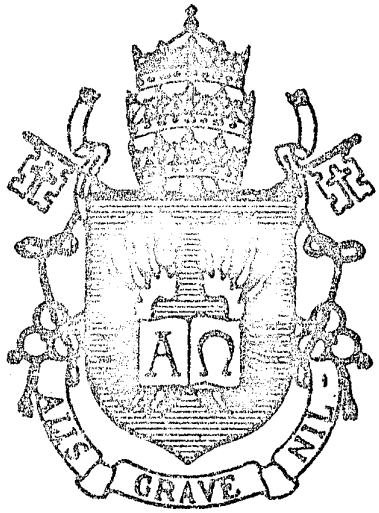


PUC

Series: Monografias em Ciência da Computação
Nº18/88

INTUITIONISTIC TYPE THEORY AND GENERAL PROBLEM THEORY: A RELATIONSHIP

Edward Hermann Haeusler

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

JA MARQUÊS DE SÃO VICENTE, 225 – CEP 22453

RIO DE JANEIRO – BRASIL

PUC/RJ - DEPARTAMENTO DE INFORMÁTICA

Series : Monografias em ciência da computação, 18/88

Editor : Paulo A. S. Veloso Dezembro, 1988

INTUITIONISTIC TYPE THEORY AND GENERAL PROBLEM THEORY : A RELATIONSHIP*

by

Edward Hermann Haeusler

* This work was partially supported by CAPES.

In charge of publications:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC/RJ - Depto. de Informática
Rua Marquês de São Vecente, 225 - Gávea
22453 - Rio de Janeiro, RJ
Brasil

ABSTRACT

Constructive reasoning may be used in some diferent fields, such as Set Theory, Logic and Problem Theory. Martin-Lof's intuitionistic type theory (ITT) [Martin-Lof84] is a way to express constructive reasoning in these fields. Veloso's general problem theory (GPT) [Veloso84] is an approach to explain the concept of problem in the context of problem solving (it is restricted to formalized problems).

In this work we explore a relationship between GPT and the problematic interpretation of ITT, searching common features. We conclude this work by giving an approach to program development based on this relationship.

Key-Words : Intuitionism, Problem, Logic, Types, Program Construction

RESUMO

O raciocínio construtivo pode ser aplicado em diferentes áreas do pensamento humano, tais como Teoria dos Conjuntos, Teoria de Problemas e Lógica. A Teoria Intuicionista dos Tipos (ITT) de Martin-Lof [Martin-Lof84] se configura em uma forma de expressar o raciocínio construtivo nas áreas citadas. A Teoria Geral dos Problemas (GPT) de Veloso [Veloso84] clarifica o conceito de problema no contexto de resolução de problemas como uma certa estrutura matemática.

Neste trabalho, exploramos um relacionamento entre GPT e uma interpretação "problemática" de ITT, buscando características comuns. Concluímos o trabalho apresentando uma proposta de metodologia de programação baseada no relacionamento estudado. Dois exemplos de aplicação são mostrados.

Palavras-Chave : Intuicionismo, Problema, Lógica, Tipos, Construção de Programas.

CONTENTS

I. Introduction

I.1 - Intuicionistic Type Theory : An overview

The main idea of ITT is to give a way to express constructive reasoning. This is achieved by writing a kind of calculus and giving some possible interpretations to it, but not needly semantics. The part concerned with the calculus is built from some atomic elements that we call propositions. So the other syntatic elements are combinations of propositions with some operators or contructors. Thus the interpretations are given by interpreting the atomic elements and interpreting the very ways to combine them with the use of such constructors. Martin-Lof calls these interpretations as judgments. He points out some of them, which it will be briefly shown here.

The basic propositions are "A set" and "a $\in$ A" and their different judgements are shown below :

A set	a $\in$ A	obs
A is a set	a is an element of A	A $\neq \phi$
A is a proposition	a is a proof of A	A truth
A is an intention (expectation)	a is a method of realizing A	A is realiz.
A is a problem	a is a method of solving A	A is solvable

The first judgement is the set theoretical, the second is the logical. The third is due to Heyting [Heyting31]. The fourth is due to Kolmogorov [Kolmogorov32], which is our goal. Briefly, we can say that the Kolmogorov's meaning of problem is " something to do " (task) and its solution is " how to do ". Kolmogorov called programs such solutions.

This last judgement will be explored in this work by relating ITT with GPT. In other words we will see the ITT as a language to problem description.

Following the constructive approach, Martin-Lof understands

that to any of the judgements there is not universes a priori, i.e, the universes must be built. This approach had begun before with Brouwer. So, we must not believe in constructed worlds, rather better, sets or categories. So the objects of the constructivism are all constantly being constructed, rather better they have a kind of potential existence. So, we can realize a set as a kind of procedure to generate its elements. This is naturally regard to the the set-theoretical way of judgement. Another thing that we must assure is the equality of elements. So we must indicate how construct equal elements. In the logical interpretation this approach means that a proposition is true when we say what we can take into account as being a proof to it, and when two proofs are proofs of the same proposition.

To implement the notions explained above, Martin-Lof uses the notion of canonical element, or canonical proof. So the normal form theorem underlies the above concept. Thus the formation rules of the set are given related to the canonical elements. The formation rules includes the rules to the generation of equal canonical elements. So *an element of a set (in general) is a method to yield canonical elements.*

All the sets (types) of the predicative intuitionistic type theory are built from already constructed sets, i.e., from sets which have already their procedures of generation. These procedures can be well defined rules of formation. So, there are some primitive sets. These sets do not need other sets for its construction, what is made by using some well defined constructors determined by the formation rules. So, they are the finite sets and the natural numbers set. So, in order to produce higher types we use the constructors which are :

1- The cartesian product (Π),

2- The disjoint union or sum of a family sets (Σ),

3- The disjoint union of two sets (+),

4- Propositional equality (I),

5- Lists of elements of a set (List),

6- Wellorderings (W).

But, we can see that we can only use these constructors a finite number of times to construct higher types from the

primitive types (sets). So we have only a finite type theory.

To strenghten the theory, Martin-Lof introduced universes in order to add transfinite types to the language. The main idea of the universes is to describe the least set closed under certain specified set forming operations. This is made in this way because intuitionistically we cannot realize the set of all sets, once time we cannot exihibit once and for all, all possible set forming formations. But the concept of universes permit us construct set of sets, what is enough. Martin-Lof gives two ways to the defining universes, a la Tarski and a la Russel. We prefer the tarskian one. We refer [Martin-Lof84] in order to get more detail.

Since we are interested in the "problematic" judgement and the purpose of this work is not to present ITT. We will present ITT already trying the relationship with GPT, in section III.

I.2 - General Problem Theory : An overview

In this section we briefly describe GPT. We can say that GPT was made in order to become precise the notion of problem. This is made by denoting problems and its solutions as mathematical structures. In this way GPT can be used to study some problem solving problems, like the reductibility of a problem.

Following Polya, Veloso takes the questions : "What are the data?", "What are the possible results?" and "What something must have in order to be a satisfatory solution?" as a base to his mathematical definition of problem.

The aim of the specification of the data domain is the limitation of the universe of the discurse, i.e., whether the problem is about objects of one kind or other kind. It is clear that this element is important in the specification of a problem, since it changes the nature of the problem. For example the problem of finding the next number in a well-ordered set has very different solutions for the case when the data domain is the natural numbers or when it is the real numbers (taking into account the choice axiom).

In the same way, the domain of the results contributes to the

specification of a problem. For, it shows where we can find the results. For example the problem of finding square roots for real numbers is different whether we want real numbers or complex numbers as results.

Finally, in order to become precise the specification, it is necessary to say how the data and the results are related. This is the same that saying what we can take into account as a solution to the problem. It is obvious that for the same pair data-results we can specify a very large number of different problems.

Thus, Veloso defines a problem P as the following mathematical structure :

$P = \langle D, R, q \rangle$

, where :

D is the data domain (tipically a set)

R is the domain of results (a set too)

$q \subseteq D \times R$ is the relationship between data and results that determinates the problem, rather better, its solutions.

Veloso calls a problem defined in the above way an unrrestricted problem.

But, what is a solution to a problem ? The answer could be : A way to associate to each member of D a member of R. In other words, we can say that a solution is a function f such that $f \subseteq q$. The reason to restrict the solutions to functions is that in this way we get a little more the computational feature, besides the accepting of solutions as relations becomes the formers closer to the specification of the problem than one could want.

Based on the definition given above, Veloso gives some definitions and shows a result that gives a relationship between them.

Def A problem P is solvable iff it has at least one solution.

Def A problem $P = \langle D, R, q \rangle$ is viable iff for each $d \in D$ there is at least one $r \in R$ such that $(d,r) \in q$.

Def The space of solution to a problem P ($\Sigma(P)$) is the set of all functions $f: D \longrightarrow R$ such that f is a solution to P.

Prop : Let $P = \langle D, R, q \rangle$ be an unrestricted problem. Then the following are equivalent :

1. P is solvable.

2. P is viable

3. The following sentence holds

$$(\forall d : D)(\exists r : R)q(d,r)$$

We can note that $\Sigma(P)$ is equal to the set of all Skolem functions for the sentence above.

We know that D and R may be non-denumerable. Thus in general the above utterances are equivalent to the choice axiom. But, as we are interested in the computational features, we will restricted ourselves to denumerable domains, so we do not need use the choice axiom. However as we will see, in ITT the choice axiom is provable. So we can formalize the above result in ITT.

By using this definition, Veloso gives a more precise definition of two main tools of problem solving, which are the reduction of a problem to other and the decomposition of problems. We will briefly give these concepts, which we will use to show their correspondence with ITT components.

- Reduction of problems

We can say that the main idea of the reduction of problems is to search a known problem and by a change of language to associate data and results of our problem to data and results of this known problem. But this must be done such that a solution to the known problem must give us a solution to our problem. This can be writen in mathematical language as following :

Def Let $P = < D, R, q >$ and $P' = < D', R', q' >$ be unrestricted problems. We say that P is reductible to P', iff there is functions τ and ρ such that :

1. $\tau : D \longrightarrow D'$ (called translation function)

2. $\rho : R' \longrightarrow R$ (called recovering function)

3. For each solution $\sigma' : D' \longrightarrow R'$ of P', the composition $\rho.\sigma'.\tau$ is a solution to P.

We call the pair $<\tau,\rho>$ as ELO and if condition 3 above holds for an ELO, we call this ELO as reduction.

- Decomposition of problems

The main idea of the decomposition is the splitting of the goal problem into smaller problems, so easier. Then we have to solve these small problems and finally to join these solutions in order to have a solution to the goal problem. Each of these

smaller problems can themselves be divided and the process can be iterated until we get problems easy enough to not need smaller ones to help us. So we have to combine the solutions obtained at each level until the highest, that is the level of the goal problem. It is interesting to note that the process of decomposition happens at the same time of the recombination, i.e., one only splits a problem in some other iff he knows how to use their solutions in order to solve the goal problem, or, in other words, iff he knows how to recombine the solutions to form a solution to the bigger. So we can say that the concept of decomposition of problem must include the recombination of the solutions. In that way, Veloso defines an n-ary decomposer, i.e., a decomposer that splits the problem in n minor problems, rather better, in n minor instances of problems.

An n-ary decomposer Δ_n for a problem $P = \langle D, R, q \rangle$ is a quadruple $\langle \Pi, \mu, \nu, \eta \rangle$ where :

1- Π is a finite family of functions $\pi_i : D \longrightarrow D$ $(i = 1,n)$ called splitting functions;

2- μ: $D \times R^n \longrightarrow R$ is called recombination function;

3- ν: $D \longrightarrow R$ is called function of immediate correspondence;

4- $\eta \subseteq D$ is called the simplicity test.

