Information Structure: Formalisation of the Notions of Access and of Data

Jean-Luc Remy

25 June 1974

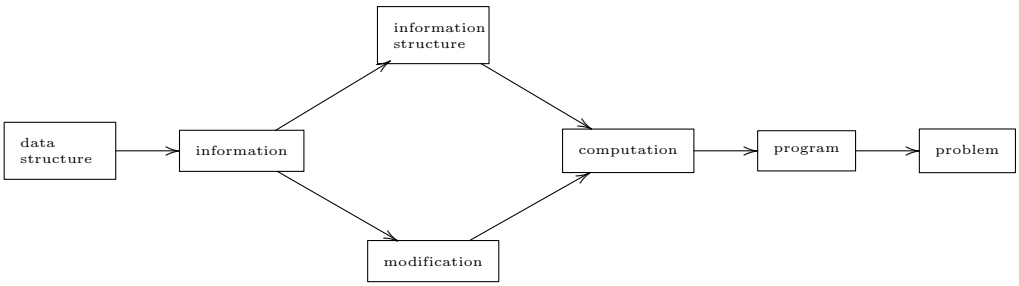
Foreword for the translation

This is a faithful translation of parts of the *Thèse de troisième cycle* of Jean-Luc Rémy entitled *Stuctures d'information, formalisation des notions d'accès et de modification d'une structure de donnée* defended at the University of Nancy (France) on the 24th June of 1974.

Notice that by lack of time, I translated only the parts that I found the most interesting, but this is a matter of taste. The footnotes I wrote during the translation are prefixed by *Note of the translator*. Possibly I made mistakes during the translation. In case of doubt, do not hesitate to send a mail at pierre.lescanne@ens-lyon.fr.

In this translation I decided to translate the French word "modification" by the English word "modification" (a lazy decision for such an important concept here), but I could have chosen "update" as well, which would be consistent with the modern literature.

If you think that this thesis has been written in 1973-1974, you are amazed by the clear-seeing of Jean-Luc Rémy. Many problems of computer science are addressed. The mathematical approach is clear and rigorous using tools known at that time. There is strict hierarchy of concepts from data, to information, to information structure and modification, to computation, to program and to problem as shows in the above picture:



Reader has to remember that logic for computer science and semantics of programming language were in their infancy. Mathematical logic in France has been basically killed by Bourbaki, who was born in Nancy and obviously not thought, in his city. This is summarised in a text due to Jean Dieudonné and

called *The Bourbaki choice*. Concerning other locations, for instance, the conference STOC (Symposium on Theory of Computing), the first conference on theoretical computer science, was created in 1969 and was mostly oriented toward complexity, POPL (Principles of Programming Languages) in 1973, other conferences in TCS were created after 1975. For all these reasons, I do not put the bibliography because it was extremely poor and carries no interesting information to any connected research, as there was none. People in Nancy worked in autonomy. When I read the document, like a translator does, I noticed that sometime things are repeated, with subtle differences. Reader has to be indulgent, because she has to remember how those documents were written. They were first handwritten, corrections at this level were difficult, for instance systematic corrections, like changing a notation, were not possible or were extremely tedious and error-prone. Then the document was typed by another person who was not a scientist. When the text was seen by its author in its almost final version, corrections were not possible (unlike now) and only signs that were not typed were added with a pencil, like calligraphic letters or mathematical signs.

January-February 2019
Pierre Lescanne

Introduction

The main aim of this work is to offer a formal framework for the study of computer science **problems**. We mean by problem a transition from a given **information** to another. More generally, a problem applies to an information of some *type*. Thus we have to make precise the notion of information, of data and of type of information (or information structure)

A preliminary idea studied in Section 1.1. is to define a **data** first by sets $E_1, ..., E_n$ and second by a set Λ of applications (or **accesses**) defined on sets of the form E_{i_1}, ..., E_{i_q} with values in one of the E_i's. We say that two data are of same type if their accesses satisfy the same properties.

A way to express the properties of the accesses of a data is to build a **formal system** in which each application of Λ is represented by a symbol and each object of E_1, ..., E_n by a functional scheme. In Chapter 0, we survey language theory, functional schemes, formal systems and propositional calculus. The definitions of **functional schemes**, of **informations** and of **interpretations** are given in Section 1.2.

This axiomatic method allows us to define information structures, the same way we define in mathematics group structure or field structure. We try, in this work, to use this method for other applications. We define systems of axioms for *informations* (Chapter 1), for *modifications* of an *information structure* (Chapter 5) and for recursive definitions of new *accesses* (Chapter 3) and (Chapter 4)

In Chapter 2, we present classical results in logic on the theorems of a functional system. In particular, we compare functional systems with two types of formal systems: propositional calculus and first order predicate calculus with equality. Eventually we exhibit two types of informations of interest, namely consistent informations and complete informations.

In Chapter 3, we address extensions of an information structure. After giving, in the first section, general properties, we focus on conservative extensions. Last, in Section 3 we study on the example of recursive modes of ALGOL 68, a few techniques of construction of interpretations.

In Chapter 4, we focus on systems of recursive equations. In the framework of formal systems, we get most of the classical results. In particular, it is possible to define a fixed-point induction rule similar to Scott's. Our work allows us, thanks to a system of axioms, to make precise the basic domains.

Chapter 5 is devoted to the study of *modifications*[1]. An information structure is completely specified by a formal system which makes precise the properties of the accesses and of a set of **elementary modifications**, that are applications defined on the set of informations. A modification can be obtained, either by composition of elementary modifications, or as a solution of a system of recursive equations. We give an axiomatic method for defining modifications. Last, to conclude the chapter we study shortly the notions of program, of computation and of problem.

We develop in the conclusion, points which might be addressed, or simply mentioned. It is very likely that definitions of the present document are not adequate, because it is a first try of formalisation, moreover it seems necessary to study, in this framework, more complex data. We end the conclusion by mentioning related works.

Warning The use of functional symbols with many variables and of objects of distinct types requires generally somewhat heavy notations. We use generally short notations which allow us to work like with one variable. The reader finds at the end of this document a notation index and a terminological index[2].

[1] *Note of the translator:* or *update*.
[2] *Note of the translator:* not translated.

Contents

Chapter 0

Survey

This chapter is not translated yet.

Chapter 1

Information structures

1.1 Functional point of view (Claude Pair and Jean-Pierre Finance)

1.1.1 Examples of data – Definitions

Let us try on examples to know what a composed information is.

Lists A list on a set E is a finite sequence of elements of E. To make precise this notion of sequence and, in particular, to express the accesses to the elements of such a sequence, we introduce a linearly ordered set F (set of "locations" in the list) of which the first element τ is the head and the last element δ is the tail of the list. The linear order is given by a succession function in F, written σ, which is a bijection of $F - \{\delta\}$ to $F - \{\tau\}$. Therefore, each element of F is the image of τ by σ^i for some i. Moreover a function ν assigns to each element of F a value in E.

