Constructive Program Synthesis in Imperative Language using Intuitionist Logic and Natural Deduction

Geiza Maria Hamazaki da Silva
e-mail: hamazaki@inf.puc-rio.br

Edward Hermann Haeusler
e-mail: hermann@inf.puc-rio.br

Paulo A. S.Veloso
Systems and Comp. Engin. Prog.,COPPE and
Comp. Sci. Dept., Inst. Math., UFRJ-Brazil

PUC-RioInf.MCC18/02 July,2002

Abstract. We present a method to extract programs from constructive derivations, which is known as constructive synthesis or proof-as-program [2]. This method comes from the *Curry-Howard* isomorphism [11] and is based on the fact that a constructive proof for a theorem, which describes a problem, can be seen as a description of the solution of a problem, i.e., an algorithm [10,15]. In contrast with other constructive program synthesizers, in our work, the program (in an imperative language) is generated from a proof in many-sorted intuitionist logic using, as deductive system, the Natural Deduction. In addition, we provided a proof that the program generated is a representation of the solution for the specified problem by the theorem, in any theory that describes the data types used.

Keywords: constructive program synthesis, intuitionist logic, natural deduction and imperative language.

Resumo. O trabalho apresenta um método de extração de programas a partir de provas construtivas, denominado síntese construtiva de programas ou *proof-as-program* [2]. Esse método tem como base o isomorfismo *Curry-Howard* [11] e o fato de que uma prova construtiva para um teorema, que descreve um problema, pode ser vista como uma descrição para a solução do problema, i.e., um algoritmo [10,15]. Em contraste com outros processos de síntese construtiva de programas no nosso trabalho o programa (em uma linguagem imperativa) é gerado a partir de uma prova em lógica intuicionista poli-sortida utilizando a Dedução Natural como sistema dedutivo. Também é apresentado a prova de que o programa gerado é uma representação da solução do problema especificado pelo teorema, em uma teoria que descreve os tipos de dados utilizados.

Palavras chaves: síntese construtiva de programas, lógica intuicionista, dedução natural e linguagem imperativa.

Sponsered by CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico)

Constructive Program Synthesis in Imperative Language using Intuitionist Logic and Natural Deduction

SILVA,G.M.H* – HAEUSLER, E.H* – VELOSO, P.A.S+
*Dept. of Informatics, PUC-Rio, Brazil
+Systems and Comp. Engin. Prog.,COPPE and Comp. Sci. Dept., Inst. Math., UFRJ-Brazil
Email: hamazaki,hermann@inf.puc-rio.br

Abstract. We present a method to extract programs from constructive derivations, which is known as constructive synthesis or proof-as-program [2]. This method comes from the *Curry-Howard* isomorphism [11] and is based on the fact that a constructive proof for a theorem, which describes a problem, can be seen as a description of the solution of a problem, i.e., an algorithm [10,15]. In contrast with other constructive program synthesizers, in our work, the program (in an imperative language) is generated from a proof in many-sorted intuitionist logic using, as deductive system, the Natural Deduction. In addition, we provided a proof that the program generated is a representation of the solution for the specified problem by the theorem, in any theory that describes the data types used.

1- Introduction

Software development has to face two major problems: the cost of non-standard software - caused by the development times and the constant need for maintenance - and the lack of confidence in the reliability of software [13]. Many researchers are interested in providing techniques for developing reliable software, which is guaranteed to be correct and documented in a way that is easy to maintain and adapt. One of these research areas is called program synthesis, which proposes to generate automatically a correct program from specifications ([4,3,9,6,1]).

There are three basic categories of program synthesis: proof-as-program, transformational synthesis[1][14,7] and knowledge based program synthesis [13]. Some authors [14,7] insert another category called inductive program synthesis

Here, we deal with the proof-as-program paradigm [2], which avoids the double work of the software designing - the implementation of the system and the program verification - which can be seen as the same programming process in different degrees of formality. So this paradigm has focused on developing a program and its correctness proof at the same time [9,3].

This idea is based on the fact that: 1- Developing a program and prove that it is correct are just two aspects of the same problem [8]; 2- A proof for an application may be regarded as a (constructive) solution for a problem [17]; 3- A program can be extracted from a (constructive) proof of the existence of a solution for the corresponding problem [3].

Thus, using formal proofs - as a method for reasoning about the specification - and proving that the extraction process of a program preserves the proof's semantics, we get an

[1] Also called deductive program synthesis

automated way to construct, from a mathematical specification, a program that is correct by construction.

The specification of the problem and the deductive rules provide information about the algorithm structures and the reasoning about their properties. There are many formal logical calculi to represent it properly, e.g., ITT (*Intuitionist type theory*) and GPT (*General problem theory* [15,17,18]). As we use GPT we will give only a brief explanation about it.

The description of a problem in predicate logic can be viewed within the goal of GPT, since it is able to describe the input and output data as well as the relation between them. It considers problems as mathematical structures, where solutions can be precisely treated and provides a framework for studying some problem-solving situations, as well as problem solving. However, these pieces of information aren't enough to assure the existence of a method that solves the problem.

Besides the specification in predicate logic and given that the sentence that describes a problem is a *theorem* of the specification, if we obtain a constructive proof we will be able to understand it not only as a syntactic transcription, but also as a description of a given object, in other words, a description of an algorithm [10].

The *Curry-Howard* (C.H.) isomorphism associates the inference rules in natural deduction for intuitionist logic (used in the proof) with the formation rules of λ-terms (commands in a programming language), in such a way that one proof of σ (a formula) can be seen as a λ-term, which has the type σ. Hence, we can say that a proof has a computational interpretation, that is, it can be seen as a set of commands in a programming language, i.e., a program [11]. This isomorphism gives the base for the construction process of the program from a proof that is generally called "*extraction of computational contents of a proof*"[2]. This process extracts a function that relates the input with the specific outputs of the program. The inputs and outputs of the program reflect the application of inference rules used to obtain the formula. The computational contents relate to the *semi-computational contents* that describe the relations between the inputs and outputs of the program. The input and output variables of the program, by the C.H. isomorphism are represented, respectively, by the variables quantified by the universal quantifier and existential quantifier, so, the theorem of the specification must be of form $\forall x \exists y\, P(x,y)$.

There are many proposals for constructive programs synthesis, which use constructive logic - for instance, the ITT- to specify the problems. These systems use as deductive system the sequent calculus ([4,3,9]) or the rewrite mechanism [6], and construct programs in logical and functional programming languages.

Based upon those considerations, this work proposes a constructive synthesizer, where the program is generated from a proof using natural deduction, avoiding the conversion that is used in the related work found in the literature. In this method, a program will be constructed in an imperative program language (Pascal-like) from a proof in many-sorted predicated intuitionist logic. Using the concept of semi-computational contents of a formula, we prove that the generated program is a true representation for the solution to the specified problem by the theorem of any many-sorted theory that describes the data types used by the problem.

In the next section, we will present our constructive synthesis program process, which is composed by the labeling of memory configurations of the program, followed by the association of each inference rule with commands in the imperative language. In the section 3, will be described the proof of correctness of the program synthesis, i.e., a proof that the generated program achieve the specification. Section 4 has an example of our constructive

[2] For more on this concept, see section 2

synthesis mechanism and finally, in the section 5, we will present the conclusion of the work.

2- Program synthesis process

In the process of program synthesis we start from the existence of one theorem prover in many-sorted predicate intuitionist logic with arithmetic, which, beyond the usual inference rules, has inference rules for equality and induction. The theorem prover constructs a normal proof in natural deduction, for a certain theorem, which is the input to the programs synthesizer.

There are restrictions related to the inference rules used by the proof (given as an input to the synthesizer): 1– the proofs cannot have the negation introduction rule, 2– the existential elimination rule can be only applied on a formula that represents the inductive hypothesis. The last restriction[3] can be weakened if we admit parameterized programs as solutions.

From the proof of the specified problem we extract the computational content. In order to accomplish this, we first map all the memory configurations for each inference rule (labeling memory configuration process), and then we make the associate commands, in an imperative programming language, with each inference rule.

Labeling of memory configurations

We can view the execution of a program as a movement of bits inside the memory. Each movement represents an operation on the data of the program, which is stored in program's variables. Hence, it is very important to know the variables of the program and the values that may be attributed to them.

