the same.

For-loops

Programming languages usually provide two kinds of programming constructs: basic and additional. Basic constructs outline the computational power of a language, that is, the range of operations that the language can perform. Additional constructs identify recurrent programming patterns obtained by specializing or assembling basic constructs. Thus, rather than contributing to the computational power of the language, they mark the use of well-behaved elaborated operations, helping software developers make program that are more comprehensible, reliable and maintainable. This methodology is extensively applied, for instance, in object-oriented programming, whose novel constructs (classes, objects, methods, interfaces...) do not extend the computational power already achieved through imperative programming, but provide an innovative way for managing it [6].

In our language, commands are the basic constructs, whereas subroutines are a first example of additional construct. We take advantage of both commands and subroutines to expand the Cousots' language with for-loops, an additional construct we will use throughout this thesis. A *for-loop* is a program which maintains a variable x , called the *index variable*, that is initialized to a fixed amount a_1 , and increases periodically by a fixed amount a_3 . Every increment is preceded by the execution of a subroutine H . The for-loop stops as soon as x goes beyond a fixed boundary a_2 . We now translate this specification in our language.

- First, let 'f', 'g', 'h', 'i' and 'j' be distinct labels. Let x be a variable, and A_1, A_2 and A_3 be arithmetic expressions. Let P' be a program such that $\mathfrak{L}[P'] \stackrel{\text{def}}{=} \{ f \}$ and $P' \stackrel{\text{def}}{=} \{ f: F \rightarrow g, \quad g: G \rightarrow h, \quad i: I \rightarrow g, \quad g: \bar{G} \rightarrow j \}$. We let

$$F \stackrel{\text{def}}{=} x := A_1$$

and we define G, \bar{G} and I according one of the following sets of definitions:

$$\begin{aligned}
\mathbf{G} &\stackrel{\text{def}}{=} \mathbf{x} \leq \mathbf{A}_2 \\
\bar{\mathbf{G}} &\stackrel{\text{def}}{=} \mathbf{x} > \mathbf{A}_2 \\
\mathbf{I} &\stackrel{\text{def}}{=} \mathbf{x} := \mathbf{x} + \mathbf{A}_3
\end{aligned} \tag{2.48}$$

or

$$\begin{aligned}
\mathbf{G} &\stackrel{\text{def}}{=} \mathbf{x} \geq \mathbf{A}_2 \\
\bar{\mathbf{G}} &\stackrel{\text{def}}{=} \mathbf{x} < \mathbf{A}_2 \\
\mathbf{I} &\stackrel{\text{def}}{=} \mathbf{x} := \mathbf{x} - \mathbf{A}_3 .
\end{aligned} \tag{2.49}$$

In either case, command \mathbf{F} is executed first; it takes care of initializing \mathbf{x} . Then \mathbf{G} and $\bar{\mathbf{G}}$ are evaluated, which test the index variable against the boundary. Such commands are called the *guard*. It is easy to verify that $\mathbf{B}[\mathbf{G}] = \neg\mathbf{B}[\bar{\mathbf{G}}]$. Thus the guard implements a conditioned branching. Lastly, we have a command \mathbf{I} that provides \mathbf{x} with its periodical increment. As we expect \mathbf{A}_2 and \mathbf{A}_3 to evaluate to fixed amounts, we require that $\mathbf{x} \notin \text{var}[\mathbf{A}_2]$ and $\mathbf{x} \notin \text{var}[\mathbf{A}_3]$.

In the rest of this thesis, unless otherwise specified, we deal with for-loops defined out of (2.48).

- Next, let us introduce a program \mathbf{H} such that $h: \mathbf{H} \rightarrow i \in \text{rou}[\mathbf{P}']$. We expect no statements in \mathbf{H} to assign \mathbf{x} or any variable in \mathbf{A}_2 or in \mathbf{A}_3 ; this ensures that \mathbf{H} does not affect the value of \mathbf{x} , \mathbf{A}_2 or \mathbf{A}_3 .
- Finally, let us collect all the commands of \mathbf{H} and \mathbf{P}' into a new program $\mathbf{P} \stackrel{\text{def}}{=} \mathbf{P}'\mathbf{U}\mathbf{H}$, with $\mathcal{L}[\mathbf{P}] \stackrel{\text{def}}{=} \{f\}$.

Now let \mathbf{P}'' be a program. We say that \mathbf{P} is a for-loop of \mathbf{P}'' if and only if $\ell: \mathbf{P} \rightarrow \ell' \in \text{rou}[\mathbf{P}'']$, where ℓ and ℓ' are labels. Whenever we say that \mathbf{P} is a for-loop and we do not specify \mathbf{P}'' , we implicitly assume that $\mathbf{P}'' = \emptyset$.

In this thesis we provide various representations of \mathbf{P} . By definition we have:

$$\mathbf{P} = \{f: \mathbf{F} \rightarrow g, g: \mathbf{G} \rightarrow h, i: \mathbf{I} \rightarrow g, g: \bar{\mathbf{G}} \rightarrow j\} \cup \mathbf{H} .$$

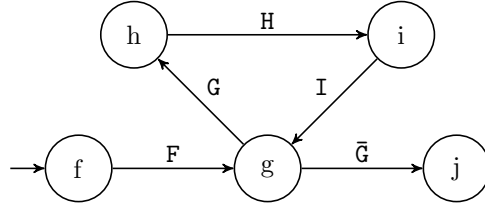
To improve readability, however, we often get rid of labels, we drop $\bar{\mathbf{G}}$ and we re-arrange \mathbf{F} , \mathbf{G} , \mathbf{H} and \mathbf{I} to fit the customary Java notation of for-loops:

```

for (F; G; I) {
  H
}

```

With reference to this notation, \mathbf{F} , \mathbf{G} and \mathbf{I} are said to form the *header* of \mathbf{P} , whereas \mathbf{H} is called its *body*. Since \mathbf{H} is a subroutine, it consists of several commands enjoying several properties. For convenience, however, we can abstract from such internal details and think to \mathbf{H} as a whole. In our language, this is the same as letting \mathbf{H} be merely a command, and $h: \mathbf{H} \rightarrow i$ a statement defining a transition from h to i through \mathbf{H} . Thanks to this assumption, we can here introduce a simpler representation of \mathbf{P} as an automaton:



Notice that GHI identifies a loop that both starts and ends in g . Each time the automaton runs across such loop, we say P *performs an iteration*. Thus we can say that iterations always start with a successful evaluation of G and end with the increment performed by I .

Formally, given $\rho, \rho' \in \mathfrak{E}[\mathbb{P}]$, we define an *iteration of P* to be any sequence

$$\langle \rho, g: G \rightarrow h \rangle \langle \rho, h: H \rightarrow i \rangle \langle \rho', i: I \rightarrow g \rangle \quad (2.50)$$

such that there exists σ and σ' with $\sigma \langle \rho, G \rangle \langle \rho, H \rangle \langle \rho', I \rangle \sigma' \in \mathfrak{S}[\mathbb{P}]$. If we drop the assumption that H is a command, we obtain a broader definition of iteration. Namely, given $\eta \in \mathfrak{S}[\mathbb{H}]$, an iteration of P is any sequence

$$\langle \rho, G \rangle \eta \langle \rho', I \rangle$$

such that there exists σ and σ' with $\sigma \langle \rho, G \rangle \eta \langle \rho', I \rangle \sigma' \in \mathfrak{S}[\mathbb{P}]$. We define the set collecting all the iterations of P to be $e(\mathfrak{S}[\mathbb{P}]) \stackrel{\text{def}}{=} e_g(\mathfrak{S}[\mathbb{P}])$, where:

$$e_\ell(\mathcal{J}) \stackrel{\text{def}}{=} \bigcup \{ e_\ell(\sigma) \mid \sigma \in \mathcal{J} \} \quad (2.51)$$

$$e_\ell(\sigma) \stackrel{\text{def}}{=} \{ \langle \rho, S \rangle \eta \langle \rho', S' \rangle \mid \exists \sigma'. \exists \sigma''. \sigma' \langle \rho, S \rangle \eta \langle \rho', S' \rangle \sigma'' = \sigma \\ \wedge \text{lab}[S] = \ell \wedge \ell \notin \text{lab}[\mathbb{P}(\eta \langle \rho', S' \rangle)] \\ \wedge \text{suc}[S'] = \ell \wedge \ell \notin \text{suc}[\mathbb{P}(\langle \rho, S \rangle \eta)] \} . \quad (2.52)$$

For the sake of simplicity, in the rest of this thesis we assume that iterations have the form shown by (2.50), where H is considered a command.

Along a trace $\sigma \in \mathfrak{S}[\mathbb{P}]$, an iteration $\langle \rho, g: G \rightarrow h \rangle \langle \rho, h: H \rightarrow i \rangle \langle \rho', i: I \rightarrow g \rangle$ is unambiguously identified by a specific value assumed by x . Such value is $\rho(x) = \rho'(x)$. Indeed, x remains constant within each iteration, since neither G nor H assign x , but changes from one iteration to another one, because of I . The set I of all the values assumed by x is called *iteration space* and can be naturally represented on a directed line with origin O .

Sometimes, during the execution of P , we need to assess how many iterations are still to be completed, that is, the number of *residual* iterations. We thereby introduce a function $R_{x, A_2, A_3}: \Sigma[\mathbb{P}] \rightarrow \mathbb{N}$ such that:

$$R_{x, A_2, A_3} \langle \rho, S \rangle = \begin{cases} A[(A_2 + A_3 - x) \text{ div } A_3] \rho - 1 & \text{if } S = I \wedge B[x < A_2] \rho \\ A[(A_2 + A_3 - x) \text{ div } A_3] \rho & \text{if } S \neq I \wedge B[x < A_2] \rho \\ 0 & \text{otherwise.} \end{cases} \quad (2.53)$$

We should note this function as $R_{[x], [A_2], [A_3]}(\rho)$; we however drop double brackets for the sake of simplicity.

In determining residual iterations, we are concerned only with traces whose first statement is a guard statement. We collect such traces in $\mathbf{d}(\mathcal{S}[\mathbb{P}]) \stackrel{\text{def}}{=} \mathbf{d}_g(\mathcal{S}[\mathbb{P}])$, where:

$$\mathbf{d}_\ell(\mathcal{T}) \stackrel{\text{def}}{=} \bigcup \{ \mathbf{d}_\ell(\sigma) \mid \sigma \in \mathcal{T} \} \quad (2.54)$$

$$\mathbf{d}_\ell(\sigma) \stackrel{\text{def}}{=} \{ \varepsilon \} \cup \{ \langle \rho, \mathbf{S} \rangle \sigma'' \mid \exists \sigma'. \exists \sigma'''. \sigma' \langle \rho, \mathbf{S} \rangle \sigma'' \sigma''' = \sigma \wedge \text{lab}[\mathbf{S}] = \ell \} . \quad (2.55)$$

Then, given $\sigma \in \mathbf{d}(\mathcal{S}[\mathbb{P}])$, we let the *number of residual iterations at σ* be:

$$R_{\mathbf{x}, \mathbf{A}_2, \mathbf{A}_3}(\sigma) \stackrel{\text{def}}{=} R_{\mathbf{x}, \mathbf{A}_2, \mathbf{A}_3}(\vdash \sigma) . \quad (2.56)$$

If $|\sigma| > 0$ and $0 \leq j < |\sigma|$, we define the *number of actual iterations of σ at σ_j* to be:

$$\#_{\mathbf{x}, \mathbf{A}_2, \mathbf{A}_3}(\sigma, \sigma_j) \stackrel{\text{def}}{=} R_{\mathbf{x}, \mathbf{A}_2, \mathbf{A}_3}(\sigma) - R_{\mathbf{x}, \mathbf{A}_2, \mathbf{A}_3}(\sigma_j) . \quad (2.57)$$

Consequently, we can define the *number of actual iterations of σ* to be:

$$\#_{\mathbf{x}, \mathbf{A}_2, \mathbf{A}_3}(\varepsilon) \stackrel{\text{def}}{=} 0 \quad (2.58)$$

$$\#_{\mathbf{x}, \mathbf{A}_2, \mathbf{A}_3}(\sigma) \stackrel{\text{def}}{=} \#_{\mathbf{x}, \mathbf{A}_2, \mathbf{A}_3}(\sigma, \vdash \sigma) . \quad (2.59)$$

Notice that by definition $\#_{\mathbf{x}, \mathbf{A}_2, \mathbf{A}_3}(\sigma)$ is nonnegative, and $\#_{\mathbf{x}, \mathbf{A}_2, \mathbf{A}_3}(\sigma) = R_{\mathbf{x}, \mathbf{A}_2, \mathbf{A}_3}(\sigma)$ if and only if σ is a maximal trace of \mathbb{P} . All in all we have:

$$0 \leq \#_{\mathbf{x}, \mathbf{A}_2, \mathbf{A}_3}(\sigma) \leq R_{\mathbf{x}, \mathbf{A}_2, \mathbf{A}_3}(\sigma) . \quad (2.60)$$

Moreover, we obtain a compositionality property of $\#_{\mathbf{x}, \mathbf{A}_2, \mathbf{A}_3}$ through the following

Proposition 2.5. *Let \mathbb{P} be a for-loop with guard label g . Let $\sigma \in \mathbf{d}(\mathcal{S}[\mathbb{P}])$. Assume there exist $\sigma', \sigma'' \in \mathbf{d}(\mathcal{S}[\mathbb{P}])$ such that $\sigma = \sigma' \sigma''$. Then $\#_{\mathbf{x}, \mathbf{A}_2, \mathbf{A}_3}(\sigma) = \#_{\mathbf{x}, \mathbf{A}_2, \mathbf{A}_3}(\sigma') + \#_{\mathbf{x}, \mathbf{A}_2, \mathbf{A}_3}(\sigma'')$.*

Proof. We consider three cases. For the sake of simplicity, we write $\#$ for $\#_{\mathbf{x}, \mathbf{A}_2, \mathbf{A}_3}$, and R for $R_{\mathbf{x}, \mathbf{A}_2, \mathbf{A}_3}$, and so on.

- If $\sigma' = \varepsilon$, then $\# \sigma' = 0$ by (2.58). Furthermore, we have $\sigma = \sigma' \sigma'' = \varepsilon \sigma'' = \sigma''$, whence $\# \sigma = \# \sigma'' = 0 + \# \sigma'' = \# \sigma' + \# \sigma''$.
- If $\sigma'' = \varepsilon$, then $\# \sigma'' = 0$ by (2.58). Furthermore, we have $\sigma = \sigma' \sigma'' = \sigma' \varepsilon = \sigma'$, whence $\# \sigma = \# \sigma' = \# \sigma' + 0 = \# \sigma' + \# \sigma''$.
- Let $\sigma' \neq \varepsilon$ and $\sigma'' \neq \varepsilon$. Then we have $\sigma'' = \sigma''' \langle \rho, i: \mathbf{x} := \mathbf{x} + \mathbf{A}_3 \rightarrow g \rangle$ and $\sigma' = \langle \rho[\mathbf{x} := x + a_3], \mathbf{S} \rangle \sigma''''$, where $\mathbf{S} \neq i: \mathbf{x} := \mathbf{x} + \mathbf{A}_3 \rightarrow g$. Moreover, let $x = \rho(\mathbf{x})$, $a_2 = \mathbf{A}[\mathbf{A}_2] \rho$ and $a_3 = \mathbf{A}[\mathbf{A}_3] \rho$.

We have

$$R(\sigma) = R(\sigma')$$

and

$$R(\vdash \sigma) = R(\vdash \sigma')$$

and

$$\begin{aligned}
 R(\neg\sigma') &= R(\neg\sigma''' \langle \rho, i: \mathbf{x} := \mathbf{x} + \mathbf{A}_3 \rightarrow \mathbf{g} \rangle) \\
 &= R \langle \rho, i: \mathbf{x} := \mathbf{x} + \mathbf{A}_3 \rightarrow \mathbf{g} \rangle && \text{by (2.4)} \\
 &= A[\![\mathbf{A}_2 + \mathbf{A}_3 - \mathbf{x} \text{ div } \mathbf{A}_3]\!] \rho - 1 && \text{by (2.53)} \\
 &= (a_2 + a_3 - x) \text{ div } a_3 - 1 \\
 &= (a_2 + a_3 - x - a_3) \text{ div } a_3 \\
 &= A[\![\mathbf{A}_2 + \mathbf{A}_3 - \mathbf{x} - \mathbf{A}_3 \text{ div } \mathbf{A}_3]\!] \rho \\
 &= A[\![\mathbf{A}_2 + \mathbf{A}_3 - \mathbf{x} \text{ div } \mathbf{A}_3]\!] \rho[\mathbf{x} := x + a_3] \\
 &= R \langle \rho[\mathbf{x} := x + a_3], \mathbf{S} \rangle && \text{by (2.53)} \\
 &= R(\langle \rho[\mathbf{x} := x + a_3], \mathbf{S} \rangle \sigma''') && \text{by (2.56)} \\
 &= R(\sigma'') .
 \end{aligned}$$

Thus:

$$\begin{aligned}
 \#\sigma &= \#(\sigma, \neg\sigma) && \text{by (2.59)} \\
 &= R(\sigma) - R(\neg\sigma) && \text{by (2.57)} \\
 &= R(\sigma') - R(\neg\sigma) \\
 &= R(\sigma') - R(\neg\sigma) + 0 \\
 &= R(\sigma') - R(\neg\sigma) + (R(\neg\sigma') - R(\neg\sigma')) \\
 &= (R(\sigma') - R(\neg\sigma')) + (R(\neg\sigma') - R(\neg\sigma)) \\
 &= \#(\sigma', \neg\sigma') + (R(\neg\sigma') - R(\neg\sigma)) && \text{by (2.57)} \\
 &= \#\sigma' + (R(\neg\sigma') - R(\neg\sigma)) && \text{by (2.59)} \\
 &= \#\sigma' + (R(\sigma'') - R(\neg\sigma)) \\
 &= \#\sigma' + (R(\sigma'') - R(\neg\sigma'')) \\
 &= \#\sigma' + \#(\sigma'', \neg\sigma'') && \text{by (2.57)} \\
 &= \#\sigma' + \#\sigma'' && \text{by (2.59)}
 \end{aligned}$$

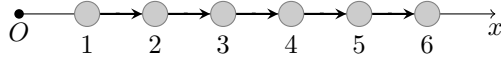
□

As a first example of for-loop, let us consider the following program P:

```

for (x := 1; x ≤ 6; x := x + 1) {
  true
}
    
```

We notice that P is in normal form. A for-loop with index variable \mathbf{x} is in *normal form* if and only if $\mathbf{F} = \mathbf{x} := 1$ and $\mathbf{I} = \mathbf{x} := \mathbf{x} + 1$. The iteration space of P is $I = \{1, 2, 3, 4, 5, 6\}$ and can naturally be represented as:



There is a gray dot for each possible value of \mathbf{x} . Each one stands for a different iteration. Black arrows capture the order in which iterations are performed.

We have a *loop nest* if the body of a for-loop is itself a for-loop. The loop nest is in normal form if and only if both the outer and the inner loop are in normal form. A very general nest involving only two for-loops is the following:

```

for (x := A11; x ≤ A12; x := x + A13) {
  for (y := A21; y ≤ A22; y := y + A23) {
    H
  }
}

```

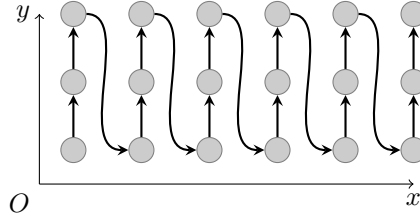
If A_{21} , A_{23} or A_{22} depend on x , then the iteration space of the inner loop is itself dependent on x . Hence we say that the inner loop is parametrized by the index of the outer loop. Let us consider a more instantiated version of our loop nest:

```

for (x := 1; x ≤ 6; x := x + 1) {
  for (y := 1; y ≤ 3; y := y + 1) {
    H
  }
}

```

Since two loops are involved, we have two index variables (x and y). We thereby identify iterations by using two values, collected in two-dimensional *index vectors*. We define the iteration space I to be the set of all its index vectors and we graphically represent it as a gray dotted two-dimensional polyhedron:



In I an ordering is established that reflects the order in which the iterations of the loop nest take place. It is the lexicographical order \preceq which, given $(x, y), (x', y') \in I$, is defined as:

$$(x, y) \preceq (x', y') \text{ if and only if } x < x' \vee (x = x' \wedge y \leq y') . \quad (2.61)$$

We say that such ordering describes a column-wise *traversal* of I and we represent it through black arrows.

Both the iteration space traversal and the automaton representation suggest that a for-loop P :

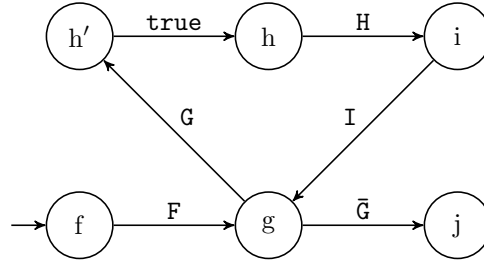
- 1) starts with command F , which initializes x ;
- 2) performs a number of iterations (possibly none);
- 3) ends with \bar{G} .

Sometimes however P is unable to get to the end for several reasons:

- the execution of P gets stuck inside H ;
- the execution of P keeps on looping within H ;
- H is the empty set.

The last reason may be quite unexpected, as almost all programming languages simply ignore empty bodies and jump at once from G to I ; this can be achieved in our language by letting $H \stackrel{\text{def}}{=} \{ h: \text{true} \rightarrow i \}$. At any rate, as the non-termination of H is irrelevant to this thesis, we are not concerned with it.

According to our definition of for-loop, H is called only by G through h . None of the commands in H is allowed to transition back to h . In case we need to simulate this behavior, we can let h' be a fresh label with respect to P , and turn P into a new for-loop



where $g: G \rightarrow h$ has been replaced by $g: G \rightarrow h'$, and $h: H \rightarrow i$ is replaced by $h': \{ h': \text{true} \rightarrow h \} \cup H \rightarrow i$.

2.15 Abstract Interpretation

Sometimes we want to check a program against a property. For instance, given a program P , we may ask: *does P terminate?* Termination is generally understood as a binary property, since a program is expected to either terminate or not. Thus the set C of the possible answers has only two elements:

$$C = \{ \text{yes}, \text{no} \} .$$

A set of answers can include more than two elements. It can even be infinite. In case we have a property whose set of answers is a singleton, the property is trivial because there is no actual choice for the answer. Properties with an empty set of answers seem to be of no use.

Not all the answers are supposed to carry the same amount of information. Back to our example, it may well happen that our termination analysis ends without any positive answer about termination. In such case, it is unknown to us if P will ever terminate; we thereby should answer: “unknown”. Notice that this answer is vaguer than “yes” or “no”. The *is vaguer* predicate is naturally realized by an order relation \leq_C on a set

$$C = \{ \text{yes}, \text{no}, \text{unknown} \} ,$$

such that

$$\begin{aligned} \text{yes} &\leq_C \text{unknown} \\ \text{no} &\leq_C \text{unknown} . \end{aligned}$$

We expect our termination analysis to be correct, that is:

- if it returns “yes”, then P actually terminates;
- if it returns “no”, then P actually does not terminate.

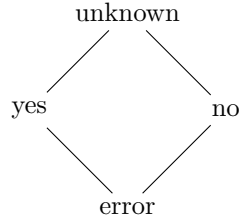
Though correct, our termination analysis might however be ill-conceived, meaning that an error might occur before completion, or the analysis might keep on running without delivering a result. In either case, we should answer “error” and urge a revision of our termination analysis. It seems appropriate to include “error” in our set C . Hence we have:

$$C = \{ \text{yes, no, unknown, error} \}$$

and

$$\begin{aligned} \text{error} &\leq_C \text{yes} \\ \text{error} &\leq_C \text{no} . \end{aligned}$$

$\langle C, \leq_C \rangle$ is a poset (it is in fact a complete lattice) and can be graphically represented as



where $c \leq_C c'$ if and only if a line segment exists with c as lower end point and c' as upper end point.

Suppose now we need to check our program P against the following property: *is it okay to run our termination analysis on P, or does it raise an error?* The correspondent poset of answers is

$$A = \{ \text{error, okay} \}$$

with

$$\text{error} \leq_A \text{okay} .$$

The graphical representation for this poset is:



To answer our new property, we can take advantage of the answer we found for the previous property. In particular, we can:

- return “okay” if and only if the answer to the first property is “yes”, “no” or “unknown”;
- return “error” if and only if the answer to the first property is “error”.

We can model this conduct by introducing a function $\alpha: C \rightarrow A$ such that:

$$\begin{aligned}\alpha(\text{unknown}) &\stackrel{\text{def}}{=} \text{okay} \\ \alpha(\text{yes}) &\stackrel{\text{def}}{=} \text{okay} \\ \alpha(\text{no}) &\stackrel{\text{def}}{=} \text{okay} \\ \alpha(\text{error}) &\stackrel{\text{def}}{=} \text{error} .\end{aligned}$$

Thus, once we know the answer to the first property, we can immediately derive the answer to the second one.

Notice however this does not work the other way round. Suppose we have found an answer to the second property. If the answer is “error”, we can return “error” for the first property too. In case the answer is “okay”, however, we cannot say if the first property evaluates to “yes”, “no” or “unknown”. A safe strategy is to opt for the vaguest answer: “unknown”. Again we can model this process through a function $\gamma: A \rightarrow C$ such that:

$$\begin{aligned}\gamma(\text{okay}) &\stackrel{\text{def}}{=} \text{unknown} \\ \gamma(\text{error}) &\stackrel{\text{def}}{=} \text{error} .\end{aligned}$$

Because for the sake of safety we choose the vaguest answer, the composition of γ after α usually results in a loss of information:

$$\gamma(\alpha(\text{unknown})) = \gamma(\text{okay}) = \text{unknown} \quad (2.62)$$

$$\gamma(\alpha(\text{yes})) = \gamma(\text{okay}) = \text{unknown} \quad (2.63)$$

$$\gamma(\alpha(\text{no})) = \gamma(\text{okay}) = \text{unknown} \quad (2.64)$$

$$\gamma(\alpha(\text{error})) = \gamma(\text{error}) = \text{error} . \quad (2.65)$$

In (2.63) “yes” is mapped to “unknown”. Likewise, in (2.64) “no” is mapped to “unknown”. There is a loss of information because “unknown” is vaguer than “yes” or “no”. The loss is caused by α , which makes “yes”, “no” and “unknown” collapse to “okay”. Once the loss has occurred, γ is not able to recover it any longer.

In (2.62), on the other side, there is no loss of information. This means that “unknown” is just an alias for “okay”. A similar arguments holds for (2.65). Every answer in A is thereby included in C , possibly with a different name. Since C is actually larger than A , we say that:

- $\langle C, \leq_C \rangle$ is a refinement of $\langle A, \leq_A \rangle$;
- $\langle A, \leq_A \rangle$ is an approximation of $\langle C, \leq_C \rangle$.

Designing, refining and approximating posets of answers for program properties is the purpose of *abstract interpretation*, which was first introduced in 1979 by Patrick and Radhia Cousot [27], [29], [30], [31], [32].

In abstract interpretation, $\langle C, \leq_C \rangle$ is called the *concrete domain*. It is supposed to collect the answers to a property which tries to describe all the information known about programs, that is, their formal semantics. For instance, we may let $\langle C, \leq_C \rangle = \langle \wp(\Sigma^+), \subseteq \rangle$, thus letting the concrete domain capture trace semantics; then, given a program P , the correct answer is $\mathbb{S}[P]$.

Poset $\langle A, \leq_A \rangle$ is the *abstract domain*. It is supposed to approximate $\langle C, \leq_C \rangle$. Continuing our example, we may let $\langle A, \leq_A \rangle$ model the denotational semantics of programs. Such semantics is vaguer than trace semantics as, for each trace, it retains only the environments of the first and the last state. So we expect A to collect pairs of environments: $\langle A, \leq_A \rangle = \langle \mathfrak{E} \times \mathfrak{E}, \subseteq \rangle$. Given program P , if we know its trace semantics $\mathcal{S}[[P]]$, then we can easily determine its denotational semantics by computing the following approximation:

$$\{ (\rho, \rho') \mid \exists \mathcal{S}. \exists \mathcal{S}'. \vdash \sigma = \langle \rho, \mathcal{S} \rangle \wedge \dashv \sigma = \langle \rho', \mathcal{S}' \rangle \wedge \sigma \in \mathcal{S}[[P]] \} .$$

The order relations on C and A qualitatively model relative precision between elements.

Function α and γ are the *abstraction map* and *concretization map* respectively. They are required to form an *adjunction*, that is,

$$\forall c \in C, a \in A. \alpha(c) \leq_A a \iff c \leq_C \gamma(a) .$$

If this is the case, then we have a *Galois connection*, noted

$$C \xleftrightarrow[\alpha]{\gamma} A .$$

In particular, if α is surjective, we have a *Galois insertion*, noted

$$C \xleftrightarrow[\alpha]{\gamma} \dashv\dashv A .$$

Thus, in a Galois insertion, each element in the abstract domain is useful at expressing the property of at least one program specification.

It can be proved that we always have a Galois insertion whenever α, γ are monotonic, $c \leq_C \gamma \circ \alpha(c)$ and $\alpha \circ \gamma(a) = a$. Monotonicity ensures that α and γ preserves the relative ordering between elements. The second condition expresses the intuitive fact that what is loosed by α cannot be recovered by γ any longer. The last condition ensures that the information carried by abstract elements can always be recovered entirely.

The interest in considering approximations is based upon the Rice theorem [56], stating that several program properties, like e.g. trace semantics, are not decidable. Approximations help at attaining decidability by narrowing the set of answers. A drawback is that they entail a loss of information: in our example above, we obtain (ρ, ρ') from σ by discarding all statements, and all environments but the first and the last one. We can tolerate some loss of information in order to attain decidability. However, if we discard too much information, we are likely to get correct answers to properties which are not significant at all. The challenge of abstract interpretation is to improve the trade-off between decidability and informativeness of properties.

2.16 A Framework for Program Transformations

Programs can be obtained out of other programs. A *syntactic transformer* is an operator $\mathfrak{t}: \mathbb{P} \rightarrow \mathbb{P}$ which takes as input a subject program P and, upon termination, produces as output a transformed program P' . In symbols:

$$\mathfrak{t}: \mathbb{P} \mapsto \mathbb{P}' .$$

Although syntactic, many a transformer is defined in terms of semantic criteria. This means that \mathfrak{t} is usually understood as making use of the program semantics $\mathcal{S}[\mathbb{P}] \in \wp(\Sigma^{\rightarrow})$. In fact, we can say that \mathfrak{t} induces a corresponding *semantic transformer* modeled by an operator $\mathfrak{t}: \wp(\Sigma^{\rightarrow}) \rightarrow \wp(\Sigma^{\rightarrow})$ such that

$$\mathfrak{t}: \mathcal{S}[\mathbb{P}] \mapsto \mathcal{S}[\mathbb{P}'] .$$

Cousot and Cousot [33] propose to interrelate \mathfrak{t} and \mathfrak{t} by means of abstract interpretation. Their approach is novel as they consider a program to be an approximation of its semantics. Indeed, the statements of a program record the existence of the variables, but not the sequence of their successive values along traces, or they usually record the chaining of commands, but not their exact sequences of execution [33]. The proposed abstract domain is $\langle \mathbb{P}/\equiv, \sqsubseteq \rangle$, where \mathbb{P}/\equiv collects classes of equivalent programs. We let the order relation \sqsubseteq on \mathbb{P}/\equiv be such that, given two programs \mathbb{P} and \mathbb{P}' :

$$\mathbb{P} \sqsubseteq \mathbb{P}' \text{ if and only if } \exists \mathbb{P}'' . \exists \mathbb{P}''' . \begin{cases} \mathbb{P}'' \equiv \mathbb{P} \\ \mathbb{P}''' \equiv \mathbb{P}' \\ \mathcal{S}[\mathbb{P}''] \subseteq \mathcal{S}[\mathbb{P}'''] . \end{cases}$$

The Galois insertion between the concrete domain and our new abstract domain is:

$$\langle \wp(\Sigma^{\rightarrow}), \subseteq \rangle \xleftarrow[\mathbb{P}^*]{\mathcal{S}^*} \langle \mathbb{P}/\equiv, \sqsubseteq \rangle , \quad (2.66)$$

where

$$\begin{aligned} \mathbb{P}^*(\mathcal{J}) &\stackrel{\text{def}}{=} [\mathbb{P}(\mathcal{J})]_{\equiv} \\ \mathcal{S}^*([\mathbb{P}]_{\equiv}) &\stackrel{\text{def}}{=} \bigcup_{\mathbb{P}' \in [\mathbb{P}]_{\equiv}} \mathcal{S}[\mathbb{P}'] . \end{aligned}$$

Cousot and Cousot also observe [33] that \mathfrak{t} acts as an abstraction map on $\wp(\Sigma^{\rightarrow})$, because it causes a loss of information about the subject program and its semantics. Thus they outline the following Galois connection

$$\langle \wp(\Sigma^{\rightarrow}), \subseteq \rangle \xleftarrow[\mathfrak{t}]{\gamma_{\mathfrak{t}}} \langle \wp(\Sigma^{\rightarrow}), \subseteq \rangle , \quad (2.67)$$

where $\gamma_{\mathfrak{t}}$ is supposed to be the correspondent concretization map. It is known that (2.67) holds if \mathfrak{t} is \emptyset -strict and additive [30]. We can attain both requirements by enforcing the following constraints to \mathfrak{t} :

$$\mathfrak{t}(\mathcal{J}) \stackrel{\text{def}}{=} \{ \mathfrak{t}(\sigma) \in \mathcal{J} \mid \sigma \in \mathcal{J} \} \quad (2.68)$$

$$\mathfrak{t}(\sigma) \stackrel{\text{def}}{=} \lambda i. \mathfrak{t}(\sigma_i) . \quad (2.69)$$

We prove our claim by stating the following

Proposition 2.6. *The semantic transformer \mathfrak{t} is \emptyset -strict. Formally:*

$$\mathfrak{t}(\emptyset) = \emptyset .$$

Proof. Immediate from (2.68). \square

Proposition 2.7. *The semantic transformer \mathfrak{t} is additive in $\langle \wp(\Sigma^\rightarrow), \subseteq \rangle$. Formally:*

$$\mathfrak{t}\left(\bigcup F^n \llbracket \mathbb{P} \rrbracket \emptyset\right) = \bigcup \mathfrak{t}(F^n \llbracket \mathbb{P} \rrbracket \emptyset) .$$

Proof. We have:

$$\begin{aligned} \mathfrak{t}\left(\bigcup F^n \llbracket \mathbb{P} \rrbracket \emptyset\right) &= \mathfrak{t}\left\{ \sigma \mid \sigma \in \bigcup F^n \llbracket \mathbb{P} \rrbracket \emptyset \right\} \\ &= \left\{ \mathfrak{t}(\sigma) \mid \sigma \in \bigcup F^n \llbracket \mathbb{P} \rrbracket \emptyset \right\} && \text{by (2.68)} \\ &= \left\{ \mathfrak{t}(\sigma) \mid \exists n \in \mathbb{N}. \sigma \in F^n \llbracket \mathbb{P} \rrbracket \emptyset \right\} \\ &= \bigcup \left\{ \mathfrak{t}(\sigma) \mid \sigma \in F^n \llbracket \mathbb{P} \rrbracket \emptyset \right\} \\ &= \bigcup \mathfrak{t}(\{ \sigma \mid \sigma \in F^n \llbracket \mathbb{P} \rrbracket \emptyset \}) && \text{by (2.68)} \\ &= \bigcup \mathfrak{t}(F^n \llbracket \mathbb{P} \rrbracket \emptyset) . \end{aligned}$$

\square

By composing (2.66) and (2.67), Cousot and Cousot obtain the following framework:

$$\begin{array}{ccc} \mathbb{P}/\equiv & & \mathbb{P}/\equiv \\ \mathcal{S}^* \downarrow & & \uparrow \mathcal{P}^* \\ \wp(\Sigma^\rightarrow) & \xrightarrow{\mathfrak{t}} & \wp(\Sigma^\rightarrow) \end{array}$$

Then they derive \mathfrak{t} as an approximation of the program transformer \mathfrak{t}^* they obtain from $\mathfrak{p}^* \circ \mathfrak{t} \circ \mathcal{S}^*$ [33]:

$$\begin{array}{ccc} [\mathbb{P}]/\equiv & \xrightarrow{\mathfrak{t}^*} & [\mathbb{P}']/\equiv \\ \mathcal{S}^* \downarrow & & \uparrow \mathcal{P}^* \\ \mathcal{J} & \xrightarrow{\mathfrak{t}} & \mathcal{J}' \end{array}$$

They also formalize the correctness of \mathfrak{t}^* through an *observational abstraction* $\alpha_{\mathcal{O}}$, which is a mapping from the concrete domain to some abstract domain \mathcal{O} . In particular, \mathfrak{t}^* is correct if and only if $\alpha_{\mathcal{O}}(\mathcal{J}) = \alpha_{\mathcal{O}}(\mathfrak{t}(\mathcal{J})) = \alpha_{\mathcal{O}}(\mathcal{J}')$. The fact that observer $\alpha_{\mathcal{O}}$ is not able to distinguish between $\alpha_{\mathcal{O}}(\mathcal{J})$ and $\alpha_{\mathcal{O}}(\mathfrak{t}(\mathcal{J}))$ has a twofold significance:

- it can mean that \mathfrak{t} keeps intact the content that $\alpha_{\mathcal{O}}$ still wants to observe in the transformed program;
- it can also mean that \mathfrak{t} acts just where the inspection ability of $\alpha_{\mathcal{O}}$ lacks.

The latter point is especially interesting for the sake of obfuscation. We discuss it at length in Section 3.5.

