An algebraic framework for reasoning about security

by

Solofomampionona Fortunat Rajaona

Thesis presented in partial fulfilment of the requirements for the degree of Master of Science in Mathematics in the Faculty of Science at Stellenbosch University

Department of Mathematics,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Supervisor: Prof. J.W. Sanders

January 2013

Declaration

Signature: [illegible] ....
S. F. Rajaona

Date: 2012/12/10 ....

Abstract

An algebraic framework for reasoning about security

S. F. Rajaona

Department of Mathematics,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Thesis: MSc (Maths)

January 2013

Stepwise development of a program using *refinement* [1, 9] ensures that the program correctly implements its requirements. The specification of a system is "refined" incrementally to derive an implementable program. The programming space includes both specifications and implementable code, and is ordered with the refinement relation which obeys some mathematical laws. Morgan proposed a modification of this "classical" refinement for systems where the confidentiality of some information is critical. Programs distinguish between "hidden" and "visible" variables and refinement has to bear some security requirement. First, we review refinement for classical programs and present Morgan's approach for *ignorance preserving* refinement. We introduce the *Shadow Semantics*, a programming model that captures essential properties of classical refinement while preserving the ignorance of hidden variables. The model invalidates some classical laws which do not preserve security while it satisfies new laws. Our approach will be algebraic, we propose algebraic laws to describe the properties of ignorance preserving refinement. Thus completing the laws proposed in [7, 8, 10, 12]. Moreover, we show that the laws are sound in the Shadow Semantics. Finally, following the approach of Hoare and He [4] for classical programs, we give a completeness result for the program algebra of ignorance preserving refinement.

Uittreksel

Algebraïese raamwerk vir die redenasie oor die veiligheid

("An algebraic framework for reasoning about security")

S. F. Rajaona

Departement Mathematik,
Universiteit van Stellenbosch,
Privaatsak X1, Matieland 7602, Suid Afrika.

Tesis: MSc (Wisk)

Januarie 2013

Stapsgewyse ontwikkeling van 'n program met behulp van verfyning verseker dat die program voldoen aan die vereistes. Die spesifikasie van 'n stelsel word geleidelik "verfyn"wat lei tot 'n implementeerbare kode, en word georden met ‘n verfyningsverhouding wat wiskundige wette gehoorsaam. Morgan stel 'n wysiging van hierdie klassieke verfyning voor vir stelsels waar die vertroulikheid van sekere inligting van kritieke belang is. Programme onderskei tussen "verborgeën "sigbare"veranderlikes en verfyning voldoen aan 'n paar sekuriteitsvereistes. Eers hersien ons verfyning vir klassieke programme en verduidelik Morgan se benadering tot onwetendheid behoud. Ons verduidelik die "Shadow Semantics", 'n programmeringsmodel wat die noodsaaklike eienskappe van klassieke verfyning omskryf terwyl dit die onwetendheid van verborge veranderlikes laat behoue bly. Die model voldoen nie aan n paar klassieke wette, wat nie sekuriteit laat behoue bly nie, en dit voldoen aan nuwe wette. Ons benadering sal algebraïese wees. Ons stel algebraïese wette voor om die eienskappe van onwetendheid behoudende verfyning te beskryf, wat dus die wette voorgestel in [7, 8, 10, 12] voltooi. Verder wys ons dat die wette konsekwent is in die ”Shadow Semantics”. Ten slotte, na aanleiding van die benadering in [4] vir klassieke programme, gee ons ’n volledigheidsresultaat vir die program algebra van onwetendheid behoudende verfyning.

Acknowledgements

I wish to express my sincere thanks to Prof. Jeff Sanders for his guidance and advices during my postgraduate studies, for his assistance and hints throughout this thesis and for his patience and his support in my academic and my personal life.

I would like to thank Prof. Barry Green, Director of the African Institute of Mathematical Sciences (AIMS)–South Africa and Prof. Rewitzky, Head of the Department of Mathematical Sciences of Stellenbosch University, who have made this project possible. I am grateful to the people who found AIMS–South Africa which supported financially my Master's study.

I thank the people at AIMS for their friendship and for their technical helps. I wish to thank Prof. McIver for giving me advices on the Shadow model.

I thank my parents and all my family in Madagascar for their constant support and encouragement. Special thanks to my wife Sophie for her patience, her love and for giving birth to our first son Mahefa during this project.

Dedications

To Mahefa,

Contents

List of Tables

Chapter 1

Introduction

Stepwise program development

Verification of software traditionally relies on simulation, testing and code review destined to find bugs. However these methods do not always guarantee total correctness. For safety or security critical systems, *formal verification* methods such as *Model Checking* and *Theorem Proving* provide mathematical proofs of correctness.

Stepwise program development using refinement [1, 9] is one approach of formal verification. Rather than proving the correctness of an existing program, refinement intends to build the program correctly. As for other formal verification methods, the first step is to write the specification in a specific mathematical language. Specifications and the code implementing them are written in the same language; they are all referred to as programs although perhaps abstract ones. This formalism enables us to treat them as mathematical objects.

Reduction of nondeterminism. Informally, a specification S is refined by an implementation I, denoted $(S \sqsubseteq I)$ if I can replace S correctly. For terminating programs this means that any output of I is also a possible output of S. The specification can be thought of as an equation and the implementation as a solution. Specifications might not be feasible, as equations might not have any solution, and they might be nondeterministic as equations might have several solutions. The main task of refinement is to reduce the nondeterminism in the specifications and to derive a deterministic code. We note that refinement does not capture the efficiency of the program in terms of time and space consumption; this can be done by choosing the most efficient between alternative refinements of the same specification. Typically matters of efficiency motivate refinement steps.

Formalisation. Having a formal programming language, the programs are given meaning by different mathematical concepts, called *semantics*. Dijkstra introduced the *Guarded Command language* [2] and described each program as a *Predicate Transformer*: a function that maps a postcondition to its weakest precondition. Algebraic laws based on this model have been established to allow refinement calculii, for instance by Back [1] and Morgan [9] where the languages are extended to include specifications. Other models include the relational model in which programs are viewed as binary relations between states.

Program Algebra. A totally algebraic approach to programming [3] states algebraic properties of the operators as axioms. These laws will give the meaning of the programs. However it is necessary that the laws are consistent, i.e. they do not contradict each other. This might be shown by proving them against one of the programming models described earlier. The program algebra of a small programming language will be presented in Chapter 2.

Security and refinement

Security. In addition to functional requirements (input-output for terminating programs), some systems might be subject to security requirements such as confidentiality, authentication or anonymity. We are particularly interested in noninterference: in a multiple agents system, certain information is concealed from certain agents. Such a requirement might be necessary for example in *distributed voting, anonymous auctions* or *private information retrieval.*

Secure refinement. Morgan proposed stepwise refinement to derive noninterference-aware programs [10]. In Chapter 3, we will see that this cannot be done in a straightforward way with classical refinement techniques. Morgan's approach was to define a programming model following the intuition of security requirement and the necessity of having reasonable properties. This model, called *Shadow semantics*, is also presented in Chapter 3.

An algebra for secure refinement. In practice, stepwise derivation of a program is done by using algebraic laws proven sound in a particular model. Stepwise derivation of security protocols has been done, for instance, by Morgan in 2006 [10] (*Dining Cryptographers Protocol*) and in 2009 [11] (*Oblivious Transfer Protocol*) and by McIver in 2009 [7] (*Private Information Retrieval*). We are interested in the algebraic laws that are used for deriving such protocols. In addition to those used by Morgan and McIver, additional laws are given in Chapter 4. They would help us to have clearer description of the algebraic properties of noninterference-aware programs. In the same chapter, we also give a normal form for programs and prove that our are complete in the Shadow semantics.

Chapter 2

Algebra of classical program refinement

This chapter focuses on the algebraic treatment of a subset of classical programs. The programs studied here are the classical analogue of the noninterference aware programs used by Morgan [10] which will be studied in Chapter 3 and Chapter 4. First, programs are described and their syntax is presented, then a list of algebraic laws characterising the program properties are given. Finally a subset of these laws is proven to be complete. We use a normal form approach based on the work of Hoare and He [4] which we will extend to include local blocks.

2.1 Notations

The following notations apply for this chapter

- x, y, z designate state variables,
- $\mathcal{T}$ designates the type of the variables,
- P, Q, R designate programs,
- e, f, g designate expressions,
- b designates a boolean expression,
- we use left-associating dot for application. For example $f.x.y$ means $(f.x).y$ or $f(x, y)$ by uncurrying.

2.2 Programs

The state of a program is represented by the values of the program variables $x_1 : \mathcal{T}_1, \ldots, x_n : \mathcal{T}_n$. These variables are often abbreviated by a list of variable x called the state space. The execution of a terminating program is characterised by the change of its state. This is controlled by assignments to the state variables, composition, conditional controls, repetition, ...

We will consider finite programs obtained by the following constructions

skip	No operation
$x := e$	Assignment
$P \fatsemi Q$	Sequential composition
$P \lhd b \rhd Q$	Conditional
$P \sqcap Q$	Nondeterministic choice
$[\![\, \mathbf{var}\, x : \mathcal{T} \cdot P \,]\!]$	Declaration of a local variable

Table 2.1: Syntax for terminating sequential programs with nondeterminism

These constitute the programming syntax for the classical programs considered here; iteration, recursion and procedures are not presented since they are not included in the security-aware programming language considered by Morgan in 2006 [10]. The program space yielded by the precedent constructions, together with iteration, recursion and procedures is part of the larger space of specifications which includes other constructs for instance guarded commands, angelic choice and miracles. We note that the programs considered are all terminating programs: for example, the evaluation of the expressions are assumed to terminate and give a unique value.

2.3 Program algebra

The properties of the primitive programs and program constructors in the syntax is described by algebraic laws. As is done in the "Laws of Programming" [3], these laws are presented here like axioms as for any other sets of axioms in Mathematics. The laws can be proved to be consistent and valid using program models such as the predicate transformers model or the relational model. The proofs of these classical laws are not covered here. However, the laws for noninterference preserving refinement in Chapter 4 will be proved using the Shadow model.

The first law concerns **skip** which designates the program that does not perform anything. It has no effect on the state variables. The sequential composition $P \fatsemi Q$ is the program that first executes program P and then executes program Q. The left and the right composition of a program P with **skip** is the program P itself, so **skip** is the identity for sequential composition.

Law 2.1.

$$\mathbf{skip} \fatsemi P \;=\; P \fatsemi \mathbf{skip} \;=\; P$$

Sequential composition is associative.

Law 2.2.

$$(P \fatsemi Q) \fatsemi R \;=\; P \fatsemi (Q \fatsemi R)$$

Nondeterministic choice $(P \sqcap Q)$ which is also known as demonic choice executes either program P or Q ; the choice between them is determined locally. This operator is idempotent, commutative and associative.

Law 2.3.

$$\begin{aligned} P \sqcap P &= P \\ P \sqcap Q &= Q \sqcap P \\ (P \sqcap Q) \sqcap R &= P \sqcap (Q \sqcap R) \end{aligned}$$

We will refer to these properties as basic properties of $\sqcap$; for instance, they allow us to write the nondeterministic choice of a (finite) set of programs $\mathcal{P}$ which is denoted $\bigsqcap \mathcal{P}$. In every occurrence of such program, it is assumed that the set $\mathcal{P}$ is non-empty.

The two following are distribution laws of nondeterministic choice over sequential composition. We do not have the second law for general computations but it holds for the particular case studied here as we only consider finite nondeterminism and terminating programs.

Law 2.4.

$$(P \sqcap Q) \fatsemi R \;=\; (P \fatsemi R) \sqcap (Q \fatsemi R)$$

Law 2.5.

$$R \fatsemi (P \sqcap Q) \;=\; (R \fatsemi P) \sqcap (R \fatsemi Q)$$

Conditional ($P \lhd b \rhd Q$) which is also written as P **if** b **else** Q executes program P if predicate b holds, otherwise executes program Q. Law 2.6 is straightforward, with Law 2.7, ensures that conditional distributes over finite nonempty nondeterminism.

Law 2.6.

$$P \lhd b \rhd Q \;=\; Q \lhd \neg b \rhd P$$

Law 2.7.

$$P \lhd b \rhd (Q \sqcap R) \;=\; (P \lhd b \rhd Q) \sqcap (P \lhd b \rhd R)$$

Assignment ($x := e$) where x is a list of variables and e is a list of expressions having the length of x, is executed by evaluating all the values of the expressions in the list e and assigning each value to the corresponding variable in x. It is assumed that the values of the variables in the list do not change until all the evaluations are complete.

Arbitrary assignment (or choice), denoted $x :\in E$ where E is a finite nonempty set of expressions, is sometimes used in the programs. It is not to be confused with generalised assignment (often denoted $x :\in b$) where x can take any value satisfying the boolean b. The latter is not part of the programming language in cases when b is **false** or allows an infinite choice. Generalised assignment lies in the specification space. For classical programs, choice can be written in terms of other programs as follows

Law 2.8.

$$x :\in E \;=\; \bigsqcap \big\{ e : E \bullet x := e \big\}$$

Assigning the value of a variable to itself does not change anything, thus it is **skip**. An assignment to a single variable can be rewritten as an assignment to multiple variables by adding identity assignment. In such an assignment, the variables assigned can be permuted as long as the corresponding expressions are subject to the same permutation.

Law 2.9.

$$(x := x) \;=\; \textbf{skip}$$

Law 2.10.

$$(x := e) \;=\; (x, y := e, y)$$

Law 2.11.

$$(x, y, z := e, f, g) \;=\; (x, z, y := e, g, f)$$

A conditional assignment is equal to an assignment to a conditional expression. Conditional expressions are assumed to be part of the programming language and are defined as follows

$$\begin{aligned} e \lhd\; b \;\rhd f &= e \quad \textbf{if } b \\ &= f \quad \textbf{if } \neg b \end{aligned}$$

Law 2.12.

$$(x := e \lhd\; b \;\rhd x := f) \;=\; (x := e \lhd\; b \;\rhd f)$$

The sequential composition of two assignments to a variable are combined to a single assignment by substituting the occurence of the variable in the rhs of the second assignment by the value that has been assigned to it. This law is straight-forward for classical programs but that is not the case, as we will see, for the security-aware programming language.

Law 2.13.

$$(x := e \,;\, x := f) \;=\; x := f.e$$

Two assignments to a variable are equal when the expressions assigned to them are equal.

Law 2.14.

$$(x := e) \;=\; (x := f) \;\textbf{iff}\; e = f$$

A local block $|[\,\mathbf{var}\, x : \mathcal{T} \bullet P\,]|$ which is often written as **begin var** $x : \mathcal{T} \bullet P$ **end** declares a local variable x for use in P. The brackets delimits the scope of the declared variable: its value is accessible only inside its scope. Such variables are called local variables as opposed to global ones and are used as intermediaries in the program.

If the variable declared is not free in the program inside its scope, then the local block containing that program is equal to the program itself

Law 2.15. *If x is not free in P then*

$$|[\,\mathbf{var}\, x : \mathcal{T} \bullet P\,]| = \ P$$

When the program inside the scope of a local block assigns only to the local variable then the local block is equal to **skip**. Such an assignment changes no global variable and is equivalent to identity.

Law 2.16.

$$|[\,\mathbf{var}\, x : \mathcal{T} \bullet x := e\,]| = \ \mathbf{skip}$$

More generally,

Law 2.17. *If x is not free in f*

$$|[\,\mathbf{var}\, x : \mathcal{T} \bullet x, y := e, f\,]| = \ (y := f)$$

Nondeterministic choice distributes a local block.