According to the operational reading of the connectives, given by the C.H. isomorphism, the variables quantified by universal quantifier are associated with the input data of the program and they are represented by the free variables since they can accept any value (of the same type of the variable) that make the formula true. The output data of the program are associated with the variables quantified by existential quantifier, which in a formula are represented by the free variables and the terms that depend on the input variables. Hence, the interpretation of the proof is based on the C.H. isomorphism, where each inference rule can be interpreted as a step in a program construction, the labeling of memory configurations process calculates the configuration of memory to each inference rule application.

The process of labeling memory configuration in a proof creates two sets: one for the input variables (free variables) and other to the output terms. In the beginning, both sets are empty. Next, the rules for labeling the input and output data will be used in the bottom-up direction. As the process reaches the proof tree leaves, we can find some variables and terms belonging to the set of free variables or to the set of output terms that will not be used as input and output of the program associated to the proof. These variables and terms reflect the memories data that are not used. They are considered residues of the labeling memory configuration process. These residues will be inserted in the set of input variables and output terms of the formulas that belong to the proof path derived from the formula, where they were detected for the first time (this process will be carried out in the top-down direction). Thus, the residues will be spread up to the proof tree root (conclusion) whose set of the input variables and the output terms will no longer be empty.

Labeling memory configurations rules

The labeling of memory configurations rules are related to the logical inference rules applications. In the presentation of the rules we will use the following notation: 1-α_T^{γ}, where

[3] These restrictions will become clearer in the sequel.

α is a formula, V the set of input variables and T the set of output terms; 2- K∪a represents the operation K∪{a}, where K is either a list of input variables or the list of output terms. The labeling rules below must be analyzed in the bottom-up direction, according to the labeling process[4].

Top-Formulae

axioms : β_T^V , where V and T are empty sets

Hypothesis: β_T^V , where V and T contains the variables of the program associate with the hypothesis

Universal quantifier elimination:

$$\frac{\forall y \alpha(y)_T^V}{\alpha(a)_T^{V \cup a}}$$

Universal quantifier introduction

$$\frac{\alpha(a)_T^{V \cup a}}{\forall y \alpha(y)_T^V}$$

Existential quantifier elimination

$$\frac{\exists y \alpha(y)_T^V \qquad \begin{array}{c}\alpha(a)_{T \cup a}^V \\ \vdots \\ \delta_{T'}^{V'}\end{array}}{\delta_{T'}^{V'}}$$

Existential quantifier introduction

$$\frac{\alpha(b)_{T \cup b}^V}{\exists y \alpha(y)_T^V}$$

Conjunction elimination

$$\frac{(\alpha \wedge \beta)_T^V}{\alpha_T^V} \qquad \frac{(\alpha \wedge \beta)_T^V}{\beta_T^V}$$

Conjunction Introduction

$$\frac{\alpha_T^V \qquad \beta_T^V}{(\alpha \wedge \beta)_T^V}$$

Disjunction elimination

$$\frac{(\alpha \vee \beta)_{T'}^{V'} \qquad \begin{array}{cc}[\alpha]_{T'}^{V'} & [\beta]_{T'}^{V'} \\ \vdots & \vdots \\ \gamma_T^V & \gamma_T^V\end{array}}{\gamma_T^V}$$

Disjunction introduction

$$\frac{\alpha_T^V}{(\alpha \vee \beta)_T^V} \qquad \frac{\beta_T^V}{(\alpha \vee \beta)_T^V}$$

Implication elimination

$$\frac{\alpha_{T'}^{V'} \qquad (\alpha \rightarrow \beta)_T^V}{\beta_T^V}$$

Implication introduction

$$\frac{\begin{array}{c}[\alpha]_{T'}^{V'} \\ \vdots \\ \beta_T^V\end{array}}{(\alpha \rightarrow \beta)_T^V}$$

Induction

$$\frac{\begin{array}{ccc}[k \prec l]_{T'}^{V'} & [\alpha(a_i)]_{T_1}^{V_1} & \\ \vdots & \vdots & \\ \alpha(k)_T^{V \cup k} & \alpha(w)_T^{V \cup w} & a_i \prec w_{T_2}^{V_2}\end{array}}{\forall y \alpha(y)_T^V}$$

Where k reflects the term associated to the base case and, w reflects the tem associated to the inductive case.

Intuitionist absurdity rule

$$\frac{\dfrac{\beta_{T_1}^{V_1} \qquad \neg\beta_{T_2}^{V_2}}{\perp_{\{\}}^{\{\}}}}{\alpha_{\{\}}^{\{\}}}$$

Remark*:* In the labeling of memory configurations process, if a proof uses the equality congruency property, the resultant formulas of this rule utilization will have the set of the input variables and the output variables changed in the following way: the substituted variables or terms will be taken out from the set[5] where they belong, and they must be added to the terms and variables, on which the replaced terms depend, in their respective set[6].

Association of inference rules with instructions of an imperative language

In the program generation process each formula is related to a program that results from the association of inference rules with commands in a given programming language.

[4] More details in [19].
[5] Output terms and input variables set.
[6] An example of the application of this rule in the section 4.

Each association is based on the content (*logical* or *semi-computational*) of the formula, in such a way that a program will reflect the semantics of the proof of the formula associated.

A formula has *logic content* when it is derived from an axiom or hypothesis and describes the nature of the objects used by the program to solve the problem proposed, i.e., it describes the data structures of a program and the set of operations that can be applied to them.

The *semi-computational content* of a formula is the set of information that express the relations between the input and the output data of a program – which is a solution for the problem specified by the formula – where for more than one input we can have one or more outputs.

The *computational content* of a formula is a function, within the semi-computational contents, which relates the input with the specific outputs of the program. They reflect the application of inference rules used to obtain the formula.

We use the following notation to describe the generation program rules: 1- $\Lambda : \alpha_T^V$ - where, Λ is a program that calculates the property described in α_T^V; 2- Λ,Ψ,Γ – programs; 3- σ – description of the memory allocation; 4- p – name of the program related to the formula.
Remark: The commands of the language, in which the program will be produced, have the same semantics of the equivalent commands in the usual imperative programming languages.

Top-Formulae

Axioms : $\sigma : \beta_T^V$ **,** where σ indicates "logical contents". **Hypothesis:** $\mathbf{p} : \delta_T^V$, where **p** is a symbol for programs

Universal quantifier elimination:

1.Axioms and non-inductive hypothesis: $\dfrac{\sigma : \forall y \alpha(y)_T^V}{\sigma : \alpha(a)_T^{V \cup a}}$ **2. Inductive hypothesis:** $\dfrac{p : \forall y \alpha(y)_T^V}{p : \alpha(a)_T^{V \cup a}}$

Universal quantifier introduction: $\dfrac{\Lambda : \alpha(a)_T^{V \cup a}}{read(a); \Lambda : \forall y \alpha(y)_T^V}$

Existential quantifier elimination: $\dfrac{p : \exists y \alpha(y)_T^V \qquad \begin{array}{c} a \leftarrow exec(p, \bar{v}) : [\alpha(a)_{T \cup a}^V] \\ \vdots \\ \Gamma : \delta_{T'}^{V'} \end{array}}{\Gamma : \delta_{T'}^{V'}}$

The command *exec*(…) gives to the program input variables the values passed by the parameters. Besides, it realizes the calling of the function and returns the last output term after the execution of ***p***.

