Improving Constructive Synthesizers by Tabulation Techniques and Domain Ordering

F. J. Galan and J. M. Canete

Abstract

Constructive synthesis is a systematic method which construct programs from specifications following the proofs-as-programs paradigm. Specifications are considered as conjectures in the context of a theory, synthesis methods construct proofs from conjectures and then programs are extracted from the proofs. The halting criteria is clear; synthesis halts when proof is completed but proof completion is a difficult and expensive task to be mechanized. Constructive synthesizers are (meta)programs which implement constructive synthesis methods. Our work is interested to show how memoization and domain ordering techniques assist synthesizers in the automatic program construction process.

Keywords: Memoization, Domain Ordering, Logic Program Synthesis, Constructive Proof.

1 Introduction

Constructive synthesis represents a systematic method to construct programs from specifications following the proofs-as-programs approach ([2]). In this work, we are interested in synthesis of logic programs. Following this idea, programming is understood as a deductive task ([5]). The specification of the problem to be synthesized is considered a *conjecture* in the context of a theory $\mathcal{T}$, a proof is constructed and then a (logic) program is extracted from the proof [4]. This is possible because each proof rule used in the proof system has an associated program construction rule which has been predefined and proved to be correct, allowing each step of the proof to generate the corresponding program fragments [1]. Programs extracted from the constructive proofs are totally correct, assuming that the proof system is sound and that the program construction rules associated with the proof system are correct. However, an automatic proofs-as-programs system is very difficult to construct; constructive synthesizers are conceived as interactive systems which construct proofs by human guidance. In order to increase the automation in the synthesis process, it is

Francisco José Galán Morillo and José Miguel Canete Valdeón, Departamento de Lenguajes y Sistemas Informaticos. Facultad de Informática y Estadística. Av. Reina Mercedes s/n 41012, Sevilla.E-mail: `galanm@lsi.us.es` and `canete@lsi.us.es`

necessary to mechanize the use of the proof rules. Without any control is imposible to synthesize programs of a automatic manner. The automatic proof construction process is favoured by the use of domain ordering techniques and memoization. The idea is to establish an order between literals in a proof; a proof rule is applied only if it respects this order. On the other hand, many proofs developed in a synthesis process are redundant. The use of memoization is essential to improve efficiency aspects.

The work is organized in the following manner, section 2 introduces definitions, section 3 defines a control mechanism in the synthesis process based on domain ordering techniques and memoization and section 4 presents conclusions and future work.

2 Definitions

Definition (Context Theory). The synthesis process is developed in a theory. From a syntactical point of view, a theory consists of the following parts: (a) *Parameter* section declares type and relation parameters, (b) *P-Restrictions* section includes restrictions on correct parameter substitutions, (c) *Sort* section declares sorts, (d) *Function symbol* and *Relation symbol* sections define the theory signature and (d) *Definition* section defines the meaning of relation symbols. A definition $D_{i.j}$ is a formula of the form $H \Leftrightarrow B$ where H is a positive literal and B a formula. The expressions $hd(H \Leftrightarrow B)$ and $bd(H \Leftrightarrow B)$ represent H and B respectively.

For example, the theory *Sequence* represents a formalization of parametric sequences. *Sequence* imports the theory of natural numbers ($Natural$) where 0 and s are function symbols to represent natural numbers and *less* is a relation between natural numbers with the following meaning: $less(0, s(y)) \Leftrightarrow true$, $less(x, 0) \Leftrightarrow false$ and $less(s(x), s(y)) \Leftrightarrow less(x, y)$. *Sequence* declares function symbols *nil* and *conc* for generating *Sequence*$(elem)$ sort, $=: (elem, elem)$ is the equality relation for *elem* values and $=:$ (*Sequence*$(elem)$,*Sequence*$(elem)$) is the equality relations for *Sequence*$(elem)$ values, *ielement* is a relation which decides the position of an element in a sequence, *length* is a relation which decides the number of elements in a sequence, *ocur* is a relation which decides the number of occurrences of an element in a sequence and *perm* is a relation which decides the permutation of a sequence.