Therefore a list is a triple (F, E, Λ) where Λ is the set of functions $\{\sigma, \nu, \tau\}$ (τ is considered here as a function with 0 variable. The access to any element goes through the head). Consing an element to a list (F, E, Λ) leads to a new list (F', E', Λ') by setting

$$\begin{aligned} F' &= F \cup \{\tau'\} \\ \Lambda' &= \{\sigma', \nu', \tau'\} \\ \sigma'(\tau') &= \tau \\ \sigma'(x) &= \sigma(x) \qquad \text{if } x \in F - \{\delta\} \\ \nu'(x) &= \nu(x) \qquad \text{if } x \in F \end{aligned}$$

Actually, we obtain a quasi-list, since the value assigned to τ' is not defined. We notice on this simple example that it is not easy to define the modifications from this point of view.

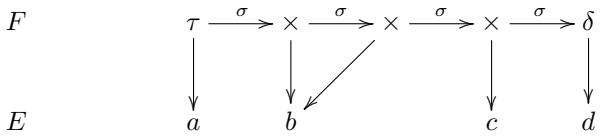


Figure 1.1: Picture of the list *abbcd*

Management of products in stocks A plant has a set p of products stored in a set d of stores.[1] For each p one has a list $d_1(p), .., d_{k_p}(p)$ of stores in which this product is stored and, for each store d, we make a list of products $p_1(d), ..., p_{\ell_d}(d)$ stored in d. Furthermore, for each product and each store we know the quantity of products stored in this store.

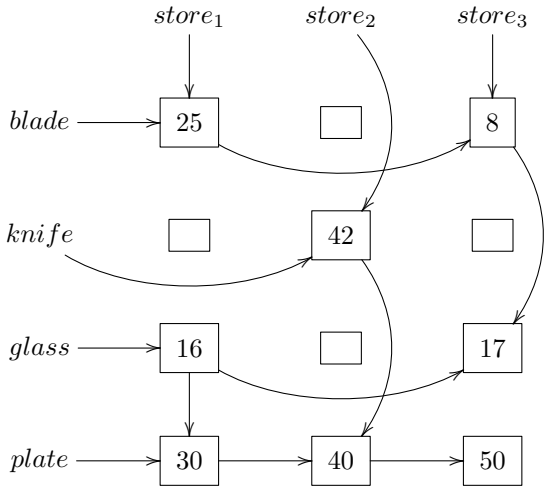


Figure 1.2: management of a stock

We can present this data by considering the following sets:

$$\begin{aligned} D &= \text{set of stores} \\ P &= \text{set of products} \\ F &= D \times P \\ \mathcal{Q} &= \text{set of quantities} \end{aligned}$$

and the following functions:

τ_1 associates with each store d the couple $(d, p_1(d))$

τ_2 associates with each product p the couple $(d_1(p), p)$

σ_1 associates with (d, p) the couple (d, p') if it exists, such that p' follows p in the list $p_1(d), p_2(d), ...$

[1] *Note of the translator:* In French a store is called a "dépôt", hence the letter "d".

σ_d associates with (d, p) the couple (d', p) if it exists, such that d' follows d in the list $d_1(p), d_2(p), ...$

π_1 associates with each couple (d, p) its first component d.

π_2 associates with each couple (d, p) its second component p.

χ associates with each couple (d, p) the quantity of product p stored in d

Moreover, we can access each product and each store. Therefore we should consider D and P as sets of 0-ary functions.

Hence this data is a quintuple (D, P, F, Q, Λ) with

$$\Lambda = D \cup P \cup \{\tau_1, \tau_2, \sigma_1, \sigma_2, \pi_1, \pi_2, \chi\}.$$

Let us notice that, in Figure 1.2, couples like $(store_2, blade)$ or $(store_1, knife)$ are not "accessible". More generally, we can define a data as follows.

Definition 1. *A **data** is m+1-tuple $(E_1, ..., E_m, \Lambda)$ where the E_i's for $i = (1, ...m)$ are sets of objects and Λ is a set of (partial) functions defined on sets of the form $E_{i_1} \times ... \times E_{i_q}$ (q is a non negative integer and $1 \leq i_j \leq m$) with values in one of the E_j's. Such a function is called an **elementary access** with q variables. A function with 0 variable is considered to be a value.*

Comment 1. *It is always possible to consider total functions instead of partial ones by introducing an element $\perp$ (undefined element). If E is a set which does not contain $\perp$ we write $E_+ = E \cup \{\perp\}$. We extend a partial function f, defined on E, into a total one $\bar{f}$, by*

$$\begin{cases} \bar{f}(x) & = & \textbf{if } f(x) \text{ is defined } \textbf{then } f(x) \textbf{ else } \perp \quad (x \in E) \\ \bar{f}(\perp) & = & \perp \end{cases}$$

1.1.2 Data structures

We formalised the notion of data. We presented also examples of structures like *list* or *stock management*. In each case, we defined those structures by expressing properties satisfied by basic sets and accesses in those data. It is difficult to state general results on the structure without giving a mathematical meaning to the notion of "property". We can provide an axiomatic answer to this problem by associating a formal system to a data structure. (This is how we define group structures and field structures, etc.) The properties that a data structure must satisfy are formulas or axioms built on an alphabet: this alphabet is made of functional symbols aka accesses, of variables, of the symbol $\equiv$ and of the symbols $\supset$ (implication), $\neg$ (negation), "(", ")" (parentheses) of the proposition calculus.

A structure is not completely specified by its formal system. It is important to make precise the transformations or the elementary modifications applicable to the data of the structure. For the lists, we should be able

- to modify the value of an element,
- to cons an element at the head of the list,

- to delete an element.

This aspect is addressed in Chapter 5.

1.2 Functional systems

In the previous approach, a data $(E_1, ..., E_p, \Lambda)$ is a "concrete" object. On another hand we consider an information as an "ideal" object which allows us to express statements. In a nutshell, an information is the set of theorems of a formal system (cf 0.3). An information is interpreted (in a logical sense) by a data.

An information structure $\mathcal{S}$ is composed of a formal system (described in this section) and of a set $\mathcal{M}$ of elementary modifications (addressed in Chapter 5). In what follows, we use sometimes expression like: the alphabet of $\mathcal{S}$, the axioms of $\mathcal{S}$, the theorems of $\mathcal{S}$, the information of $\mathcal{S}$, instead of: the alphabet of $\mathcal{F}$, the axioms of $\mathcal{F}$, the theorems of $\mathcal{F}$, the information of $\mathcal{F}$.

Definition 2. *A **functional system** is a formal system*[2] $\mathcal{F} = (\mathcal{L}, F, \mathcal{X}, H)$*, which satisfies the conditions given in the next section. Formula in F are propositional formulas on a set A, that is that A satisfies the equation:*

$$F \quad = \quad A \quad \cup \quad \neg F \quad \cup \quad (F \supset F)$$

1.2.1 Alphabet

Given a non negative integer m which is the number of types of object of the system, we write $[m]$ the interval of integers between 1 and m, said otherwise $[m] = \{n \in \mathbb{N} \mid 1 \leq n \leq m\}$.