Existential quantifier introduction: $\dfrac{\Lambda : \alpha(b)_T^{V \cup b}}{\Lambda; write \quad (b) : \exists y \alpha(y)_T^V}$

Conjunction elimination: $\dfrac{\Lambda : (\alpha \wedge \beta)_T^V}{\Lambda : \alpha_T^V} \qquad \dfrac{\Lambda : (\alpha \wedge \beta)_T^V}{\Lambda : \beta_T^V}$

Conjunction Introduction: $\dfrac{\Lambda : \alpha_T^V \qquad \Psi : \beta_T^V}{\Lambda \otimes \Psi : (\alpha \wedge \beta)_T^V}$

Disjunction elimination: $\dfrac{\sigma : (\alpha \vee \beta)_{T'}^{V'} \qquad \begin{array}{cc} \sigma : [\alpha]_{T'}^{V'} & \sigma : [\beta]_{T'}^{V'} \\ \vdots & \vdots \\ \Lambda : \gamma_T^V & \Psi : \gamma_T^V \end{array}}{if \ (\alpha) \ then \ (\Lambda) \ else \ (if \ (\beta) \ then \ (\Psi)) : \gamma_T^V}$

Disjunction introduction: $\dfrac{\Psi : \alpha_T^V}{\Psi : (\alpha \vee \beta)_T^V} \qquad \dfrac{\Lambda : \beta_{T'}^{V'}}{\Lambda : (\alpha \vee \beta)_{T'}^{V'}}$

Implication elimination: The assertion associated to the implication elimination rule is: $\dfrac{\Psi : \alpha_T^V \; \Lambda : (\alpha \to \beta)_{T'}^{V'}}{[\Lambda \leftarrow \{exec(p, \vec{v}) = \Psi\}] : (\alpha \to \beta)_{T'}^{V'}}$, where [Λ⇐{exec([p],v) = Ψ}], denotes the substitution of the supposed procedure call (p) by the real procedure call (Ψ) in the program(Λ). According to the proof restrictions (seen below), the minor premise of implication elimination rule always has logic contents. Thus, the program to be generated by the application of this rule is the program associated to the major premise. So, we have the following assertion: $\dfrac{\sigma : \alpha_T^V \quad \Lambda : (\alpha \to \beta)_{T'}^{V'}}{\Lambda : \beta_{T'}^{V'}}$

Implication introduction: The assertion related to this rule is: $\dfrac{\begin{array}{c} p : [\alpha]_{T'}^{V'} \\ \vdots \\ \Lambda : \beta_T^V \end{array}}{Dec\, p \quad \Lambda : (\alpha \to \beta)_T^V}$, where *Dec* is a label to mark the utilization of procedure p. However, according to the proof's restrictions, the hypothesis used in the proof process has only logic content. Thus, we have the following assertion: $\dfrac{\begin{array}{c} \sigma : [\alpha]_{T'}^{V'} \\ \vdots \\ \Lambda : \beta_T^V \end{array}}{\Lambda : (\alpha \to \beta)_T^V}$

***Intuitionist absurdity rule*:** $\dfrac{\begin{array}{c} \sigma : \beta_T^V \\ \vdots \\ \perp_{\{\}}^{\{\}} \end{array}}{\sigma : \gamma_{\{\}}^{\{\}}}$

***Induction*:** This rule is an alternative form of the application for the introduction of the universal quantifier. Consequently, the program generated will have the command related to the application of this rule (*read*(…)) and a recursive program formed by a conditional command

$$\dfrac{\begin{array}{ccc} [z \prec l\ ]_{T'}^{V'} & [p : \alpha(a_i)_{T_1}^{V_1}\ ] & \\ \vdots & \vdots & \vdots \\ \Psi : \alpha(z)_T^{V \cup z} & \Lambda : \alpha(k)_T^{V \cup k} & a_i \prec k_{T_2}^{V_2} \end{array}}{\begin{array}{l} \text{Procedure Rec } \{\ \text{read(y);} \\ \quad \text{if } (y < l) \text{ then } \{\Psi\} \\ \quad \text{else} \\ \quad \{\ [\Lambda \leftarrow \{\text{exec}([p], \vec{v}) = \text{Rec}\}^*]\ \} \\ \} \end{array} \quad : \forall y \alpha(y)_T^V}$$

Observations: 1- If this rule will be the last rule to be applied on the proofs process, instead of *Procedure*, this program will be declared as *Program*; 2- In the assertion above, the * means the command related to the renaming of a hypothetical program ($\text{exec}([p], \vec{v})$) by the recursive call (**Rec**) in the program (Λ).

Remarks about introduction of implication rule: 1- the restriction that the minor premise always has logic contents is due to the fact that if it would have a program associated to it, the calling point in the program related to the major premise should be changed. As we are extracting programs from normal proofs, associated to the major premise, we only have the name of the program, in such a way that, we do not know in which place we have to change the procedure label by the procedure calling; 2- Due to the restrictions on the proof structure, the minor premise can be derived from the inductive hypothesis, which has a program associated. After the execution of the associated program we will have the configuration of

memory for the execution of the program related to the major premise. Thus, we can interpret the program related to the minor premise as logic contents, satisfying the restriction of the implication elimination rule.

Restrictions about the rules used in the proof

The following rules will not be used because the extraction of their computational contents is not ordinary: **1- Negation Introduction rule-** This rule is directly related to the rule of the intuitionist absurdity, which according to the extraction of semi-computational contents, expresses that the allocation of memories is empty. So, we have empty memory allocation associated with the premise of the negation introduction rule. However there are examples that show that there is a semi-computational content for this rule, see[19]. Hence, this rule has to be better studied. **2- Existential quantifier elimination -** When we extract the semi-computational contents of a proof with this rule, we consider that there is a program associated to the major premise, which will be referenced through the label (*exec*) during the extraction process of the semi-computational contents (it will be changed by the program call). In the case of inductive hypothesis, the program related to the existential quantifier is the same program to be constructed. So, we will know what the program call that will substitute the label. Otherwise the program referred to is only hypothetical, and we will not know the shape of its call; thus, the user would be in charge of giving this information to the program.

Semi-computational contents (SCC)

In the program synthesis process, we extract the computational contents of the formula, but to make easier the proof of correctness, we use the semi-computational contents of the formula (SCC).

In the definitions about of *SCC* that will be presented: θ - is a set of formulae; σ - expresses a configuration of memory, attributions of values to variables over which certain properties are true; Γ- is the set of axioms that describes the theory, where the solution of the problem (proof's conclusion) is represented; $a\widehat{\ }C$ - expresses the concatenation of the element *a* with the object C; and $C\widehat{\ }a$ - expresses the concatenation of the object C with the element *a*.

Below we present some examples of the definition of $SCC^{\theta}_{M}(\alpha^{V}_{T})$, according to the structure α:

(i) Atomic formula:

$$SCC^{\theta}_{M}(\mathrm{A}^{V}_{T}) = \left\{ \left\langle nil, \left\langle \vec{b}, \vec{o} \right\rangle, nil \right\rangle \middle/ \begin{array}{l} \forall v_i \in V, \forall b_i \in \vec{b} \text{ e } \forall \sigma, (\sigma(v_i) = b_i) \\ \text{and} \\ \forall t_i \in T \text{ e } \forall o_i \in \vec{o}, [[t_i]]_{\sigma} = o_i \\ \text{and} \\ \textit{If } M \underset{H,\sigma}{\models} \theta \textit{ then } M \underset{H,\sigma}{\models} \mathrm{A}^{V}_{T} \end{array} \right\}$$

(ii) Universal Quantifier (∀)

$$SCC^{\theta}_{M}(\forall x \alpha(x)^{V}_{T}) = \left\{ \forall b_i \left(\left\langle b_i \widehat{\ } L, \left\langle \vec{b}, \vec{o} \right\rangle, W \right\rangle \middle/ \left\langle L, \left\langle b_i \widehat{\ } \vec{b}, \vec{o} \right\rangle, W \right\rangle \in SCC^{\theta}_{M}\left(\alpha(h)^{V \cup h}_{T}\right) \right) \right\}$$

(iii) Existential Quantifier (∃)

$$SCC_M^{\theta}(\exists x\alpha(x)_T^V) = \left\{ \left\langle L, \left\langle \vec{b}, \vec{o} \right\rangle, W \overset{\frown}{} o_i \right\rangle \Big/ \left\langle L, \left\langle \vec{b}, \vec{o} \overset{\frown}{} o_i \right\rangle, W \right\rangle \in SCC_M^{\theta}(\alpha(h)_{T\cup h}^V) \right\}$$

(v) Disjunction (∨)

$$SCC_M^{\theta}((\alpha \vee \beta)_T^V) = SCC_M^{\theta}(\alpha_T^V) \bigcup SCC_M^{\theta}(\beta_T^V)$$

(iv) Absurd Intuitionist (⊥)

$$SCC_M^{\theta}(\perp_T^V) = \{\ \}$$

Remark: The definition of SCC of conjunction and implication are not presented because they have many details, for example, in the introduction of a conjunction we have a program to the left side and other to the right side, we know that the conjunction connective is commutative, but in a program in the most of times the order is important.

The following definitions will be used in the correctness proof of the synthesis process, and it shows that for all the input data we have the same values in the output variables and the same output data.

Definition 1 - $U \subseteq \mathrm{SCC}^{\theta}{}_M(\alpha^V{}_T)$, U is complete if and only if :

$\forall k_1 \in U$, such that $k_1 = \left\langle L, \left\langle \vec{b}, \vec{o} \right\rangle, W \right\rangle$ and $\forall k_2 \in U$, such that $k_2 = \left\langle L', \left\langle \vec{b}, \vec{o}' \right\rangle, W' \right\rangle$

we have that : $L = L', \vec{o} = \vec{o}'$ e $W = W'$

Definition 2 – Let: $k =< L, < \vec{b}, \vec{o} >, W >$, where $\vec{b}, \vec{o} \in M$ (Herbrand's model).