Theory *Sequence*;
Parameters:
  Types: *elem*;
  Relations: $\preceq: (elem, elem)$;
Import $Natural$;
Sorts: $Sequence(elem)$;
Function symbols:
  $nil\ : \rightarrow Sequence(elem)$;
  $conc\ :\ (elem, Sequence(elem)) \rightarrow Sequence(elem)$;
Relation symbols:
  $=:\ (elem, elem)$;
  $=:\ (Sequence(elem), Sequence(elem))$;

$ielement : (Sequence(elem), Nat, elem);$
$ocur : (elem, Secuence(elem), Nat);$
$length : (Secuence(elem), Nat);$
$perm : (Secuence(elem), Secuence(elem));$

Definitions:

$$\begin{aligned}
D_{1.1}: &\quad ielement(L, 0, e) \Leftrightarrow \exists B(L = conc(e, B));\\
D_{1.2}: &\quad ielement(L, s(i), e) \Leftrightarrow \exists a, B(L = conc(a, B) \wedge ielement(B, i, e));\\
\\
D_{2.1}: &\quad length(L, n) \Leftrightarrow \forall i(less(i, n) \Leftrightarrow ielement(L, i, a))\\
\\
D_{3.1}: &\quad ocur(e, nil, n) \Leftrightarrow n = 0;\\
D_{3.2}: &\quad ocur(e, conc(x, Y), z) \Leftrightarrow \neg e = z \wedge ocur(e, Y, z);\\
D_{3.3}: &\quad ocur(e, conc(x, Y), s(z)) \Leftrightarrow e = x \wedge ocur(e, Y, z);\\
\\
D_{4.1}: &\quad perm(L, S) \Leftrightarrow \forall a, z(nocc(a, L, z) \Leftrightarrow nocc(a, S, z));\\
\\
D_{5.1}: &\quad x = y \Leftrightarrow x \preceq y \wedge y \preceq x;\\
\\
D_{6.1}: &\quad nil = nil \Leftrightarrow true;\\
D_{6.2}: &\quad nil = conc(a, B) \Leftrightarrow false;\\
D_{6.3}: &\quad conc(x, L) = L \Leftrightarrow false;\\
D_{6.4}: &\quad conc(a, A) = conc(b, B) \Leftrightarrow a = b \wedge A = B;
\end{aligned}$$

P-Restrictions:

$x \preceq z \Leftarrow x \preceq y \wedge y \preceq z;$

End $Sequence;$

Definition (Conjecture Synthesis). To synthesize (typed) logic programs, the synthesis conjecture is [1]

$$\mathcal{T} \vdash \forall x_1, ..., x_n \, \exists_{Boolean} \, b \, (r(x_1, ..., x_n) \hookrightarrow b)$$

where $\mathcal{T}$ is the context theory, r is the relation to be synthesized, $x_1, ..., x_n$ are input variables (with types $\tau_1, ..., \tau_n$ respectively), b is a proof witness (metavariable) [6] and the meaning of the $\hookrightarrow$ operator is defined by

$\mathcal{T} \vdash \forall x_1, ..., x_n r(x_1, ..., x_n)$ is equivalent to $b = true$
$\mathcal{T} \vdash \neg(\forall x_1, ..., x_n r(x_1, ..., x_n))$ is equivalent to $b = false$

For example, the synthesis conjecture for *length* in *Sequence* is:

$$Sequence \vdash \forall L, n \exists b_1 (length(L, n)) \hookrightarrow b_1 \ (*)$$

The first step is to produce a proof of this conjecture. Proofs are represented by trees (synthesis trees).

Definition (Synthesis Tree). A synthesis tree is a representation of the synthesis process for a relation r in $\mathcal{T}$. A *node* in the tree represents a stage of the synthesis process. It is composed of (a) an index, nd, to identify each node in the tree, (b) a formula, F_{nd}, and (c) a metavariable, b_{nd}, to represent a proof of F_{nd}. For example, the expression $(*)$ is represented by the following node:

Node Index	*Formula*	*Proof witness*
(1)	$\forall L, n(length(L, n))$	b_1

An *edge* represents a proof step. Each proof rule has an associated program construction rule. We emphasize three proof rules: unfold, fold and induction rules.

Definition (Unfold Rule). Let F_{nd} be the formula of a node nd. Let A be a positive literal in F_{nd}. Suposse that (a) $\{D_{i.1}, ..., D_{i.n}\}$ with $i, n > 0$ is the subset of definitions in a theory $\mathcal{T}$ such that A is unifiable with $hd(D_{i.1}), ..., hd(D_{i.n})$ with most general unifiers $\sigma_1, ..., \sigma_n$, respectively and (b)F_{nd_j} is the formula $((F_{nd})^A_{bd(D_{i.j})})\sigma_j$, for $j = 1, ..., n$ where $(F_{nd})^A_{bd(D_{i.j})}$ denotes the replacement in F_{nd} of A by $bd(D_{i.j})$.

Applying unfold rule to node nd produces n descendant nodes nd_j. Each nd_j has a formula F_{nd_j}. The metavariable b_{nd} is equal to $\bigvee b_{nd_j}$. The program construction rule associated to unfold rule is $F_{nd} \Leftrightarrow \bigvee F_{nd_j}$.