2.17 Deriving Semantic Transformers

In the previous Section we presented the Cousots' framework as a tool for deriving syntactic transformers from semantic ones. In this Section we outline the inverse approach by revising notions from the literature [28], [33]. In particular, we suppose we are given a syntactic transformer \mathfrak{t} and look for a semantic transformer \mathfrak{s} such that the following diagram commutes:

$$\begin{array}{ccc} \mathbb{P} & \xrightarrow{\mathfrak{t}} & \mathbb{P} \\ \mathfrak{S} \downarrow & & \downarrow \mathfrak{S} \\ \wp(\Sigma^*) & \xrightarrow{\mathfrak{t}} & \wp(\Sigma^*) \end{array}$$

If we redraw this diagram in terms of a program $P \in \mathbb{P}$, we obtain the following commutation of mappings:

$$\begin{array}{ccc} P & \xrightarrow{\mathfrak{t}} & \mathfrak{t}[[P]] \\ \mathfrak{S} \downarrow & & \downarrow \mathfrak{S} \\ \mathfrak{S}[[P]] & \xrightarrow{\mathfrak{t}} & \mathfrak{t}(\mathfrak{S}[[P]]) = \mathfrak{S}[[\mathfrak{t}[[P]]]] \end{array}$$

The equality on the right bottom

$$\mathfrak{t}(\mathfrak{S}[[P]]) = \mathfrak{S}[[\mathfrak{t}[[P]]]] \quad (2.70)$$

demands that the semantic transformation \mathfrak{t} of the semantics \mathfrak{S} of the subject program P is just the semantics \mathfrak{S} of the syntactically transformed program $\mathfrak{t}[[P]]$. In order to successfully prove the equality, we take advantage of the fixpoint form of \mathfrak{S} as defined by (2.47). Hence we have:

$$\mathfrak{t}\left(\text{lfp}_{\emptyset}^{\subseteq} F[[P]]\right) = \text{lfp}_{\emptyset}^{\subseteq} F[[\mathfrak{t}[[P]]]] \quad (2.71)$$

We can restate this equality in an equivalent form:

$$\mathfrak{t}\left(\bigcup_{n \in \mathbb{N}} F^n[[P]]\emptyset\right) = \bigcup_{n \in \mathbb{N}} F^n[[\mathfrak{t}[[P]]]]\emptyset \quad (2.72)$$

In the rest of this Section we drop the pedix of \bigcup for the sake of simplicity. We observe that only the right hand side is a pure fixpoint construction, whereas the left hand side is not, because of the semantic transformer \mathfrak{t} .

To make a pure fixpoint construction out of the left hand side, we need to introduce a fixpoint operator $F_{\mathfrak{t}}[[P]]: \Sigma^* \rightarrow \Sigma^*$ which is required to satisfy the following condition:

$$F_{\mathfrak{t}}[[P]](\mathfrak{t}(\mathcal{J})) = \mathfrak{t}(F[[P]]\mathcal{J}) \quad (2.73)$$

This equality is known as *local commutation condition* [33]. We now expand it in order to derive a good definition for $F_{\mathfrak{t}}[[P]]$:

$$\begin{aligned}
F_t[\mathbb{P}](t(\mathcal{J})) &= t(F[\mathbb{P}]\mathcal{J}) \\
&= t(\mathcal{J}[\mathbb{P}] \cup \{ \sigma s s' \mid \sigma s \in \mathcal{J} \wedge s' \in S[\mathbb{P}] s \}) && \text{by (2.46)} \\
&= t(\mathcal{J}[\mathbb{P}]) \cup t(\{ \sigma s s' \mid \sigma s \in \mathcal{J} \wedge s' \in S[\mathbb{P}] s \}) && \text{by (2.68)}.
\end{aligned}$$

We obtained a union of two terms. We consider them separately. For the first term we have:

$$\begin{aligned}
t(\mathcal{J}[\mathbb{P}]) &= t(\{ \langle \rho, \mathbf{S} \rangle \in \Sigma[\mathbb{P}] \mid \text{lab}[\mathbf{S}] \in \mathcal{L}[\mathbb{P}] \}) && \text{by (2.45)} \\
&= t(\{ \langle \rho, \mathbf{S} \rangle \mid \langle \rho, \mathbf{S} \rangle \in \Sigma[\mathbb{P}] \wedge \text{lab}[\mathbf{S}] \in \mathcal{L}[\mathbb{P}] \}) \\
&= \{ t\langle \rho, \mathbf{S} \rangle \mid \langle \rho, \mathbf{S} \rangle \in \Sigma[\mathbb{P}] \wedge \text{lab}[\mathbf{S}] \in \mathcal{L}[\mathbb{P}] \} && \text{by (2.68)}.
\end{aligned}$$

For the second term we have:

$$\begin{aligned}
t(\{ \sigma s s' \mid \sigma s \in \mathcal{J} \wedge s' \in S[\mathbb{P}] s \}) \\
&= t\{ \sigma s s' \mid \sigma s \in \mathcal{J} \wedge s' \in S[\mathbb{P}] s \wedge s = \neg \sigma s \} && \text{by (2.4)} \\
&= \{ t(\sigma s s') \mid \sigma s \in \mathcal{J} \wedge s' \in S[\mathbb{P}] s \wedge s = \neg \sigma s \} && \text{by (2.68)} \\
&= \{ t(\sigma s) t(s') \mid \sigma s \in \mathcal{J} \wedge s' \in S[\mathbb{P}] s \wedge s = \neg \sigma s \} && \text{by (2.69)} \\
&= \{ t(\sigma s) t(s') \mid t(\sigma s) \in t(\mathcal{J}) \wedge s' \in S[\mathbb{P}] s \wedge s = t'(t(\sigma s)) \} && \text{by (2.68)} \\
&= \{ \eta t(s') \mid \eta \in t(\mathcal{J}) \wedge s' \in S[\mathbb{P}] s \wedge s = t'(\eta) \} .
\end{aligned}$$

Notice that we introduced an *auxiliary function* $t': \Sigma^* \rightarrow \Sigma$. This function is broadly meant to take a trace $\eta = t(\sigma s)$, undo t to obtain σs , and return the last state s . In fact, we do not expect t' to yield s , but any state s' such that $S[\mathbb{P}] s' = S[\mathbb{P}] s$. We thereby require t' to only satisfy the following constraint:

$$S[\mathbb{P}](t'(t(\sigma s))) = S[\mathbb{P}] s . \quad (2.74)$$

By grouping the two terms together, we can restate condition (2.73) as follows:

$$\begin{aligned}
F_t[\mathbb{P}](t(\mathcal{J})) &= \{ t\langle \rho, \mathbf{S} \rangle \mid \langle \rho, \mathbf{S} \rangle \in \Sigma[\mathbb{P}] \wedge \text{lab}[\mathbf{S}] \in \mathcal{L}[\mathbb{P}] \} \\
&\quad \cup \{ \eta t(s') \mid \eta \in t(\mathcal{J}) \wedge s' \in S[\mathbb{P}] s \wedge s = t'(\eta) \} . \quad (2.75)
\end{aligned}$$

We use this result to derive the definition of $F_t[\mathbb{P}]$. Given $\mathcal{J} \subseteq \Sigma^*$, we let:

$$\begin{aligned}
F_t[\mathbb{P}]\mathcal{J} &\stackrel{\text{def}}{=} \{ t\langle \rho, \mathbf{S} \rangle \mid \langle \rho, \mathbf{S} \rangle \in \Sigma[\mathbb{P}] \wedge \text{lab}[\mathbf{S}] \in \mathcal{L}[\mathbb{P}] \} \\
&\quad \cup \{ \eta t(s') \mid \eta \in \mathcal{J} \wedge s' \in S[\mathbb{P}] s \wedge s = t'(\eta) \} . \quad (2.76)
\end{aligned}$$

It is immediate to verify that this definition of $F_t[\mathbb{P}]$ complies with the local commutation condition (2.73).

We are now ready to turn the left hand side of (2.72) into a pure fixpoint construction. The procedure we carry out is a mere application of the fixpoint transfer theorem proved by Kleene and reported by Cousot & Cousot [28]:

$$\begin{aligned}
t\left(\bigcup F^n[\mathbb{P}]\emptyset\right) &= \bigcup t(F^n[\mathbb{P}]\emptyset) && \text{by Proposition 2.7} \\
&= \bigcup F_t^n[\mathbb{P}](t(\emptyset)) && \text{by (2.73)} \\
&= \bigcup F_t^n[\mathbb{P}]\emptyset && \text{by Proposition 2.6.}
\end{aligned}$$

By transitivity and commutativity of $=$, we can rewrite this chain of equations simply as:

$$\bigcup F_t^n \llbracket \mathbb{P} \rrbracket \emptyset = t \left(\bigcup F^n \llbracket \mathbb{P} \rrbracket \emptyset \right) . \quad (2.77)$$

Moreover, by transitivity of $=$, from (2.77) and (2.72) we get

$$\bigcup F_t^n \llbracket \mathbb{P} \rrbracket \emptyset = \bigcup F^n \llbracket t \llbracket \mathbb{P} \rrbracket \emptyset \rrbracket , \quad (2.78)$$

that is, we get an equality whose both sides are pure fixpoint constructions. By definition of \bigcup , this equality holds if the following sufficient condition is true:

$$\forall n \in \mathbb{N}. F_t^n \llbracket \mathbb{P} \rrbracket \emptyset = F^n \llbracket t \llbracket \mathbb{P} \rrbracket \emptyset \rrbracket . \quad (2.79)$$

Indeed, such condition is stronger than (2.78) but, unlike (2.78), it is amenable to be proved by induction on $n \in \mathbb{N}$.

In the next chapters, we present some known syntactic transformers and process them in our framework. In particular, for each syntactic transformer t :

- (i) we define a semantic transformer t and an auxiliary function t' ;
- (ii) we show the commutation equality

$$t(\mathcal{S} \llbracket \mathbb{P} \rrbracket) = \mathcal{S} \llbracket t \llbracket \mathbb{P} \rrbracket \rrbracket \quad (2.70)$$

restated in equivalent form

$$\bigcup F_t^n \llbracket \mathbb{P} \rrbracket \emptyset = \bigcup F^n \llbracket t \llbracket \mathbb{P} \rrbracket \emptyset \rrbracket \quad (2.78)$$

by proving its sufficient condition

$$\forall n \in \mathbb{N}. F_t^n \llbracket \mathbb{P} \rrbracket \emptyset = F^n \llbracket t \llbracket \mathbb{P} \rrbracket \emptyset \rrbracket \quad (2.79)$$

by induction on $n \in \mathbb{N}$.

It is trivial to notice that $F_t^0 \llbracket \mathbb{P} \rrbracket \emptyset = \emptyset = F^0 \llbracket t \llbracket \mathbb{P} \rrbracket \emptyset \rrbracket$. Thus the base of the induction is always true.

To show $F_t^{n+1} \llbracket \mathbb{P} \rrbracket \emptyset = F^{n+1} \llbracket t \llbracket \mathbb{P} \rrbracket \emptyset \rrbracket$, assume by inductive hypothesis that $F_t^n \llbracket \mathbb{P} \rrbracket \emptyset = \mathcal{J} = F^n \llbracket t \llbracket \mathbb{P} \rrbracket \emptyset \rrbracket$. By (2.76) we have that $\sigma \in F_t^{n+1} \llbracket \mathbb{P} \rrbracket \emptyset$ if and only if

$$\sigma \in t(\mathcal{J} \llbracket \mathbb{P} \rrbracket) \vee \sigma \in \{ \eta t(s) \mid \eta \in \mathcal{J} \wedge s \in \mathcal{S} \llbracket \mathbb{P} \rrbracket s' \wedge s' = t'(\eta) \} .$$

On the other side, by (2.46), we have that $\sigma \in F^{n+1} \llbracket t \llbracket \mathbb{P} \rrbracket \emptyset \rrbracket$ if and only if

$$\sigma \in \mathcal{J} \llbracket t \llbracket \mathbb{P} \rrbracket \rrbracket \vee \sigma \in \{ \sigma' r' r \mid \sigma' r' \in \mathcal{J} \wedge r \in \mathcal{S} \llbracket t \llbracket \mathbb{P} \rrbracket \rrbracket r' \} ,$$

which is equivalent to

$$\sigma \in \mathcal{J} \llbracket t \llbracket \mathbb{P} \rrbracket \rrbracket \vee \sigma \in \{ \eta r \mid \eta \in \mathcal{J} \wedge r \in \mathcal{S} \llbracket t \llbracket \mathbb{P} \rrbracket \rrbracket r' \wedge r' = \neg \eta \} .$$

Hence a sufficient condition to our thesis is provided by the following couple of equalities:

$$\begin{aligned} \mathbf{t}(\mathcal{J}[\mathbb{P}]) &= \mathcal{J}[\mathbf{t}[\mathbb{P}]] \\ \{ \eta \mathbf{t}(s) \mid \eta \in \mathcal{T} \wedge s \in \mathcal{S}[\mathbb{P}] \ s' \wedge s' = \mathbf{t}'(\eta) \} &= \{ \eta r \mid \eta \in \mathcal{T} \wedge r \in \mathcal{S}[\mathbf{t}[\mathbb{P}]] \ r' \wedge r' = \neg \eta \} . \end{aligned}$$

We now turn latter equality into a simpler form. Let

$$R_{\mathbf{t}}(\eta) \stackrel{\text{def}}{=} \{ r \mid r \in \mathcal{S}[\mathbf{t}[\mathbb{P}]] \ r' \wedge r' = \neg \eta \} . \quad (2.80)$$

Then a sufficient condition to the latter equality is

$$\forall \eta \in \mathcal{T}. \{ \mathbf{t}(s) \mid s \in \mathcal{S}[\mathbb{P}] \ s' \wedge s' = \mathbf{t}'(\eta) \} = R_{\mathbf{t}}(\eta) .$$

All in all, we are able to prove $\forall n \in \mathbb{N}. F_{\mathbf{t}}^n[\mathbb{P}] \emptyset = F^n[\mathbf{t}[\mathbb{P}]] \emptyset$ by showing that:

$$\mathbf{t}(\mathcal{J}[\mathbb{P}]) = \mathcal{J}[\mathbf{t}[\mathbb{P}]] \quad (2.81)$$

$$\forall \eta \in \mathcal{T}. R_{\mathbf{t}}(\eta) = \{ \mathbf{t}(s) \mid s \in \mathcal{S}[\mathbb{P}] \ s' \wedge s' = \mathbf{t}'(\eta) \} . \quad (2.82)$$

Assignment Insertion

Let us define, in the framework described above, the transformation of assignment-insertion that we exploit later for embedding watermarks.

Suppose we wish to insert an assignment $\mathbf{w} := \mathbf{A}$ in a program \mathbb{P} , at entrypoint $w \in \text{lab}[\mathbb{P}]$. Then:

- we replace any $\ell: \mathbf{C} \rightarrow w$ in \mathbb{P} with $\ell: \mathbf{C} \rightarrow w_0$, where w_0 is a fresh label with respect to \mathbb{P} ;
- we insert $w_0: \mathbf{w} := \mathbf{A} \rightarrow w$ in \mathbb{P} .

We call the resulting program \mathbb{P}' . The algorithm that turns \mathbb{P} into \mathbb{P}' is \mathbf{i} such that:

$$\mathcal{L}[\mathbf{i}[\mathbb{P}]] \stackrel{\text{def}}{=} \mathcal{L}[\mathbb{P}] \quad (2.83)$$

$$\mathbf{i}[\mathbb{P}] \stackrel{\text{def}}{=} \bigcup \{ \mathbf{i}[\mathbb{S}] \mid \mathbb{S} \in \mathbb{P} \} \quad (2.84)$$

$$\mathbf{i}[w: \mathbf{C} \rightarrow \ell'] \stackrel{\text{def}}{=} \{ w: \mathbf{w} := \mathbf{A} \rightarrow w_0, w_0: \mathbf{C} \rightarrow \ell' \} \quad (2.85)$$

$$\mathbf{i}[\ell: \mathbf{C} \rightarrow \ell'] \stackrel{\text{def}}{=} \{ \ell: \mathbf{C} \rightarrow \ell' \} \quad \text{where } \ell \neq w . \quad (2.86)$$

If there is a dependency between $w: \mathbf{w} := \mathbf{A} \rightarrow w_0$ and a command that comes later, then $\mathcal{S}[\mathbb{P}']$ is likely to differ a lot from $\mathcal{S}[\mathbb{P}]$. To ensure that \mathbb{P}' behaves just like \mathbb{P} , we should:

- save the original value of \mathbf{w} in a temporary variable;
- perform the new assignment;
- restore the original value before \mathbf{w} is used again.

This however requires the insertion of three assignments – one for each of the previous points.

Alternatively, we can choose \mathbf{w} among the variables of \mathbb{X} that are fresh in \mathbb{P} . Indeed, if \mathbf{w} is fresh then no dependencies exist between $w: \mathbf{w} := \mathbf{A} \rightarrow w_0$ and the commands that come later in the flow. Hence the new assignment does not noticeably perturb the semantics of \mathbb{P} . Let us see this in detail. Suppose we have

two commands $\ell: \mathbf{C} \rightarrow \mathbf{w}$ and $\mathbf{w}: \mathbf{C}' \rightarrow \ell'$, and two environments ρ and $\rho' \in \mathbf{C}[\mathbf{C}] \rho$, such that $\text{dom}[\rho] = \text{dom}[\rho'] = \text{var}[\mathbf{P}]$. Then

$$\langle \rho, \ell: \mathbf{C} \rightarrow \mathbf{w} \rangle \langle \rho', \mathbf{w}: \mathbf{C}' \rightarrow \ell' \rangle$$

is a trace. After the insertion of the new assignment, we have

$$\langle \rho, \ell: \mathbf{C} \rightarrow \mathbf{w} \rangle \langle \rho', \mathbf{w}: \mathbf{w} := \mathbf{A} \rightarrow \mathbf{w0} \rangle \langle \rho'[\mathbf{w} := \mathbf{A}[\mathbf{A}]\rho'], \mathbf{w0}: \mathbf{C}' \rightarrow \ell' \rangle .$$

We know that $\mathbf{w} \notin \text{var}[\mathbf{C}']$, because \mathbf{w} is fresh in \mathbf{P} . Thus \mathbf{w} does not interfere with \mathbf{C}' .

The trace transformation we described just now is implemented by a function i on states, which is defined as follows:

$$\begin{aligned} i \langle \rho, \ell: \mathbf{C} \rightarrow \ell' \rangle = \\ \text{if } \ell \neq \mathbf{w} \text{ then} \\ \langle \rho, \ell: \mathbf{C} \rightarrow \ell' \rangle \end{aligned} \quad (2.87)$$

$$\begin{aligned} i \langle \rho, \mathbf{w}: \mathbf{C} \rightarrow \ell' \rangle = \\ \text{if } \ell = \mathbf{w} \wedge \rho(\Delta) = \mathbf{U} \text{ then} \\ \langle \rho, \mathbf{w}: \mathbf{w} := \mathbf{A} \rightarrow \mathbf{w0} \rangle \end{aligned} \quad (2.88)$$

$$\begin{aligned} \text{if } \ell = \mathbf{w} \wedge \rho(\Delta) = 1 \text{ then} \\ \langle \rho[\Delta := \mathbf{U}], \mathbf{w0}: \mathbf{C} \rightarrow \ell' \rangle . \end{aligned} \quad (2.89)$$

A state $\langle \rho, \mathbf{S} \rangle$ is left unchanged (by (2.87)), unless $\text{lab}[\mathbf{S}] = \mathbf{w}$. In such case the value of variable Δ determines how $\langle \rho, \mathbf{S} \rangle$ is to be transformed. In particular, if Δ denotes the undefined value, the insertion has not been performed yet, and it is thereby performed presently through (2.88). Conversely, if Δ is 1, the insertion has already been carried out and the state requires only minor changes, performed by (2.89). Variable Δ , which we call the discriminant, is meant to be used only for the sake of transforming states. Therefore we expect it to be fresh in \mathbf{P} .

Notice that i sets Δ to the undefined value through (2.89), but it never sets it to 1. This is in fact accomplished by the auxiliary function i' for assignment insertion:

$$\begin{aligned} i' \langle \rho', \mathbf{w}: \mathbf{w} := \mathbf{A} \rightarrow \mathbf{w0} \rangle = \\ \text{if true then} \\ \langle \rho'[\Delta := 1], \mathbf{z}: \mathbf{w} := \mathbf{A} \rightarrow \mathbf{w} \rangle \end{aligned} \quad (2.90)$$

$$\begin{aligned} i' \langle \rho', \ell: \mathbf{C} \rightarrow \ell' \rangle = \quad \text{where } \ell \neq \mathbf{w} \\ \text{if true then} \\ \langle \rho', \ell: \mathbf{C} \rightarrow \ell' \rangle . \end{aligned} \quad (2.91)$$

Only in case the input state includes $\mathbf{w} := \mathbf{A}$, (2.90) sets Δ to 1. Such value is naturally propagated to any state s in the semantics of (2.90) and is taken into account when s is transformed by i . As the novel value of \mathbf{w} is expected to propagate to subsequent states as well, command $\mathbf{w} := \mathbf{A}$ was included in (2.90).

In order to show that i is the semantic counterpart of \mathbf{i} , we prove the following

Theorem 2.8. *Let P be a program. Then $\forall n \in \mathbb{N}. F_i^n[[P]]\emptyset = F^n[[i[[P]]]\emptyset$.*

Proof. We let $\mathfrak{t} = i$, $\mathfrak{t} = i$ and $\mathfrak{t}' = i'$, and just prove (2.81) and (2.82).

First we prove (2.81) by showing that $\langle \rho, S \rangle \in \mathcal{J}[[i[[P]]]$ if and only if $\langle \rho, S \rangle \in i(\mathcal{J}[[P]])$.

We have $\langle \rho, S \rangle \in \mathcal{J}[[i[[P]]]$ if and only if there exists $\ell: C \rightarrow \ell' \in P$ such that $S \in i[[\ell: C \rightarrow \ell']]$ and $\text{lab}[S] \in \mathcal{L}[[i[[P]]]$.

- Suppose $\ell = w$.

By (2.85), $S \in i[[w: C \rightarrow \ell']]$ if and only if $S \in \{ w: w := A \rightarrow w0, w0: C \rightarrow \ell' \}$, where $w0 \notin \mathcal{L}[[P]]$ is not fresh with respect to P . Notice moreover that by (2.83) $w0 \notin \mathcal{L}[[i[[P]]] = \mathcal{L}[[P]]$.

This is equivalent to $S = w: w := A \rightarrow w0$, which in turn is equivalent to $\langle \rho, S \rangle = \langle \rho', w: w := A \rightarrow w0 \rangle$, with $\rho' = \rho$, and ultimately to $\langle \rho, S \rangle \in i \langle \rho', w: C \rightarrow \ell' \rangle$.

- Suppose $\ell \neq w$.

By (2.86), $S \in i[[\ell: C \rightarrow \ell']]$ if and only if $S \in \{ \ell: C \rightarrow \ell' \}$ if and only if $S = \ell: C \rightarrow \ell'$ if and only if $\langle \rho, S \rangle = \langle \rho', \ell: C \rightarrow \ell' \rangle$ with $\rho' = \rho$ if and only if $\langle \rho, S \rangle \in i \langle \rho', \ell: C \rightarrow \ell' \rangle$.

Thus in both cases we have $\langle \rho, S \rangle \in i \langle \rho', \ell: C \rightarrow \ell' \rangle$, $\rho = \rho'$ and $\ell \in \mathcal{L}[[i[[P]]]$; by (2.83), the latter assertion can be restated as $\ell \in \mathcal{L}[[P]]$.

This is equivalent to state that there exists $\langle \rho', \ell: C \rightarrow \ell' \rangle \in \mathcal{J}[[P]]$ such that $\langle \rho, S \rangle \in i \langle \rho', \ell: C \rightarrow \ell' \rangle$ or, ultimately, that $\langle \rho, S \rangle \in i(\mathcal{J}[[P]])$.

We now prove (2.82).

- Let η be such that $\neg \eta = \langle \rho', w: w := A \rightarrow w0 \rangle$.

On the one hand,

let $\rho'' = \rho' [w := A[[A]] \rho']$.

Then $R_i(\eta) = \{ \langle \rho'', w0: C \rightarrow \ell' \rangle \mid w0: C \rightarrow \ell' \in P \}$

On the other hand,

we have $i'(\neg \eta) = (2.90)$. Let $\rho = \rho' \left[\begin{array}{l} w := A[[A]] \rho' \\ \Delta := 1 \end{array} \right]$.

Then

$i(s) = i \langle \rho, w: C \rightarrow \ell' \rangle$ where $w: C \rightarrow \ell' \in P$

$= \langle \rho [\Delta := \mathcal{U}], w0: C \rightarrow \ell' \rangle$

$= \left\langle \rho' \left[\begin{array}{l} w := A[[A]] \rho' \\ \Delta := 1 \end{array} \right] [\Delta := \mathcal{U}], w0: C \rightarrow \ell' \right\rangle$

$= \langle \rho' [w := A[[A]] \rho'], w0: C \rightarrow \ell' \rangle$

$= \langle \rho'', w0: C \rightarrow \ell' \rangle$.

- Let η be such that $\neg \eta = \langle \rho', \ell: C \rightarrow \ell' \rangle$ where $\ell \neq w$.

On the one hand,

let $\rho'' \in C[[C]] \rho'$.

If $\ell' = w$ then $R_i(\eta) = \{ \langle \rho'', w: w := A \rightarrow w0 \rangle \}$.

If $\ell' \neq w$ then $R_i(\eta) = \{ \langle \rho'', \ell': C \rightarrow \ell'' \rangle \mid \exists C'. \exists \ell''. \ell': C' \rightarrow \ell'' \in P \}$.

On the other hand,

we have $i'(\neg \eta) = (2.91)$. Let $\rho \in C[[C]] \rho'$.

If $\ell' = w$ then

$$\begin{aligned}
i(s) &= i \langle \rho, w: C \rightarrow \ell'' \rangle \text{ where } w: C \rightarrow \ell'' \in P \\
&= (2.88) \\
&= \langle \rho, w: w := A \rightarrow w0 \rangle \\
&= \langle \rho'', w: w := A \rightarrow w0 \rangle .
\end{aligned}$$

If $\ell' \neq w$ then

$$\begin{aligned}
i(s) &= i \langle \rho, \ell': C' \rightarrow \ell'' \rangle \text{ where } \ell': C' \rightarrow \ell'' \in P \\
&= (2.87) \\
&= \langle \rho, \ell': C' \rightarrow \ell'' \rangle \\
&= \langle \rho'', \ell': C' \rightarrow \ell'' \rangle .
\end{aligned}$$

□

Software Steganography

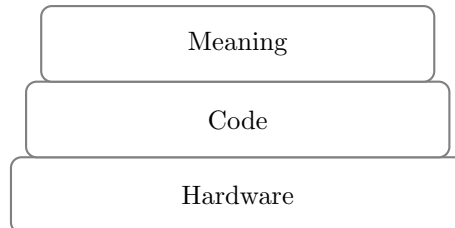
The *security through obscurity* principle we presented in Chapter 1 has at least three applications: isolationism, cryptography and steganography. As the last one seems to suit software best, we discuss at length two instances of its: code obfuscation and software watermarking. In the last Section, we deal with obfuscation and watermarking based on abstract interpretation, a methodology which might be key in providing a unifying framework for the steganography of software.

3.1 A Grab of Software

Due to its highly informational nature, software is a resource that one would like, according to their needs and aims, to protect from or subject to a thorough inspection. Apart from its many possible instantiations, software can be defined as something turning a multi-purpose machine into a machine with a more specific purpose [62]. In particular, software enjoys the following three properties:

1. it carries a *meaning*, e.g. the details of a machine functionality, mathematically modeled by formal semantics;
2. it consists in *code*, written according to the syntax of a programming language;
3. it is transmitted through physical supports, e.g. magnetic and optical devices, generically referred to as *hardware*.

Being opposed to hardware, software only concerns the code and the meaning it denotes.



In the three-layered architecture provided by software properties, code plays a twofold role: it acts as a content wrt. hardware and, at the same time, as a carrier wrt. the meaning. Hence, to put into practice software protection, first we must

choose whether we intend to deal with code as a content or as a carrier: in the former case we apply cryptography, in the latter steganography. An example of software concealment through cryptography is offered by encrypted, oligomorphic and polymorphic viruses [100], whose code starts with a constant decryptor followed by the encrypted virus body [107]. Here encryption was introduced in an attempt to protect the code of these kinds of malicious software from syntactic recognition by anti-virus scanners. The reasons why this protection scheme was easily broken reside not in encryption itself, but in the presence of quite the same decryptor in the header section of the different encrypted variants of the virus [107]. As regards software concealment through steganography, a general and well-founded theory about the provable secure concealment of the information conveyed by software still lacks.

3.2 Protection of Software

The widespread physical distribution of computing power and the dramatic increase in the networking of computers make information likely to disseminate without any control. It is worth noting that a provably safe policy to preserve information from being inspected is isolationism: if there is no sharing, there can be no undesired observers. Current military protection systems, for instance, traditionally depend to large degree on isolationism and it is still a challenge for them to integrate with networking systems [91]. However, isolationism is unacceptable in scientific and development environments, and in general whenever we wish to benefit from the work of others. This especially applies in the field of software: except for personal entertainment, there is usually no point in writing, for example, a device driver (a software that interact with a commercial computer hardware device) or a worm (a malicious software consuming the bandwidth of the network) but not releasing them in the wild.

Whenever isolationism is unavailable, one of the most promising ways to achieve a provably secure software concealment is the exploitation of computational advantage provided by security through obscurity [15]. Computational advantage shows up historically in cryptography, where performing the decryption without knowing the suitable key is computationally unfeasible [102]. However, as a tool for the protection of software, cryptography is of little use: on the one hand, encrypted software falls short because it can be neither compiled nor interpreted; on the other hand, the assumption that decryption keys will remain unknown to unauthorized users is broken down by the about 47 millions of entries yielded by querying Google about **keygens**, that is, illegal application-specific key generator tools. With steganography these difficulties do not occur: one can keep the key secret and still distribute the concealed version of its software, because steganography is not going to compromise software executability.

From the steganographer perspective, code is the carrier of a meaning. Even though the inherent meaning of code is its semantics, that is, the specialized machine functionality it describes, actually code may denote a different meaning for every different observer. For an anti-virus scanner, the meaning of code is whether it includes or not any *virus signature*, that is, a fixed sequence of bytes from a

sample of a virus [71]. For an analyzer looking for similarities between two software applications, the meaning of code is its birthmark, that is, a sequence of unique characteristics used to identify the code [108]: if two applications have the same birthmark, they are likely to derive from the same source. For a verification tool aimed at checking safety-critical software for embedded systems, the meaning of code may be whether it terminates or not when executed. Furthermore, imaginative observers may devise unconventional ways of interpreting code, so that e.g. code structure and semantics may become themselves covert channels for information exchange. To evade meaning disclosure without interfering with software purpose, the steganographer alters the code retaining its expected functionality, but at the same preventing undesired observers from getting the meaning they are interested in. In such a case software is concealed via *code obfuscation*.

Sometimes one may also wish to expand the inherent meaning of code with additional information. Again, this can be achieved by modifying the code so that it encodes the new information but it keeps its own expected functionality. The provided information takes part into the code as a *mark*. The practice of marking dates back early in the human history, encompassing for instance brand marks burnt in the skin of the cattle and watermarks embossed in banknotes, the former ones being visible and robust, that is, difficult to be removed, whereas the latter ones being (semi)invisible and fragile, that is, intended to not survive any kind of copying [76]. Aiming at hiding information within software, the steganographer resorts to marks which are (semi)invisible, that is, visible only to qualified observers, and usually (but not necessarily) robust. Hence, the field of techniques and algorithms for embedding and detecting additional information in software is known as *software watermarking* [21] and the marks carrying the additional information are named *watermarks*.

3.3 Code Obfuscation

To the best of our knowledge, the basic ideas about code obfuscation were first introduced in 1993 by Frederick B. Cohen [15], who suggested to provide operating systems protection through *program evolution*. In Cohen's view, a program (that is, the code of a software application) evolves if it undergoes a code transformation which preserves its input-output behavior. Typical examples of evolutionary transformations are instruction reordering, variable substitution and garbage insertion. With more evolution, the program should denote higher cost of analysis, even if it may have a worse performance. Cohen notices that although an exhaustive set of equivalent programs is easily described mathematically, the equivalence of two programs is undecidable [14]. According to him, this result seems to indicate that evolution has the potential for increasing complexity of analysis [15]. However, since evolution is as general as Turing machine computation [115], the determination of whether one program can evolve from another is undecidable, too [14]. Cohen notes that, as a consequence, this is not particularly helpful in terms of designing practical evolutionary schemes [15]. Furthermore, he states that one cannot blindly trust protection through evolution, because any protection scheme other than physical one depends, in the case of real-life computers, on the operations of a

finite state machine and, ultimately, any finite state machine can be examined and modified at will, given enough time and effort [15]. Nevertheless, evolutionary transformations can be useful, since they can help in delaying program analyses.

About five years later, Christian Collberg and Clark Thomborson recovered the main concepts of program evolution to describe a methodology for thwarting reverse engineering of software applications [22, 23], calling it *code obfuscation*. Reverse engineering is a typical first step for an observer to take advantage of the software: having gained physical access to the application, the reverse engineer can decompile it (using disassemblers or decompilers [13]) and then analyze its data structure and control flow, possibly with the aid of reverse engineering tools such as program slicers [111]. This kind of inspection is easier in programming languages like Java, whose hardware-independent virtual machine bytecode retains virtually all the information of the original Java source, and whose programs often make use of well-known Java standard libraries. Thus, concerned with the relative ease of extracting proprietary algorithms and data structures from applications, Collberg and Thomborson propose code obfuscation as a tool for protecting intellectual property.

Meanwhile, because of a lack of stealth in pre-existing viruses, the concepts of program evolution proved to appetize also malware writers, who refer to program evolution as *metamorphism* [107]. Metamorphic viruses are self-propagating malicious programs consisting of self-mutating code [66]. The idea is that each successive generation of a metamorphic virus modifies the code while leaving the malicious semantics unchanged. This especially applies on CISC architectures, such as the Intel IA-32 [26], whose instruction set is rich and where a lot of instructions denote overlapping semantics. Here, replacement of instructions with semantically equivalent ones, together with insertion of junk code, proves to be an easy device to escape common anti-virus scanners, which usually detect viruses by identifying in the subject code virus *signatures*, that is, virus-specific patterns such as known sequences or statistic distribution of virus instructions [25]. To detect a metamorphic virus, common scanners should keep a database of all the signatures it may assume. This is not an easy task since, in principle, a metamorphic virus can perform an unlimited number of mutations.

Notions of Code Obfuscation

As code obfuscation (in the rest of this thesis the phrase “code obfuscation” subsumes both program evolution and metamorphism) appears to be a multi-purpose tool which still keeps to draw the attention of many people, it becomes opportune to understand its true potentialities and limits. The first attempt to theoretically establish code obfuscation is due to Collberg and Thomborson [23], who provide an engineering approach to the issue. They consider any program transformation preserving software behavior as experienced by the user, and they measure its potential effectiveness as a program obfuscator in terms of potency, resilience, cost and stealth. A program transformation has a non-trivial *potency* if the transformed (obfuscated) program is more difficult to understand than the original one; the comprehension difficulty is modeled through ordinary software engineering metrics, such as the number of instructions and arguments [57], the number of nested

conditions [59] or the number of references to local variables [89]. The *resilience* of a transformation measures the hardness of undoing it with an automatic deobfuscator, taking into account both the amount of time required by the obfuscator to be built and the execution time and space it consumes; one-way transformations, such as program comments removal, are highly resilient because they can never be undone. The *cost* of an obfuscating transformation measures the computational overhead added to the subject program by obfuscation; there is usually a trade-off between potency and resilience on the one hand and cost on the other hand, because non-trivial obfuscation intuitively entails non-trivial costs. Finally, the *stealth* of an obfuscating transformation ensures that the statistical properties of the obfuscated program do not significantly differ from those of the subject program; hence, stealth is a context-sensitive notion, since what is stealthy in one program may not be stealthy in another one.

Collberg and Thomborson also classify obfuscating transformation according to the kind of information they target [23]. *Layout obfuscators*, such as Java-obfuscator Crema [117], act on code information that is useless at execution time, reducing the amount of information available to a human reader; they are typically trivial and they include for instance the removal of program comments and the scramble of identifiers. *Data obfuscators* act on program data structures, affecting the procedures for storing, reading and interpreting data, and the logic aggregation and ordering of information within data structures [23], [127], for instance by turning a two-dimensional array into a one-dimensional array and vice versa. *Control code obfuscators* operate on the control flow of programs and possibly perform: splitting and merging of code fragments in new program procedures [43]; altering iteration order and scope in program loops [2]; randomizing, when possible, the placement of any item in the subject program; inserting irrelevant instructions (referred above as garbage or junk code) or bogus dead code, that is, misleading instructions which are never executed.

Opaque Predicates

In particular, bogus dead code insertion often relies upon the existence of *opaque predicates* [24], that is, predicates whose value is known at obfuscation time, but it is difficult for a deobfuscator to deduce: the idea is to use, for example, an always-true opaque predicate as the boolean condition of a selection construct and attach the bogus dead code to the false branch. This is not apparent to the deobfuscator which has to take into account both branches while performing a static program analysis, and cannot be sure that the false branch is never executed even though dynamic program analysis suggests this conjecture. Stealthy and resilient opaque predicates, together with stealthy bogus dead code, are the major building blocks in the design of transformations that obfuscate the control flow [23], [24], [74], [90].