We have Λ (is a program) that calculates *k* if and only if:

$$\underset{\text{Hoare}}{\vdash} \{in = L \wedge (v_1 = b_1, ..., v_n = b_n)\} \Lambda \{(t_1 = o_1, ..., t_m = o_m) \wedge out = W\}$$

where $V = \{v_1, \ldots, v_n\}$ and $T = \{t_1, \ldots, t_m\}$.

$\forall k_1 \in U$, such that $k_1 = \left\langle L, \left\langle \vec{b}, \vec{o} \right\rangle, W \right\rangle$ and $\forall k_2 \in U$, such that $k_2 = \left\langle L', \left\langle \vec{b}, \vec{o}' \right\rangle, W' \right\rangle$

we have that : $L = L', \vec{o} = \vec{o}'$ and $W = W'$

Definition 3 - $\theta \underset{\mathrm{M},\sigma}{\models} \Lambda{:}(\alpha^{\mathrm{V}}{}_{\mathrm{T}})$ if and only if:

$$\exists U \text{ complete} \subseteq CSC_M^{\theta}\left(\alpha_T^V\right), \text{ such that, } \forall u \in U, \text{ the program } \Lambda \text{ calculates } u.$$

Considering that M is a *Herbrand* model (H) [5], that is, a structure concerning *Herbrand* universe [5] for the language $L(\Gamma)$, such that:

$$M \underset{H}{\models} \Gamma_T^V$$

Obs: The Herbrand's model to the labeled formulae is the same to the no labelled formulae.

3 – Correctness proof of the synthesis process

The program synthesis process is composed of two parts: the labeling of memory configuration of the proof's formulas (*LabelMemoConfig*) and the extraction of computational contents of the proof (*GenProg*).

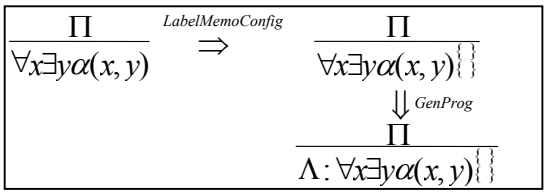

Figure2 – Schema of the program construction

With this method, we obtain a program that has the same semantics of the proof, and to guarantee it, we must to proof the correctness of the system, which is achieved by the proof of the theorem that use following lemma:

Lemma: Let Δ be a set of formulas, M a *Herbrand´*s model for and Π a proof of the formula α: $\dfrac{\Delta}{\Pi}{\scriptstyle \alpha}$ and Π'= *GenProg*(*MarkVarTerms*(Π)). Let $(\Pi_1 \prec \Pi)$[7], then:

a) A subset SCC' of SCC is said complete if and only if all $\rho \in$ SCC' have the same files with the value list of the input, and the same value list of the output in the memory and the same files with the output values; and

b) $\theta \models_{M,\sigma} \Lambda : \alpha_T^V$ if and only if there is a SCC' complete equal or within to SCC of a formula, such that, for all $\rho \in$ SCC', the program Λ realizes the transformation of the input and output data expressed by ρ.

Proof: The proof of the lemma was carried out by induction in the length of the proof, through the comparison of the changed syntactic semantics – of the program – with the semi-computational contents of the inference rules.

Part a) - It 's guaranteed by the intuitionist calculus correction.

Part b)

Base case: Let Δ a set of axioms

1-Axioms

According to the extraction of semi-computational contents related to axioms, they express logic contents. By the lemma assumptions, *M* is the *Herbrand* 's model for them.

$$\models_M \sigma : \Delta$$

[7] $\Pi_1 \prec \Pi$ express that Π_1 is a derivation contained in Π.

2- Hypothesis (not axioms)

a) Hypothesis that has logical contents – From part "a" of the lemma we have:

$$\beta_i \models_{M,\sigma} \sigma : (\beta_i)_T^V$$

b) Hypothesis that has semi-computational contents (inductive hypothesis)

According to the lemma hypothesis, these are correct by construction, so:

$$\Delta \models_{M,\sigma} p_i : \delta$$

Inductive case[8]:

Observation: In the proofs that will be presented, we will use a notation abuse: δ_i instead of $p_i{:}\delta_i$.

- *Universal quantifier introduction*

Suppose that the rule of the quantifier introduction is the last rule applied on a derivation D: $D = \left\{ \begin{array}{c} \Delta, \beta_1, ..., \beta_n, \delta_1, ..., \delta_k \\ \vdots \\ \Lambda : \alpha(h)_T^{V \cup h} \\ \hline read(h); \Lambda : \forall y \alpha(y)_T^V \end{array} \right.$

By the inductive hypothesis: $\Delta, \beta_1, \ldots \beta_n, \delta_1, \ldots, \delta_k \models_{M,\sigma} \Lambda : \alpha(h)_T^{V \cup h}$ if and only if:

$\exists U$ complete $\subseteq SCC_M^{\theta}\left(\alpha(h)_T^{V \cup h}\right)$, such that $\forall k \in U$ being $k =< L, < h \overset{\frown}{} \vec{b}, \vec{o} >, W >$,

we have that : $\vdash_{Hoare} \{\text{in} = \text{L} \wedge (\vec{v} = \vec{b}) \wedge (h = b_i)\} \Lambda \{(\vec{t} = \vec{o}) \wedge \text{out} = \text{W}\}$ where $b_i \in \vec{b}$ and $i = 1...n$

The result of the composition of the command *read*(…) with the program has the following semantic rule:

(1)

$$\frac{\left\{in = \left\langle b_i \right\rangle L \wedge (\vec{v} = \vec{b}) \wedge b_i = b_i \right\} \; read(h) \; \left\{ in = L \wedge (\vec{v} = \vec{b}) \wedge h = b_i \right\} \; \left\{ in = L \wedge (\vec{v} = \vec{b}) \wedge (h = b_i) \right\} \Lambda \left\{ (\vec{t} = \vec{o}) \wedge out = W \right\}}{\left\{ in = \left\langle b_i \right\rangle L \wedge (\vec{v} = \vec{b}) \wedge b_i = b_i \right\} \; read(h); \Lambda \left\{ (\vec{t} = \vec{o}) \wedge out = W \right\}}$$

[8] The semantics of the commands rules used are alike presented in [HW72].

Given that:

$$SCC_M^{\theta}(\forall x\alpha(x)_T^V) = \left\{ \left\langle b_i \frown L, \langle \vec{b}, \vec{o} \rangle, W \right\rangle \middle/ \forall b_i \in M \left(\left\langle L, \left\langle b_i \frown \vec{b}, \vec{o} \right\rangle, W \right\rangle \in SCC_M^{\theta}\left(\alpha(h)_T^{V \cup h}\right) \right) \right\}$$

$$\text{If } U' \subseteq \mathrm{SCC}_M^{\theta}(\forall y\alpha(y)_T^V), \text{ then : } U' = \left\{ \left\langle b_i \frown L, \langle \vec{b}, \vec{o} \rangle, W \right\rangle \middle/ \left\langle L, \left\langle b_i \frown \vec{b}, \vec{o} \right\rangle, W \right\rangle \in U \right\}$$

As *U'* is formed from *U*, we have that *U'* is complete.