$\mathcal{L} = L \cup V \cup \{\equiv\} \cup \{(,), \neg, \supset\}$ is the alphabet of F.

$\equiv$ is the symbol of equality and its use is ruled by the axioms Eg.

L is the set of functional symbols.

$V = \bigcup_{j=1}^{m} V_j$ is the set of variables.

We define an application **source** from L to $[m]$ (cf 0.1): **source**(f) is a word on $[m]$, that is a sequence of integers between 1 and m. The length of **source**(f) is called the **arity** of f. If it is equal to $0, 1, 2, q$, the function is said to be 0-ary, unary, binary, q-ary. The 0-ary functions are also symbols of constant. We define an application **goal** from L to $[m]$. With **source** and **goal** we build a function **profile** from L to $[m]^* \times [m]$ such that

$$\textbf{profile}(f) = (\textbf{source}(f), \textbf{goal}(f))$$

Notations: In what follows, we use the letters a, b, c for the constant symbols and the letters f and g for the function symbols. $x = (x_1, ..., x_n)$ is always a n-tuple of **distinct** variables. We write $L(x) = L \cup \{x_1, ..., x_n\}$. Furthermore, we write $L' = L \cup V$.

[2] *Note of the translator:* $\mathcal{X}$ are the axioms, and H are the inference rules, described in Chapter 0.

The function symbols are the elementary **accesses** of the system. This is justified by the fact that the objects of the systems are obtained by composition of theses accesses (see below, especially Definition 5).

Example 1. *In the list structure we consider two types of objects (the places and the values).* $L = \{t, s, v, \mathbf{nil}, \mathbf{nil}'\}$

$$\begin{array}{lll} \mathbf{profile}(t) = (1) & \mathbf{profile}(\mathbf{nil}) = (1) & \mathbf{profile}(\mathbf{nil}') = (2) \\ \mathbf{profile}(s) = (1,1) & \mathbf{profile}(v) = (1,2) & \end{array}$$

Comment 2. nil *is the dummy element which is put conveniently as the end of the list. We set* $\sigma(\delta) = \sigma(\mathbf{nil}) = \mathbf{nil}$ *and* $\nu(\mathbf{nil}) = \mathbf{nil}'$.

1.2.2 Functional schemes of the system

We consider only some functional schemes (see 0.2), namely those that are obtained by taking, in the composition, profiles into account. Let us introduce now a few notations.

Notations:

1. If $A_1, ..., A_m$ are sets, if $i = (i_1, ...i_n) \in [m]^*$, we write $A_i = A_{i_1} \times ... \times A_{i_n}$. In what follows, i is always an element of $[m]^*$.
2. If $u = (u_1, ...u_n) \in (L'^*)^n$ and if $f \in L$, we write fu instead of $fu_1...u_n$.
3. If $A \subseteq (L'^*)^n$ then $fA = \{fu \mid u \in A\}$.

Definition 3. *A functional scheme compatible with the function* **profile** *is an element of the language* S' *on* L' *defined by* $S' = \bigcup_{j=1}^{m} S'_j$ *where* $(S'_1, ..., S'_m)$ *is the unique solution of the system*

$$S'_j = \bigcup \{f S'_i \mid f \in L \wedge \mathbf{profile}(f) = (i, j)\} \cup V_j$$

We call ***terms*** *the functional schemes compatible with* **profile**. *A function scheme is of* ***type*** j *if* $u \in S'_j$

For $x = (x_1, ...x_n)$ we write $S_j(x) = S'_j \cap L(x)^*$. Specifically, $S_j = S_j(\varepsilon)$ is the set of the variable-free terms[3] of type j. We write $S = \bigcup_{j=1}^{m} S_j$ and $S(x) = \bigcup_{j=1}^{m} S_j(x)$.

We assume that $S_j \neq \emptyset$ **for** $j = 1, ..., m$.

Example 2. *In the list structure, the variable-free terms are the elements of the set* S *defined by*

$$\begin{array}{rcl} S & = & S_1 \cup S_2 \\ S_1 & = & sS_1 \cup \{t, \mathbf{nil}\} \\ S_2 & = & vS_1 \cup \{\mathbf{nil}'\} \end{array}$$

[3] *Note of the translator:* These *variable-free terms* are those we call today *ground terms* or *closed terms.*

Notations: We use the letters u, v, w, to denote functional schemes or n-tuples of functional schemes. When these functional schemes contain variables we use the letters h or k.

Let h be a functional scheme, i be $(i_1, ..., i_n) \in [m]^*$, x be $(x_1, ..., x_n)$ and u be $(u_1, ...u_n) \in S'_j$. We write $h_x[u]$ (or $h[u]$ if there is no ambiguity) the term obtained by replacing in h all the occurrences of $x_1, ...x_n$ by $u_1, ...u_n$ respectively. We assume $\mathbf{type}(u) = \mathbf{type}(x) = i$.

Definition 4. *An **interpretation of** S is a m+1-tuple $E = (E_1, ..., E_m, E_{m+1})$ where the E_i's are non empty sets and r is an application associating with each $f \in L$ such that* $\mathbf{profile}(f) = (i, j)$ *an application from E_i into E_j. If $E = (E_1, ..., E_m)$ we write $R = (E, r)$.*

We use in what follows the λ-notation. Let E be $(E_1, ..., E_m)$, $i \in [m]^*$ and x be $(x_1, ..., x_n) \in V_i$. If $u(x)$ is an expression for a function, $\lambda x.u(x)$ represents the function which associates $u(x)$ with all x.

Example 3. *The notation $\lambda x.x_k$ represents the function π_k which is the function which associates x_k with $(x_1, ..., x_n) \in E_i$.*

Proposition 1. *Let (E, r) be an interpretation of S and x be $(x_1, ..., x_n)$. Then r determines a unique application $\hat{r}$ from $S(x)$ to the set of applications from E_i to $\bigcup_{i=1}^{m} E_i$ such that*

$\hat{r}(x_k) = \lambda x.x_k$ *for $k = 1, ...m$*

$\hat{r}(a) = \lambda x.a$ *if a is a constant,*

$\hat{r}(g\ u_1 ...\ u_m) = r(g)(\hat{r}(u_1), ..., \hat{r}(u_n))$ *if $g \in L$*

The proof is straightforward and similar to the proof for general functional schemes (Section 0.2).