The design of resilient opaque predicates is an instance of the exploitation of the security through obscurity principle in code obfuscation. The first opaque predicates [23], [24] were derived from the number theory, such as

$$(x(x + 1))^2 \bmod 4 = 0$$

or

$$x^2 = (7y^2 - 1) ,$$

which are respectively always true and always false whatever x and y are evaluated to. The drawback of this kind of opaque predicates is that their obscurity do not reside in their design: a deobfuscator might merely collect them in a database and identify them easily.

Since a deobfuscator usually employs various static and dynamic analysis techniques, it seems natural and more effective to construct opaque predicates on problems that are hard to handle by such analyses. Unlike those derived from the number theory, these opaque predicates are trustful for obfuscation purposes, because they are provably resilient wrt. well defined program analyses. Alias analysis of complex dynamic structures is known to be a difficult problem [64], [101]; thus, predicates describing invariants preserved by the set of pointers of the structure are both opaque and provably resilient. Among the other problems whose invariants are difficult to determine at deobfuscation time, there are parallel program analysis [24] and distributed systems analysis; in the latter, the value of the opaque predicate depends on predetermined embedded message communication patterns between different processes that maintain the predicate [74]. The notion of opacity entailing provable resilience has been generalized to program properties and also extended to security issues like non-interference and anonymity [8], [9], [70].

Positive Results

Upon computational advantage, which is distinctive of the security through obscurity approach, not only opaque predicates, but also others provably potent obfuscating schemes have been designed. Wang *et al.*, for instance, transform the original control flow of the subject program into a flattened one, where each basic block of instructions can be the successor or predecessor of any other basic block, and where the actual control flow is determined dynamically by a dispatcher controlled by a variable, whose value is established at the end of each basic block through pointer manipulations complicated by the introduction of aliasing; the authors prove that determining precise indirect branch target addresses of dispatchers in presence of aliased pointers is a NP-hard problem [120]; this technique, originally restricted to intra-procedural analyses, has been extended to inter-procedural analyses, too [86]. Another interesting proposal is the semantics preserving insertion of hard combinatorial problems inside programs, explored by Chow *et al.*, which makes the deobfuscation process PSPACE-complete [11].

However, these results, although successful, should in practice be evaluated with caution. While NP-complete and PSPACE-complete problems are generally considered intractable [45], this intractability is based on the difficulty of the hardest problem instances. However, some NP-complete problems are easy in the average case. Thus for use in security, of greater interest than worst-case is average-case complexity analysis [121] and the ability to prove that the probability of easy instances arising is very low. It is worth mentioning that there are many cryptographic examples where difficult problems have been used to justify the security of proposals, which were later proven to be easily broken due to the fact that the particular problem instances built into actual instantiations turned out, for various reasons, to be far

weaker than the hardest, or average, problem instances [116]. This suggests care of use, but does not disqualify the results.

Negative Results

By contrast, there are the impossibility claims of Barak *et al.*, who prove that the following device does not exist: a software virtual black box obfuscator which can protect *every* program code from revealing more than program input-output behavior [3]. More recently, an even stronger theoretical result has been proven [49]. While on the surface these assertions are rather negative for practitioners interested in software obfuscation, upon deeper examination (and despite the rather suggestive title of the papers¹), the results simply arise from the choice of definitions, models and question posed.

Actually, the non-existence of such a virtual black box generator would appear to be of little concern after having discussed to what proportion of programs of practical interest these results apply, whether there exist obfuscators able to obfuscate program of practical interest, whether a more practical model can be defined, allowing some level of non-critical information to leak from the execution of a program, provided that it is not useful to the external observer [116]. By relaxing the constraints of Barak’s model, it is reasonable and of practical interest to study the possibility of obfuscating programs. Some of the authors of the impossibility results have later achieved some positive results on code obfuscation [73] that, together with other works [10], [122], show, under certain assumptions, how to obfuscate classes of real-life programs.

Besides, many researchers are interested in transformations that raise the difficulty of inspecting, e.g. by reverse engineering, a program, even if they cannot make it impossible as requested by Barak’s definition. In fact, an obfuscating transformation that requires a very expensive analysis, in terms of resources and time, to be undone, protects a software by making its inspection uneconomical [63]. This approach is very appropriate where software protection is required for a limited time period, e.g. in case of forced aging and renewal of software [65], which is performed in order to make software cycle at a rate faster than an observer can break it [52]. In Appendix A we provide a list of software products for obfuscating code.

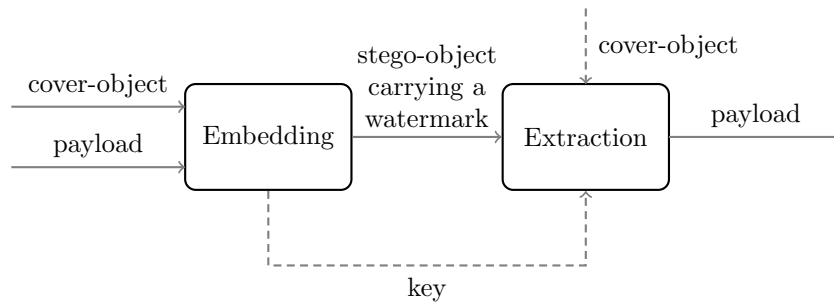
However, inspection infeasibility can be critical in malware detection. A detection scheme going beyond purely syntactic nature of virus signatures exploits templates [12], trying to express malicious intent while abstracting from syntactic details which differ in the obfuscated variants of a metamorphic virus; this approach, using symbolic variables to handle variable renaming and generalized control flow graphs to deal with instruction reordering, is able to withstand a limited set of obfuscations commonly used by malware writers. In the field of static analysis there exist a malware detection scheme based on suspicious system call sequences [4] and another one where the subject program and the malware are modeled through automata and detection consists in finding a non-empty intersection between the languages they denote. Model checking techniques have recently been used

¹ “On the (Im)possibility of Obfuscating Programs” [3] and “On the Impossibility of Obfuscation with Auxiliary Input” [49].

to specify malicious behavior through a linear temporal logic [105], or other temporal logic [69]. Program slicing [123] and flow analysis are exploited in a static analysis tool [72] which detects malware relying upon peculiar program properties, named tell-tale signs, that characterize the maliciousness of a program. However, all these approaches are still of limited application and it is not clear the interaction occurring among them.

3.4 Software Watermarking

The basic definitions of software watermarking concepts appeared in the early papers by Collberg and Thomborson [20], [21], who mainly focus on expected properties of watermarks in view of their integrity and reliability. The items involved in software watermarking are the following.

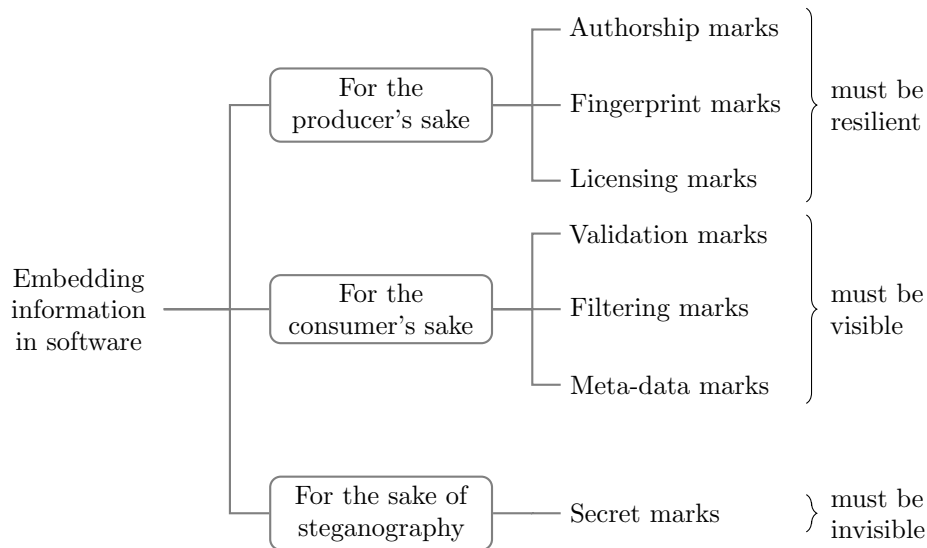


- *Watermark*, or *embedded-object*, or *package*, or *stegomark*² [34]: something to hide in something else [92].
- *Payload*: the information conveyed by the embedded object.
- *Cover-object* [92] or *program* or *carrier*: the innocuous object where you embed something [92].
- *Stego-key* [92] or *key*: additional secret data that may be needed in the hiding process; in particular, the same key (or a related one) is usually needed to extract the embedded object again [92].
- *Stego-object* [34, 92] or *marked object* [92]: the output of the hiding process; something that has the embedded object hidden in it [92].

Once embedded in a program, a watermark should be such that:

- it denotes a high data-rate, where the *data-rate* is intended as the number of message text bits which are encoded per bit added to the subject program [18], or as the ratio of size of the watermark that can be embedded to the size of the subject program [17];
- it is *cheap* (also, it denotes *high fidelity* [1]), that is, it does not adversely affect the performance of the program;

² “Stego” (Greek $\sigma\tau\epsilon\gamma\omega$, literally *roof*, *cover*) means “to keep by covering”.



- it is *stealthy*, meaning that it preserves any statistical property of the program.

Watermarks can be either resilient or fragile. A *resilient* or *robust* [92] watermark is able to survive distortions [84] based on automatic program analyses. A *fragile watermark* does not survive any kind of copying [84] or is destroyed as soon as the object is modified too much [92].

In principle, watermarks can be either visible or invisible. A watermark is said to be *visible* or *perceptible* if it can be extracted without knowing secrets. It can be used as a deterrent against misuse or to embed carry meta-information [19] or, in case of fingerprints or filtering marks, to make the limited rights of the purchaser immediately apparent [84]. On the other side, a watermark is *invisible*, or *imperceptible* or *transparent* [92], if it cannot be grasped by human or machine analysis [67]. Sometimes this means that the watermark is hard to discover without a suitable exposition tool [84] or even it can only be extracted using a secret not available to the unqualified end user [19]. In our steganographic approach to the protection of software, watermarks can be only invisible. In the rest of this thesis we are not concerned with visible watermarks.

Watermarks can be classified according to their payload [84]:

- *Authorship marks* [84], which are the *watermarks* [92] by antonomasia and are sometimes called *signatures* [34], convey information identifying the author [84] or the tool [67] that developed the software.
- *Fingerprint marks* [84] or *labels* [92] convey information identifying the serial number or the purchaser of the software [84].
- *Licensing marks* [19, 84] convey information controlling how the cover object can be used [84].
- *Validation marks* [19, 84] convey information used by the end user to verify that the marked object is still essentially the same as when it was authored [84].
- *Filtering marks* [19] or *classification marks* [19] convey the parental rating about the content of the cover object [19].

- *Meta-data marks* [19] convey information that the end user might find useful [19].
- *Secret marks* [19] convey information used for unnoticed communication between two parties (steganography) [19].

Authorship marks are required to be reliable, that is, they must denote a distinctive feature that allows everybody to argue that their presence in a program is the result of deliberate actions [21]. Two strategies are known to achieve this:

- we can embed in our program a natural number that is too great to be feasibly factorized and claim it as our authorship mark by producing its factorization [34];
- we can encode a personal string, e.g. our name, through a standard one-way hash function like MD5 and embed the result in our program, then claim it as our authorship mark by producing our personal string and comparing its encoded version with the authorship mark [87, 113].

Stegoanalysis

The payload of a watermark is valuable information which one may want to either expose or destroy, according to one's purposes. We distinguish three classes of users:

- *final* [92] or *end users* [19] check (if ever) that the stego-object is/behaves as they expect;
- *passive observers* try to get the watermark without disrupting the stego-object [94];
- *active offenders* try to remove the watermark, either retaining (a similarity to) the (supposed) cover-object (as in the case of an illegal redistributor or a silent censor) or not [94].

Passive observers typically perform *collusive attacks*, trying to localize fingerprints by comparing copies of the same program marked with different fingerprints. This sort of attacks is usually exploited as a preliminary step to distortive or subtractive attacks. As a countermeasure, distinct obfuscations for distinct copies could make the copies so different that their comparison become useless [34].

Active offenders are likely to perform *distortive attacks*, which aim at modifying the watermarked program in order to prevent the correct extraction of watermarks while preserving program semantics and, possibly, without degrading program performance. These attacks often take the form of obfuscating transformations [22]. Offenders also can perform *subtractive attacks*, which use public knowledge about watermarking techniques to localize and delete watermarks. For example, if a technique is known to conceal watermarks in the dead code of program, then dead code elimination is an elegant and effective attack to sweep them away.

A special category is the one of *additive attacks*, which affect a program with watermark W by embedding a foreign watermark $W' \neq W$. Unfortunately, the program cannot furnish evidence that W was inserted before W' even if W' does not override W . This is a serious problem whenever W and W' are authorship marks or fingerprints because, as far as we are concerned, there do not exist any

software watermarking methods which have been proposed so far that can thwart this sort of attacks [16, 21]. So, if the watermarking technique is to be made public, as required by a sound interpretation of the security through obscurity principle, one can only rely to trustworthy authorities at embedding time to certify the temporal precedence of their watermark.

It is generally agreed that any existing technique for software watermarking and code obfuscation is not able to withstand *manual attacks* supported by enough manpower, time and human motivation [23, 34]. In other words, a sufficiently determined attacker will eventually be able to defeat any watermark. The goal, then, is to design watermarking techniques that are expensive enough to break, that for most attackers breaking them is not worth the trouble [16]. Both evaluation metrics for and attacks against software watermarking techniques have been addressed in a more formal framework [18].

Embedding and Extraction Routines

The act of watermarking a program is expressed in the literature through numerous verbs: to embed [67, 92], to inlay [34], to store, to encode [42, 67], to implant [113], to synthesize [113], to generate [42, 113] and to hide [99]. There are two kinds of embedding:

- in *active embedding* the embedded object is integrated *as part of the design process* [113], by augmenting fully specified designs or by patching incomplete designs, in either case without disrupting the original specifications [112, 113];
- in *passive embedding* the embedded object is added to a design *by making use of existing structures*, thus requiring no redesign but allowing limited tracking flexibility [112].

It has been shown that some ill-conceived watermarking techniques can embed only a very limited set of payloads [125].

The crucial point of watermarking, however, is the ability to soundly recover invisible watermarks while both minimizing the number of false positives and negatives. Especially false negatives may be a reason of concern:

“The real problem is not so much inserting the marks as recognising them afterwards. Thus progress may come not just from devising new marking schemes, but in developing ways to recognise marks that have been embedded [...] and thereafter subjected to distortion” [93].

Several verbs are used to designate this second routine: to extract [34, 125], to expose [18], to recognize [126], to audit [42], to detect [113], to recover [99]. Nonetheless, Zhu and Thomborson [125, 126] make a clear distinction between extraction and recognition.

An *extraction* routine takes in input the watermarked program and returns the payload carried by the watermark. In general, this routines induces a partition on the set of the payloads and is able to extract only the representatives of each class. This means that if watermark W belongs to a class whose representative is $[W]$, the extraction algorithm yields $[W]$ even though W is embedded in the subject program [125]; hence, the reliable extractor is the one that induces the partition with only singleton-classes.

Positive and negative *recognition* or *detection* algorithms judge, respectively, the presence and the absence of watermarks in the subject program [126]. Their precision is related to the number of, respectively, false positives and false negatives they denote. Recognizers may play a valuable role in the problem of retrieving watermarks from distorted programs [126]. It is worth noting that distortion, obtained by e.g. code obfuscation, may also be used to protect watermarks from undesired recognition [3].

Both kinds of retrieval can be either *blind* and *informed*, the latter being performed with the knowledge of the original cover-object.

		key	cover object	embedded object	extraction	recognition	
visible marks	asymmetric marking				✓	✓	blind retrieval
	public marking	✓			✓	✓	
invisible marks	semi-private marking	✓	✓			✓	informed retrieval
	private marking I	✓	✓		✓	✓	
	private marking II	✓	✓	✓		✓	

The different characteristics of the extraction routine combine variously in the following schemes.

- *Private marking I* (also, either *informed extraction* [125] or *informed recognition* [126]) performs extraction/recognition using the cover-object as a hint to find where the embedded watermark could be [92].
- *Private marking II* performs *recognition* using both the cover-object and the embedded object [92].
- *Semi-private marking* performs *recognition* using just the embedded object [92].
- *Public marking* (also *blind marking* [92], *blind extraction* [125], *blind recognition* [126]) requires neither the cover-object nor the embedded object at extraction/recognition time [92]. It may require the key.
- *Asymmetric marking* performs extraction/recognition without using keys; it expects the watermark to be resilient enough an offender is not able to removed it [92].

Lastly, watermarking techniques can be divided into two categories, regarding if at extraction time the watermarked program is executed or not.

In *static watermarking*, watermarks are stored in the program source either as data, e.g. an image or a string, or code, e.g. in the code control structure. So, the watermark can be extracted from the text of the program or the program syntax without any need of execution. Unfortunately, all static watermarks are susceptible to simple distortive or subtractive attacks. For example, Moskowitz [79] describes a static data watermarking method in which the watermark is embedded in an image using one of the many media watermarking algorithm; this image is

then stored in the static data section of the program; however, the presence of an image at the top of a program is highly unusual. Davidson [42] describes a static code watermark in which a fingerprint is encoded in the basic block sequence of a program's control flow graph; this scheme is easily subverted by permuting the order of the blocks [16]. Qu and Potkonjak embed the watermark in register interference graphs [98]; here not only the data-rate is minimal, but also the watermark can be distorted simply by register renumbering transformations [81]. Venkatesan *et al.* embed the watermark in an extra control flow graph with marked basic blocks that is added to the program [118]; here the data-rate is high, but the scheme is vulnerable to transformations such as block splitting and instruction reordering [17]. Stern *et al.* embed the watermark in the relative frequencies of instructions using a spread spectrum technique [106]: the data-rate is low and the scheme is easily subverted by inserting redundant instructions and applying code optimization [16].

In *dynamic watermarking*, watermarks are stored in program execution states and so program execution is required in order to extract the watermark. Dynamic watermarking was first proposed by Collberg and Thomborson [21]; their scheme embeds the watermark in the topology of a data structure that is built on the heap at runtime given some secret input sequence to the program; unfortunately, this particular scheme is vulnerable to any attack that is able to modify the pointer topology of the program's fundamental data types [16]. The proposed scheme is an instance of the class of *dynamic data structure watermarking* techniques, which store watermarks in the program data if and when executed with particular inputs; generally, the watermark is exhibited by a watermark extraction routine examining these marked program data [22]. Instead, in *dynamic execution trace watermarking*, watermarks are stored within the trace of the program as it is being run with a special input; the watermark is extracted by monitoring some (possibly statistical) properties of the trace. An instance of this class of schemes is the path-based watermarking technique relying upon the dynamic branching behavior of programs [16], which proves to be resilient to a wide variety of attacks. There is also a third, and usually trivial, class of dynamic watermarking techniques, embedding *Easter Egg watermarks*, whose defining characteristic is that, when a predefined input is entered in the program, the program performs some action that is immediately perceptible by the user, such as displaying a copyright message or image [22]. One difficulty with Easter Eggs, and dynamic watermarking in general, is that the special input revealing the watermark can be localized by monitoring program execution, using standard instrumentation techniques, and removed by debugging techniques, in which case it must be considered ineffective [34].

A recent survey of software watermarking is in [128]. Watermarking techniques also abound in [55], [77], [78], [81], [82], [83], [90], [95], [106].

3.5 Software Steganography by Abstract Interpretation

Software is a very malleable engineering product, especially if expressed in a high-level or in a semi-compiled programming language, such as the Java bytecode [53]. On the one hand, it easily takes the shape of the content it carries, so it is hard to dissimulate *specific* information within it: this is what makes difficult, for

instance, to embed perfectly stealthy watermarks, to perfectly prevent information leakage [103] or to perfectly obstruct reverse engineering for code reuse. On the other hand, software quickly lends itself to any kind of transformations, so it is easy to deform it in order to thwart *accurate* information retrieval: this makes problematic, for instance, to design effectively resilient watermarks, to safely detect metamorphic malware and to dam up malware infection.

The main point is the lack of a solid foundation of accuracy in software analysis and observation. Abstract interpretation may provide a suitable framework for reasoning about the information that software applications can carry and disclose.

The inability of an observer $\alpha_{\mathcal{O}}$ (see Section 2.16) to notice that a program transformation \mathfrak{t} changes some program property δ is the basis of the more generalized notion of obfuscation potency \mathfrak{t} stated in 2005 by Dalla Preda and Giacobazzi [37]. Given set Δ of all the program properties derived as abstractions of concrete domain \mathcal{D} , each potent transformation \mathfrak{t} partitions Δ into the class of *preserved* properties and the class of the *masked*, that is, not preserved, properties. The obfuscation potency stems from the non-emptiness of the latter class. On the other side, the most concrete preserved property $\delta_{\mathfrak{t}}$, whose existence proof and constructive characterization have been provided [37], fully describes the content that \mathfrak{t} keeps intact. Given an observer $\alpha_{\mathcal{O}} \in \Delta$, that is, an observer obtained as an abstraction of \mathcal{D} where \mathcal{D} is of course the most clever observer, if $\alpha_{\mathcal{O}}$ belongs to the class of the properties preserved by \mathfrak{t} and \mathfrak{t} is potent, then \mathfrak{t} is considered to act as an obfuscator wrt. $\alpha_{\mathcal{O}}$. If $\alpha_{\mathcal{O}}$ is not preserved, then it is sensible to the changes performed by \mathfrak{t} , so \mathfrak{t} cannot evade its inspection ability. The degree of abstraction of $\delta_{\mathfrak{t}} \in \Delta$ is related to the aptitude of \mathfrak{t} for obfuscation wrt. all the observers in Δ ; hence, if $\delta_{\mathfrak{t}_1} \sqsupseteq \delta_{\mathfrak{t}_2}$, then \mathfrak{t}_1 has a better aptitude than \mathfrak{t}_2 [37].

Another abstract interpretation based comparison between observers was introduced later [36] in the context of opaque predicate insertion for control code obfuscation. Here a simple algorithm \mathfrak{t}_{op} for putting in the subject code a given set of opaque predicates is attained by abstraction of the semantic specification of the transformation via the Cousot's framework [33]. Due to the scanty aptitude of \mathfrak{t}_{op} for obfuscation in the sense specified above [36], the authors focus on the resilience of the inserted opaque predicates wrt. to agnostic observers which have *not* been derived as abstractions from \mathcal{D} ; in fact, an observer $\alpha_{\mathcal{O}}$ derived from \mathcal{D} is expected to be quite clever, since \mathcal{D} retains full knowledge about the real behavior of opaque predicates. Agnostic observers can improve their inspection ability, providing that they are made complete wrt. the opaque predicate they wish to observe [36]. *Completeness* is a significant concept in the abstract interpretation theory, because it captures the precision of an abstraction wrt. a property of interest. By taking advantage of a systematic way for minimally transforming abstractions in order to make them complete [47, 48], the authors measure the magnitude of the inspection inability of $\alpha_{\mathcal{O}}$ through the amount of information required to make $\alpha_{\mathcal{O}}$ complete [36]. Moreover, if that amount is greater for an opaque predicate P_1 than for another opaque predicate P_2 , then P_1 can be said to be more resilient wrt. $\alpha_{\mathcal{O}}$ than P_2 [36].

Complete observers are formidable because they own the knowledge they need to perform an intelligent tamper attack to software. In the case of opaque predicates, this was investigated by Dalla Preda *et al.* [39], which show how easily

observers can be made complete wrt. common numeric opaque predicates and also compare their theoretical outcomes to some experimental results. In the case of malware detection, Dalla Preda *et al.* [35] provide a characterization of malware behavior using abstractions of the trace semantics aimed at hiding those irrelevant aspects of the malicious behavior that are commonly targeted for mutation in metamorphic malware; then they provide a notion of *relative completeness* of malware detectors/eradicators, relating it to the completeness of the trace-based detector [35].

The recognition of software content via abstract interpretation is also the base of a novel watermarking scheme proposed by Cousot and Cousot in 2004 [34]. Their watermark consists of a numeric payload and a stegomark. The stegomark consists in a couple of assignments to a program variable x . These instructions are arranged so that x appears to be assigned stochastically during standard execution, while it constantly evaluates to the stegosignature if interpreted in an abstract domain parametrized by a numeric stegokey. At extraction time, the watermarked program is entered as the input of an abstract interpreter parametrized by the stegokey, and the stegosignature is easily retrieved among the set of program variables which prove to be constant [34].

A first attempt at unifying obfuscation and watermarking in unique framework based on abstract interpretation has been carried out by Giacobazzi [46].

Loop Affine Transformations

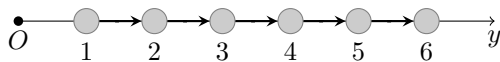
For-loops, which we introduced near the end of Chapter 2, can undergo a deal of loop transformations. In this Chapter we show that three of them, namely loop bumping, loop reversal and loop skewing, are instances of a more general transformation we call *loop affine transformation*, which can be used to move information from environments to statements and back.

4.1 Loop Bumping

One of the simplest loop transformation is perhaps *loop bumping*, which translates the iteration space of a for-loop by a fixed amount r . Consider for instance the following for-loop:

```
for (y := 1; y ≤ 6; y := y + 1) {  
  w := A + y  
}
```

Its iteration space traversal can be represented as:



After a bumping by r , we obtain:

```
for (y := 1 + r; y ≤ 6 + r; y := y + 1) {  
  w := A + (y - r)  
}
```

Notice that bumping redefines the header of the loop, enforcing a translation of the iteration space. At the same time, it undoes the translation in the body to preserve the value of all variables other than y . By kind of example, let $r = -1$. Then our bumped loop instantiates to:

```

for (y := 1 + (-1); y ≤ 6 + (-1); y := y + 1) {
  w := A + (y - (-1))
}

```

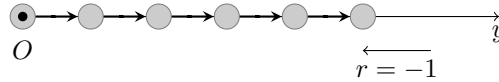
We can rewrite this program in an equivalent and shorter form:

```

for (y := 0; y ≤ 5; y := y + 1) {
  w := A + (y + 1)
}

```

The corresponding iteration space traversal is



4.2 Loop Reversal

Another simple transformation is *loop reversal*, that inverts the order in which the iterations of a for-loop are performed. Back to our example, if we apply reversal instead of bumping we obtain:

```

for (y := 6; y ≥ 1; y := y - 1) {
  w := A + y
}

```

Its related iteration space traversal is:



It can be easily obtained from the original one by applying a reflection with respect to $y = \frac{7}{2}$, which is the axis of the segment with end points $a_1 = 1$ and $a_2 = 6$. In general we have

$$y = a_1 + \frac{a_2 - a_1}{2} . \quad (4.1)$$

Then the algebraic relation implementing the reflection with respect to (4.1) is

$$y' = -y + 2 \left(a_1 + \frac{a_2 - a_1}{2} \right) ,$$

where y and y' are the old and the new value of y respectively. As it is customary in geometry, we implement the transformation by expressing the old values in terms of the new ones:

$$y = -y' + (a_1 + a_2) .$$

Reversal thereby can be performed by replacing y with $-y + (A_1 + A_2)$ in the header of:

```

for (y := A1; y ≤ A2; y := y + A3) {
  w := A + y
}

```

This results in the following object

```

for (-y + (A1 + A2) = A1;
     -y + (A1 + A2) ≤ A2;
     -y + (A1 + A2) = -y + (A1 + A2) + A3) {
  w := A + y
}

```

which ultimately yields the reversed for-loop:

```

for (y := A2; y ≥ A1; y := y - A3) {
  w := A + y
}

```

The transformation is legal as long as there are no dependencies among iterations [80].

Another way to perform reversal is to leave the iteration space unchanged and perform the reflection in the body:

```

for (y := A1; y ≤ A2; y := y + A3) {
  w := A + (-y + (A1 + A2))
}

```

By combining together both strategies, we obtain a fake reversal:

```

for (y := A2; y ≥ A1; y := y - A3) {
  w := A + (-y + (A1 + A2))
}

```

Here the value of the index variable, although coming from a reversed iteration space, is reversed again in the body. Thus there is no reversal at all: the transformation always preserves the semantics and, as a consequence, it can be performed unconditionally.

4.3 Principle of Loop Affine Transformations

Fake reversal and bumping have a common pattern: not only they perform a transformation of the iteration space just to undo it when the index variable is used, but they both are affine transformations of the iteration space. Thus they can be unified in a more general transformation, which we call *loop affine transformation*.

In order to provide a broader definition of loop affine transformations, let us consider a loop nest involving two for-loops:

```

for (x := A11; x ≤ A12; x := x + A13) {
  for (y := A21; y ≤ A22; y := y + A23) {
    H
  }
}

```

If we target the inner loop, we must take into account the existence of two index variables, x and y . Accordingly, the most general affine transformation of the value of y is

$$y' = px + qy + r \quad , \quad (4.2)$$

where $p, q, r \in \mathbb{Z}$ and $q \neq 0$. Because we target the inner loop only, we leave x unchanged:

$$x' = x \quad . \quad (4.3)$$

Let us express these affine transformations through a function $\alpha_{p,q,r} : \mathbb{Z}^2 \rightarrow \mathbb{Z}$ such that $\alpha_{p,q,r} : (x, y) \mapsto px + qy + r$. Some properties of this function are proved by the following

Proposition 4.1. *Let $\alpha_{p,q,r}$ be defined as above. Then for all $x, y \in \mathbb{Z}$:*

- (i) $\alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x, \alpha_{p,q,r}(x, y)) = y$;
- (ii) $\alpha_{p,q,r}(x, y) + k = \alpha_{p,q,r}\left(x, y + \frac{k}{q}\right)$, where $k \in \mathbb{Z}$.

Proof. We prove both properties separately.

$$\begin{aligned}
\text{Property (i): } \alpha_{p,q,r}\left(x, \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x, y)\right) &= px + q\alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x, y) + r \\
&= px + q\left(-\frac{p}{q}x + \frac{1}{q}y - \frac{r}{q}\right) + r \\
&= px + (-px + y - r) + r \\
&= y \quad .
\end{aligned}$$

$$\begin{aligned}
\text{Property (ii): } \alpha_{p,q,r}(x, y) + k &= (px + qy + r) + k \\
&= px + (qy + k) + r \\
&= px + q\left(y + \frac{1}{q}k\right) + r \\
&= \alpha_{p,q,r}\left(x, y + \frac{1}{q}k\right) \quad .
\end{aligned}$$

We thereby proved the whole statement. \square

By using α to restate both (4.2) and (4.3), we obtain the following map:

$$x' = \alpha_{1,0,0}(x, y) \quad (4.4)$$

$$y' = \alpha_{p,q,r}(x, y) \quad . \quad (4.5)$$

By expressing the old values of x and y in terms of their new values, we get the inverse map:

$$x = \alpha_{1,0,0}(x', y') = x' \quad (4.6)$$

$$y = \alpha_{-\frac{p}{q}, +\frac{1}{q}, -\frac{r}{q}}(x', y') \quad (4.7)$$

We have (4.6) because of (4.4) and $\alpha_{1,0,0}(x, y) = x$. We obtain (4.7) from property (i) in Proposition 4.1 by replacing $\alpha_{p,q,r}(x, y)$ with y' , and then x with x' ; these replacements are justified by (4.5) and (4.6) respectively. As an overall consequence, we have that (4.6) and (4.7) undo (4.4) and (4.5) respectively, and vice versa.

To apply loop affine transformation to our loop nest, we take advantage of (4.6) and (4.7). In particular, we replace any occurrence of x with x , and any occurrence of y with $\alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x, y)$. We get

```

for (x := A11; x ≤ A12; x := x + A13) {
  for (α-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x, y) = A21;
       α-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x, y) ≤ A22;
       α-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x, y) = α-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x, y) + A23) {
    H[x/x] [α-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x, y)/y]
  }
}
    
```

This of course is not a loop nest. In fact, the equations in the header of the inner loop cannot be meant as assignments on y , because their left hand side is different from y . We can remove this inconvenience by applying $\alpha_{p,q,r}(x, y)$ to both sides of each (in)equality. Consider for instance

$$\alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x, y) = A_{21} \quad ;$$

by applying $\alpha_{p,q,r}(x, y)$ to both its sides, we obtain

$$\alpha_{p,q,r}\left(x, \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x, y)\right) = \alpha_{p,q,r}(x, A_{21}) \quad ;$$

then, by Proposition 4.1, we finally get

$$y = \alpha_{p,q,r}(x, A_{21}) \quad ,$$

which can be easily meant as $y := \alpha_{p,q,r}(x, A_{21})$. We apply this procedure to $\alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x, y) \leq A_{22}$ and $\alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x, y) = \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x, y) + A_{23}$ as well.

If $q > 0$, we have:

```

for (x := A11; x ≤ A12; x := x + A13) {
  for (y := αp,q,r(x, A21);
       y ≤ αp,q,r(x, A22);
       y := y + |q|A23) {
    H[x/x] [α-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x, y)/y]
  }
}
    
```

If $q < 0$, we have:

```

for (x := A11; x ≤ A12; x := x + A13) {
  for (y := αp,q,r(x, A21);
       y ≥ αp,q,r(x, A22);
       y := y - |q|A23) {
    H[x/x] [α-p/q, 1/q, -r/q(x,y)/y]
  }
}

```

In either case, y is remapped according to (4.5); conversely, the values of the variables other than y are preserved, since (4.5) is undone in the body by (4.7). For this reason, we can perform loop affine transformation unconditionally.

Above we were concerned with two cases, where either $q > 0$ or $q < 0$. We made this distinction because the guard is reversed when $q < 0$, whereas it is not when $q > 0$. Likewise, the increment changes its sign when $q < 0$, whereas it does not when $q > 0$. We can however unify both cases by taking advantage of the following syntactic equivalences between commands:

$$\begin{aligned}
\alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, y) \leq A_2 &\equiv \begin{cases} y \leq \alpha_{p,q,r}(\mathbf{x}, A_2) & \text{if } q > 0 \\ y \geq \alpha_{p,q,r}(\mathbf{x}, A_2) & \text{if } q < 0 \end{cases} \\
\alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, y) > A_2 &\equiv \begin{cases} y > \alpha_{p,q,r}(\mathbf{x}, A_2) & \text{if } q > 0 \\ y < \alpha_{p,q,r}(\mathbf{x}, A_2) & \text{if } q < 0 \end{cases} \\
y := y + qA_3 &\equiv \begin{cases} y := y + |q|A_3 & \text{if } q > 0 \\ y := y - |q|A_3 & \text{if } q < 0 \end{cases} .
\end{aligned}$$

Then, notwithstanding if either $q > 0$ or $q < 0$, the for-loop resulting from loop affine transformation is:

```

for (x := A11; x ≤ A12; x := x + A13) {
  for (y := αp,q,r(x, A21);
       α-p/q, 1/q, -r/q(x, y) ≤ A2;
       y := y + qA3) {
    H[x/x] [α-p/q, 1/q, -r/q(x,y)/y]
  }
}

```

4.4 Loop Skewing

Table 4.1 shows that some known loop transformations reduce to loop affine transformation by suitable choice of p , q and r . Because the inner loop is parametrized by \mathbf{x} , we allow \mathbf{x} to parametrize p , q and r as well. This is apparent in loop skewing, which turns rectangular iteration spaces into non-rectangular parallelograms. For instance, it turns

	Expected for-loop's form	p	q	r
Normalization	any	0	$\frac{1}{A_{23}}$	$\frac{A_{23} - A_{21}}{A_{23}}$
Bumping	normal	0	1	any
Fake reversal	normal	0	-1	$A_{22} + 1$
Skewing	normal	1	1	0

Table 4.1. Coefficients for some loop affine transformations.

```

for (x := 1; x ≤ 6; x := x + 1) {
  for (y := 1; y ≤ 3; y := y + 1) {
    H
  }
}

```

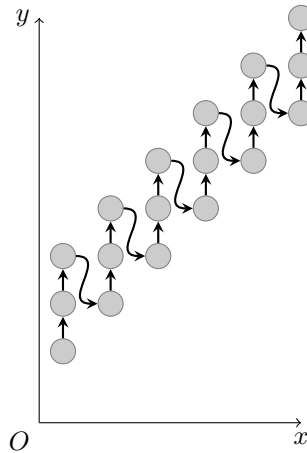
into a new loop

```

for (x := 1; x ≤ A12; x := x + 1) {
  for (y := 1 + x;
       y ≤ A22 + x;
       y := y + 1) {
    H[y-x/y]
  }
}

```

whose iteration space traversal is represented by



Skewing, being a loop affine transformation, can be performed unconditionally.

4.5 Normalization

The first and foremost important loop affine transformation is normalization, which converts for-loops in normal form. Given again our general loop nest

```

for (x := A11; x ≤ A12; x := x + A13) {
  for (y := A21; y ≤ A22; y := y + A23) {
    H
  }
}

```

after the normalization of the inner loop we get

```

for (x := A11; x ≤ A12; x := x + A13) {
  for (y := 1;
       y ≤  $\alpha_{0, \frac{1}{A_{23}}, \frac{A_{23}-A_{21}}{A_{23}}}(x, A_{22})$ ;
       y := y + 1) {
    H [ $\alpha_{0, A_{23}, A_{23}-A_{21}}(x, y)/y$ ]
  }
}

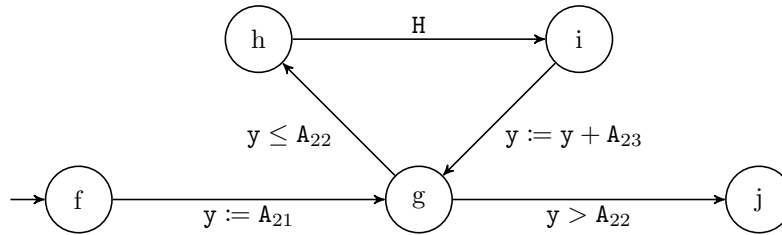
```

The values for p , q and r exploited in normalization are those specified by [80] and are possibly parametrized by x .