From the inductive hypothesis, the program Λ calculates K_2.

$$\text{Let } K_1 \in U', K_1 = \left\langle b_i \frown L, \langle \vec{b}, \vec{o} \rangle, W \right\rangle, \text{such that : } K_2 = \left\langle L, \left\langle b_i \frown \vec{b}, \vec{o} \right\rangle, W \right\rangle \in (U \subseteq SCC_M^{\theta}(\alpha(h)_T^{V \cup h}))$$

Thus, from (1), we have that (*read*(*h*);Λ) calculates K_1. Let K_1 being as arbitrary triple that belongs to *U'*, we can conclude that: $\Delta, \beta_1, ..., \beta_n, \delta_1, ..., \delta_k \models_{M,\sigma} read(...); \Lambda : \forall y\alpha(y)_T^V$

- *Existential quantifier elimination*

Suppose that the elimination rule of ∃ is the last rule to be applied on a derivation D:

$$D = \left\{ \dfrac{\begin{array}{cc} \Delta, \beta_1, ..., \beta_n, \delta_1, ..., \delta_k & a \leftarrow exec(\Lambda, \vec{v}) : [\alpha(a)_{T \cup a}^V] \\ \vdots & \vdots \\ \Lambda : \exists y\alpha(y)_T^V & \mathrm{T} : \gamma_{T'}^{V'} \end{array}}{\mathrm{T} : \gamma_{T'}^{V'}} \right.,$$

This by the inductive hypothesis:

1- $\Delta, \beta_1, \ldots, \beta_n, \delta_1, \ldots, \delta_k \models_{M,\sigma} \Lambda : \exists y\alpha(y)^{V}{}_{T}$, if and only if:

$$\exists U_1 \text{ complete} \subseteq SCC_M^{\theta}\left(\exists y\alpha(y)_T^V\right), \text{such that } \forall k \in U \text{ being } k = < L, < \vec{b}, \vec{o} >, W \frown o_i >,$$

$$\text{we have that : } \vdash_{\text{Hoare}} \left\{ \mathrm{in} = \mathrm{L} \wedge (\vec{v} = \vec{b}) \right\} \Lambda \left\{ (\vec{\mathrm{t}} = \vec{o}) \wedge \mathrm{out} = \mathrm{W}\langle o_i \rangle \right\}, \text{where } o_i \in \vec{o} \text{ and } i = 1...n$$

2- $\Delta,\beta_1,\ldots\ \beta_n,\ \delta_1\ ,\ \ldots,\ \delta_k \models_{M,\sigma} T:\ \gamma^{V'}_{T'}$, if and only if:

$$\exists U_2 \text{ complete} \subseteq SCC^{\theta}_{M}\left(\gamma^{V'}_{T'}\right), \text{such that } \forall k \in U_2 \text{ being } k =< L,< \vec{b},\vec{o} >, W >,$$

$$\text{we have that } \vdash_{\text{Floare}} \left\{\text{in} = \text{L} \wedge (\vec{v} = \vec{b})\right\} \text{T} \left\{(\vec{\text{t}} = \vec{o}) \wedge \text{out} = \text{W}\right\}, \text{where } o_i \in \vec{o} \text{ and } i = 1...n$$

In the inductive hypothesis 2 we have that δ_j is associated to the following program: $a \leftarrow exec(\Lambda, \vec{v})$.

The command *exec*(…) has the following semantic rule:

$$\frac{\{in = L \wedge (\vec{v} = \vec{o})\}\ \Lambda\ \left\{(\vec{t} = \vec{o}) \wedge out = W\left\langle o_i \right\rangle\right\}}{\{in = L \wedge (\vec{v} = \vec{o})\}\ Z \leftarrow exec(\Lambda, \vec{v})\ \left\{(Z = o_i) \wedge (\vec{t} = \vec{o}) \wedge out = W\right\}}$$

Given that:

$$SCC^{\theta}_{M}(\exists y \alpha(y)^{V}_{T}) = \left\{\left\langle L, \left\langle \vec{b}, \vec{o} \right\rangle, W \overset{\frown}{} o_i \right\rangle \middle/ \left\langle L, \left\langle \vec{b}, \vec{o} \overset{\frown}{} o_i \right\rangle, W \right\rangle \in SCC^{\theta}_{M}(\alpha(h)^{V}_{T \cup h})\right\}$$

From the inductive hypothesis (2):

$\left\langle L, \left\langle \vec{b}, \vec{o} \right\rangle, W \right\rangle \in U_2$ from the supposition that $b_i \leftarrow exec(\Lambda, \vec{v})$, such that $b_i \in \vec{b}$.

From the inductive hypothesis (1) the program Λ calculate o $SCC^{\theta}_{M}(\exists y \alpha(y))$, that is described by the triple : $k = \left\langle L, \left\langle \vec{b}, \vec{o} \right\rangle, W \overset{\frown}{} o_i \right\rangle$.

We can observe based upon the semantic of the command *exec*(…) that it returns the last term written in the output file, then, $exec(\Lambda, V) = o_i^{\Lambda}$.

Thus, $SCC^{\theta}_{M}(\gamma^{V'}_{T'}) = \left\langle L, \left\langle \vec{b}, \vec{o} \right\rangle, W \right\rangle$, where $b_i = o_i^{\Lambda}$ such that : $b_i \in \vec{b}$ e $o_i^{\Lambda} \in U_1$.

So, let *k* being an arbitrary triple that belongs to U_1, we can conclude that:

$\Delta,\beta_1,\ldots\ \beta_n,\ \delta_1\ ,\ \ldots,\ \delta_k \models_{M,\sigma} T:\ \gamma^{V'}_{T'}$

- *Induction rule*

Suppose that the rule of induction inference is the last rule applied on the derivation D:

$$D = \left\{ \begin{array}{l} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Delta, \beta_1,...,\beta_n, \delta_1,...,\delta_k, [p : \alpha(a)_T^{V \cup a}] \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad p : \exists x \beta(x)_{T''}^{V''} \\ \Delta, \beta_1,...,\beta_n, \delta_1,...,\delta_k, [b \prec l]_{T'}^{V'} \qquad\qquad\qquad\qquad q \leftarrow exec(p, \vec{v}) : \beta(q)_{T''}^{V''} \\ \qquad\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ \qquad \Lambda : \alpha(b)_T^{V \cup b} \qquad\qquad\qquad\qquad\qquad \Psi : \alpha(c)_T^{V \cup c} \qquad\qquad\qquad\qquad a \prec c_T^V \\ \hline \qquad \text{Procedure Rec } \{ \quad \text{read}(x); \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad : \forall y \alpha(y)_T^V \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \end{array} \right.$$

$$\begin{array}{l} \text{if } (x < l) \text{ then } \{\Lambda\} \\ \text{else} \\ \{ \ [\Psi \Leftarrow \{\text{exec}(p, \bar{v}) = \text{Rec}\}^*]\} \\ \} \end{array}$$

where : β is a subformula of $\alpha(a)$, b is the term related to the base case, c is the term related with the inductive case, x is an input variable that will be equal to b or to c, and "*" represents the substitution of the hypothetical program by its recursive call.

By the inductive hypothesis:

1- $\Delta, \beta_1,...,\beta_n, \delta_1,...,\delta_k, y \prec 1 \models_{M,\sigma} \Lambda : \alpha(b)_T^{V \cup b}$ if and only if:

$\exists U_1$ complete $\subseteq SCC_M^{\theta \cup (y \prec l)}\left(\alpha(b)_T^{V \cup b}\right)$, such that $(\Delta, \beta_1,...,\beta_n, \delta_1,...,\delta_k) = \theta$ and $\forall k \in U_1$ being $k =< L, < \overset{\frown}{b}\ \vec{b}, \vec{o} >, W >$, we have that $\vdash_{\text{Hoare}} \left\{ in = L \wedge (\vec{v} = \vec{b}) \wedge b = b_i \right\} \Lambda \left\{ (\vec{t} = \vec{o}) \wedge out = W \right\}$ where $b_i \in \vec{b}$ and $i = 1..n$

2- $\Delta, \beta_1,...,\beta_n, \delta_1,...,\delta_k, p{:}\alpha(a) \models_{M,\sigma} \Psi : \alpha(c)_T^{V \cup c}$ if and only if:

$\exists U_2$ complete $\subseteq CSC_M^{\theta \cup \alpha(a)}\left(\alpha(c)_T^{V \cup c}\right)$, such that $(\Delta, \beta_1,...,\beta_n, \delta_1,...,\delta_k) = \theta$ e $\forall k \in U_2$ being $k =< L, < \overset{\frown}{c}\ \vec{b}, \vec{o} >, W >$, we have that $\vdash_{\text{Hoare}} \left\{ in = L \wedge (\vec{v} = \vec{b}) \wedge c = b_i \right\} \Psi \left\{ (\vec{t} = \vec{o}) \wedge out = W \right\}$ where $b_i \in \vec{b}$ e $i = 1..n$

By the derivation of D, we can note that the conclusions $\alpha(b)$ and $\alpha(c)$ are proved from the exclusives hypothesis, thus we are going to associate a conditional command to these conclusions.

