Partial Deduction in the Framework of Structural Synthesis of Programs

Mihhail Matskin, Jan Komorowski, John Krogstie
Department of Computer Systems
Norwegian University of Science and Technology
N-7033 Trondheim, Norway
{misha, janko}@idt.unit.no, john.krogstie@ac.com

Abstract. The notion of *partial deduction* known from logic programming is defined in the framework of *Structural Synthesis of Programs* (SSP). Partial deduction for unconditional computability statements in SSP is defined. Completeness and correctness of partial deduction in the framework of SSP are proven. Several tactics and stopping criteria are suggested.

1 Introduction

The main motivation for this work is to provide incremental formal program development in the framework of the proof-as-programs approach [3, 13]. In our case, it is Structural Synthesis of Programs [16, 20] (hence abbreviation SSP). In SSP, a specification at hand is transformed into a set of formulae of a logical language complete with respect to intuitionistic propositional logic. A problem to be solved is formulated as a theorem to be proven in this logic. A program is then extracted from the proof of the theorem. This approach has been implemented (see, for instance, [21]) and commercialized. It can be further strengthened by applying a principle similar to Partial Deduction as known in logic programming [7, 12, 9].

In this paper, we formulate the principle of partial deduction for SSP and prove its correctness and completeness in the new framework.

The rest of the paper is organized as follows. A brief informal introduction to SSP is given in Sect. 2. The next section presents partial deduction in SSP. Stopping criteria and tactics are discussed in Sect. 4. Concluding remarks are given in Sect. 5, where the use of partial SSP is briefly discussed. It is assumed that the reader is acquainted with partial deduction in logic programming.

2 Introduction to Structural Synthesis of Programs (SSP)

A formal foundation of SSP was developed by Mints and Tyugu [16, 20] and applied in a number of programming systems [21]. Here we give a brief introduction to the method and define notions which will be used later in the paper. First, the LL language of SSP is defined. Then inference rules are formulated

using sequent notation. Program extraction is discussed next. It is obtained by extending the LL language.

2.1 Logical language (LL)

The logical language of SSP has only the following kinds of formulae:

1. Propositional variables: A, B, C ....
   A propositional variable A corresponds to an object variable a from the source problem, and it expresses the fact that a value of the object variable can be computed. A propositional variable will be termed an *atom* in the following.
2. Unconditional computability statements:

   $$A_1 \& \ldots \& A_k \rightarrow B$$

   This expresses computability of the value of the object variable b corresponding to B from values of $a_1 \ldots a_k$ corresponding to $\underline{A}$ ($\underline{A}$ is an abbreviation for $A_1 \& \ldots \& A_k$). $\underline{A}$ will be termed the body of the statement, whereas B is termed the head. Thus *body* $(\underline{A} \rightarrow B) = \underline{A}$ and *head* $(\underline{A} \rightarrow B) = B$.
3. Conditional computability statements

   $$(\underline{A^1} \rightarrow B^1) \& \ldots \& (\underline{A^n} \rightarrow B^n) \rightarrow (\underline{C} \rightarrow D)$$

   A conditional computability statement such as $(A \rightarrow B) \rightarrow (C \rightarrow D)$ expresses computability of d from c depending on the computation of b from a. We also use $(\underline{A} \rightarrow B)$ as an abbreviation for $(\underline{A^1} \rightarrow B^1) \& \ldots \& (\underline{A^n} \rightarrow B^n)$. In the statement $(\underline{A} \rightarrow B) \rightarrow (\underline{C} \rightarrow D)$, $(\underline{A} \rightarrow B)$ will be termed the body of the statement, whereas $(\underline{C} \rightarrow D)$ will be termed the head. The functions *head* and *body* are defined as above, whereas $body(body((A \rightarrow B) \rightarrow (C \rightarrow D))) = A$ etc.

2.2 Structural Synthesis Rules (SSR)

A sequent notation is used for the derivation rules. A sequent $\Gamma \vdash X$, where Γ is a list of formulae and X is a formula, means that the formula X is derivable from the formulae appearing in Γ. Axioms have a form of $\Gamma, X \vdash X$.

The SSR inference rule are as follows.