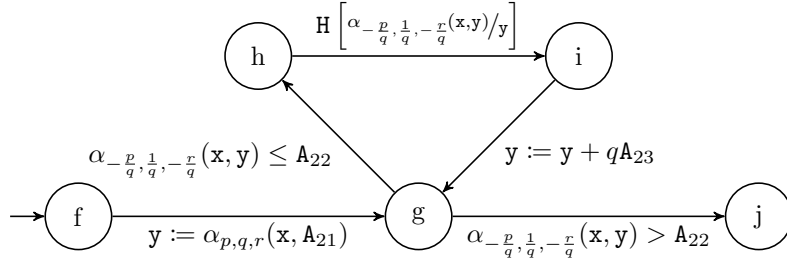
Assuming that the target for-loop is in normal form, Table 4.1 describes how to choose p , q and r to mimic bumping, fake reversal and skewing.

4.6 Syntactic Transformer

The algorithm that implements loop affine transformation takes p , q and r as parameters and a for-loop as argument. Such for-loop can be defined out of either (2.48) or (2.49). Assume we are in the former case; then our for-loop is supposed to be:



According to our previous discussion, loop affine transformation turns this loop into:



The syntactic algorithm that formally performs the transformation is defined as follows:

$$\mathcal{L}[\mathfrak{a}[\mathbb{P}]] \stackrel{\text{def}}{=} \mathcal{L}[\mathbb{P}] \quad (4.8)$$

$$\mathfrak{a}[\mathbb{P}] \stackrel{\text{def}}{=} \bigcup \{ \mathfrak{a}[\mathbb{S}] \mid \mathbb{S} \in \mathbb{P} \} \quad (4.9)$$

$$\mathfrak{a}[\mathbb{f}: \mathbb{y} := \mathbb{A}_{21} \rightarrow \mathbb{g}] \stackrel{\text{def}}{=} \{ \mathbb{f}: \mathbb{y} := \alpha_{p,q,r}(\mathbf{x}, \mathbb{A}_{21}) \rightarrow \mathbb{g} \} \quad (4.10)$$

$$\mathfrak{a}[\mathbb{g}: \mathbb{y} > \mathbb{A}_{22} \rightarrow \mathbb{j}] \stackrel{\text{def}}{=} \left\{ \mathbb{g}: \alpha_{-p/q, 1/q, -r/q}(\mathbf{x}, \mathbb{y}) > \mathbb{A}_{22} \rightarrow \mathbb{j} \right\} \quad (4.11)$$

$$\mathfrak{a}[\mathbb{g}: \mathbb{y} \leq \mathbb{A}_{22} \rightarrow \mathbb{h}] \stackrel{\text{def}}{=} \left\{ \mathbb{g}: \alpha_{-p/q, 1/q, -r/q}(\mathbf{x}, \mathbb{y}) \leq \mathbb{A}_{22} \rightarrow \mathbb{h} \right\} \quad (4.12)$$

$$\mathfrak{a}[\mathbb{h}: \mathbb{H} \rightarrow \mathbb{i}] \stackrel{\text{def}}{=} \left\{ \mathbb{h}: \mathbb{H} \left[\alpha_{-p/q, 1/q, -r/q}(\mathbf{x}, \mathbb{y}) / \mathbb{y} \right] \rightarrow \mathbb{i} \right\} \quad (4.13)$$

$$\mathfrak{a}[\mathbb{i}: \mathbb{y} := \mathbb{y} + \mathbb{A}_{23} \rightarrow \mathbb{g}] \stackrel{\text{def}}{=} \{ \mathbb{i}: \mathbb{y} := \mathbb{y} + q\mathbb{A}_{23} \rightarrow \mathbb{g} \} \quad (4.14)$$

Notice that to transform a for-loop defined out of (2.49), we should have three additional definitions, obtained from (4.12), (4.11) and (4.14) by replacing \leq , $>$ and $+$ with \geq , $<$ and $-$ respectively. Without loss of generality, in the following we are not concerned with such additional definitions.

4.7 Semantic Transformer

The syntactic transformer \mathfrak{a} includes five statement transformer, one for each statement found in a for-loop. Likewise, in the semantic transformer we have five state transformers. To improve readability, given a couple of environments ρ and ρ' , we let

$$\begin{aligned} x &\stackrel{\text{def}}{=} \mathbb{A}[x] \rho & x' &\stackrel{\text{def}}{=} \mathbb{A}[x] \rho' \\ y &\stackrel{\text{def}}{=} \mathbb{A}[y] \rho & y' &\stackrel{\text{def}}{=} \mathbb{A}[y] \rho' . \end{aligned}$$

Then we define the semantic transformer \mathfrak{a} as follows:

$$\begin{aligned}
& \mathbf{a} \langle \rho, \mathbf{f}: \mathbf{y} := \mathbf{A}_{21} \rightarrow \mathbf{g} \rangle = \\
& \quad \text{if true then} \\
& \quad \quad \langle \rho, \mathbf{f}: \mathbf{y} := \alpha_{p,q,r}(\mathbf{x}, \mathbf{y}) \rightarrow \mathbf{g} \rangle \tag{4.15}
\end{aligned}$$

$$\begin{aligned}
& \mathbf{a} \langle \rho, \mathbf{g}: \mathbf{y} > \mathbf{A}_{22} \rightarrow \mathbf{j} \rangle = \\
& \quad \text{if true then} \\
& \quad \quad \langle \rho[\mathbf{y} := \alpha_{p,q,r}(x, y)], \mathbf{g}: \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) > \mathbf{A}_{22} \rightarrow \mathbf{j} \rangle \tag{4.16}
\end{aligned}$$

$$\begin{aligned}
& \mathbf{a} \langle \rho, \mathbf{g}: \mathbf{y} \leq \mathbf{A}_{22} \rightarrow \mathbf{h} \rangle = \\
& \quad \text{if true then} \\
& \quad \quad \langle \rho[\mathbf{y} := \alpha_{p,q,r}(x, y)], \mathbf{g}: \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) \leq \mathbf{A}_{22} \rightarrow \mathbf{h} \rangle \tag{4.17}
\end{aligned}$$

$$\begin{aligned}
& \mathbf{a} \langle \rho, \mathbf{h}: \mathbf{H} \rightarrow \mathbf{i} \rangle = \\
& \quad \text{if true then} \\
& \quad \quad \langle \rho[\mathbf{y} := \alpha_{p,q,r}(x, y)], \mathbf{h}: \mathbf{H} \left[\alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) / \mathbf{y} \right] \rightarrow \mathbf{i} \rangle \tag{4.18}
\end{aligned}$$

$$\begin{aligned}
& \mathbf{a} \langle \rho, \mathbf{i}: \mathbf{y} := \mathbf{y} + \mathbf{A}_{23} \rightarrow \mathbf{g} \rangle = \\
& \quad \text{if true then} \\
& \quad \quad \langle \rho[\mathbf{y} := \alpha_{p,q,r}(x, y)], \mathbf{i}: \mathbf{y} := \mathbf{y} + q\mathbf{A}_{23} \rightarrow \mathbf{g} \rangle \tag{4.19}
\end{aligned}$$

There is a natural correspondence between (4.10) and (4.15), because they both work on the same statement and they transform it in the same way. A similar argument holds for (4.11) and (4.16), for (4.12) and (4.17), and so on. Additionally, state transformers work on environments. In (4.10) the initial environment ρ is preserved. In the following definitions, ρ is turned to $\rho[\mathbf{y} := \alpha_{p,q,r}(x, y)]$ to account for the transformed initialization and the transformed increment that are found in the transformed loop.

4.8 Local Commutation Condition

Each state transformer in the auxiliary function \mathbf{a}' is meant to perfectly undo its correspondent state transformer in \mathbf{a} :

$$\begin{aligned}
& \mathbf{a}' \langle \rho', \mathbf{f}: \mathbf{y} := \mathbf{A} \rightarrow \mathbf{g} \rangle = \\
& \quad \langle \rho', \mathbf{f}: \mathbf{y} := \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{A}) \rightarrow \mathbf{g} \rangle \tag{4.20}
\end{aligned}$$

$$\begin{aligned}
& \mathbf{a}' \langle \rho', \mathbf{g}: \mathbf{A} \leq \mathbf{A}_{22} \rightarrow \mathbf{h} \rangle = \\
& \quad \langle \rho'[\mathbf{y} := \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x', y')], \mathbf{g}: \alpha_{p,q,r}(\mathbf{x}, \mathbf{A}) \leq \mathbf{A}_{22} \rightarrow \mathbf{h} \rangle \tag{4.21}
\end{aligned}$$

$$\begin{aligned}
& \mathbf{a}' \langle \rho', \mathbf{h}: \mathbf{H}' \rightarrow \mathbf{i} \rangle = \\
& \quad \langle \rho'[\mathbf{y} := \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x', y')], \mathbf{h}: \mathbf{H}' \left[\alpha_{p,q,r}(\mathbf{x}, \mathbf{y}) / \mathbf{y} \right] \rightarrow \mathbf{i} \rangle \tag{4.22}
\end{aligned}$$

$$\begin{aligned}
& \mathbf{a}' \langle \rho', \mathbf{i}: \mathbf{y} := \mathbf{y} + \mathbf{A}_{23} \rightarrow \mathbf{g} \rangle = \\
& \quad \langle \rho'[\mathbf{y} := \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x', y')], \mathbf{i}: \mathbf{y} := \mathbf{y} + \frac{\mathbf{A}_{23}}{q} \rightarrow \mathbf{g} \rangle \tag{4.23}
\end{aligned}$$

In particular, (4.20) undoes the transformation of (4.15), whereas (4.21) undoes the transformation of (4.17) and so on. We did not introduce state transformers in \mathbf{a}' that undoes (4.16). In fact if such state appears in some trace in the semantics of the transformed for-loop, it is the last state in that trace. In such case there is no need to define \mathbf{a}' , since no state is supposed to come after.

We can now formally prove that \mathbf{a} and \mathbf{a}' are respectively the syntactic and the semantic counterpart of any loop affine transformation.

Theorem 4.2. *Let P be a for-loop. Then $\forall n \in \mathbb{N}. F_{\mathbf{a}}^n[[P]]\emptyset = F^n[[\mathbf{a}[[P]]]]\emptyset$.*

Proof. We let $\mathfrak{t} = \mathbf{a}$, $\mathfrak{t} = \mathbf{a}$ and $\mathfrak{t}' = \mathbf{a}'$, and just prove (2.81) and (2.82).

First we prove (2.81).

$$\begin{aligned}
\mathbf{a}(\mathcal{J}[[P]]) &= \mathbf{a}(\{ \langle \rho, S \rangle \mid \rho \in \mathfrak{E}[[P]] \wedge S \in P \wedge \text{lab}[[S]] \in \mathfrak{L}[[P]] \}) && \text{by (2.45)} \\
&= \{ \mathbf{a} \langle \rho, S \rangle \mid \rho \in \mathfrak{E}[[P]] \wedge S \in P \wedge \text{lab}[[S]] \in \mathfrak{L}[[P]] \} && \text{by (2.68)} \\
&= \{ \mathbf{a} \langle \rho, f: y := A_{21} \rightarrow g \rangle \mid \rho \in \mathfrak{E}[[P]] \} \\
&= \{ \langle \rho, f: y := \alpha_{p,q,r}(\mathbf{x}, A_{21}) \rightarrow g \rangle \mid \rho \in \mathfrak{E}[[P]] \} && \text{by (4.15)} \\
&= \{ \langle \rho, f: y := \alpha_{p,q,r}(\mathbf{x}, A_{21}) \rightarrow g \rangle \mid \rho \in \mathfrak{E}[[\mathbf{a}[[P]]]] \} && \text{by definition of } \mathbf{a} \\
&= \{ \langle \rho, S \rangle \mid \rho \in \mathfrak{E}[[P]] \wedge S \in \mathbf{a}[[P]] \wedge \text{lab}[[S]] \in \mathfrak{L}[[\mathbf{a}[[P]]]] \} \\
&= \mathcal{J}[[\mathbf{a}[[P]]]] && \text{by (2.45)}
\end{aligned}$$

We now prove (2.82).

- Let η be such that $\neg \eta = \langle \rho', f: y := \alpha_{p,q,r}(\mathbf{x}, A_{21}) \rightarrow g \rangle$.

On the one hand,

let $\rho'' = \rho' [y := \mathbf{A}[\alpha_{p,q,r}(\mathbf{x}, A_{21})]] \rho'$.

$$\begin{aligned}
\text{Then } R_{\mathbf{a}}(\eta) &= \left\{ \left\langle \rho'', g: \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) \leq A_{22} \rightarrow h \right\rangle, \right. \\
&\quad \left. \left\langle \rho'', g: \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) > A_{22} \rightarrow j \right\rangle \right\}.
\end{aligned}$$

On the other hand,

we have $\mathbf{a}'(\neg \eta) = (4.20)$

$$\begin{aligned}
&= \left\langle \rho', f: y := \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \alpha_{p,q,r}(\mathbf{x}, \mathbf{y})) \rightarrow g \right\rangle \\
&= \left\langle \rho', f: y := A_{21} \rightarrow g \right\rangle
\end{aligned}$$

by Proposition 4.1.

Let $\rho = \rho' [y := \mathbf{A}[\alpha_{p,q,r}(\mathbf{x}, A_{21})]] \rho'$.

If $\mathbf{B}[\mathbf{y} \leq A_{22}] \rho$ then

$$\mathbf{a}(s) = \mathbf{a} \langle \rho, g: \mathbf{y} \leq A_{22} \rightarrow h \rangle$$

$$= (4.17)$$

$$= \left\langle \rho [y := \alpha_{p,q,r}(x, y)], g: \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) \leq A_{22} \rightarrow h \right\rangle$$

$$= \left\langle \rho' [y := \mathbf{A}[[A_{21}]] \rho'] [y := \alpha_{p,q,r}(x, y)], g: \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) \leq A_{22} \rightarrow h \right\rangle$$

$$= \left\langle \rho' [y := \mathbf{A}[\alpha_{p,q,r}(\mathbf{x}, A_{21})]] \rho', g: \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) \leq A_{22} \rightarrow h \right\rangle$$

$$= \left\langle \rho'', g: \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) \leq A_{22} \rightarrow h \right\rangle.$$

If $\mathbb{B}[\mathbf{y} > \mathbf{A}_{22}] \rho$ then

$$\begin{aligned}
\mathbf{a}(s) &= \mathbf{a} \langle \rho, \mathbf{g}: \mathbf{y} > \mathbf{A}_{22} \rightarrow \mathbf{j} \rangle \\
&= (4.16) \\
&= \langle \rho[\mathbf{y} := \alpha_{p,q,r}(x, y)], \mathbf{g}: \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) > \mathbf{A}_{22} \rightarrow \mathbf{j} \rangle \\
&= \langle \rho'[\mathbf{y} := \mathbf{A}[\mathbf{A}_{21}] \rho'][\mathbf{y} := \alpha_{p,q,r}(x, y)], \mathbf{g}: \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) > \mathbf{A}_{22} \rightarrow \mathbf{j} \rangle \\
&= \langle \rho'[\mathbf{y} := \mathbf{A}[\alpha_{p,q,r}(\mathbf{x}, \mathbf{A}_{21})] \rho'], \mathbf{g}: \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) > \mathbf{A}_{22} \rightarrow \mathbf{j} \rangle \\
&= \langle \rho'', \mathbf{g}: \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) > \mathbf{A}_{22} \rightarrow \mathbf{j} \rangle .
\end{aligned}$$

- Let η be such that $\neg \eta = \langle \rho', \mathbf{g}: \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) \leq \mathbf{A}_{22} \rightarrow \mathbf{h} \rangle$.

On the one hand,

let $\rho'' = \rho'$.

$$\text{Then } R_{\mathbf{a}}(\eta) = \left\{ \langle \rho'', \mathbf{h}: \mathbf{H} \left[\alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) / \mathbf{y} \right] \rightarrow \mathbf{i} \rangle \right\}.$$

On the other hand,

we have

$$\begin{aligned}
\mathbf{a}'(\neg \eta) &= (4.21) \\
&= \langle \rho'[\mathbf{y} := \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x', y')], \mathbf{g}: \alpha_{p,q,r}(\mathbf{x}, \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y})) \leq \mathbf{A}_{22} \rightarrow \mathbf{h} \rangle \\
&= \langle \rho'[\mathbf{y} := \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x', y')], \mathbf{g}: \mathbf{y} \leq \mathbf{A}_{22} \rightarrow \mathbf{h} \rangle \text{ by Proposition 4.1.}
\end{aligned}$$

$$\text{Let } \rho = \rho'[\mathbf{y} := \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x', y')].$$

Then

$$\begin{aligned}
\mathbf{a}(s) &= (4.22) \\
&= \langle \rho[\mathbf{y} := \alpha_{p,q,r}(x, y)], \mathbf{h}: \mathbf{H} \left[\alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) / \mathbf{y} \right] \rightarrow \mathbf{g} \rangle \\
&= \langle \rho'[\mathbf{y} := \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x', y')][\mathbf{y} := \alpha_{p,q,r}(x, y)], \mathbf{h}: \mathbf{H} \left[\alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) / \mathbf{y} \right] \rightarrow \mathbf{g} \rangle \\
&= \langle \rho'[\mathbf{y} := \mathbf{A}[\alpha_{p,q,r}(\mathbf{x}, \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x', y'))] \rho'], \mathbf{h}: \mathbf{H} \left[\alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) / \mathbf{y} \right] \rightarrow \mathbf{g} \rangle \\
&= \langle \rho'[\mathbf{y} := \mathbf{A}[\mathbf{y}] \rho'], \mathbf{h}: \mathbf{H} \left[\alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) / \mathbf{y} \right] \rightarrow \mathbf{g} \rangle \text{ by Proposition 4.1} \\
&= \langle \rho', \mathbf{h}: \mathbf{H} \left[\alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) / \mathbf{y} \right] \rightarrow \mathbf{g} \rangle \\
&= \langle \rho'', \mathbf{h}: \mathbf{H} \left[\alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) / \mathbf{y} \right] \rightarrow \mathbf{g} \rangle .
\end{aligned}$$

- Let η be such that $\neg \eta = \langle \rho', \mathbf{h}: \mathbf{H} \left[\alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) / \mathbf{y} \right] \rightarrow \mathbf{i} \rangle$.

On the one hand,

$$\text{we have } R_{\mathbf{a}}(\eta) = \left\{ \langle \rho'', \mathbf{i}: \mathbf{y} := \mathbf{y} + q\mathbf{A}_{23} \rightarrow \mathbf{g} \rangle \mid \rho'' \in \mathbf{C} \left[\left[\mathbf{H} \left[\alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x', y') / \mathbf{y} \right] \right] \rho' \right\}.$$

On the other hand,

we have

$$\begin{aligned}
\mathbf{a}'(\neg\eta) &= (4.22) \\
&= \left\langle \rho' \left[\mathbf{y} := \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x', y') \right], \mathbf{h}: \mathbf{H} \left[\alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) / \mathbf{y} \right] \left[\alpha_{p, q, r}(\mathbf{x}, \mathbf{y}) / \mathbf{y} \right] \rightarrow \mathbf{i} \right\rangle \\
&= \left\langle \rho' \left[\mathbf{y} := \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x', y') \right], \mathbf{h}: \mathbf{H} \left[\alpha_{p, q, r} \left(\mathbf{x}, \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) \right) / \mathbf{y} \right] \rightarrow \mathbf{i} \right\rangle \\
&= \left\langle \rho' \left[\mathbf{y} := \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x', y') \right], \mathbf{h}: \mathbf{H} \rightarrow \mathbf{i} \right\rangle \text{ by Proposition 4.1.}
\end{aligned}$$

Let $\rho \in \mathbf{C}[\mathbf{H}] \rho' \left[\mathbf{y} := \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x', y') \right]$.

Then $\mathbf{a}(s) = (4.19)$

$$= \left\langle \rho \left[\mathbf{y} := \alpha_{p, q, r}(x, y) \right], \mathbf{i}: \mathbf{y} := \mathbf{y} + q\mathbf{A}_{23} \rightarrow \mathbf{g} \right\rangle ,$$

where $\rho \left[\mathbf{y} := \alpha_{p, q, r}(x, y) \right] \in \mathbf{C} \left[\mathbf{H} \left[\alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x', y') / \mathbf{y} \right] \right] \rho'$ by Lemma 2.4.

- Let η be such that $\neg\eta = \langle \rho', \mathbf{i}: \mathbf{y} := \mathbf{y} + q\mathbf{A}_{23} \rightarrow \mathbf{g} \rangle$.

On the one hand,

let $\rho'' = \rho' \left[\mathbf{y} := \mathbf{A}[\mathbf{y} + q\mathbf{A}_{23}] \rho' \right]$.

$$\begin{aligned}
\text{Then } R_{\mathbf{a}}(\eta) &= \left\{ \left\langle \rho'', \mathbf{g}: \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) \leq \mathbf{A}_{22} \rightarrow \mathbf{h} \right\rangle, \right. \\
&\quad \left. \left\langle \rho'', \mathbf{g}: \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) > \mathbf{A}_{22} \rightarrow \mathbf{j} \right\rangle \right\} .
\end{aligned}$$

On the other hand,

we have $\mathbf{a}'(\neg\eta) = (4.23)$

$$\begin{aligned}
&= \left\langle \rho' \left[\mathbf{y} := \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x', y') \right], \mathbf{i}: \mathbf{y} := \mathbf{y} + \frac{q\mathbf{A}_{23}}{q} \rightarrow \mathbf{g} \right\rangle \\
&= \left\langle \rho' \left[\mathbf{y} := \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x', y') \right], \mathbf{i}: \mathbf{y} := \mathbf{y} + \mathbf{A}_{23} \rightarrow \mathbf{g} \right\rangle
\end{aligned}$$

$$\begin{aligned}
\text{Let } \rho &= \rho' \left[\mathbf{y} := \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x', y') \right] \left[\mathbf{y} := \mathbf{A}[\mathbf{y} + \mathbf{A}_{23}] \rho' \right] \\
&= \rho' \left[\mathbf{y} := \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x', y') \right] \left[\mathbf{y} := \mathbf{A}[\mathbf{y} + \mathbf{A}_{23}] \rho' \left[\mathbf{y} := \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x', y') \right] \right] \\
&= \rho' \left[\mathbf{y} := \mathbf{A}[\mathbf{y} + \mathbf{A}_{23}] \rho' \left[\mathbf{y} := \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x', y') \right] \right] \\
&= \rho' \left[\mathbf{y} := \mathbf{A} \left[\alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) + \mathbf{A}_{23} \right] \rho' \right] .
\end{aligned}$$

If $\mathbf{B}[\mathbf{y} \leq \mathbf{A}_{22}] \rho$ then

$$\mathbf{a}(s) = \mathbf{a} \left\langle \rho, \mathbf{g}: \mathbf{y} \leq \mathbf{A}_{22} \rightarrow \mathbf{h} \right\rangle$$

$$= (4.17)$$

$$= \left\langle \rho \left[\mathbf{y} := \alpha_{p, q, r}(x, y) \right], \mathbf{g}: \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x, y) \leq \mathbf{A}_{22} \rightarrow \mathbf{h} \right\rangle$$

and

$$\begin{aligned}
\rho \left[\mathbf{y} := \alpha_{p, q, r}(x, y) \right] &= \rho' \left[\mathbf{y} := \mathbf{A} \left[\alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) + \mathbf{A}_{23} \right] \rho' \right] \left[\mathbf{y} := \alpha_{p, q, r}(x, y) \right] \\
&= \rho' \left[\mathbf{y} := \mathbf{A} \left[\alpha_{p, q, r} \left(\mathbf{x}, \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) + \mathbf{A}_{23} \right) \right] \rho' \right] \\
&= \rho' \left[\mathbf{y} := \mathbf{A} \left[\alpha_{p, q, r} \left(\mathbf{x}, \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(\mathbf{x}, \mathbf{y}) \right) + q\mathbf{A}_{23} \right] \rho' \right] \text{ by Proposition 4.1} \\
&= \rho' \left[\mathbf{y} := \mathbf{y} + q\mathbf{A}_{23} \right] \rho' \text{ by Proposition 4.1} \\
&= \rho'' .
\end{aligned}$$

If $\mathbf{B}[\mathbf{y} > \mathbf{A}_{22}] \rho$ then

$$\begin{aligned}
\mathbf{a}(s) &= \mathbf{a} \langle \rho, \mathbf{g}: \mathbf{y} > \mathbf{A}_{22} \rightarrow \mathbf{j} \rangle \\
&= (4.16) \\
&= \left\langle \rho[\mathbf{y} := \alpha_{p,q,r}(x, y)], \mathbf{g}: \alpha_{-\frac{p}{q}, \frac{1}{q}, -\frac{r}{q}}(x, y) > \mathbf{A}_{22} \rightarrow \mathbf{j} \right\rangle
\end{aligned}$$

and $\rho[\mathbf{y} := \alpha_{p,q,r}(x, y)] = \rho''$ by the same argument above.

□

Loop Unrolling

Any for-loop subsumes a number of iterations. The execution of such iterations entails an overhead, due to the evaluation of the guard and of the increment. To reduce overhead, program compilers usually let for-loops undergo transformations aimed at lowering their number of iterations [2]. In this Chapter we introduce some of these transformations, namely loop peeling, loop splitting and loop unrolling, and we show that they all reduce to loop unrolling, which is recast as a transformation that shifts information only from environments to statements – not the other way round.

For the sake of simplicity, throughout this Chapter we consider a for-loop P defined out of (2.48):

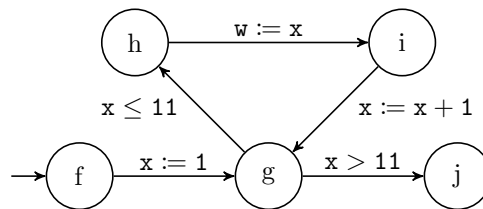
```

for (x := A1; x ≤ A2; x := x + A3) {
  H
}

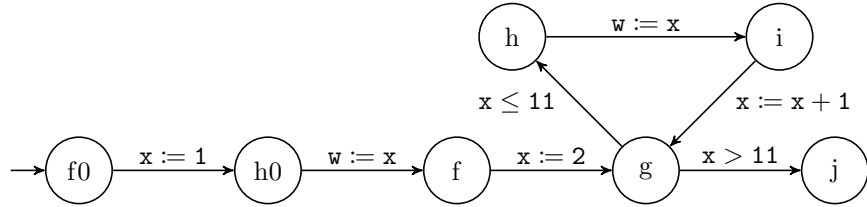
```

5.1 Loop Peeling

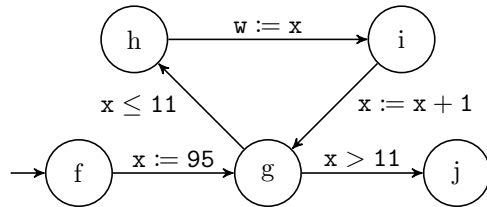
The simplest way to reduce the number of iterations of P is to get rid of its first iteration. This is carried out by *loop peeling* [2]. Consider for instance the following for-loop:



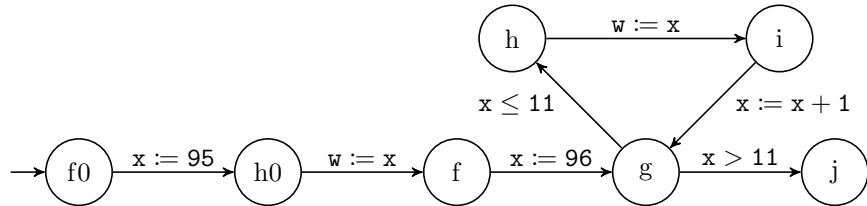
After peeling off its first iteration, we obtain:



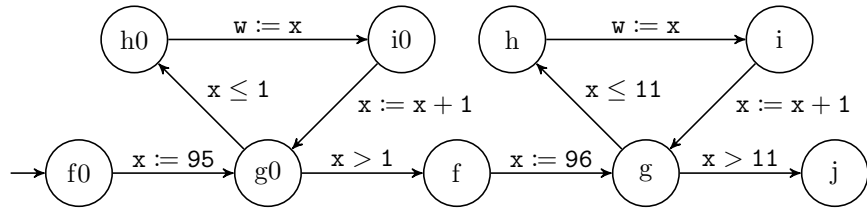
It is easy to verify both the original and the transformed program carry out the same task. In the former program, however, all assignments on w are performed after the guard has been checked. In the latter one, on the other side, the first assignment on w is performed unconditionally. This can be harmful in case the guard branches to j without entering the loop. Let us consider a slight modification of our example loop, where we raised the initial value of x to 95:



This loop does not perform any iterations. However, after the peeling takes place, one assignment on w occurs nonetheless:



To ensure correctness, we need to check the guard before w is assigned. A straightforward solution is to wrap the assignment we peeled off in a new for-loop and let such loop come just before the old for-loop:



We have therefore a sequence of two for-loops, which we note as:

```

for (x := 95; x ≤ 1; x := x + 1) {
  w := x
}
for (x := 96; x ≤ 11; x := x + 1) {
  w := x
}

```

Since we have two loops instead of one, we have come up with a double overhead, indeed. This is the price we had to pay to properly peel out the first iteration. Notice that the two loops differs not only in the initial value they provide to x , but also in the value which they test x against. This modification accounts for the way we accomplish peeling: not by detecting the first iteration (if any) in order to close it off in the first for-loop, but rather by specifying how many iterations we want the second for-loop to subsume. To make this point more clear, let us consider again our first example; if we let it undergo peeling correctly, we obtain:

```

for (x := 1; x ≤ 1; x := x + 1) {
  w := x
}
for (x := 2; x ≤ 11; x := x + 1) {
  w := x
}

```

Observe that the bound of the first loop was set to 1. If we expected the second loop to perform no iterations, we would have set that bound to 11. However, because we wanted the second loop to subsume ten iterations, we reduced it from 11 to 1. As a side-effect, we closed off the first iteration in its own for-loop. Thus we just accomplished loop peeling.

5.2 Loop Splitting

Consider again our original for-loop:

```

for (x := 1; x ≤ 11; x := x + 1) {
  w := x
}

```

It is easy to verify that its iteration space is $I = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$. Suppose we want to apply loop peeling, yet requiring the second loop to subsume only $l = 5$ iterations instead of ten. Then we have:

```

for (x := 1; x ≤ 6; x := x + 1) {
  w := x
}
for (x := 7; x ≤ 11; x := x + 1) {
  w := x
}

```

The original iteration space was split into two iteration spaces:

- $\{7, 8, 9, 10, 11\}$, which is traversed last and includes the last $l = 5$ iterations of the original for-loop;
- $\{1, 2, 3, 4, 5, 6\}$, which is traversed first and collects all the remaining iterations.

We refer to $l \in \mathbb{N}$ as the *splitting term*. In general, given our generic for-loop P defined as

```
for (x := A1; x ≤ A2; x := x + A3) {
  H
}
```

and given a splitting term l , we can replace P with the following sequence of for-loops:

```
for (x := A1; x ≤ A2 - l; x := x + A3) {
  H
}
for (; x ≤ A2; x := x + A3) {
  H
}
```

Such transformation is called *loop splitting* [80]. Notice the deficiencies that are found in the header of the second loop; indeed, x needs not to be initialized, since its value is provided by the first loop. Loop splitting partitions the iterations of P between the two for-loops. In particular, the second loop subsumes:

- all the iterations of P , if l is greater than the total number of iterations;
- only the last l iterations of P , otherwise.

A formal argument for this behavior is provided later by Proposition 5.2. The first loop subsumes all the iterations that are not subsumed by the second one. Since the two loops are executed sequentially, the order of execution of the iterations of P is preserved.

Loop splitting is trivial if $l = 0$. Loop peeling is just an instance of loop splitting. Notice that, while splitting can be performed unconditionally, peeling must comply with the following enabling condition:

$$\exists l \in \mathbb{N}. \forall \rho \in \mathfrak{E}[P]. A[A_1] \rho + l = A[A_2] \rho .$$

5.3 Loop Unrolling

Loop splitting provides for-loops with a lowered number of iterations. In our example above, from a for-loop with eleven iterations

```
for (x := 1; x ≤ 11; x := x + 1) {
  w := x
}
```

we obtained two for-loops with six and five iterations:

```

for (x := 1; x ≤ 6; x := x + 1) {
    w := x
}
for (; x ≤ 11; x := x + 1) {
    w := x
}

```

However, splitting does not reduce the overall number of iterations, which in our example amounts to eleven before as well as after splitting is performed. To actually lower that amount, we need a different strategy.

The first loop performs six iterations, resulting in the following sequence of commands:

```

x := 1,
x ≤ 6, w := x, x := x + 1,
x ≤ 6, w := x, x := x + 1,
x ≤ 6, w := x, x := x + 1,
x ≤ 6, w := x, x := x + 1,
x ≤ 6, w := x, x := x + 1,
x ≤ 6, w := x, x := x + 1,
x > 6.

```

Notice that each iteration performs an assignment on w . If an iteration performed $u = 3$ assignments, only two iterations would be needed:

```

x := 1,
x ≤ 6, w := x,
w := x,
w := x, x := x + 1,
x ≤ 6, w := x,
w := x,
w := x, x := x + 1,
x > 6.

```

To ensure that w is assigned correctly, we also need to replace every x occurring in an arithmetic expression with $x + m$, where m accounts for the suppressed increments:

```

x := 1,
x ≤ 4, w := x + 0,
w := x + 1,
w := x + 2, x := (x + 2) + 1,
x ≤ 4, w := x + 0,
w := x + 1,
w := x + 2, x := (x + 2) + 1,
x > 4.

```

Notice that the new bound 4, which x is tested against, is obtained as a difference between the previous bound, 6, and $u - 1 = 3 - 1 = 2$. Recall that 6, in turn, was obtained as a difference between the original bound 11 and $l = 5$. All in all,

the new bound is obtained as $11 - (l + u - 1) = 11 - (5 + 3 - 1) = 4$. A formal argument for this choice is provided later by Proposition 5.3.

The sequence of commands displayed above is just the one resulting from the execution of the following for-loop:

```
for (x := 1; x ≤ 4; x := x + 3) {
  w := x
  w := x + 1
  w := x + 2
}
```

In the field of program compilers, the transformation that yields such loop is called *loop unrolling* and $u = 3$ is called the *unrolling factor* [2]. We require $u \in \mathbb{N}$ and $u > 0$.

If we consider again the two for-loops we got from splitting and we unroll the first one by $u = 3$, we obtain:

```
for (x := 1; x ≤ 4; x := x + 3) {
  w := x
  w := x + 1
  w := x + 2
}
for (; x ≤ 11; x := x + 1) {
  w := x
}
```

Now, as these for-loops perform two and five iterations respectively, the overall number of iterations is seven. Since the original for-loop subsumed eleven iterations, we achieved an actual reduction in the overall number of iterations.

In our examples above we used $x := x + 1$ as increment. We now deal with the case when the increment amount is greater than one. Consider for instance the following for-loop:

```
for (x := 1; x ≤ 18; x := x + 5) {
  w := x
}
```

Such loop performs four iterations, each time incrementing x by 5. The correspondent sequence of commands is:

```

x := 1,
x ≤ 18, w := x, x := x + 5,
x ≤ 18, w := x, x := x + 5,
x ≤ 18, w := x, x := x + 5,
x ≤ 18, w := x, x := x + 5,
x > 18.
```

Suppose we want to perform loop unrolling by $l = 0$ and $u = 4$. Then we have:

```

                                x := 1,
x ≤ 3, w := x + 0,
                                w := x + 5,
                                w := x + 10,
                                w := x + 15, x := (x + 15) + 5,
x > 3.

```

Here, recalling that the original increment amount is 5, we replaced every x occurring in an arithmetic expression with $x + m \cdot 5$, where m counts the number of suppressed increments. Furthermore we obtained the new bound 3 as a difference between the original bound 18 and $(l + u - 1) \cdot 5 = (0 + 4 - 1) \cdot 5 = 15$. As before, the sequence of commands displayed above is just the one resulting from:

```

for (x := 1; x ≤ 3; x := x + 20) {
    w := x
    w := x + 5
    w := x + 10
    w := x + 15
}

```

So far we have unrolled for-loops denoting a number of iterations that is a multiple of u . We can also unroll for-loops which do not comply with this assumption. Suppose we unroll the previous for-loop by $l = 0$ and by $u' = 3$ instead of $u = 4$. In such case unrolling encompasses only three of the four iterations subsumed by the loop. Hence we have

```

                                x := 1,
x ≤ 8, w := x + 0,
                                w := x + 5,
                                w := x + 10, x := (x + 10) + 5,
x > 8,
x ≤ 18, w := x,      x := x + 5
x > 18.

```

This sequence of commands is also obtained by the following for-loops:

```

for (x := 1; x ≤ 8; x := x + 15) {
    w := x
    w := x + 5
    w := x + 10
}
for (; x ≤ 18; x := x + 5) {
    w := x
}

```

The iteration that could not take part in the unrolling is performed by the second loop. A for-loop that subsumes iterations left over by an unrolled loop is called the *epilogue* of the unrolled loop [2]. In our example the second loop is thereby the epilogue of the first loop.