Thus, the generated command will possess the following semantics:

$$\frac{\left\{in = L \wedge (\vec{v} = \vec{b}) \wedge (b \prec l) \wedge b = b_i\right\} \Lambda \ \{(\vec{t} = \vec{o}) \wedge out = W\} \quad \left\{in = L \wedge (\vec{v} = \vec{b}) \wedge \neg(y \prec l) \wedge c = b_i\right\} \ \Psi \ \{(\vec{t} = \vec{o}) \wedge out = W\}}{\left\{in = L \wedge (\vec{v} = \vec{b}) \wedge x = b_i\right\} \text{if } (x \prec l) \text{ then } \{\Lambda\} \text{else } \{\Psi\} \ \{(\vec{t} = \vec{o}) \wedge out = W\}}$$

where x = c (when x≥l) or x= b (where x<l).

By the inductive hypothesis, from the proof of $\alpha(b)$ with the hypothesis $(b \prec l)$ we can extract the $SCC_M^{\theta \cup (y \prec l)}(\alpha(b)_T^{V \cup b})$, and from the proof of $\alpha(c)$ with $\alpha(a)$ we can extract the $SCC_M^{\theta \cup \alpha(a)}(\alpha(c)_T^{V \cup c})$. Since $(b \prec l)$ and $\alpha(a)$ are exclusive hypothesis, we have that $SCC_M^{\theta \cup (y \prec l)}(\alpha(b)_T^{V \cup b}) \cup SCC_M^{\theta \cup \alpha(a)}(\alpha(c)_T^{V \cup c}) = SCC_M^{\theta}(\alpha(x)_T^{V \cup x})$.

So, if $U' \subseteq SCC_M^{\theta}(\alpha_T^V)$, then : $U' = U_1 \cup U_2$.

Since *U'* is formed from U_1 and U_2, we have that *U'* is complete, because the list of input values of U_1 (which is related to the elements less than *l*) and of U_2 (which is related to the elements greater or equal than *l*) are disjoints.

Therefore:

(1) $\Delta, \beta_1, \ldots \beta_n, \delta_1, \ldots, \delta_k \models_{M,\sigma} \mathit{If}\ (x \prec l)\ \mathit{then}\ \{\Lambda\}\ \mathit{else}\ \{\Psi\}: \alpha(x)^{V \cup x}{}_T$

The result of the composition of the command *read*(…) with the conditional command, which was generated above, has the following semantic rule :

(2)

$$\frac{\left\{in = \left\langle b_i \right\rangle L \wedge (\vec{v} = \vec{b}) \wedge b_i = b_i\right\} \ \mathit{read}(x) \ \left\{in = L \wedge (\vec{v} = \vec{b}) \wedge x = b_i\right\} \qquad \begin{array}{l}\left\{ in = L \wedge (\vec{v} = \vec{b}) \wedge x = b_i \right\} \\ \text{if } (x \prec l) \text{ then } \{\Lambda\} \text{else } \{\Psi\} \\ \left\{ (\vec{t} = \vec{o}) \wedge out = W \right\}\end{array}}{\left\{in = \left\langle b_i \right\rangle L \wedge (\vec{v} = \vec{b}) \wedge b_i = b_i\right\} \ \mathit{read}(x);\ \text{if } (x \prec l) \text{ then } \{\Lambda\} \text{else } \{\Psi\} \left\{(\vec{t} = \vec{o}) \wedge out = W\right\}}$$

Given that:

$$SCC_M^{\theta}(\forall x \alpha(x)_T^V) = \left\{ \left\langle b_i \overset{\frown}{} L, \left\langle \vec{b}, \vec{o} \right\rangle, W \right\rangle \Big/ \forall b_i \in M \left(\left\langle L, \left\langle b_i \overset{\frown}{} \vec{b}, \vec{o} \right\rangle, W \right\rangle \in SCC_M^{\theta}\left(\alpha(y)_T^{V \cup y}\right) \right) \right\}$$

$$\text{Se } (U_\forall \subseteq \text{SCC}_M^{\theta}(\forall y \alpha(y)_T^V), \text{então}: U_\forall = \left\{ \left\langle b_i \overset{\frown}{} L, \left\langle \vec{b}, \vec{o} \right\rangle, W \right\rangle \Big/ \left\langle L, \left\langle b_i \overset{\frown}{} \vec{b}, \vec{o} \right\rangle, W \right\rangle \in U' \right\}$$

As $U_\forall$ is formed from U', we can say that $U_\forall$ is complete.

$$\text{Seja } K_1 \in U_\forall, K_1 = \left\langle b_i \overset{\frown}{} L, \left\langle \vec{b}, \vec{o} \right\rangle, W \right\rangle, \text{tal que}: K_2 = \left\langle L, \left\langle b_i \overset{\frown}{} \vec{b}, \vec{o} \right\rangle, W \right\rangle \in (U' \subseteq SCC_M^{\theta}(\alpha(x)_T^{V \cup x}))$$

From (1), we have that the program: *If* $(x \prec l)$ *then* $\{\Lambda\}$ *else* $\{\Psi\}$ calculates K_2.

Thus, from (2), we have that (*read*(*x*); *If* ($x \prec l$) *then* {Λ} *else* {Ψ}) calculates K_1. Let K_1 being an arbitrary triple that belongs to $U_\forall$, we can conclude that:

$$\Delta, \beta_1, ..., \beta_n, \delta_1, ..., \delta_k \models_{M,\sigma} read(x); if\ (x \prec l) then \{\Lambda\} else \{\Psi\} : \forall y \alpha(y)_T^V$$

In the program generated above, it will be added a tag (*Procedure*…) in a such way that the command *read*(…) together with the conditional command will be the body of a procedure.

Thus, we will have the following program associated to the conclusion ($\forall y \alpha(y)^V{}_T$):

Procedure Rec{
  read(x);
  if ($x \prec l$) *then* {Λ} *else* {Ψ(*Rec*)}
}

By the inductive hypothesis (2) ($\Delta, \beta_1, ..., \beta_n, \delta_1, ..., \delta_k, p{:}\alpha(a) \models_{M,\sigma} \Psi : \alpha(c)_T^{V \cup c}$):
$< L, < c \overset{\frown}{\vec{b}}, \vec{o} >, W > \in U_2$ from the supposition that $p : \alpha(a)_T^{V \cup a}$.

By construction, we have that the supposed program, associated to the program variable *p*, calculates $SCC_M^{\theta}(\alpha(a)_T^{V \cup a})$.

This program is associated to the inductive hypothesis ($\alpha(a)$), which can have an existential quantifier in its formula. In this way, we can have in the body of the Ψ program the tag: "…←*exec*(*p*,…)". In the construction of the program with this tag, the program calling associated to it, which in this case (induction rule), will replace it, which will be the proper constructed program. Thus, we will have a recursive calling.

Given the semantic of a recursive calling:

$$\frac{\left\{ in = L \wedge (\vec{v} = \vec{b}) \right\} Z \leftarrow exec(\Lambda, \vec{v}) \left\{ (Z = o_i) \wedge (\vec{t} = \vec{o}) \wedge out = W \right\} \vdash \begin{matrix} \left\{ in = L \wedge (\vec{v} = \vec{b}) \right\} \Psi \left\{ (\vec{t} = \vec{o}) \wedge out = W \right\}, \\ \text{Body}(\Lambda) = \Psi \end{matrix}}{\left\{ in = L \wedge (\vec{v} = \vec{b}) \right\}\ Z \leftarrow \Lambda\ \left\{ (Z = o_i) \wedge (\vec{t} = \vec{o}) \wedge out = W \right\}}$$

We have that the program associated to the inductive hypothesis is the fixed point of the generated program. So we can assure that the generated program is recursive.

Thus:

$$\Delta, \beta_1, ..., \beta_n, \delta_1, ..., \delta_k \models_{M,\sigma} \begin{cases} \text{Procedure Rec } \{ & \\ \quad \text{read}(x); & \\ \quad \text{if } (x < l) \text{ then } \{\Lambda\} & : \forall y \alpha(y)_T^V \\ \quad \text{else} & \\ \quad \{[\Psi \Leftarrow \{\text{exec}(p, \vec{v}) = \text{Rec}\}^*]\} & \end{cases}$$

Remark: the proof of the other inference rules is made in an analogous way (in a similar way) of the presented rules

Theorem: Let Π be a proof for a formula of the form $\forall x \exists y \alpha(x,y)$, from an axiom set (Δ) and a set of hypothesis that are not axioms (θ), and Λ the program provided by the function *GenProg*(*LabelMemoConfig* (Π)), then: $\theta, \Delta \models_{M,\sigma} \Lambda : \forall x \exists y \alpha(x,y)^{\{\}}_{\{\}}$

Proof: Applying the previous lemma we have the theorem proof.