For example, applying unfold rule on literal *length* in (1) produces:

$$\forall L, n, i(less(i, n) \Leftrightarrow \exists a(ielement(L, i, a))) \hookrightarrow b_2 \quad (2)$$

program fragment:

$$(length(L, n))_1 \Leftrightarrow (\forall i(less(i, n) \Leftrightarrow \exists a(ielement(L, i, a))))_2 \quad [1]$$

and metavariable $b_1 = b_2$.

Definition (Fold Rule). Let F_{nd} be the formula of a node nd. Let R be a subformula of F_{nd}. Suposse that (a) $\{D_{i.1}, ..., D_{i.n}\}$ with $i, n > 0$ is the subset of definitions in a theory $\mathcal{T}$ such that $bd(D_{i.j})$ is unifiable with R via most general unifier σ_j, for $j = 1, ..., n$, (b)F_{nd_j} is the formula $((F_{nd})^R_{hd(D_{i.j})})\sigma_j$ where $(F_{nd})^R_{hd(D_{i.j})}$ denotes the replacement in F_{nd} of R by $hd(D_{i.j})$.

Applying fold rule to node nd produces n descendant nodes nd_j. Each nd_j has a formula F_{nd_j}. The metavariable b_{nd} is equal to $\bigvee b_{nd_j}$. The program construction rule associated to unfold rule is $F_{nd} \Leftrightarrow \bigvee F_{nd_j}$.

For example, applying fold rule on subformula $\forall Y, l, k(less(k, l) \Leftrightarrow \exists a(ielement(Y, k, a)))$ in (9.2.2.2) produces:

$$length(Y, l) \hookrightarrow b_{10.2.2.2} \quad (10.2.2.2)$$

program fragment:

$$\begin{array}{l}(\forall Y, l, k(less(k, l) \Leftrightarrow \exists a(ielement(Y, k, a))))_{9.2.2.2} \Leftrightarrow \quad [9.2.2.2] \\ \qquad (length(Y, l))_{10.2.2.2}\end{array}$$

and metavariable $b_{9.2.2.2} = b_{10.2.2.2}$.

Definition (Induction Rule): Let F_{nd} be the formula of a node nd. Let x be a universally quantified variable in a literal of F_{nd}. Let τ be the type of x generated by n function symbols, f_j, for $j = 1, ..., n$. Applying induction rule on x in F_{nd} produces n descendant nodes nd_j. Each nd_j has a formula $F_{nd}\sigma_j$ where σ_j denotes the substitution obtained from x and f_j. The metavariable b_{nd} is equal to $\bigwedge b_{nd_j}$. The program construction rule is equal to $F_{nd} \Leftrightarrow \bigvee F_{nd}\sigma_j$.

If x is existentially quantified variable then induction rule produces similar effects but the metavariable b_{nd} is equal to $\bigvee b_{nd_j}$.

For example, applying induction rule on n in (2) produces two descendant nodes, $nd_{3.1}$ and $nd_{3.2}$ respectively:

$$\begin{array}{ll} \forall L, i(less(i,0) \Leftrightarrow \exists a(ielement(L,i,a))) \hookrightarrow b_{3.1} & (3.1) \\ \forall L, l, i(less(i,s(l)) \Leftrightarrow \exists a(ielement(L,i,a))) \hookrightarrow b_{3.2} & (3.2) \end{array}$$

program fragment:

$$\begin{array}{ll} (\forall i(less(i,n) \Leftrightarrow \exists a(ielement(L,i,a))))_2 \Leftrightarrow & [2] \\ \quad (\forall i(less(i,0) \Leftrightarrow \exists a(ielement(L,i,a))))_{3.1} & \\ \quad\quad \vee & \\ \quad (\forall i, l(less(i,s(l)) \Leftrightarrow \exists a(ielement(L,i,a))))_{3.2} & \end{array}$$

and metavariable $b_2 = b_{3.1} \wedge b_{3.2}$.

Definition (Proofs-as-Logic Programs): A (sub)proof can be interpreted as a logic (sub)program iif the (sub)proof is completed using unfold, fold and induction rules.