The main rôle in a decomposer is the family Π which splits the instances of the problem P in n subdomains, i.e., in n classes of instances of the problem. μ recombines the results into the result of the problem, but as in general we need the original datum to do the recombination, μ has elements of d as argument. The function ν associates simplest data tested by the predicate η (we can see it so, if we want) to their respective results. A good example of decomposition is one of recursivity over two independent data: To solve an instance of a problem recursively we try to find the simplest cases of solution to each partition of data (basic cases) and find a way to recombine these solutions into one more complex and so on until the solution of the goal instance. It is clear that we have also a concept of reduction within each partition of data. But could be need the use of the original data in the recombination, as we have already mentioned above.

Well, only the decomposer do not help us. We need write down a condition saying when a decomposer is sound to solve a problem. This is expressed by the following function σ :

$$\sigma(d) = \begin{cases} \nu(d) & \text{if } d \in \eta \\ \mu(d,\ \sigma(\pi_1(d)),\ldots\ldots,\ \sigma(\pi_n(d))) & \end{cases}$$

which must define a solution to P.

One can note that we need some comparation criterion to say when an instance is smaller than other. Well, this can be expressed by a well-founded relation in D w.r.t. the simplest instances ($d \in \eta$). Veloso shows that if a decomposer Δ_n over a problem $P = \langle D, R, q\rangle$, where D is well-founded, is such that <u>if</u> all mininum element of D is in the domain of q and whenever d is not a minimum element (easiest element) there is $(d_1,r_1),\ldots,(d_n,r_n)$ and $(d,r) \in q$ such that $d_i < d$; <u>then</u> P has an n-ary decomposer [Veloso84].

All what we have said above is about uniform deomposition, i.e., when we can decompose always with n-ary decomposer (n constant). It 's normal finding problems to which we once time need a division and other time may be other division. So, to solve this Veloso defines the problem P^+ which express the same intension of P, but with respect to finite lists of elements of D (D^+) and their respectives R^+. Thus a decomposition for any arity is now interpreted as a unary decomposition of P^+, and it is shown that a problem P has an n-ary decomposition iff P^+ has a decomposition.

We will speak about operations on problems in the sequel, when we show the relationship between ITT and GPT. We can look forward saying that this concept is in some sense essential in the relationship. This can be viewed since now by noting that the fundamental idea of the intuitionism is the building of objects. Well a intuitionistic theory of problems must have the same fundaments. So we need the notions of building problems. This is given by a intuitionistic interpretation of the operations on problems. This will be done in our work by finding associations between the ITT constructors and the operations on problems in

GPT. But, it is clear that we need a intuitionistic view of " *what is a problem ?* ". This is discussed in the first subsection of next section.

II. ITT x GPT

This section is the main section of our text and its goal is to show the relationship associating operations and constructors as mentioned at ending of subection I.2 . Well, firstly, in section II.1 we discuss the concept of problem, after we show the relationship based on the concept developed in section II.1.

II.1 - The intuitionistic concept of problem

As we have already shown, Martin-Lof [Martin-Lof84] gives four kinds of judgements for the proposition a ∈ A. But, he basically develops ITT with respect to two of them, that are the logicistic and the set-theoretical. Here we develop the "problematic" judgement with the same constructive appeal. We would like to remind that this way of judgement was proposed by Martin-Lof himself.

What is a problem under the constructive point of view? Well, let's see the constructive view of set, that is, a set is a kind of process to construct their elements. This is in a certain way equivalent to say that a set is given when we know what are their elements, rather better, what can be taken into account as an element of it (as do Martin-Lof). Thus we define a problem by saying what can be taken into account as a solution to it. So the proposition a ∈ A is correctly interpreted as the problem A is solvable. So the inviable problems [Veloso84] are equals among them, since that nothing can be taken into account as a solution to them. In this way, it seems need a kind of representative solution for a problem, that is a natural solution to the problem. That solution must mirror the structure of the problem. We call it as canonical solution. So if the problem is a case alternative then its canonical solution must be an alternative of solutions, where each of them is a solution to the respective case in the

problem. A problem that is a pair of independent problems must have as canonical solution a pair of solutions to the respective problem components.

Well, may happen, as we see daily, that a solution to a problem does not mirror the structure of the problem. But, we can argue that if the problem has solution then it has a canonical solution. So, non-canonical solutions can be effectively transformed into canonical solutions. The first utterance is the equivalent to the normal form theorem in proof theory, while the last is equivalent to the normalization theorem. Thus, we will define the concept of solution in general to a problem as a solution that can be effectively transformed into a canonical solution to it. As we seen this is very strong, but we have good reasons to want it :

1- We want to get the concept of problem based on its solutions,

2- So, we must take into account all solutions,

3- Because of our constructive definition of problem we have only ability to verify canonical solutions.

4- We need verify our solutions

5- So we need the normal form theorem and more,

6- We need an effective way to transform solutions into their respective canonical solutions, for we want a computational way to find out whether an object is or not a solution to a problem, even it is not a canonical problem.

Besides, as in the set-theoretical judgement, we need the concept of equality. So we have that a problem is given when *we give how are their solutions (that is distinct of what is their solution) and when two solutions to the same problem are equals*. We also say that any solution to a problem defined in this way is a program. So we have the concept of metaprogram as being that effective process responsable by the transformation of any solution into their associated canonical solution. We can note that some program methodologies are based just on these metaprograms.

II.1.1 - Problems and canonical solutions

In this subsection we show the relationship between ITT and GPT. This is based on the correspondence between the types of ITT and certain operations or even kinds of problems described in GPT. This is done looking to the canonical solutions, since they define the problem. So, it is out of the scope of this section any consideration about the corresponding metaprogram. So in the following the proposition $a \in A$ means that a is a (general) solution that can be reduced to a canonical solution a' to A.

We go on by showing the rules in ITT that permit us to build problems from already built problems, so permiting us to define their canonical solutions and the equality among them.

In the following we will denote that A is a problem as "A Prob".

According our definition two problems are equal iff they have the same solutions. This is shown by the below rule :

$$\frac{a \in A}{a \in B}$$

Trivial facts aboout problems are that any problem is equal to itself and any solution to a given problem is equal to itself. Thus we have the following rules :

Reflexivity

$$\frac{a \in A}{a = a \in A} \qquad \frac{A \text{ Prob}}{A = A}$$

Besides, we have :

Simetry

$$\frac{a = b \in A}{b = a \in A} \qquad \frac{A = B}{B = A}$$

And :

Transitivity

$$\frac{a = b \in A \quad b = c \in A}{a = c \in A} \qquad \frac{A = B \quad B = C}{A = C}$$

This rules are explicitily about canonical solutions, but we

know that because of the metaprogram concept they are implicitily about general solutions.

II.1.3 - Hypothetical judgements, substitution rules, program sucessing and serial execution.

A very important concept in Martin-Lof is that of hypothetical judgement. That is, a judgement done based on a priori established judgements. We restrict ourselves to the case of one only hypothesis. The general case is an immediate consequence.

A well-known concept in set theory is that of indexed families of sets. That is, we contruct a family A(i) based on another set, the set of indexes. Seeing this throught a constructive view, the building of an indexed family is a process that assumes a previously built set, rather botter, assumes the existence of its building process. This is undoubtly a hypothetical reasoning. Indeed, all ways in ITT of hypothetical reasoning are represented by the above mentioned. This is a consequence of the fact that we can interpret the family concept just as a proof by using hypothesis (this is done in the logical interpretation of ITT).

Well, let's see what is the meaning of an indexed problem. It is clear that the index is also a problem. Indeed, the indexes are solutions to the problem index. Let's B(x) Prob (x ∈ A) be the proposition that represents this indexing. The solutions of B(x) depend on the solutions of A. This tell us, in another way, that to solve B we need to solve A, for any solution b(x) of B(x) is a program that receives as entry a solution of A. This clearly is very related tc the concept of serial execution of programs, which is related to the concept of sequencing of problems. So, we can associate to this way of hypothetical reasoning the composition of problems, and to its solutions the sequencing of programs.

How are the solutions to a problem B(x) Prob (x ∈ A) ? We know that any solution must take into account solutions to A. But, at the same time, they cannot include properly any solution to A. Thus, as we have briefly shown, they are schemes of solutions to

B(x) that must use solutions to A, which we call b(x). Indeed, these schemes represents a class of solutions rather than one solution. Which problem b(x) solves ? The answer is : none. It only says how to solve any instance of B(x). When we get a solution a to A (a ∈ A) we can instanciate B(x) as B(a). Then b(a) is a solution to B(a), so B(a) **Prob** and, once time that to verify this we instantiated the hypothesis as a ∈ A, the conclusion does not depend on it but rather on a ∈ A. It is equally clear that equal solutions to A must result in equal instantiated problems. The below rules summarize the explained. That rules can be called substitution rules.

$$\frac{a \in A \qquad \begin{array}{c}[x \in A]\\ B(x)\ \mathbf{Prob}\end{array}}{B(a)\ \mathbf{Prob}} \qquad\qquad \frac{a = c \in A \qquad \begin{array}{c}[x \in A]\\ B(x)\ \mathbf{Prob}\end{array}}{B(a) = B(c)}$$

, where

$$\begin{array}{c}x \in A\\ B(x)\ \mathbf{Prob}\end{array}$$

denotes that B(x) is a problem assuming that x is a solution to A. And the square brackets around the assumed facts says that the conclusion of the rule does not depend more on them, like Natural Deduction [Prawitz65].

We also want to express the fact that two problems depend on the same problems. Well, this judgement is denoted by the proposition B(x) = D(x) (x ∈ A). And it is clear that in this way the problems must have the same particularisations. This is said by the below substitution rule. This agree with our way of judgement, since according to the concept of program, for if the execution of two equal programs is preceded by a same program the results must be the same.

$$\frac{a \in A \qquad \begin{array}{c}[x \in A]\\ B(x) = D(x)\end{array}}{B(a) = D(a)}$$

And using the above rule we can deduce in ITT the following derived rule.

$$\frac{a = c \in A \qquad B(x) = D(x)}{B(a) = D(c)}\ [x \in A]$$

, which has as deduction :

$$\frac{\dfrac{a = c \in A \qquad B(x)\ \mathrm{Prob}}{B(a) = B(c)} \qquad \dfrac{c \in A \qquad B(x) = D(x)}{B(c) = D(c)}\ [x \in A]}{B(a) = D(c)}$$

, where the premiss is weaker than $a = c \in A$ as well $B(x)$ Prob $(x \in A)$ is weaker than $B(x) = D(x)$ $(x \in A)$.

We can note that this interpretation of ITT give us a kind of problem calculus, informally at least.

As we saw previously, a solution to the problem $B(x)$ Prob $(x \in A)$ is something that depends on of any solution of A. So, according to ITT we use the notation $b(x)$ to denote it. It is clear that $b(a)$ must be a solution to $B(a)$, i.e, to give a solution to a particularisation of a indexed problem is the same that to particularize the solution to the indexed problem. Also, equal particularisations must yield equal solutions. These two ideas are respectively explicitated by the following substitution rules.

$$\frac{a \in A \qquad b(x) \in B(x)}{b(a) \in B(a)}\ [x \in A] \qquad \frac{a = c \qquad b(x) \in B(x)}{b(a) = b(c) \in B(a)}\ [x \in A]$$

Well, as a trivial consequence of the concept of indexed problem and its solution, we have that if two solutions to an indexed problem are equals then they are equals regard to any particularisation of the problems. This is said by the following substitution rule.

$$\frac{a \in A \qquad b(x) = d(x) \in B(x)}{b(a) = d(a) \in B(a)}\ [x \in A]$$

With the above explained ways of hypothetical judgement Martin-Lof constructs all types of his theory. We note that the judgements produce the substitution rules at the calculus level,

and these rules are rules of particularisations of indexed problems, which mirror the idea of dependence between problems. We instantly give the sequencial meaning to these kinds of problems, seeking the intuition in their solutions, which can represent the idea of serial execution. This last idea is not so rigorous as we could think, once time that we can think in indexing a problem without properly use this indexing and so the serial execution does not have more a necessary execution sequence.

II.2 - Type problems and their relation to GPT

In this section we give the problem interpretation to each type in ITT. Since ITT is a constructive theory, it builds types from previously constructed, or at least already defined. So we does not need worry about the how are the solutions of the types used in the building of a new type.