We now gather all the points we have made about loop unrolling. Let us consider our generic for-loop P which is defined as

$$\begin{array}{l} \text{for } (x := A_1; x \leq A_2; x := x + A_3) \{ \\ \quad H \\ \} \end{array}$$

If we let P undergo loop unrolling with splitting term $l \in \mathbb{N}$ and unrolling factor $u \in \mathbb{N}$, $u > 0$, we get:

$$\begin{array}{l} \text{for } (x := A_1; x \leq A_2 - (l + u - 1)A_3; x := x + uA_3) \{ \\ \quad H \\ \quad H[x + A_3/x] \\ \quad H[x + 2A_3/x] \\ \quad \vdots \\ \quad H[x + mA_3/x] \\ \quad \vdots \\ \quad H[x + (u-1)A_3/x] \\ \} \\ \text{for } (; x \leq A_2; x := x + A_3) \{ \\ \quad H \\ \} \end{array}$$

Notice that the second for-loop is P without the initialization of x .

It is easy to verify that loop splitting is just an instance of loop unrolling where $u = 1$.

5.4 Principle of Loop Unrolling

In the previous Section we introduced loop unrolling by reasoning not on iterations, which are semantic notions, but on sequences of commands, which were chosen as an informal yet convenient abstraction. This allowed us to make several points about unrolling without getting immediately lost in a mass of details. In the present Section we take a more formal approach, based on iterations.

Let P be our generic for-loop and let

$$\sigma \in \text{dg}(\mathcal{S}[P]) . \quad (5.1)$$

The number of residual iterations of such trace is $R_{x, A_2, A_3}(\sigma)$. For the sake of simplicity, we shorten R_{x, A_2, A_3} to R . So σ has $R(\sigma)$ residual iterations.

Let us define:

- a sequence $\lambda_j \in \mathbb{N}$. l_j of splitting terms, such that $l_0 = 0$;
- a sequence $u_j \in \mathbb{N}$. u_j of unrolling factors, such that $u_0 = 1$.

Whenever we are to perform loop unrolling, we specify an index $i \in \mathbb{N}$ and obtain a splitting term l_i and an unrolling factor u_i .

Suppose we want to unroll \mathbb{P} by splitting term l_i and unrolling factor u_i . In the first place, this entails that σ is partitioned in two subtraces σ' and σ'' such that:

$$\sigma', \sigma'' \in \mathbf{d}_g(\mathcal{S}[\mathbb{P}]) \ , \quad (5.2)$$

$$\sigma' \sigma'' = \sigma \quad (5.3)$$

and

$$\#\sigma' \stackrel{\text{def}}{=} \min(\#\sigma, (R(\sigma) \ominus l_i) \div u_i) \ . \quad (5.4)$$

We know from the previous Section that unrolling makes two for-loops out of \mathbb{P} . Subtrace σ' just accounts for the first for-loop, whereas σ'' accounts for the second one. We also know that the second for-loop is just \mathbb{P} ; hence we expect unrolling to act on σ'' as the identical transformation. The first for-loop retains the initialization of \mathbb{P} , but must be provided with novel guard, novel increment and novel body; thus we expect unrolling to deeply affect σ' .

The number of iterations subsumed by σ' is predicted by (5.4). As regards σ'' , by (5.3), (5.2) and Proposition 2.5, we have:

$$\#\sigma'' = \#\sigma - \#\sigma' \ . \quad (5.5)$$

Then by (5.4) and (5.5) we obtain:

$$\#\sigma'' = \begin{cases} \#\sigma - (R(\sigma) \ominus l_i) \div u_i & \text{if } \#\sigma > (R(\sigma) \ominus l_i) \div u_i \\ 0 & \text{otherwise} \end{cases} \ . \quad (5.6)$$

Equations (5.4) and (5.6) imply that the second for-loop subsumes all the iterations of σ whenever l_i or u_i are great enough, namely if $l_i > R(\sigma)$ or $(R(\sigma) \ominus l_i) \div u_i = 0$; this ultimately means $\sigma'' = \sigma$. Otherwise, the first for-loop is given as many iterations of σ as possible, not more than $(R(\sigma) \ominus l_i) \div u_i$ anyhow, while the second for-loop is given the remaining iterations of σ , if any.

We now devise a guard for the first for-loop. First, we provide an upper bound to the number of the residual iterations at σ'' .

Proposition 5.1. *If $\sigma'' \neq \varepsilon$ then $R(\sigma'') \leq l_i + u_i - 1$.*

Proof. Let $\sigma'' \neq \varepsilon$.

$$\#\sigma'' = \#\sigma - (R(\sigma) \ominus l_i) \div u_i \quad \text{by (5.6)}$$

$$\#(\sigma'', \dashv\sigma'') = \#(\sigma, \dashv\sigma) - (R(\sigma) \ominus l_i) \div u_i \quad \text{by (2.59)}$$

$$R(\sigma'') - R(\dashv\sigma'') = (R(\sigma) - R(\dashv\sigma)) - (R(\sigma) \ominus l_i) \div u_i \quad \text{by (2.57)}$$

$$R(\sigma'') - R(\dashv\sigma) = (R(\sigma) - R(\dashv\sigma)) - (R(\sigma) \ominus l_i) \div u_i \quad \text{by (5.3),}$$

whence:

$$R(\sigma'') = R(\sigma) - (R(\sigma) \ominus l_i) \div u_i \ .$$

We provide an upper bound to the rightmost term as follows:

$$\begin{aligned}
R(\sigma) - (R(\sigma) \ominus l_i) &:: u_i \\
&\leq (l_i + (R(\sigma) \ominus l_i) :: u_i + (R(\sigma) \ominus l_i) \bmod u_i) - (R(\sigma) \ominus l_i) :: u_i \\
&= l_i + (R(\sigma) \ominus l_i) \bmod u_i \\
&\leq l_i + u_i - 1 .
\end{aligned}$$

Then, by transitivity, we have $R(\sigma'') \leq l_i + u_i - 1$. \square

We are concerned with the last $l_i + u_i - 1$ residual iterations of σ .

- If $\sigma' = \varepsilon$ then $\sigma'' = \sigma$ by (5.3); furthermore, $R(\sigma) = R(\sigma'') \leq l_i + u_i - 1$.
- If $\sigma' \neq \varepsilon$ then $R(\sigma) = R(\sigma') > R(\sigma'')$. Moreover:
 - if $R(\sigma'') = l_i + u_i - 1$ then the last $l_i + u_i - 1$ residual iterations at σ perfectly coincide with the residual iterations at σ'' ;
 - if $R(\sigma'') < l_i + u_i - 1$ then the last $l_i + u_i - 1$ residual iterations at σ not only encompass every residual iteration at σ'' , but also include all but the first $R(\sigma') - (l_i + u_i - 1)$ residual iterations at σ' .

Next we provide a lower and an upper bound to $R(\sigma') - (l_i + u_i - 1)$.

Proposition 5.2. *If $\sigma' \neq \varepsilon$ then $(R(\sigma') \ominus l_i) :: u_i - u_i < R(\sigma') - (l_i + u_i - 1) \leq (R(\sigma') \ominus l_i) :: u_i$.*

Proof. Let $\sigma' \neq \varepsilon$. Then by (5.2) and by (5.4) we have:

$$\begin{aligned}
(R(\sigma') \ominus l_i) :: u_i &> 0 \\
(R(\sigma') \ominus l_i) \operatorname{div} u_i &> 0 && \text{because } u_i > 0 \\
(R(\sigma') \ominus l_i) &\geq u_i > 0 \\
R(\sigma') - l_i &> 0 && \text{because } R(\sigma') \ominus l_i > 0 .
\end{aligned}$$

So $R(\sigma') - l_i = R(\sigma') \ominus l_i$. Then we have:

$$\begin{aligned}
(R(\sigma') - l_i) \bmod u_i &\leq u_i - 1 \\
-u_i + 1 + (R(\sigma') - l_i) \bmod u_i &\leq 0 \\
(R(\sigma') - l_i) - u_i + 1 + (R(\sigma') - l_i) \bmod u_i &\leq R(\sigma') - l_i \\
(R(\sigma') - l_i) - u_i + 1 + (R(\sigma') - l_i) \bmod u_i &\leq (R(\sigma') - l_i) :: u_i + (R(\sigma') - l_i) \bmod u_i \\
(R(\sigma') - l_i) - u_i + 1 &\leq (R(\sigma') - l_i) :: u_i \\
(R(\sigma') - l_i) - u_i + 1 &\leq (R(\sigma') \ominus l_i) :: u_i \\
R(\sigma') - (l_i + u_i - 1) &\leq (R(\sigma') \ominus l_i) :: u_i .
\end{aligned}$$

Furthermore we have:

$$\begin{aligned}
0 &< 1 + (R(\sigma') - l_i) \bmod u_i \\
R(\sigma') - l_i - u_i &< (R(\sigma') - l_i - u_i) + 1 \\
&\quad + (R(\sigma') - l_i) \bmod u_i \\
(R(\sigma') - l_i) :: u_i + (R(\sigma') - l_i) \bmod u_i - u_i &< (R(\sigma') - l_i - u_i) + 1 \\
&\quad + (R(\sigma') - l_i) \bmod u_i \\
(R(\sigma') - l_i) :: u_i - u_i &< (R(\sigma') - l_i - u_i) + 1 \\
(R(\sigma') \ominus l_i) :: u_i - u_i &< (R(\sigma') - l_i - u_i) + 1 \\
(R(\sigma') \ominus l_i) :: u_i - u_i &< R(\sigma') - (l_i + u_i) - 1 .
\end{aligned}$$

□

By (5.4), there are just $(R(\sigma) \ominus l_i) : u_i$ residual iterations at σ' which are not at the same time residual iterations at σ'' . Proposition 5.2 appoints the last u_i of them as the interval which $R(\sigma') - (l_i + u_i - 1)$ ranges over.

As a candidate guard for the first for-loop, we introduce a boolean expression B such that $B[B] \rho = \text{true}$ if there are more than $l_i + u_i - 1$ residual iterations at ρ , and $B[B] \rho = \text{false}$ otherwise. The following Proposition suggests we could let $B \stackrel{\text{def}}{=} x \leq A_2 - (l_i + u_i - 1)A_3$.

Proposition 5.3. *Let $\langle \rho, S \rangle$ be a state in $\sigma' \sigma''$. Then $B[x \leq A_2 - (l_i + u_i - 1)A_3] \rho = \text{false}$ if and only if $0 \leq R_{x, A_2, A_3} \langle \rho, S \rangle \leq l_i + u_i - 1$.*

Proof. Let $x \stackrel{\text{def}}{=} A[x] \rho$, $a_2 \stackrel{\text{def}}{=} A[A_2] \rho$ and $a_3 \stackrel{\text{def}}{=} A[A_3] \rho$. We have:

$$\begin{aligned} B[x \leq A_2 - (l_i + u_i - 1)A_3] \rho = \text{false} &\iff x > a_2 - (l_i + u_i - 1)a_3 \\ &\iff x - a_2 > -(l_i + u_i - 1)a_3 \\ &\iff a_2 - x < (l_i + u_i - 1)a_3 . \end{aligned}$$

In case $a_2 \leq x$, then we have $a_2 - x \leq 0$. Because $l_i \geq 0$ and $u_i > 0$, we have $l_i + u_i - 1 \geq 0$. Hence:

$$\begin{aligned} a_2 - x < (l_i + u_i - 1)a_3 &\iff a_2 - x \leq 0 < (l_i + u_i - 1)a_3 \vee a_2 - x < 0 \leq (l_i + u_i - 1)a_3 \\ &\iff 0 \leq (l_i + u_i - 1)a_3 \\ &\iff 0 \leq l_i + u_i - 1 \quad \text{since } a_3 > 0 \text{ by hypothesis} \\ &\iff 0 \leq 0 \leq l_i + u_i - 1 \quad \text{by (2.53)} \\ &\iff 0 \leq R \langle \rho, S \rangle \leq l_i + u_i - 1 . \end{aligned}$$

In case $a_2 > x$ then, by (2.53), $R \langle \rho, S \rangle = (a_2 + a_3 - 1 - x) \text{div } a_3$. Then we have:

$$\begin{aligned} a_2 - x < (l_i + u_i - 1)a_3 &\iff 0 \leq a_2 - x < (l_i + u_i - 1)a_3 \\ &\iff 0 \leq a_2 - x + (a_3 - 1) < (l_i + u_i - 1)a_3 + (a_3 - 1) \\ &\iff 0 \leq a_2 + a_3 - x - 1 < (l_i + u_i - 1)a_3 + a_3 - 1 \\ &\iff 0 \leq (a_2 + a_3 - x - 1) \text{div } a_3 < ((l_i + u_i - 1)a_3 + a_3 - 1) \text{div } a_3 \\ &\iff 0 \leq (a_2 + a_3 - x - 1) \text{div } a_3 \leq ((l_i + u_i - 1)a_3 + a_3 - 1) \text{div } a_3 \\ &\iff 0 \leq R \langle \rho, S \rangle \leq ((l_i + u_i - 1)a_3 + a_3 - 1) \text{div } a_3 \\ &\iff 0 \leq R \langle \rho, S \rangle \leq l_i + u_i - 1 . \end{aligned}$$

□

Let us gather the results we have proved in this Section.

- If $\sigma' = \varepsilon$, then $x \leq A_2 - (l_i + u_i - 1)A_3$ is false throughout $\sigma = \sigma''$.

- If $\sigma' \neq \varepsilon$, then $\mathbf{x} \leq \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3$ is true at the beginning of σ' and gets forever false at the end of one residual iteration at σ' not at σ'' . By (5.4), the number of such iterations is $(R(\sigma) \ominus l_i) : u_i$. The one at the end of which $\mathbf{x} \leq \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3$ gets false is to be found, according to Proposition 5.2, within the last u_i .

Unrolling is supposed to make an iteration of the first for-loop out of the u_i leftmost untransformed iterations of σ' . Notice that the number of residual iterations at σ' not at σ'' is a perfect multiple of u_i . So $\mathbf{x} \leq \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3$ is going to get forever false within the last iteration of the first for-loop. We can thereby elect $\mathbf{x} \leq \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3$ as the guard of the first for-loop.

If s is a state in an actual iteration of σ' , then unrolling transforms it especially relying upon $\#(\sigma', s)$ and upon the statement included in s . By (2.50), s can be instantiated to

$$\begin{aligned} \langle \rho, \mathbf{g}: \mathbf{G} \rightarrow \mathbf{h} \rangle &= \langle \rho, \mathbf{g}: \mathbf{x} \leq \mathbf{A}_2 \rightarrow \mathbf{h} \rangle , \\ \langle \rho', \mathbf{h}: \mathbf{H} \rightarrow \mathbf{i} \rangle &= \langle \rho', \mathbf{h}: \mathbf{H} \rightarrow \mathbf{i} \rangle \end{aligned}$$

or

$$\langle \rho'', \mathbf{i}: \mathbf{I} \rightarrow \mathbf{g} \rangle = \langle \rho'', \mathbf{i}: \mathbf{x} := \mathbf{x} + \mathbf{A}_3 \rightarrow \mathbf{g} \rangle ,$$

where $\rho, \rho', \rho'' \in \mathfrak{C}[\mathbf{P}]$. We suppose $\rho' = \rho$ and $\rho'' \in \mathfrak{C}[\mathbf{H}] \rho'$, so that the concatenation of these states is an iteration of σ' . We call it our *current* iteration. By (2.53), our current iteration is preceded in σ' by a number of actual iterations amounting to:

$$\#(\sigma', \langle \rho, \mathbf{g}: \mathbf{G} \rightarrow \mathbf{h} \rangle) = \#(\sigma', \langle \rho', \mathbf{h}: \mathbf{H} \rightarrow \mathbf{i} \rangle) = \#(\sigma', \langle \rho'', \mathbf{i}: \mathbf{I} \rightarrow \mathbf{g} \rangle) - 1 .$$

Since unrolling takes the u_i leftmost untransformed iterations of σ' and turn them into an iteration of the first for-loop, we have that the iterations of the first for-loop currently amount to:

$$\#(\sigma', \langle \rho, \mathbf{g}: \mathbf{G} \rightarrow \mathbf{h} \rangle) \operatorname{div} u_i .$$

Our current iteration just makes for the next iteration of the first for-loop, which is under construction. So far, the number of actual iterations involved in this construction, including the current one, is:

$$\#(\sigma', \langle \rho, \mathbf{g}: \mathbf{G} \rightarrow \mathbf{h} \rangle) \operatorname{mod} u_i .$$

We can generalize such amount in function of the state s we are actually considering:

$$m(s) \stackrel{\text{def}}{=} \begin{cases} \#(\sigma', s) \operatorname{mod} u_i & \text{if } s = \langle \rho, \mathbf{g}: \mathbf{G} \rightarrow \mathbf{h} \rangle \vee s = \langle \rho', \mathbf{h}: \mathbf{H} \rightarrow \mathbf{i} \rangle \\ (\#(\sigma', s) - 1) \operatorname{mod} u_i & \text{if } s = \langle \rho'', \mathbf{i}: \mathbf{I} \rightarrow \mathbf{g} \rangle . \end{cases}$$

To transform s , unrolling acts separately on its statement and its environment. We write m instead of $m(s)$ whenever s is clear from the context.

If $m \langle \rho, \mathbf{g}: \mathbf{G} \rightarrow \mathbf{h} \rangle = 0$, the current iteration is the first one to take part to the construction. Only in such case, the guard of \mathbf{P} is replaced with the guard of the first for-loop:

$$g: x \leq A_2 \rightarrow h \quad \mapsto \quad \begin{cases} gmi: x \leq A_2 - (l_i + u_i - 1)A_3 \rightarrow hmi & \text{if } m = 0 \\ gmi: \text{true} \rightarrow hmi & \text{if } 0 < m \leq u_i - 1 . \end{cases}$$

If $m \langle \rho'', i: I \rightarrow g \rangle = u_i - 1$, the current iteration is the last one to take part to the construction. Only in such case, the increment of P is replaced with the increment of the first for-loop:

$$i: x := x + A_3 \rightarrow g \quad \mapsto \quad \begin{cases} imi: \text{true} \rightarrow g(m+1)i & \text{if } 0 \leq m < u_i - 1 \\ imi: x := x + u_i A_3 \rightarrow g0i & \text{if } m = u_i - 1 . \end{cases}$$

Regarding $h: H \rightarrow i$, and supposing that $m = m \langle \rho', h: H \rightarrow i \rangle$, we have:

$$h: H \rightarrow i \quad \mapsto \quad hmi: H[x+mA_3/x] \rightarrow imi \quad \text{where } 0 \leq m \leq u_i - 1 .$$

To account for the m suppressed increments and to counterbalance the modification of H, we update the environments as follows:

$$\begin{aligned} \rho &\mapsto \rho[x := x - mA[A_3]] \\ \rho' &\mapsto \rho'[x := x - mA[A_3]] \\ \rho'' &\mapsto \rho''[x := x - mA[A_3]] . \end{aligned}$$

By transforming the states of σ' one by one, unrolling turns the first $(R_{\sigma'}(\ominus)l_i) \cdot u_i$ iterations of P into the $(R_{\sigma'}(\ominus)l_i) \text{ div } u_i$ iterations of the first for-loop.

To appreciate the transformation, let us consider the for-loop P we introduced at the beginning of the previous Section, which was unrolled by splitting term 5 and unrolling factor 3, thus resulting in a new program P':

```
// Before unrolling: P           // After unrolling: P'
for (x := 1; x ≤ 11; x := x + 1) { for (x := 1; x ≤ 4; x := x + 3) {
  w := x                          w := x
}                                   w := x + 1
                                   w := x + 2
                                   }
                                   for (; x ≤ 11; x := x + 1) {
                                   w := x
                                   }
```

Assume the sequence of splitting terms and unrolling factors is modeled after the following table:

i	0	1	2	3	4	5	6	...
l	0	3	0	5	...
u	1	7	5	3	...

To attain unrolling as in the example, we let $i \stackrel{\text{def}}{=} 6$, thus obtaining $l_i = 5$ and $u_i = 3$.

We now consider a maximal trace $\sigma \in \mathbb{S}[\mathbb{P}]$ and review its transformation state by state. We list the states of σ in a column and we apply the unrolling

transformation to each state. If s is turned into s' , then we write s' just on the right of s . Thus the right column records the states of σ after the transformation.

The first state in our trace σ is:

$$\langle \mathcal{U}, \mathcal{U}, \quad f: x := 1 \quad \rightarrow \quad g \rangle \quad \langle \mathcal{U}, \mathcal{U}, \quad f: x := 1 \quad \rightarrow \quad g06 \rangle$$

Since in P there are only two variables, x and w , any state noted $\langle \rho, S \rangle$ is here written down as $\langle \rho(x), \rho(w), S \rangle$. Label g in the original state claims for the guard of P to be checked next. In the transformed state we find $G06$ instead of g . Thus in the next state we expect the guard of the first for-loop to be checked.

The guard for the first for-loop is $x \leq 11 - (l_6 + u_6 - 1) \cdot 1$, that is, $x \leq 4$. Index variable x is to be given 1 as initial value and is to be incremented by $(u_6 - 1) + 1$, that is, by 3. Since $u_6 = 3$ and

$$\begin{aligned} \#\sigma' &= \min(\#\sigma, (R(\sigma) \ominus l_6) : \cdot u_6) && \text{by (5.4)} \\ &= \min(11, (11 - 5) : \cdot 3) \\ &= \min(11, 6) = 6 \quad , \end{aligned}$$

the first for-loop is going to perform two iterations. The first of such iterations is made out of the first $u_6 = 3$ actual iterations of P :

$$\begin{array}{ll} \langle 1, \mathcal{U}, \quad g: x \leq 11 \quad \rightarrow \quad h \rangle & \langle 1, \mathcal{U}, \quad g06: x \leq 4 \quad \rightarrow \quad h06 \rangle \\ \langle 1, \mathcal{U}, \quad h: w := x \quad \rightarrow \quad i \rangle & \langle 1, \mathcal{U}, \quad h06: w := x \quad \rightarrow \quad i06 \rangle \\ \langle 1, 1, \quad i: x := x + 1 \quad \rightarrow \quad g \rangle & \langle 1, 1, \quad i06: \text{true} \quad \rightarrow \quad g16 \rangle \\ \langle 2, 1, \quad g: x \leq 11 \quad \rightarrow \quad h \rangle & \langle 1, 1, \quad g16: \text{true} \quad \rightarrow \quad h16 \rangle \\ \langle 2, 1, \quad h: w := x \quad \rightarrow \quad i \rangle & \langle 1, 1, \quad h16: w := x + 1 \quad \rightarrow \quad i16 \rangle \\ \langle 2, 2, \quad i: x := x + 1 \quad \rightarrow \quad g \rangle & \langle 1, 2, \quad i16: \text{true} \quad \rightarrow \quad g26 \rangle \\ \langle 3, 2, \quad g: x \leq 11 \quad \rightarrow \quad h \rangle & \langle 1, 2, \quad g26: \text{true} \quad \rightarrow \quad h26 \rangle \\ \langle 3, 2, \quad h: w := x \quad \rightarrow \quad i \rangle & \langle 1, 2, \quad h26: w := x + 2 \quad \rightarrow \quad i26 \rangle \\ \langle 3, 3, \quad i: x := x + 1 \quad \rightarrow \quad g \rangle & \langle 1, 3, \quad i26: x := x + 3 \quad \rightarrow \quad g06 \rangle \end{array}$$

Notice that, after the transformation, x evaluates to 1 throughout the iteration. Still, the values assumed by w are left unchanged. The index $i = 6$ was appended to every label in order to tell the statements of the first for-loop from the statements of the second for-loop (namely, the statements of P). Amount $m \in [0, u_6 - 1]$ was included, too, for two reasons:

- to provide statements with distinguishing labels, and
- to make each statement transition only to the the statement included in the next state.

As a main consequence, a nonempty path was established from $g06$ to itself that transitions through all statements just once. Along this path x is never assigned but in the last statement, whose command is the increment of the first for-loop. The increment updates the value of x and ask for the guard to be checked again, thus introducing the second iteration:

$\langle 4, 3, \quad g: x \leq 11 \quad \rightarrow \quad h \rangle$ $\langle 4, 3, \quad h: w := x \quad \rightarrow \quad i \rangle$ $\langle 4, 4, \quad i: x := x + 1 \rightarrow g \rangle$ $\langle 5, 4, \quad g: x \leq 11 \quad \rightarrow \quad h \rangle$ $\langle 5, 4, \quad h: w := x \quad \rightarrow \quad i \rangle$ $\langle 5, 5, \quad i: x := x + 1 \rightarrow g \rangle$ $\langle 6, 5, \quad g: x \leq 11 \quad \rightarrow \quad h \rangle$ $\langle 6, 5, \quad h: w := x \quad \rightarrow \quad i \rangle$ $\langle 6, 6, \quad i: x := x + 1 \rightarrow g \rangle$	$\langle 4, 3, \quad g06: x \leq 4 \quad \rightarrow \quad h06 \rangle$ $\langle 4, 3, \quad h06: w := x \quad \rightarrow \quad i06 \rangle$ $\langle 4, 4, \quad i06: \text{true} \quad \rightarrow \quad g16 \rangle$ $\langle 4, 4, \quad g16: \text{true} \quad \rightarrow \quad h16 \rangle$ $\langle 4, 4, \quad h16: w := x + 1 \rightarrow i16 \rangle$ $\langle 4, 5, \quad i16: \text{true} \quad \rightarrow \quad g26 \rangle$ $\langle 4, 5, \quad g26: \text{true} \quad \rightarrow \quad h26 \rangle$ $\langle 4, 5, \quad h26: w := x + 2 \rightarrow i26 \rangle$ $\langle 4, 6, \quad i26: x := x + 3 \rightarrow g06 \rangle$
--	---

The increment of the second iteration updates x to 7 and again requires the guard of the first for-loop to be checked again. While acknowledging $x > 4$, a transition is needed from $g06$ to g , in order to enter the second for-loop. The following state just accomplishes this task:

$$\langle 7, 6, \quad g06: x > 4 \quad \rightarrow \quad g \rangle$$

The remaining $\#\sigma'' = 5$ iterations are left unchanged:

$\langle 7, 6, \quad g: x \leq 11 \quad \rightarrow \quad h \rangle$ $\langle 7, 6, \quad h: w := x \quad \rightarrow \quad i \rangle$ $\langle 7, 7, \quad i: x := x + 1 \rightarrow g \rangle$ $\langle 8, 7, \quad g: x \leq 11 \quad \rightarrow \quad h \rangle$ $\langle 8, 7, \quad h: w := x \quad \rightarrow \quad i \rangle$ $\langle 8, 8, \quad i: x := x + 1 \rightarrow g \rangle$ $\langle 9, 8, \quad g: x \leq 11 \quad \rightarrow \quad h \rangle$ $\langle 9, 8, \quad h: w := x \quad \rightarrow \quad i \rangle$ $\langle 9, 9, \quad i: x := x + 1 \rightarrow g \rangle$ $\langle 10, 9, \quad g: x \leq 11 \quad \rightarrow \quad h \rangle$ $\langle 10, 9, \quad h: w := x \quad \rightarrow \quad i \rangle$ $\langle 10, 10, \quad i: x := x + 1 \rightarrow g \rangle$ $\langle 11, 10, \quad g: x \leq 11 \quad \rightarrow \quad h \rangle$ $\langle 11, 10, \quad h: w := x \quad \rightarrow \quad i \rangle$ $\langle 11, 11, \quad i: x := x + 1 \rightarrow g \rangle$	$\langle 7, 6, \quad g: x \leq 11 \quad \rightarrow \quad h \rangle$ $\langle 7, 6, \quad h: w := x \quad \rightarrow \quad i \rangle$ $\langle 7, 7, \quad i: x := x + 1 \rightarrow g \rangle$ $\langle 8, 7, \quad g: x \leq 11 \quad \rightarrow \quad h \rangle$ $\langle 8, 7, \quad h: w := x \quad \rightarrow \quad i \rangle$ $\langle 8, 8, \quad i: x := x + 1 \rightarrow g \rangle$ $\langle 9, 8, \quad g: x \leq 11 \quad \rightarrow \quad h \rangle$ $\langle 9, 8, \quad h: w := x \quad \rightarrow \quad i \rangle$ $\langle 9, 9, \quad i: x := x + 1 \rightarrow g \rangle$ $\langle 10, 9, \quad g: x \leq 11 \quad \rightarrow \quad h \rangle$ $\langle 10, 9, \quad h: w := x \quad \rightarrow \quad i \rangle$ $\langle 10, 10, \quad i: x := x + 1 \rightarrow g \rangle$ $\langle 11, 10, \quad g: x \leq 11 \quad \rightarrow \quad h \rangle$ $\langle 11, 10, \quad h: w := x \quad \rightarrow \quad i \rangle$ $\langle 11, 11, \quad i: x := x + 1 \rightarrow g \rangle$
--	--

The final state is left unchanged, too:

$$\langle 12, 11, \quad g: x > 11 \quad \rightarrow \quad j \rangle \quad \langle 12, 11, \quad g: x > 11 \quad \rightarrow \quad j \rangle$$

It is easy to verify that the sequence of states in the right column is a trace; let us call it μ . By collecting all the statements along the states of μ , we obtain a new program P'' :

```
// After unrolling: P''
for (x := 1; x ≤ 4; x := x + 3) {
  w := x
  true
  true
  w := x + 1
  true
  true
  w := x + 2
}
for (; x ≤ 11; x := x + 1) {
  w := x
}
```

Therefore $\mu \in \mathcal{S}[P'']$. Notice that P'' just reduces to P' after we get rid of the **true** statements.

Although $\sigma \in \mathcal{S}[P]$ and $\mu \in \mathcal{S}[P'']$ are about the same length and they agree on the values they provide to w , program P'' is much longer than P . In order not to be deceived by the loop headers and **true** statements, let us consider only the assignments on w . Then we find only one of such assignments in P ,

$$h: w := x \rightarrow i ,$$

whereas we find four of them in P'' :

$$\begin{aligned} h05: w := x \rightarrow i05 \\ h15: w := x + 1 \rightarrow i15 \\ h25: w := x + 2 \rightarrow i25 \\ h: w := x \rightarrow i . \end{aligned}$$

Yet, the semantics of w is preserved because each assignment in P'' carries out just a fraction of the task performed by the only assignment of P .

Indeed, $h: w := x \rightarrow i \in P$ implements the body in all the eleven iterations of σ , thus covering the entire iteration space I ,

$$I = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 \} . \quad (5.7)$$

In particular, it can be thought as trivially partitioning I into

$$\{ \{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 \} \} , \quad (5.8)$$

where the existence of a unique class entails that, as far as we are not concerned with environments, all iterations become indistinguishable, thereby collapsing or *folding* to the same set of statements.

On the other side, $h05: w := x \rightarrow i05 \in P''$ implements in μ the body of only the first and the fourth iteration of σ , thus covering $\{1, 4\}$. Through similar arguments, we eventually obtain the results we report here below:

$$\begin{array}{ll}
 h05: w := x \rightarrow i05 & \{1, 4\} \\
 h15: w := x + 1 \rightarrow i15 & \{2, 5\} \\
 h25: w := x + 2 \rightarrow i25 & \{3, 6\} \\
 h: w := x \rightarrow i & \{7, 8, 9, 10, 11\} .
 \end{array}$$

If we collect the four sets on the right, we get again a partition of I :

$$\{\{1, 4\}, \{2, 5\}, \{3, 6\}, \{7, 8, 9, 10, 11\}\} . \quad (5.9)$$

Since (5.9) is a refinement of (5.8), it provides a less radical folding scheme where iterations, rather than eventually collapsing all together once environments are discarded, collapse if and only if they are in the same class.

The class an iteration is ascribed to originates from the sort of transformation unrolling let such iteration undergo. Our example above broadly suggests that unrolling transforms a state $\langle \rho, \mathbf{S} \rangle$ by shifting some information from ρ to \mathbf{S} . In the first place, this implies that information is dragged out from environments; consider for instance how the value of x evolves throughout the first three iterations: while in σ it marks the progress of the for-loop, in μ it becomes insensitive to the progress and remains constant. In the second place, the informational shift is directed to statements, thus implying that labels or commands are modified to record the incoming information. As the recording is not required to be complete, in general the informational shift is lossy. This explains why iterations, and in particular their statements, usually do not differentiate perfectly, that is, why each iteration does not form its own class in the partitioning of I .

In our example, for any state $\langle \rho, \mathbf{S} \rangle$ in σ' , unrolling shifts

$$(\rho(x) - 1) \bmod u_i \quad (5.10)$$

from ρ to \mathbf{S} . Because (5.10) is modeled after the modulo function $\lambda n. n \bmod u_i$, because $u_i = 3$ and because $\#\sigma' = 6$, in (5.9) we have $\{1, 4\}$, $\{2, 5\}$ and $\{3, 6\}$, instead of $\{1\}$, $\{2\}$, \dots $\{6\}$. As regards the states of σ'' , unrolling performs a void shift, thus leaving these states unchanged. Unrolling thereby attempts nothing to prevent the last $\#\sigma' = 5$ iterations from collapsing all together. As a result, in (5.9) we also have the last class $\{7, 8, 9, 10, 11\}$. All in all, in (5.9) there are $u_6 + 1 = 3 + 1 = 4$ iterations.

In general, an iteration space I is partitioned into $u_i + 1$ classes if $(R(\sigma) \ominus l_i) : u_i$ and $i > 0$; otherwise it is partitioned into one class only. Later, in Section 5.8, we take a more formal approach to the partitions induced by loop unrolling.

Regardless of the approach, the more the classes in the partition, the larger the amount of information that has been shifted from environments to statements. Hence loop unrolling can be seen as a natural incomplete information extractor, which syntactically exhibits iterations by displaying them as a sequence of programs. We take advantage of loop unrolling in the next Chapter to design a watermarking technique for embedding an authorship mark or a fingerprint within a for-loop.

5.5 Syntactic Transformer

The algorithm u that implements loop unrolling takes i , l and u as parameters and a for-loop P as argument. If $i = 0$, it returns P , that is, it performs neither unrolling nor splitting. If $i > 0$ then P is split by l_i in two for-loops: the first one is unrolled by u_i , whereas the second one is unrolled by the trivial unrolling factor $u_0 = 1$.

$$\mathcal{L}[u[P]] \stackrel{\text{def}}{=} \mathcal{L}[P] \quad (5.11)$$

$$u[P] \stackrel{\text{def}}{=} \bigcup \{ u[S] \mid S \in P \} \quad (5.12)$$

$$u[\ell: C \rightarrow g] \stackrel{\text{def}}{=} \{ \ell: C \rightarrow g \mid i = 0 \} \quad \text{where } \ell \neq i \quad (5.13)$$

$$\cup \{ \ell: C \rightarrow g0i' \mid i' > 0 \wedge i' = i \}$$

$$u[g: x > A_2 \rightarrow j] \stackrel{\text{def}}{=} \{ g: x > A_2 \rightarrow j \}$$

$$\cup \{ g0i': x > A_2 - (l_{i'} + u_{i'} - 1)A_3 \rightarrow g \mid i' > 0 \wedge i' = i \}$$

$$(5.14)$$

$$u[g: x \leq A_2 \rightarrow h] \stackrel{\text{def}}{=} \{ g: x \leq A_2 \rightarrow h \}$$

$$\cup \{ g0i': x \leq A_2 - (l_{i'} + u_{i'} - 1)A_3 \rightarrow h0i' \mid i' > 0 \wedge i' = i \}$$

$$\cup \{ gmi: \text{true} \rightarrow hmi \mid m \in (0, u_i - 1] \}$$

$$(5.15)$$

$$u[h: H \rightarrow i] \stackrel{\text{def}}{=} \{ h: H \rightarrow i \} \quad (5.16)$$

$$\cup \{ h0i': H \rightarrow i0i' \mid i' > 0 \wedge i' = i \}$$

$$\cup \{ hmi: H[x+mA_3/x] \rightarrow imi \mid m \in (0, u_i - 1] \}$$

$$u[i: x := x + A_3 \rightarrow g] \stackrel{\text{def}}{=} \{ i: x := x + A_3 \rightarrow g \}$$

$$\cup \{ i(u_{i'} - 1)i': x := x + u_{i'}A_3 \rightarrow g0i' \mid i' > 0 \wedge i' = i \}$$

$$\cup \{ imi: \text{true} \rightarrow g(m+1)i \mid m \in [0, u_i - 1] \}$$

$$(5.17)$$

Notice that, instead of an explicit definition for transforming $f: x := A_1 \rightarrow g$, we have introduced (5.13). This more general definition allows us to unroll for-loops which lack a statement for the initialization of x , as in the case of the second for-loop resulting from the unrolling of P .

5.6 Semantic Transformer

The semantic transformer u for loop unrolling is recursively defined through state transformers. Each state transformer takes in input a state s and two values m and i , assuming that $m = m(s)$ and that i points to the splitting term and unrolling factor that are intended for s .