The following example shows a completed subproof for (5.1.1) and its associated program :

$$\begin{array}{ll} \forall L, i(less(i,0) \Leftrightarrow \exists a(ielement(L,i,a))) \hookrightarrow b_{3.1} & (3.1) \\ \forall L, i(false \Leftrightarrow \exists a(ielement(L,i,a))) \hookrightarrow b_{4.1} & (4.1)\ (unfold\, less(i,0)) \\ \forall i(false \Leftrightarrow \exists a(ielement(nil,i,a))) \hookrightarrow b_{5.1.1} & (5.1.1)\ (induction\ on\ L) \\ \forall x, Y, i(false \Leftrightarrow \exists a(ielement(conc(x,Y),i,a))) \hookrightarrow b_{5.1.2} & (5.1.2)\ (induction\ on\ L) \end{array}$$

$$\begin{array}{ll} false \Leftrightarrow \exists a(ielement(nil,0,a)) \hookrightarrow b_{6.1.1.1} & (6.1.1.1)\ (induction\ on\ i) \\ (\forall k(false \Leftrightarrow \exists a(ielement(nil,s(k),a)))) \hookrightarrow b_{6.1.1.2} & (6.1.1.2)\ (induction\ on\ i) \end{array}$$

$$\begin{array}{ll} false \Leftrightarrow \exists a(\exists B(nil = conc(a,B))) \hookrightarrow b_{7.1.1.1} & (7.1.1.1)\ (unfold\, nil = conc(a,B)) \\ false \Leftrightarrow false \hookrightarrow b_{8.1.1.1} & (8.1.1.1)\ (\Leftrightarrow) \\ true \hookrightarrow b_{9.1.1.1} & (9.1.1.1) \end{array}$$

$$\begin{array}{ll} false \Leftrightarrow \exists a(\exists B(nil = conc(a,B) \wedge ielement(B,k,a))) \hookrightarrow b_{7.1.1.2} & (7.1.1.2)\ (unfold) \\ false \Leftrightarrow \exists a(\exists B(false \wedge ielement(B,k,a))) \hookrightarrow b_{8.1.1.2} & (8.1.1.2)\ (unfold) \\ false \Leftrightarrow false \hookrightarrow b_{8.1.1.2} & (8.1.1.2)\ (\Leftrightarrow) \\ true \hookrightarrow b_{9.1.1.2} & (9.1.1.2) \end{array}$$

program fragment:

$$
\begin{array}{ll}
(\forall L, i(less(i,0) \Leftrightarrow \exists a(ielement(L,i,a))))_{3.1} \Leftrightarrow & [3.1] \\
\quad (\forall L, i(false \Leftrightarrow \exists a(ielement(L,i,a))))_{4.1} \Leftrightarrow & [4.1] \\
\quad\quad (\forall i(false \Leftrightarrow \exists a(ielement(nil,i,a))))_{5.1.1} & [5.1.1] \\
\quad\quad\quad \vee & \\
\quad\quad (\forall x, Y, i(false \Leftrightarrow \exists a(ielement(conc(x,Y),i,a))))_{5.1.2} & [5.1.2]
\end{array}
$$

$$
\begin{array}{ll}
(\forall i(false \Leftrightarrow \exists a(ielement(nil,i,a))))_{5.1.1} \Leftrightarrow & [5.1.1] \\
\quad (false \Leftrightarrow \exists a(ielement(nil,0,a)))_{6.1.1.1} & [6.1.1.1] \\
\quad\quad \vee & \\
\quad (\forall k(false \Leftrightarrow \exists a(ielement(nil,s(k),a))))_{6.1.1.2} & [6.1.1.2]
\end{array}
$$

$$
\begin{array}{ll}
(false \Leftrightarrow \exists a(ielement(nil,0,a)))_{6.1.1.1} \Leftrightarrow & [6.1.1.1] \\
\quad (false \Leftrightarrow \exists a(\exists B(nil = conc(a,B))))_{7.1.1.1} \Leftrightarrow & [7.1.1.1] \\
\quad\quad (false \Leftrightarrow false)_{8.1.1.1} \Leftrightarrow & [8.1.1.1] \\
\quad\quad\quad (true)_{9.1.1.1} & [9.1.1.1]
\end{array}
$$

$$
\begin{array}{ll}
(\forall k(false \Leftrightarrow \exists a(ielement(nil,s(k),a))))_{6.1.1.2} \Leftrightarrow & [6.1.1.2] \\
\quad (false \Leftrightarrow \exists a(\exists B(nil = conc(a,B) \wedge ielement(B,k,a))))_{7.1.1.2} \Leftrightarrow & [7.1.1.2] \\
\quad\quad (false \Leftrightarrow \exists a(\exists B(false \wedge ielement(B,k,a))))_{8.1.1.2} \Leftrightarrow & [8.1.1.2] \\
\quad\quad\quad (true)_{9.1.1.2} & [9.1.1.2]
\end{array}
$$