Comment 3. *Strictly speaking we should write $\hat{r}_x$ instead of $\hat{r}$. However let us notice that if $(x_1, ..., x_n) \subseteq (y_1, ..., y_p)$ and if $u \in S(x)$ then we can assimilate $\hat{r}_x(u)$ to $\hat{r}_y(u)$. In particular if u is a variable free term of type j ($u \in S_j$), then $\hat{r}(u)$ is a constant application which we assimilate to its value.*

Definition 5. *An interpretation (E, r) of S is **strict** if $\hat{r}(S_j) = E_j$ for $j \in 1, ...m$.*

1.2.3 Atomic formulas

The set A of atomic formulas is defined by

$$A = \bigcup_{j=1}^{m} \quad S'_j \equiv S'_j$$

Predicates: Assume that L contains a symbol *true* of type j. We call propositional symbol (or predicate) every symbol of type j. From now on, we write $p\ u_1 ...\ u_n$ instead of $p\ u_1 ...\ u_n = true$. This is only a rewrite. Similarly $p \vee q$ is a rewrite of $\neg p \supset q$.

Example 4. *The system $\mathcal{N}$ formalising arithmetic has two types of objects. The type* integer *and the type* boolean*. The symbols of $\mathcal{N}$ are: (the profiles are given between parentheses)*

$$\begin{array}{rlllllll} 0 & (1) & S & (1) & + & (1,1,1) & * & (1,1,1) \\ true & (2) & < & (1,1,2) & & & & \end{array}$$

The formula $S0 < 0$ is a rewrite of the formula $< (S0, 0) \equiv true$.

In what follows we drop the type when it corresponds to a boolean and we write a predicate by providing just $\mathbf{source}(p)$. In the above example $\mathbf{source}(<) = (1, 1)$. Similarly we write $x + y$ instead of $+xy$ and $x * y$ instead of $*xy$.

1.2.4 Formulas

The set F is the solution of the fixed-point equation:

$$F \quad = \quad A \ \cup \ \neg F \ \cup \ F \supset F$$

Likewise we could have said:

1. An atomic formula is a formula,
2. If p is a formula then $\neg p$ is a formula,
3. If p and q are formulas then $p \supset q$ is a formula,
4. Every formula is obtained by the three above rules.

In order to prove that a property P is true, it is sufficient to prove (reasoning by induction of the length of the formula) that

1. P is satisfied for every atomic formula,
2. If $P(p)$ is satisfied then $P(\neg p)$ is satisfied,
3. If $P(p)$ and $P(q)$ are satisfied, then $P(p \supset q)$ is satisfied.

Recall the rewriting rules stated in Section 2.1. We can write:

$$\begin{array}{lll} p \vee q & \text{for} & \neg p \supset q \\ p \wedge q & \text{for} & \neg(\neg p \vee \neg q) \\ p \leftrightarrow q & \text{for} & (p \supset q) \wedge (q \supset q) \end{array}$$

Similarly we write $u \not\equiv v$ instead of $\neg(x \equiv y)$.

1.2.5 Axioms

$\mathcal{X} = Prop(A) \cup Eg \cup X$ is the set of axioms of $\mathcal{F}$.

- The formulas of $Prop(A)$ are the axioms of the propositional system built on A (cf Section 0.4).
- The formulas of Eg define the equality $\equiv$. For each j of $[m]$, let x_j, y_j and z_j be three variables of type j.

  More generally, for each $i \in [m]^*$, such that $q = |i| > 0$, let $x_{i_1}, ..., x_{i_q}$ and $y_{i_1}, ..., y_{i_q}$ be two sequences of distinct variables of type i, such that $x_{i_k} \neq x_{i_\ell}$ for $k, \ell \in [1, q]$. Assume

  - $p_j \quad = \quad x_j \equiv x_j$ for $j \in [m]$
  - $q_j \quad = \quad x_j \equiv y_j \supset x_j \equiv z_j \supset y_j \equiv z_j$ for $j \in [m]$, and
  - if f is a q-ary symbol ($q \neq 0$) such that $\mathbf{source}(f) = i$

$$r_f \quad = \quad x_{i_1} \equiv y_{i_1} \supset ... \supset x_{i_q} \equiv y_{i_q} \supset f x_{i_1} ... x_{i_q} \equiv f y_{i_1} ... y_{i_q}$$

$$Eg \quad = \quad \{p_j \mid j \in [m]\} \cup \{q_j \mid j \in m]\} \cup \{r_f \mid f \in L \text{ and } \mathsf{arity}(f) > 0\}$$

- The formulas X are the proper formulas of the functional system.

Example 5.

1. *The proper axioms of the* list *structure are*

   - $sx \equiv sy \supset x \equiv y \vee sx \equiv \mathbf{nil}$
   - $sx \equiv t \supset t \equiv \mathbf{nil}$
   - $s\mathbf{nil} \equiv \mathbf{nil}$
   - $vx \equiv \mathbf{nil}' \leftrightarrow x \equiv \mathbf{nil}$

   Those axioms mean that

   - *either the list is infinite, in this case* $s^k t \not\equiv s^\ell t$ $(k \neq \ell)$
   - *or the list is of finite length , in this case* $s^n t \equiv \mathbf{nil}$ *and* $s^k t \not\equiv s^\ell t$ $(0 \leq k, \ell < n)$ $(k \neq \ell)$.

2. *The proper axioms of the structure* $\mathcal{N}$ *(non negative integers) are Peano axioms:*

$Sx \not\equiv 0$	$x.(Sy) \equiv x + x.y$
$Sx \equiv Sy \supset x \equiv y$	$\neg(x < 0)$
$x + 0 \equiv x$	$x < Sy \leftrightarrow (x < y \vee x \equiv y)$
$S(x + y) \equiv (Sx) + y$	$x < y \ \vee \ x \equiv y \ \vee \ y < x$
$x.0 = 0$	

3. *In each structure it is convenient to introduce for $(i,j) \in [m] \times [m]$ a function $cond_{i,j}$ of profile (i,i,j,j,j) formalising the "conditional" function. The axioms of $cond_{i,j}$ are:*

$$cond_{i,j}(x,x,z,t) \equiv z$$

$$x \not\equiv y \supset cond_{i,j}(x,y,z,t) \equiv t$$

In what follows, we omit the indices i,j, since the context tells them.

1.2.6 Inference rules

We have two inference rules:

The *modus ponens* $p, p \supset q \vdash q$.

The *substitution* $p \vdash p_x[u]$ ($p \in F$, $u \in V$, $\mathbf{type}(u) = \mathbf{type}(x)$).

Let us notice that if $p \in T$, the formula scheme $\{p_x[u] \mid u \in S_j\}$ is contained in T.

We study in the next chapter[4] the exact scope of the substitution rule. It is possible to avoid this rule provided we take, as axioms, copies of the formulas of $\mathcal{X}$.

However using variables offers advantages.

1. Handling axioms is nicer than handling schemes. Likewise, given a formula p of $L(x)$, a proof of p is more readable than a proof of $p[u]$ ($u \in S_j$).