- 1. $(\rightarrow$ -)

  $$\frac{\vdash \underline{A} \rightarrow V; \underline{\Gamma \vdash A}}{\underline{\Gamma} \vdash V}$$

  where $\underline{\Gamma \vdash A}$ is a set of sequents

- 2. $(\rightarrow +)$

  $$\frac{\Gamma, \underline{A} \vdash B}{\Gamma \vdash \underline{A} \rightarrow B}$$

– 3. $(\rightarrow - -)$

$$\frac{\vdash (\underline{A} \rightarrow B) \rightarrow (\underline{C} \rightarrow V); \underline{\Gamma, A \vdash B}; \underline{\Delta \vdash C}}{\underline{\Gamma, \Delta} \vdash V}$$

where $\underline{\Gamma, A \vdash B}$ and $\underline{\Delta \vdash C}$ are sets of sequents.

It is somewhat surprising but LL with simple implicative formulae is equivalent to the intuitionistic propositional calculus. The intuitionistic completeness of SSR was proven [16]. In addition, an experiment with a programming system based on SSR was done [22]. In this experiment, all intuitionistic propositional theorems (about 100 formulae) contained in [6] were proven automatically.

2.3 Program extraction

In order to be able to extract programs from proofs, the language LL is extended as follows.

We write

$$\underline{A} \underset{f}{\rightarrow} B$$

where f is a term representing the function which computes b from $a_1, \ldots, a_n$. Analogously, we write

$$\underline{(\underline{A} \underset{g}{\rightarrow} B)} \rightarrow (\underline{C} \underset{F(\underline{g},\underline{c})}{\longrightarrow} D)$$

where g is a term representing function which is synthesized in order to compute B from A and F is a term representing computation of D from C depending on $\underline{g}$ and $\underline{c}$ ($\underline{g}$ is a tuple of terms).

Following the notation in [20], $A(a)$ means that a is a term whose evaluation gives the value of the object variable corresponding to logical variable A. In the logical axioms which are written as $\Gamma, X(x) \vdash X(x)$, x is a variable or a constant.

The resulting language is termed $LL1$.

The modified inference rules (SSR1) are as follows:

– 1. $(\rightarrow -)$

$$\frac{\vdash \underline{A} \underset{f}{\rightarrow} V; \underline{\Gamma \vdash A(a)}}{\underline{\Gamma} \vdash V(f(\underline{a}))}$$

– 2. $(\rightarrow +)$

$$\frac{\Gamma, \underline{A} \vdash B(b)}{\Gamma \vdash \underline{A} \underset{\lambda \underline{a}.b}{\longrightarrow} B}$$

– 3. $(\rightarrow - -)$

$$\frac{\vdash \underline{(\underline{A} \underset{g}{\rightarrow} B)} \rightarrow (\underline{C} \underset{F(\underline{g},\underline{c})}{\longrightarrow} V); \underline{\Gamma, A \vdash B(b)}; \underline{\Delta \vdash C(c)}}{\underline{\Gamma, \Delta} \vdash V(F(\underline{\lambda a.b}, \underline{c}))}$$

A problem to be solved is formulated as a theorem to be proven. An example theorem and its proof are given in the next section.

2.4 A Specification Level

Whereas the $LL1$ language specifies the internal language for structural synthesis, higher level languages are defined for expressing problem specifications. One of these, the NUT language [21], is used in the sequel. It is an object-oriented language where methods and structural relations can be translated into formulae of the $LL1$ language.

The working example in this paper is a definition of the **inverter**, and-port and nand-port of logical circuits.

ELEMENT

```
var Delay:numeric;
```

INVERTER

```
super ELEMENT;
vir InINV, OutINV: bool;
rel inv: InINV -> OutINV
                  {OutINV := not InINV}
```

INVERTER is a subclass of **ELEMENT**, having two virtual boolean primitive type specifiers `InINV` and `OutINV`. Virtual components can be used in computations, but their values are not contained in the value of the object. The method `inv` specifies computations of `OutINV` from `InINV` which are performed by sequence of statements (body of the method) in the curly brackets. Actually, we can consider this method as a language construction for $InINV \underset{f}{\rightarrow} OutINV$ where f refers to the body of the method, i. e. to {`OutINV := not InINV`}.