For the state $\langle \rho, \ell: C \rightarrow g \rangle$ that comes before the iterations of P we have:

$$\begin{aligned}
u(\langle \rho, \ell: \mathbf{C} \rightarrow \mathbf{g} \rangle, m, i) &= \text{where } \ell \neq i \\
&\text{if } i = 0 \text{ then} \\
&\quad \langle \rho, \ell: \mathbf{C} \rightarrow \mathbf{g} \rangle
\end{aligned} \tag{5.18}$$

$$\begin{aligned}
&\text{if } i > 0 \text{ then} \\
&\quad \langle \rho, \ell: \mathbf{C} \rightarrow \mathbf{g}0i \rangle
\end{aligned} \tag{5.19}$$

The transformation of $\langle \rho, \mathbf{g}: \mathbf{x} \leq \mathbf{A}_2 \rightarrow \mathbf{h} \rangle$ relies on m . In case $m = 0$, boolean expression $\mathbf{x} \leq \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3$ is checked to determine whether the current iteration of \mathbf{P} makes for the next iteration of the first for-loop. If not, the additional state transitioning from the first to the second for-loop is inserted:

$$\begin{aligned}
u(\langle \rho, \mathbf{g}: \mathbf{x} \leq \mathbf{A}_2 \rightarrow \mathbf{h} \rangle, m, i) &= \\
&\text{if } (i = 0 \wedge m = 0) \vee (i > 0 \wedge \mathbf{B}[\mathbf{x} > \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3] \rho \wedge \rho(\Delta) = 1) \text{ then} \\
&\quad \langle \rho[\Delta := \mathbf{U}], \mathbf{g}: \mathbf{x} \leq \mathbf{A}_2 \rightarrow \mathbf{h} \rangle
\end{aligned} \tag{5.20}$$

$$\begin{aligned}
&\text{if } i > 0 \wedge m = 0 \wedge \mathbf{B}[\mathbf{x} \leq \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3] \rho \text{ then} \\
&\quad \langle \rho, \mathbf{g}0i: \mathbf{x} \leq \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3 \rightarrow \mathbf{h}0i \rangle
\end{aligned} \tag{5.21}$$

$$\begin{aligned}
&\text{if } i > 0 \wedge m = 0 \wedge \mathbf{B}[\mathbf{x} > \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3] \rho \wedge \rho(\Delta) = \mathbf{U} \text{ then} \\
&\quad \langle \rho, \mathbf{g}0i: \mathbf{x} > \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3 \rightarrow \mathbf{g} \rangle
\end{aligned} \tag{5.22}$$

$$\begin{aligned}
&\text{if } 0 < m \leq u_i - 1 \text{ then} \\
&\quad \langle \rho[\mathbf{x} := \mathbf{A}[\mathbf{x} - m\mathbf{A}_3] \rho], \mathbf{g}mi: \mathbf{true} \rightarrow \mathbf{h}mi \rangle
\end{aligned} \tag{5.23}$$

Whenever $\langle \rho, \mathbf{g}: \mathbf{x} > \mathbf{A}_2 \rightarrow \mathbf{h} \rangle$ is met, surely there are no actual iterations of \mathbf{P} left over. If the transition from the first to the second for-loop has not occurred yet, the additional state is inserted. In any case, $\langle \rho, \mathbf{g}: \mathbf{x} > \mathbf{A}_2 \rightarrow \mathbf{h} \rangle$ is appended last.

$$\begin{aligned}
u(\langle \rho, \mathbf{g}: \mathbf{x} > \mathbf{A}_2 \rightarrow \mathbf{h} \rangle, 0, i) &= \\
&\text{if } i = 0 \vee (i > 0 \wedge \rho(\Delta) = 1) \text{ then} \\
&\quad \langle \rho[\Delta := \mathbf{U}], \mathbf{g}: \mathbf{x} > \mathbf{A}_2 \rightarrow \mathbf{j} \rangle
\end{aligned} \tag{5.24}$$

$$\begin{aligned}
&\text{if } i > 0 \wedge \rho(\Delta) = \mathbf{U} \text{ then} \\
&\quad \langle \rho, \mathbf{g}0i: \mathbf{x} > \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3 \rightarrow \mathbf{g} \rangle
\end{aligned} \tag{5.25}$$

States $\langle \rho, \mathbf{h}: \mathbf{H} \rightarrow \mathbf{i} \rangle$ and $\langle \rho, \mathbf{i}: \mathbf{x} := \mathbf{x} + \mathbf{A}_3 \rightarrow \mathbf{g} \rangle$ are transformed according to the principles discussed in Section 5.4:

$$\begin{aligned}
u(\langle \rho, \mathbf{h}: \mathbf{H} \rightarrow \mathbf{i} \rangle, m, i) &= \\
&\text{if } i = 0 \text{ then} \\
&\quad \langle \rho, \mathbf{h}: \mathbf{H} \rightarrow \mathbf{i} \rangle
\end{aligned} \tag{5.26}$$

$$\begin{aligned}
&\text{if } i > 0 \text{ then} \\
&\quad \langle \rho, \mathbf{h}mi: \mathbf{H}[\mathbf{x} + m\mathbf{A}_3/\mathbf{x}] \rightarrow \mathbf{i}mi \rangle
\end{aligned} \tag{5.27}$$

$$\begin{aligned}
u(\langle \rho, \mathbf{i}: \mathbf{x} := \mathbf{x} + \mathbf{A}_3 \rightarrow \mathbf{g} \rangle, m, i) &= \\
&\text{if } i = 0 \text{ then} \\
&\quad \langle \rho, \mathbf{i}: \mathbf{x} := \mathbf{x} + \mathbf{A}_3 \rightarrow \mathbf{g} \rangle
\end{aligned} \tag{5.28}$$

if $i > 0 \wedge m = u_i - 1$ then

$$\langle \rho[x := A[x - (u_i - 1)A_3]] \rho \rangle, i(u_i - 1)i: x := x + u_i A_3 \rightarrow g0i \rangle \quad (5.29)$$

if $i > 0 \wedge 0 \leq m < u_i - 1$ then

$$\langle \rho[x := A[x - mA_3]] \rho \rangle, imi: \text{true} \rightarrow g(m + 1)i \rangle \quad (5.30)$$

5.7 Local Commutation Condition

The auxiliary function u' includes the following state transformers, each one meant to perfectly undo a state transformer in u :

$$u' \langle \rho', \ell: \mathbf{C} \rightarrow g0i \rangle = \quad \text{where } \ell \neq i \\ \langle \rho', \ell: \mathbf{C} \rightarrow g \rangle, 0, i \rangle \quad (5.31)$$

$$u' \langle \rho', \ell: \mathbf{C} \rightarrow g \rangle = \quad \text{where } \ell \neq i \\ \langle \rho', \ell: \mathbf{C} \rightarrow g \rangle, 0, 0 \rangle \quad (5.32)$$

$$u' \langle \rho', g0i: x \leq A_2 - (l_i + u_i - 1)A_3 \rightarrow h0i \rangle = \\ \langle \rho', g: x \leq A_2 \rightarrow h \rangle, 0, i \rangle \quad (5.33)$$

$$u' \langle \rho', gmi: \text{true} \rightarrow hmi \rangle = \\ \langle \rho'[x := A[x + mA_3]] \rho' \rangle, g: x \leq A_2 \rightarrow h \rangle, m, i \rangle \quad (5.34)$$

$$u' \langle \rho', g: x \leq A_2 \rightarrow h \rangle = \\ \langle \rho', g: x \leq A_2 \rightarrow h \rangle, 0, 0 \rangle \quad (5.35)$$

$$u' \langle \rho', hmi: \mathbf{H} \rightarrow imi \rangle = \\ \langle \rho'[x := A[x + mA_3]] \rho' \rangle, h: \mathbf{H}^{[x - mA_3/x]} \rightarrow i \rangle, m, i \rangle \quad (5.36)$$

$$u' \langle \rho', h: \mathbf{H} \rightarrow i \rangle = \\ \langle \rho', h: \mathbf{H} \rightarrow i \rangle, 0, 0 \rangle \quad (5.37)$$

$$u' \langle \rho', imi: \text{true} \rightarrow g(m + 1)i \rangle = \\ \langle \rho'[x := A[x + mA_3]] \rho' \rangle, i: x := x + A_3 \rightarrow g \rangle, m + 1, i \rangle \quad (5.38)$$

$$u' \langle \rho', imi: x := x + u_i A_3 \rightarrow gmi \rangle = \\ \langle \rho'[x := A[x + (u_i - 1)A_3]] \rho' \rangle, i: x := x + A_3 \rightarrow g \rangle, m, i \rangle \quad (5.39)$$

$$u' \langle \rho', i: x := x + A_3 \rightarrow g \rangle = \\ \langle \rho', i: x := x + A_3 \rightarrow g \rangle, 0, 0 \rangle \quad (5.40)$$

Notice that each transformer takes a state $s = \langle \rho, \mathbf{S} \rangle$ and returns a state s' and two values m and i . Such values are just those found in $\text{suc}[\mathbf{S}]$ and they are meant to be used to transform any state possibly coming after s' .

The auxiliary function also includes a special state transformer for undoing the additional state providing the transition from the first to the second for-loop:

$$u' \langle \rho', g0i: x > A_2 - (l_i + u_i - 1)A_3 \rightarrow g \rangle = \\ \langle \rho'[\Delta := 1], i: \text{true} \rightarrow g \rangle, 0, 0 \rangle \quad (5.41)$$

Observe that, whichever is the value of $i > 0$, this state transformer set i to 0, thus enforcing trivial unrolling on the second for-loop.

We can now formally prove that \mathfrak{u} and u are respectively the syntactic and the semantic counterpart of loop unrolling.

Theorem 5.4. *Let P be a for-loop. Then $\forall n \in \mathbb{N}. F_{\mathfrak{u}}^n[[P]]\emptyset = F^n[[\mathfrak{u}[[P]]]]\emptyset$.*

Proof. We let $\mathfrak{t} = \mathfrak{u}$, $\mathfrak{t} = u$ and $\mathfrak{t}' = u'$, and just prove (2.81) and (2.82).

First we prove (2.81). Let $i = 0$.

$$\begin{aligned}
u(\mathcal{J}[[P]]) &= u(\{ \langle \rho, \mathbf{S} \rangle \mid \rho \in \mathfrak{C}[[P]] \wedge \mathbf{S} \in P \wedge \text{lab}[[\mathbf{S}]] \in \mathfrak{L}[[P]] \}) && \text{by (2.45)} \\
&= \{ u(\langle \rho, \mathbf{S} \rangle, 0, 0) \mid \rho \in \mathfrak{C}[[P]] \wedge \mathbf{S} \in P \wedge \text{lab}[[\mathbf{S}]] \in \mathfrak{L}[[P]] \} && \text{by (2.68)} \\
&= \{ u(\langle \rho, \mathbf{f}: \mathbf{y} := \mathbf{A}_1 \rightarrow \mathbf{g} \rangle, 0, 0) \mid \rho \in \mathfrak{C}[[P]] \} \\
&= \{ \langle \rho, \mathbf{f}: \mathbf{x} := \mathbf{A}_1 \rightarrow \mathbf{g} \rangle \mid \rho \in \mathfrak{C}[[P]] \} && \text{by (5.18)} \\
&= \{ \langle \rho, \mathbf{f}: \mathbf{x} := \mathbf{A}_1 \rightarrow \mathbf{g} \rangle \mid \rho \in \mathfrak{C}[[\mathfrak{u}[[P]]]] \} && \text{by definition of } u \\
&= \{ \langle \rho, \mathbf{S} \rangle \mid \rho \in \mathfrak{C}[[P]] \wedge \mathbf{S} \in u[[P]] \wedge \text{lab}[[\mathbf{S}]] \in \mathfrak{L}[[\mathfrak{u}[[P]]]] \} \\
&= \mathcal{J}[[\mathfrak{u}[[P]]]] && \text{by (2.45)}
\end{aligned}$$

Let $i > 0$.

$$\begin{aligned}
u(\mathcal{J}[[P]]) &= u(\{ \langle \rho, \mathbf{S} \rangle \mid \rho \in \mathfrak{C}[[P]] \wedge \mathbf{S} \in P \wedge \text{lab}[[\mathbf{S}]] \in \mathfrak{L}[[P]] \}) && \text{by (2.45)} \\
&= \{ u(\langle \rho, \mathbf{S} \rangle, 0, i) \mid \rho \in \mathfrak{C}[[P]] \wedge \mathbf{S} \in P \wedge \text{lab}[[\mathbf{S}]] \in \mathfrak{L}[[P]] \} && \text{by (2.68)} \\
&= \{ u(\langle \rho, \mathbf{f}: \mathbf{y} := \mathbf{A}_1 \rightarrow \mathbf{g} \rangle, 0, i) \mid \rho \in \mathfrak{C}[[P]] \} \\
&= \{ \langle \rho, \mathbf{f}: \mathbf{x} := \mathbf{A}_1 \rightarrow \mathbf{g}0i \rangle \mid \rho \in \mathfrak{C}[[P]] \} && \text{by (5.19)} \\
&= \{ \langle \rho, \mathbf{f}: \mathbf{x} := \mathbf{A}_1 \rightarrow \mathbf{g}0i \rangle \mid \rho \in \mathfrak{C}[[\mathfrak{u}[[P]]]] \} && \text{by definition of } u \\
&= \{ \langle \rho, \mathbf{S} \rangle \mid \rho \in \mathfrak{C}[[P]] \wedge \mathbf{S} \in u[[P]] \wedge \text{lab}[[\mathbf{S}]] \in \mathfrak{L}[[\mathfrak{u}[[P]]]] \} \\
&= \mathcal{J}[[\mathfrak{u}[[P]]]] && \text{by (2.45)}
\end{aligned}$$

We now prove (2.82). Let $i = 0$.

- Let η be such that $\neg \eta = \langle \rho', \mathbf{f}: \mathbf{x} := \mathbf{A}_1 \rightarrow \mathbf{g} \rangle$.
On the one hand,
let $\rho'' = \rho'[\mathbf{x} := \mathbf{A}[[\mathbf{A}_1]]\rho']$.
Then $R_{\mathfrak{u}}(\eta) = \{ \langle \rho'', \mathbf{g}: \mathbf{x} \leq \mathbf{A}_2 \rightarrow \mathbf{h} \rangle, \langle \rho'', \mathbf{g}: \mathbf{x} > \mathbf{A}_2 \rightarrow \mathbf{j} \rangle \}$.
On the other hand,
we have $u'(\neg \eta) =$ (5.32)
 $= \langle \rho', \mathbf{f}: \mathbf{x} := \mathbf{A}_1 \rightarrow \mathbf{g} \rangle, 0, 0$

Let $\rho = \rho'[\mathbf{x} := \mathbf{A}[[\mathbf{A}_1]]\rho']$.

If $\mathbf{B}[[\mathbf{x} > \mathbf{A}_2]]\rho$ then

$$\begin{aligned}
u(s, 0, 0) &= u(\langle \rho, \mathbf{g}: \mathbf{x} > \mathbf{A}_2 \rightarrow \mathbf{j} \rangle, 0, 0) \\
&= (5.24) \\
&= \langle \rho, \mathbf{g}: \mathbf{x} > \mathbf{A}_2 \rightarrow \mathbf{j} \rangle \\
&= \langle \rho'[\mathbf{x} := \mathbf{A}[[\mathbf{A}_1]]\rho'], \mathbf{g}: \mathbf{x} > \mathbf{A}_2 \rightarrow \mathbf{j} \rangle \\
&= \langle \rho'', \mathbf{g}: \mathbf{x} > \mathbf{A}_2 \rightarrow \mathbf{j} \rangle .
\end{aligned}$$

If $\mathbb{B}[\mathbf{x} \leq \mathbf{A}_2] \rho$ then

$$\begin{aligned} u(s, 0, 0) &= u(\langle \rho, g: \mathbf{x} \leq \mathbf{A}_2 \rightarrow h \rangle, 0, 0) \\ &= (5.20) \\ &= \langle \rho, g: \mathbf{x} \leq \mathbf{A}_2 \rightarrow h \rangle \\ &= \langle \rho'[\mathbf{x} := \mathbf{A}[\mathbf{A}_1] \rho'], g: \mathbf{x} \leq \mathbf{A}_2 \rightarrow h \rangle \\ &= \langle \rho'', g: \mathbf{x} \leq \mathbf{A}_2 \rightarrow h \rangle . \end{aligned}$$

- Let η be such that $\neg \eta = \langle \rho', g: \mathbf{x} \leq \mathbf{A}_2 \rightarrow h \rangle$.

On the one hand,

let $\rho'' = \rho'$.

Then $R_{\mathbf{u}}(\eta) = \{ \langle \rho'', h: \mathbf{H} \rightarrow i \rangle \}$.

On the other hand,

we have $u'(\neg \eta) = (5.35)$

Let $\rho = \rho'$.

Then

$$\begin{aligned} u(s, 0, 0) &= u(\langle \rho, h: \mathbf{H} \rightarrow i \rangle, 0, 0) \\ &= (5.26) \\ &= \langle \rho', h: \mathbf{H} \rightarrow i \rangle \\ &= \langle \rho'', h: \mathbf{H} \rightarrow i \rangle . \end{aligned}$$

- Let η be such that $\neg \eta = \langle \rho', h: \mathbf{H} \rightarrow i \rangle$.

On the one hand,

let $\rho'' \in \mathbb{C}[\mathbf{H}] \rho'$.

Then $R_{\mathbf{u}}(\eta) = \{ \langle \rho'', i: \mathbf{x} := \mathbf{x} + \mathbf{A}_3 \rightarrow g \rangle \}$.

On the other hand,

we have $u'(\neg \eta) = (5.37)$

Let $\rho \in \mathbb{C}[\mathbf{H}] \rho'$.

Then

$$\begin{aligned} u(s, 0, 0) &= u(\langle \rho, i: \mathbf{x} := \mathbf{x} + \mathbf{A}_3 \rightarrow g \rangle, 0, 0) \\ &= (5.28) \\ &= \langle \rho, i: \mathbf{x} := \mathbf{x} + \mathbf{A}_3 \rightarrow g \rangle \\ &= \langle \rho'', i: \mathbf{x} := \mathbf{x} + \mathbf{A}_3 \rightarrow g \rangle . \end{aligned}$$

- Let η be such that $\neg \eta = \langle \rho', i: \mathbf{x} := \mathbf{x} + \mathbf{A}_3 \rightarrow g \rangle$.

On the one hand,

let $\rho'' = \rho'[\mathbf{x} := \mathbf{A}[\mathbf{x} + \mathbf{A}_3] \rho']$.

Then $R_{\mathbf{u}}(\eta) = \{ \langle \rho'', g: \mathbf{x} \leq \mathbf{A}_2 \rightarrow h \rangle, \langle \rho'', g: \mathbf{x} > \mathbf{A}_2 \rightarrow j \rangle \}$.

On the other hand,

we have $u'(\neg \eta) = (5.40)$

Let $\rho = \rho'[\mathbf{x} := \mathbf{A}[\mathbf{x} + \mathbf{A}_3] \rho']$.

If $\mathbb{B}[\mathbf{x} > \mathbf{A}_2] \rho$ then

$$\begin{aligned} u(s, 0, 0) &= u(\langle \rho, g: \mathbf{x} > \mathbf{A}_2 \rightarrow j \rangle, 0, 0) \\ &= (5.24) \\ &= \langle \rho, g: \mathbf{x} > \mathbf{A}_2 \rightarrow j \rangle \\ &= \langle \rho'[\mathbf{x} := \mathbf{A}[\mathbf{x} + \mathbf{A}_3] \rho'], g: \mathbf{x} > \mathbf{A}_2 \rightarrow j \rangle \\ &= \langle \rho'', g: \mathbf{x} > \mathbf{A}_2 \rightarrow j \rangle . \end{aligned}$$

If $\mathbf{B}[\mathbf{x} \leq \mathbf{A}_2] \rho$ then

$$\begin{aligned} u(s, 0, 0) &= u(\langle \rho, g: \mathbf{x} \leq \mathbf{A}_2 \rightarrow \mathbf{h} \rangle, 0, 0) \\ &= (5.20) \\ &= \langle \rho, g: \mathbf{x} \leq \mathbf{A}_2 \rightarrow \mathbf{h} \rangle \\ &= \langle \rho'[\mathbf{x} := \mathbf{A}[\mathbf{x} + \mathbf{A}_3]] \rho' \rangle, g: \mathbf{x} \leq \mathbf{A}_2 \rightarrow \mathbf{h} \rangle \\ &= \langle \rho'', g: \mathbf{x} \leq \mathbf{A}_2 \rightarrow \mathbf{h} \rangle \end{aligned}$$

Let $i > 0$.

- Let η be such that $\neg \eta = \langle \rho', f: \mathbf{x} := \mathbf{A}_1 \rightarrow g \rangle$.

On the one hand,

$$\text{let } \rho'' = \rho'[\mathbf{x} := \mathbf{A}[\mathbf{A}_1]] \rho'.$$

$$\begin{aligned} \text{Then } R_{\text{u}}(\eta) &= \{ \langle \rho'', g0i: \mathbf{x} \leq \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3 \rightarrow \mathbf{h}0i \rangle, \\ &\quad \langle \rho'', g0i: \mathbf{x} > \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3 \rightarrow g \rangle \}. \end{aligned}$$

On the other hand,

$$\text{we have } u'(\neg \eta) = (5.31)$$

$$\text{Let } \rho = \rho'[\mathbf{x} := \mathbf{A}[\mathbf{A}_1]] \rho'.$$

If $\mathbf{B}[\mathbf{x} > \mathbf{A}_2] \rho$ then

$$\begin{aligned} u(s, 0, i) &= u(\langle \rho, g: \mathbf{x} > \mathbf{A}_2 \rightarrow \mathbf{j} \rangle, 0, i) \\ &= (5.25) \\ &= \langle \rho, g0i: \mathbf{x} > \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3 \rightarrow g \rangle \\ &= \langle \rho'[\mathbf{x} := \mathbf{A}[\mathbf{A}_1]] \rho' \rangle, g0i: \mathbf{x} > \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3 \rightarrow g \rangle \\ &= \langle \rho'', g0i: \mathbf{x} > \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3 \rightarrow g \rangle. \end{aligned}$$

If $\mathbf{B}[\mathbf{x} \leq \mathbf{A}_2] \rho \wedge \mathbf{B}[\mathbf{x} > \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3] \rho$ then

$$\begin{aligned} u(s, 0, i) &= u(\langle \rho, g: \mathbf{x} \leq \mathbf{A}_2 \rightarrow \mathbf{h} \rangle, 0, i) \\ &= (5.22) \\ &= \langle \rho, g0i: \mathbf{x} > \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3 \rightarrow g \rangle \\ &= \langle \rho'[\mathbf{x} := \mathbf{A}[\mathbf{A}_1]] \rho' \rangle, g0i: \mathbf{x} > \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3 \rightarrow g \rangle \\ &= \langle \rho'', g0i: \mathbf{x} > \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3 \rightarrow g \rangle. \end{aligned}$$

If $\mathbf{B}[\mathbf{x} \leq \mathbf{A}_2] \rho \wedge \mathbf{B}[\mathbf{x} \leq \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3] \rho$ then

$$\begin{aligned} u(s, 0, i) &= u(\langle \rho, g: \mathbf{x} \leq \mathbf{A}_2 \rightarrow \mathbf{h} \rangle, 0, i) \\ &= (5.21) \\ &= \langle \rho, g0i: \mathbf{x} \leq \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3 \rightarrow \mathbf{h}0i \rangle \\ &= \langle \rho'[\mathbf{x} := \mathbf{A}[\mathbf{A}_1]] \rho' \rangle, g0i: \mathbf{x} \leq \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3 \rightarrow \mathbf{h}0i \rangle \\ &= \langle \rho'', g0i: \mathbf{x} \leq \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3 \rightarrow \mathbf{h}0i \rangle \end{aligned}$$

- Let η be such that $\neg \eta = \langle \rho', g0i: \mathbf{x} \leq \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3 \rightarrow \mathbf{h}0i \rangle$.

On the one hand,

$$\text{let } \rho'' = \rho'.$$

$$\text{Then } R_{\text{u}}(\eta) = \{ \langle \rho'', \mathbf{h}0i: \mathbf{H} \rightarrow \mathbf{i}0i \rangle \}.$$

On the other hand,

$$\text{we have } u'(\neg \eta) = (5.33)$$

$$\text{Let } \rho = \rho'.$$

Then

$$\begin{aligned}
u(s, 0, i) &= u(\langle \rho, h: \mathbf{H} \rightarrow i \rangle, 0, i) \\
&= (5.27) \\
&= \langle \rho, h0i: \mathbf{H} \rightarrow i0i \rangle \\
&= \langle \rho', h0i: \mathbf{H} \rightarrow i0i \rangle \\
&= \langle \rho'', h0i: \mathbf{H} \rightarrow i0i \rangle .
\end{aligned}$$

- Let $0 \leq m \leq u_i - 1$. Let η be such that $\neg\eta = \langle \rho', hmi: \mathbf{H} \rightarrow imi \rangle$.

– Let $0 \leq m < u_i - 1$.

On the one hand,

let $\rho'' \in \mathbf{C}[\mathbf{H}] \rho'$.

Then $R_u(\eta) = \{ \langle \rho'', imi: \mathbf{true} \rightarrow g(m+1)i \rangle \}$.

On the other hand,

we have $u'(\neg\eta) = (5.36)$

$$= \langle \langle \rho' [x := \mathbf{A}[x + mA_3]] \rho' \rangle, h: \mathbf{H} [x - mA_3/x] \rightarrow i \rangle, m, i \rangle .$$

Let $\rho \in \mathbf{C}[\mathbf{H} [x - mA_3/x]] \rho' [x := \mathbf{A}[x + mA_3]] \rho'$.

Then

$$\begin{aligned}
u(s, m, i) &= u(\langle \rho, i: x := x - mA_3 \rightarrow g \rangle, m, i) \\
&= (5.30)
\end{aligned}$$

$$= \langle \rho [x := \mathbf{A}[x - mA_3]] \rho', imi: \mathbf{true} \rightarrow g(m+1)i \rangle .$$

By Lemma 2.4, we have $\rho [x := \mathbf{A}[x - mA_3]] \rho' \in \mathbf{C}[\mathbf{H}] \rho'$.

So we can identify $\rho [x := \mathbf{A}[x - mA_3]] \rho'$ with ρ'' .

– Let $m = u_i - 1$.

On the one hand,

let $\rho'' \in \mathbf{C}[\mathbf{H}] \rho'$.

Then $R_u(\eta) = \{ \langle \rho'', i(u_i - 1)i: x := x + u_i A_3 \rightarrow g0i \rangle \}$.

On the other hand,

we have $u'(\neg\eta) = (5.36)$

$$= \langle \langle \rho' [x := \mathbf{A}[x + mA_3]] \rho' \rangle, h: \mathbf{H} [x - mA_3/x] \rightarrow i \rangle, m, i \rangle .$$

Let $\rho \in \mathbf{C}[\mathbf{H} [x - mA_3/x]] \rho' [x := \mathbf{A}[x + mA_3]] \rho'$.

Then

$$\begin{aligned}
u(s, m, i) &= u(\langle \rho, i: x := x - mA_3 \rightarrow g \rangle, m, i) \\
&= (5.29)
\end{aligned}$$

$$= \langle \rho [x := \mathbf{A}[x - mA_3]] \rho', i(u_i - 1)i: x := x + u_i A_3 \rightarrow g0i \rangle .$$

By Lemma 2.4, we have $\rho [x := \mathbf{A}[x - mA_3]] \rho' \in \mathbf{C}[\mathbf{H}] \rho'$.

So we can identify $\rho [x := \mathbf{A}[x - mA_3]] \rho'$ with ρ'' .

- Let $0 \leq m < u_i - 1$. Let η be such that $\neg\eta = \langle \rho', imi: \mathbf{true} \rightarrow g(m+1)i \rangle$.

On the one hand,

let $\rho'' = \rho'$.

Then $R_u(\eta) = \{ \langle \rho'', g(m+1)i: \mathbf{true} \rightarrow h(m+1)i \rangle \}$.

On the other hand,

we have $u'(\neg\eta) = (5.38)$

Let $\rho = \rho'$.

Then

$$s = \langle \rho' [x := A[x + mA_3]] \rho [x := A[x + A_3]] \rho' [x := A[x + mA_3]] \rho' \rangle, g: x \leq A_2 \rightarrow h \rangle$$

$$= \langle \rho' [x := A[x + (m+1)A_3]] \rho' \rangle, g: x \leq A_2 \rightarrow h \rangle$$

and

$$u(s, m, i) = u(\langle \rho' [x := A[x + (m+1)A_3]] \rho' \rangle, g: x \leq A_2 \rightarrow h \rangle, m, i)$$

$$= (5.23)$$

$$= \langle \rho'', g(m+1)i: \mathbf{true} \rightarrow h(m+1)i \rangle .$$

- Let η be such that $\neg\eta = \langle \rho', g(m+1)i: \mathbf{true} \rightarrow h(m+1)i \rangle$.

On the one hand,

let $\rho'' = \rho'$.

Then $R_u(\eta) = \{ \langle \rho'', h(m+1)i: H[x+(m+1)A_3/x] \rightarrow i(m+1)i \rangle \}$.

On the other hand,

we have $u'(\neg\eta) = (5.34)$

Let $\rho = \rho'$.

Then

$$u(s, m+1, i) = u(\langle \rho' [x := A[x + (m+1)A_3]] \rho' \rangle, h: H \rightarrow i \rangle, m+1, i)$$

$$= (5.27)$$

$$= \langle \rho, h(m+1)i: H[x+(m+1)A_3/x] \rightarrow i(m+1)i \rangle$$

$$= \langle \rho', h(m+1)i: H[x+(m+1)A_3/x] \rightarrow i(m+1)i \rangle$$

$$= \langle \rho'', h(m+1)i: H[x+(m+1)A_3/x] \rightarrow i(m+1)i \rangle .$$

- Let $m = u_i - 1$. Let η be such that $\neg\eta = \langle \rho', i(u_i - 1)i: x := x + u_i A_3 \rightarrow g0i \rangle$.

On the one hand,

let $\rho'' = \rho' [x := A[x + u_i A_3]] \rho'$.

Then $R_u(\eta) = \{ \langle \rho'', g0i: x \leq A_2 - (l_i + u_i - 1)A_3 \rightarrow h0i \rangle, \langle \rho'', g0i: x > A_2 - (l_i + u_i - 1)A_3 \rightarrow g \rangle \}$.

On the other hand,

we have $u'(\neg\eta) = (5.31)$

Let $\rho = \rho' [x := A[x + u_i A_3]] \rho'$.

If $B[x > A_2] \rho$ then

$$u(s, 0, i) = u(\langle \rho, g: x > A_2 \rightarrow j \rangle, 0, i)$$

$$= (5.25)$$

$$= \langle \rho, g0i: x > A_2 - (l_i + u_i - 1)A_3 \rightarrow g \rangle$$

$$= \langle \rho' [x := A[x + u_i A_3]] \rho' \rangle, g0i: x > A_2 - (l_i + u_i - 1)A_3 \rightarrow g \rangle$$

$$= \langle \rho'', g0i: x > A_2 - (l_i + u_i - 1)A_3 \rightarrow g \rangle .$$

If $B[x \leq A_2] \rho \wedge B[x > A_2 - (l_i + u_i - 1)A_3] \rho$ then

$$u(s, 0, i) = u(\langle \rho, g: x \leq A_2 \rightarrow h \rangle, 0, i)$$

$$= (5.22)$$

$$= \langle \rho, g0i: x > A_2 - (l_i + u_i - 1)A_3 \rightarrow g \rangle$$

$$= \langle \rho' [x := A[x + u_i A_3]] \rho' \rangle, g0i: x > A_2 - (l_i + u_i - 1)A_3 \rightarrow g \rangle$$

$$= \langle \rho'', g0i: x > A_2 - (l_i + u_i - 1)A_3 \rightarrow g \rangle .$$

If $\mathbb{B}[\mathbf{x} \leq \mathbf{A}_2] \rho \wedge \mathbb{B}[\mathbf{x} \leq \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3] \rho$ then

$$\begin{aligned}
\mathbf{u}(s, 0, i) &= \mathbf{u}(\langle \rho, \mathbf{g}: \mathbf{x} \leq \mathbf{A}_2 \rightarrow \mathbf{h} \rangle, 0, i) \\
&= (5.21) \\
&= \langle \rho, \mathbf{g}0i: \mathbf{x} \leq \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3 \rightarrow \mathbf{h}0i \rangle \\
&= \langle \rho'[\mathbf{x} := \mathbf{A}[\mathbf{x} + u_i\mathbf{A}_3]\rho'], \mathbf{g}0i: \mathbf{x} \leq \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3 \rightarrow \mathbf{h}0i \rangle \\
&= \langle \rho'', \mathbf{g}0i: \mathbf{x} \leq \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3 \rightarrow \mathbf{h}0i \rangle .
\end{aligned}$$

- Let $m = 0$.

Let η be such that $\neg\eta = \langle \rho', \mathbf{g}0i: \mathbf{x} > \mathbf{A}_2 - (l_i + u_i - 1)\mathbf{A}_3 \rightarrow \mathbf{g} \rangle$.

On the one hand,

let $\rho'' = \rho'$.

Then $R_{\mathbf{u}}(\eta) = \{ \langle \rho'', \mathbf{g}: \mathbf{x} > \mathbf{A}_2 \rightarrow \mathbf{j} \rangle, \langle \rho'', \mathbf{g}: \mathbf{x} \leq \mathbf{A}_2 \rightarrow \mathbf{h} \rangle \}$.

On the other hand,

we have $\mathbf{u}'(\neg\eta) = (5.41)$

Let $\rho = \rho'$.

If $\mathbb{B}[\mathbf{x} > \mathbf{A}_2] \rho$ then

$$\begin{aligned}
\mathbf{u}(s, 0, i) &= \mathbf{u}(\langle \rho, \mathbf{h}: \mathbf{x} > \mathbf{A}_2 \rightarrow \mathbf{j} \rangle, 0, 0) \\
&= (5.24) \\
&= \langle \rho, \mathbf{h}: \mathbf{x} > \mathbf{A}_2 \rightarrow \mathbf{j} \rangle \\
&= \langle \rho', \mathbf{h}: \mathbf{x} > \mathbf{A}_2 \rightarrow \mathbf{j} \rangle \\
&= \langle \rho'', \mathbf{h}: \mathbf{x} > \mathbf{A}_2 \rightarrow \mathbf{j} \rangle .
\end{aligned}$$

If $\mathbb{B}[\mathbf{x} \leq \mathbf{A}_2] \rho$ then

$$\begin{aligned}
\mathbf{u}(s, 0, i) &= \mathbf{u}(\langle \rho, \mathbf{h}: \mathbf{x} \leq \mathbf{A}_2 \rightarrow \mathbf{h} \rangle, 0, 0) \\
&= (5.20) \\
&= \langle \rho, \mathbf{h}: \mathbf{x} \leq \mathbf{A}_2 \rightarrow \mathbf{h} \rangle \\
&= \langle \rho', \mathbf{h}: \mathbf{x} \leq \mathbf{A}_2 \rightarrow \mathbf{h} \rangle \\
&= \langle \rho'', \mathbf{h}: \mathbf{x} \leq \mathbf{A}_2 \rightarrow \mathbf{h} \rangle .
\end{aligned}$$

□

5.8 Folding

By the end of Section 5.4 we observed that, by shifting some information from environments to statements, loop unrolling induces a partition on the set of iterations: the more refined is the partition, the larger is the amount of information shifted. In that Section we reasoned informally on an example, abstracting iterations to index vectors and considering partitions on the iteration space. In the present Section we take advantage of the semantic transformer \mathbf{u} to recast our observations in a more formal framework. In particular, we consider the set of actual iterations and investigate how the choice of i affects the partitioning of that set.

Let $\sigma \in \mathbf{d}_{\mathbf{g}}(\mathbb{S}[\mathbb{P}])$. Then the set of all the iterations that are found in σ is $\mathbf{e}_{\mathbf{g}}(\sigma)$. We define on $\mathbf{e}_{\mathbf{g}}(\sigma)$ an equivalence relation \sim_i such that, given two iterations $\theta, \theta' \in \mathbf{e}_{\mathbf{g}}(\sigma)$,

$$\theta \sim_i \theta' \text{ if and only if } \mathbb{P}(\mathbf{u}(\theta)) = \mathbb{P}(\mathbf{u}(\theta')) . \quad (5.42)$$

Thus θ and θ' are in relation if and only if, after having been transformed by splitting term l_i and unrolling factor u_i , they still have the same statements. Since the definition of \sim_i is modeled after an equality, it is easy to prove that \sim_i is an equivalence relation.

The partition induced by u on $\mathbf{e}_g(\sigma)$ is $\mathbf{e}_g(\sigma)/\sim_i$. We claim that every class of iterations in $\mathbf{e}_g(\sigma)/\sim_i$ ultimately reduces to one of the following sets:

$$\mathbf{e}_{g,\leq,i}(\sigma) \stackrel{\text{def}}{=} \{ \theta \in \mathbf{e}_g(\sigma) \mid R(\theta) \leq R(\sigma) - (R(\sigma) \ominus l_i) \cdot u_i \} \quad (5.43)$$

$$\begin{aligned} \mathbf{e}_{g,m,i}(\sigma) \stackrel{\text{def}}{=} \{ \theta \in \mathbf{e}_g(\sigma) \mid R(\theta) > R(\sigma) - (R(\sigma) \ominus l_i) \cdot u_i \\ \wedge \#(\sigma, \vdash \theta) \bmod u_i = m \} \end{aligned} \quad (5.44)$$

Recall that, by (5.2), (5.3) and (5.4), unrolling divides σ into $\sigma', \sigma'' \in \mathbf{d}_g(\mathcal{S}[\mathbb{P}])$ such that $\sigma'\sigma'' = \sigma$ and $\#\sigma' = \min(\#\sigma, (R(\sigma) \ominus l_i) \cdot u_i)$, and transform σ' by splitting term l_i and unrolling factor u_i , whereas it leaves σ'' unchanged. Then:

- (5.43) collects the iterations of σ'' ;
- (5.44) collects every iteration θ of σ' such that $\#(\sigma, \vdash \theta) \bmod u_i = m$, where m is a parameter supposedly ranging from 0 to $u_i - 1$.

This is proved in the following

Theorem 5.5. $\mathbf{e}_g(\sigma)/\sim_i = \{ \mathbf{e}_{g,\leq,i}(\sigma) \} \cup \{ \mathbf{e}_{g,m,i}(\sigma) \mid 0 \leq m \leq u_i - 1 \}$.

Proof. Let $\theta \in \mathbf{e}_g(\sigma)$. Then, of course, $[\theta]_{\sim_i} \in \mathbf{e}_g(\sigma)/\sim_i$. Moreover, by definition of quotient set, for all $\theta' \in \mathbf{e}_g(\sigma)$ we have that $\theta' \in [\theta]_{\sim_i}$ if and only if $\theta' \sim_i \theta$.