At this point, from [1], [2], [3.1], [4.1], [5.1.1], [5.1.2], [6.1.1.1], [7.1.1.1], [8.1.1.1], [9.1.1.1], [6.1.1.2], [7.1.1.2], [8.1.1.2], [9.1.1.2] we obtain the following program:

$$
\begin{array}{ll}
(length(L,n))_1 \Leftrightarrow & ((((n = 0 \wedge L = nil \wedge i = 0 \wedge true)_{6.1.1.1} \\
& \quad\quad \vee \\
& (n = 0 \wedge L = nil \wedge i = s(k) \wedge true)_{6.1.1.2})_{5.1.1} \\
& \quad \vee \\
& (\forall x, Y, i(false \Leftrightarrow \exists a(ielement(conc(x,Y),i,a))))_{5.1.2})_{3.1} \\
& \quad \vee \\
& (\forall i, l(less(i,s(l)) \Leftrightarrow \exists a(ielement(L,i,a))))_{3.2})_2
\end{array}
$$

The construction of the program is completed if proofs (5.1.2) and (3.2) are completed. At this point, we can consider that the program is partially constructed.

3 Control in the Synthesis Process

Today, synthesizers need a lot of human guidance. An automatic use of proof rules without any control presents an important problem: the synthesis is not effective. The risk of *infinite* and *redundant* synthesis is clear.

Example 1: *Redundant Proofs-Redundant Synthesis.*

A synthesis process of (5.1.2) without any human guidance can produce redundant synthesis:

Applying induction on i in (5.1.2) produces:

$$\begin{array}{ll} \forall x, Y, i(false \Leftrightarrow \exists a(ielement(conc(x, Y), i, a))) \hookrightarrow b_{5.1.2} & (5.1.2) \\ \forall x, Y(false \Leftrightarrow \exists a(ielement(conc(x, Y), 0, a))) \hookrightarrow b_{6.1.2.1} & (6.1.2.1) \\ \forall x, Y, k(false \Leftrightarrow \exists a(ielement(conc(x, Y), s(k), a))) \hookrightarrow b_{6.1.2.2} & (6.1.2.2) \end{array}$$

program fragment:

$$\begin{array}{ll} (\forall x, Y, i(false \Leftrightarrow \exists a(ielement(conc(x, Y), i, a))))_{5.1.2} \Leftrightarrow & [5.1.2] \\ \quad (\forall x, Y(false \Leftrightarrow \exists a(ielement(conc(x, Y), 0, a))))_{6.1.2.1} & \\ \quad\quad \vee & \\ \quad (\forall x, Y, k(false \Leftrightarrow \exists a(ielement(conc(x, Y), s(k), a))))_{6.1.2.2} & \end{array}$$

and metavariable $b_{5.1.2} = b_{6.1.2.1} \wedge b_{6.1.2.2}$.

A proof of (5.1.2) implies a proof of (6.1.2.1) and a proof of (6.1.2.2). A proof of (6.1.2.1) is obtained by applying unfold rule (4 times (7.1.2.1), (8.1.2.1), (9.1.2.1) y (10.1.2.1)) and then applying definition of $\Leftrightarrow$ ((11.1.2.1)):

$$\begin{array}{ll} \forall x, Y(false \Leftrightarrow \exists a \exists B(conc(x, Y) = conc(a, B))) \hookrightarrow b_{7.1.2.1} & (7.1.2.1) \\ \forall x, Y(false \Leftrightarrow \exists a \exists B(x = a \wedge Y = B)) \hookrightarrow b_{8.1.2.1} & (8.1.2.1) \\ \forall x, Y(false \Leftrightarrow \exists a \exists B(x = a \wedge Y = B)) \hookrightarrow b_{9.1.2.1} & (9.1.2.1) \\ \forall x, Y(false \Leftrightarrow true) \hookrightarrow b_{10.1.2.1} & (10.1.2.1) \\ (false) \hookrightarrow b_{11.1.2.1} & (11.1.2.1) \end{array}$$

However, a proof of (6.1.2.2) is not necessary because $b_{5.1.2}$ is equal to $false$; it does not depend of the value of $b_{6.1.2.2}$. Therefore, the synthesis of (6.1.2.2) is redundant; the synthesized program does not depend of the proof of (6.1.2.2).

Example 2: *Infinite Proofs-Infinite Synthesis.*

A synthesis process of (3.2) without any human guidance can produce infinite synthesis.