AND

```
super ELEMENT;
vir InAND1, InAND2, OutAND: bool;
rel and: InAND1,InAND2->OutAND
                {OutAND:=InAND1 & InAND2}
```

NAND

```
super INVERTER;
super AND;
vir InNAND1, InNAND2, OutNAND: bool;
rel InAND1 = InNAND1; InAND2 = InNAND2;
    OutNAND = OutINV; InINV = OutAND;
nand: InNAND1, InNAND2 -> OutNAND{specification}
```

Symbol "=" denotes equality methods which are transformed into two formulae of $LL1$. For example, `InAND1 = InNAND1` is transformed into $InANDI \underset{asg}{\longrightarrow} InNANDI$ and $InNANDI \underset{asg}{\longrightarrow} InANDI$, where asg is the name of the standard function performing assignment. `specification` in the last method indicates that a function for the **nand** method has not been developed yet.

The interested reader can find a full description of NUT in [23].

The description of the class **NAND** can be now transformed into the following set of logical formulae (called problem-oriented axioms), where *and*, *inv* and *asg* indicate methods which implement the corresponding formulae:

$$\vdash InINV \underset{inv}{\longrightarrow} OutINV; \vdash InAND1 \& InAND2 \underset{and}{\longrightarrow} OutAND;$$
$$\vdash InAND1 \underset{asg}{\longrightarrow} InNAND1; \vdash InNAND1 \underset{asg}{\longrightarrow} InAND1;$$
$$\vdash InAND2 \underset{asg}{\longrightarrow} InNAND2; \vdash InNAND2 \underset{asg}{\longrightarrow} InAND2;$$
$$\vdash OutNAND \underset{asg}{\longrightarrow} OutINV; \vdash OutINV \underset{asg}{\longrightarrow} OutNAND;$$
$$\vdash InINV \underset{asg}{\longrightarrow} OutAND; \vdash OutAND \underset{asg}{\longrightarrow} InINV;$$

The theorem to be proven is $\vdash InNAND1, InNAND2 \rightarrow OutNAND$. Using the set of formulae and the inference rules from Sect. 2.2, the following proof can be made:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{InNAND1(in1) \vdash InNAND1(in1); \vdash InNAND1 \underset{asg}{\longrightarrow} InAND1;}}{InNAND2(in2) \vdash InNAND1(in2); InNAND2 \underset{asg}{\longrightarrow} InAND2}(\rightarrow -)}{InNAND2(in2) \vdash InAND(asg(in2))}}{InNAND1(in1) \vdash InAND1(asg(in1))}}{\vdash InAND1 \& InAND2 \underset{and}{\longrightarrow} OutAND}}{InNAND1(in1), InNAND2(in2) \vdash OutAND(and(asg(in1), asg(in2))}(\rightarrow -)}{\vdash OutAND \underset{asg}{\longrightarrow} InINV;}}{InNAND1(in1), InNAND2(in2) \vdash InINV(asg(and(asg(in1), asg(in2)))}(\rightarrow -)}{\vdash InINV \underset{inv}{\longrightarrow} OutINV;}}{InNAND1(in1), InNAND2(in2) \vdash OutINV(inv(asg(and(asg(in1), asg(in2)))}(\rightarrow -)}{\vdash OutINV \underset{asg}{\longrightarrow} OutNAND}}{InNAND1(in1), InNAND2(in2) \vdash OutNAND(asg(inv(asg(and(asg(in1), asg(in2))))}(\rightarrow -)}{\vdash InNAND1 \& InNAND2 \underset{\lambda in1\ in2.asg(inv(asg(and(asg(in1),asg(in2)))))}{\longrightarrow} OutAND}(\rightarrow +)$$

Q.E.D.

The extracted program is as follows

$$\lambda in1\ in2.asg(inv(asg(and(asg(in1), asg(in2))))))$$

or in a simplified form

$$\lambda in1\ in2.inv(and(in1, in2))$$

The example considered here is, of course, very small. The approach is, however, scalable. Problem specifications containing thousands of variables and computability statements have been developed [17].

3 Partial deduction in SSP

Partial deduction (a.k.a. partial evaluation in logic programming) is a specialization principle related to the law of syllogism [12, 8]. Our motivation for transferring Partial Deduction (PD) to the framework of SSP is based on the following observations:

- PD provides a specialization of logic programs. The most important features of PD is that more efficient programs can be generated. Efficiency for a residual program can be discussed in terms of the size of derivation trees.

  The same observation is true for SSP. Residual program specification can be more efficient in terms of synthesis of programs.