Let us focus on the latter statement. By (5.42), $\theta' \sim_i \theta$ if and only if $\mathbb{p}(u(\theta)) = \mathbb{p}(u(\theta'))$. By definition of u and \mathbb{p} , this equality holds if and only if θ is transformed by splitting term m' and unrolling factor i' , and θ is transformed by splitting term m'' and unrolling factor i'' , and $m' = m''$ and $i' = i''$ and $\mathbb{p}(\theta) = \mathbb{p}(\theta')$. We now consider two cases.

Suppose $R(\theta') \leq R(\sigma) - (R(\sigma) \ominus l_i) \cdot u_i$. At the same time, this entails $i' = i'' = i = 0$. Ultimately, it is equivalent to $\theta' \in \mathbf{e}_{g,\leq,i}(\sigma)$. Thus we have $\theta' \in [\theta]_{\sim_i}$ if and only if $\theta' \in \mathbf{e}_{g,\leq,i}(\sigma)$, whence $[\theta]_{\sim_i} = \mathbf{e}_{g,\leq,i}(\sigma)$.

Suppose $R(\theta') > R(\sigma) - (R(\sigma) \ominus l_i) \cdot u_i$. At the same time, this entails $i' = i'' = i \geq 0$ and $m' = \#(\sigma, \vdash \theta) \bmod u_{i'} = \#(\sigma, \vdash \theta') \bmod u_{i''} = m''$. This in turn is equivalent to $\theta' \in \mathbf{e}_{g,\#(\sigma,\vdash\theta) \bmod u_{i'},i''}(\sigma)$. Because $i' = i'' = i \geq 0$ and $0 \leq \#(\sigma, \vdash \theta) \bmod u_{i'} \leq u_{i'} - 1$, it is ultimately equivalent to $\theta' \in \mathbf{e}_{g,m,i''}(\sigma)$, where $0 \leq m \leq u_i - 1$. Thus we have $\theta' \in [\theta]_{\sim_i}$ if and only if $\theta' \in \mathbf{e}_{g,m,i''}(\sigma)$, whence $[\theta]_{\sim_i} = \mathbf{e}_{g,m,i''}(\sigma)$. \square

We now discuss the special case when $i = 0$. There is no unrolling at all when $i = 0$, as in such case every state transformer in u acts as the identical function. Moreover we have:

$$\begin{aligned} \#\sigma &\leq (R(\sigma) \ominus l_0) \cdot u_0 \\ &= (R(\sigma) \ominus 0) \cdot 1 \\ &= R(\sigma) . \end{aligned}$$

As a consequence, $\#\sigma' = \#\sigma$ by (5.4) and $\#\sigma'' = 0$ by (5.5). We thereby expect $\mathbf{e}_{g,\leq,0}(\sigma)$ to be the empty set and $\mathbf{e}_{g,0,0}(\sigma)$ to just be $\mathbf{e}_g(\sigma)$.

Proposition 5.6. $e_{g,\leq,0}(\sigma) = \emptyset$.

Proof. We have:

$$\begin{aligned}
e_{g,\leq,0}(\sigma) &= \{ \theta \in e_g(\sigma) \mid R(\theta) \leq R(\sigma) - (R(\sigma) \ominus l_0) : \cdot u_0 \} && \text{by (5.43)} \\
&= \{ \theta \in e_g(\sigma) \mid R(\theta) \leq R(\sigma) - (R(\sigma) \ominus 0) : \cdot 1 \} \\
&= \{ \theta \in e_g(\sigma) \mid R(\theta) \leq R(\sigma) - R(\sigma) \} \\
&= \{ \theta \in e_g(\sigma) \mid R(\theta) \leq 0 \} \\
&= \emptyset .
\end{aligned}$$

□

Proposition 5.7. $e_{g,0,0}(\sigma) = e_g(\sigma)$.

Proof. We have:

$$\begin{aligned}
e_{g,0,0}(\sigma) &= \{ \theta \in e_g(\sigma) \mid R(\theta) > R(\sigma) - (R(\sigma) \ominus l_0) : \cdot u_0 \\
&\quad \wedge \#(\sigma, \vdash \theta) \bmod u_0 = 0 \} && \text{by (5.44)} \\
&= \{ \theta \in e_g(\sigma) \mid R(\theta) > R(\sigma) - (R(\sigma) \ominus 0) : \cdot 1 \\
&\quad \wedge \#(\sigma, \vdash \theta) \bmod 1 = 0 \} \\
&= \{ \theta \in e_g(\sigma) \mid R(\theta) > R(\sigma) - R(\sigma) \wedge \text{true} \} \\
&= \{ \theta \in e_g(\sigma) \mid R(\theta) > 0 \} \\
&= e_g(\sigma) .
\end{aligned}$$

□

Thus, whenever $i = 0$, we expect $e_g(\sigma)/\sim_i$ to have only one class.

Theorem 5.8. $e_g(\sigma)/\sim_0 = \{ e_g(\sigma) \}$.

Proof. We have:

$$\begin{aligned}
e_g(\sigma)/\sim_0 &= \{ e_{g,\leq,i}(\sigma) \} \cup \{ e_{g,m,0}(\sigma) \mid 0 \leq m \leq u_0 - 1 \} && \text{by Theorem 5.5} \\
&= \{ e_{g,\leq,i}(\sigma) \} \cup \{ e_{g,m,0}(\sigma) \mid 0 \leq m \leq 1 - 1 \} \\
&= \{ e_{g,\leq,i}(\sigma) \} \cup \{ e_{g,0,0}(\sigma) \} \\
&= \{ e_{g,\leq,i}(\sigma) \} \cup \{ e_g(\sigma) \} && \text{by Proposition 5.7} \\
&= \emptyset \cup \{ e_g(\sigma) \} && \text{by Proposition 5.6} \\
&= \{ e_g(\sigma) \} .
\end{aligned}$$

□

As a main consequence of Theorem 5.8, \sim_i is always a refinement of \sim_0 . This implies that in $e_g(\sigma)/\sim_i$ there are as many classes as in $e_g(\sigma)/\sim_0$, or even more. This in turn entails that whenever unrolling is not trivial, it is effective in making iterations differentiate, even if only partially. Hence the formal argument provided by Theorem 5.8 allows us to consider loop unrolling an information extractor. We take advantage of loop unrolling in the next Chapter to design a watermarking technique for embedding an authorship mark or a fingerprint within a for-loop.

Hiding Software Watermarks in Looping Constructs

We take advantage of loop unrolling, which in the previous Chapter was shown to be a sort of information extractor, to introduce a novel watermarking technique. Such proposal is a joint work [38] with Mila Dalla Preda and Roberto Giacobazzi. We provide a public marking scheme with the passive embedding of an invisible authorship mark, which we call the signature for short. We exploit unrolling as the engine of both the embedding and the extraction algorithm. Our watermarking scheme qualitatively complies with all the claims we made in Chapter 1 but the last one.

6.1 Steganographic Approach to Software Watermarking

Most of the existing watermarking techniques target a program feature which can assume *many* configurations, but hide the watermark in just *one* of them. Consider, for example, the watermarking technique [98] that modifies the register allocation: although there are many allocations that suit the program data flow, only one is designated to be the signature and thereby used in the marked program. The same idea applies in [42], where a distinctive permutation of basic blocks is selected among the many possible ones. Both [42] and [98] are static techniques, because they affect only the layout of programs. Notice that a statically watermarked program exhibits only the watermark configuration and rules out all the other ones: this may help, rather than hinder, attackers, not to mention the ease of subverting layout while preserving functionality.

Dynamic watermarking techniques exploit configurations that programs assume at runtime, thus allowing many candidate configurations to coexist in the same program. For instance, the path-based technique [16] targets the runtime branching behavior of programs: a program executes different paths on different inputs, but only the special input provides the path that outlines the signature. Likewise, the threading technique [85] yields multi-thread programs in which different configurations arise from how race conditions between threads are resolved; once again, a special input provides the configuration associated to the signature. Such dynamic techniques are not trivial to thwart: both branching and threading behaviors are tied to functionality, hence their distortion may result in a distortion of functionality.

The coexistence of watermarked and unwatermarked configurations within the same program also characterizes the abstract watermarking technique [34]. Here a configuration is a parametric abstract domain saying whether a watermark variable w , which is assigned twice and computed through the Horner scheme, is constant or not. Observe that the main point is not the use of the Horner scheme but the fact that w is constant only in the domain parametrized by a key, while other domains consider w to have stochastic behavior [34].

Contrary to [34], loops are the basic block of the dynamic watermarking technique we propose here. We know from the previous Chapters that a loop P is a programming construct in which a program H , called the loop body, is executed repeatedly, thus giving rise to sequences of iterations. In the proposed technique, *any* subsequence of such sequences is a candidate watermarking configuration. We consider *loop-based watermarks*, that is, watermarks that are computed by means of a looping construct Q . Our aim is to turn Q into a new set of statements Q' , called the *stegomark* [34], such that there is only *one* subsequence of P in which Q' yields the signature – otherwise it does not produce significant results. Once we have Q' , we watermark P by replacing H with $H \cup Q'$.

For the sake of example, consider the following program P

```

y := 0
for (x := z; x ≤ 50; x := x + 1) {
  y := y + x
}

```

which performs 50 iterations if z evaluates to 1. Let the *Beast Software Corporation* have signature 666, computed in two iterations by the following program Q :

```

w := 53
for (x := 17; x ≤ 18; x := x + 1) {
  w := w + x · x
}

```

To watermark P , *Beast* moves both $w := 53$ and $w := w + x \cdot x$ in the body H of the original loop, thus obtaining program

```

y := 0
for (x := z; x ≤ 50; x := x + 1) {
  y := y + x
  w := 53
  w := w + x · x
}

```

Notice that w is initialized to 53 at each iteration. To implement our watermarking technique, we want w to receive the correct initialization only at a specific iteration, which we call the *promoter*. We thereby look for some arithmetic expression that has value 53 only at the promoter. We notice that, when z evaluates to 1 and x evaluates to 17, then y evaluates to 136. In such case, consequently, arithmetic expression $y - 83$ evaluates to 53. So, once we replace $w := 53$ with $w := y - 83$, the

iteration represented by index vector 17 becomes the promoter of our watermark, provided that z evaluates to 1. Obviously, the replacement yields:

```

y := 0
for (x := z; x ≤ 50; x := x + 1) {
  y := y + x
  w := y - 83
  w := w + x · x
}

```

At extraction time, we first need to expose the promoter and, as the watermark loop subsumes two iterations, the iterations that comes after the promoter. We apply loop unrolling to split the previous for-loop into two loops, so that the first one collects the first sixteen iterations:

```

y := 0
for (x := z; x ≤ 16; x := x + 1) {
  y := y + x
  w := y - 83
  w := w + x · x
}
for (; x ≤ 50; x := x + 1) {
  y := y + x
  w := y - 83
  w := w + x · x
}

```

Hence the promoter is the first iteration of the second for-loop. We apply unrolling to such loop in order to expose the promoter and the following iteration. We obtain:

```

y := 0
for (x := z; x ≤ 16; x := x + 1) {
  y := y + x
  w := y - 83
  w := w + x · x
}
for (; x ≤ 18; x := x + 2) {
  y := y + x
  w := y - 83
  w := w + x · x
  true
  true
  y := y + (x + 1)
  w := y - 83
  w := w + (x + 1) · (x + 1)
}
for (; x ≤ 50; x := x + 1) {
  y := y + x
  w := y - 83
  w := w + x · x
}

```

Because we are interested in the second for-loop, we can get rid of the third for-loop for sure. In the second for-loop there are two statements $w := y - 83$ that initialize w . The correct initialization is provided only by the statement in the promoter, that is, the first one. Thus we must expunge the other. As a consequence, statement $y := y + (x + 1)$ becomes useless, and can be removed as well. We also wipe out the `true` statements. In the first for-loop, both the assignments on w are useless, because w is reinitialized within the second for-loop. Thus we get rid of them, too. Notice that the statements we threw away are the same that are discarded by a *backward slicing* [123] with *criterion* $w := w + (x + 1) \cdot (x + 1)$. We thereby get the following program:

```

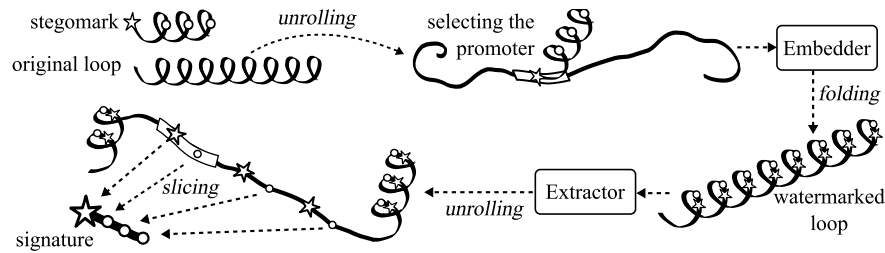
y := 0
for (x := z; x ≤ 16; x := x + 1) {
  y := y + x
}
for (; x ≤ 18; x := x + 2) {
  y := y + x
  w := y - 83
  w := w + x · x
  w := w + (x + 1) · (x + 1)
}

```

It is easy to verify that, given a maximal trace $\langle \rho, S \rangle \sigma \langle \rho', S' \rangle$ in the semantics of this program, if $\rho(z) = 1$ then $\rho'(w) = 666$, that is, w evaluates to the signature of the *Beast Software Corporation*.

6.2 Exposing Iterations for the sake of Watermarking

In the watermarking technique we exemplified above, we took advantage of loop unrolling both at embedding and extraction time to expose the iterations of the target for-loop. This is made apparent in the following diagram, which graphically sketches both the embedding and the extraction phase:



Let the longer spring stand for an original loop P performing $N > 0$ iterations on input ρ . Let the starred spring stand for a loop Q that performs $n \leq N$ iterations to compute a loop-based watermark.

At embedding time, we must derive from Q a new program Q' that gets the correct initialization only in the promoter of P , provided that P is run on input ρ . First, we unroll P entirely to syntactically expose its iterations. Among the first $N - n$ iterations, we designate a promoter. Then we establish a dependence that binds Q to the promoter, thus obtaining Q' . Finally, we insert Q' in the body of P , thus obtaining P' . Unrolling P' is no longer of help for an attacker to determine the promoter, because Q' appears in every iteration.

At extraction time we use loop unrolling to expose in P' only the subsequence of n iterations which starts from the promoter. Next, we apply backward slicing using as criterion the last statement of Q' included in the last iteration of the subsequence. After that, we get the semantics of the slice assuming ρ as initial environment. We collect the final values of the variables of Q' in a set S of candidate signatures. Finally, we identify the signature among the elements of S .

Analogy between Watermark Extraction and DNA Transcription

The extraction phase, as depicted in the lower part of our Figure above, might remind several researchers of DNA stretches that unwind during the transcription step in protein biosynthesis within cell nuclei [5]. Transcription is the process through which the information coded in a small DNA stretch is extracted and recoded in a complementary RNA molecule. In particular:

1. DNA unwinds and produces a small stretch containing an entry point for transcription, called the promoter;
2. the complementary RNA molecule gets all the information coded between the promoter and the closest standard termination point;
3. splicing is applied to the RNA molecule to discard some stretches of its that carry irrelevant information.

We can draw an analogy that likens the DNA molecule to the watermarked loop P' . In particular:

- the stretch produced by the unwound DNA molecule corresponds to the sequence of iterations exposed by the unrolled loop;
- the promoter of the transcription corresponds to the promoter of the watermark;
- the termination point of transcription corresponds to feeding a splitting term to the unrolling transformation;
- the splicing of the RNA molecule corresponds to the backward slicing of the unrolled loop.

The possibility of inserting proprietary information in the DNA molecule has been initially explored in [104]. Although our watermarking scheme cannot be applied to DNA, it could provide intriguing insights for further research.

6.3 Encoding the Signature

While commenting our Figure above, we provided a general watermarking scheme that is suitable for the embedding of *any* kind of loop-based watermark Q . In the specific watermarking technique we describe in the present and in the following Sections, we consider loop-based watermarks in which the signature stems from the evaluation of a polynomial through an Horner scheme. We got the idea of using polynomials and Horner schemes from a paper by Cousot and Cousot [34].

Suppose $n, a, b \in \mathbb{Z}$ and $n > 0$. Let P_n be a n -degree polynomial such that:

$$P_n(x) \stackrel{\text{def}}{=} ax^n + b \sum_{j=0}^{n-1} x^j \quad (6.1)$$

We assume that a signature is a natural number \mathfrak{s} such that, given $\xi \in \mathbb{Z}$, we have:

$$\mathfrak{s} = P_n(\xi) . \quad (6.2)$$

The program Q that implements the evaluation of $P_n(x)$ at $x = \xi$ through the Horner scheme is:

$$\begin{aligned} & \mathfrak{w} := a \\ & \text{for } (\mathfrak{x} := 0; \mathfrak{x} \leq n - 1; \mathfrak{x} := \mathfrak{x} + 1) \{ \\ & \quad \mathfrak{w} := \xi \cdot \mathfrak{w} + b \\ & \} \end{aligned} \quad (6.3)$$

Since this program computes the signature through a for-loop, we are dealing with a loop-based watermark. By (6.2) and (6.1) we get

$$\mathfrak{s} = a\xi^n + b \sum_{j=0}^{n-1} \xi^j ,$$

whence we can easily derive a :

$$a \stackrel{\text{def}}{=} \frac{\mathfrak{s}}{\xi^n} - \frac{b}{\xi^n} \sum_{j=0}^{n-1} \xi^j . \quad (6.4)$$

In order to ensure that $a \in \mathbb{Z}$, we require:

- ξ^n to be a divisor of \mathfrak{s} ;
- b to be a nonzero multiple of ξ^n , namely:

$$b \neq 0, \quad b = \xi^{n+n'} z \quad \text{where } n' \in \mathbb{N} \text{ and } z \in \mathbb{Z} . \quad (6.5)$$

Let us consider an example. We assume the signature is $\mathfrak{s} = 120\,736 = 2^5 \cdot 7^3 \cdot 11$. We choose $\xi \stackrel{\text{def}}{=} 2 \cdot 7 = 14$. We let $n' \stackrel{\text{def}}{=} 11$ and $z \stackrel{\text{def}}{=} 15$. Then $b = 14^{3+11} \cdot 15 = 245\,760$. By (6.4), we also get:

$$a \stackrel{\text{def}}{=} \frac{120\,736}{14^3} - \frac{245\,760}{14^3} \sum_{j=0}^2 14^j = 199\,948 .$$

All in all, we derived the following polynomial:

$$P_3(x) = -199\,948x^3 + 245\,760x^2 + 245\,760x + 245\,760 .$$

The program Q that computes this polynomial at $x = \xi = 14$ is:

```
w := -199948
for (x := 0; x ≤ 2; x := x + 1) {
  w := 14 · w + 245760
}
```

Notice that the number of iterations of the loop is just the degree $n = 3$ of the polynomial. We do not use this snippet entirely. In fact, the watermark consists only of the two assignments on w .

6.4 Tracing Appropriate Locations for the Watermark

To watermark a program, we first detect all for-loops therein with integer index variable. Any of such loops is appropriate for the embedding of the watermark, so we can feel free to choose randomly.

6.5 Embedding

Consider, without loss of generality, the following minimal for-loop P:

```
for (x := A1; x ≤ A2; x := x + A3) {
  y := A
}
```

The algorithm (see Figure 6.1) for embedding our loop-based watermark in P is a program transformer that takes in input P and outputs a new program P' which includes both the watermark and the functionality of P . The algorithm is parametrized by the parameters of the watermark, namely w , n , ξ , a and b . We expect w to be a fresh variable in P . There is also an additional parameter, namely the initial environment ρ on which we base the promoter.

$\text{EMBED}_{w,n,\xi,a,b,\rho}[\mathbb{P}]$
Check whether there are at least n iterations in P .
 1. Let $N = \min \{ R(\sigma) \mid \exists \mathbf{S}. \langle \rho, \mathbf{S} \rangle \sigma \in \mathcal{S}[\mathbb{P}] \}$.
 2. If $N < n$ then give up!
Designate the promoter.
 3. Let $i = \text{rnd}(\mathbb{N} \setminus \{0\})$.
 4. Let $l_i = 0$.
 5. Let $u_i = \max \{ R(\sigma) \mid \exists \mathbf{S}. \langle \rho, \mathbf{S} \rangle \sigma \in \mathcal{S}[\mathbb{P}] \}$.
 6. Unroll P by splitting term l_i and unrolling factor u_i ; get P' and P'' .
 7. Let $m = \text{rnd}([0, N - n])$.
Check whether y denotes a unique value in the promoter.
 8. Slice P' backward by criterion $\text{hmi}: y := A \rightarrow \text{imi}$; get P''' .
 9. Let $Y = \{ \rho'(y) \mid \exists \sigma. \exists \mathbf{S}. \sigma \in \mathcal{S}[P'''] \wedge \vdash \sigma = \langle \rho, \mathbf{S} \rangle \wedge \vdash \sigma = \langle \rho', \text{hmi}: y := A \rightarrow \text{imi} \rangle \}$.
 10. If $\neg(\exists y. Y = \{y\})$, then give up!
Perform embedding.
 11. Let $r = \text{rnd}(\mathbb{Z} \setminus \{0\})$.
 12. Insert $w := \xi \cdot w + b$ at label 'i' in P ; get P'''' .
 13. Insert $w := r \cdot y + (a - ry)$ at label 'i0' in P'''' ; get P''''' .
 14. Return P''''' as the watermarked program and (ρ, m, n) as the key.

Fig. 6.1. Embedding algorithm

The algorithm checks that every trace in $\mathcal{S}[\mathbb{P}]$ with initial environment ρ denotes at least n residual iterations. This is a critical step, because some ill-conceived programs have infinite traces that must be checked one by one. If the check is passed successfully, N records the minimum amount of residual iterations.

Next, an index value i is chosen randomly among the nonzero natural numbers. Splitting factor l_i is assigned 0, whereas unrolling factor u_i is assigned the maximal amount of residual iterations. This of course is another critical step in the algorithm.

The unrolling of P by l_i and u_i results in two for-loops P' and P'' . The body of P' syntactically exhibits the longest sequence of iterations P can perform. The promoter is designated among the first $N - n$ iterations, by choosing randomly a value $m \in [0, N - n]$. Because by design $N \geq n > 0$, this choice ensures that the m -th iteration, that is, the promoter, is always followed by at least $n - 1$ iterations. Hence, at extraction time, we can successfully display n iterations from the promoter on, thus retrieving the signature without fail.

The dependence between the watermark and the promoter is established in terms of a flow dependence between w and y . In particular, the initial value that w gets in Q is made dependent on the value assumed by y in the promoter. For the sake of reliability, once ρ is fixed, the value of y in the promoter must be unique.

The body of the promoter consists in statement $hmi: y := A[x+mA_3/x] \rightarrow imi$. The algorithm uses this statement as a criterion to slice P' backward, thus getting P''' . Then it collects in set Y the values that y assumes at the end of each maximal trace of P''' with initial environment ρ . We have that y assumes a unique value y if and only if Y is a singleton $\{y\}$. As the computation of Y involves traces, it represents another critical step in the algorithm. If this check is passed successfully, too, the algorithm builds stegomark Q' and inlays it in original for-loop P .

We build the stegomark from the two assignments appearing in Q :

$$\begin{aligned} w &:= a \\ w &:= \xi \cdot w + b \end{aligned}$$

We just replace a with $a + (y - y)r$, where $r \stackrel{\text{def}}{=} \text{rnd}(\mathbb{Z} \setminus \{0\})$ is randomly chosen among the nonzero integer numbers. While a always evaluate to a , the evaluation of $a + (y - y)r$ depends on y , and reduces to a if and only if y evaluates to y . This is guaranteed to happen only in the promoter. If y denotes stochastic behavior, that is, it changes its value from one iteration to another, the knowledge of m becomes essential at extraction time to detect the promoter and get the correct initialization of w . This improves both the reliability and the stealth of the watermark. To not let a be explicitly recorded in the syntax of the watermarked program, we can use $r \cdot y + (a - ry)$ instead of $a + (y - y)r$. Hence stegomark Q' can be defined as:

$$\begin{aligned} w &:= r \cdot y + (a - ry) \\ w &:= \xi \cdot w + b \end{aligned} \tag{6.6}$$

Notice ξ and b are not obfuscated and, by (6.5), b is known to be a multiple of ξ^n . If n' was fixed in (6.5), e.g. $n' \stackrel{\text{def}}{=} 0$, then n could be easily retrieved – since ξ divides b precisely $n + n'$ times. This would be unpleasant because n is going to be part of the secret watermarking key. By letting n' be selected randomly in (6.5), what it is known is that $0 < n \leq n + n'$: the greater is n' , the larger is the margin of uncertainty of n .

If Q' is the stegomark, then the algorithm inserts the second assignment at ‘i’ in P and then the first assignment at ‘i0’. The resulting watermarked loop is:

```

for (x := A1; x ≤ A2; x := x + A3) {
  y := A
  w := r · y + (a - ry)
  w := ξ · w + b
}

```

Because w is fresh in P , the inserted assignments do not perturb the semantics noticeably.

Along with the watermarked for-loop, the embedding algorithm returns a tuple including ρ , m and n . Such tuple is a secret key that we exploit at extraction time to correctly parametrize the extraction algorithm.

6.6 Locating the Watermark

The key does not mention the for-loop that hosts the watermark. As a consequence, whenever we try to extract our signature from a program, we must run the extraction algorithm on every for-loop of that program.

6.7 Extracting and Decoding the Watermark

Suppose that, in the overall process of extracting our signature \mathfrak{s} from a program, we are processing a for-loop P . The key instruments the extraction algorithm to get from P a set S of presumed signatures, which we are expected to retrieve \mathfrak{s} among. Notice that we are not assuming that P is watermarked. So S might result in the empty set.

EXTRACT $_{\rho,m,n}[[P]]$

Check whether there are at least n iterations in P .

1. Let $N = \min \{ R(\sigma) \mid \exists \mathfrak{S}. \langle \rho, \mathfrak{S} \rangle \sigma \in \mathcal{S}[[P]] \}$.
 2. If $N < n$, then give up!
- Display the m -th iteration of P and the following $n - 1$ iterations.*
3. Let $l_1 = N - m$.
 4. Let $u_1 = 1$.
 5. Unroll P by splitting term l_1 and unrolling factor u_1 ; get P' and P'' .
 6. Let $l_2 = N - m - n$.
 7. Let $u_2 = n$.
 8. Unroll P'' by splitting term l_2 and unrolling factor u_2 ; get P''' and P'''' .
- Retrieve candidate stegomarks and obtain candidate signatures.*
9. Let $S = \emptyset$.
 10. For each $\ell 0 2: \mathbf{z} := \mathbf{A}_1 \rightarrow \ell' 0 2 \in P'''$:
 11. Let $R = \{ \ell m' 2: \mathbf{z} := \mathbf{A}_1 \rightarrow \ell' m' 2 \in P''' \mid 1 \leq m' \leq n - 1 \}$.
 12. Let $P''''' = ((P' \cup P''') \setminus R) \cup \text{true}[[R]]$.
 13. For each $\ell''(n - 1) 2: \mathbf{z} := \mathbf{A}_2 \rightarrow \ell''(n - 1) 2 \in P'''''$ with $\ell \neq \ell''$:
 14. Slice P''''' backward by criterion $\ell''(n - 1) 2: \mathbf{z} := \mathbf{A}_2 \rightarrow \ell''(n - 1) 2$; get P'''''' .
 15. Let $Z = \{ \mathbf{A}[[\mathbf{A}_2]] \rho' \mid \exists \sigma. \exists \mathfrak{S}. \sigma \in \mathcal{S}[[P'''''']] \wedge \vdash \sigma = \langle \rho, \mathfrak{S} \rangle \wedge \neg \sigma = \langle \rho', \ell''(n - 1) 2: \mathbf{z} := \mathbf{A}_2 \rightarrow \ell''(n - 1) 2 \rangle \}$.
 16. Let $S = S \cup Z$.
 17. Return S .

Fig. 6.2. Extraction algorithm

Extraction

We know, from the previous Section, that loop-based watermarks requiring n iterations are embedded only in for-loops performing, on input ρ , at least n iterations. The extraction algorithm (see Figure 6.2) checks this condition on the for-loop P it is given in input. This is the first critical step.

We also expect these statements to be the only statements that assign z within P''' . Furthermore, we expect a subset of these statements to implement an Horner scheme. In particular:

- we expect the first statement with command $z := A_1$, that is, S , to provide z with the correct initialization;
- we expect the n statements with command $z := A_2$ to provide z with the n updates required by the Horner scheme.

Finally, we expect the remaining $n - 1$ statements with command $z := A_1$ to be irrelevant to the Horner scheme. So the next step is to get rid of such statements.

The algorithm collects them in a set R . By computing $P'''' = ((P' \cup P''') \setminus R) \cup \text{true}[[R]]$, it obtains:

$$\begin{aligned}
 P'''' = \{ & \dots\dots\dots \\
 & \dots \ell'02: z := A_1 \rightarrow \ell'02, \dots \\
 & \dots \ell''02: z := A_2 \rightarrow \ell''02, \dots \\
 & \dots \ell'12: \text{true} \rightarrow \ell'12, \dots \\
 & \dots \ell''12: z := A_2 \rightarrow \ell''12, \dots \\
 & \dots \ell'22: \text{true} \rightarrow \ell'22, \dots \\
 & \dots \ell''22: z := A_2 \rightarrow \ell''22, \dots \\
 & \dots \ell'32: \text{true} \rightarrow \ell'32, \dots \\
 & \dots \ell''32: z := A_2 \rightarrow \ell''32, \dots \\
 & \dots\dots\dots \\
 & \dots \ell(n-1)2: \text{true} \rightarrow \ell(n-1)2, \dots \\
 & \dots \ell''(n-1)2: z := A_2 \rightarrow \ell''(n-1)2, \dots \}
 \end{aligned}$$

In order to get the value of z , the algorithm slices P'''' backward by criterion $\ell''(n-1)2: z := A_2 \rightarrow \ell''(n-1)2$, thus obtaining P''''' . Such program is expected to implement (6.2) and to be equivalent to (6.3). Therefore, whenever P''''' is executed on input ρ , variable z is expected to eventually evaluate to signature \mathfrak{s} .

Decoding

To get the final value of z , the algorithm considers every maximal trace $\sigma \in \mathcal{S}[[P''''']]$ with initial environment ρ , and evaluates A_2 in the last environment of the trace. Of course, this is another critical step in the algorithm. All the values obtained are collected in a set Z .

The algorithm recovers the values for every pair of statements in the body of P''' that are candidate stegomarks. All the candidate signatures are collected in S , which is returned by the algorithm upon termination. We only need to retrieve our signature \mathfrak{s} among the candidate signature collected in S .

6.8 Determining the Signature

Signature \mathfrak{s} must reliably identify the author of the watermarked program. To this end, the author can let \mathfrak{s} be the product of a set of prime numbers. If some factors

of \mathfrak{s} are large enough, its factorization is computationally unfeasible, yet the author is able to produce it.

In principle any natural number, regardless of its magnitude, can be a signature. Yet, real computers can solely deal with finite and discrete data. Thus \mathfrak{s} , as well as any program number, cannot exceed a prefixed maximum M . A solution has however been suggested [34] to turn a large signature into a set of smaller numbers. Suppose $k > 0$. Let n_1, n_2, \dots, n_k be natural numbers which are pairwise coprime, and let p be their product. The *Chinese remainder theorem* [7] establishes an isomorphism between $[0, p - 1]$ and $[0, n_1 - 1] \times \dots \times [0, n_k - 1]$. Thus, we can represent $0 \leq \mathfrak{s} < p$ in terms of k smaller numbers $0 \leq \mathfrak{s}_j < n_j$, where $1 \leq j \leq k$. If we constraint $n_j \leq M$, the k smaller numbers do not exceed M , so we can safely embed them in as many for-loops.

At extraction time, we run the extraction algorithm on every for-loop of the watermarked program. We then retrieve our signature in the union of all the sets of candidate signatures, possibly assembling several numbers through the Chinese remainder theorem. False positives may be obtained at extraction time, both in the case of watermarked and unwatermarked loops. However, it is unlikely that their factorization is computationally unfeasible and yet known by a malicious claimer.

6.9 Quantitative Evaluation

We can embed a watermark in a program as long as the program has at least one for-loop. We know from the previous Section that the signature can be of any size. So the data-rate is theoretically infinite. If we apply the Chinese remainder theorem, we can have several watermarks to embed. Actually, there are no limits to the number of watermarks we can inlay in a for-loop P , because each watermark exploits a variable which is fresh in P . To improve stealth, however, it is recommendable to restrict the number of watermarks per loop.

Suppose in each for-loop we implant not more than one watermark. Then the data-rate is practically limited by the number of for-loops in the subject program. To estimate such number, we downloaded some open-source Java libraries from the World Wide Web. We fed each library to CLOC [40], a Perl [119] script for counting the line of codes. We determined the number of for-loops through a Bash [109] script based on `grep` and considering only for-loop with integer index value. We obtained the figures reported in Table 6.1.

The Table is divided into two groups. The first group reports the results of four libraries we came across by randomly surfing the Web. They are representative of generic Java source code. The ratio of the number of for-loops to the number of lines of code is rather low in every entry but the first one. The minimum, 0.77%, is denoted by JDOM [110], a library for the Java representation of XML documents. If we consider the four libraries all together and divide the overall number of for-loops by the overall number of lines of codes, we obtain 1.34%. The average ratio is a bit higher, 2.46%, meaning that in a generic Java library about 4 for-loops are expected to be found for every 200 lines of code.

We believe there are two reasons for the especially high ratio of the first entry, that is, the Fhourstone Benchmark [114]. First, this library is small; hence the

	Lines of code	For-loops	Ratio
<i>Randomly chosen Java libraries</i>			
[114] The Fhourstone Benchmark 3.1	476	30	6.30%
[110] JDOM 1.1.1	10 084	78	0.77%
[50] Google Collections Library 1.0	16 333	187	1.14%
[97] Byte Code Engineering Library (BCEL)	23 631	383	1.62%
Total	50 524	678	1.34%
Average ratio			2.46%
<i>Java libraries for numerical computing</i>			
[96] SciMark 2.0	1 786	132	7.39%
[61] Java Matrix Package (Jama) 1.0.2	2 843	275	9.67%
[51] Matrix-Toolkits-Java (MTJ) 0.9.12	15 872	719	4.53%
[88] Open source Java Algorithms (ojAlgo)	33 666	1 056	3.14%
Total	54 167	2 182	4.03%
Average ratio			6.18%

Table 6.1. Ratio of the number of for-loop constructs to the number of lines of codes of some open-source Java libraries. The first and the second column respectively provide references and names of the libraries. *Lines of code* does not include blank lines and comment lines. *For-loops* records the number of for-loops with integer index variable.

correspondent ratio is very sensitive to the number of for-loops. Second, the library implements *Connect Four*,¹ a board game which is amenable to a programming style based on arrays and matrices. Muchnich [80] argues that for-loops occur very often whenever there are arrays and matrices to deal with. In particular, the author points out numerical computing as a field where for-loops are massively used.

The last four entries in Table 6.1 describe the results of four numerical libraries for Java. Again, the highest ratios, 7.39% and 9.67%, are denoted by the entries with a lower number of lines of code, that is, SciMark [96] and Jama [61]. Notice however that ojAlgo [88], the entry with the largest size in the whole Table, has ratio 3.14%, which is better than the average ratio of the first group. The overall and average ratios of the second group, 4.03% and 6.18% respectively, are about three times as great as the correspondent ratios in the first group. So, whenever we are given a numerical Java library, we expect to find more than 12 for-loops for every 200 lines of code.

All in all, our watermarking technique seems to be especially suitable for programs whose data structures are implemented through array and matrices, as in the case of numerical computing. Anyway, it seems that a not nearly negligible amount of for-loops can be found in randomly chosen programs. Therefore we assume that a program that is complex enough to be worth protecting has also enough for-loops for the embedding of a signature.

¹ The Italian for this game is *Forza quattro*.

6.10 Attacks and Countermeasures

We assumed at embedding time that watermark variable w is fresh in the for-loop P to be watermarked. If P is actually a subroutine of a program P' , we also require that w is fresh in P' , too. Therefore w is always an additional variable in P' , consuming additional memory. If P' has plenty of variables and the memory is expensive, we can alternatively embed a stegomark in P by assigning variables of P' that are *dead at P*, that is, that are not exploited any longer when P is executed. At any rate, an attacker could realize, for instance by slicing P' backward from the output, that the watermark variable does not affect the value of the output and therefore eliminate the stegomark. To avoid this inconvenience, we *must* introduce fake dependencies between the output and the watermark variable, for example by using *opaque predicates* which require hard program analyses to be removed [24]. Still, the watermark might be easily discovered by an attacker because, as our example in Section 6.3 shows, parameters a, b, \dots tend to be much bigger than the numeric values which customarily are found inside arithmetic expressions. To overcome this problem we can compute the parameters using functions fed with small numbers.

Watermarked programs can include more than one signature. However, they do not record which signature was inserted first, and which one was inserted later. Thus an attacker can embed its own signature in a watermarked program and claim authorship, thereby accomplishing an *additive attack*. To the best of our knowledge, vulnerability to additive attacks is a common drawback to all the exiting watermarking techniques [16, 21]. Hence one must rely upon trustworthy authorities at the embedding time to certify the temporal precedence of his/her signature. This could be avoided if the insertion of the signature coincided with a not reversible semantics-preserving program *evolution* [15]: in such a case the order of insertion of signatures would become relevant, especially if later evolutions were strictly dependent on earlier ones.