Law 2.18.

$$|[\,\mathbf{var}\, x : \mathcal{T} \bullet P \sqcap Q\,]| = |[\,\mathbf{var}\, x : \mathcal{T} \bullet P\,]| \sqcap |[\,\mathbf{var}\, x : \mathcal{T} \bullet Q\,]|$$

The following law says that a local variable is initialised arbitrarily after its declaration.

Law 2.19.

$$|[\,\mathbf{var}\, x : \mathcal{V} \bullet P\,]| = \bigsqcap \{ e : \mathcal{V} \bullet \ |[\,\mathbf{var}\, x : \mathcal{T} \bullet x := e \,⨾\, P\,]| \ \}$$

The program space is ordered by the refinement relation denoted $\sqsubseteq$.

Definition 2.1 (Classical refinement)**.** A program P is refined by Q when Q is at least as deterministic as P:

$$P \sqsubseteq Q \quad \equiv \quad P \sqcap Q = P$$

Following the basic properties of $\sqcap$ (Law 2.3), the relation $\sqsubseteq$ is reflexive, transitive and antisymmetric. This relation is a lattice ordering on the space of specifications and its restriction on the program space is a partial order.

Law 2.20.

$$P \sqsubseteq P$$

Law 2.21.

$$\begin{pmatrix} P \sqsubseteq Q \\ Q \sqsubseteq R \end{pmatrix} \Rightarrow P \sqsubseteq R$$

Law 2.22.

$$\begin{pmatrix} P \sqsubseteq Q \\ Q \sqsubseteq P \end{pmatrix} \Rightarrow P = Q$$

All the programming operators given here are monotonic with respect to the refinement order. The following laws illustrate this. They are mostly important when developing program from specification: refinement of larger programs can be derived from refinement of their smaller parts.

Law 2.23. *If $P \sqsubseteq Q$ then*

$$P \,;\, R \ \sqsubseteq \ Q \,;\, R$$

Law 2.24. *If $P \sqsubseteq Q$ then*

$$P \sqcap R \ \sqsubseteq \ Q \sqcap R$$

The set of laws that we have given in the previous section is not exhaustive; other laws are listed elsewhere [3, 4]. We have given laws that illustrate some properties of the programming language. A subset of these laws has been proved to be complete [4] i.e any inequation between programs can be proved using these laws.

2.4 Reduction to normal form

Hoare and He [4] proposed a normal form for more general programs apart from those in the programming language. For the programming space studied here the corresponding normal form is a finite nondeterministic choice of assignment

$$\bigsqcap\{e : E \bullet x := e\},$$

where x is the vector of all variables. For a singleton $E = \{e\}$, the nondeterministic choice reduces to an assignment $x := e$.

To prove that any program can be reduced to this normal form, it is sufficient to prove that the primitive programs (in the syntax) can be reduced into normal form and that combinations of programs in normal form using the connectors (in the syntax) yield programs also reducible to normal form. We illustrate the reduction of programs to normal form as done by Hoare and He, adding the elimination of local blocks.

1. **skip** is the identity assignment which is in normal form (Law 2.9).

2. Any assignment can be written as a total assignment (assignment to all variables) in a uniform order by using Law 2.10 and Law 2.11, thus reducing it to an assignment to the variable vector x which is in normal form.

3. The following equations illustrate that a conditional between two normal forms yields a normal form.

$$\begin{aligned}
& \left(\bigsqcap\{e : E \bullet x := e\}\right) \lhd\ b\ \rhd \left(\bigsqcap\{f : F \bullet x := f\}\right) \\
= \quad & \text{Law 2.6 and Law 2.7} \\
& \bigsqcap\left\{e : E;\ f : F \bullet x := e \lhd\ b\ \rhd x := f\right\} \\
= \quad & \text{Law 2.12} \\
& \bigsqcap\left\{e : E;\ f : F \bullet x := e \lhd\ b\ \rhd f\right\}
\end{aligned}$$

4. The following equations illustrate the elimination of sequential composition between normal forms

$$\begin{aligned}
& \left(\bigsqcap\{e : E \bullet x := e\}\right) \mathbin{;} \left(\bigsqcap\{f : F \bullet x := f\}\right) \\
= \quad & \text{Law 2.4 and Law 2.5} \\
& \bigsqcap\{e : E;\ f : F \bullet x := e \mathbin{;} x := f\}
\end{aligned}$$

$$= \qquad \text{Law 2.13}$$
$$\bigsqcap\{e : E;\ f : F \bullet x := f.e\}$$

5. Nondeterministic choice between normal forms yields a normal form because of the properties of $\sqcap$

$$\left(\bigsqcap\{e : E \bullet x := e\}\right) \sqcap \left(\bigsqcap\{e : E \bullet x := e\}\right)$$
$$= \qquad \text{(Law 2.3)}$$
$$\bigsqcap\{g : E \cup F \bullet x := g\}$$

6. The following illustrates the elimination of local blocks

$$|[\,\mathbf{var}\,x : \mathcal{V} \bullet \left(\bigsqcap\{f : F \bullet x := f\}\right)]|$$
$$= \qquad \text{Law 2.19}$$
$$\bigsqcap\left\{e : \mathcal{V} \bullet\ |[\,\mathbf{var}\,x : \mathcal{V} \bullet x := e \,\S\, \left(\bigsqcap\{f : F \bullet y := f\}\right)]|\ \right\}$$
$$= \qquad \text{Law 2.5}$$
$$\bigsqcap\left\{e : \mathcal{V} \bullet\ |[\,\mathbf{var}\,x : \mathcal{V} \bullet \bigsqcap\{f : F \bullet x := e \,\S\, y := f\}]|\ \right\}$$
$$= \qquad \text{Law 2.13}$$
$$\bigsqcap\left\{e : \mathcal{V} \bullet\ |[\,\mathbf{var}\,x : \mathcal{V} \bullet \bigsqcap\{f : F \bullet x, y := e, f.e\}]|\ \right\}$$
$$= \qquad \text{Law 2.18}$$
$$\bigsqcap\left\{e : \mathcal{V} \bullet \bigsqcap\{f : F \bullet\ |[\,\mathbf{var}\,x : \mathcal{V} \bullet x, y := e, f.e]|\}\right\}$$
$$= \qquad \text{Law 2.17}$$
$$\bigsqcap\left\{e : \mathcal{V} \bullet \bigsqcap\{f : F \bullet\ |[\,\mathbf{var}\,x : \mathcal{V} \bullet y := f.e]|\}\right\}$$
$$= \qquad \text{Law 2.3 and Law 2.15}$$
$$\bigsqcap\left\{e : \mathcal{V};\ f : F \bullet y := f.e\right\}$$

The following laws permit the comparison of programs in normal form. The first of them results directly from the definition of $\sqsubseteq$ and the basic properties of $\sqcap$ (Law 2.3).

Law 2.25.

$$P \sqsubseteq \bigsqcap \mathcal{Q}\ \mathbf{iff}\ \forall\, Q : \mathcal{Q} \bullet P \sqsubseteq Q$$

Law 2.26.

$$\bigsqcap\{e : E \bullet x := e\} \sqsubseteq x := f \textbf{ iff } f \in E$$

These two laws achieve the completeness of the laws given for the subset of programs studied here. In the following, the programs will be modified to consider secrecy, thus the laws given in this chapter will be reconsidered but the same approach of normal form will be used in Chapter 4.

Chapter 3

Noninterference-preserving refinement

This chapter retraces Morgan's approach (introduced in [10]) of using refinement techniques for reasoning about noninterference. First we discuss what were his objectives and what assumptions he has made. Then we will look at some of the consequences on the algebraic laws and the adjustments on the programming syntax. Finally, we present Morgan's programming model that satisfies the objectives: the Shadow Semantics.

We will change slightly our notation from this chapter to be more convenient when translating the programs into their semantics.

- as in [12] the program variables $\mathsf{x}, \mathsf{v}, \mathsf{w}, \mathsf{h}$ will be written in $\mathsf{Sans\ Serif}$,
- v, w, h, k designate the values that those variable may assume,
- H, K designate set of values,
- e, f designate expressions,
- E, F designate set-valued expressions,
- $\mathcal{V}, \mathcal{H}, \mathcal{K}$ designate types of the program variables,
- $\mathcal{P}, \mathcal{Q}$ designate (finite) sets of programs,
- $E \times F$ designate the Cartesian product of the sets E and F,
- $\mathsf{Proj}_i.E$ designate the projection of E onto its i-th component, sometimes shortened to E_i,

- for a quantifier $\mathbf{Q}$ we use $(\mathbf{Q}x : \mathcal{T} \mid r \bullet e.x)$ to define a quantification over the elements of $\mathcal{T}$ satisfying r,
- we use $\{x : \mathcal{T} \mid r \bullet e.x\}$ to define the set of elements of the form $e.x$ for x in $\mathcal{T}$ satisfying r. For example $\{x : \mathbb{N}^+ \mid x \bmod 2 = 0 \bullet \frac{1}{x}\} = \{\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ldots\}$.

3.1 Assumptions and objectives

Morgan [10, 11] assumes that the program state is partitioned into *hidden* and *visible* variables. The attacker is assumed not to infer the value of the hidden variables whereas he can monitor the value of the visible ones. In addition, he is assumed to have knowledge of the source code. Using traditional refinement techniques for program development while preserving secrecy of the hidden variable rise some difficulties like the "Refinement Paradox" : assuming a hidden variable h of boolean type, the refinement $\mathsf{h} :\in \{0, 1\} \sqsubseteq \mathsf{h} := 0$ is classically valid but not secure: an execution of the second program will leak the value of h while the first one keeps it hidden. Morgan assumed that secure refinement does not allow such refinement.

Morgan requires that refinement holds only if *an implementation is no less secure than its specification*: an implementation can leak the value of a hidden variable only if the specification can do so. Thus the classical definition of refinement, decrease of nondeterminism, is strengthened with the preserving of ignorance (of the hidden variable). As observed in Chapter 1, the stepwise development of classical programs using refinement relies on the monotonicity of program constructors with respect to the refinement order; as this allows us to reason about small parts of a program separately. Morgan listed monotonicity as one of the desirable properties that should be preserved for secure refinement. In addition: if some parts of a program involve only visible variables, it should be possible to use classical refinement to reason about them; if the classical laws do not involve security (when they involve only sequential composition, nondeterministic choice, identity without explicit assignment or conditional) then they should remain valid.

These algebraic properties lead to make further assumptions. Using "gendaken" experiments, Morgan proved that it is necessary to assume a strong attacker: not only can he monitor the value of the visible variable but also he can observe the program flow; for instance, he knows which branch of a nondeterministic choice or conditional has been taken. Moreover, he has *perfect recall*: he can remember any value that was taken by the visible variable, even already overwritten.

$\mathsf{x} :\in E$	Choose
$\lvert[\, \mathbf{vis}\, \mathsf{v} : \mathcal{V} \bullet P \,]\rvert$	Declaration of a visible variable
$\lvert[\, \mathbf{hid}\, \mathsf{h} : \mathcal{H} \bullet P \,]\rvert$	Declaration of a hidden variable
$\mathbf{reveal}\ e$	Publish an expression
$\langle\!\langle P \rangle\!\rangle$	Atomic block

Table 3.1: Syntax of the programming language, in addition to Table 2.2. In [10] and [11] Morgan uses the brackets $\{\{\cdot\}\}$ to denote atomic blocks

3.2 Program syntax and examples

Some adjustments are made in the programming syntax (as seen in Table 3.2). We will use x when the context does not distinguish between visible and hidden variables.

Local blocks

The declaration of local variables have to distinguish between visible and hidden variables. Thus, local blocks are of two types: the local blocks introducing a visible variable ($\lvert[\, \mathbf{vis} \cdot]\rvert$) and those introducing a hidden variable ($\lvert[\, \mathbf{hid} \cdot]\rvert$). When we do not distinguish between them, we will use $\lvert[\, \mathbf{dec} \cdot]\rvert$.

McIver [7] allows multiple agents view of the program, thus having the declaration of variables indexed from the point of view of each agent (making use of $\mathbf{vis}_{\mathsf{list}}$ and $\mathbf{hid}_{\mathsf{list}}$ where list is a list of agents). However, for this study we will adopt a single attacker point of view as is the case in [10].

Atomic choice

In the classical setting, the command choose $:\in$ is syntactic sugar for a nondeterministic choice of assignment. The equation

$$\mathsf{x} :\in E = \bigsqcap\{e : E \bullet \mathsf{x} := e\} \tag{3.2.1}$$

allows the choice operator $:\in$ to be discarded from the program syntax. However, for the security and refinement programming language, the command choose is defined to execute atomically and differs from a nondeterministic choice of assignment.

Thus, two kinds of nondeterminism are considered. In a *composite nondeterministic choice* (which we will refer to as nondeterministic choice, denoted $\sqcap$,) the attacker can observe the execution: it knows which branch of the nondeterministic choice has been taken. In an *atomic nondeterministic choice* (which we will refer to as choice, denoted $:\in$), the attacker cannot see how the choice was resolved. Thus, the program choose $\mathsf{x} :\in E$ differs from the nondeterministic choice of assignment as the one in the rhs of (3.2.1). The nondeterministic choice operator $\sqcap$ expresses nondeterminism between the execution of some fragments of programs (atomic programs or larger programs).

An assignment is a particular case of choice where E is a singleton

$$\mathsf{x} := e \;=\; \mathsf{x} :\in \{e\}; \tag{3.2.2}$$

however, assignment and choice are both part of the programming language. For classical programs, multiple assignment was defined and denoted by

$$\mathsf{x}, \mathsf{y}, \mathsf{z} := e, f, g$$

where each variable gets the value of the expression which is in the same position as the variable.

For the operator choose $:\in$, the rhs will be a set of tuples rather than a tuple of sets. This convention is to allow a multiple choice from a set of vectors

$$\mathsf{x}, \mathsf{y}, \mathsf{z} := \{(e, f, g) : E\}$$

as this would capture cases of the expressions assigned which are dependent on each other, such as

$$\mathsf{x}, \mathsf{y}, \mathsf{z} := \{n : [1, N] \cdot (n, 2n, 3n)\}$$

which cannot be written as $\mathsf{x}, \mathsf{y}, \mathsf{z} :\in E, F, G$

Reveal

The command **reveal** is one particularity of the programming language: the program **reveal** e publishes an expression e, which might leak some information about h without changing neither v nor h.

Atomic blocks

Atomicity is introduced to limit the capability of the attacker. The program $\langle\langle P \rangle\rangle$ designates the atomic execution of program P. The attacker is assumed not to see the execution inside atomic blocks but only between them.

Example 3.1 (Refinement Paradox)**.** As we have observed earlier, the traditional aspect of refinement which is the decrease of (composite) nondeterminism does not always apply. For a hidden boolean variable h:

$$\mathsf{h} \in \{0,1\} \not\sqsubseteq \mathsf{h} := 0.$$

Example 3.2 (Perfect recall [10])**.** As already observed also, the attacker can keep track of the value assigned to a visible variable even after overwriting it:

$$(\mathsf{v} := \mathsf{h} \fatsemi \mathsf{v} := 0) \not\sqsubseteq \mathsf{v} := 0.$$

Example 3.3. [7] The execution of conditional choices reveals the value of the conditional:

$$(\mathbf{skip} \lhd\ e.\mathsf{v}.\mathsf{h}\ \rhd \mathbf{skip}) \not\sqsubseteq \mathbf{skip}.$$

Example 3.4. [7] For a hidden boolean variable h

$$\mathsf{h} :\in \{0,1\} \not\sqsubseteq \mathsf{h} := 0 \sqcap \mathsf{h} := 1.$$

- in the composite case, afterwards the adversary knows which of atoms $\mathsf{h} := 0$ or $\mathsf{h} := 1$ was executed, and thus knows the value of h too; yet
- in the atomic case, afterwards he knows only that the effect was to set h to 0 or to 1, and thus knows only that $\mathsf{h} \in \{0,1\}$.