Since a problem is defined whenever we say how are its solutions and when they are equals, our explanations will be as following (as in [Martin-Lof84]) :

1- By giving formation rules, which tell us the conditions need to construct a new type. These conditions are in general informations saying that such and such constituent object must be of such and such type.

2- By giving introduction rules, which tell us how are the canonical solutions of the introduced type problem.

3- By giving elimination rules, which tell us how retrieve informations about any of the constituent types from the new type informations. These informations are according with our definition of problem, about the canonical solutions.

4- By giving equality rules, which tell us when two canonical solutions of the new type are equals. These rules generally include either implicitly or explicitly the information about when two canonical solutions to each of the constituent types are equals.

We can note that this structured way of construction is very related to the Natural Deduction way of model the mathematical reasoning, which are not allways constructive.

II.2.1 - cartesian product and the problem sequencing

In our discussion about the hypothetical judgements in the context of problems, we related the indexed problem to the very computational notion of serial execution. But we have not spoken anything about the type of that kind of problem. In ITT, Martin-Lof formalizes the notion of substitution by giving a type to the previously discussed notion of hypothetical judgement. He does this by defining as canonical element of this type, objects able to be applied to elements of one of the constituent types and yielding as results of that application elements of the other constituent type. In other words, he construct the type of functions from one previuosly defined type to other previously defined type.

In this subsection we interpret the above mentioned type seeing it within the context of problems and relate it to the problem sequencing, or better problem composition of GPT.

Given a problem A and an indexed problem B(x) Prob (x ∈ A) we can construct the problem A seq B, which says nothing more than "in order to solve B first solve A", or in a constructive viewpoint " a solution to A seq B is a program that to each solution of A gives a solution to B ". This type is the cartesian product of the set-theoretical interpretation in ITT. Let's see how are its rules.

Π-Formation

$$\frac{A\ \mathrm{Prob} \qquad [x \in A]\ B(x)\ \mathrm{Prob}}{(\Pi x \in A)\ B(x)\ \mathrm{Prob}} \qquad\qquad \frac{A = C \qquad [x \in A]\ B(x) = D(x)}{(\Pi x \in A)\ B(x) = (\Pi x \in C)\ D(x)}$$

, where (Πx ∈ A) B(x) = A seq B.

Intuitively, a canonical solution to A seq B is something that from a solution to A yields a solution to B. That is, it is able to particularize the problem B to that solution of A and solve this particularisation. This is done by using the parametrised solution to B that is b(x). We remind that B must be formally indexed by A, since this is a necessary condition to

deduce "A seq B Prob". However, that indexing may be nonsense, for we may get B without having any free variable. This is the case of the logicistic interpretation of (Πx ∈ A) B as the intuitionistic implication, i.e, as A ⟶ B.

In the context of programs, a (canonical) program solves (Πx ∈ A) B(x) must be executed by executing the program b(x), that solves B(x), having as input the output of the program that solves A. That program is sugestly called as λxb(x), and the application of it to a (a solution to A) is denoted by AP(λxb(x), a), which is nothing more than b(a). That is the meaning of the below rules.

Π-introduction

$$\frac{\begin{array}{c}[x \in A]\\ b(x) \in B(x)\end{array}}{\lambda x b(x) \in B(x)} \qquad \frac{\begin{array}{c}[x \in A]\\ b(x) = d(x) \in B(x)\end{array}}{\lambda x b(x) = \lambda x d(x) \in B(x)}$$

, where there is the restriction of x does not occur free in any other premiss else the one discharged by the rule.

We have to remark that λxb(x) is a canonical solution even whether b(a) is not for some a. For we must see it as representing the meta-program that from the general solution b(x) obtain the canonical solution b'(x), which has the ability of apply it to a canonical solution a' of A, obtained from the general solution a. So we have the following rules.

Π-elimination

$$\frac{c \in (\Pi x \in A)\ B(x) \qquad a \in A}{AP(c,\ a) \in B(a)}$$

$$\frac{c = d \in (\Pi x \in A)\ B(x) \qquad a = b \in A}{AP(c,\ a) = AP(d,\ a) \in B(a)}$$

, where c and d are general solutions which can be converted (by a meta-program) to a canonical solution of the kind λxb(x).

We can roughtly say that λxb(x) is nothing more than a name to b(x). And we can say that its operational meaning is given by the following equality rule.

$$\frac{[x \in A]}{\dfrac{a \in A \qquad b(x) \in B(x)}{AP(\lambda x b(x), a) = b(a) \in B(x)}}$$

$$\frac{c \in (\Pi x \in A)\ B(x)}{c = \lambda x AP(c, x) \in (\Pi x \in N) B(x)}$$

Endler [Endler86], by studying the first order formulas that defines the relation q in a general problem <D, R, q> proposed the problem implication of two problems as the one which the relation is defined by the implication of the two formulas that define the respective problems. Thus we can also relate A seq B to the problem implication, since the type in question is already related to the logical implication by the logicistic interpretation of ITT.

An interesting question to be answered : As we can relate A seq B to the problem composition as well as the problem implication, which of the two relationships is more interesting ? Are they fundamentally the same ?

Anyway, we can note that the programs that solve A seq B as being the sequencing of programs, can be denoted by a;b where a solves A and b solves B. This notation does not conflict the previous one, since the particularisation is done by the execution of a itself.

As an example of use of the above rules as components of a problem-calculus, let's show that the abstract problem $A \longrightarrow (B \longrightarrow A)$ is solvable. That is, we want to show $A \longrightarrow (B \longrightarrow A)$ **Prob** by using the above rules. In order to prove the solvability of any problem in ITT, we have to show a solution to it, and that is the only way to do this, i.e., we have to find a c such that $c \in A \longrightarrow (B \longrightarrow A)$. Thus :

1- We assume :

- A **Prob**
- $B(x)$ **Prob** $(x \in A)$

So, with these two hypothesis we can conclude that $B(x)$ **Prob**. Moreover we can from $x \in A$, to conclude that $\lambda y x \in (\Pi y \in B(x))\ A$, since A **Prob** and $B(x)$ **Prob** have as consequence $(\Pi y \in B(x))\ A$ **Prob**. Note that the hypothesis $x \in A$ is implicit in the above reasoning.

Thus we build the following deduction, which is the calculus counterpart of our reasoning.

2-

$$\frac{\dfrac{[x \in A]}{\lambda yx \in (\Pi y \in B(x))\ A}}{\lambda x \lambda yx \in (\Pi x \in A)(\Pi y \in B(x))A}$$

We prove $A \longrightarrow (B \longrightarrow A)$ **Prob** by showing that $\lambda x \lambda yx \in A \longrightarrow (B \longrightarrow A)$. Note that the solution has the form of a program that says : in order to solve the problem A seq (B seq A) we must apply $\lambda x \lambda yx$ to a solution of A finding the solution λya to B seq A, hence applying it to a solution to B we have a as a solution to A, what is true, since we begin by assuming that $a \in A$. This is the procedure to solve the problem $A \longrightarrow (B \longrightarrow A)$.

We would like to point out that the above deduction is not completely formal, since we used out-proof arguments in order to affirm our hypothesis $(x \in A)$ and the conclusion of the Π-introduction.

As the reader can note by the above example, ITT is (can be viewed as) a constructive calculus to prove that such and such object is a problem. But, to the constructivism a problem is defined by giving how are its solutions (again, what is different of "which are its solutions"), so to prove that a given object is a problem we have to find how are its canonical solutions. Thus, the calculus serves to this, since the use of introduction and elimination rules as well formation and equality rules become precise the structure of the problem by specifying, not necessarily giving, its canonical solutions. Hence, we can think ITT as a calculus to solve in an abstract level problems, but not to solve really problems, since it is only able to say how are the canonical solutions. However, we can extract from a prove a real program by particularization.

Next we show that (A seq (B seq C)) seq ((A seq B) seq (A seq C)) **Prob**, which, in our opinion is more formal :

1- We assume :

- A **Prob**
- $B(x)$ **Prob** $(x \in A)$
- $C(x,y)$ **Prob** $(x \in A,\ y \in C(x,y))$

This is done, since a proof of Obj_1 seq Obj_2 Prob is build canonically by assuming Obj_1 Prob. The above assumptions are the ones that assure (A seq (B seq C)) Prob. Thus from them we can assume as formal hypothesis :

- $x \in A$
- $f \in (\Pi x \in A)\ B(x)$
- $g \in (\Pi x \in A)\ ((\Pi y \in B(x))\ C(x,y))$

2- Then, we build the following deduction :

$$\dfrac{\dfrac{\dfrac{\dfrac{[x \in A][f \in (\Pi x \in A)B(x)]}{Ap(f,x) \in B(x)} \quad \dfrac{[x \in A][g \in (\Pi x \in A)((\Pi y \in B(x))C(x,y)]}{Ap(g,x) \in (\Pi y \in B(x))C(x,y)}}{Ap(Ap(g,x),Ap(f,x)) \in C(x,\ Ap(f,x))}}{\lambda x Ap(Ap(g,x),Ap(f,x)) \in (\Pi x \in A)C(x,\ Ap(f,x))}}{\lambda f \lambda x Ap(Ap(g,x),Ap(f,x)) \in (\Pi f \in (\Pi x \in A)B(x))(\Pi x \in A)C(x,\ Ap(f,x))}}{\lambda g \lambda f \lambda x Ap(Ap(g,x),Ap(f,x)) \in}$$

$(\Pi g \in (\Pi x \in A)((\Pi y \in B(x))C(x,y))(\Pi f \in (\Pi x \in A)B(x))(\Pi x \in A)C(x,\ Ap(f,x))$

, where the conclusion is only :

$\lambda g \lambda f \lambda x Ap(Ap(g,x),Ap(f,x)) \in$

(A seq (B seq C)) seq ((A seq B) seq (A seq C))

Again, the solution is a program that says :

1- Execute f with the result of the execution of x as input (remind that x is a program)

2- Execute "partially" g with the result of the execution of x as input.

3- Execute the result obtained in 2 with the result obtained in 1 as input.

Here, executing "partially" a program with a value as input is to develop the execution until it cannot be possible to continue evaluation (execution) without the other input values. Well, in the context of λ-calculus this is trivial.

Let's see now an example that shows the abstract level we get by using ITT.

Let A be the problem of either to read the first element of a sequential file or to inform that the file is empty without modifying the file. We assume $a \in A$, that is, a is a program that solves A. Now we want to show that the problem of reading the nth

element of a sequential file (n fixed), informing its emptiness if is the case, is solvable.

Well, first we have to find the structure of the problem. We can do this by pointing out the fact that reading the second element is the same of read the first of the rest of the file after one read operation. We can go on, in order to get the nth case for any n. So our problem of read the nth is the composition (n times) of A with itself. That is $A \rightarrow (A \rightarrow (A \rightarrow \ldots (A \rightarrow A)))$ where there are n A's in the expression. We will abbreviate the above notation as A^n. Thus we want to show for a fixed n that A^n Prob.

Thus the problem of read the first element of a file of lenght n has many cases, that are : 1- when the file was not read, 2- when the file was read only once time, 3- when the file was read only twice time .... n- when the file was read n-1 times, and finally when the file is empty. So, we can realize such kind of problem as being sliced into n+1 subproblems each of them related to its respective above case. But, there is something interesting in this kind of problem, that is we can index each of their parts with another of them. For example, we index the problem of read a once time read file with the not read file, and so on. In section II.4 we will show how it is that construction. For a while, we will assume that we can index this problem with itself, but reminding this really means the above explained.