2. When we define new accesses, that are new functional schemes (Chapter 3 and 4), the substitution rule applies to terms containing occurrences of new symbols. Thus in the structure $\mathcal{N}$ of non negative integers, define the quotient of two numbers by the formulas:

$$x \not\equiv y \supset (x < y * (\mathbf{quotient}(x,y) + 1) \wedge (y * \mathbf{quotient}(x,y)) < x)$$

$$\vee\ y * \mathbf{quotient}(x,y) \equiv x$$

The axioms of addition, multiplication and predecessor apply to $\mathbf{quotient}(m,n)$ (which allows computing $\mathbf{quotient}(m,n)$ for each couple (m,n)).

1.2.7 Informations

Definition 6. *An information $\mathcal{F}$ is a saturated subset I containing the theorems of the formal system.*

I is therefore an information of the structure if

1. $\mathcal{X} \subseteq I$.

2. For all p and q of F, $p \in I$ and $p \supset q \in I$ imply $q \in I$.

[4] *Note of the translator:* Not translated yert.

3. For all p of F, for all variable x and for all u of the same type as x, $p \in I$ implies $p_x[u] \in I$.

Definition 7. *Let I be an information of $\mathcal{F}$. A set Y of formulas is a system of axioms of I if I is the set of theorems of the functional system $\mathcal{F}'$ of the proper axioms Y.*

Example 6. *1.* ***Lists:*** *we can deduce from the axioms of the* list *structure that for $k, \ell, k \neq \ell$*

$$\vdash s^k t \equiv s^\ell \supset s^k = \mathbf{nil}$$

In order to specify an information, we need to specify the length and the relations between the values of the list. For instance

$$\begin{cases} s^3 t \equiv \mathbf{nil} \\ s^2 t \not\equiv \mathbf{nil} \\ vt \equiv vs^2 t \\ vt \not\equiv vst \end{cases}$$

We get an information by taking the first three axioms or more generally a strict subset of the axioms. Actually the first information is complete, while the others are not. We make it precise in Chapter 2.

2. ***Lists with values in a given set:*** *In most of the cases the* list *structure is not studied for itself. A set of values is given, for instance, the non negative integers. This means that S_2 contains other elements beside $vt, vst, \ldots$. Thus the* list *structure with integer values is obtained by appending the structure* list *to the structure $\mathcal{N}$. A list with integer values is characterised by a system like:*

$$\begin{cases} s^n t \equiv \mathbf{nil} \\ vs^k t \equiv n_k \qquad (0 \leq k < n) \end{cases}$$

where the n_k's are terms of $\mathcal{N}$. Notice that we can find pathological situations, like the information with axioms

$$\begin{cases} vt \equiv S^3 0 \\ vst \equiv S^2 0 \\ vs^2 t \not\equiv S^n 0 \qquad n \geq 0 \\ s^2 t \not\equiv \mathbf{nil} \\ s^3 t \equiv \mathbf{nil} \end{cases}$$

It is complete despite $vs^2 t$ does not have "an integer value". This is inherent to the chosen formalisation: the integers are an infinite sequence and there exists no formula stating that $vs^2 t$ must have an integer value.

1.2.8 Interpretation of an information

Let I be an information, $R = (E, r)$ be a strict interpretation and $x = (x_1, ..., x_n) \in V_i$. There exists a function $\tilde{r}$ from the set of formulas built on $L(x)$ in the set of applications from E into $B = \{true, false\}$ such that:

$$\begin{aligned} \tilde{r}(u \equiv v) &= \textbf{if } \hat{r}(u) = \hat{r}(v) \textbf{ then } true \textbf{ else } false \\ \tilde{r}(\neg p) &= H_{\neg}(\tilde{r}(p)) \\ \tilde{r}(p \supset q) &= H_{\neg}(\tilde{r}(p), \tilde{r}(q)) \end{aligned}$$

Definition 8. *R is an **interpretation of** I if for all interpretation of I built on $L(x)$*

$$\tilde{r}(p) = \lambda p.true \tag{1.1}$$

Proposition 2. *R is an interpretation of I if and only if for each variable-less formula p of I*

$$\tilde{r}(p) = true.$$

If Y is s system of axioms of X it suffices for (1.1) to be true for all formulas of I.

Proof. The condition is clearly necessary.

1. On the other hand, let p be a formula of I on $L(x)$. For each $u \in S_j$, $p[u]$ is a theorem of I.

   $$\tilde{r}(p[u]) = true.$$

   But $\tilde{r}(p[u]) = \tilde{r}(p)(\tilde{r}(u))$.

   For all $j \in S_j$, $E_j = \hat{r}(S_j)$. Henceforth $E_i = \hat{r}(S_i)$

   hence $\tilde{r}(p) = \lambda x.true$.

2. If $p \in Prop(A)$, then p is a tautology. Hence $\tilde{r}(p) = \lambda x.true$. Likewise if $p \in Eg$.

   If q is deduced for p and $p \supset q$ by the modus ponens:

   $$\tilde{r}(p) = \lambda x.true \qquad \tilde{r}(p \supset q) = \lambda x.true$$

   Hence $\tilde{r}(q) = \lambda x.true$ by definition of $H_{\supset}$.

   If $p \in I$ and $p = q_y[u]$ then $\tilde{r}(p) = \tilde{r}(q)(\hat{r}(u)) = \lambda x.true$

Hence R is an interpretation of I whenever $\tilde{r}(p) = \lambda x.true$ for $p \in Y$. □

1.2.9 Information structures

If $\mathcal{F}$ is a functional system, we write $\mathcal{I}(\mathcal{F})$ the set of all the informations of $\mathcal{F}$ and, for all n, we write $\mathcal{A}_n(\mathcal{F})$ the set of the applications from $[\mathcal{I}(\mathcal{F})]^n$ into $\mathcal{I}(\mathcal{F})$. Let

$$\mathcal{A}(\mathcal{F}) = \bigcup_{n \geq 0} \mathcal{A}_n(\mathcal{F})$$

Definition 9. *An* ***information structure*** $\mathcal{S}$ *is given by a functional system* $\mathcal{F}$ *and a subset* $\mathcal{M}$ *of* $\mathcal{A}(\mathcal{F})$*.* $\mathcal{M}$ *is the set of* ***elementary modifications*** *of* $\mathcal{S}$*.*

We consider again the modifications of an information structure in Chapter 5.

Chapter 2

Mathematical Study of Functional Systems

This chapter is not translated yet.

Chapter 3

Extension of an Information Structure

This chapter is not translated yet.

Chapter 4

Recursive Equations in Information Structures

This chapter is not translated yet.

Chapter 5

Modifications, Computations, Problems

As we said in the introduction the aim of this work is to provide a meaning to the notions of problem and of program. This study has just started and we present in this chapter definitions that seem to be the most adequate.

5.1 Elementary modification

In order to define an information structure, it is not enough to define the objects and the accesses to these objects. It is also important to specify the modifications which allow going from one information to another. Recall the notations introduced in Section 1.2.9.

Notations: Let $\mathcal{F}$ be a formal system and $\mathcal{I}(\mathcal{F})$ the set of informations of $\mathcal{F}$. For all $n \geq 0$ we write $\mathcal{A}_n(\mathcal{F})$ the set of applications from $\mathcal{I}(\mathcal{F})^n$ to $\mathcal{I}(\mathcal{F})$. Let $\mathcal{A}(\mathcal{F}) = \bigcup_{n\geq 0} \mathcal{A}_n(\mathcal{F})$.