If h is replaced by a visible variable v refinement holds in 3.4. As v is visible, by observing its final value, the attacker would know which branch of the nondeterministic choice has been taken. Thus we have

Example 3.5. For a visible boolean variable v

$$\mathsf{v} :\in \{0,1\} = \mathsf{v} := 0 \sqcap \mathsf{v} := 1.$$

Therefore, in this case the atomic choice is an explicit choice.

However, the following example shows that an atomic choice to a visible variable is not always an explicit choice.

Example 3.6. For v, h visible and hidden variables of boolean type

$$\mathsf{h} :\in \{0,1\} \fatsemi \mathsf{v} :\in \{\mathsf{h} \oplus 1, \mathsf{h} \oplus 0\} \quad \not\sqsubseteq \quad \mathsf{h} :\in \{0,1\} \fatsemi \mathsf{v} := (\mathsf{h} \oplus 1) \sqcap \mathsf{v} := (\mathsf{h} \oplus 0)$$

where $\oplus$ designates exclusive-or.

- in the composite case, afterwards the adversary knows which of atoms $\mathsf{v} := (\mathsf{h} \oplus 1)$ or $\mathsf{v} := (\mathsf{h} \oplus 0)$ was executed. He also knows the value assigned to the visible variable v, then can deduce h by computing $\mathsf{h} = (\mathsf{v} \oplus b)$ if the branch $\mathsf{v} := (\mathsf{h} \oplus b)$ was executed.

- in the atomic case, afterwards he knows that the effect was to set v to the value of $(\mathsf{h}\oplus 0)$ or to $(\mathsf{h}\oplus 1)$ i.e that v is 0 or 1. He also knows value b assigned to v, he cannot infer whether b was computed from $\mathsf{h} \oplus 1$ or $\mathsf{h} \oplus 0$. The two later expressions can all give b for an appropriated value of h. This is called the *Key-Complete Condition* of $\oplus$ [10]. Thus, the attacker knows only the value of v (and that $\mathsf{v} \in \{\mathsf{h} \oplus 0, \mathsf{h} \oplus 1\}$ which gives no more information).

Clearly, the equation (3.2.1) does not hold in general for both hidden and visible variables. The second example has shown that it holds for visible variables in some cases. In fact the second example is a particular case of *visible-only reasoning.*

It is necessary to have a more formal way to explain these refinements. In the following, we introduce Morgan's Shadow model to serve that purpose.

3.3 The Shadow model

The attacker is assumed to have the capability of monitoring the execution of the program and the value of the visible variable after each atomic step. Thus, at each state of the program, the attacker knows the whole path that was executed with all the successive values of the visible variable v. However, rather than characterising the state as the history of executed commands and associated values of v, Morgan introduced a set-valued variable H called the "Shadow". It records the set of possible values of the hidden variable h for all possible executions of a program yielding the same visible variable. The state of the program is thus determined by triples $(v, h, H) : \mathcal{V}\times\mathcal{H}\times\mathbb{P}\mathcal{H}$ corresponding to the variables $(\mathsf{v}, \mathsf{h}, \mathsf{H})$. Each triple satisfies $h \in H$ as the actual value of h must be one of its possible values in the Shadow.

Morgan [10] provides an operational semantics which translates v, h-programs (noninterference aware) into classical programs with $\mathsf{v}, \mathsf{h}, \mathsf{H}$. He also gives a predicate transformer model which avoid this translation and where the logic used is the first order logic augmented with a modality characterising the knowledge of the attacker. However, we will adopt the program semantics which was given by McIver in 2009 [7]: programs are modeled as $\mathcal{V} \to \mathcal{H} \to \mathbb{P}.\mathcal{H} \to (\mathcal{V}\times\mathcal{H}\times\mathbb{P}.\mathcal{H})$. We will refer to this model as the Shadow Semantics. Table 3.3 summarises the

Program P	Semantics $[\![P]\!].v.h.H$
skip	$\{(v,h,H)\}$
reveal $e.\mathsf{v}.\mathsf{h}$	$\{(v,h,\{k:H \mid e.v.k = e.v.h\})\}$
$\mathsf{v} := e.\mathsf{v}.\mathsf{h}$	$\{(e.v.h,h,\{k:H \mid e.v.k = e.v.h\})\}$
$\mathsf{v} :\in F.\mathsf{v}.\mathsf{h}$	$\{v' : F.v.h \bullet (v',h,\{k:H \mid v' \in F.v.k\})\}$
$\mathsf{h} := e.\mathsf{v}.\mathsf{h}$	$\{(v,e.v.h,\{k:H \bullet e.v.k\})\}$
$\mathsf{h} :\in F.\mathsf{v}.\mathsf{h}$	$\{h' : F.v.h \bullet (v,h',\{k:H;\ k' : F.v.k \bullet k'\})\}$
$\langle\!\langle P \rangle\!\rangle$	$\mathsf{addShadow}.$(“classical semantics of P”)
$P_1 \mathbin{;} P_2$	$\mathsf{lift}.[\![P_2]\!].([\![P_1]\!].v.h.H)$
$P_1 \sqcap P_2$	$[\![P_1]\!].v.h.H \cup [\![P_2]\!].v.h.H$
$P_t \lhd b.\mathsf{v}.\mathsf{h} \rhd P_f$	$[\![P_t]\!].v.h.\{k:H \mid b.v.k\} \cup [\![P_f]\!].v.h.\{k:H \mid \neg b.v.k\}$

Table 3.2: Semantics of programs without loops (from [7]). The function $\mathsf{lift}.[\![P_2]\!]$ applies $[\![P_2]\!]$ to all triples in its set-valued argument, un-Currying each time, and then takes the union of all results.

Program P	Semantics $[\![P]\!].v.h.H$
$|[\,\textbf{vis}\,\mathsf{w} : \mathcal{W} \bullet P\,]|$	$\mathsf{Proj}_{(v,h,H)}.\bigcup\{w:\mathcal{W} \bullet ([\![P]\!].v.w.h.H)\}$
$|[\,\textbf{hid}\,\mathsf{k} : \mathcal{K} \bullet P\,]|$	$\mathsf{Proj}_{(v,h,H)}.\bigcup\{k:\mathcal{K} \bullet ([\![P]\!].v.h.k.(H \times \mathcal{K}))\}$
$(\mathsf{v},\mathsf{h}) :\in E$	$\{(v',h') : E.v.h \bullet (v',h',\{k:H;\ k':\mathcal{H} \mid E.v.k \ni (v',k') \bullet k'\})\}$

Table 3.3: Semantics of local blocks and simultaneous choice of a hidden and a visible variable

semantics of the programming constructs. McIver and Morgan gave an extension of this semantics that includes looping programs in 2011 [8].

We give an extension of the Shadow Semantics [7] to include the semantics of local blocks and simultaneous choice (Table 3.3). The following shows how the semantics of choose a couple (v,h) reduces to the semantics of choose a visible v when the hidden variable h is not assigned (which is in the present context equivalent to assigning it to itself or **skip**).

$$
\begin{aligned}
& [\![(\mathsf{v},\mathsf{h}) :\in E.\mathsf{v}.\mathsf{h}\times\{\mathsf{h}\}]\!].v.h.H \\
= & \qquad \text{semantics of simultaneous choice} \\
& \Big\{(v',h') : (E.v.h,h) \bullet \Big(v',h',\big\{k : H;\ k' : \mathcal{H} \mid (v',k') \in E.v.k\times\{k\} \bullet k'\big\}\Big)\Big\} \\
= & \qquad \text{calculus} \\
& \Big\{(v',h') : (E.v.h,h) \bullet \Big(v',h',\big\{k : H \mid (v',k) \in E.v.k\times\{k\}\big\}\Big)\Big\} \\
= & \qquad \text{calculus} \\
& \Big\{v' : E.v.h \bullet \Big(v',h,\big\{k : H \mid v' \in E.v.k\big\}\Big)\Big\} \\
= & \qquad \text{semantics of} :\in \text{for visible} \\
& [\![\mathsf{v} :\in E.\mathsf{v}.\mathsf{h}]\!].v.h.H
\end{aligned}
$$

Similarly, the semantics of choose a couple (v,h) reduces to the semantics of choose a hidden h when the visible variable v is not assigned.

$$
\begin{aligned}
& [\![(\mathsf{v},\mathsf{h}) :\in \{\mathsf{v}\}\times F.\mathsf{v}.\mathsf{h}]\!].v.h.H \\
= & \qquad \text{semantics of} :\in \text{for } (v,h) \\
& \Big\{(v',h') : (v,F.v.h) \bullet \Big(v',h',\big\{k : H;\ k' : \mathcal{H} \mid (v',k') \in (v,F.v.k) \bullet k'\big\}\Big)\Big\} \\
= & \qquad \text{calculus} \\
& \Big\{h' : F.v.h \bullet \Big(v,h',\big\{k : H;\ k' : F.v.k \bullet k'\big\}\Big)\Big\} \\
= & \qquad \text{semantics of} :\in \text{for hidden} \\
& [\![\mathsf{h} :\in F.\mathsf{v}.\mathsf{h}]\!].v.h.H
\end{aligned}
$$

The semantics of atomic blocks requires the definition of $\mathsf{addShadow}$: this function maps the classical semantics ($[\![P]\!]_{\mathsf{cl}}.v.k$) of a program to its shadow-enhanced semantics.

Definition 3.1 (Atomic Shadow Semantics).

$$\mathsf{addShadow}.P.v.h.H =$$
$$\left\{(v',h',H') : \mathcal{V}\times\mathcal{H}\times\mathbb{P}\mathcal{H} \mid \left(\begin{array}{l} [\![P]\!]_{\mathsf{cl}}.v.h \ni (v',h') \\ H' = \left\{k' : \mathcal{H} \mid (\exists\, k : H \bullet [\![P]\!]_{\mathsf{cl}}.v.k \ni (v',k'))\right\} \end{array}\right)\right\}$$

Refinement in the Shadow model

Recall that in the context of noninterference, an implementation I refines its specification S, denoted $S \sqsubseteq I$) I is more deterministic and it preserves ignorance (of the hidden variable). The first aspect which is the classical aspect means that from any initial state, any possible (visible) outcome of I is also a possible (visible) outcome of S. However the second aspect means that the Shadow does not shrink with refinement. As well described by Morgan in 2012 [12], the definition of refinement in the Shadow model should be stronger.

Indeed, the classical aspect should also be applied to the hidden variable: for instance an assignment $\mathsf{v} := \mathsf{h}$ occurring afterwards justifies this, as it would make the hidden outcome visible. Also, the second aspect has to be strengthened: the Shadow should not shrink with refinement for every possible outcome of v by the implementation I. For instance the possible occurrence of a conditional like ($\mathbf{skip} \lhd \mathsf{v} = e \rhd \mathsf{h} :\in \mathcal{H}$) afterwards forces to consider relation between Shadows for each value v that v can take (and hence for each value that h can take by combining $\mathsf{v} := \mathsf{h}$ and ($\mathbf{skip} \lhd \mathsf{v} = e \rhd \mathsf{h} :\in \mathcal{H}$) afterwards). Further, one can consider a conditional both on v and h. Therefore, the relation between Shadows must be considered for every pair of outcomes (v', h').

Definition 3.2 (Refinement). For programs $P_{\{1,2\}}$, P_1 is securely refined by P_2 and written $P_1 \sqsubseteq P_2$ when for all v, h, H that satisfy $h \in H$

$$\forall (v',h',H_2') : [\![P_2]\!].v.h.H \bullet (\exists\, H_1 : \mathbb{P}\mathcal{H} \mid H_1' \subseteq H_2' \bullet (v',h',H_1') \in [\![P_1]\!].v.h.H)$$

In one of his papers [12], Morgan retraces the building process of the Shadow model and the definition of *Shadow* refinement (Definition 3.2). He shows that this definition meet the following objectives:

- If $S \sqsubseteq I$, for any programming context $\mathcal{C}$ determined by the syntax (excluding atomic block): any value of the visible variable that $\mathcal{C}(I)$ can output can also produced by $\mathcal{C}(S)$, if $\mathcal{C}(I)$ can leak the value of h then so can $\mathcal{C}(S)$

- If $S \sqsubseteq I$, then for any context $\mathcal{C}$ (excluding atomic block) $\mathcal{C}(S) \sqsubseteq \mathcal{C}(I)$

These legitimise the choice of the definition of refinement with respect to the desired algebraic properties namely monotonicity of contexts with respect to $\sqsubseteq$ and the preservation of certain class of classical laws as stated in the

Theorem 3.1 (Transfer Principle [12]). *Let $\mathcal{C}_1(X, Y, \ldots, Z)$ and $\mathcal{C}_2(X, Y, \ldots, Z)$ be two contexts involving only visible variables,and satisfying the classical refinement ($\sqsubseteq_c$)*

$$C_1(X, Y, \ldots, Z) \sqsubseteq_c \mathcal{C}_2(X, Y, \ldots, Z)$$

for all classical program fragments $X, Y, \ldots, Z$ — even those in which further visible variables not appearing already in $C_{\{1,2\}}$ might occur. Then the secure refinement

$$C_1(X, Y, \ldots, Z) \sqsubseteq \mathcal{C}_2(X;\ Y;\ \ldots;\ Z)$$

holds for all program fragments $X, Y, \ldots, Z$ over both visible and hidden variables

Now we can come back to some of to an earlier example and use the Shadow refinement to check it. Below the program are the associated possible states.

Example 3.4. [12]

$$\begin{array}{ccc} \mathsf{h} :\in \{0,1\} & \not\sqsubseteq & \mathsf{h} := 0 \sqcap \mathsf{h} := 1 \\ (v,0,\{0,1\}),(v,1,\{0,1\}) & & (v,0,\{0\}),(v,1,\{1\}) \end{array}$$

The shadow in the rhs associated with $(v, 0)$ is $\{0\}$ while the only shadow associated with $(v, 0)$ in the lhs is $\{0, 1\} \not\subseteq \{0\}$, thus the refinement does not hold.

Example 3.5. [12]

$$\begin{array}{ccc} \mathsf{h} :\in \{0,1\} \sqcap \mathsf{h} :\in \{2,3\} & \not\sqsubseteq & \mathsf{h} :\in \{0,1,2\} \\ (v,0,\{0,1\}),(v,1,\{0,1\}) & & (v,0,\{0,1,2\}),(v,1,\{0,1,2\}) \\ (v,2,\{2,3\}),(v,3,\{0,1\}) & & (v,2,\{0,1,2\}) \end{array}$$

The shadow in the rhs associated with $(v, 2)$ is $\{0, 1, 2\}$ while the only shadow associated with $(v, 2)$ in the lhs is $\{2, 3\} \not\subseteq \{0, 1, 2\}$, thus the refinement does not hold.

The Shadow Semantics is a model that validates secure refinement while invalidating the insecure ones. Yet, actual program development makes use of algebraic reasoning: refinement steps are justified by algebraic laws. However, these laws must be sound in the model. The following chapter concerns the algebra of security-aware programs.

Chapter 4

Algebra of ignorance-sensitive programs

In their papers [7, 8, 10, 11], Morgan and McIver list several laws to support reasoning about ignorance-preserving refinement. These laws were used in the development of some security protocols such as the Dining Cryptographers, Oblivious Transfer protocol, Millionaire's protocol and the more recent Three Judges Protocol. However, the laws did not cover all the operators and need to be completed. Our goal is first to add new laws to give more description to the programming language. Then, to give a complete set of these laws: any ignorance-preserving refinement between two programs can be proved using only the laws in the set. To achieve this completeness result, we will use the technique of normal form as used for classical programs (as detailed in Chapter 2). But, in contrast to that chapter, the laws will be proved sound using the Shadow Model of Chapter 3.