- Our experience with the NUT system shows that users are not always able to specify completely goals such as $\underline{A} \rightarrow G$. In many cases, the user knows/remembers only the input parameters of the problem and s/he does not care about the output parameters. In this case, s/he is satisfied with result that everything that is computable from inputs is computed. For example, the user may know that inputs of the problem are values for `InNAND1` and `InNAND2` of the `NAND` element and s/he would like to know which other values can be computed. This feature is supported by the program statement `compute`, e.g. `NAND.compute(InNAND1, InNAND2)`. This problem arises especially for specifications containing a large number of variables (very large circuits). These goals are hardly expressible in $LL1$. The problem is that one would take into account a set of all variables of the specification and all subsets of this set. Additionally, an order over the subsets has to be defined and possible output parameters considered as a disjunction of the subsets. This is a very inefficient approach.

  On the other hand PD allows us to make efficient derivations even if the output parameters of the problem are unknown. This issue will be discussed in Sect. 4.

- A problem similar to the above mentioned one, but with another type of incompleteness, may also arise. Consider a situation in which the user knows outputs of problem but s/he is not able to specify inputs. Actually, this relates mostly to debugging of specifications. If the original problem is not solvable, what should be added into the problem specification in order to make it solvable?

  Also in this case PD provides a good framework for making assumptions.

It is possible to apply PD at different levels in the framework of SSP:

- At the $LL1$ specification.
- At the resulting functional expressions.
- At the meta-specification.

Here we consider the first of the above cases and, in particular, the unconditional computability statements of $LL1$. The second case has been studied elsewhere and by numerous authors, see, for instance, [2, 1, 5]. Work has also been done on applying PD at the meta-specification level [4, 15].

Before moving further on we formalize the notion PD in the SSP framework.

3.1 Partial deduction of $LL1$ specifications

It is natural to informally express PD of unconditional computability statements as follows.

Let A, B and C be propositional variables and $A \underset{f}{\rightarrow} B, B \underset{h}{\rightarrow} C, C \underset{g}{\rightarrow} D$ be unconditional computability statements. Some possible partial deductions are the following: $A \xrightarrow[\lambda a.h(f(a))]{} C$, $A \xrightarrow[\lambda a.g(h(f(a)))]{} D$, $B \xrightarrow[\lambda b.g(h(b))]{} D$. It is easy to notice that the first partial deduction corresponds to forward chaining (from inputs to outputs), the third one corresponds to backward chaining (from outputs to inputs) and the second one to either forward or backward chaining.

The main difference between our definition of PD and the ones considered in [12, 8, 9] are: (i) the use of logic specifications rather then logic programs and (ii) the use of intuitionistic (propositional) logic rather then classical (first order) logic.

Definition 1 ($LL1$ specification) *Let $\mathcal{U}$ be a set of unconditional computability statements $\underline{A} \underset{f}{\rightarrow} B$ and $\mathcal{S}$ be a set of conditional computability statements $(\underline{A} \underset{g}{\rightarrow} B) \rightarrow (\underline{C} \xrightarrow[F(\underline{g},\underline{c})]{} D)$ then $P = \mathcal{U} \cup \mathcal{S}$ is called a $LL1$* **specification.**

Definition 2 ($LL1$ unconditional resultant) *An $LL1$* **unconditional resultant** *is an unconditional computability statement $\underline{A} \xrightarrow[\lambda \underline{a}.f(a)]{} G$ where f is a term representing the function which computes G from potentially composite functions over $a_1, \ldots, a_n$*

The assumption that G is an atom can be relaxed by transformation of $\underline{A} \underset{f}{\rightarrow} \underline{G}$ into $\{\underline{A} \underset{f}{\rightarrow} G_1, ..., \underline{A} \underset{f}{\rightarrow} G_m\}$ and vice versa.

In our definitions we use the notion of computation rule (or selection function) from logic programming [11].

Definition 3 (Derivation of an $LL1$ unconditional resultant) *Let $\Re$ be a fixed computation rule. A* **derivation of a $LL1$ unconditional resultant** *R_0 is a finite sequence of $LL1$ resultants: $R_0 \Rightarrow_{\Re} R_1 \Rightarrow_{\Re} R_2 \Rightarrow_{\Re} \ldots$, where,*

(1) for each j, R_j is an unconditional computability statement of the form

$$\underline{B_1}\&...\&B_i\&...\&\underline{B_n} \xrightarrow[\lambda \underline{b_1}...\underline{b_i}...\underline{b_n}.f(\underline{b_1},...,\underline{b_i},...,\underline{b_n})]{} G$$

and R_{j+1} (if any) is of the form:

$$\underline{B_1}\&...\&\underline{C}\&...\&\underline{B_n} \xrightarrow[\lambda \underline{b_1}...\underline{c}...\underline{b_n}.f(\underline{b_1},...,h(\underline{c}),...\underline{b_n})]{} G$$

if

- *the $\Re$-selected atom in R_j is B_i and*
- *there exists an unconditional computability statement $\underline{C} \underset{h}{\rightarrow} B_i \in P$*

called the matching computability statement. B_i is called the matching atom.