Definition 1. *An information system $\mathcal{S}$ is given by a functional system $\mathcal{F}$ and a subset $\mathcal{M}$ of $\mathcal{A}(\mathcal{F})$. The set $\mathcal{M}$ is the set of* ***elementary modifications****. of $\mathcal{S}$.*

We set

$$\mathcal{M}_n \quad = \quad \mathcal{M} \cap \mathcal{A}_n(\mathcal{F})$$

An elementary modification associates with sets $I_1, ..., I_n$ of theorems another set I' of theorems. A way especially interesting to realise this association is to "embed" $I_1, ..., I_n$ and I' in the same information structure and to define the modification by a system of axioms.

Let us have a look to an example: we want to add an element in a list after $s^k t$ and we want to set its value a. Intuitively this leads to replace $s^{k+1}t$ by $s^{k+2}t$ ... etc. In order to do that we introduce a new symbol s_1 and we consider the bijection $\sigma_1 : L \to L \cup \{s_1\} \setminus \{s\}$, such that $\sigma(s) = s_1$ and the other elements

of L are kept unchanged. There exists one and only one application, still written σ_1 defined on S' such that

- $\sigma_1(x) = x$ for $x \in V$
- $\sigma_1(gu_1...u_n) = \sigma_1(g)\sigma_1(u_1)...\sigma_1(u_n)$ for all $g \in L$ (with $\mathbf{source}(g) = (i_1, ..., i_n)$) and all sequences $(u_1, ..., u_n)$ of types $i_1, ..., i_n$. σ_1 can be naturally prolonged to formulas of $\mathcal{F}$.

Let us consider the functional system $\mathcal{F}''$, defined as follows:

- the alphabet of $\mathcal{F}''$ is $L'' = L \cup L_1$
- and the set of proper axioms is

$$X'' \;=\; X \cup \sigma_1(X) \cup Y$$

where $Y = \{sss_1^k t \equiv s^{k+1}t, \quad x \not\equiv s^{k+1}t \supset sx \equiv s_1 x, \quad vs^k t \equiv a\}$

Call $\mathcal{F}_1$ the system with alphabet L_1 and proper axioms $\sigma_1(X)$. Intuitively $\mathcal{F}_1$ represent the initial information while the terms of L represent the modified information. More precisely, given an information I of $\mathcal{F}$, one may define successively :

- the information I of I_1 transcribed by σ_1 and written $\sigma_1(I)$,
- the information I'', extension of I_1 to $\mathcal{F}''$, and written $\mathcal{E}(I)$, and
- the information I', restriction of I'' to $\mathcal{F}$, written $\mathcal{R}(I'')$.

I' is an information of $\mathcal{F}$ defined by

$$I' \;=\; \mathcal{R} \circ \mathcal{E}'' \circ \sigma_1(I) \;=\; m(I)$$

We say that the bijection m is associated with the bijection σ_1 and the system of axioms Y. In what follows, we write the function $m = adj(s^k t, a)$. Notice that adj is a modification scheme which associates with each "place" $s^k t$ and each "value" a, a modification $adj(s^k t, a)$.

More generally, here is a way to define a modification with n arguments. Assume the bijection σ_1, ..., σ_n from L to respectively L_1, ..., L_n satisfying the following properties:

1. $\forall x, y \in L \;\; \forall k \in [n] \;\; (\sigma_k(x) = y \Rightarrow x = y)$
2. $\forall x, y \in L \;\; \forall k, \ell \in [n] \;\; (\sigma_k(x) = \sigma_\ell(y) \Rightarrow x = y)$

Let $L'' \;=\; L \;\cup\; \bigcup_{i=1}^{n} L_k$. From the hypotheses 1. and 2., one can define a unique application **profile** on L'' such that

$\mathbf{profile}(\sigma_k(x)) = \mathbf{profile}(x)$ for all x of L and all $k \in [n]$.

For each $k \in [n]$, σ_k defines naturally a transcription, still written σ_k from the set of formulas on L' to the set of formulas

$$\begin{aligned} Y_3 \;\; = \;\; & \{t_1 \equiv \mathbf{nil} \supset (t \equiv t_2 \wedge sx \equiv s_2 x)\} \\ & \cup \{(t_1 \not\equiv \mathbf{nil} \wedge s_1 t_1 \equiv \mathbf{nil}) \supset (t \equiv t_1 \wedge st \equiv t_2 \wedge (x \not\equiv t \supset s_1 x \equiv s_2 x)\} \end{aligned}$$

Another example of modification: the conditional

Let $\mathcal{S}$ be any information structure with vocabulary L. For each variable-less formula p, let us introduce a modification $cond(p)$ with three arguments, such that for each I_1, I_2, I_3 (I_1 consistent)

$$cond(I_1, I_2, I_3) = \begin{cases} I_2 & \textbf{if} \quad p \in I_1 \\ I_3 & \textbf{if} \quad \neg p \in I_1 \\ T & \textbf{otherwise} \end{cases}$$

where T is the set of theorems of $\mathcal{S}$.

For $j = 1, 2, 3$, let σ_j be a bijection from a set L on sets L_1, L_2, L_3. Moreover, L, L_1, L_2, L_3 are assumed to be pairwise disjoint. Let us write $f_j = \sigma_j(f)$ for $f \in L$ and $j = 1, 2, 3$. For all $i = (i_1, ...i_q) \in [m]^*$, let x be $(x_{i_1}, ..., x_{i_q})$ a sequence of distinct variables of types $i_1, ...i_q$. Let $p_1 = \sigma_1(p)$. $cond(p)$ can be defined by the bijections σ_1, σ_2, σ_3 and by the set

$$\begin{aligned} Y = \{&p_1 \supset f x_{i_1} ... x_{i_q} \equiv f_2 x_{i_1} ... x_{i_q} \mid f \in L \wedge \textbf{source}(f) = i\} \\ \cup \{&\neg p_1 \supset f x_{i_1} ... x_{i_q} \equiv f_3 x_{i_1} ... x_{i_q} \mid f \in L \wedge \textbf{source}(f) = i\} \end{aligned}$$

Indeed one checks by induction that, for all term $u \in \mathcal{S}'$

$$\begin{aligned} p \in I_1 &\Rightarrow u \equiv \sigma_2(p) \in I'' \\ \neg p \in I_1 &\Rightarrow u \equiv \sigma_3(p) \in I'' \end{aligned}$$

Example 1. *Let $\mathcal{S}$ be the list structure and p be the formula $t \equiv$ **nil**, let us write Λ the information of $\mathcal{S}$ generated by p, in other words, the empty list. If I_1 is complete then I_1 contains p or I_1 contains $\neg p$. Hence we can write for I_1 assumed to be complete.*

$$cond(p)(I_1 I_2, I_3) = \textbf{if } I_1 \supset \Lambda \textbf{ then } I_2 \textbf{ else } I_3$$

Since Λ is complete, the conditions I_1 consistent and $I_1 \supset \Lambda$ imply $(I_1)_0 = \Lambda_0$ (where I_0 is the set of variable-less formulas of I).