4 - Example

In this section we show our program synthesis mechanism through an example, in which a program that calculates the remainder of a division is generated. The proof tree will be presented in blocks. The block of main proof - which has the theorem to be proved - will be a solid frame. The others, with traced frames, represent the branches that are connected with others by the numeration presented in the upper left side of the frame.

Example:

Proof Tree:

$$\dfrac{\dfrac{\dfrac{\vdots}{\alpha(a)} \quad \dfrac{\vdots}{\beta(a)}}{\alpha(a) \wedge \beta(a)} I\wedge}{\forall x(\alpha(x) \wedge \beta(x))} I\forall$$

Block Representation

(I) $\dfrac{\vdots}{\alpha(a)} \quad \dfrac{\vdots}{\beta(a)}$

$$\boxed{\dfrac{\dfrac{(\mathrm{I})}{\alpha(a) \wedge \beta(a)} I\wedge}{\forall x(\alpha(x) \wedge \beta(x))} I\forall}$$

To make easier the understanding of the proof, we use infix notation for addition, subtraction and multiplication operations, and the equality predicate and comparison predicates either. The functional s(x) expresses the operation of successor.

The program is generated from the theorem proof: $\forall v \forall u\big((v \to 0) \to \exists r \exists k\big(k * v = r = u\big) \wedge \big(r < v\big)\big)$ on the basis of the axioms: ∀x(x*1=x), ∀x(x+0=x), ∀q(0*q=0), ∀z∀p((z=p)→(z-p=0)), ∀z∀p((p>0)→(z-p<z)), ∀z∀q((z*s(q))→(z*q+z)), ∀z∀p∀q((z=p-q)→(z+q=p)) and the hypothesis: l=y.

(I)

$$\dfrac{\dfrac{\dfrac{\sigma:\forall x(1*x=x)^{\{\}}_{\{1\}}}{\sigma:1*x=x^{\{x\}}_{\{1\}}} E\forall \quad \dfrac{\sigma:\forall h(h+0=h)^{\{\}}_{\{0\}}}{\sigma:x+0=x^{\{x\}}_{\{0\}}} E\forall}{\sigma:1*x+0=x^{\{x\}}_{\{0,1\}}} \quad \dfrac{\sigma:[b=l]^{\{b,l\}(1)}_{\{\}} \quad \dfrac{\dfrac{\sigma:\forall z\forall p((z=p)\to(z-p=0))^{\{\}}_{\{0\}}}{\sigma:\forall p((b=p)\to(b-p=0))^{\{b\}}_{\{0\}}} E\forall}{\sigma:(b=l)\to(b-l=0)^{\{b,l\}}_{\{0\}}} E\forall}{\sigma:b-l=0^{\{b,l\}}_{\{0\}}} E\to}{\sigma:1*x+(b-l)=x^{\{x,b,l\}}_{\{(b-l),1\}}} Eq$$

(I)Continuation

$$\dfrac{\sigma:[x<0]^{\{x\}}_{\{0\}} \quad \dfrac{\sigma:[b=l]^{\{b,l\}(1)}_{\{\}} \quad \dfrac{\dfrac{\sigma:\forall z\forall p((z=p)\to(z-p=0))^{\{\}}_{\{0,1\}}}{\sigma:\forall p((b=p)\to(b-p=0))^{\{b\}}_{\{0,1\}}} E\forall}{\sigma:(b=l)\to(b-l=0)^{\{b,l\}}_{\{0,1\}}} E\forall}{\sigma:b-l=0^{\{b,l\}}_{\{0,1\}}} E\to}{\sigma:b-l<x^{\{x,b,l\}}_{\{(b-l),1\}}} E\to$$

(II)

$$\dfrac{\dfrac{\dfrac{\sigma:\forall q(0*q=0)^{\{\}}_{\{0\}}}{\sigma:0*x=0^{\{x\}}_{\{0\}}} E\forall \quad \dfrac{\sigma:\forall h(0+h=h)^{\{\}}_{\{0\}}}{\sigma:0+b=b^{\{b\}}_{\{0\}}} E\forall}{\sigma:0*x+b=b^{\{x,b\}}_{\{b,0\}}} Eq \quad \dfrac{\sigma:[b<l]^{\{b,l\}(1)}_{\{\}} \quad \sigma:l=x^{\{x,b\}}_{\{l,0\}}}{\sigma:b<x^{\{x,b\}}_{\{b,0\}}}}{}$$

(III)

$$\dfrac{\dfrac{\dfrac{\Lambda : \exists r \exists k'((k'^{*}x + r = d - x) \wedge (r < x))^{\{x,d\}}_{\{\}} \quad \text{(inductive hypothesis)}}{r \leftarrow exec(\Lambda, \vec{v}) : \exists k'((k'^{*}x + r = d - x) \wedge (r < x))^{\{x,d\}}_{\{r,k'\}}} E\exists}{k' \leftarrow exec(r \leftarrow exec(\Lambda, \vec{v}), \vec{v}) : ((k'^{*}x + r = d - x) \wedge (r < x))^{\{x,d\}}_{\{r,k'\}}} E\exists}{k' \leftarrow exec(r \leftarrow exec(\Lambda, \vec{v}), \vec{v}) : (k'^{*}x + r = d - x)^{\{x,d\}}_{\{r,k'\}} \; k' \leftarrow exec(r \leftarrow exec(\Lambda, \vec{v}), \vec{v}) : (r < x)^{\{x,d\}^{***}}_{\{r,k'\}}} E\wedge$$

$$\dfrac{\dfrac{k' \leftarrow exec(r \leftarrow exec(\Lambda, \vec{v}), \vec{v}) : (k'^{*}x + r = d - x)^{\{x,d\}}_{\{r,k'\}} \quad \sigma : (k'^{*}x + r = d - x) \rightarrow (k'^{*}x + r + x = d)^{\{x,d\}}_{\{\}}}{\sigma : (k'^{*}x + x + r = d)^{\{x,d\}}_{\{r,k'\}}} E\rightarrow \quad \dfrac{\dfrac{\dfrac{\sigma : \forall z \forall q(s(q) * z = q * z + z)^{\{\}}_{\{\}}}{\sigma : \forall q(s(q) * x = q * x + x)^{\{x\}}_{\{\}}} E\forall}{\sigma : (s(k') * x = k'^{*}x + x)^{\{x\}}_{\{k'\}}} E\forall}{\sigma : (s(k') * x + r = d)^{\{x,d\}}_{\{r,k'\}}} E\forall}{\sigma : (s(k') * x + r = d)^{\{x,d\}}_{\{r,k'\}}} Eq$$

$$\dfrac{k' \leftarrow exec(r \leftarrow exec(\Lambda, \vec{v}), \vec{v}) : (r < x)^{\{x,d\}^{***}}_{\{r,k'\}}}{k' \leftarrow exec(r \leftarrow exec(\Lambda, \vec{v}), \vec{v}) : r < x^{\{x,d\}}_{\{r,k'\}}}$$

$$\dfrac{\sigma : [x > 0]^{\{x\}}_{\{0\}} \quad \dfrac{\dfrac{\sigma : \forall z \forall p((p > 0) \rightarrow (z - p < z))^{\{\}}_{\{0\}}}{\sigma : \forall p((p > 0) \rightarrow (d - p < d))^{\{d\}}_{\{0\}}} E\forall}{\sigma : (x > 0) \rightarrow (d - x < d)^{\{x,d\}}_{\{0\}}} E\forall}{\sigma : d - x < d^{\{x,d\}}_{\{0\}}}$$