Based on the above paragraph we assume $A(x)$ Prob $(x \in A)$. This means that knowing the old first element was read we know read the next (now the first in the new file). The same reasoning can be applied any times. Thus, it is reasonable to assume :

1-

$$A(x_1, x_2, \ldots, x_n) \text{ Prob } (x_1 \in A,\ x_2 \in A(x_1),\ \ldots,\ x_n \in A(x_1, \ldots, x_{n-1}))$$

, for any $n \geq 1$

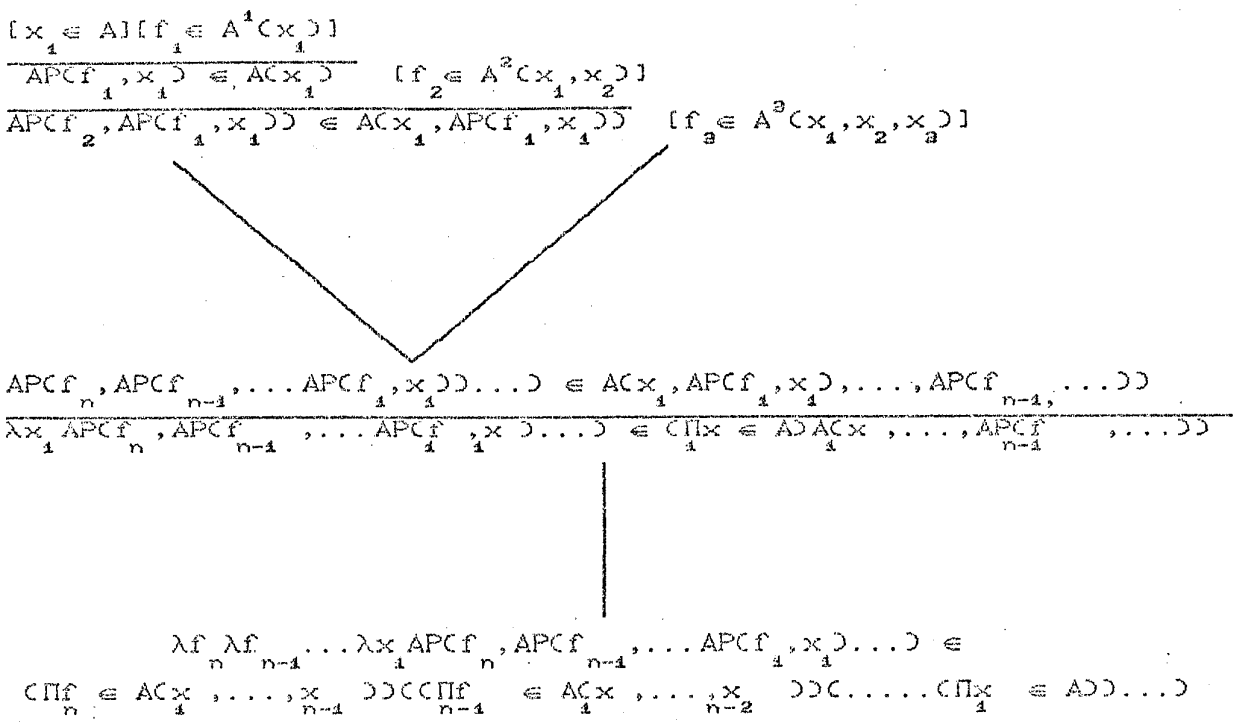
2- So we can assume the following hypothesis :

- $x_1 \in A$
- $f_i \in (\Pi x_1 \in A) \ldots ((\Pi x_i \in A(x_1, \ldots, x_{i-1})) \ldots) A(x_1, \ldots, x_i)$

, for $1 \leq i \leq n$, and we abreviate as

- $f_i \in A^i(x_1, \ldots, x_i)$

3- Thus we have conditions to build the following deduction :



, where the conclusion says that A^n has solutions of the form :

$$\lambda f_n \lambda f_{n-1} \ldots \lambda x_1 AP(f_n, AP(f_{n-1}, \ldots AP(f_1, x_1)\ldots)\ldots)$$

We would like to point out that the intuitive meaning of the above expression is just to apply a process of reading the first element any times sequencially. With the sake of clarity we can think that each f_i is just the same reading process from the viewpoint of program, but f_1 is particularized in execution time to the context of a previously read file, perhaps read by the same program. The same reasonins can be applied to the other f's as well as to the first reading done by x_1 (at least hypothetically).

The above example shows the abstraction power of the language, since we do not worry even with giving a name to the type of files.

We must note that we do not have conditions until now to specify the above example n varying. It is a necessary condition the fact that n is fixed in A^n.

II.2.1.1- cartesian product and the concept of problem reduction.

We can note a similarity between the concept of problem reduction and the type (Πx ∈ A)B(x). We say B is reducible to A iff any solution to B is also a solution of A (via some viewpoint changes, like data transformations and so on as explained in section I.2). The other way around, we can also say that each solution to A can be viewed also as a solution to B (again via transformations) if B is reducible to A. But this last utterance can be read as the fact that B is in some way indexed by A. So (Πx ∈ A)B(x) can denote that B is reducible to A, and the canonical element λxb(x) ∈ (Πx ∈ A)B(x) gets any solution to A and becomes it into a solution to B.

Based on the above relationship we prove now in ITT that if B is reducible to A and A is not solvable then B also is. First we must give the meaning of a non solvable problem in ITT. Well, we relate the non solvable problem with the empty set that is denoted by the type N_0. N_0 does not have introduction rule, since it does not have any element. Following the constructivistic view of problem, we must point out that there is only one non-solvable problem, since a problem is defined by giving how are its canonical elements and N_0 having none element is surely related to the non solvable problem. But N_0 has an elimination rule, that is:

$$\frac{c \in N_0}{R_0(c) \in C(c)}$$

, which says that if C is particularized by a solution c to N_0 then $R_0(c)$ is its solution. But the program $R_0(\ )$ is the partial program that cannot execute. Martin-Lof [Martin-Lof84] related it to the statement abort due to Dijkstra [Dijkstra76]. We prefer relate it to a completely undefined partial program. It is interesting to point out that N_0 is related to the absurdum (⊥) in the logical context, since that nothing can count as a proof to ⊥.

Thus we want to show that if (Πx ∈ N_0)B(x) Prob then B = N_0. In order to get B = N_0, we need to prove :

$$\frac{a \in B}{a \in N_0}$$

Since N_0 does not have elements then bottom-up direction is an equivalent of the *ex falso quodlibet* logical rule. Hence we must prove the other direction. In order to do this we assume that B has some element and search an inconsistency. We know that this is a legal intuitionistic inference, hence can be expressed in ITT.

Let's assume $a \in B$. Well, since each element of B has an equivalent canonical element $b(x)$ for some $x \in N_0$ a must have also. But, nothing belongs to N_0, so we get the inconsistency.

As the reader must have noted, the above proof is not formalized in ITT. This is because we do not have an explicity rule of equality between types. But we used logical inferences which can be expressed in ITT as show Martin-Lof in [Martin-Lof84]. Indeed, the proved fact has its intuitive meaning forced by the N_0-elimination rule.

II.2.2- Disjoint union and problem conjunction.

An important concept in ITT is that of the disjoint union of a family of sets. Well, this interpretation is related to the set-theoretical judgement. In the logical judgement this concept is interpreted as the existential quantification as a counterpart to the universal quantification that together the implication can be the logical interpretation of the cartesian product. In this section we search the "problematic" judgement of the disjoint union.

A common way to specify the objects of data types which are heterogeneous aggregates is to put them together in a sole structure marking each of them in such a way that we know their respective original types. A well-known data structure that obeys this kind of construction is the RECORD of pascal.

In intuitionistic logic we can say that a proposition of the type $\exists x P(x)$ is true iff there is a witness to $P(x)$. Well, in provability terms the above utterance is equivalent to : A type $\exists x P(x)$ is provable iff there is a term t such that there is a proof Π of $P(t)$. Thus we can roughly say that pairs (t,Π) of the above kind determinines the type $\exists x P(x)$. It is clear that we need something that says the types of each element of the pair. We can also note that Π depends on t, so we can think t as an index to Π. Hence, in order to specify the type $\exists x P(x)$ we need a structure that keeps together a proof Π and its index t. This is a type very related to the kind described in the above paragraph.

Motivated by the above observations among others, Martin-Lof gave to ITT this power of specification, or better, ways to build such type. This is just the idea of disjoint union of a family of sets. This type is also known as coproduct of a family of sets. Firstly, it is built from the assumption that we have a family of sets indexed by another set, or in a more general view a hypothetical judgement. Its elements are just pairs which have as first element an index and as second the respective indexed set.

In the context of problems we see the indexing of problems as particularization. Thus a problem of the described type must have

as canonical solutions , pairs which have as first element a canonical solution to a problem and as second a solution to the particularization of a problem by the first element. This can be better seen in the following rules :

Σ-Formation

$$\frac{A\ \mathrm{Prob} \qquad B(x)\ \mathrm{Prob}\ [x \in A]}{(\Sigma x \in A)\ B(x)\ \mathrm{Prob}}$$

Σ-Introduction

$$\frac{a \in A \qquad b \in B(a)}{(a,b) \in (\Sigma x \in A)\ B(x)}$$

Well, which problem in GPT could be associated to the above problem type? We would like to answer that question in the same determined way that we answered for the cartesian product case. But, indeed we do not find a good relationship to the coproduct in the general case. However, if we built the coproduct type with no indexing, then we could relate it to the problem intersection in GPT. This is very interesting, since in that case we can also interpret the coproduct as the logical conjunction in the logical interpretation as do Martin-Lof.

The problem intersection or conjuntion of two problems $\langle D_1, R_1, q_1 \rangle$ and $\langle D_2, R_2, q_2 \rangle$ in GPT is simply the problem defined by $\langle D_1 \sqcap D_2,\ R_1 \sqcap R_2,\ q_1 \sqcap q_2 \rangle$. But we require that D_1 and D_2 must belong to the same type universe, with the same about R_1 and R_2. If we guess the definer relations q_1 and q_2 as being determined by its definer formulas α_1 and α_2, then the relation that defines the conjuntion of the two problems is defined by the formula $\alpha_1 \wedge \alpha_2$. This last viewpoint motivated the name of that kind of problem in GPT. However, if the above requirement about the domains and results is not true then the problem defined by $\alpha_1 \wedge \alpha_2$ is a kind of cartesian product (in set theory, not in ITT) between the problems. The simplest case of a cartesian product problem in GPT is $\langle D_1 \times D_2,\ R_1 \times R_2,\ q_1 \times q_2 \rangle$, which can be defined by a formula $\alpha_1 \wedge \alpha_2$ with no occurence of common free variables between α_1 and

α_2.

If there are not common free variables between α_1 and α_2, then the problem type associated to the conjunction (intersection or cartesian product) is $(\Sigma x \in A)B$, where A is the problem type associated to the problem defined by α_1 and B is the one associated to the problem defined by α_2, for the solutions to B does not depend on any solution to A. If it is not the above case then we must related to $(\Sigma x \in A)B(x)$ and so we must first to find a solution $a \in A$ and after finding $b(a) \in B(a)$.

From the relationships discussed in the above paragraph we see that solutions to $(\Sigma x \in A)B$ are pairs of independent programs, so they can represent independent parallel procedures (programs). But if the problem type is $(\Sigma x \in A)B(x)$ we cannot think about a trivial kind of parallelism as the above mentioned. We do not know even if we can think about parallelism.

It is clear that if a problem C would be simultaneously indexed by to problems and one of them is indexed by the other, we had a case of indexing by a type $(\Sigma x \in A)B(x)$ problem. So, from a solution c to $(\Sigma x \in A)B(x)$ we could want to know the solution to C. This is done by an operator (E) that gets c becoming it into a canonical solution c' of $(\Sigma x \in A)B(x)$ which is certainly of the form (a,b) $a \in A$ and $b \in B(a)$, then it particularizes d (a solution to C) with the pair $(a;b)$. This operator has metaprogram's features as the reader can note. It is clear that this operation done by E must give solutions to C and besides preserve canonical solutions, i.e, if $d(a,b)$ has an associated canonical solution, then $E((a,b),(x,y)d(x,y))$ must have the same canonical solution. The notation $(x,y)d(x,y)$ only denote that x and y are bound in d. Martin-Lof did not use the λ notation, for it could cause confusion with the canonical elements of the cartesian product in ITT. The above ideas are summarized in the following rules.