4.1 Definitions and notations

The notations in Chapter 3 are still used here. In addition,

- for a set of vectors E, E_i designates the projection on the i-th component; $E_{i,j}$ designates the projection on the pair of components i and j.

4.2 Algebraic laws

We are going to list laws for security-aware programs and explain them. They are all sound in the Shadow semantics, most proofs are found in Appendix. For all these laws, we assume two global variables v and h.

Assignment and (atomic) choice

The first law describes an assignment as a particular case of choice. This law will allow us to state the laws in terms of only choice ($:\in$). However, particular instances of certain laws will be stated with assignment $:=$ when it is relevant. The following are the equivalent of the classical laws (Law 2.10, Law 2.11) for choice and they serve the same purpose. They allow us to write any choice into a total choice (where all the variables appear in the lhs of the choose operator in a standard order).

Law 4.1.

$$\mathsf{x} := e \;=\; \mathsf{x} :\in \{e\}$$

Law 4.2.

$$\mathsf{x},\mathsf{y} :\in E \;=\; \mathsf{x},\mathsf{y},\mathsf{z} :\in \{(e,f) : E \cdot (e,f,z)\}$$

Law 4.3.

$$\mathsf{x},\mathsf{y} :\in E \;=\; \mathsf{y},\mathsf{x} :\in \{(e,f) : E \cdot (f,e)\}$$

As observed for classical programs, any assignment can be written as an assignment to the vector of all the variables. However, in our framework, rather than a single vector, we will use a pair (v,h) in which v is the vector of all visible variables and h is the vector of all hidden ones. These first laws are straightforward and will not be proved.

skip and reveal

Law 4.4 states that the program **skip** is the identity of sequential composition. Law 4.5 which was observed by Morgan and can be used as a definition of **reveal**.

Law 4.4.

$$\mathbf{skip} = \mathsf{v},\mathsf{h} :\in \{\mathsf{v},\mathsf{h}\}$$

Law 4.5.

$$\mathbf{reveal}\, e.\mathsf{v}.\mathsf{h} \;=\; |[\, \mathbf{vis}\, \mathsf{w} : \mathcal{W} \cdot \mathsf{w} := e.\mathsf{v}.\mathsf{h} \,]|$$

Proof. In Table 3.3, **reveal** $e.\mathsf{v}.\mathsf{h}$ changes only the Shadow H maintaining the global variables. Assigning $e.\mathsf{v}.\mathsf{h}$ to a local visible variable also changes the Shadow H in the same way and the local visible is projected away. □

Nondeterministic choice

Law 4.6 is straightforward as for classical programs.

Law 4.6. *For non-empty sets of programs $\mathcal{P}$ and $\mathcal{Q}$*

$$(\textstyle\bigsqcap \mathcal{P}) \sqcap (\bigsqcap \mathcal{Q}) \;=\; \bigsqcap(\mathcal{P} \cup \mathcal{Q}).$$

Proof. It follows from the definition of the nondeterministic choice of a set of programs. □

Local blocks

Law 4.7 states that a local variable is assumed to be initialised arbitrarily after its declaration, as is the case for classical programs. The proof (in Appendix) is done for local visible variable and needs slight changes to apply for hidden ones. Law 4.8 and its generalisation (Law 4.9) state that a local block assigning only to a hidden local variable is **skip**. Law 4.10 states that superposing the declarations of two local variables is equal to the declaration of a local variable couple. Law 4.11 states that nondeterministic choice distributes local blocks.

Law 4.7.

$$|[\,\mathbf{dec}\,\mathsf{x} : \mathcal{T} \bullet P\,]| = |[\,\mathbf{dec}\,\mathsf{x} : \mathcal{T} \bullet (\mathsf{x} :\in \mathcal{T} \,;\, P)\,]|$$

Proof. In Appendix 4.7. □

Law 4.8.

$$|[\,\mathbf{hid}\,\mathsf{k} : \mathcal{H} \bullet \mathsf{k} :\in E\,]| = \;\mathbf{skip}$$

Proof. In Appendix 4.8. □

Law 4.9. *If* k *is not free in* $E_1 (= \mathsf{Proj}_1.E)$

$$|[\,\mathbf{hid}\,\mathsf{k} : \mathcal{H} \bullet \mathsf{x},\mathsf{k} :\in E\,]| = \;\mathsf{x} :\in E_1$$

Law 4.10.

$$|[\,\mathbf{dec}\,\mathsf{x} : \mathcal{T}_1 \bullet \; |[\,\mathbf{dec}\,\mathsf{y} : \mathcal{T}_2 \bullet P\,]|\,]| = |[\,\mathbf{dec}\,(\mathsf{x},\mathsf{y}) : \mathcal{T}_1 \times \mathcal{T}_2 \bullet P\,]|$$

Proof. Projecting away successively the two local variables is the same as projecting away the couple of local variables. □

Law 4.11.

$$|[\,\mathbf{dec}\,\mathsf{x} : \mathcal{T} \cdot P \sqcap Q\,]| = |[\,\mathbf{dec}\,\mathsf{x} : \mathcal{T} \cdot P\,]| \sqcap |[\,\mathbf{dec}\,\mathsf{x} : \mathcal{T} \cdot Q\,]|$$

Proof. It follows from the fact that the union of the projections is equal to the projection of the union. □

We have seen that an important difference between classical and noninterference-aware programs is that 4.8 applies only to hidden local variables. However, under specific circumstances, a choice to a local visible variable can be eliminated. Law 4.12 is a particular case: the assignment to the local variable is irrelevant when the two variables get assigned to the same expression. Law 4.13 is another particular case: when the expression assigned to the local visible variable does not involve h then the assignment can be discarded.

Law 4.12. *If* $e = f$ *then*

$$|[\,\mathbf{vis}\,\mathsf{w} : \mathcal{W} \cdot \mathsf{w}, \mathsf{v} := e, f\,]| = \mathsf{v} := f$$

Law 4.13. *If* h *is not free in* e *then*

$$|[\,\mathbf{vis}\,\mathsf{w} : \mathcal{W} \cdot \mathsf{w}, \mathsf{v} := e, f\,]| = \mathsf{v} := f$$

These two laws follows from Law 4.1 defining refinement, which is also the most general law to eliminate local visible variables. We note that the condition in the two laws are not necessary conditions. In fact, assignment to a local visible variable can be discarded if it reveals no more information on h than the assignment to the global visible variable.

Conditional

As for classical sequential programs, a conditional joining two assignments to the same variables may be replaced by a single assignment of a conditional expression to the same variables. This can be done here, but with the precaution that the boolean expression b defining the conditional is published, so the law for classical sequential programs is modified to include an assignment of the value of the condition to a local variable. Law 4.15 states that nondeterministic choice distributes conditional. Law 4.16 is straightforward: it implies that nondeterministic choice on either side of the conditional distributes the conditional. Law 4.17 states that conditional between local blocks can be merged into a local block containing a conditional.

Law 4.14.

$$(\mathsf{v},\mathsf{h} :\in E) \lhd\ b\ \rhd (\mathsf{v},\mathsf{h} :\in F)\ = |[\ \textbf{vis}\,\mathsf{w} : \mathcal{W} \bullet \mathsf{w},\mathsf{v},\mathsf{h} :\in \{b\} \times (E \lhd\ b\ \rhd F)\ ]|$$

Proof. In Appendix 4.14. □

Law 4.15.

$$P \lhd\ b\ \rhd (Q \sqcap R)\ =\ (P \lhd\ b\ \rhd Q) \sqcap (P \lhd\ b\ \rhd R)$$

Proof. In Appendix 4.15. □

Law 4.16.

$$P \lhd\ b\ \rhd Q\ =\ Q \lhd \neg b \rhd P$$

Proof. Propositional calculus. □

Law 4.17. *If* x *is not free in* Q *and* y *is not free in* P *then*

$$|[\ \textbf{dec}\,\mathsf{x} : \mathcal{T}_1 \bullet P\ ]| \lhd b \rhd |[\ \textbf{dec}\,\mathsf{y} : \mathcal{T}_2 \bullet Q\ ]| = |[\ \textbf{dec}\,(\mathsf{x},\mathsf{y}) : \mathcal{T}_1 \times \mathcal{T}_2 \bullet P \lhd\ b\ \rhd Q\ ]|$$

Proof. In Appendix 4.17 □

Sequential composition

Law 4.18 and Law 4.19 state that (explicit) nondeterministic choice distributes sequential composition. These laws hold for the classical programs discussed in Chapter 2 (but not for all computations in the specification space) and can be proved in the Shadow semantics. Law 4.20 shows how a sequential composition of local blocks can be merged into one local block. Law 4.21 is a general law for sequential composition of choices. It states that overwriting (v, h) requires keeping track of the first value of v using a local visible variable w. A particular case of this law (Law 4.22) reflects the adversary's perfect recall. It states that two successive assignments to v can be merged as long as its first value is stored, that is achieved using a local visible variable w. Afterwards, the knowledge of h is affected by both the value of v and the value of w.

Law 4.18.

$$(P \sqcap Q) \fatsemi R \;=\; (P \fatsemi R) \sqcap (Q \fatsemi R)$$

Proof. The application of $\mathsf{lift}.[\![R]\!]$ on the union of the semantics of P and Q is the same as the union of the application $\mathsf{lift}.[\![R]\!]$ respectively on P and Q (because $\mathsf{lift}.f.(A \cup B) = \mathsf{lift}.f.A \cup \mathsf{lift}.f.B$). Because of the context involved, this proof can also be done using the Transfer Principle. □

Law 4.19.

$$R \fatsemi (P \sqcap Q) \;=\; (R \fatsemi P) \sqcap (R \fatsemi Q)$$

Proof. The semantics of $(P \sqcap Q)$ applied to a triple equals the union of the semantics of P and Q. By lifting, the semantics applies to the set of triples in R, (because if $f.a = g.a. \cup h.a$ then $\mathsf{lift}.f.A = \mathsf{lift}.g.A \cup \mathsf{lift}.h.A$) .This proof can also be done using the Transfer Principle □

Law 4.20. *If the variable* x *is not free in* Q *and the variable* y *is not free in* P *then*

$$|[\,\mathbf{dec}\,\mathsf{x} : \mathcal{T}_1 \cdot P\,]| \fatsemi |[\,\mathbf{dec}\,\mathsf{y} : \mathcal{T}_2 \cdot Q\,]| = |[\,\mathbf{dec}\,(\mathsf{x},\mathsf{y}) : \mathcal{T}_1 {\times} \mathcal{T}_2 \cdot P \fatsemi Q\,]|$$

Proof. In Appendix 4.20. □

Law 4.21.

$$(\mathsf{v},\mathsf{h} :\in E.\mathsf{v}.\mathsf{h} \mathbin{⨾} \mathsf{v},\mathsf{h} :\in F.\mathsf{v}.\mathsf{h}) \ = \ |[\, \mathbf{vis}\, \mathsf{w} : \mathcal{W} \bullet \mathsf{w},\mathsf{v},\mathsf{h} :\in \{(e_1, e_2) : E.\mathsf{v}.\mathsf{h};\ (f_1, f_2) : F.e_1.e_2 \bullet (e_1, f_1, f_2)\}\,]|$$

Proof. In Appendix 4.21. □

Law 4.22. *Perfect recall*

$$(\mathsf{v} := e \mathbin{⨾} \mathsf{v} := f.\mathsf{v}) \ = |[\, \mathbf{vis}\, \mathsf{w} : \mathcal{W} \bullet (\mathsf{w},\mathsf{v} := e, f.e)\,]|$$

Example 4.1. If v is assigned to itself in the second choice, then w and v have the same value, so Law 4.12 discards w

$$\mathsf{v},\mathsf{h} :\in E.\mathsf{v}.\mathsf{h} \mathbin{⨾} \mathsf{h} :\in F.\mathsf{v}.\mathsf{h} \ = \ \mathsf{v},\mathsf{h} :\in \{(e_1, e_2) : E.\mathsf{v}.\mathsf{h};\ f : F.e_1.e_2 \bullet (e_1, f)\}$$

If v is assigned to itself in the first choice, then the expression assigned to w (which is v) does not involve h, so Law 4.13 discards w

$$\mathsf{h} :\in E.\mathsf{v}.\mathsf{h} \mathbin{⨾} \mathsf{v},\mathsf{h} :\in F.\mathsf{v}.\mathsf{h} \ = \ \mathsf{v},\mathsf{h} :\in \{e : E.\mathsf{v}.\mathsf{h};\ (f_1, f_2) : F.\mathsf{v}.e \bullet (f_1, f_2)\}$$

Example 4.2. Consider $\mathsf{h} :\in \{0,1,2,3\} \mathbin{⨾} \mathsf{v} := \mathsf{h} \bmod 2$. The assignment to v would reveal some information on h. Indeed, using Law 4.21 and discarding the local variable w by Law 4.12, this composition equals $\mathsf{v},\mathsf{h} :\in \{(0,0),(0,2),(1,1),(1,3)\}$. By observing the value v the attacker knows the parity of h.

Atomic blocks

As observed in Chapter 2, atomic blocks were introduced in [10] to allow some computation to be hidden from the attacker. Programs inside atomic brackets are treated as classical programs i.e. there is no distinction between visible and hidden variables. This is possible because apart from this distinction, the remaining uncommon command **reveal** can be expressed using the other operators (Law 4.5).

Law 4.23. *If $P = Q$ in the classical context then $\langle\!\langle P \rangle\!\rangle = \langle\!\langle Q \rangle\!\rangle$*

Proof. It follows directly from the semantics of $\langle\!\langle \cdot \rangle\!\rangle$. □

It is important to note that the equalities in 4.23 cannot be weakened to refinement as illustrated by the following counter-example.

Example 4.3. [10] In the classical setting $(\mathsf{h} := 0 \sqcap \mathsf{h} := 1) \sqsubseteq \mathsf{h} := 0$. However for noninterference-aware programs $\langle\!\langle(\mathsf{h} := 0) \sqcap (\mathsf{h} := 1)\rangle\!\rangle \not\sqsubseteq \langle\!\langle\mathsf{h} := 0\rangle\!\rangle$.

Atomic block is the only non-monotonic combinator with respect to the refinement ordering. Law 4.24 states that a nondeterministic choice of assignments executed atomically is equivalent to an atomic choice.

Law 4.24.

$$\langle\!\langle \textstyle\bigsqcap\{(e,f) : E \bullet (\mathsf{v},\mathsf{h}) := (e,f)\}\rangle\!\rangle = (\mathsf{v},\mathsf{h}) :\in E$$

Proof. In Appendix 4.24. □

.

4.3 Reduction to normal form

The primitive commands: assignment, (atomic) choice, skip and reveal

Recall that the normal form for classical programs is

$$\textstyle\bigsqcap\{e : E \bullet x := e\}.$$

We not that choice $:\in$ is not captured by this normal form. However, we have seen that by Law 4.2 and Law 4.3, we can write any choice as a total choice. Thus, at first, we propose

$$\textstyle\bigsqcap\{E : \mathcal{E} \bullet x :\in E\}$$

to be the normal form. However, this does not also capture all our programs. For instance, the basic command **reveal** cannot be written in this form. We will also see that unlike classical programs, the sequential composition, the conditional cannot be eliminated between this normal form. Moreover, although $:\in$ is a nondeterministic choice, it differs from the explicit nondeterministic choice $\sqcap$ (as explained in Chapter 3). Thus, as for classical programs, our final normal form should include explicit nondeterministic choice. Yet, a nondeterministic choice of atomic choice still cannot express the basic command **reveal**. These observations conduct us to take a normal form that includes a local visible variable as the following:

$$\sqcap\{E : \mathcal{E} \bullet \; [\![\, \mathbf{vis}\, \mathsf{w} : \mathcal{W} \bullet (\mathsf{w}, \mathsf{v}, \mathsf{h} :\in E.\mathsf{v}.\mathsf{h}) \,]\!]\}.$$

This is our normal form; it is important to notice that the expression in the rhs of the choose operator does not depend on the local variable w. The previous normal form for choice and assignment can be written in this form by adding an identity assignment to w.