We can embed different signatures in copies of the same program. In this scenario, signatures are called fingerprints and are meant to identify the purchasers of each copy, rather than the author of the program. Licensing numbers are typical examples of fingerprints. An attacker may localize fingerprints by looking for discrepancies among several purchased copies. To face these *collusive attacks* we could, in principle, let each copy undergo a different obfuscating transformation, thereby making copies so different that their comparison for the sake of fingerprint detection becomes unfruitful.

By now it should be clear that our watermarking technique is quite far from being inherently resilient. Not only it does not withstand manual attacks supported by enough manpower, time and human motivation, as it is assumed for any watermarking technique proposed so far [23, 34], but it is especially vulnerable because, contrary to e.g. the threading watermarking technique [85], it fails to bind the watermark to the semantics of the watermarked program. Indeed, all the countermeasures we discussed in the present and in the previous Section just try to cope with this deficiency.

The main point of our technique is the way programs are addressed. Rather than engineering products to be protected against intellectual property infringement, they are foremost considered carriers of information. The steganographic approach

to software watermarking is thereby recast as the science of moving such information from manifest to latent at embedding time, and the other way round at extraction time. In our technique, we take advantage of loop unrolling as the engine of both embedding and extraction algorithm, thus relying upon for-loops. We think that our approach may be extended to other programming constructs which, like for-loops, provide code reuse, such as recursive functions in the functional paradigm and objects in the object-oriented paradigm.

A

Tools

In this Appendix, we survey 56 tools for software protection, almost all of them performing code obfuscation. For each tool we provide the name, the developer and the programming language the tools targets; we also specify the kind of performed obfuscation, the price and the operating systems or platforms the tool needs to run; we conclude possibly reporting some features and giving a pointer to the website where the tool can be purchased or downloaded.

We specify the kind of performed obfuscation through the $\circ-\circ-\circ$ symbol. In particular:

- $\bullet-\circ-\circ$ indicates that the tool perform layout obfuscation;
- $\circ-\bullet-\circ$ indicates that the tool perform control obfuscation;
- $\circ-\circ-\bullet$ indicates that the tool perform data obfuscation.

Every combination of the three is allowed. Our survey dates to November 22, 2007.

Ada Obfuscator *by Semantic Designs*. Target: Ada.

Obfuscation type: $\bullet-\circ-\circ$ Price: unknown. Platform: Windows.

Identifier renaming, comments and white spaces removal.

[http:](http://www.semdesigns.com/Products/Obfuscators/AdaObfuscator.html)

[//www.semdesigns.com/Products/Obfuscators/AdaObfuscator.html](http://www.semdesigns.com/Products/Obfuscators/AdaObfuscator.html)

Allatori *by Smardec*. Target: Java.

Obfuscation type: $\bullet-\bullet-\bullet$ Price: \$ 300. Platform: any.

Identifier renaming (trying to give the same name to as many identifiers as possible), use of goto, obfuscation of debug information, string encryption, watermarking.

<http://www.allatori.com>

CafeBabe *by Alexander*. Target: Java.

Obfuscation type: $\bullet-\circ-\circ$ Price: \$ 0. Platform: any.

<http://www.geocities.com/CapeCanaveral/Hall/2334/Programs/cafebabe.html>

Clisecure *by SecureTeam*. Target: .NET.

Obfuscation type: $\bullet-\circ-\bullet$ Price: \$ 1200. Platform: Windows.

<http://www.secureteam.net>

- Cloakware Security Suite** *by Cloakware*. Target: Java, C, C++.
 Obfuscation type: ●-●-● Price: unknown. Platform: any.
<http://www.cloakware.com>
- C Obfuscator** *by Semantic Designs*. Target: C, Visual C 6.
 Obfuscation type: ●-○-○ Price: unknown. Platform: Windows.
 Identifier renaming.
<http://www.semdesigns.com/Products/Obfuscators/CObfuscator.html>
- CodeVeil** *by Xheo*. Target: .NET.
 Obfuscation type: ●-○-● Price: \$ 900. Platform: Windows.
<http://www.xheo.com/products/codeveil/default.aspx>
- C++ Obfuscator** *by Semantic Designs*. Target: C++, Visual C++ 6.
 Obfuscation type: ●-○-○ Price: unknown. Platform: Windows.
 Identifier renaming.
<http://www.semdesigns.com/Products/Obfuscators/CppObfuscator.html>
- C# Obfuscator** *by Semantic Designs*. Target: C#.
 Obfuscation type: ●-○-○ Price: unknown. Platform: Windows.
 Identifier renaming, comments and white spaces removal.
<http://www.semdesigns.com/Products/Obfuscators/CSharpObfuscator.html>
- DashO** *by PreEmptive Solutions*. Target: Java.
 Obfuscation type: ●-●-● Price: \$ 1900. Platform: Windows.
 Identifier renaming possibly exploiting methods overloading, use of goto in the control flow, string encryption, watermarking.
<http://www.preemptive.com/products/dasho/>
- Decompiler.NET** *by Jungle Creatures*. Target: .NET.
 Obfuscation type: ●-●-● Price: \$. Platform: Windows.
 Identifier renaming, string encryption, dead code elimination, if-merging...
http://www.junglecreatures.com/Jungle_Portal/docs/Decompiler.NET/Try.aspx
- Demeanor** *by Wise Owl*. Target: .NET.
 Obfuscation type: ●-●-○ Price: \$ 800. Platform: Windows.
<http://www.wiseowl.com/purchase/purchase.aspx>
- .NET Obfuscator** *by Dynu*. Target: .NET.
 Obfuscation type: ●-●-○ Price: \$ 50. Platform: Windows.
<http://www.dynu.com/dynuobfuscator.asp>
- .NET Reactor** *by Eziriz*. Target: .NET, C#.
 Obfuscation type: ●-●-● Price: \$ 180-280. Platform: Windows, Mac OS, Linux, Solaris, BSD.
<http://www.eziriz.com/products.htm>
- .Obfuscator** *by Aspose*. Target: .NET.
 Obfuscation type: ●-○-○ Price: \$ 0. Platform: Windows.
 Identifier renaming.
<http://www.aspose.com/Products/Aspose.Obfuscator/>

- dotfuscator** by *PreEmptive Solutions*. Target: .NET.
 Obfuscation type: ●-●-● Price: \$ 1900. Platform: Windows.
 Identifier renaming possibly exploiting methods overloading, use of goto in the control flow, string encryption, watermarking.
<http://www.preemptive.com/products/dotfuscator/>
- dotNet Protector** by *PV Logiciels*. Target: .NET.
 Obfuscation type: ●-○-○ Price: \$ 400. Platform: Windows.
<http://dotnetprotector.pvlog.com/Home.aspx>
- Goliath .NET Obfuscator** by *Cantelmo Software*. Target: Java.
 Obfuscation type: ●-○-● Price: \$ 370. Platform: Windows.
<http://www.cantelmosoftware.com/eng/obfuscator.html>
- IL-Obfuscator** by *Lesser-Software*. Target: .NET.
 Obfuscation type: ●-●-● Price: \$ 30. Platform: Windows.
http://www.lessor-software.com/en/content/products/LSW%20DotNet-Tools/LSW_DotNet_IL-Obfuscator.htm
- Jarg** by *Hidetoshi Ohuchi*. Target: Java.
 Obfuscation type: ●-●-○ Price: \$ 0. Platform: any.
 Identifier renaming, dead code removal, NOP removal.
<http://jarg.sourceforge.net>
- Jasob** by *Jasob*. Target: Javascript, CSS.
 Obfuscation type: ●-○-○ Price: \$ 200-600. Platform: Windows.
 Identifier renaming, comments and white spaces removal, use of escape sequences.
<http://www.jasob.com>
- JavaGuard** by *glurk*. Target: Java.
 Obfuscation type: ○-○-○ Price: \$ 0. Platform: any.
<http://sourceforge.net/projects/javaguard/>
- Java Obfuscator** by *Semantic Designs*. Target: Java.
 Obfuscation type: ●-○-○ Price: unknown. Platform: Windows.
 Identifier renaming, comments and white spaces removal.
<http://www.semdesigns.com/Products/Obfuscators/JavaObfuscator.html>
- JAX** by *IBM*. Target: Java.
 Obfuscation type: ●-●-○ Price: unknown. Platform: any.
 dead code removal, debugging information removal, method inlining, class hierarchy transformation, identifier renaming.
<http://researchweb.watson.ibm.com/jax/>
- JBCO** by *Sable*. Target: Java.
 Obfuscation type: ●-●-● Price: \$ 0. Platform: any.
 Identifier renaming, embedding constant values as fields, some data obfuscation, use of goto, if replaced with try/catch, instruction reordering. . . .
<http://www.sable.mcgill.ca/JBCO/>
- JCIPHER** by *Kindisoft*. Target: JavaScript.
 Obfuscation type: ●-●-○ Price: \$ 420. Platform: any.
<http://www.kindisoft.com/products/>

JCloak by *Force5*. Target: Java.

Obfuscation type: ●-○-○ Price: \$ 600. Platform: Windows, Solaris.

Identifier renaming.

<http://www.force5.com/JCloak/ProductJCloak.html>

Jet by *Excelsior*. Target: Java.

Obfuscation type: ○-○-○ Price: \$ 1200-4500. Platform: Windows, Linux.

This is a Java2IA32 compiler.

<http://www.excelsior-usa.com/landing/jet-obfuscator.html>

Jobfuscate by *Duckware*. Target: Java.

Obfuscation type: ●-○-○ Price: \$ 100. Platform: Windows.

Identifier renaming.

<http://www.duckware.com/jobfuscate/>

JODE by *Jochen Hoenicke*. Target: Java.

Obfuscation type: ●-○-○ Price: \$ 0. Platform: any.

Identifier renaming, debugging information removal, dead code removal.

<http://jode.sourceforge.net>

Jshrink by *Eastridge Technology*. Target: Java.

Obfuscation type: ●-○-● Price: \$ 100. Platform: any.

Identifier renaming, string encryption.

<http://www.e-t.com/jshrink.html>

KlassMasterTM by *Zelix*. Target: Java.

Obfuscation type: ●-●-● Price: \$ 200-400. Platform: any.

Identifier renaming, use of goto, string encryption.

<http://www.zelix.com/klassmaster/>

Loco by *Universiteit Gent (NL)*. Target: C, C++.

Obfuscation type: ○-●-○ Price: \$ 0. Platform: any.

Control flow flattening.

<http://diablo.elis.ugent.be/obfuscation>

Marvin by *Dr. Java*. Target: Java.

Obfuscation type: ●-●-● Price: unknown. Platform: any.

<http://www.drjava.de/obfuscator>

Obfuscator .NET by *Macrobject*. Target: .NET.

Obfuscation type: ●-●-○ Price: \$ 100. Platform: Windows.

<http://www.macrobobject.com/en/obfuscator/>

PCGuard by *sofpro*. Target: .NET.

Obfuscation type: ●-○-○ Price: \$ 370-600. Platform: Windows.

Very extended watermarking, fingerprinting.

<http://www.sofpro.com/pcgw32.htm>

Phoenix Protector by *NTCore*. Target: .NET.

Obfuscation type: ●-●-● Price: \$ 250. Platform: Windows.

<http://ntcore.com/phoenix.php>

PL/SQL Obfuscator by *Semantic Designs*. Target: PLSQL.

Obfuscation type: ●-○-○ Price: unknown. Platform: Windows.

Identifier renaming, comments and white spaces removal.

[http:](http://www.semdesigns.com/Products/Obfuscators/PLSQLObfuscator.html)

[//www.semdesigns.com/Products/Obfuscators/PLSQLObfuscator.html](http://www.semdesigns.com/Products/Obfuscators/PLSQLObfuscator.html)

- Postbuild** *by Xenocode*. Target: .NET.
 Obfuscation type: ●—●—● Price: \$ 500. Platform: Windows.
<http://www.xenocode.com/Products/Postbuild/>
- proGuard** *by Eric Lafortune*. Target: Java.
 Obfuscation type: ●—●—○ Price: \$ 0. Platform: any.
 Identifier renaming, debugging information removal, some instruction restructuration.
<http://proguard.sourceforge.net>
- QND-Obfuscator** *by Desaware*. Target: .NET.
 Obfuscation type: ●—○—○ Price: \$ 40. Platform: Windows.
<http://www.desaware.com/products/books/net/obfuscating/index.aspx>
- RetroGuard** *by RetroLogic*. Target: Java.
 Obfuscation type: ●—○—○ Price: \$ 139. Platform: any.
<http://www.retrologic.com>
- Salamander .NET Obfuscator** *by Remotesoft*. Target: .NET, C#.
 Obfuscation type: ●—○—○ Price: \$ 800. Platform: Windows.
<http://www.remotesoft.com/salamander/obfuscator.html>
- Sandmark** *by the University of Arizona (US)*. Target: Java.
 Obfuscation type: ●—●—● Price: \$ 0. Platform: any.
 Algorithms for both obfuscation and watermarking.
<http://sandmark.cs.arizona.edu>
- secureSWF professional** *by Kindisoft*. Target: Adobe's Flash SWF and SWC files.
 Obfuscation type: ●—●—● Price: \$ 420. Platform: Windows.
<http://www.kindisoft.com/products/>
- Skater .NET Obfuscator** *by Rustemsoft*. Target: .NET.
 Obfuscation type: ●—●—● Price: \$ 100-570. Platform: Windows.
<http://www.rustemsoft.com/Skater.htm>
- {smartassembly}** *by Cachupa*. Target: .NET.
 Obfuscation type: ●—●—● Price: \$ 400-800. Platform: Windows.
<http://www.smartassembly.com>
- Smokescreen** *by Robert Lee*. Target: Java.
 Obfuscation type: ●—●—● Price: \$ 550. Platform: any.
 Identifier renaming, string encryption, instruction reordering, fake exceptions.
<http://www.leesw.com/smokescreen/>
- Spices .Obfuscator** *by 9rays.net*. Target: .NET.
 Obfuscation type: ●—●—● Price: \$ 400. Platform: Windows.
 Includes watermarking.
<http://www.9rays.net/Products/Spices.Obfuscator/>
- SystemC Obfuscator** *by Semantic Designs*. Target: SystemC.
 Obfuscation type: ●—○—○ Price: unknown. Platform: Windows.
 Identifier renaming, comments and white spaces removal.
<http://www.semdesigns.com/Products/Obfuscators/SystemCObfuscator.html>

SystemVerilog Obfuscator *by Semantic Designs*. Target: SystemVerilog.

Obfuscation type: ●-○-○ Price: unknown. Platform: Windows.

Identifier renaming, comments and white spaces removal.

<http://www.semdesigns.com/Products/Obfuscators/SystemVerilogObfuscator.html>

ThicketTM Obfuscator *by Semantic Designs*. Target: PHP.

Obfuscation type: ●-○-○ Price: unknown. Platform: Windows.

Identifier renaming, comments and white spaces removal.

[http:](http://www.semdesigns.com/Products/Obfuscators/PHP0bfuscator.html)

[//www.semdesigns.com/Products/Obfuscators/PHP0bfuscator.html](http://www.semdesigns.com/Products/Obfuscators/PHP0bfuscator.html)

VBScript Obfuscator *by Semantic Designs*. Target: VBScript.

Obfuscation type: ●-○-○ Price: unknown. Platform: Windows.

Identifier renaming.

<http://www.semdesigns.com/Products/Obfuscators/VBScriptObfuscator.html>

Verilog Obfuscator *by Semantic Designs*. Target: Verilog.

Obfuscation type: ●-○-○ Price: unknown. Platform: Windows.

Identifier renaming, comments and white spaces removal.

[http:](http://www.semdesigns.com/Products/Obfuscators/VerilogObfuscator.html)

[//www.semdesigns.com/Products/Obfuscators/VerilogObfuscator.html](http://www.semdesigns.com/Products/Obfuscators/VerilogObfuscator.html)

VHDL Obfuscator *by Semantic Designs*. Target: VHDL.

Obfuscation type: ●-○-○ Price: unknown. Platform: Windows.

Identifier renaming, comments and white spaces removal.

[http:](http://www.semdesigns.com/Products/Obfuscators/VHDL0bfuscator.html)

[//www.semdesigns.com/Products/Obfuscators/VHDL0bfuscator.html](http://www.semdesigns.com/Products/Obfuscators/VHDL0bfuscator.html)

yGuard *by yWorks*. Target: Java.

Obfuscation type: ●-○-○ Price: \$ 0. Platform: any.

Identifier renaming.

http://www.yworks.com/en/products_yguard_about.htm

References

1. G. Arboit. A method for watermarking java programs via opaque predicates. In *Proc. Int. Conf. Electronic Commerce Research (ICECR-5)*, 2002.
2. D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.
3. B. Barak, O. Goldreich, R. Impagliazzo, R. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *CRYPTO: Proceedings of Crypto*, November 2001.
4. J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie, and N. Tawbi. Static detection of malicious code in executable programs. *Int. J. of Req. Eng.*, 2001.
5. A. Berk, C. A. Kaiser, H. F. Lodish, J. Darnell, M. P. Scott, M. Krieger, P. Matsudaira, and S. L. Zipursky. *Molecular Cell Biology*. W. H. Freeman & Co., fifth edition, 2003.
6. M. Bertacca and A. Guidi. *Introduzione a Java*. McGraw-Hill, first edition, October 1997.
7. E. D. Bolker. *Elementary Number Theory: An Algebraic Approach*. Dover Books on Mathematics. Dover Publications, 2007.
8. J. Bryans, M. Koutny, L. Mazaré, and P. Y. A. Ryan. Opacity generalised to transition systems. In Theo Dimitrakos, Fabio Martinelli, Peter Y. A. Ryan, and Steve A. Schneider, editors, *Revised Selected Papers of the 3rd International Workshop on Formal Aspects in Security and Trust (FAST'05)*, volume 3866 of *Lecture Notes in Computer Science*, pages 81–95, Newcastle upon Tyne, UK, 2006. Springer.
9. J. Bryans, M. Koutny, and P. Y. A. Ryan. Modelling dynamic opacity using petri nets with silent actions. In Theodosios Dimitrakos and Fabio Martinelli, editors, *Formal Aspects in Security and Trust*, pages 159–172. Springer, 2004.
10. R. Canetti. Towards realizing random oracles: Hash functions that hide all partial information. *Lecture Notes in Computer Science*, 1294:455–??, 1997.
11. S. Chow, Y. Gu, H. Johnson, and V. A. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In *ISC '01: Proceedings of the 4th International Conference on Information Security*, pages 144–155, London, UK, 2001. Springer-Verlag.
12. M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 32–46, Washington, DC, USA, 2005. IEEE Computer Society.
13. C. Cifuentes and K. J. Gough. Decompilation of binary programs. Technical Report FIT-TR-1994-03, Queensland University of Technology, Brisbane (Australia), 1994.
14. F. B. Cohen. Computer viruses: theory and experiments. *Comput. Secur.*, 6(1):22–35, 1987.

15. F. B. Cohen. Operating system protection through program evolution. *Comput. Secur.*, 12(6):565–584, 1993.
16. C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn, and M. Stepp. Dynamic path-based software watermarking. *SIGPLAN Not.*, 39(6):107–118, 2004.
17. C. Collberg, A. Huntwork, E. Carter, and G. Townsend. Graph theoretic software watermarks: Implementation, 2004.
18. C. Collberg, S. Jha, D. Tomko, and H. Wang. Uwstego: A general architecture for software watermarking, 2001.
19. C. Collberg, G. Myles, and J. Nagra. Surreptitious software. Draft, 2006-2008.
20. C. Collberg and C. Thomborson. the limits of software watermarking, 1998.
21. C. Collberg and C. Thomborson. Software watermarking: Models and dynamic embeddings. In *Principles of Programming Languages 1999, POPL'99*, San Antonio, TX, January 1999.
22. C. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation – tools for software protection. Technical Report TR00-03, University of Arizona, Feb, 10 2000.
23. C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, University of Auckland, July 1997.
24. C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998*, San Diego, CA, 1998.
25. M. Cooper, S. Northcutt, M. Fearnow, and K. Frederick. *Intrusion Signatures and Analysis*. Sams, January 2001.
26. Intel Corporation. IA-32 Intel Architecture Software Developer's Manual.
27. P. Cousot. Abstract interpretation. *ACM Comput. Surv.*, 28(2):324–328, 1996.
28. P. Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Theoretical Computer Science*, 277(1-2):47–103, 2002.
29. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
30. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.
31. P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.
32. P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretations. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 83–94, New York, NY, USA, 1992. ACM.
33. P. Cousot and R. Cousot. Systematic Design of Program Transformation Frameworks by Abstract Interpretation. In *Conference Record of the 19th ACM Symposium on Principles of Programming Languages*, pages 178–190, New York, 2002. ACM Press.
34. P. Cousot and R. Cousot. An abstract interpretation-based framework for software watermarking. In *Conference Record of the Thirtyfirst Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Venice, Italy, 2004. ACM Press, New York.
35. M. Dalla Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics-based approach to malware detection. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 377–388, New York, NY, USA, 2007. ACM Press.

36. M. Dalla Preda and R. Giacobazzi. Control code obfuscation by abstract interpretation. In *Proc. of the 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM '05)*, pages 301–310, 2005.
37. M. Dalla Preda and R. Giacobazzi. Semantic-based code obfuscation by abstract interpretation. In *Proc. of the 32nd International Colloquium on Automata, Languages and Programming (ICALP '05)*, volume 3580 of *Lecture Notes in Computer Science*, pages 1325–1336. Springer-Verlag, 2005.
38. M. Dalla Preda, R. Giacobazzi, and E. Visentini. Hiding software watermarks in loop structures. In Maria Alpuente and German Vidal, editors, *Static Analysis: Proceedings of the 15th International Static Analysis Symposium*, volume 5079, pages 174–188. Springer, July 2008.
39. M. Dalla Preda, M. Madou, K. De Bosschere, and R. Giacobazzi. Opaque predicates detection by abstract interpretation. In *Proc. of the 11th International Conference on Algebraic Methodology and Software Technology (AMAST '06)*, volume 4019 of *Lecture Notes in Computer Science*, pages 81–95. Springer-Verlag, 2006.
40. A. Danial. Count lines of code (cloc) 1.09. <http://cloc.sourceforge.net/>, March 2010.
41. J. W. Davidson and S. Jinturkar. An aggressive approach to loop unrolling. Technical report, Proc. Compiler Construction '96, 1995.
42. R. L. Davidson and N. Myhrvold. Method and systems for generating and auditing a signature for a computer program. US patent 5.559.884, Assignee: Microsoft Corporation, 1996.
43. J. Dean. Whole-program optimization of object-oriented languages. Technical Report TR-96-11-05, University of Washington, 1996.
44. J. Fritzing and M. Mueller. Java security. Technical report, Sun Microsystems, Inc., 1996.
45. M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.
46. R. Giacobazzi. Hiding information in completeness holes - new perspectives in code obfuscation and watermarking. In *Proc. of The 6th IEEE International Conferences on Software Engineering and Formal Methods (SEFM'08)*, pages 7–20. IEEE Press., 2008.
47. R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples and refinements in abstract model-checking. In P. Cousot, editor, *Proc. of The 8th International Static Analysis Symposium, SAS'01*, volume 2126 of *Lecture Notes in Computer Science*, pages 356–373. Springer-Verlag, 2001.
48. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *Journal of the ACM*, 47(2):361–416, 2000.
49. S. Goldwasser and Y. T. Kalai. On the impossibility of obfuscation with auxiliary input. In *FOCS '05: Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*, pages 553–562, Washington, DC, USA, 2005. IEEE Computer Society.
50. Google Inc. Google collections library 1.0. <http://code.google.com/p/google-collections/>, December 2009.
51. Google Inc. Matrix-Toolkits-Java (MTJ) 0.9.12. <http://code.google.com/p/matrix-toolkits-java/>, April 2009.
52. J. R. Gosler. Software protection: Myth or reality? In *CRYPTO*, pages 140–157, 1985.
53. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.
54. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, third edition, June 2005.

55. D. Grover. *The protection of computer software: its technology and applications*. The British Computer Society Monographs in Informatics, Cambridge university press, 1992.
56. Jr. H. Rogers. *Theory of recursive functions and effective computability*. MIT Press, Cambridge, MA, USA, 1987.
57. MH Halstead. *Elements of Software Science*. Elsevier, New York, 1977.
58. R. Harper. Practical foundations for programming languages. Draft, December 2009.
59. W. A. Harrison and K. I. Magel. A complexity measure based on nesting level. *SIGPLAN Not.*, 16(3):63–74, March 1981.
60. J. L. Hennessy, D. A. Patterson, and A. C. Arpaci-Dusseau. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, fourth edition, 2007.
61. J. Hicklin, C. Moler, P. Webb, R. F. Boisvert, B. Miller, R. Pozo, and K. Remington. Java Matrix Package (Jama) 1.0.2. <http://math.nist.gov/javanumerics/jama/>, July 2005.
62. C.A.R. Hoare. Private communication, September 2007.
63. F. Hohl. Time limited blackbox security: Protecting mobile agents from malicious hosts. *Lecture Notes in Computer Science*, 1419:92–??, 1998.
64. S. Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Transactions on Programming Languages and Systems*, 19(1):1–6, January 1997.
65. M. Jakobsson and M. K. Reiter. Discouraging software piracy using software aging. In *Digital Rights Management Workshop*, pages 1–12, 2001.
66. Lord Julius. Metamorphism. *29A Magazine*, 5, 2000.
67. A. B. Kahng, J. Lach, W. H. Mangione-Smith, S. Mantik, I. L. Markov, M. Potkonjak, P. Tucker, H. Wang, and G. Wolfe. Constraint-based watermarking techniques for design IP protection. *IEEE TCAD: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(10):1236–1252, October 2001.
68. A. Kerckhoffs. La cryptographie militaire. *Journal des sciences militaires*, IX:5–83, January 1883.
69. J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting malicious code by model checking. In Klaus Julisch and Christopher Krügel, editors, *DIMVA*, volume 3548 of *Lecture Notes in Computer Science*, pages 174–187. Springer, 2005.
70. Y. Lakhnech and L. Mazare. Probabilistic opacity for a passive adversary and its application to chaum’s voting scheme. Cryptology ePrint Archive, Report 2005/098, 2005. <http://eprint.iacr.org/>.
71. A. Lakhotia and M. Mohammed. Imposing order on program statements to assist Anti-Virus scanners. In *WCRE*, pages 161–170, 2004.
72. R. W. Lo, K. N. Levitt, and R. A. Olsson. Mcf: a malicious code filter. *Computers & Security*, 14(6):541–566, 1995.
73. B. Lynn, M. Prabhakaran, and A. Sahai. Positive results and techniques for obfuscation, 2004.
74. A. Majumdar and C. Thomborson. Manufacturing opaque predicates in distributed systems for code obfuscation. In *ACSC ’06: Proceedings of the 29th Australasian Computer Science Conference*, pages 187–196, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.
75. V. Manca. *Metodi informazionali*. Bollati Boringhieri, September 2003.
76. M. L. Miller, I. J. Cox, J. P. M. G. Linnartz, and T. Kalker. A review of of watermarking principles and practices. In K. K. Parhi and T. Nishitani, editors, *Digital Signal Processing for Multimedia Systems*, pages 461–485. IEEE, 1999.
77. A. Monden, H. Iida, et al. A watermarking method for computer programs (in japanese). In *Proceedings of the 1998 Symposium on Cryptography and Information Security, SCIS’98*. Institute of Electronics, Information and Communication Engineers, January 1998.

78. A. Monden, H. Iida, and K. Matsumoto. A practical method for watermarking java programs. In *The 24th Computer Software and Applications Conference*, pages 191–197, 2000.
79. S. A. Moskowitz and M. Cooperman. Method for stega-cipher protection of computer code. US patent 5.745.569, Assignee: The Dice Company, 1996.
80. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, second edition, August 1997.
81. G. Myles and C. Collberg. Software watermarking through register allocation: Implementation, analysis, and attacks. In *ICISC*, pages 274–293, 2003.
82. G. Myles and C. Collberg. Software watermarking via opaque predicates: Implementation, analysis, and attacks. *Electronic Commerce Research*, 6(2):155–171, 2006.
83. G. Myles and H. Jin. Self-validating branch-based software watermarking. In Mauro Barni, Jordi Herrera-Joancomartí, Stefan Katzenbeisser, and Fernando Pérez-González, editors, *Information Hiding*, volume 3727 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2005.
84. J. Nagra, C. Thomborson, and C. Collberg. Software watermarking: Protective terminology, 2001.
85. J. Nagra and C. D. Thomborson. Threading software watermarks. In *Information Hiding*, volume 3200 of *Lecture Notes in Computer Science*, pages 208–223. Springer, 2004.
86. Ogiso, Sakabe, Soshi, and Miyaji. Software Obfuscation on a Theoretical Basis and Its Implementation. *TIEICE: IEICE Transactions on Communications/Electronics/Information and Systems*, 2003.
87. A. Oliveira. Techniques for the creation of digital watermarks in sequential circuit designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1101–1117, September 2001.
88. Optimatika. Open source Java Algorithms (oJAlgo). <http://ojalgo.org/>, 2010.
89. E. I. Oviedo. Control flow, data flow and program complexity. In *Proc. of COMPSAC '80*, pages 146–152, Chicago, 1980.
90. J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, and Y. Zhang. Experience with software watermarking. In *ACSAC*, pages 308–316, 2000.
91. J. Pappalardo. As military becomes more reliant on networks, vulnerabilities grow. *National Defense*, October 2005.
92. F. A. P. Petitcolas, R. J. Anderson, and M. G. Kuhn. Information hiding – a survey. *Proceedings of the IEEE*, 87(7):1062–1078, July 1999.
93. F. A.P. Petitcolas, R. J. Anderson, and M. G. Kuhn. Attacks on copyright marking systems. In *Information Hiding*, pages 218–238, 1998. [cite-seer.nj.nec.com/petitcolas98attacks.html](http://citeseer.nj.nec.com/petitcolas98attacks.html).
94. B. Pfitzmann. Information hiding terminology - results of an informal plenary meeting and additional proposals. In *Proceedings of the First International Workshop on Information Hiding*, pages 347–350, London, UK, 1996. Springer-Verlag.
95. J. Pieprzyk. Fingerprints for copyright software protection. In *Proceedings of the 3rd International Workshop on Information Hiding*, pages 178–190, 1999.
96. R. Pozo and B. Miller. Scimark 2.0. <http://math.nist.gov/scimark2/>, November 2007.
97. The Apache Jakarta Project. Byte Code Engineering Library (BCEL). <http://jakarta.apache.org/bcel/>, June 2006.
98. G. Qu and M. Potkonjak. Hiding signatures in graph coloring solutions. In *Information Hiding Workshop*, volume 1768 of *LNCS*, pages 348–367. Springer, 1999.
99. V. Vazirani R. Venkatesan and S. Sinha. A graph theoretical approach to software watermarking. In *4th Information Hiding Workshop*, volume 2713 of *LNCS*, pages 157–168. Springer, 2001.

100. Rajaat. Polymorphism. *29A Magazine*, 3, 1999.
101. G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, 1994.
102. R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.
103. A. Sabelfeld and A. Myers. Language-based information-flow security, 2003.
104. B. Shimanovsky, J. Feng, and M. Potkonjak. Hiding data in dna. In *Revised Papers from the 5th International Workshop on Information Hiding*, London, UK, 2003. Springer-Verlag.
105. P. K. Singh and A. Lakhota. Static verification of worm and virus behavior in binary executables using model checking. In *IAW*, pages 298–300. IEEE, 2003.
106. J. P. Stern, G. Hachez, F. Koeune, and J. Quisquater. Robust object watermarking: Application to code. In *Information Hiding*, pages 368–378, 1999.
107. P. Szor and P. Ferrie. Hunting for metamorphic. Symantec Security Response. White Paper., June 2003.
108. H. Tamada, M. Nakamura, A. Monden, and K. Matsumoto. Detecting the theft of programs using birthmarks. Information Science Technical Report NAIST-IS-TR2003014 ISSN 0919-9527, Graduate School of Information Science, Nara Institute of Science and Technology, Nov 2003.
109. The GNU Project. The gnu bash reference manual, version 4.0. <http://www.gnu.org/software/bash/manual/>, February, 13 2009.
110. The JDOM™ Project. JDOM 1.1.1. <http://www.jdom.org/>, July 2009.
111. F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.
112. I. Torunoglu and E. Charbon. Watermarking-based copyright protection of sequential functions, 1999.
113. I. Torunoglu and E. Charbon. Watermarking-based copyright protection of sequential functions. *IEEE JSSC*, 35(3):434–440, March 2000.
114. J. Tromp. The Fhourstones Benchmark (version 3.1). <http://homepages.cwi.nl/~tromp/c4/fhour.html>.
115. A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.
116. P. C. van Oorschot. Revisiting software protection. *Information Security, LNCS*, 2851:1–13, 2003.
117. H. P. van Vliet. Crema – the java obfuscator, 1996.
118. R. Venkatesan, V. Vazirani, and S. Sinha. A graph theoretic approach to software watermarking. *Lecture Notes in Computer Science*, 2137:157–??, 2001.
119. L. Wall. The Perl programming language. <http://www.perl.org/>.
120. C. Wang, J. Hill, J. Knight, and J. Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, University of Virginia, 12 2000.
121. J. Wang. Average-case computational complexity theory. In *Alan L. Selman, Editor, Complexity Theory Retrospective, In Honor of Juris Hartmanis on the Occasion of His Sixtieth Birthday, July 5, 1988*, volume 2. Springer, 1997.
122. H. Wee. On obfuscating point functions, 2005.
123. M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.
124. G. Winskel. *The Formal Semantics of Programming Languages. An Introduction*. Massachusetts Institute of Technology, 1993.

125. W. Zhu and C. Thomborson. Extraction in software watermarking. In Sviatoslav Voloshynovskiy, Jana Dittmann, and Jessica J. Fridrich, editors, *MM&Sec*, pages 175–181. ACM, 2006.
126. W. Zhu and C. Thomborson. Recognition in software watermarking. In *MCPS '06: Proceedings of the 4th ACM international workshop on Contents protection and security*, pages 29–36, New York, NY, USA, 2006. ACM.
127. W. Zhu, C. Thomborson, and F. Wang. Applications of homomorphic functions to software obfuscation. In Hsinchun Chen, Fei-Yue Wang, Christopher C. Yang, Daniel Dajun Zeng, Michael Chau, and Kuiyu Chang, editors, *WISI*, volume 3917 of *Lecture Notes in Computer Science*, pages 152–153. Springer, 2006.
128. W. Zhu, C. D. Thomborson, and F. Wang. A survey of software watermarking. In Paul B. Kantor, Gheorghe Muresan, Fred Roberts, Daniel Dajun Zeng, Fei-Yue Wang, Hsinchun Chen, and Ralph C. Merkle, editors, *ISI*, volume 3495 of *Lecture Notes in Computer Science*, pages 454–458. Springer, 2005.

Acknowledgements

As a PhD student, I have been brought up for three years in a darwinistic environment. My experience was troublesome, but not pointless. By working individually for long, I was forced to question and assess my talent, thereby gaining invaluable knowledge. To the many inquiries of mine, Saverio Miroddi and Adriano Vincenzi provided the sharpest answers. Giovanni Gallo, Francesco Stefanni and Margherita Zorzi were my peers in our personal quests. Alessandro Rigoni and Barbara Venturi's taste for music, art and literature soothed my mind over and over again.

Throughout my PhD I could meet several people that made my life rich. Roberto Giacobazzi, Isabella Mastroeni and Mila Dalla Preda were the members of my research team in the Dipartimento di Informatica at the Università degli Studi di Verona (Italy); they showed me how beautifully code obfuscation and software watermarking might fit in a formal framework based on abstract interpretation. Christian Collberg and Maurizio Gabbrielli accepted to review this thesis: their warm suggestions helped me a lot. Christian Collberg was also my tutor at the University of Arizona, during my three-months long stay in Tucson (Arizona); he paid me his best attention and helped me not to be disheartened. Chizanya Mpinja was my buddy on the road to San Diego (California); her friendship made me feel at home even if I were abroad. Several researchers and PhD students from all around the world let me enjoy their company; it was quietly inspirational to interact with them, especially with those in our common work space in Verona. Many students attended my Fondamenti dell'Informatica exercise sessions and office hours at the Università degli Studi di Verona in the 2008-2009 Academic Year: their patience and enthusiasm were precious to me and my work.

I owe my long-running education program to my parents, and their wish to grant my brother and me what they could not afford in their youth. My training has been influenced by Suor Maria Chiara Teresa del Padre Buono's spiritual progresses within the Serve di Maria Oblate Sacerdotali religious order in Verona. The Fondazione Segni Nuovi of Verona has guided me in the Catholic worldview, pointing out the necessity of striving not for the best society ever, which is going to pass away anyhow, but for a true notion of the Divine who, by his very nature, conveys the brightness any human endeavor needs.

Verona, May 20, 2010

Sommario Il software conserva la maggior parte del know-how che occorre per svilupparlo. Poiché oggi il software può essere facilmente duplicato e ridistribuito ovunque, il rischio che la proprietà intellettuale venga violata su scala globale è elevato. Una delle più interessanti soluzioni a questo problema è dotare il software di un watermark. Ai watermark si richiede non solo di certificare in modo univoco il proprietario del software, ma anche di essere resistenti e pervasivi. In questa tesi riformuliamo i concetti di robustezza e pervasività a partire dalla semantica delle tracce. Evidenziamo i cicli quali costrutti di programmazione pervasivi e introduciamo le trasformazioni di ciclo come mattone di costruzione per schemi di watermarking pervasivo. Passiamo in rassegna alcune fra tali trasformazioni, studiando i loro principi di base. Infine, sfruttiamo tali principi per costruire una tecnica di watermarking pervasivo. La robustezza rimane una difficile, quanto affascinante, questione ancora da risolvere.