(2) for each j, R_j is an unconditional computability statement of the form

$$\underline{A_1}\&...\&\underline{A_i}\&...\&\underline{A_m} \xrightarrow[\lambda \underline{a_1}...\underline{a_i}...\underline{a_m}.f(\underline{a_1},...,\underline{a_i},...,\underline{a_m})]{} F$$

and R_{j+1} (if any) is of the form:

$$\underline{A_1}\&...\&\underline{A_i}\&...\&\underline{A_m} \xrightarrow[\lambda \underline{a_1}...\underline{a_i}...\underline{a_m}.h(\underline{a_i},f(\underline{a_1},...,\underline{a_i},...,\underline{a_m}))]{} H$$

if

- *the $\Re$-selected atoms in R_j are $\underline{A_i}$ and/or $\underline{F}$. $\underline{F}$ denotes a conjunction of F (heads of computability statements) from resultants $R_0, ..., R_j$*
- *there exists an unconditional computability statement $\underline{A_i}\&\underline{F} \underset{h}{\rightarrow} H \in P$*

called the matching computability statement. A_i, F are called the matching atoms.

(3) If R_{j+1} does not exist, R_j is called a leaf resultant.

Definition 3 gives two ways of derivating resultants. (1) corresponds to backward derivations and (2) corresponds to forward derivations. We will use terms derivation form (1) and derivation form (2), correspondingly.

Definition 4 (Partial deduction of a propositional goal) *A* **partial deduction of a propositional goal** *$\underline{A} \rightarrow G$ in a specification P is the set of leaf resultants derived from $G \rightarrow G$ (derivation form (1) and in this case $\underline{A}$ cannot be selected atoms) or from $\underline{A} \rightarrow$ (derivation form (2) and in this case G cannot be selected atom).*

The assumption that G is an atom can be relaxed to allow a conjunction of atoms, $\underline{A} \rightarrow \underline{G}$, and a partial deduction of $\underline{A} \rightarrow \underline{G}$ in P is the union of partial deductions of $\mathbf{G} = \{\underline{A} \rightarrow G_1, \ldots, \underline{A} \rightarrow G_m\}$ in P. These partial deductions are called *residual unconditional computability statements.*

Definition 5 (Partial deduction of an $LL1$ specification) *A* **partial deduction of a $LL1$ specification** *P wrt* **G** *is a specification P' (also called a residual $LL1$ specification) obtained from P by replacing computability statements having G_i in their heads (derivation form (1)) or only variables from $\underline{A}$ in their bodies (derivation form (2)) by corresponding partial deductions.*

The notions of correctness and completeness are defined as follows. Let P be a $LL1$ specification, $\underline{A} \to G$ a propositional goal, and P' a partial deduction of P wrt to $\underline{A} \to G$.

Definition 6 (Correctness of partial deduction of $LL1$ specification) *The function (or its computational equivalent) for the computation of $\underline{A} \to G$ is derivable (can be extracted from a proof of $\underline{A} \to G$) from P if it is derivable from P'.*

Completeness is the converse:

Definition 7 (Completeness of partial deduction of $LL1$ specifications) *The function (or its computational equivalent) for the computation of $\underline{A} \to G$ is derivable (can be extracted from a proof of $\underline{A} \to G$) from P' if it is derivable from P.*

Our definitions of correctness and completeness require only derivation of a function which implements the goal (all functions which satisfy this criteria are computational equivalents). This may correspond, for example, to the following case. Let an $LL1$ *specification* contains computability statements $\underline{C} \underset{h}{\to} B$ and $\underline{C} \underset{f}{\to} B$. Then functions f and h are computational equivalents.

The notions of correctness and completeness are defined with respect to the possibility of proving the goal and extracting a function from the proof [10]. Our proof of correctness and completeness is based on proving that derivation of $LL1$ unconditional resultant is a derivation (proof + program extraction) in a calculus corresponding to the $LL1$ language.