Applying induction rule on n in (3.2) produces:

$$\begin{array}{ll} \forall l, i(less(i, s(l)) \Leftrightarrow \exists a(ielement(nil, i, a))) \hookrightarrow b_{4.2.1} & (4.2.1) \\ \forall x, Y, l, i(less(i, s(l)) \Leftrightarrow \exists a(ielement(conc(x, Y), i, a))) \hookrightarrow b_{4.2.1} & (4.2.2) \end{array}$$

Applying induction rule on i in (4.2.1) produces:

$$\begin{array}{ll} \forall l(less(0, s(l)) \Leftrightarrow \exists a(ielement(nil, 0, a))) \hookrightarrow b_{5.2.1.1} & (5.2.1.1) \\ \forall l, k(less(s(k), s(l)) \Leftrightarrow \exists a(ielement(nil, s(k), a))) \hookrightarrow b_{5.2.1.2} & (5.2.1.2) \end{array}$$

Applying unfold rule on $less(s(k), s(l))$ in (5.2.1.2) produces:

$$\forall l, k(less(k, l) \Leftrightarrow \exists a(ielement(nil, s(k), a))) \hookrightarrow b_{6.2.1.2} \quad (6.2.1.2)$$

Applying unfold rule on $ielement(nil, s(k), a)$ in (6.2.1.2) produces:

$$\forall l, k(less(k, l) \Leftrightarrow \exists a \exists B(nil = conc(a, B)) \hookrightarrow b_{7.2.1.2} \quad (7.2.1.2)$$

Applying unfold rule on $nil = conc(a, B)$ in (7.2.1.2) produces:

$$\forall l, k(less(k, l) \Leftrightarrow false) \hookrightarrow b_{8.2.1.2} \quad (8.2.1.2)$$

Applying induction rule on l in (8.2.1.2) produces:

$$\begin{array}{ll} \forall k(less(k, 0) \Leftrightarrow false) \hookrightarrow b_{9.2.1.2.1} & (9.2.1.2.1) \\ \forall k, l^{'}(less(k, s(l^{'})) \Leftrightarrow false) \hookrightarrow b_{9.2.1.2.2} & (9.2.1.2.2) \end{array}$$

Applying induction rule on k in (9.2.1.2.2) produces:

$$\begin{array}{ll} \forall l^{'}(less(0, s(l^{'})) \Leftrightarrow false) \hookrightarrow b_{10.2.1.2.2.1} & (10.2.1.2.2.1) \\ \forall k^{'}, l^{'}(less(s(k^{'}), s(l^{'})) \Leftrightarrow false) \hookrightarrow b_{10.2.1.2.2.2} & (10.2.1.2.2.2) \end{array}$$

Applying unfold rule on $less(s(k^{'}), s(l^{'}))$ in (10.2.1.2.2.2) produces:

$$\forall k^{'}, l^{'}(less(k^{'}, l^{'})) \Leftrightarrow false) \hookrightarrow b_{11.2.1.2.2.2} \quad (11.2.1.2.2.2)$$

At this point, we observe that (11.2.1.2.2.2) is a variant of (8.2.1.2) and then the synthesis process fall in a loop.

3.1 Domain Ordering, Proof Rules and Infinite Synthesis

To improve constructive synthesizers it is necessary to guide automatic proof completions. The guidance is based on the establishment of a order between literals in a proof.

Definition (Term Domain Order). Let t, s be two terms of sort τ. t is more general than s ($t > s$) or s is more specific than t iff t is a variable and s is a non-variable term or $t = f(t_1, ..., t_n)$ and $s = f(s_1, ..., s_n)$ and for all $i = 1..n$, $t_i > s_i$.

For example, $x > 0$ and $conc(0, Y) > conc(0, conc(0, Z))$ but $conc(0, conc(s(0),$-$nil)) \not> conc(s(0), conc(s(0), nil))$ and $conc(x, nil) \not> conc(s(0), Y)$

Definition (Relation Domain Order). Let $r(t_1, ..., t_n)$ and $r(s_1, ..., s_n)$ be two atoms. $r(t_1, ..., t_n)$ is more general than $r(s_1, ..., s_n)$ iff for all $i = 1..n$, $t_i > s_i$.

For example, $less(x, 0) > less(s(k), 0)$ and $less(i, n) > less(0, s(y))$ but $less(0, x) \not> less(y, s(k))$ and $ielement(Y, s(k), a) \not> ielement(nil, i, a)$

Definition (Unfold Rule with Order). Let F_{nd} be the formula of a node nd. Let A be a positive literal in F_{nd}. Suposse that (a) $\{D_{i.1}, ..., D_{i.n}\}$ with $i, n > 0$ is the subset of definitions in a theory $\mathcal{T}$ such that A is unifiable with $hd(D_{i.1}), ..., hd(D_{i.n})$ with most general unifiers $\sigma_1, ..., \sigma_n$, respectively. It is allowed to apply unfold rule on a literal A using $D_{i.j}$ iff there are mgu's, σ_j, such that $A = hd(D_{i.j})\sigma_j$ for $j = 1, ..., n$. Applying unfold rule to node nd produces n descendant nodes nd_j. Each nd_j has a formula F_{nd_j} equal to $((F_{nd})^{A}_{bd(D_{i.j})})\sigma_j$.