We have seen that **skip**, **reveal**, choice and assignment can be written in our normal form. Now, we will see how we can eliminate the other commands operating on programs in normal form.

Elimination of nondeterministic choice

Nondeterministic choice between programs in normal form can be reduced to normal form by Law 4.6.

Elimination of conditional

A conditional joining two programs in normal forms can be reduced to normal form as follows. We start with a program of the form

$$\sqcap\{E : \mathcal{E} \bullet \; [\![\, \mathbf{vis}\, \mathsf{w} : \mathcal{W} \bullet (\mathsf{w}, \mathsf{v}, \mathsf{h} :\in E.\mathsf{v}.\mathsf{h}) \,]\!]\} \\ \lhd \; b \; \rhd \sqcap\{F : \mathcal{F} \bullet \; [\![\, \mathbf{vis}\, \mathsf{u} : \mathcal{U} \bullet (\mathsf{w}, \mathsf{v}, \mathsf{h} :\in F.\mathsf{v}.\mathsf{h}) \,]\!]\}.$$

Then:

- by Law 4.15 the nondeterministic choices distribute the conditional;
- by Law 4.17 each pair of local blocks joined by the conditional can be merged;
- by Law 4.14 the conditional joining choices can be reduced to a choice to conditional, yielding an assignment of the value of b to a local visible variable;
- by Law 4.10 the scopes of the local variables can be merged yielding a program in normal form.

Elimination of sequential composition

A sequential composition of programs in normal form can be reduced to normal form as follows. We start from a program of the form

$$\sqcap\{E : \mathcal{E} \bullet\ |[\,\mathbf{vis}\,\mathsf{w} : \mathcal{W} \bullet (\mathsf{w},\mathsf{v},\mathsf{h} :\in E.\mathsf{v}.\mathsf{h})\,]|\}$$
$$\fatsemi \sqcap\{F : \mathcal{F} \bullet\ |[\,\mathbf{vis}\,\mathsf{u} : \mathcal{U} \bullet (\mathsf{w},\mathsf{v},\mathsf{h} :\in F.\mathsf{v}.\mathsf{h})\,]|\}.$$

Then:

- by Law 4.18 and Law 4.19 the nondeterministic choices distributes the sequential composition;
- by Law 4.20, each pair of local blocks joined by a $\fatsemi$ can be merged into a single local block,
- by Law 4.21, each sequential composition of choices can be reduced to a single choice.

Elimination of hid local blocks

A program $|[\,\mathbf{hid}\,\mathsf{k} : \mathcal{K} \bullet P\,]|$ with P is in normal form can be reduced to normal form. The two local blocks can be merged by Law 4.10. Because of the law of initialisation of local blocks, the rhs of the choice in P can be rewritten not to be a function of the local hidden variable using laws of composition. Then, the choice to the local hidden variable k can be discarded by Law 4.9.

$$|[\,\mathbf{hid}\,\mathsf{k} : \mathcal{K} \bullet P\,]|$$

$=$ Definition of normal form

$$|[\,\mathbf{hid}\,\mathsf{k} : \mathcal{K} \bullet \sqcap\left\{E : \mathcal{E} \bullet\ |[\,\mathbf{vis}\,\mathsf{w} : \mathcal{W} \bullet \mathsf{w},\mathsf{v},\mathsf{h} :\in E.\mathsf{v}.\mathsf{h}.\mathsf{k}\,]|\ \right\}]|$$

$=$ Law 4.11

$$|[\,\mathbf{hid}\,\mathsf{k} : \mathcal{K} \bullet\ |[\,\mathbf{vis}\,\mathsf{w} : \mathcal{W} \bullet \sqcap\left\{E : \mathcal{E} \bullet \mathsf{w},\mathsf{v},\mathsf{h} :\in E.\mathsf{v}.\mathsf{h}.\mathsf{k}\right\}]|\,]|$$

$=$ Law 4.10 twice

$$|[\,\mathbf{vis}\,\mathsf{w} : \mathcal{W} \bullet\ |[\,\mathbf{hid}\,\mathsf{k} : \mathcal{K} \bullet \sqcap\left\{E : \mathcal{E} \bullet \mathsf{w},\mathsf{v},\mathsf{h} :\in E.\mathsf{v}.\mathsf{h}.\mathsf{k}\right\}]|\,]|$$

$=$ Law 4.7

$$|[\,\mathbf{vis}\,\mathsf{w} : \mathcal{W} \bullet\ |[\,\mathbf{hid}\,\mathsf{k} : \mathcal{K} \bullet \left(\mathsf{k} :\in \mathcal{K} \fatsemi \sqcap\left\{E : \mathcal{E} \bullet \mathsf{w},\mathsf{v},\mathsf{h} :\in E.\mathsf{v}.\mathsf{h}.\mathsf{k}\right\}\right)]|\,]|$$

$=$ $\sqcap$ distributes $\fatsemi$ Law 4.19

$$|[\,\mathbf{vis}\,\mathsf{w} : \mathcal{W} \bullet\ |[\,\mathbf{hid}\,\mathsf{k} : \mathcal{K} \bullet \left(\sqcap\left\{E : \mathcal{E} \bullet k :\in \mathcal{K} \fatsemi \mathsf{w},\mathsf{v},\mathsf{h} :\in E.\mathsf{v}.\mathsf{h}.\mathsf{k}\right\}\right)]|\,]|$$

$=$ Law 4.21

$$|[\,\mathbf{vis}\,\mathsf{w} : \mathcal{W} \bullet\ |[\,\mathbf{hid}\,\mathsf{k} : \mathcal{K} \bullet \Big(\prod\Big\{E : \mathcal{E} \bullet \big\{\kappa : \mathcal{K} \bullet \mathsf{w},\mathsf{v},\mathsf{h},\mathsf{k} :\in \{e : E.\mathsf{v}.\mathsf{h}.\kappa;\ l : \mathcal{K} \cap (E.v.h)^{-1}.e \bullet (e,l)\}\big\}\Big\}\Big)\,]|\,]|$$

$=$ Law 4.9 as k is not free in $F.\mathsf{v}.\mathsf{h}.\kappa = \{e : E.\mathsf{v}.\mathsf{h}.\kappa;\ l : \mathcal{K} \cap (E.\mathsf{v}.\mathsf{h})^{-1}.e \bullet (e,l)\}$

$$|[\,\mathbf{vis}\,\mathsf{w} : \mathcal{W} \bullet \prod\big\{E : \mathcal{E} \bullet \mathsf{w},\mathsf{v},\mathsf{h} :\in \{\kappa : \mathcal{K} \bullet F.\mathsf{v}.\mathsf{h}.\kappa\}\big\}\,]|$$

$=$ Law 4.11, $G.\mathsf{v}.\mathsf{h} = \{\kappa : \mathcal{K} \bullet F.\mathsf{v}.\mathsf{h}.\kappa\}$

$$\prod\big\{E : \mathcal{E} \bullet\ |[\,\mathbf{vis}\,\mathsf{w} : \mathcal{W} \bullet \mathsf{w},\mathsf{v},\mathsf{h} :\in G.\mathsf{v}.\mathsf{h}\,]|\ \big\}$$

Elimination of vis local blocks

A program $|[\,\mathbf{vis}\,\mathsf{u} : \mathcal{U} \bullet P\,]|$ where P in normal form can be reduced to normal form. The two blocks can be merged by Law 4.10. As for **hid** local blocks, because of the initialisation law, the rhs of the choice in P can be written to be independent from w. However the assignment to w is be discarded yet the result is in normal form.

$$|[\,\mathbf{vis}\,\mathsf{u} : \mathcal{U} \bullet P\,]|$$

$=$ Definition of normal form

$$|[\,\mathbf{vis}\,\mathsf{u} : \mathcal{U} \bullet \prod\big\{E : \mathcal{E} \bullet\ |[\,\mathbf{vis}\,\mathsf{w} : \mathcal{W} \bullet \mathsf{w},\mathsf{v},\mathsf{h} :\in E.\mathsf{u}.\mathsf{v}.\mathsf{h}\,]|\ \big\}\,]|$$

$=$ Law 4.11

$$|[\,\mathbf{vis}\,\mathsf{u} : \mathcal{U} \bullet\ |[\,\mathbf{vis}\,\mathsf{w} : \mathcal{W} \bullet \prod\big\{E : \mathcal{E} \bullet \mathsf{w},\mathsf{v},\mathsf{h} :\in E.\mathsf{u}.\mathsf{v}.\mathsf{h}\big\}\,]|\,]|$$

$=$ Law 4.10 twice

$$|[\,\mathbf{vis}\,\mathsf{w} : \mathcal{W} \bullet\ |[\,\mathbf{vis}\,\mathsf{u} : \mathcal{U} \bullet \prod\big\{E : \mathcal{E} \bullet \mathsf{w},\mathsf{v},\mathsf{h} :\in E.\mathsf{u}.\mathsf{v}.\mathsf{h}\big\}\,]|\,]|$$

$=$ Law 4.7

$$|[\,\mathbf{vis}\,\mathsf{w} : \mathcal{W} \bullet\ |[\,\mathbf{vis}\,\mathsf{u} : \mathcal{U} \bullet \big(\mathsf{u} :\in \mathcal{U} \,;\, \prod\big\{E : \mathcal{E} \bullet \mathsf{w},\mathsf{v},\mathsf{h} :\in E.\mathsf{u}.\mathsf{v}.\mathsf{h}\big\}\big)\,]|\,]|$$

$=$ Law 4.19

$$|[\,\mathbf{vis}\,\mathsf{w} : \mathcal{W} \bullet\ |[\,\mathbf{vis}\,\mathsf{u} : \mathcal{U} \bullet \big(\prod\big\{E : \mathcal{E} \bullet \mathsf{u} :\in \mathcal{U} \,;\, \mathsf{w},\mathsf{v},\mathsf{h} :\in E.\mathsf{u}.\mathsf{v}.\mathsf{h}\big\}\big)\,]|\,]|$$

$=$ Law 4.21

$$|[\,\mathbf{vis}\,\mathsf{w} : \mathcal{W} \bullet\ |[\,\mathbf{vis}\,\mathsf{u} : \mathcal{U} \bullet \big(\prod\big\{E : \mathcal{E} \bullet \mathsf{u},\mathsf{w},\mathsf{v},\mathsf{h} :\in \{\nu : \mathcal{U} \bullet E.\nu.\mathsf{v}.\mathsf{h}\}\big\}\big)\,]|\,]|$$

$=$ Law 4.10

$$|[\,\mathbf{vis}\,\mathsf{u},\mathsf{w} : \mathcal{U}\times\mathcal{W} \bullet \big(\prod\big\{E : \mathcal{E} \bullet \mathsf{u},\mathsf{w},\mathsf{v},\mathsf{h} :\in \{\nu : \mathcal{U} \bullet E.\nu.\mathsf{v}.\mathsf{h}\}\big\}\big)\,]|$$

Elimination of atomic blocks

The particularity of atomic blocks is that classical algebraic laws apply inside them. When interpreted classically, the noninterference-aware programming language is the same as its classical analogue. Thus the laws used inside a local block are those listed in Chapter 2. From the completeness result of that chapter, any program inside the local block can be written in classical normal form as follows:

$$\langle\!\langle \textstyle\bigsqcap\{E : \mathcal{E} \cdot\ |[\, \mathbf{vis}\, \mathsf{w} : \mathcal{W} \cdot (\mathsf{w}, \mathsf{v}, \mathsf{h} :\in E.\mathsf{v}.\mathsf{h})\,]|\}\rangle\!\rangle$$

$=$ Law 2.17 (classical), because w is not free in $E.\mathsf{v}.\mathsf{h}$

$$\langle\!\langle \textstyle\bigsqcap\{E : \mathcal{E} \cdot (\mathsf{v}, \mathsf{h} :\in E_{1,2}.\mathsf{v}.\mathsf{h})\}\rangle\!\rangle$$

$=$ Law 2.8 (classical)

$$\langle\!\langle \textstyle\bigsqcap\{e, f : \left(\bigcup \mathcal{E}\right)_{1,2}.\mathsf{v}.\mathsf{h} \cdot \mathsf{v}, \mathsf{h} := e, f\big)\}\rangle\!\rangle$$

$=$ Law 4.24

$$\mathsf{v}, \mathsf{h} :\in \left(\bigcup \mathcal{E}\right)_{1,2}.\mathsf{v}.\mathsf{h}$$

Finally we get a program in the normal form, which is actually an atomic program.

Refinement

Refinement is now defined as decrease of composite nondeterminism and preservation of ignorance. Recall that composite nondeterminism is a demonic choice in which the attacker knows the computation whereas an atomic choice is a demonic choice in which he can not know how the choice was made. Composite nondeteministic choices make use of $\sqcap$ operating between program fragments whereas the atomic nondeterministic choice is expressed by $:\in$. Some instances of atomic choice can be considered as composite ones. In $\mathsf{v}, \mathsf{h} :\in \{(0,0), (1,1)\}$, the attacker always knows how the choice was made by observing the final value of the visible variable.

The first aspect of refinement, which is the decrease of nondeterminism, is characterised as for classical programs in Chapter 2 by

Law 4.25.

$$P \sqsubseteq Q \;\equiv\; P \sqcap Q = P$$

A nondeterministic (explicit) choice of programs refines its specification if and only if each program in the branches refines the specification. This is a direct consequence of the definition of the refinement order and the properties of $\sqcap$.

Law 4.26.

$$Q \sqsubseteq \bigsqcap \mathcal{P} \textbf{ iff } \forall P : \mathcal{P} \bullet Q \sqsubseteq P$$

The previous law allows treatment individually of the branches of a nondeterministic choice in the rhs of the refinement. Each refinement can then be decided by the following law, which we state as a theorem.

Theorem 4.1.

$$\bigsqcap \left\{ E : \mathcal{E} \bullet \ |[\, \textbf{vis}\, w \bullet \mathsf{w}, \mathsf{v}, \mathsf{h} :\in E.\mathsf{v}.\mathsf{h} \,]| \ \right\} \sqsubseteq |[\, \textbf{vis}\, w \bullet \mathsf{w}, \mathsf{v}, \mathsf{h} :\in F.\mathsf{v}.\mathsf{h} \,]|$$

$$\textbf{iff}$$

$$\begin{aligned}
&\forall\, v, h :\in \mathcal{V} \times \mathcal{H};\ \forall (w', v', h') : F.v.h \bullet \\
&\exists\, E \in \mathcal{E} \wedge \exists (w^{\sim}, v', h') \in E.v.h \mid \\
&\forall\, k : \mathcal{H} \bullet E_{0,1}.v.k \ni (w^{\sim}, v') \Rightarrow \begin{pmatrix} F_{0,1}.v.k \ni (w', v') \Rightarrow E_2.v.k \subseteq (F_2.v.h \cup F_2.v.k) \\ F_{0,1}.v.k \not\ni (w', v') \Rightarrow E_2.v.k \subseteq F_2.v.h \end{pmatrix}
\end{aligned}$$

Proof. In Appendix 4.1. □

In the following, we list more practical corollaries of this theorem.