$$\dfrac{\dfrac{\sigma : b \leq l^{\{b,l\}}_{\{\}}}{\sigma : (b < l \vee b = l)^{\{b,l\}}_{\{\}}} \quad \dfrac{\dfrac{\dfrac{(I)}{\sigma : (1 * x + (b - l) = x)^{\{x,b,l\}}_{\{(b-l),1\}}} \quad \dfrac{(I)}{\sigma : ((b - l) < x)^{\{x,l\}}_{\{b-l,1\}}} I\wedge}{\sigma : (1 * x) + (b - l) = x) \wedge ((b - l) < x)^{\{x,b,l\}}_{\{(b-l),1\}}} I\exists}{write(1) : \exists k((k * x + (b - l) = x) \wedge ((b - l) < x))^{\{x,b,l\}}_{\{(b-l)\}}} I\exists \quad \dfrac{\dfrac{\dfrac{(II)}{\sigma : (0 * x + b = b)^{\{x,b\}}_{\{b,o\}}} \quad \dfrac{(II)}{\sigma : (b < x)^{\{x,b\}}_{\{b,o\}}} I\wedge}{\sigma : (0 * x + b = b) \wedge (b < x))^{\{x,b\}}_{\{b\}}} I\exists}{write(0) : \exists k((0 * x + b = b) \wedge (b < x))^{\{x,b\}}_{\{b\}}} I\exists}{write(1); write(b - l) : \exists r \exists k((k * x + r = x) \wedge (r < x))^{\{x,b,l\}}_{\{\}}} I\exists}{write(0); write(b) : \exists r \exists k((k * x + r = b) \wedge (r < x))^{\{x,b\}}_{\{\}}} I\exists}{if\,(b = l) then\,\{write(1); write(b - l)\}\, else\,\{if\,(b < l) then\,\{write(0); write(b)\}\} : \exists r \exists k((k * x + r = n) \wedge (r < x))^{\{x,n,l\}}_{\{\}}} E\vee$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(III)}{\sigma : (s(k') * x + r = d)^{\{x,d\}}_{\{r,k'\}}} \quad \dfrac{(III)}{k' \leftarrow exec(r \leftarrow exec(\Lambda, \vec{v}), \vec{v})(r < x)^{\{x,d\}}_{\{r,k'\}}} I\wedge}{k' \leftarrow exec(r \leftarrow exec(\Lambda, \vec{v}), \vec{v}) : (s(k') * x + r = d) \wedge (r < x)^{\{x,d\}}_{\{r'\}}} I\exists}{k' \leftarrow exec(r \leftarrow exec(\Lambda, \vec{v}), \vec{v}); write(s(k')) : \exists k((k * x + r = d) \wedge (r < x))^{\{x,d\}}_{\{\}}} I\exists}{k' \leftarrow exec(r \leftarrow exec(\Lambda, \vec{v}), \vec{v}); write(s(k')); write(r) : \exists r \exists k((k * x + r = d) \wedge (r < x))^{\{x,d\}}_{\{\}}} I\rightarrow \quad \dfrac{(IV)}{\sigma : d - x < d^{\{x,d\}}_{\{0\}}}$$

$$I\forall(1)(induction)$$

Procedure *resto* {
read(*b*); *if* (*b* ≤ *l*) *then* { *if* (*b* = *l*) *then* {*write*(1); *write*(*b* − *l*)}
else {*if* (*b* < *l*) *then* {*write*(0); *write*(*b*)}}}
else {*write*(*b* − *x*); *r* ← (*k'* ← *resto*); *write*(*s*(*k'*)); *write*(*r*)}
} *resto*;

$$\forall u((x > 0) \rightarrow \exists r \exists k((k * x + r = u) \wedge (r < x)))^{\{x,l\}}_{\{0\}}$$

$$I\forall$$

ProgramRestoDiv {
Procedure *resto* {
read(*b*); *if* (*b* ≤ *l*) *then* { *if* (*b* = *l*) *then* {*write*(1); *write*(*b* − *l*)}
else {*if* (*b* < *l*) *then* {*write*(0); *write*(*b*)}}}
else {*write*(*b* − *x*); (*r*, *k*) := *resto*); *write*(*s*(*k'*)); *write*(*r*)}}
read(*x*); *l* := *x***; *resto*;
}

$$\forall y \forall u((y > 0) \rightarrow \exists r \exists k((k * y + r = u) \wedge (r < y)))^{\{l\}}_{\{0\}}$$

**In the proof hypothesis *y*=*l* represents a memory restriction, where *l* has the value of *y*.

Remark: In our system the communications between the functions (parameter passing) is made by read and write file operations.

5 – Conclusion

In this work we have presented an automatic method of program synthesis operating as follows: it transforms a given proof based on a specification to a program (in a imperative language), guaranteeing the correctness of the latter.

The syntactical restrictions imposed on the proof, from which we extract the semi-computational contents may cause some loss of the expressive power, thus limiting the domain of application of the synthesis procedure.

Among the main contributions of this work, we stress the proposal of a new synthesizer that generates legible programs in an imperative language along with a correctness proof of this mapping. The other synthesizers exposed in the existing literature generate programs that are not very legible in functional or logical programming languages. Also, our constructive program synthesis procedure receives as input a declarative specification in predicate logic, which allows us to express the problem in a simple way than the synthesizers using intuitionistic type theory (**Nuprl**[4], **Oyster**[3] e **NJL**[9]) and equational logic (**Lemma** [6]).

Among the directions to extend this work, we expect to investigate the feasibility of relaxing some restrictions on the proofs, so as to extract semi-computational contents from proofs that use the negation, introduction rule or have existentially quantified formulas as hypothesis. Also, the usage of more that one proof method seems attractive and program synthesizers based on such ideas are being investigated.

References

[1] BENL, H., BEGER, U., SCHWICHTENBERG, H., SEISENBERGER, M. and ZUBER,W. Proof theory at work: Program development in the Minlog system, *Automated Deduction, W. Bibeland P.H.Schmitt,*(eds.), Vol II, Kluwer 1998

[2] BATES, J.L. and CONSTABLE, R.L.- "Proof as Programs". *ACM Transactions on Programming Languagens and Systems,* 7(1): 113-136,1985.

[3] BUNDY, A., SMAIL, A., and WIGGINS, G. A. – "The synthesis of logic programs from inductive proofs". In J. Lloyd (ed.), *Computational Logic*, pp 135-149. Springer-Verlag,1990.

[4] CALDWELL, J.L., IAN, P., UNDERWOOD, J.G. – "Search algorithms in Type Theory". http://meru.cs.uwyo.edu/~jlc/papers.html

[5] CHANG, C., LEE, R.C – "*Symbolic Logic and Mechanical Theorem Prover*". Academic Press, 1973

[6] CHARARAIN, J. e MULLER, S. – "Automated synthesis of recursive programs from a ∀∃ logical Specification". *Journal of Automated Reasoning* 21: 233-275, 1998.

[7] DEVILLE, Y., LAU, K. – "Logic Program Synthesis"- *Journal of Logic Programming* 1993:12:1-199

[8] FLOYD, R. – "Assigning meaning to programs". *Symposia in Applied Mathematics* 19:19-32, 1967.

[9] GOTO, S. – "Program synthesis from natural deductions proofs". *International Joint Conference on Artificial Intelligence*, 339-341. Tokyo 1979.

[10] GIRARD, J., LAFONT, Y. and TAYLOR, P. – *Proof and Types*. Cambrigde University Press, 1989.

[11] HOWARD, W.A. – "The Formulae-as-Types Notion of Construction". In Hindley, J.R., Seldin, J.P.(ed.), *To H.B. Curry: Essays on combinatory logic, Lambda Calculus and Formalisation*. Academic Press, 1980.

[12] HOARE, C.A.R and WHIRTH, N. – "An axiomatic Definition of the Programming Language PASCAL" –December, 1972. *Acta Informatica 2*: 335-355, Springer-Verlag, 1973.

[13] KREITZ, C. – "Program synthesis - Automated Deduction - A basis for Applications", pp 105-134, Kluwer,1998.

[14] LAU, K. and WIGGINS, G. – "A tutorial on Synthesis of Logic Programs form Specifications". In P. Van Hentenryck (ed.), *Proceedings of the Eleventh International Conference on Logic Programming*, pp 11-14, MIT Press, 1994.

[15] MARTIN-LÖF, P. – "Intuitionistic Type Theory".*Edizioni di filosofia e Scienza,* Biblioplolis, 1984.

[16] MANNA,Z. and WALDINGER,R. – "A Deductive Approach to program synthesis". *ACM transactions on Programming Languages and Systems*, 2(1):90-121, 1980

[17] VELOSO, P.A.S. – "Outlines of a mathematical theory of general problems". *Philosophia Naturalis*, 21(2/4): 234-362, 1984.

[18] HAEUSLER, E.H. – "Extracting Solution from Constructive Proofs: Towards a Programming Methodology". *Brasilian Eletronic Journal on Mathematics of Computation* (BEJMC). No. 0- Vol 0, 1999. (http://gmc.ucpel.tche.br/bejmc)

[19] SILVA, G.M.H. – "Um Estudo em Síntese Construtiva de Programas utilizando Lógica Intuicionista". Dissertação de Mestrado – Departamento de Informática -PUC-Rio. 1999.