Generally speaking if I is a **complete** information generated by a finite set of variable-less axioms $p_1, ..., p_n$, write $p = p_1 \wedge ... \wedge p_n$. Then if I_1 is complete:

$$\begin{aligned} &p \in I_1 \;\Leftrightarrow\; I \subseteq I_1 \;\Leftrightarrow\; (I_1)_0 = I_0 \\ &p \in I_1 \;\Leftrightarrow\; p \notin I_1 \end{aligned}$$

Hence $cond(p)(I_1, I_2, I_3) =$ **if** $(I_1)_0$ **else** I_2 **then** I_3. For a given information I_1 we write **if** $p \in I_1$ **then** I_2 **else** I_3 instead of $cond(p)(I_1, I_2, I_3)$.

5.2 Modification generated by a set of elementary modifications

We address modifications generated by $\mathcal{M}$, like we addressed accesses to an information structure.

Recall that **the set $\mathcal{I}(\mathcal{F})$ of informations of $\mathcal{F}$ ordered by inclusion is a completely inductive set** (Section 2.2.1). We can apply to modifications of $\mathcal{S}$ the fixed point theory presented in Section 4.2. For that we assume that all the elementary modifications of $\mathcal{S}$ are continuous applications on $\mathcal{I}(\mathcal{F})$. We check immediately that this is case when the modifications are defined axiomatically.

If $\mathcal{B}$ is a set of continuous applications of $\mathcal{I}(\mathcal{F})$ ($\mathcal{B} \subseteq \mathcal{A}(\mathcal{F})$) and if $\mathcal{T}$ is a functional generated by $\mathcal{B}$ (Section 4.5), we know that $\mathcal{T}$ is continuous and admits a fixed-point. We say that $\mathcal{B}$ **is stable by the fixed-point operator** if for each functional generated by $\mathcal{B}$, the components of the least fixed-point belong to $\mathcal{B}$.

$\mathcal{B}$ **is stable by composition** if for all $m \in \mathcal{B} \cap \mathcal{A}_n(\mathcal{F})$ all $m_1, ..., m_n \in \mathcal{B} \cap \mathcal{A}_k(\mathcal{F})$, then $m(m_1, ...m_n) \in \mathcal{B}$, where

$$m(m_1, ...m_n)(I_1, ..., I_k) = m(m_1(I_1, ..., I_k), ..., m_n(I_1, ..., I_k))$$

For $n > 0$, $k \in [n]$, one write π_k^n the modification $\lambda I_1 ... I_n . I_k$. Let $\mathcal{P}$ be the set of those **projections**.

Definition 3. [1] *The set $\widehat{\mathcal{M}}$ of modifications of $\mathcal{S}$ is the least set of $\mathcal{A}(\mathcal{F})$ containing $\mathcal{M} \cap \mathcal{P}$ and stable by composition and by the fixed-point operator.*

Example 2. *Let us consider in the* list *structure the elementary modifications* ***head***, ***tail***, ***add***, **cond**$(t \equiv \textbf{nil})$. *Then the concat operation on the* list *structure, defined as follows, is a modification of the* list *structure:*

$$concat(I_1, I_2) \;=\; \textbf{if } t \equiv \textbf{nil then } I_2 \textbf{ else } \textbf{\textit{add}}(\textbf{\textit{head}}(I_1), concat(\textsf{Tail}(I_1), I_2).$$

The definition of $\widehat{M}$ leads naturally to the following type of induction.

Let P be a property on $\widehat{M}$ such that P is satisfied on $\mathcal{M} \cup \mathcal{P}$ and stable by composition and by the fixed-point operator. Then $P(m)$ is satisfied for all m in $\mathcal{M}$. One shows by induction the following result.

Proposition 1. *m belongs to $\mathcal{M}$ if m is the first component of a least fixed point of a functional generated by $\mathcal{M}$.*

We can therefore take the following definition

Definition 4. *A* ***program*** *on an information structure $\mathcal{S}$ is a fixed-point system π on $\mathcal{M}$:*

$$\pi : (m_1, ..., m_n) = \mathcal{T}(m_1, ..., m_n)$$

The modification m_π, induced by π, is the first component of the least fixed-point of $\mathcal{T}$.

[1] *Note of the translator:* Definition 2 does not exist in the manuscript.

Computations in an information structure Let us first consider the case where all the elementary modifications are functions from $\mathcal{I}(\mathcal{F})$ to $\mathcal{I}(\mathcal{F})$ (monadic case). We call computation on $\mathcal{S}$ each sequence of elementary modifications. Therefore a computation is an element of $\mathcal{M}^*$.

Example 3. *We can consider on the* list *structure the computation*

$$(\mathsf{tail}, \mathsf{tail}, \mathsf{head})$$

With this computation is associated the modification $\mathsf{head} \circ \mathsf{tail} \circ \mathsf{tail}$ *which associates with each list I the list made of the third element of the list I, if this element exists and the empty list otherwise. More generally if $c = m_1...m_n$ is a computation of $\mathcal{S}$, one associates with c the modification $m_n \circ ... \circ m_1$. If $n = 0$, m is the identity of $\mathcal{I}(\mathcal{F})$ into itself (in other words the function π_1^1).*

To associate a computation with arbitrary modifications, let us notice that we can associate with a computation $c = m_1...m_n$ the functional scheme $c' = m_n...m_1X$ on $\mathcal{M} \cup \{X\}$ where X is a variable. In what follows we make no difference between c and c'.

In the general case let $\mathcal{V} = \{X_1, X_2, ...X_n, ...\}$ an infinite set of variables. Each variable is a 0-ary symbol and each modification of $\mathcal{M}$ can be seen as a n-ary symbol. We can state the following definition:

Definition 5. *A computation of $\mathcal{S}$ is a functional scheme on $\mathcal{M} \cup \mathcal{V}$.*

Example 4. $\mathbf{add}.\mathbf{head}.X_1.\mathbf{tail}.X_1$ *is the computation associated with the modification* $\lambda I.\mathbf{add}(\mathbf{head}(I), \mathbf{tail}(I))$.

Let c be a computation on $\mathcal{M} \cup X$, the modification associated with c is defined recursively as follows

1. For $k = 1, ..., n$, π_k^n is associated with X_k.
2. If $m \in \mathcal{M}_k$ and if, for $(j = 1, ..., k)$, m_j is associated with c_j, then $m(m_1, ..., m_k)$ is associated with to the computation $m_1, ..., m_k$.