Corollary 4.1. *For sets of expressions $E \in \mathcal{E}$ that do not contain* h

$$\bigsqcap \left\{ E : \mathcal{E} \bullet (\mathsf{v}, \mathsf{h}) :\in E \right\} \sqsubseteq (\mathsf{v}, \mathsf{h}) :\in F \textbf{ iff } \exists\, \mathcal{E}_s : \mathbb{P}.\mathcal{E} \bullet$$

- $F \subseteq \bigcup \mathcal{E}$
- $\mathsf{Pr}_2.(\bigcup \mathcal{E}_s) \subseteq \mathsf{Pr}_2.F$

Corollary 4.2. *For sets of expressions $E \in \mathcal{E}$ that do not contain* h

$$\bigsqcap \left\{ E : \mathcal{E} \bullet \mathsf{h} :\in E \right\} \sqsubseteq \mathsf{h} :\in F \textbf{ iff } \exists\, \mathcal{E}_s : \mathbb{P}.\mathcal{E} \bullet F = \bigcup \mathcal{E}_s$$

Corollary 4.3. *For sets of expressions $E \in \mathcal{E}$ that do not contain* h

$$\sqcap \left\{ E : \mathcal{E} \bullet \mathsf{v} :\in E \right\} \sqsubseteq \mathsf{v} :\in F \textbf{ iff } F \subseteq \bigcup \mathcal{E}$$

Corollary 4.4.

$$\textbf{reveal}\, e \sqsubseteq \textbf{reveal}\, f \;\; \textbf{iff} \;\; \forall\, v : \mathcal{V} \bullet \;\pi_{e.v} \subseteq \pi_{f.v}$$

where $\pi_{e.v}$ designates the canonical surjection:

$$\mathcal{H} \longrightarrow \mathbb{P}\mathcal{H};\; h \longmapsto (e.v)^{-1}.(e.v).h$$

4.4 Soundness and completeness

We have seen that the given subset of the laws, are sound in the shadow model. Those laws permit us to reduce any security-aware program into the proposed normal form. We have also given a theorem to prove refinement between programs in normal form. Therefore any refinement that holds in the Shadow model can be proved using algebraic laws (particularly the classical laws 2.3, 2.4, 2.5, 2.6, 2.7, 2.9, 2.10, 2.11, 2.12, 2.13, 2.15, 2.17, 2.18, 2.19, 2.25, 2.26, the security-aware laws 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.10, 4.11, 4.14, 4.15, 4.16, 4.17, 4.18, 4.19, 4.20, 4.21, 4.24, 4.26 and Theorem 4.1), this is our completeness result. It implies that program development can be done by using only algebraic laws without need of the semantics. In summary:

Theorem 4.2 (Completeness)**.** *For noninterference-aware programs P and Q*

If $P \sqsubseteq Q$ in the Shadow semantics then $P \sqsubseteq Q$ can be proved algebraically

This completeness result implies for instance that any other law can be proved using the complete set of laws. For example:

Law 4.27.

$$\langle\!\langle \langle\!\langle P \rangle\!\rangle \rangle\!\rangle = \langle\!\langle P \rangle\!\rangle$$

Proof. Using the normal form of P

$\langle\!\langle \bigsqcap\{E : \mathcal{E} \cdot\ |[\, \mathbf{vis}\, \mathsf{w} : \mathcal{W} \cdot (\mathsf{w}, \mathsf{v}, \mathsf{h} :\in E.\mathsf{v}.\mathsf{h}) \,]|\}\rangle\!\rangle$

$=$ elimination of atomic blocks

$\mathsf{v}, \mathsf{h} :\in \left(\bigcup \mathcal{E}\right)_{1,2}.\mathsf{v}.\mathsf{h}$

$=$ Law 4.24

$\langle\!\langle \mathsf{v}, \mathsf{h} :\in \left(\bigcup \mathcal{E}\right)_{1,2}.\mathsf{v}.\mathsf{h}\rangle\!\rangle$

$=$ elimination of atomic blocks

$\langle\!\langle \langle\!\langle \bigsqcap\{E : \mathcal{E} \cdot\ |[\, \mathbf{vis}\, \mathsf{w} : \mathcal{W} \cdot (\mathsf{w}, \mathsf{v}, \mathsf{h} :\in E.\mathsf{v}.\mathsf{h}) \,]|\}\rangle\!\rangle\rangle\!\rangle.$

$\square$

Chapter 5

Conclusion

We have presented a complete set of laws for classical sequential nondeterministic programs: in the classical context refinement between programs can be proved entirely algebraically (Chapter 2). This has motivated us to establish algebraic laws for stepwise refinement of security protocols. Our work has been based on Morgan's approach to secure refinement which distinguishes between visible and hidden program variables. The first issue about secure refinement is to avoid the Refinement Paradox $h := \{0, 1\} \sqsubseteq h := 0$ which has been overcome by assuming two kinds of nondeterminism: atomic $h :\in \{0, 1\}$ and composite $(h := 0 \sqcap h := 1)$.

Algebraic requirements (monotonicity, classical treatment of visible-only program fragments) lead Morgan to make further assumptions: the attacker can see program flow (perfect recall, resolution of explicit nondeterminism, resolution of conditional). While these led him to construct the Shadow semantics, a model for noninterference-aware programs, they have inspired us to establish the algebraic laws. We have given sufficient laws to obtain normal form. As noninterference aware programs behave differently from classical ones, the normal form

$$\bigsqcap\{E : \mathcal{E} \cdot \; [\![\, \mathbf{vis}\, \mathsf{w} : \mathcal{W} \cdot (\mathsf{w}, \mathsf{v}, \mathsf{h} :\in E.\mathsf{v}.\mathsf{h}) \,]\!]\}$$

differs from the classical one. Although the list of laws is not exhaustive, aiming to get the normal form has led us to explore properties of all program constructors. We have proved that the laws were sound in the Shadow semantics. Finally we have proved the completeness of the laws with respect to the shadow semantics.

A model has been built by Morgan for correctness and security of programs . We have contributed to the corresponding refinement algebra by establishing laws in a more systematic way: the completeness of the laws ensures that derivation of protocols can be done entirely algebraically. In addition, the laws and the normal form given here can prove further sound laws without translating them into semantics. Time has not allowed us to give the case study (an anonymous auction protocol) that illustrates the use of these laws. This would be part of the future work, together with the treatment of iteration and recursion. Other projects are to extend the program to include other specification constructs and to establish algebraic laws for refinement of programs that feature both noninterference and probability [5, 6].

Appendices

Soundness proofs

Law 4.7.

$$|[\,\mathbf{dec}\,\mathsf{x} : \mathcal{V} \bullet P\,]| = |[\,\mathbf{dec}\,\mathsf{x} : \mathcal{V} \bullet (\mathsf{x} :\in \mathcal{V} \,;\, P)\,]|$$

Proof. We will prove for the case of only **vis** , the case of **hid** is similar.

$$\llbracket |[\,\mathbf{vis}\,\mathsf{w} : \mathcal{V} \bullet (\mathsf{w} :\in \mathcal{V} \,;\, P)\,]| \rrbracket.v.h.H$$

$=$ semantics of local blocks

$$\mathsf{Proj}_{v,h,H}\cdot \bigcup \left\{ w : \mathcal{V} \bullet \llbracket (w :\in \mathcal{V} \,;\, P) \rrbracket.w.v.h.H \right\}$$

$=$ semantics of $;$

$$\mathsf{Proj}_{v,h,H}\cdot \bigcup \left\{ w : \mathcal{V} \bullet \mathsf{lift}.\llbracket P \rrbracket.\llbracket w :\in \mathcal{V} \rrbracket.w.v.h.H \right\}$$

$=$ semantics of $:\in$ for visible variables, $\mathcal{V}$ does not involve h

$$\mathsf{Proj}_{v,h,H}\cdot \bigcup \left\{ w : \mathcal{V} \bullet \mathsf{lift}.\llbracket P \rrbracket.\{ w' : \mathcal{V} \bullet (w', v, h, H) \} \right\}$$

$=$ definition of lift

$$\mathsf{Proj}_{v,h,H}\cdot \bigcup \left\{ w : \mathcal{V} \bullet \bigcup \{ w' : \mathcal{V} \bullet .\llbracket P \rrbracket.w'.v.h.H \} \right\}$$

$=$ w is not free in $\llbracket P \rrbracket.w'.v.h.H$

$$\mathsf{Proj}_{v,h,H}\cdot \bigcup \{ w' : \mathcal{V} \bullet .\llbracket P \rrbracket.w'.v.h.H \}$$

$=$ semantics of local blocks

$$\llbracket |[\,\mathbf{vis}\,\mathsf{w} : \mathcal{V} \bullet P\,]| \rrbracket$$

□

Law 4.8.

$$|[\,\mathbf{hid}\,\mathsf{k} : \mathcal{H} \bullet \mathsf{k} :\in E\,]| = \mathbf{skip}$$

Proof.

$$\llbracket \textbf{begin hid}\,\mathsf{k} : \mathcal{H} \bullet \mathsf{k} :\in E\ \textbf{end} \rrbracket.v.h.H$$

$=$ semantics of local blocks

$$\mathsf{Proj}_{v,h,H} \cdot \bigcup \{ k : \mathcal{H} \bullet \llbracket k :\in E \rrbracket.v.(h,k).(H \times \mathcal{H}) \}$$

$=$ semantics of $:\in$

$$\mathsf{Proj}_{v,h,H} \cdot \bigcup \{ k : \mathcal{H} \bullet \{ k' : E.v.(h,k) \bullet (v, (h,k'), H \times \{ l : \mathcal{H};\ l' : E.v.(h,l) \bullet l' \}\}\}$$

$=$ set union distributes Proj, calculus

$$\{(v,h,H)\}$$

$=$ identity is skip

$$\llbracket \textbf{skip} \rrbracket.v.h.H$$

$\square$

Law 4.14.

$$(\mathsf{v},\mathsf{h} :\in E) \lhd b \rhd (\mathsf{v},\mathsf{h} :\in F) = |[\ \textbf{vis}\,\mathsf{w} : \mathcal{W} \bullet \mathsf{w},\mathsf{v},\mathsf{h} :\in \{b\} \times (E \lhd b \rhd F)\ ]|$$

Proof.

$$\llbracket e :\in E \lhd b \rhd \mathsf{v},\mathsf{h} :\in F \rrbracket.v.h.H$$

$=$ semantics of conditional

$$\llbracket \mathsf{v},\mathsf{h} :\in E \rrbracket.v.h.H_b \cup \llbracket \mathsf{v},\mathsf{h} :\in F \rrbracket.v.h.H_{\neg b}$$

$=$ semantics of $:\in$ twice

$$\{(v',h') : E.v.h \bullet (v',h',\{k : H_b \operatorname{;} k' : \mathcal{H} \mid E.v.k \ni (v',k')\}\}$$
$$\cup \{(v',h') : F.v.h \bullet (v',h',\{k : H_{\neg b} \operatorname{;} k' : \mathcal{H} \mid F.v.k \ni (v',k')\}\}$$

$=$ definition of H_b

$$\mathsf{Proj}_{v,h,H} \cdot \bigcup \Big\{ w : \mathcal{W} \bullet \{(v',h') : E.v.h \bullet (b.v.h, v', h', \{k : H \mid b.v.k \operatorname{;} k' : \mathcal{H} \mid E.v.k \ni (v',k')\}\}$$
$$\cup \{(v',h') : F.v.h \bullet (\neg b.v.h, v', h', \{k : H \mid \neg b.v.k \operatorname{;} k' : \mathcal{H} \mid F.v.k \ni (v',k')\}\} \Big\}$$

$=$ semantics of $:\in$

$$\mathsf{Proj}_{v,h,H} \cdot \bigcup \{ w : \mathcal{W} \bullet \llbracket w, v, h :\in \{b\} \times E \lhd b \rhd F \rrbracket \}$$

$=$ semantics of local blocks

$$[\![|[\, \mathbf{vis}\, \mathsf{w} : \mathcal{W} \bullet \mathsf{w}, \mathsf{v}, \mathsf{h} :\in \{b\} \times (E \lhd b \rhd F)\,]|]\!].v.h.H$$

$\square$

Law 4.15.

$$P \lhd b \rhd (Q \sqcap R) = (P \lhd b \rhd Q) \sqcap (P \lhd b \rhd R)$$

Proof.

$$[\![P \lhd b \rhd (Q \sqcap R)]\!].v.h.H$$

$=$ semantics of conditional, $H_b = \{k : H \mid b.v.k\}$

$$[\![P]\!].v.h.H_b \cup [\![Q \sqcap R]\!].v.h.H_{\neg b}$$

$=$ semantics of $\sqcap$

$$[\![P]\!].v.h.H_b \cup [\![Q]\!].v.h.H_{\neg b} \cup [\![R]\!].v.h.H_{\neg b}$$

$=$ set calculus

$$[\![P]\!].v.h.H_b \cup [\![Q]\!].v.h.H_{\neg b} \cup [\![P]\!].v.h.H_b \cup [\![R]\!].v.h.H_{\neg b}$$

$=$ semantics of conditional

$$[\![P \lhd b \rhd Q]\!].v.h.H \cup [\![P \lhd b \rhd R]\!].v.h.H$$

$=$ semantics of $\sqcap$

$$[\![(P \lhd b \rhd Q) \sqcap (P \lhd b \rhd R)]\!].v.h.H$$

$\square$

Law 4.17. *If* x *is not free in* Q *and* y *is not free in* P *then*

$$|[\, \mathbf{dec}\, \mathsf{x} : \mathcal{T}_1 \bullet P\,]| \lhd b \rhd |[\, \mathbf{dec}\, \mathsf{y} : \mathcal{T}_2 \bullet Q\,]| = |[\, \mathbf{dec}\, (\mathsf{x}, \mathsf{y}) : \mathcal{T}_1 \times \mathcal{T}_2 \bullet P \lhd b \rhd Q\,]|$$

Proof.

$$[\![\; |[\, \mathbf{dec}\, \mathsf{x} : \mathcal{T}_1 \bullet P\,]| \lhd b \rhd |[\, \mathbf{dec}\, \mathsf{y} : \mathcal{T}_2 \bullet Q\,]| \;]\!].v.h.H$$

$=$ semantics of conditional

$$[\![|[\mathsf{x} : \mathcal{T}_1 \bullet P\,]|]\!].v.h.H_b \cup [\![|[\,\mathbf{dec}\,\mathsf{y} : \mathcal{T}_2 \bullet Q\,]|]\!].v.h.H_{\neg b}$$

$=$ semantics of local blocks

$$\mathsf{Proj}_{v,h,H}\cdot \bigcup \{x : \mathcal{T}_1 \bullet [\![P]\!].x.v.h.H_b\} \cup \mathsf{Proj}_{v,h,H}\cdot \bigcup \{y : \mathcal{T}_2 \bullet [\![Q]\!].x.v.h.H_{\neg b}\}$$

$=$ x is not free in Q and y is not free in P

$$\mathsf{Proj}_{v,h,H}\cdot \bigcup \{x : \mathcal{T}_1;\ (x', v', h', H') : [\![P]\!].x.v.h.H_b \bullet (x', y, v', h', H')\}$$

$$\cup\ \mathsf{Proj}_{v,h,H}\cdot \bigcup \{y : \mathcal{T}_2;\ (y', v', h', H') : [\![Q]\!].y.v.h.H_{\neg b} \bullet (x, y', v', h', H')\}$$