Σ-Elimination

$$\frac{c \in (\Sigma x \in A)B(x) \qquad \begin{array}{c}[x \in A,\ y \in B(x)] \\ d(x,y) \in C((x,y))\end{array}}{E(c,(x,y)d(x,y)) \in C(c)}$$

Σ-Equality

$$\frac{a \in A \quad b \in B(a) \quad d(x,y) \in C((x,y)) \quad [x \in A,\ y \in B(x)]}{E((a,b),(x,y)d(x,y)) = d(a,b) \in C((a,b))}$$

We will denote A and B Prob to denote the problematic interpretation of (Σx ∈ A)B(x) Prob.

Well, once time it was given a solution c ∈ (Σx ∈ A) B(x) we could want to know the solution to A that particularized the solution to B(x), as well as the solution to that particularization. In other words we could want to know the solutions that take part in the solution to (Σx ∈ A)B(x). In terms of the logical interpretation of the type in question we want to have means to know, separately, the witness as well as the proof. Well, if we instantiate the Σ-elimination rule with A taking the role of C((x,y)), thus having d(x,y) as being x, we will conclude that E(c, (x,y)x) ∈ A, with c ∈ (Σx ∈ A)B(x). This is just what we wanted. The same could be done regard to B(x) instead of A and concluding E(c, (x,y)y) ∈ B(x). We call these instances of the E meta-program as left projection (P) and right projection (Q) respectively. We have so the following rules for them.

Left projection

$$\frac{c \in (\Sigma x \in A)B(x)}{P(c) \in A}$$

Right projection

$$\frac{a \in A \quad b \in B(a)}{Q(a,b) \in B(a)}$$

And by using the equality rules we have.

$$\frac{a \in A \quad b \in B(a)}{P((a,b)) = a \in A} \qquad \frac{a \in A \quad b \in B(a)}{Q((a,b)) = b \in B(a)}$$

It is interesting to note that observing the above relationships we conclude that GPT is more general than ITT as a description language to problems. For, in GPT we can describe a unsolvable problem from two solvable problem, as for example by intersection. However, this cannot happen with ITT, since the concept of intersection has its counterpart in the coproduct type

just in the case of dependence between the problems (free variables occuring in the two definer formulas), the intersection must be related to a type $(\Sigma x \in A)B(x)$. But, since A and B are solvables then there are canonical elements $a \in A$ and $b(a) \in B(a)$, thus $(a,b(a)) \in (\Sigma x \in A)B(x)$, hence is not N_o and so is solvable. In other words, ITT only has ability to describe as a type coproduct the intersection of solvable problems.

We remind that the above rules can be viewed as a deductive system which we hope is sound regard to some semantics and that semantics must be such that if the premisses are solvables then the conclusion also is. Thus it is impossible to build a non solvable problem from solvables premisses.

The above observation can be viewed also as a consequence of the fact that all non solvable problems are included in the same expressability condition, that is the problem N_o. Hence, for one that does not give meaning to the strictly classical propositions, ITT has the same expressability power of GPT. The same is true about one that is interested in the computational aspect of problems. For, we have already seen that ITT proves that an object is a solvable problem by giving the form of its canonical solutions, which are indeed programs.

As an example, we show that A seq (B seq (A and B)) Prob, which has $A \longrightarrow (B \longrightarrow (A \wedge B))$ as its logical interpretation.

The proposition format needs an indexing of B by A, so we assume :

1- A Prob

2- B(x) Prob $(x \in A)$

Thus we can assume :

1- $x \in A$

2- $y \in B(x)$

So we build the following derivation :

$$\dfrac{\dfrac{\dfrac{x \in A \qquad y \in B(x)}{(x,y) \in (\Sigma x \in A)\ B(x)}}{\lambda y(x,y) \in (\Pi y \in B(x))(\Sigma x \in A)B(x)}}{\lambda x\lambda y(x,y) \in (\Pi x \in A)(\Pi y \in B(x))(\Sigma x \in A)B(x)}$$

Thus the conclusion is just $\lambda x\lambda y(x,y) \in A$ seq (B seq (A and B)) from which we conclude A seq (B seq (A and B)) Prob. And

the solution (program) says simply that " to solve the problem A and B from A and from B, get a solution to A and put together a solution to B ".

Similary, we can prove (A and B) seq A Prob as well (A and B) seq B. But these proofs were already commented when we introduced the left and right projections respectively.

In order to show another example of the abstraction level of ITT we show that " verifying whether among the n first elements of a sequential file there is one equal to a fixed value " is a problem, under the assumptions that reading the first element is a problem and the equality test also is. The equality informs via a message the result of the test, or better, this specification take part in the equality problem specification (in GPT for example).

We will call the problem of read the first element of a sequential file A. Thus the problem of test if the first element of a file is equal to some fixed value can be expressed by B(x) Prob (x ∈ A), since we need a solution to the reading problem in order to test equality. B(x) can be viewed as a particularization of the equality problem. Thus, our problem when n = 1 is (Σx ∈ A)B(x), for we need the both solutions, to A and to B(x), in order to solve the whole problem. If n > 1 then we can reduce (a GPT concept !) to the simplest, by a similar way we did in the example of reading a sequential file. As in the mentioned example we will assume that we can index A with itself with the sake of simplicity of notation. Besides, we must realize B(x) as being the equality test for all cases of the problem A, i.e., for none read operation, for one, etc. This is the assumption number 2 below.

Thus, in order to prove the problem, we can assume :

1- $x_1 \in A$

2- $b(x) \in B(x)$ $(x \in A(x_1,\ldots,x_i))$ $1 \leq i \leq n$

3- $f_i \in \Lambda^i(x_1,\ldots,x_i)$

With the sake of space economy we build the derivation to the case when n = 2. The other cases are build from that by an iteration process.

From the assumption 2 above, we can assume :

2'- $\lambda y b(y) \in (\Pi y \in A(x))B(y)$

First we build the following deduction, which uses a

substitution rule already presented. We call it as ONE.

$$\dfrac{\dfrac{\dfrac{\dfrac{x \in A \qquad b(x) \in B(x)}{(x, b(x)) \in (\Sigma x \in A)B(x)}}{P((x, b(x)) \in A} \qquad \dfrac{f_1 \in (\Pi x_1 \in A)A(x_1) \qquad [y \in A(x)]}{b(y) \in B(y)}}{Ap(f_1, P((x, b(x)))) \in A(P((x, b(x))))}}{b(Ap(f_1, P((x, b(x))))) \in B(Ap(f_1, P((x, b(x))))) } \qquad \lambda y b(y) \in (\Pi y \in A(x))B(y)$$

Now, we use the above derivation to get our conclusion. Here we use ξ to short $Ap(f_1, P((x, b(x))))$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[x \in A] \qquad b(x) \in B(x)}{(x, b(x)) \in (\Sigma x \in A)B(x)}}{P((x, b(x)) \in A} \qquad [f_1 \in (\Pi x \in A)A(x)]}{\xi \in A(P((x, b(x))))}}{(\xi, b(\xi)) \in (\Sigma y \in A(P(x, b(x))))B(y)}}{\lambda f_1 (\xi, b(\xi)) \in (\Pi f_1 \in ((\Pi x \in A)A(x)))(\Sigma y \in A(P(x, b(x))))B(y))}}{\lambda x \lambda f_1 (\xi, b(\xi)) \in (\Pi x \in A)((\Pi f_1 \in ((\Pi x \in A)A(x)))(\Sigma y \in A(P(x, b(x))))B(y)))} \qquad \dfrac{\dfrac{[x \in A] \qquad [f \in (\Pi x_1 \in A)A(x)]}{\text{ONE}}}{b(\xi) \in B(\xi)}$$

We note that $b(x) \in B(x)$ remains in the deduction. But this must not worry us, since it can be removed by assuming $(\Pi x \in A)B(x)$ Prob, then including the assumption $g \in (\Pi x \in A)B(x)$ in the proof.

We see that following our intuition about the problem we obtained as conclusion B seq (A seq (A ...(A seq (A and B))..). This happens, because of the form of the program we obtained to solve the problem. It gets the solution to the equality problem and uses it from the first reading on, doing only particularizations. While the reading part of the solution must update itself regarding to the file. We remind that the assumption about $b(x) \in B(x)$ works out for any reading problem (none reading, one reading, etc). We can say that this proof proceeded by a hybrid bottom-up x top-down process. We think that this is a

consequence of the small distance between ITT and the programming language concept. It is interesting to note in this case that it was easier to build the proof based on the mental procedure than state previously a formulae to be proved.

II.2.3 - The disjoint union of two sets and the problem alternation.

In the previous sections, we used a kind of auto-indexing in order to prove problems, or better, theorems in ITT problematic interpretation, over input sequential files. Well, in this section we will give a problematic interpretation of the disjoint union of two sets [Martin-Lof84] that assures the soundness of the above mentioned use.

Remind the definition of a problem $P = \langle D; R, q \rangle$. We can define this problem from n distinct subproblems $P_i = \langle D_i, R_i, q_i \rangle$ $(i = 1,n)$ such that $D_i \cap D_j = \emptyset$ for $i \neq j$ and $D = D_1 \cup \ldots \cup D_n$ as well $R = R_1 \cup \ldots \cup R_n$. That is, P is defined by the alternative analysis of the P_i's or in another way P is decomposed into them. Note that the above used union operation can be viewed as the disjoint union operation.

ITT has a type set-theoretically related to the disjoint union. So, we relate that type to the problem defined by an alternative analysis of cases, which we call problem alternating.

We can realize that a solution to a problem constituted by two alternative cases is a solution to each of them, if both are solvables, together with a selection mechanism. The selection only chooses the convenient solution by checking what subproblem must be taken into account. We remind that they are disjoint. So, done this selection a solution to the whole problem is a solution to one of its constituent parts. We can see the selection part imbeded in the problem alternating concept, since all of them have this part. Thus, for our purposes we can abstract this part from the problem alternating description. Then, doing this abstraction, a solution to a problem constituted by two alternative cases A and B is only a solution either to A or to B. The solution, to A or to

B, itself denotes the choice.

So, from our above observations a solution to a problem alternating can be denoted by a solution to one of its components together with the information from which of them is this solution. If we use two new constants i and j to denote this information, then the canonical solutions to the alternation of A and B is either i(c) or j(c) where c is a solution to A or to B respectively. Then i denotes that A was choosen and j that B was. Here, we restricted ourselves to the case A and B are solvables.

ITT uses the symbol + to denote the disjoint union, we will use case to denote the "problematic" judgement of +. Note that we have specialized the ITT notation w.r.t. our way of judgement. Thus, according to the above explained we have the following rules to the case which are the rules to +.

case-Formation

$$\frac{A\ \text{Prob} \qquad B\ \text{Prob}}{A\ \underline{\text{case}}\ B\ \text{Prob}}$$

case-Introduction

$$\frac{a \in A}{i(a) \in A\ \underline{\text{case}}\ B} \qquad \frac{b \in B}{j(b) \in A\ \underline{\text{case}}\ B}$$

If we wanted to express the indexing of a problem C(x) by a problem alternating A case B how would be the solutions to C(x) ? Well, an indexing by a problem alternating A case B presumes an implicit alternating indexing by each of A and B. So, a canonical solution to C(x) (x ∈ A case B) must verify whether a solution c to A case B is of the form i(a) for a ∈ A or of the form j(b) for b ∈ B. According to the case we must choose the indexed solution to the respective indexing (by A or by B). It is clear that the above instructions are the description of the meta-program that gives a canonical solution to C(c) from a solution c ∈ A case B. The above ideas are summarized in the below elimination rule, where D represents the mentioned meta-program.

case-Elimination

$$\frac{c \in A \text{ case } B \qquad \begin{matrix}[x \in A] \\ d(x) \in C(i(x))\end{matrix} \qquad \begin{matrix}[y \in B] \\ e(y) \in C(j(y))\end{matrix}}{D(c, (x)d(x), (y)e(y)) \in C(c)}$$

Note that in order to describe the indexing by a problem alternating we had to use the selection process embeded in the problem alternating concept. But, we did this in the meta level.