Lemma 1 *Derivation form (1) of $LL1$ unconditional resultants is a derivation by SSR1 inference rules.*

Proof Consider the case, when

$$R_j = \underline{B_1}\&...\&B_i\&...\&\underline{B_n} \xrightarrow[\lambda \underline{b_1}...\underline{b_i}...\underline{b_n}.f(\underline{b_1},...,\underline{b_i},...,\underline{b_n})]{} G$$

or in a short form

$$R_j = \underline{B_1}\&B_2\&\underline{B_3} \xrightarrow[\lambda \underline{b_1}b_2\underline{b_3}.f(\underline{b_1},b_2,\underline{b_3})]{} G$$

and the matching computability statement is $\underline{C} \underset{h}{\to} B_2$

According to **Definition 3** the $LL1$ unconditional resultant will be

$$\underline{B_1}\&\underline{C}\&\underline{B_3} \xrightarrow[\lambda \underline{b_1}\underline{c}\underline{b_3}.f(\underline{b_1},h(\underline{c}),\underline{b_3})]{} G$$

R_j and the matching computability statement have the form of the following problem-oriented axioms in the calculus for $LL1$ language:

$\vdash \underline{B_1}\&B_2\&\underline{B_3} \xrightarrow[\lambda \underline{b_1}b_2\underline{b_3}.f(\underline{b_1},b_2,\underline{b_3})]{} G$

$\vdash \underline{C} \underset{h}{\to} B_2$

Given these axioms, the following derivation is obtained by SSR1 inference rules:

$$\frac{\underline{C(c) \vdash C(c)}; \quad \underline{B_1(b_1) \vdash B_1(b_1)}; \quad \underline{B_3(b_3) \vdash B_3(b_3)}; \quad \underline{C} \underset{h}{\rightarrow} B_2;}{\underline{C(c)} \vdash B_2(h(\underline{c})); \quad \vdash \underline{B_1}\&B_2\&\underline{B_3} \xrightarrow[\lambda \underline{b_1} b_2 \underline{b_3}.f(\underline{b_1},b_2,\underline{b_3})}{} G;}(\rightarrow -)$$

$$\frac{}{\underline{B_1(b_1)}, \underline{C(c)}, \underline{B_3(b_3)} \vdash G((\lambda \underline{b_1} b_2 \underline{b_3}.f(\underline{b_1},\underline{b_2},\underline{b_3}))(\underline{b_1}, h(\underline{c}), \underline{b_3}))}(\rightarrow -)$$

$$\frac{}{\vdash \underline{B_1}\&\underline{C}\&\underline{B_3} \xrightarrow[\lambda \underline{b_1} c \underline{b_3}.((\lambda \underline{b_1} b_2 \underline{b_3}.f(\underline{b_1},\underline{b_2},\underline{b_3}))(\underline{b_1},h(\underline{c}),\underline{b_3}))}{} G}(\rightarrow +)$$

After a simplification of the λ-expression under the arrow in the last sequent we end up with:

$$\underline{B_1}\&\underline{C}\&\underline{B_3} \xrightarrow[\lambda \underline{b_1} c \underline{b_3}.f(\underline{b_1},h(\underline{c}),\underline{b_3})]{} G$$

which is equal to the $LL1$ unconditional resultant.

The above derivation is a derivation of the inference rule which corresponds to the PD step:

$$\frac{\vdash \underline{B_1}\&B_2\&\underline{B_3} \xrightarrow[\lambda \underline{b_1} b_2 \underline{b_3}.f(\underline{b_1},b_2,\underline{b_3})]{} G; \quad \vdash \underline{C} \underset{h}{\rightarrow} B_2}{\vdash \underline{B_1}\&\underline{C}\&\underline{B_3} \xrightarrow[\lambda \underline{b_1} c \underline{b_3}.f(\underline{b_1},h(\underline{c}),\underline{b_3})]{} G}$$

If the unconditional computability statements are in the form of resultants and matching statements (see **Definition 3**), then a derivation in the calculus corresponding to the LL1 language will be done in accordance to the derived inference rule.

Q.E.D.

Lemma 2 *Derivation form (2) of LL1 unconditional resultants is a derivation by SSR1 inference rules.*

Proof The proof of **Lemma 2** is analogous to the proof of **Lemma 1**. The only difference is in the derivation of the inference rule for PD. We describe only the essential part of the proof and omit the part of the proof which is common with the previous proof.