$=$ $f(A \cup B) = f(A) \cup f(B)$

$$\mathsf{Proj}_{v,h,H}\cdot \bigcup \{x, y : \mathcal{T}_1 \times \mathcal{T}_2 \bullet [\![P]\!].x.y.v.h.H_b \cup [\![Q]\!].x.y.v.h.H_{\neg b}\}$$

$=$ semantics of local blocks

$$[\![|[\,\mathbf{dec}\,\mathsf{x}, \mathsf{y} : \mathcal{T}_1 \times \mathcal{T}_2 \bullet P \lhd\ b\ \rhd Q\,]|]\!].v.h.H$$

□

Law 4.20. *If the variable* x *is not free in* Q *and the variable* y *is not free in* P *then*

$$|[\,\mathbf{dec}\,\mathsf{x} : \mathcal{T}_1 \bullet P\,]|\ ⨟\ |[\,\mathbf{dec}\,\mathsf{y} : \mathcal{T}_2 \bullet Q\,]| = |[\,\mathbf{dec}\,(\mathsf{x}, \mathsf{y}) : \mathcal{T}_1 \times \mathcal{T}_2 \bullet P ⨟ Q\,]|$$

Proof.

$$[\![\ |[\,\mathbf{dec}\,\mathsf{x} : \mathcal{T}_1 \bullet P\,]|\ ⨟\ |[\,\mathbf{dec}\,\mathsf{y} : \mathcal{T}_2 \bullet\ ]|\ ]\!].v.h.H$$

$=$ semantics of ⨟

$$\mathsf{lift}.[\![|[\,\mathbf{dec}\,\mathsf{x} : \mathcal{T}_2 \bullet Q\,]|]\!].[\![|[\,\mathbf{dec}\,\mathsf{x} : \mathcal{T}_1 \bullet P\,]|]\!].v.h.H$$

$=$ semantics of local blocks

$$\mathsf{lift}.[\![|[\,\mathbf{dec}\,\mathsf{x} : \mathcal{T}_2 \bullet Q\,]|]\!].\mathsf{Proj}_{v,h,H}\cdot \bigcup \{x : \mathcal{T}_1 \bullet [\![P]\!].x.v.h.H\}$$

$=$ definition of lift

$$\bigcup \left\{x : \mathcal{T}_1 \bullet \bigcup \{\mathsf{lift}.[\![|[\,\mathbf{dec}\,\mathsf{y} : \mathcal{T}_2 \bullet Q\,]|]\!].\mathsf{Proj}_{v,h,H}[\![P]\!].x.v.h.H\right\}$$

$=$ definition of lift

$$\bigcup \left\{x : \mathcal{T}_1 \bullet \bigcup \{(v', h', H') : \mathsf{Proj}_{v,h,H}[\![P]\!].x.v.h.H \bullet [\![\mathbf{dec}\,\mathsf{y} : \mathcal{T}_2 \bullet Q]\!].v'.h'.H'\}\right\}$$

$=$ semantics of local blocks

$$\bigcup \left\{x : \mathcal{T}_1 \bullet \bigcup \{(v', h', H') : \mathsf{Proj}_{v,h,H}[\![P]\!].x.v.h.H \bullet \mathsf{Proj}_{v,h,H}\cdot \bigcup\{y : \mathcal{T}_2 \bullet [\![Q]\!].y.v'.h'.H'\}\}\right\}$$

$=$ set calculus

$$\bigcup\Big\{x : \mathcal{T}_1 \bullet \bigcup\{y : \mathcal{T}_2 \bullet \bigcup\big\{(v', h', H') : \mathsf{Proj}_{v,h,H}[\![P]\!].x.v.h.H \bullet \mathsf{Proj}_{v,h,H}.[\![Q]\!].y.v'.h'.H'\}\big\}\Big\}$$

$=$ set calculus

$$\bigcup\Big\{x : \mathcal{T}_1 \bullet \bigcup\{y : \mathcal{T}_2 \bullet \mathsf{Proj}_{v,h,H}.\bigcup\big\{(v', h', H') : \mathsf{Proj}_{v,h,H}.[\![P]\!].x.v.h.H \bullet [\![Q]\!].y.v'.h'.H'\}\big\}\Big\}$$

$=$ y is not free in P and x is not free in Q

$$\bigcup\Big\{x : \mathcal{T}_1 \bullet \bigcup\{y : \mathcal{T}_2 \bullet \mathsf{Proj}_{v,h,H}.\bigcup\big\{(x', y, v', h', H') : [\![P]\!].x.y.v.h.H \bullet [\![Q]\!].x'.y.v'.h'.H'\}\big\}\Big\}$$

$=$ set calculus

$$\bigcup\big\{(x, y) : \mathcal{T}_1\times\mathcal{T}_2 \bullet \mathsf{Proj}_{v,h,H}.\mathsf{lift}.[\![Q]\!].[\![P]\!].x.y.v.h.H\big\}$$

$=$ semantics of $⨾$

$$\bigcup\big\{(x, y) : \mathcal{T}_1\times\mathcal{T}_2 \bullet \mathsf{Proj}_{v,h,H}.[\![P ⨾ Q]\!].x.y.v.h.H\big\}$$

$=$ semantics of local blocks

$$[\![|[\,\mathbf{dec}\,(\mathsf{x},\mathsf{y}) : \mathcal{T}_1\times\mathcal{T}_2 \bullet P ⨾ Q\,]|]\!].v.h.H$$

□

Law 4.21.

$$(\mathsf{v},\mathsf{h} :\in E.\mathsf{v}.\mathsf{h} ⨾ \mathsf{v},\mathsf{h} :\in F.\mathsf{v}.\mathsf{h}) = \\ |[\,\mathbf{vis}\,\mathsf{w} : \mathcal{W} \bullet \mathsf{w},\mathsf{v},\mathsf{h} :\in \{(e_1, e_2) : E.\mathsf{v}.\mathsf{h};\ (f_1, f_2) : F.e_1.e_2 \bullet (e_1, f_1, f_2)\}\,]|$$

Proof.

$$[\![\mathsf{v},\mathsf{h} :\in E.\mathsf{v}.\mathsf{h} ⨾ \mathsf{v},\mathsf{h} :\in F.\mathsf{v}.\mathsf{h}]\!].v.h.H$$

$=$ semantics of $⨾$

$$\mathsf{lift}.[\![\mathsf{v},\mathsf{h} :\in F.\mathsf{v}.\mathsf{h}]\!].[\![\mathsf{v},\mathsf{h} :\in E.\mathsf{v}.\mathsf{h}]\!].v.h.H$$

$=$ semantics of $:\in$

$$\mathsf{lift}.[\![v, h :\in F.v.h]\!].\big\{(v', h') : E.v.h \bullet (v', h', \{k : H;\ k' : \mathcal{H} \mid E.v.k \ni (v', k') \bullet k'\}\big\}$$

$=$ definition of lift

$$\bigcup\big\{(v', h') : E.v.h \bullet [\![(v, h) :\in F.v.h]\!].v'.h'.\{k : H;\ k' : \mathcal{H} \mid E.v.k \ni (v', k') \bullet k'\}\big\}$$

$=$ semantics of $:\in$

$$\bigcup\Big\{(v', h') : E.v.h \bullet \big\{(v'', h'') : F.v'.h' \bullet$$

$$(v'', h'', \{k : H;\ k' : \mathcal{H} \mid E.v.k \ni (v', k');\ k'' : \mathcal{H} \mid F.v'.k' \ni (v'', k'') \bullet k''\}\}\Big\}$$

$=$ rewrite without $\cup$

$$\Big\{(v', h') : E.v.h;\ (v'', h'') : F.v'.h' \bullet$$
$$(v'', h'', \{k : H;\ k' : \mathcal{H} \mid E.v.k \ni (v', k');\ k'' : \mathcal{H} \mid F.v'.k' \ni (v'', k'') \bullet k''\}\}$$

$=$ rename $(v', h', k', v'', h'', k'')$ to $(w', e_2, e_2^{\sim}, v', h', k')$

$$\Big\{(w', e_2) : E.v.h;\ (v', h') : F.w'.e_2 \bullet$$
$$(v', h', \{k : H;\ (w', e_2^{\sim}) : E.v.k;\ k' : \mathcal{H} \mid F.w'.e_2^{\sim} \ni (v', k') \bullet k'\})\}$$

$=$ w' introduced into the triples and discarded by Proj

$$\mathsf{Proj}_{v,h,H}.\bigcup\Big\{w : \mathcal{W} \bullet \big\{(w', e_2) : E.v.h;\ (v', h') : F.w'.e_2 \bullet$$
$$(w', v', h', \{k : H;\ (w', e_2^{\sim}) : E.v.k;\ k' : \mathcal{H} \mid F.w'.e_2^{\sim} \ni (v', k') \bullet k'\})\}\Big\}$$

$=$ semantics of $:\in$

$$\mathsf{Proj}_{v,h,H}.\bigcup\Big\{w : \mathcal{W} \bullet$$
$$[\![w, v, h :\in \{(e_1, e_2) : E.v.h;\ (f_1, f_2) : F.e_1.e_2 \bullet (e_1, f_1, f_2)\}]\!].v.h.H\Big\}$$

$=$ semantics of $\sqcap$

$$[\![\ |[\ \mathbf{vis}\,\mathsf{w} : \mathcal{W} \bullet \mathsf{w}, \mathsf{v}, \mathsf{h} :\in \{(e_1, e_2) : E.\mathsf{v}.\mathsf{h};\ (f_1, f_2) : F.e_1.e_2 \bullet (e_1, f_1, f_2)\}\ ]|\ ]\!].v.h.H$$

□

Law 4.24.

$$\langle\!\langle \sqcap\{(e, f) : E.\mathsf{v}.\mathsf{h} \bullet (\mathsf{v}, \mathsf{h}) := (e, f)\}\rangle\!\rangle = (\mathsf{v}, \mathsf{h}) :\in E.\mathsf{v}.\mathsf{h}$$

Proof.

$$[\![\langle\!\langle \sqcap\{(e, f) : E.\mathsf{v}.\mathsf{h} \bullet (\mathsf{v}, \mathsf{h}) := (e, f)\}\rangle\!\rangle]\!].v.h.H$$

$=$ semantics of $\langle\!\langle\cdot\rangle\!\rangle$

$$\mathsf{addShadow}.\sqcap\{(e, f) : E.\mathsf{v}.\mathsf{h} \bullet (\mathsf{v}, \mathsf{h}) := (e, f)\}.v.h.H$$

$=$ definition of $\mathsf{addShadow}$

$$\Big\{(v', h', H') \mid$$

$$\left(\begin{array}{l} (v',h') \in [\![\sqcap\{(e,f) : E.v.h \cdot (v,h) := (e,f)\}]\!]_{\mathsf{cl}}.v.h \\ H' = \bigcup\Big\{k : H \cdot \{k' : \mathcal{H} \mid [\![\sqcap\{(e,f) : E.v.h \cdot (v,h) := (e,f)\}]\!]_{\mathsf{cl}}.v.k \ni (v',k')\}\Big\} \end{array}\right)\Bigg\}$$

$=$ classical semantics of $:\in$

$$\Bigg\{(v',h',H') \mid \left(\begin{array}{l} (v',h') \in E.v.h \\ H' = \bigcup\Big\{k : H \cdot \{k' : \mathcal{H} \mid E.v.k \ni (v',k')\}\Big\} \end{array}\right)\Bigg\}$$

$=$ redundant (v',h',H')

$$\Big\{(v',h') : E.v.h \cdot \big(v',h',\bigcup\{k : H \cdot \{k' : \mathcal{H} \mid E.v.k \ni (v',k')\}\}\big)\Big\}$$

$=$ calculus

$$\Big\{(v',h') : E.v.h \cdot \big(v',h',\{k : H;\ k' : \mathcal{H} \mid E.v.k \ni (v',k') \cdot k'\}\big)\Big\}$$

$=$ semantics of simultaneous $:\in$

$$\Big[\!\!\Big[(\mathsf{v},\mathsf{h}) :\in E.\mathsf{v}.\mathsf{h}\Big]\!\!\Big].v.h.H$$

$\square$

Theorem 4.1.

$$\sqcap\{E : \mathcal{E} \cdot \ |[\,\mathbf{vis}\,\mathsf{w} \cdot \mathsf{w},\mathsf{v},\mathsf{h} :\in E.\mathsf{v}.\mathsf{h}\,]|\ \} \ \sqsubseteq |[\,\mathbf{vis}\,\mathsf{w} \cdot \mathsf{w},\mathsf{v},\mathsf{h} :\in F.\mathsf{v}.\mathsf{h}\,]|$$

iff

$$\forall\, v,h :\in \mathcal{V}\times\mathcal{H};\ \ \forall (w',v',h') : F.v.h \cdot$$
$$\exists\, E \in \mathcal{E} \wedge \exists (w^{\sim},v',h') \in E.v.h \mid$$
$$\forall\, k : \mathcal{H} \cdot E_{0,1}.v.k \ni (w^{\sim},v') \Rightarrow \left(\begin{array}{c} F_{0,1}.v.k \ni (w',v') \Rightarrow E_2.v.k \subseteq (F_2.v.h \cup F_2.v.k) \\ F_{0,1}.v.k \not\ni (w',v') \Rightarrow E_2.v.k \subseteq F_2.v.h \end{array}\right)$$

Proof. To prove this law in the model, we need to show the shadow refinement (Definition 3.2). Translating both programs into their semantics yields for the lhs

$$[\![\sqcap\{E : \mathcal{E} \cdot \ |[\,\mathbf{vis}\,\mathsf{w} : \mathcal{W} \cdot \mathsf{w},\mathsf{v},\mathsf{h} :\in E.\mathsf{v}.\mathsf{h}\,]|\ \}]\!].v.h.H$$

$=$ semantics

$$\bigcup\Big\{E : \mathcal{E} \cdot \{w',v',h' : E.v.h \cdot \big(v',h',\{k : H;\ k' : \mathcal{H} \mid E.v.k \ni (w',v',k') \cdot k'\}\}\Big\}$$

$=$ rename w' to $w^{\sim}$

$$\bigcup\Big\{E : \mathcal{E} \cdot \{w^{\sim},v',h' : E.v.h \cdot \big(v',h',\{k : H;\ k' : \mathcal{H} \mid E.v.k \ni (w^{\sim},v',k') \cdot k'\}\big)\}\Big\}$$

and for the rhs

$$\llbracket |[\,\mathbf{vis}\,\mathsf{w}\bullet \mathsf{w},\mathsf{v},\mathsf{h} :\in F.\mathsf{v}.\mathsf{h}\,]| \rrbracket.v.h.H$$

$=$ semantics

$$\left\{w', v', h' : F.v.h \bullet \left(v', h', \{k : H;\ k' : \mathcal{H} \mid F.v.k \ni (w', v', k') \bullet k'\}\right)\right\}$$

By the definition of refinement in the shadow semantics: for any initial triples (v, h, H) each final triples in the rhs must correspond to a final triple in the lhs whose values of the variables are the same but with no larger shadow. Using the semantics of the programs given above we have:

$\bigsqcap \left\{E : \mathcal{E} \bullet\ |[\,\mathbf{vis}\,\mathsf{w}\bullet \mathsf{w},\mathsf{v},\mathsf{h} :\in E.\mathsf{v}.\mathsf{h}\,]|\ \right\}\ \sqsubseteq |[\,\mathbf{vis}\,\mathsf{w}\bullet \mathsf{w},\mathsf{v},\mathsf{h} :\in F.\mathsf{v}.\mathsf{h}\,]|$