Finally, we give the meaning of D in a rule-stated form by giving the equality rules.

case-Equality

$$\frac{a \in A \qquad \begin{matrix}[x \in A] \\ d(x) \in C(i(x))\end{matrix} \qquad \begin{matrix}[y \in B] \\ e(y) \in C(j(y))\end{matrix}}{D(c, (x)d(x), (y)e(y)) = d(a) \in C(c)}$$

$$\frac{b \in B \qquad \begin{matrix}[x \in A] \\ d(x) \in C(i(x))\end{matrix} \qquad \begin{matrix}[y \in B] \\ e(y) \in C(j(y))\end{matrix}}{D(c, (x)d(x), (y)e(y)) = e(b) \in C(c)}$$

This type problem is interpreted as the logic disjunction in the logical way of judgement. This agree with our interpretation, since the definer formula to the relation of a GPT description of a problem alternating is a disjunction of the definer formulas to the relations of the two constituent problems.

As an example of such type in the problem calculus we prove (A seq C) seq ((B seq C) seq ((A case B) seq C)) Prob., which says how is the sequencing with a problem alternating.

We assume :

- C(z) Prob (z ∈ A case B)

Thus, we can assume C(i(x)) Prob (x ∈ A) and C(j(y)) Prob (y ∈ B). Consequently, we also assume A Prob and B Prob.

Thus we can assume :

1.- x ∈ A

2- $y \in B$

3- $f \in (\Pi x \in A)C(i(x))$

4- $g \in (\Pi y \in B)C(j(y))$

Next we build the following deduction.

$$\dfrac{\dfrac{\dfrac{\dfrac{x \in A \quad [f \in (\Pi x \in A)C(i(x))]}{[z \in A \ \underline{case}\ B] \quad AP(f,x) \in C(i(x))} \qquad \dfrac{y \in B \quad [g \in (\Pi y \in B)C(j(y))]}{AP(g,y) \in C(j(y))}}{D(z,(x)AP(f,x),(y)AP(g,y)) \in C(z)}}{\lambda z D(z,(x)AP(f,x),(y)AP(g,y)) \in (\Pi z \in A \ \underline{case}\ B)C(z)}}{\lambda g \lambda z D(z,(x)AP(f,x),(y)AP(g,y)) \in (\Pi y \in B)C(j(y)) \ \underline{seq}\ (\Pi z \in A \ \underline{case}\ B)C(z)}}{\lambda f \lambda g \lambda z D(z,(x)AP(f,x),(y)AP(g,y)) \in (\Pi x \in A)C(i(x)) \ \underline{seq}\ ((\Pi y \in B)C(j(y)) \ \underline{seq}\ (\Pi z \in A \ \underline{case}\ B)C(z))}$$

The above deduction shows that we can deduce (A $\underline{seq}$ C) $\underline{seq}$ ((B $\underline{seq}$ C) $\underline{seq}$ ((A $\underline{case}$ B) $\underline{seq}$ C)) Prob by assuming $x \in A$ and $y \in B$. But, these assumptions are non-formal consequences of A Prob and B Prob respectively. However, these assumptions are necessary in order tyo prove anything about A and B. We note that the conclusion has x and y as free variables, which indeed are saying, for any solution $x \in A$ and any solution $y \in B$ we have that $\lambda f \lambda g \lambda z D(z,(x)AP(f,x),(y)AP(g,y))$ is a solution to our problem. note that the solution has the intuitive meaning about the indexing of a problem by an alternating one. If we wanted formalize the conclusion we could use two Π-introduction rules, thus bounding x and y. Then, we would have A $\underline{seq}$ (B $\underline{seq}$ ...) Prob as conclusion. Note that this deduction formalizes the indexing by a problem alternating, mentioned in a previous paragraph in this subsection

Let's turn to our problem of reading a sequential input file from none until n readings. We split the problem into n+1 disjoint subproblems. And this was done in such a way that each of them was indexed by other. Let's repeat the construction.

A_1 is the problem of read a sequential input file that was not read. $A_2(x)$ is the problem of read a file that was read one time, thus it presumes a solution $x \in A_1$. We go on until :

$A_n(x_1,\ldots,x_{n-1})$ Prob $(x_1 \in A_1, \ldots, x_{n-1} \in A_{n-1}(x_1,\ldots,x_{n-2}))$

So the problem of reading a file from none until n readings can be viewed as a problem alternating of these problems. Since the problem alternating is built two by two we have only to build step by step until our desired problem building. With the sake of clarity we do the building when n = 2. Thus :

$$\frac{\begin{array}{c} x \in A_1 \\ | \\ a_1(x) \in A_2(x) \end{array}}{j(a_1(x)) \in A_1 \ \underline{case}\ A_2(x)}$$

$$\frac{x \in A_1}{i(x) \in A_1 \ \underline{case}\ A_2(x)}$$

Thus, we proved that in a problem alternating which one constituent problem is indexed by the other, the problem itself can be indexed by the former. Hence, the use of the assumption $A(x)$ Prob $(x \in A)$ is sound, since we can see in the above conclusion $A(x)$ as A_1 $\underline{case}$ $A_2(x)$ and A as A_1. It is clear that the result can be extended to the general case $A_n(x_1,\ldots,x_n)$ $\underline{Prob}$ $(x_1 \in A_1, \ldots, x_n \in A_{n-1}(x_1,\ldots,x_{n-1}))$ by iterating the above building process.

An use, which is undoubtly impredicative, is indexing the problem alternating $(A_1$ $\underline{case}$ $A_2(x))$ with the indexed constituent problem $(A_2(x))$. But, the reader can observe that we did not use such kind of assumption in section II.2.1.

We would like to point out that we can build a problem alternating by using a not solvable problem as one of its constituents. However, we will not remove this partial undefiness from any problem indexed by it. Thus with the sake of easiness we will not use to build a problem from the not solvable (N_0).

II.2.3.1 - $\underline{Decomposition\ of\ problems\ and\ the\ disjoint\ union}$

Let A case B be a problem. We already saw that in order to solve it we have to solve A and solve B, according to the case. Then we use the solution found as a solution to A case B. In other words we can say that A + B is decomposable into A and B. The recombination function is the identity and the subdivision function is the choice between cases, which is determinated by the type of the subproblems themselves.

Let P be a decomposable problem, which is decomposed into only two subproblem. Well, the ways of recombination of the solutions to both, in order to get a solution to P, are variations of alternation and conjuntion. Thus we can roughly conclude that if a problem is decomposable into two other problems, then such problem in ITT is either of type A case B or of type $(\Sigma x \in A)B$ where A and B are the ITT vision of its subproblems.

We remind that if a problem is a conjuntion of two other problems, then it clearly is a decomposable problem.

Thus, from the above discussed we conclude :

Theorem.* A problem P is decomposable into subproblems P_1 and P_2 iff P^{ITT} is either of type P'_1 case P'_2 or of type $(\Sigma x \in P'_1)P'_2$, where $P_i^{ITT} = P'_i$ $(i = 1,2)$.

We cannot include in the above theorem the type seq, since its components are not at the same level of decomposition. Do not forget the type seq is related to the reduction which can be viewed as a unary decomposition.

III - A proposal of a programming methodology based on the relationship ITT x GPT.

In this section we define a scheme of a programming methodology which drives its user forward a solution (program) to a problem P, from a GPT-specification of P. It is clear that our proposal does not have its scope well-defined, since it is actually a scheme. We give two examples of its use.

III.1 - The proposal.

We have as data to the methodology a specification <D, R, q> to a solvable (recursive) problem P. We assume that q is given by one of its defining formulas. We call it α.

In order to apply the methodology, the following conditions are need.

1- D can be effectively described in ITT.

2- R can be effectively described in ITT.

Thus, our methodology has the following steps :

M1- Find in α its components of the kind $t \in T$, where T is either D or R.

M2- Replace the above mentioned components by $t \in T'$, where T' is T translated to ITT. We call these replaced parts set-parts.

M3- Translate α into α', by interpreting the logical constants occuring in α as their respective types in ITT. That is, apply the logicistic way of judgement to α producing α' in ITT. Then, read α' as a problem description in ITT. That is, apply our relationship to the parts that are not set-parts.

M3.1- Define the set Axioms = $\{$ A **Prob** / A is a set-part of $\alpha'\}$

M4- Prove α' **Prob** from Axioms and possibly the inference

rules about new types (user-types) defined in the translation to ITT.

M5- Indeed, A proof of α' Prob is a proof of $t \in \alpha'$. Then, find such t in the proof obtained in M4. This t read as a program is the solution to the problem P.

III.2- The problem Axiom of Choice.

This example is an addaptation of the proof of the Axiom of Choice shown in [Martin-Lof84].

The problem description is already given with D and R expressed in ITT set-theoretical way of judgement, which is written as "A set" for a proposition A.

$D = (C(x,y)\ \text{set}\ (y \in (B(x)\ \text{set}\ (x \in A))))$

$R = (\Pi x \in A)\ B(x)$

$\alpha = (\forall x \in A)(\exists y \in B(x))C(x,y) \rightarrow$

$(\exists f \in (\Pi x \in A)B(x))(\forall x \in A)C(x,f(x))$

The above specification can be roughly read as "For any indexed family $B(x)$ $(x \in A)$ of non-empty sets exists a choice function". $C(x,y)$ intuitively means that it is possible to realize a set built only from elements of the family $B(x)$ $(x \in A)$. Martin-Lof used an equivalent (AC2 in [Rubin & Rubin70] pp 5) form of axiom of choice rather than the traditional.

The step M1 is already done by the specification itself. Thus the step M2 is not necessary.

We build $Ax = \langle C(x,y)\ \text{Prob}\ (y \in B(x))\ \text{Prob}\ (x \in A)),\ B(x)\ \text{Prob}\ (x \in A),\ A\ \text{Prob},\ (\Pi x \in A)B(x)\ \text{Prob}\ \rangle$. The reader can note that we include the implicitly assumed utterances that are need to assume $C(x,y)\ \text{Prob}\ (y \in B(x))\ \text{Prob}\ (x \in A))$.

The step M3 produces then α' :

$(\Pi x \in A)(\Sigma y \in B(x))C(x,y) \rightarrow (\Sigma f \in (\Pi x \in A)B(x))(\Pi x \in A)C(x,AP(f,x))$

, where $AP(f,x)$ is $f(x)$ in ITT.

Then, we prove α'.

$$\dfrac{\dfrac{\dfrac{\dfrac{[x \in A]\ [z \in (\Pi x \in A)(\Sigma y \in B(x))C(x,y)]}{Ap(z,x) \in (\Sigma y \in B(x))C(x,y)}}{P(Ap(z,x)) \in B(x)}}{\lambda x P(Ap(z,x)) \in (\Pi x \in A)B(x)} \qquad \begin{array}{c}[z \in (\Pi x \in A)(\Sigma y \in B(x))C(x,y)] \\ | \\ ONE \\ |\end{array}}{(\lambda x P(Ap(z,x)), \lambda x Q(Ap(z,x))) \in (\Sigma f \in (\Pi x \in A)B(x))(\Pi x \in A)C(x, Ap(f,x))}$$

$$\dfrac{}{\lambda z(\lambda x P(Ap(z,x)), \lambda x Q(Ap(z,x))) \in (\Pi x \in A)(\Sigma y \in B(x))C(x,y) \rightarrow (\Sigma f \in (\Pi x \in A)B(x))(\Pi x \in A)C(x, Ap(f,x))}$$

, where ONE is the following deduction :

$$\dfrac{\dfrac{\dfrac{[x \in A]\ \ z \in (\Pi x \in A)(\Sigma y \in B(x))C(x,y)}{Ap(z,x) \in (\Sigma y \in B(x))C(x,y)}}{Q(Ap(z,x)) \in C(x, P(Ap(z,x)))} \qquad \begin{array}{c}[x \in A], z \in (\Pi x \in A)(\Sigma y \in B(x))C(x,y) \\ | \\ TWO \\ |\end{array}}{Q(Ap(z,x)) \in C(x, Ap(\lambda x P(Ap(z,x)), x))}$$

$$\dfrac{}{\lambda x Q(Ap(z,x)) \in (\Pi x \in A)C(x, Ap(\lambda x P(Ap(z,x)), x))}$$

, where TWO is the following equality deduction :

$$\dfrac{\dfrac{\dfrac{\dfrac{[x \in A]\ \ z \in (\Pi x \in A)(\Sigma y \in B(x))C(x,y)}{Ap(z,x) \in (\Sigma y \in B(x))C(x,y)}}{P(Ap(z,x)) \in B(x)}}{\lambda x P(Ap(z,x)) \in (\Pi x \in A)B(x)} \qquad \begin{array}{c}[x \in A], [y \in B(x)] \\ | \\ C(x,y)\ \ \mathbf{Prob} \end{array}}{Ap(\lambda x P(Ap(z,x)), x) = P(Ap(z,x)) \in B(x)}$$

$$\dfrac{}{C(x,\ Ap(\lambda x P(Ap(z,x)), x) = C(x,\ P(Ap(z,x)))}$$

Thus we can read the λ-ITT-term in the conclusion of the deduction as the program that solves the problem axiom of choice

III.3 - simultaneous recursion.