R_j and matching computability statements in calculus for $LL1$ language have the form of the following problem-oriented axioms:

$$\vdash \underline{A_1}\&\underline{A_2}\&\underline{A_3} \xrightarrow[\lambda \underline{a_1} \underline{a_2} \underline{a_3}.f(\underline{a_1},\underline{a_2},\underline{a_3})]{} F$$

$$\vdash \underline{A_2}\&F \underset{h}{\rightarrow} H$$

Hence, the following derivation is obtained by SSR1 inference rules:

$$\underline{A_1(a_1) \vdash A_1(a_1)};\ \underline{A_2(a_2) \vdash A_2(a_2)};\ \underline{A_3(a_3) \vdash A_3(a_3)};$$

$$\frac{\underline{A_1}\&\underline{A_2}\&\underline{A_3} \xrightarrow[\lambda \underline{a_1 a_2 a_3}.f(\underline{a_1},\underline{a_2},\underline{a_3})]{} F}{\underline{A_1(a_1)}, \underline{A_2(a_2)}, \underline{A_3(a_3)} \vdash F((\lambda \underline{a_1 a_2 a_3}.f(\underline{a_1},\underline{a_2},\underline{a_3}))(\underline{a_1},\underline{a_2},\underline{a_3}));\vdash \underline{A_2}\&F \underset{h}{\rightarrow} H}(\rightarrow -)$$

$$\frac{}{\underline{A_1(a_1)}, \underline{A_2(a_2)}, \underline{A_3(a_3)} \vdash H(h(\underline{a_2}, (\lambda \underline{a_1 a_2 a_3}.f(\underline{a_1},\underline{a_2},\underline{a_3}))(\underline{a_1},\underline{a_2},\underline{a_3})))}(\rightarrow -)$$

$$\frac{}{\vdash \underline{A_1}\&\underline{A_2}\&\underline{A_3} \xrightarrow[\lambda \underline{a_1}\lambda \underline{a_2}\lambda \underline{a_3}\ h(\underline{a_2},(\lambda \underline{a_1 a_2 a_3}.f(\underline{a_1},\underline{a_2},\underline{a_3}))(\underline{a_1},\underline{a_2},\underline{a_3}))]{} F}(\rightarrow +)$$

A simplification of the λ-expression in the last sequent leads to

$$\underline{A_1}\&\underline{A_2}\&\underline{A_3} \xrightarrow[\lambda \underline{a_1 a_2 a_3}.h(\underline{a_2},f(\underline{a_1},\underline{a_2},\underline{a_3}))]{} F$$

which is equal to the $LL1$ unconditional resultant.

The above derivation is a derivation of the inference rule which corresponds to the PD step:

$$\frac{\vdash \underline{A_1}\&\underline{A_2}\&\underline{A_3} \xrightarrow[\lambda \underline{a_1 a_2 a_3}.f(\underline{a_1},\underline{a_2},\underline{a_3})]{} F;\quad \vdash \underline{A_2}\&F \underset{h}{\rightarrow} H}{\vdash \underline{A_1}\&\underline{A_2}\&\underline{A_3} \xrightarrow[\lambda \underline{a_1 a_2 a_3}.h(\underline{a_2},f(\underline{a_1},\underline{a_2},\underline{a_3}))]{} H}$$

In case of a conjunction $\underline{F}$ in the matching computability statement, a set of resultants R_j $(0 > \mathrm{j} > \mathrm{j}{+}1)$ for all F_i from $\underline{F}$ is considered.

Q.E.D.

Theorem 1 (Correctness and Completeness of PD in $LL1$ specification) *Partial Deduction of a $LL1$ specification is correct and complete.*

Proof of the theorem immediately follows from **Lemma 1** and **Lemma 2**. Since PD is a proof by SSR1 inference rules proof of $\underline{A} \rightarrow G$ in P' is a subproof of $\underline{A} \rightarrow G$ in P where paths to resultants are replaced by the derived resultants themselves. It means that both specifications P and P' are equal with respect to derivation of the goal and extraction of a program.

3.2 Partial deduction of conditional computability statements

Until now we have considered only unconditional computability statements as sources for PD. In case of conditional computability statements, the main framework for definitions is the same. However, particular definitions become more complicated. Here we do not consider all definitions for PD of conditional statements but rather point out most important moments.