Such a modification is obtained by composition of elementary modifications and projections. We do not study in the present work the computation which should be associated with a modification defined recursively and hence with a program. We have to embed the set of computations in a completely inductive set. The generalised computation associated with any program $m = \mathcal{T}(m)$ should be the least fixed-point of a functional $\widetilde{\mathcal{T}}$ deduced from $\mathcal{T}$. Works have been done in this direction [Scott], [Nivat], [Finance].

Before giving examples of programs, let us introduce the notion of problem. Indeed, a program (or an algorithm) computes the solution, if it exists, of a given problem.

5.3 Formalisation of the notion of problem

5.3.1 Introduction

In computer science, a problem is expressed as a set of relations between compound objects. We can distinguish among objects:

- *Constant objects* whose value is determined by the statement of the problem.
- Objects whose values are let free by the statement of the problem. They have to satisfy conditions. These objects are the *data* of the problem.
- Object whose value depends on the values of the data. Some of those objects are simply used for intermediary computations. The others are the *results* of the problem. Clearly, the data and the results are part of the problem.

Let us give a first example, whose data are a set of equalities.

Example 5. *Assume that we have to compute the monthly salary of a worker with a health plan, knowing the number of work hours and the hourly wadge. The problem can be defined by:*

$$\textit{total} = \textit{gross salary - salary deduction}$$

$$\textit{salary deduction} = \textbf{if } \textit{gross salary} > \textit{threshold} \textbf{ then } \textit{ss1 +ss2} \textbf{ else } \textit{gross salary} \times \textit{0.065}$$

$$\textit{threshold} = \textit{2320}$$

$$\textit{ss1} = \textit{gross salary} \times \textit{0.01}$$

$$\textit{ss2} = \textit{threshold} \times \textit{0.02}$$

$$\textit{gross salary} = \#\ \textit{hours} \times \textit{hourly wadge}$$

In this example, data are "# hours" and "hourly wadge". It is possible to change the problem by removing the equation "threshold = 2320" and by considering "threshold" as a data, which changes each year. The result can be "total", but it can also be {total, salary deduction}.

Example 6 (Definition of the *gcd* of two integers)**.** *Given two integers a and b, the gcd of a and b can be defined by:*

$$\textit{gcd } \textbf{\textit{divides}}\ a \ \wedge\ \textit{gcd } \textbf{\textit{divides}}\ b \ \wedge\ [x \textbf{ \textit{divides}}\ a \wedge x \textbf{ \textit{divides}}\ b] \supset x \textbf{ \textit{divides}}\ \textit{gcd}$$

In this problem, the statement is the above formula, the data are a and b and the result is gcd.

The interest (and the difficulty) of this kind of problem is that we cannot immediately deduce an algorithm for computing the gcd. This search of an algorithm computing the solution of a problem is sometime called *program synthesis* [Manna-Waldinger]. It is well known that, in general, program synthesis cannot be made fully automatic. Research groups address methodologies for program construction and in parallel develop proof techniques to guarantee correctness of the found program.

Let us give a program computing the *gcd*. It is just a translation in terms of modifications of Euclid's algorithm.

$$gcd(a,b) \;=\; \textbf{if } b = 0 \textbf{ then } a \textbf{ else } gcd(b, r(a,b))$$

where $r(a,b)$ is the remainder of the division of a by b.

For this, let us introduce assignments. they are is modification schemes.

If $a_1, ..., a_n$ are identifiers, that are constant symbols, and $u_1, ..., u_n$ terms of same types, the modification $assign(a_1, ..., a_n; u_1, ..., u_n)$ associates with each information I the information I' containing the theorems: $a_i \equiv u_i$ for $1 \leq i \leq n$. Precisely, $assign(a_1, ..., a_n; u_1, ..., u_n)$ can be defined by the couple (σ, Y). where σ is a bijection associating with each a_i a symbol a'_i and fixing the other symbols and where Y is the set of formulas $a_i \equiv \sigma(u_i)$ for $1 \leq i \leq n$.

Taking into account this definition, Euclid's algorithm can be translated into a program:

$$m(I) = \;\textbf{if } b \equiv 0 \in I \textbf{ then } assign(gcd; a) \textbf{ else } m \circ assign(a, b; b, r(a,b))(I)$$

5.4 Definitions

Definition 6. ***A problem*** $\mathfrak{P}$ ***on a structure*** $\mathcal{S}$ *is given by a set* $L(\mathfrak{P})$ *of new accesses and a set* $X(\mathfrak{P})$ *of axioms, i.e., formulas on the alphabet* $L \cup L(\mathfrak{P})$. $L(\mathfrak{P})$ *and* $X(\mathfrak{P})$ *are respectively the* ***set of unknowns*** *and the* ***statement*** *of the problem.*

Example 7. *We studied in Chapter 4, recursive problems. In this kind of problem* $L(\mathfrak{P}) = \{f_1, ..., f_n\}$ *and* $X(\mathfrak{P})$ *is a set of recursive equations.*

The statement of a problem can be more complex, e.g., the *gcd* example.

We call data of $\mathfrak{P}$ any information of $\mathcal{S}$. $L(\mathfrak{P})$ and $X(\mathfrak{P})$ create an extension of $\mathcal{S}$ as a structure $\mathcal{S}'$ with alphabet $L \cup L(\mathfrak{P})$ and with set of proper axioms $X \cup X(\mathfrak{P})$. We call **result** of $\mathfrak{P}$ associated with an information I of $\mathcal{S}$ a complete information I'_0 containing $\mathcal{E}(I_0)$ (where I_0 is the restriction of I to $\mathcal{F}_0$ (Section 2.1)).

We say that a problem $\mathfrak{P}$ is **deterministic** if for each complete information I, $\mathcal{E}(I)$ is complete as well, hence if $\mathfrak{P}$ associates with each complete data one and only one result.

Chapter 6

Conclusion

This work can followed by others in several directions.

1. Until now we have formalised only simple problems, like integers and lists. It should be possible to address more complex fields, like data bases or operating systems. A first work (Jean-Pierre Finance) uses information structures to formalise the semantics of a programming language. Claude Pair studies the most frequent information structures, but he does not use functional systems. In program correctness proofs, the formalisation presented here takes into account the axiomatics of the domain.

2. In order to address more complex structures, it is necessary to take into account algebraic operations. Thus we could be able to reduce the study of lists of lists or of tree-like structures to these of lists.

3. In Chapter 3, we build several interpretations of the mode construction in ALGOL 68. We should be able to generalise those constructions and to prove the consistence of non trivial structures.

4. The concepts of computation, of program and of problem have only been superficially addressed. The first job should be to address the computations associated with a recursive program or simply with an iterative one. A first step in this direction has been done by Jean-Pierre Finance in the case of one variable modifications. Similar research has been done out of the framework of information structures, by Scott and Nivat. Ideas they developed can surely be reused as a starting point.

5. For problems, we try to write their statements (for example for the *gcd*) out of any algorithm. Is this always possible? If not, how should the definition be modified? For instance, until now, we did not get a pleasant statement for the merging of two sorted lists.