To derive the next example of application of the methodology, we used a simultaneous recursion. In this subsection we show this mechanism as an instance of the wellordering type [Martin-Lof84]. This wellordering type describes the wellordered constructive sets.

It is clear that we define the wellordering with the aid of a set of support. Besides, we need the ordering relation which is given by a function such that applied to the set of support gives us the predecessors of a given element. If the set of support is empty then we reach an initial element of the ordering.

We show here the rules to the wellordering type (W). The interested reader is invited to see [Martin-Lof84] in order to get more details.

W-Introduction

$$\frac{a \in A \qquad b \in B(a) \to (Wx \in A)B(x)}{\sup(a,b) \in (Wx \in A)B(x)}$$

, where we can see A as the ordered set and B(a) the set of support which gives the predecessors of a by means of the function b. In the general case we need a family of set of support.

The W-Elimination can be viewed as the transfinite induction (or recursion) over the wellordering.

W-Elimination

$$\frac{c \in (Wx \in A)B(x) \qquad [x \in A,\ y \in B(x) \to (Wx \in A)B(x),\ z \in (\Pi v \in B(x))C(Ap(y,v))] \quad \vdots \quad d(x,y,z) \in C(\sup(x,y))}{T(c,(x,y,z)d(x,y,z)) \in C(c)}$$

And the operational meaning of T is given by the following equality rule.

W-Equality

$$\frac{a \in A \quad b \in B(a) \to (Wx \in A)B(x) \qquad [x \in A,\ y \in B(x) \to (Wx \in A)B(x),\ z \in (\Pi v \in B(x))C(Ap(y,v))] \quad \vdots \quad d(x,y,z) \in C(\sup(x,y))}{T(\sup(a,b),(x,y,z)d(x,y,z)) = d(a,b,\lambda v T(Ap(b,v),(x,y,z)d(x,y,z))) \in C(c)}$$

We can build the set of the natural numbers by using the type W, instead of take it as primitive. See [Martinn-Lof84].

In order to obtain the simultaneous recursion over two values, we need build a set that have the pairs (x,y) with either x or y equal to zero as the initial elements. Besides we need to have as only predecessor of (x',y') the pair (x,y). So our set have an infinite number of initial elements and each element has only one predecessor. We call 2ω this set. It is obtained by using the universe rules [Martin-Lof84] with respect to the W-type.

Below, we give the 2ω-Introduction rules as instances of W-Introduction by doing $A = N \times N$, $B((0,x)) = B((y,0)) = N_0$ and $B((x',y')) = N_1$. Where N_1 is the type (set) of the finite sets with only one element. As introduction rule to N_1 we have $0_1 \in N_1$. Thus :

$$\frac{x \in N}{sup((x,0), \lambda y R_0(y)) \in 2\omega}$$

$$\frac{y \in N}{sup((0,y), \lambda y R_0(y)) \in 2\omega}$$

$$\frac{sup((x,y),f) \in 2\omega}{sup((x',y'), \lambda z sup((x,y),f))) \in 2\omega}$$

Note that the rules are defined only to canonical elements, but this will not worry us, for we really will use a simpler version of 2ω.

Next we show the 2ω-Equality which explain the simultaneous resursion mechanism. To the two initial cases, we have :

$$\frac{d_{oy} \in C(sup((0,y), \lambda y R_0(y)))}{T(sup((0,y), \lambda y R_0(y)),(x,y,z)d_{oy}) = d_{oy} \in C(sup((0,y), \lambda y R_0(y)))}$$

$$\frac{d_{xo} \in C(sup((x,0), \lambda y R_0(y)))}{T(sup((x,0), \lambda y R_0(y)),(x,y,z)d_{xo}) = d_{xo} \in C(sup((x,0), \lambda y R_0(y)))}$$

To the other case, we do the following in order to get the instance of the W-Equality :

1- We take a ∈ A as (x',y') ∈ N x N.

2- Instead of the premiss about the predecessor function, we take the element sup((x,y),f) ∈ 2ω.

3- Since B((x',y')) has only one element (0_1), we replace z ∈ (Πv ∈ B((x',y')))(C(Ap(λysup((x,y),f), v)' by Ap(z,0_1) ∈ C(sup((x,y),f)).

4- Finally we replace a' ∈ A and b ∈ B(a) ⟶ 2ω by its 2ω-Introduction conclusion.

Thus :

$$\frac{\begin{array}{c}[(x',y') \in N \times N,\ sup((x,y),f) \in 2\omega,\ Ap(z,0_1) \in C(sup((x,y),f))] \\ \Big\downarrow \\ sup((a',b'),\lambda ysup((a,b),g)) \in 2\omega \qquad d(x,y,f,Ap(z,0_1)) \in C(sup((x',y'),\lambda ysup((x,y),f)))\end{array}}{T(sup((a',b'),\lambda ysup((a,b),g)),(x,y,f,z)d(x,y,f,Ap(z,0_1))) = d(a',b',g,T(sup((a,b),g),(x,y,f,z)d(x,y,f,Ap(z,0_1))))}$$

We can note that the rules depend mainly on the x and y and z, since the predecessor function is implicit in their values ,remind it is the natural predecessor function extendedto N x N, we can relate directly N x N to 2ω and so simplify the above rule such that :

$$\frac{\begin{array}{c}[(x,y) \in N \times N,\ z \in C(x,y)] \\ \Big\downarrow \\ (a',b') \in N \times N \qquad d(x,y,z) \in C(x',y')\end{array}}{T((a',b'),(x,y,z)d(x,y,z)) = d(a',b',T((a,b),(x,y,z)d(x,y,z))) \in C(a',b')}$$

, where we get (x',y') ∈ N x N from (x,y) ∈ N x N in the discharged assumption.

Thus, doing the same with respect to the initial cases and

writing all cases in a only rule we have the desired simultaneous recursion rule :

$$\frac{(a,b) \in N \times N \quad d_{oy} \in C(0,y) \quad d_{xo} \in C(x,0) \quad d(x,y,z) \in C(x',y') \;\; [(x,y) \in N \times N,\ z \in C(x,y)]}{R2\omega((a,b),\ d_{oy}\ ,\ d_{xo}\ ,\ (x,y,z)d(x,y,z)) \in C(a,b)}$$

And the equality rule says that :

1- If a = 0, then we have $d_{oy} \in C(0,b)$.

2- If b = 0, then we have $d_{xo} \in C(a,0)$.

3- Else we have a = k' and b = l' and $d(k',l',R2\omega(k,l,(x,y,z)d(x,y,z))) \in C(a,b)$

writing all cases in a only rule we have the desired simultaneous recursion rule :

$$\frac{(a,b) \in N \times N \quad d_{oy} \in C(0,y) \quad d_{xo} \in C(x,0) \quad d(x,y,z) \in C(x',y') \;\; [(x,y) \in N \times N,\ z \in C(x,y)]}{R2\omega((a,b),\ d_{oy}\ ,\ d_{xo}\ ,\ (x,y,z)d(x,y,z)) \in C(a,b)}$$

And the equality rule says that :

1- If a = 0, then we have $d_{oy} \in C(0,b)$.

2- If b = 0, then we have $d_{xo} \in C(a,0)$.

3- Else we have a = k' and b = l' and $d(k',l',R2\omega(k,l,(x,y,z)d(x,y,z))) \in C(a,b)$

III.4 - An Arithmetical problem.

Let's apply the methodology to the problem of comparing two natural numbers and finding the minor between them. Thus, we have a problem $P = \langle D, R, q \rangle$ where :

$D = \mathbb{N} \times \mathbb{N}$,

$R = \mathbb{N}$ and

$\alpha = \forall x \forall y \exists z ((x = z \vee y = z) \wedge z \leq x \wedge z \leq y)$ defines q.

It is clear that there are other formulas defining q. We only pick up one of them without any special reason for the above.

Following the steps of the methodology :

M1. We must describe $\mathbb{N}$ in ITT, but Martin-Lof did this in [Martin-Lof84]. Thus we only write down the natural numbers ITT description.

N-Formation

$$N \text{ set}$$

N-Introduction

$$0 \in N \qquad \frac{a \in N}{a' \in N}$$

, where a' denotes the successor.

The elimination rule has the meaning of the recursion (or induction) over a unique natural value.

N-Elimination

$$\frac{c \in N \qquad d \in C(0) \qquad \begin{array}{c} (x \in N,\ y \in C(x)) \\ | \\ e(x,y) \in C(x') \end{array}}{R(c,d,(x,y)e(x,y)) \in C(c)}$$

The operation R proceeds by evaluating c which will result in either 0 or a'. In the first case we follow by evaluating d resulting in an element of C(0). In the second case we substitute a for x and $R(a,d,(x,y)e(x,y))$ for y in e(x,y) and evaluate it. We can note that this behaves as a recursive evalution and indeed it is. The above explained is formalized by the following equality rules.

N-Equality

$$\frac{d \in C(0) \qquad \begin{array}{c}(x \in N,\ y \in C(x))\\ \vdots \\ e(x,y) \in C(x')\end{array}}{R(0,d,(x,y)e(x,y)) = d \in C(0)}$$

$$\frac{a \in N \qquad d \in C(0) \qquad \begin{array}{c}(x \in N,\ y \in C(x))\\ \vdots \\ e(x,y) \in C(x')\end{array}}{R(a',d,(x,y)e(x,y)) = e(a,d,R(a,d,(x,y)e(x;y))) \in C(a')}$$

An usefull fact is that if two natural numbers are equal then their respective sucessors are equal too.

$$\frac{a = b \in N}{a' = b' \in N}$$

M2-M3. Once time the set-theoretical parts of α are simple, we have to translate α itself into ITT. But, we must first give meaning in ITT to the concepts of equality (=) and the usual relation of order in $\mathbb{N}$ ($\leq$).

The equality between canonical elements given by the equality rules is not usefull for us, since it is a judgement rather than a propositional. Thus Martin-Lof defined the propositional equality, which we can relate to a judgement. This is done by building the type I(A,a,b) where A is a set and a and b are supposed to belong to A. I(A,a,b) has only one element which can be shown iff $a = b \in A$. This element is called r. Below we show the introduction and elimination rules to I. Note that this equality is not restricted to the natural numbers.

I-Formation

$$\frac{A \text{ set} \qquad a \in A \qquad b \in A}{I(A,a,b)\ \ Set}$$

I-Introduction

$$\frac{a = b \in A}{r \in I(A,a,b)}$$

I-Elimination

$$\frac{c \in I(A,a,b)}{a = b \in A}$$

I-Equality

$$\frac{c \in I(A,a,b)}{c = r \in I(A,a,b)}$$

We would like to point out that this concept of propositional equality is related to the problem of finding out whether two solutions are solutions to the same problem.