$\equiv$ definition of refinement in the semantics

$\forall (v, h, H) \mid H \ni h \bullet$

$\forall (v', h', \{k : H;\ k' : \mathcal{H} \mid F.v.k \ni (w', v', k') \bullet k'\}) \mid w', v', h' \in F.v.h \bullet$

$\exists\, E \in \mathcal{E} \wedge \exists (v', h', \{k : H;\ k' : \mathcal{H} \mid E.v.k \ni (w^{\sim}, v', k') \bullet k'\}) \mid w^{\sim}, v', h' \in E.v.h \bullet$

$\{k : H;\ k' : \mathcal{H} \mid E.v.k \ni (w^{\sim}, v', k') \bullet k'\} \subseteq \{k : H;\ k' : \mathcal{H} \mid F.v.k \ni (w', v', k') \bullet k'\}$

$\equiv$ $v', k', w', w^{\sim}$ are the only free variable inside the shadows

$\forall (v, h, H) \mid H \ni h \bullet$

$\forall\, w', v', h' : F.v.h \bullet$

$\exists\, E \in \mathcal{E} \wedge \exists\, w^{\sim}, v', h' \in E.v.h \mid$
$\{k : H;\ k' : \mathcal{H} \mid E.v.k \ni (w^{\sim}, v', k') \bullet k'\} \subseteq \{k : H;\ k' : \mathcal{H} \mid F.v.k \ni (w', v', k') \bullet k'\}$

$\equiv$ $E_2.v.k = \mathsf{Proj}_2.E.v.k = \{k' : \mathcal{H} \mid (w^{\sim}, v', k') \in E.v.k\}$, same for $F_2.v.k$

$\forall (v, h, H) \mid H \ni h \bullet$

$\forall\, w', v', h' : F.v.h \bullet$

$\exists\, E \in \mathcal{E} \wedge \exists\, w^{\sim}, v', h' \in E.v.h \mid$
$\bigcup \{k : H \mid E_{0,1}.v.k \ni (w^{\sim}, v') \bullet E_2.v.k\} \subseteq \bigcup \{k : H \mid F_{0,1}.v.k \ni (w', v') \bullet F_2.v.k\}$

$\equiv$ definition of lift

$\forall (v, h, H) \mid H \ni h \bullet$

$\forall\, w', v', h' : F.v.h \bullet$
$\exists\, E \in \mathcal{E} \wedge \exists\, w^{\sim}, v', h' \in E.v.h \mid$

$\mathsf{lift}.E_2.v.\{k : H \mid E_{0,1}.v.k \ni (w^{\sim}, v')\} \subseteq \mathsf{lift}.F_2.v.\{k : H \mid F_{0,1}.v.k \ni (w', v')\}$

It is necessary and sufficient to prove that the inequality holds for singleton Hs.

In fact we can prove particular cases of H to be a sufficient condition. It is sufficient consider only the sets of pair $\{h, k\}$ for $k \in \mathcal{H}$.

Indeed, assume that

$$\forall\, k : \mathcal{H} \bullet F.(\{h, k\} \cap A) \subseteq G.(\{h, k\} \cap B).$$

A shadow H is of the form $\{h, k_1, \ldots, k_n\}$, then

$$\begin{aligned} F.(\{h\} \cap A) &\subseteq G.(\{h, k\} \cap B) \\ F.(\{h, k_1\} \cap A) &\subseteq G.(\{h, k_1\} \cap B) \\ &\vdots \\ F.(\{h, k_n\} \cap A) &\subseteq G.(\{h, k_n\} \cap B) \end{aligned}$$

$\Rightarrow$ $\qquad f(A) \cup f(B) = f(A \cup B)$

$$F.(\{h, k_1, \ldots, k_n\} \cap A) \subseteq G.(\{h, k_1, \ldots, k_n\} \cap B).$$

Therefore we can write

$\bigsqcap \{E : \mathcal{E} \bullet\ |[\,\mathbf{vis}\,\mathsf{w} \bullet \mathsf{w}, \mathsf{v}, \mathsf{h} :\in E.\mathsf{v}.\mathsf{h}\,]|\ \} \sqsubseteq |[\,\mathbf{vis}\,\mathsf{w} \bullet \mathsf{w}, \mathsf{v}, \mathsf{h} :\in F.\mathsf{v}.\mathsf{h}\,]|$

$\equiv$

$\forall\, v, h :\in \mathcal{V} \times \mathcal{H};\ \forall\, k : \mathcal{H}$

$\forall\, w', v', h' : F.v.h \bullet$

$\exists\, E \in \mathcal{E} \wedge \exists\, w^{\sim}, v', h' \in E.v.h \mid$

$\mathsf{lift}.E_2.v.\big(\{h, k\} \cap \{k^{\sim} : \mathcal{H} \mid E_{0,1}.v.k \ni (w^{\sim}, v')\}\big)$

$\subseteq$

$\mathsf{lift}.F_2.v.\big(\{h, k\} \cap \{k^{\sim} : \mathcal{H} \mid F_{0,1}.v.k \ni (w', v')\}\big)$

$\equiv$ $\qquad \{h, k\} \subseteq \mathcal{H}$

$\forall\, v, h :\in \mathcal{V} \times \mathcal{H};\ \forall\, k : \mathcal{H}$

$\forall\, w', v', h' : F.v.h \bullet$

$\exists\, E \in \mathcal{E} \wedge \exists\, w^{\sim}, v', h' \in E.v.h \mid$

$\mathsf{lift}.E_2.v.\big(k^{\sim} : \{h, k\} \mid E_{0,1}.v.k \ni (w^{\sim}, v')\}\big) \subseteq \mathsf{lift}.F_2.v.\big(k^{\sim} : \{h, k\} \mid F_{0,1}.v.k \ni (w', v')\}\big)$

$\equiv$ $(\forall\, k : \mathcal{H} \bullet p.k) \Rightarrow p.h$

$\forall\, v, h :\in \mathcal{V}{\times}\mathcal{H};\ \forall\, k : \mathcal{H}$

$\forall\, w', v', h' : F.v.h \bullet$

$\exists\, E \in \mathcal{E} \wedge \exists\, w^{\sim}, v', h' \in E.v.h \mid$

$E_2.v.h \subseteq F_2.v.h \wedge$

$\mathsf{lift}.E_2.v.\big(k^{\sim} : \{h, k\} \mid E_{0,1}.v.k \ni (w^{\sim}, v')\}\big) \subseteq \mathsf{lift}.F_2.v.\big(k^{\sim} : \{h, k\} \mid F_{0,1}.v.k \ni (w', v')\}\big)$

$\equiv$ $q \equiv (p \Rightarrow q) \wedge (\neg p \Rightarrow q)$

$\forall\, v, h :\in \mathcal{V}{\times}\mathcal{H};\ \forall\, k : \mathcal{H}$

$\forall\, w', v', h' : F.v.h \bullet$

$\exists\, E \in \mathcal{E} \wedge \exists\, w^{\sim}, v', h' \in E.v.h \mid$

$E_2.v.h \subseteq F_2.v.h \wedge$

$E_{0,1}.v.k \ni (w^{\sim}, v') \Rightarrow$

$\mathsf{lift}.E_2.v.\big(k^{\sim} : \{h, k\} \mid E_{0,1}.v.k \ni (w^{\sim}, v')\}\big) \subseteq \mathsf{lift}.F_2.v.\big(k^{\sim} : \{h, k\} \mid F_{0,1}.v.k \ni (w', v')\}\big) \wedge$

$E_{0,1}.v.k \not\ni (w^{\sim}, v') \Rightarrow$

$\mathsf{lift}.E_2.v.\big(k^{\sim} : \{h, k\} \mid E_{0,1}.v.k \ni (w^{\sim}, v')\}\big) \subseteq \mathsf{lift}.F_2.v.\big(k^{\sim} : \{h, k\} \mid F_{0,1}.v.k \ni (w', v')\}\big)$

$\equiv$ $(p \Rightarrow q) \Rightarrow (p \Rightarrow (p \wedge q))$

$\forall\, v, h :\in \mathcal{V}{\times}\mathcal{H};\ \forall\, k : \mathcal{H}$

$\forall\, w', v', h' : F.v.h \bullet$

$\exists\, E \in \mathcal{E} \wedge \exists\, w^{\sim}, v', h' \in E.v.h \mid$

$E_2.v.h \subseteq F_2.v.h \wedge$

$E_{0,1}.v.k \ni (w^{\sim}, v') \Rightarrow \mathsf{lift}.E_2.v.\{h, k\} \subseteq \mathsf{lift}.F_2.v.\big(k^{\sim} : \{h, k\} \mid F_{0,1}.v.k \ni (w', v')\}\big) \wedge$

$E_{0,1}.v.k \not\ni (w^{\sim}, v') \Rightarrow \mathsf{lift}.E_2.v.\{h\} \subseteq \mathsf{lift}.F_2.v.\big(k^{\sim} : \{h, k\} \mid F_{0,1}.v.k \ni (w', v')\}\big)$

$\equiv$ $E_2.v.h \subseteq F_2.v.h \subseteq \mathsf{lift}.F_2.v.\big(k^{\sim} : \{h, k\} \mid F_{0,1}.v.k \ni (w', v')\}\big)$

$\forall\, v, h :\in \mathcal{V}{\times}\mathcal{H};\ \forall\, k : \mathcal{H}$

$\forall\, w', v', h' : F.v.h \bullet$

$\exists\, E \in \mathcal{E} \wedge \exists\, w^{\sim}, v', h' \in E.v.h \mid$

$E_2.v.h \subseteq F_2.v.h \wedge$

$E_{0,1}.v.k \ni (w^{\sim}, v') \Rightarrow \mathsf{lift}.E_2.v.\{h, k\} \subseteq \mathsf{lift}.F_2.v.\big(k^{\sim} : \{h, k\} \mid F_{0,1}.v.k \ni (w', v')\}\big) \wedge$

$E_{0,1}.v.k \not\ni (w^{\sim}, v') \Rightarrow \mathbf{true}$

$\equiv$ $q \equiv (p \Rightarrow q) \wedge (\neg p \Rightarrow q)$

$\forall\, v, h :\in \mathcal{V}{\times}\mathcal{H};\ \forall\, k : \mathcal{H}$

$\forall\, w', v', h' : F.v.h \bullet$

$\exists\, E \in \mathcal{E} \wedge \exists\, w^{\sim}, v', h' \in E.v.h \mid$
$E_2.v.h \subseteq F_2.v.h \wedge$
$E_{0,1}.v.k \ni (w^{\sim}, v') \Rightarrow$
$\big(F_{0,1}.v.k \ni (w', v') \Rightarrow \mathsf{lift}.E_2.v.\{h,k\} \subseteq \mathsf{lift}.F_2.v.\big(k^{\sim} : \{h,k\} \mid F_{0,1}.v.k \ni (w',v')\}\big) \wedge$
$F_{0,1}.v.k \not\ni (w', v') \Rightarrow \mathsf{lift}.E_2.v.\{h,k\} \subseteq \mathsf{lift}.F_2.v.\big(k^{\sim} : \{h,k\} \mid F_{0,1}.v.k \ni (w',v')\}\big)\big)$

$\equiv$ $(p \Rightarrow q) \Rightarrow (p \Rightarrow (p \wedge q))$

$\forall\, v, h :\in \mathcal{V} \times \mathcal{H};\ \forall\, k : \mathcal{H}$
$\forall\, w', v', h' : F.v.h \bullet$
$\exists\, E \in \mathcal{E} \wedge \exists\, w^{\sim}, v', h' \in E.v.h \mid$
$E_2.v.h \subseteq F_2.v.h \wedge$
$E_{0,1}.v.k \ni (w^{\sim}, v') \Rightarrow$
$\big(F_{0,1}.v.k \ni (w', v') \Rightarrow \mathsf{lift}.E_2.v.\{h,k\} \subseteq \mathsf{lift}.F_2.v\{h,k\} \wedge$
$F_{0,1}.v.k \not\ni (w', v') \Rightarrow \mathsf{lift}.E_2.v.\{h,k\} \subseteq \mathsf{lift}.F_2.v.\{h\}\big)$

$\equiv$ definition of lift

$\forall\, v, h :\in \mathcal{V} \times \mathcal{H};\ \forall\, k : \mathcal{H}$
$\forall\, w', v', h' : F.v.h \bullet$
$\exists\, E \in \mathcal{E} \wedge \exists\, w^{\sim}, v', h' \in E.v.h \mid$
$E_2.v.h \subseteq F_2.v.h \wedge$
$E_{0,1}.v.k \ni (w^{\sim}, v') \Rightarrow$
$\big(F_{0,1}.v.k \ni (w', v') \Rightarrow (E_2.v.h \cup E_2.v.k) \subseteq (F_2.v.h \cup F_2.v.k) \wedge$
$F_{0,1}.v.k \not\ni (w', v') \Rightarrow (E_2.v.h \cup E_2.v.k) \subseteq F_2.v.h\big)$

let $A = E_2.v.h,\ B = F_2.v.h,\ C = E_2.v.k$ and $D = F_2.v.k$
$\equiv$ then $(A \subseteq B) \wedge (A \cup C \subseteq B \cup D) \Rightarrow C \cup (B \cup D)$
and $(A \subseteq B) \wedge (A \cup C \subseteq B) \Rightarrow C \cup B$

$\forall\, v, h :\in \mathcal{V} \times \mathcal{H};\ \forall\, k : \mathcal{H}$
$\forall\, w', v', h' : F.v.h \bullet$
$\exists\, E \in \mathcal{E} \wedge \exists\, w^{\sim}, v', h' \in E.v.h \mid$
$E_2.v.h \subseteq F_2.v.h \wedge$
$E_{0,1}.v.k \ni (w^{\sim}, v') \Rightarrow$
$\big(F_{0,1}.v.k \ni (w', v') \Rightarrow E_2.v.k \subseteq (F_2.v.h \cup F_2.v.k) \wedge$
$F_{0,1}.v.k \not\ni (w', v') \Rightarrow E_2.v.k \subseteq F_2.v.h\big).$

This corresponds to the condition in our theorem as the condition $E_2.v.h \subseteq F_2.v.h$ is included in the main condition.

$\square$

List of References

[1] Ralph-Johan J Back, Abo Akademi, and J Von Wright. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1998.

[2] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[3] C.A.R. Hoare, I.J. Hayes, He Jifeng, C.C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sorensen, J.M. Spivey, and B.A. Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–686, 1987.

[4] C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming.* Prentice Hall International Series in Computer Science. 1998.

[5] A. McIver, L. Meinicke, and C. Morgan. Security, probability and nearly fair coins in the cryptographers' café. *FM 2009: Formal Methods*, pages 41–71, 2009.

[6] A. McIver, L. Meinicke, and C. Morgan. Compositional closure for bayes risk in probabilistic noninterference. *Automata, Languages and Programming*, pages 223–235, 2010.

[7] A.K. Mciver. The Secret Art of Computer Programming. In *Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing*, ICTAC '09, pages 61–78. Springer-Verlag, 2009.

[8] A.K. McIver and C.C. Morgan. Compositional refinement in agent-based security protocols. *Formal Aspect of Computing*, 23(6):711–737, 2011.

[9] C.C. Morgan. *Programming from Specifications.* Prentice Hall International Series in Computer Science. 2 edition, 1994.

[10] C.C. Morgan. The Shadow Knows: Refinement of ignorance in sequential programs. In *Mathematics of Program Construction*, Lecture Notes in Computer Science, pages 359–378. Springer, 2006.

[11] C.C. Morgan. The Shadow Knows: Refinement and security in sequential programs. *Science of Computer Programming*, 74:629–653, 2009.

[12] C.C. Morgan. Compositional noninterference from first principles. *Formal Aspect of Computing*, 24(1):3–26, 2012.