Definition (Fold Rule with Order). Let F_{nd} be the formula of a node nd. Let R be a subformula of F_{nd}. Suposse that (a) $\{D_{i.1}, ..., D_{i.n}\}$ with $i, n > 0$ is the subset of definitions in a theory $\mathcal{T}$ such that R is unifiable with $bd(D_{i.1}), ..., bd(D_{i.n})$ with most general unifiers $\sigma_1, ..., \sigma_n$, respectively. It is allowed to apply fold rule on R using $D_{i.j}$ iff there are mgu's, σ_j, such that $R = bd(D_{i.j})\sigma_j$ for $j = 1, ..., n$. Applying fold rule to node nd produces n descendant nodes nd_j. Each nd_j has a formula F_{nd_j} equal to $((F_{nd})^R_{hd(D_{i.j})})\sigma_j$.

Definition (Induction Rule with Order). It is allowed to apply induction rule on a quantified variable iff each literal in the scope of this variable is more specific than such literal in the ancestor nodes (if they exist).

For example, the proof shown in Example 2 is constructed without order considerations, the rows 3 and 4 show that the induction rule has been applied on $less(k, l)$ but it is no more specific than their ancestors $less(i, n)$ and $less(i, s(l))$.

Literal	*Node Index*	*Antecesor Literal*	*Node Index*
$less(i, n)$	(2)		
$less(i, s(l))$	(4.2.1)	$less(i, n)$	(2)
$less(k, l)$	(8.2.1.2)	$less(i, n)$	(2)
$less(k, l)$	(8.2.1.2)	$less(i, s(l))$	(4.2.1)

Considering proof rules with order, at point (8.2.1.2) it is not allowed to apply induction and then we overcome the synthesis loop.

3.2 Memoization and Redundant Synthesis

The synthesis of a conjecture can produce many redundant subconjectures. The use of memoization assists in efficiency considerations because redundant subconjectures are not synthesized. The information in synthesis tree nodes play an essential role to prevent redundant proofs in two sense: (a) metavariables b assists in proof completions by lazy evaluation on proofs and (b) synthesis subtrees can be reuse if subconjectures in respective root nodes are variant formulas.

Example 1 in section 3 shows an example of (a)[1], the proof of subconjecture is represented by means of the metavariable $b_{5.1.2} = b_{6.1.2.1} \wedge b_{6.1.2.2}$. It is not necessary to construct a completed proof for $b_{5.1.2}$ because the proof witness $b_{6.1.2.1}$ is equal to $false$.

Node Index	*Formula*	*Proof witness*
(1)	$length(L, n)$	$b_1 = b_2$
...	...	...
(5.1.1)	$\forall i(false \Leftrightarrow \exists a(ielement(nil, i, a)))$	$b_{5.1.1} = true \wedge true$
...	...	...
(5.1.2)	$false \Leftrightarrow \exists a(ielement(conc(x, Y), i, a))$	$b_{5.1.2} = false \wedge b_{6.1.2.2}$
(6.1.2.1)	$false \Leftrightarrow \exists a(ielement(conc(x, Y), 0, a))$	$b_{6.1.2.1} = false \wedge b_{6.1.2.2}$
(6.1.2.2)	$false \Leftrightarrow \exists a(ielement(conc(x, Y), s(k), a))$	$b_{6.1.2.2} = b_{7.1.2.2}$
...	...	...

[1]Variables without quantification are considered as universally quantified variables

An example of (b) can be obtained of the synthesis of *perm* relation:

Node Index	*Formula*	*Proof witness*
(1)	$perm(L, S)$	$b_1 = b_2$
(2)	$nocc(a, L, z) \Leftrightarrow nocc(a, S, z)$	$b_2 = b_{3.1} \wedge b_{3.2}$
(3.1)	$nocc(a, nil, z) \Leftrightarrow nocc(a, S, z)$	$b_{3.1} = b_{4.1.1} \wedge b_{4.1.2}$
(3.2)	$nocc(a, conc(x, Y), z) \Leftrightarrow nocc(a, S, z)$	$b_{3.2} = b_{4.2.1} \wedge b_{4.2.2}$
(4.1.1)	$nocc(a, nil, z) \Leftrightarrow nocc(a, nil, z)$	$b_{4.1.1} = true$
(4.1.2)	$nocc(a, nil, z) \Leftrightarrow nocc(a, conc(v, W), z)$	$b_{4.1.2} = b_{5.1.2.1} \wedge b_{5.1.2.2}$
(4.2.1)	$nocc(a, conc(x, Y), z) \Leftrightarrow nocc(a, nil, z)$	$b_{4.2.1} = b_{5.2.1.1} \wedge b_{5.2.1.2}$
(4.2.2)	$nocc(a, conc(x, Y), z) \Leftrightarrow nocc(a, conc(v, W), z)$	$b_{4.2.2} = b_{5.2.2.1} \wedge b_{5.2.2.2}$
...	...	...

We observe that nodes (4.1.2) and (4.2.1) present variant formulas. Therefore, it is possible to reuse the synthesis of (4.1.2) for (4.2.1) and then $b_{4.2.1} = b_{4.1.2}$ preventing from the synthesis of (5.2.1.1) and (5.2.1.2).

4 Conclusion and Future Work

In constructive synthesis, although program extraction can be mechanized [3], producing the proofs remains a non trivial task and needs human interaction [7]. Therefore synthesizers construct programs with low effectiveness. To improve effectiveness by means of automatic proof construction two problems arise (a) synthesis loops and (b) redundant synthesis. For preventing synthesis loops, we adopt, in the context of a proof, an order between literals. Normally, this order guides the proof construction process producing subconjectures to be synthesize in subsequent synthesis steps. For example, a completed synthesis for *length* can be obtained in only one synthesis step, however, the synthesis of *perm* needs subsequent synthesis steps; the first step produces a partial constructions of the program. For example, the program for *perm* depends on subsequent synthesis of subconjectures just as $\forall a, W, k(false \Leftrightarrow nocc(a, W, k))$. In this sense, (automatic) constructive synthesis can be seen as a interactive process of stepwise refinement. On the other hand, the adoption of proof witnesses avoids redundant proofs because it is possible to establish partial evaluation on proofs in the synthesis process. The memorizing of subconjectures allows to reuse of synthesis subtrees avoiding redundant synthesis.

Our future work is centered on detecting subsumptions between conjectures in subsequent synthesis steps. For example, the synthesis of *perm* implies the synthesis of subconjectures just as (a) $\forall a, W, k(false \Leftrightarrow nocc(a, W, k))$ and (b) $\forall a, W, k(false \Leftrightarrow nocc(a, W, s(k)))$. Let $\forall a, W, k(sub_a(a, W, k) \Leftrightarrow (false \Leftrightarrow nocc(a, W, k)) \hookrightarrow b_a$ be a conjecture synthesis for (a) and let $\forall a, W, k(sub_b(a, W, s(k)) \Leftrightarrow (false \Leftrightarrow nocc(a, W, s(k)))) \hookrightarrow b_b$ be a conjecture synthesis for (b). However, it is not necessary to synthesize from these two conjectures because (a) subsumes (b). The synthesis of (a) constructs a program which is a generalization of the program for (b). Therefore, the program for (b) can be obtained as instantiation of the program for (a).

References

[1] A. Smaill A. Bundy and G. Wiggins. The synthesis of logic programs from inductive proofs. *Symposium on Computational Logic, pp 135-149. Springer-Verlag*, 1990.

[2] J. L. Bates and R. L. Constable. Proofs as programs. *ACM Transaction on Programming Languages and Systems 7(1)*, 1985.

[3] L. Fribourg. Extracting logic programs from proofs that use extended prolog execution and induction. *In Proceedings of the ICLP'90, pp 685-699. MIT Press*, 1990.

[4] I. Kraan y J. Hesketh G. Wiggins, A. Bundy. Synthesis and transformation of logic program from constructive, inductive proof. *En Proceedings del LOP-STR'91, Springer-Verlag*, 1992.

[5] F. J. Galan and M. Toro. Object oriented software systems defined by logical constructive method. *In proceedings of the Joint Conference on Declarative Programming Appia-Gulp-Prode, M. Falaschi, Vilares-Ferro and Freire-Nistal eds.*, 1998.

[6] D. Basin I. Kraan and A. Bundy. Middle-out reasoning for logic program synthesis. *In Proceedings of the ICLP'93, pp 441-455, MIT Press*, 1989.

[7] K. K. Lau Y. Deville. Logic program synthesis. *Journal of Logic Programming, 19-20. Elsevier Science*, 1994.