Regarding to the relation $\leq$, we define the type $M(x)$ built as indexed set of natural number. It is indexed by natural numbers too. The meaning is that $M(x)$ contains all natural numbers y such that $y \leq x$. But, this is done constructively. Below we show the rule that defines, to our immediate purpose, the sets $M(x)$.

M-Formation

$$\frac{a \in N}{M(a)\ Set}$$

M-Introduction

$$x \in M(x) \qquad \frac{b' \in M(a)}{b \in M(a)}$$

Besides, we need a rule about the transitivity of the relation $\leq$. This is performed by the following rule.

M-Transitivity

$$\frac{a \in M(b) \qquad b \in M(c)}{a \in M(c)}$$

An usefull fact is that zero belongs to all $M(a)$. This is proved as following.

$$\frac{a \in N \qquad 0 \in M(0) \qquad \dfrac{[0 \in M(x)]\quad \dfrac{x' \in M(x')}{x \in M(x')}}{0 \in M(x')}}{R(a,0,0) \in M(a)}$$

Thus, by using one of N-equality rules we get the desired $0 \in M(a')$.

Another usefull fact about M is that if a is less than b then a' is less than b'. We prefered to state it in a rule way :

$$\frac{a \in M(b)}{a' \in M(b')}$$

The reader can note that the definition of $M(x)$ is a case of the use of the language of ITT to define non-usual ITT-types. Thus in the methodology step of proving we add these rules to the

deductive system. It is clear that we could define these types only by using axioms but this seems too artificial for us.

Thus our desired formula translated into ITT results in α':

$$(\Pi x \in N)(\Pi y \in N)(\Sigma k \in N)((I(k,x,N) + I(k,y,N)) \wedge (M(x) \wedge M(y)))\ Prob$$

Where we remind that $(\Pi x \in A)B \cong A \wedge B$.

M4. Now we prove α' from $Ax = \{ N\ Prob,\ M(x)\ Prob\ (x \in A) \}$ using the inference rules of ITT plus the inference rules about M(x). We remind that we do not need to assume N x N Prob, where N x N ≅ (Πx ∈ N)N.

Well, in order to prove α', we can assume :

1- $x \in N$

2- $y \in N$

Doing $(I(k,x,N) + I(k,y,N)) \wedge (M(x) \wedge M(y)) \cong C(k,x,y)$. Let ONE be the following deduction.

$$\dfrac{\dfrac{\dfrac{0 = 0 \in N}{r \in I(0,0,N)}}{i(r) \in I(0,0,N) + I(0,y,N)} \qquad \dfrac{0 \in M(0) \quad 0 \in M(y)}{(0,0) \in M(0) \wedge M(y)}}{(i(r),(0,0)) \in (I(0,0,N) + I(0,y,N)) \wedge (M(0) \wedge M(y))}$$

$$\Big\downarrow$$

$$\dfrac{0 \in N}{(0,(i(r),(0,0))) \in (\Sigma k \in N)C(k,0,y)}$$

Where $0 \in M(b')$ is deduced using the M-Introduction rules and the rule of N-Elimination.

Let ONE' be the deduction of

$$(0,(j(r),(0,0))) \in (\Sigma k \in N)C(k,x,0)$$

, which is analagous to ONE.

Let TWO be the following deduction.

$$\dfrac{\dfrac{t \in (\Sigma k \in N)((I(k,a,N) + I(k,b,N)) \wedge M(a) \wedge M(b))}{Q(t) \in ((I(P(t),a,N) + I(P(t),b,N)) \wedge (M(a) \wedge M(b)))}}{P(Q(t)) \in I(P(t),a,N) + I(P(t),b,N)}$$

$$\dfrac{\dfrac{[x \in I(P(t),a,N)]}{P(t) = a \in N}}{\dfrac{P(t)' = a' \in N}{x \in I(P(t)',a',N)}} \qquad \dfrac{\dfrac{[y \in I(P(t),b,N)]}{P(t) = b \in N}}{\dfrac{P(t)' = b' \in N}{y \in I(P(t)',b',N)}}$$

$$\dfrac{i(x) \in I(P(t)',a',N) + I(P(t)',b',N) \qquad j(x) \in I(P(t)',a',N) + I(P(t)',b',N)}{D(P(Q(t)),\ (x)i(x),\ (y)j(y)) \in I(P(t)',a',N) + I(P(t)',b',N)}$$

Let THREE be the following deduction :

$$\dfrac{\dfrac{\dfrac{t \in (\Sigma k \in N)((I(k,a,N) + I(k,b,N)) \wedge M(a) \wedge M(b))}{Q(t) \in ((I(P(t),a,N) + I(P(t),b,N)) \wedge (M(a) \wedge M(b)))}}{Q(Q(t)) \in M(a) \wedge M(b)}}{\dfrac{P(Q(Q(t))) \in M(a)}{P(Q(Q(t)))' \in M(a')}}$$

Let THREE' be the following deduction :

$$\dfrac{\dfrac{\dfrac{t \in (\Sigma k \in N)((I(k,a,N) + I(k,b,N)) \wedge M(a) \wedge M(b))}{Q(t) \in ((I(P(t),a,N) + I(P(t),b,N)) \wedge (M(a) \wedge M(b)))}}{Q(Q(t)) \in M(a) \wedge M(b)}}{\dfrac{Q(Q(Q(t))) \in M(b)}{Q(Q(Q(t)))' \in M(b')}}$$

Let FOUR be the following deduction.

$$\dfrac{\dfrac{t \in (\Sigma k \in N)C(k,a,b) \quad \text{THREE}}{P(Q(Q(t)))' \in M(a')} \quad \dfrac{t \in (\Sigma k \in N)C(k,a,b) \quad \text{THREE'}}{Q(Q(Q(t)))' \in M(b')}}{(P(Q(Q(t)))', Q(Q(Q(t)))') \in M(a') \wedge M(b')} \quad \dfrac{t \in (\Sigma k \in N)C(k,a,b) \quad \text{TWO}}{D(P(Q(t)), (x)i(x), (y)j(y)) \in I(P(t)', a', N) + I(P(t)', b', N)}$$

$$(D(P(Q(t)), (x)i(x), (y)j(y)), (P(Q(Q(t)))', Q(Q(Q(t)))')) \in C(P(t)', a', b')$$

Doing

$$m(t) \cong (D(P(Q(t)), (x)i(x), (y)j(y)), (P(Q(Q(t)))', Q(Q(Q(t)))')).$$

Let FIVE be the following :

$$\dfrac{\dfrac{t \in (\Sigma k \in N)C(k,a,b)}{P(t) \in N}}{P(t)' \in N} \quad \dfrac{t \in (\Sigma k \in N)C(k,a,b) \quad \text{FOUR}}{m(t) \in C(P(t)', a', b')}$$

$$(P(t)', m(t)) \in (\Sigma k \in N)C(k,a',b')$$

Thus, doing $(0,(j(r),(0,0))) \cong d0y$, $(0,(i(r),(0,0))) \cong dx0$ and $(\Sigma k \in N)C(k,x,y) \cong C(x,y)$, we use the 2ω recursion showed previously and get :

$$\dfrac{x \in N \quad y \in N}{(x,y) \in N \times N} \quad \dfrac{}{d0y \in C(0,y)}\ \text{ONE} \qquad \dfrac{}{dx0 \in C(x,0)}\ \text{ONE'} \qquad \dfrac{[t \in (\Sigma k \in N)C(k,a,b)]}{(P(t)', m(t)) \in C(a',b')}\ \text{FIVE}$$

$$R2\omega(C(x\ y), d0y, d0x, (t)(P(t)', m(t))) \in C(x,y).$$

Thus, doing $S(x,y) \cong R2\omega(C(x,y),\ldots)$ we have $S(x,y) \in (\Sigma k \in N)C(k,x,y)$. Thus by using two sucessive applications of Π-Introduction we have as final conclusion $\lambda x \lambda y S(x,y) \in (\Pi x \in N)(\Pi y \in N)(\Sigma k \in N)C(k,x,y)$. So we proved our desired proposition.

With the sake of checking the solution, let's apply it to x=10 and y=1.

Thus, by calling PROOF the explained final deduction and using two succesive applications of the Π-Elimination and the correponding Π-equality rules we get a proof of S(10,1) ∈ (Σk ∈ N)C(k,10,1). That is :

P2ω(C10,1), dOy, dOx, (t)(P(t)', m(t))) ∈ (Σk ∈ N)C(k,10,1)

, and by using 2ω-Equality we get :

(P(R2ω(C(9,0),dOy,dxO,(t)(P(t)',m(t))))',
m(R2ω(C(9,0),dOy,dxO,(t)(P(t)',m(t))))) ∈ (Σk ∈ N)C(k,9,0)

, again, by 2ω-Equality we have :

(P((O,(j(r),(O,O))))', m((O,(j(r),(O,O))))) ∈ (Σk∈N)C(k,9,0)

Thus, evaluating by means of the Equality rules we get :

m((O,(j(r),(O,O)))) =
(D(P(Q((O,(j(r),(O,O)))))),(x)i(x),(y)j(y)),
(P(Q(Q((O,(j(r),(O,O)))))),Q(Q(Q((O,(j(r),(O,O))))))')) =
(j(r), (1,1))

So, finally we get (1, (j(r),(1,1))) ∈ (Σk ∈ N)C(k,10,1), from which we find that 1 is the desired number and (j(r),(1,1)) can be viewed as a proof of this.

We can note that S(x,y) performs the steps equivalently to the below program :

M(x,y) :

```
        IF x = 0 THEN 0 ELSE
                IF y = 0 THEN 0 ELSE
                        M(x,y) + 1.
```

Thus, we can say this program was in a certain way produced by the proposed methodology

IV. Conclusion.

As first conclusion, we can note that GPT is more general then ITT, since the last being intuitionistic cannot describe non-computable problems.

As a second and more interesting conclusion, we see that this work can be extended forward giving a model to ITT as being the GPT.

Finally, as a more practical conclusion and future work, we can develop the proposed methodology forward an automatic program generator. To complete this task we can see at least two subtasks (big problems !) :

1- To build an automatic proof procedure to ITT.

2- To find criterion to specify problems in GPT such that the corresponding formula in ITT correctly represent the problem. Remind that we could describe the $\leq$ in other way in our example.

About the first subtask we can say that it is closer a program generation than we like. About the second one we can, at least currently, say that it is more pragmatic than we like.

Then, our immediate research program is to restrict the class of problems thus solving the program generator to a determined scope. After we can extend the scope.

Finally we can observe that the language of the methodology does not restrict the class of problems to be solved by computable means (assuming Church Thesis), since it has power enough to descrive all the recursive functions.

Acknowledgements

I am grateful to Paulo Augusto S. Veloso for the initial idea and the discussions which were of great help in the development of the work. I want to thank Luiz Carlos Pereira for the discussions we had about ITT.

BIBLIOGRAPHY

[DIJSKSTRA76] Dijskstra, E. W. A discipline of programming, Prentice Hall, 1976.

[ENDLER86]Endler, Markus The reduction method and the decomposition of problems : Some aspects. Master Thesis, 1987, Depto de informática PUC/RJ.

[HEYTING31] Heyting, A. Die intuitionistische Grundlegung der Mathematik, Erkenntnis, vol. 2, 1931.

[KOLMOGOROV32] Kolmogorov, A. N. Zur Deutung der intuitionistischen Logik, Mathematische Zeitschrift, vol. 35, 1932.

[MARTIN-LOF84] Martin-Lof, Per Intuitionistic Type Theory, Bibliopolis, Edizioni di Filosofia e Scienze, 1984.

[PRAWITZ65] Prawitz, Dag Natural Deduction, Stockholm, 1965.

[RUBIN & RUBIN70] Rubin, H & Rubin, J Equivalents of the Axiom of Choice, Studies in Logic, North-Holland, 1970.

[VELOSO84] Veloso, P. A. S. Aspectos de uma Teoria Geral dos Problemas, Cadernos de filosofia e Historia da Ciencia, 1984.

[VELOSO84] Veloso, P. A. S. Outlines of a Mathematical Theory of general Problems, Philosophia Naturalis, vol.21 (#2-4) 1984.