Definition 8 (*LL*1 **conditional resultant**) *A LL*1 **conditional resultant** *is a conditional computability statement* $\underline{(D \to E)} \to \underline{A} \xrightarrow[\lambda \underline{a}\underline{g}.F(\underline{a},\underline{g})]{} G$ *where F is a term representing the function which computes G from potentially composite functions over* $a_1, \ldots, a_n$, $g_1, \ldots, g_n$ ($g_1, \ldots, g_n$ *are terms for all* $\underline{(D \to E)}$).

We would like to notice that the main difference in this case is the treatment of nested implications.

There are several possibilities to apply conditional computability statements in PD. Conditional resultant can be a matching statement for derivation of unconditional resultant as follows.

Definition 9 (Derivation of an *LL*1 **unconditional resultant)** *[with conditional computability statements]. Let* $\Re$ *be a fixed computation rule. A* **derivation of an** *LL*1 **unconditional resultant** R_0 **with conditional computability statements** *is a finite sequence of LL*1 *resultants:* $R_0 \Rightarrow_{\Re} R_1 \Rightarrow_{\Re} R_2 \Rightarrow_{\Re} \ldots$, *where,*

(4) if R_j *is an unconditional computability statement of the form*

$$\underline{B_1}\&\ldots\&\underline{B_i}\&\ldots\&\underline{B_n} \xrightarrow[\lambda \underline{b_1}\ldots\underline{b_i}\ldots\underline{b_n}.f(\underline{b_1},\ldots,\underline{b_i},\ldots,\underline{b_n})]{} G$$

then R_{j+1} *(if any) is of the form:*

$$\underline{(D \to E)} \to (\underline{B_1}\&\ldots\&\underline{C}\&\ldots\&\underline{B_n} \xrightarrow[\lambda \underline{g}\underline{b_1}\ldots\underline{c}\ldots\underline{b_n}.f(\underline{b_1},\ldots,h(\underline{g},\underline{c}),\ldots,\underline{b_n})]{} G)$$

if

- *the* $\Re$*-selected atom in* R_j *is* B_i *and*
- *there exists a conditional computability statement*

$$\underline{(D \underset{g}{\to} E)} \to (\underline{C} \xrightarrow[h(\underline{g},\underline{c})]{} B_i) \in P$$

called the matching computability statement. B_i *is called the matching atom.*

(5) if R_j *is an unconditional computability statement of the form*

$$\underline{A_1}\&\ldots\&\underline{A_i}\&\ldots\&\underline{A_m} \xrightarrow[\lambda \underline{a_1}\ldots\underline{a_i}\ldots\underline{a_m}.f(\underline{a_1},\ldots,\underline{a_i},\ldots,\underline{a_m})]{} F$$

then R_{j+1} *(if any) is of the form:*

$$\underline{(D \to E)} \to (\underline{A_1}, \ldots, \underline{A_i}, \ldots, \underline{A_m} \xrightarrow[\lambda \underline{g}\underline{a_1}\ldots\underline{a_i}\ldots\underline{a_m}.h(\underline{g},\underline{a_i},\underline{f(a_1,\ldots,a_i,\ldots,a_m)})]{} H)$$

if

- *the $\Re$-selected atoms in R_j are $\underline{A_i}$ and/or $\underline{F}$. $\underline{F}$ denotes a conjunction of F (heads of computability statements) from resultants $R_0, ..., R_j$*
- *there exists a conditional computability statement*

$$\underbrace{(\underline{D} \to E)}_{g} \to (\underline{A_i}, \underline{F} \xrightarrow[h(\underline{g},\underline{a_i},\underline{f})]{} H) \in P$$

called the matching computability statement. A_i, F are called the matching atoms.

Derivation of conditional resultants by unconditional computability statements can be considered as separate derivations of the head and body of conditional statements. Derivation of heads is similar to derivations of unconditional resultants. However, derivation of bodies is more complicated. We remind that bodies of conditional statements describe a function (program) to be synthesized. Actually, they are subgoals of a general goal. In our case, PD of a goal (subgoal) does not make sense (c.f. Definition 4 where $\underline{A}$ and G cannot be selected atoms in derivations form (1) and (2) correspondingly). The only way to apply PD in this case is to have an ordered set of derivations with respect to the goal and the subgoals.

This is not the only problem with PD of bodies of conditional statements. Assumptions from the head of the conditional statement can be used during derivation of subgoals. For example, if we have a conditional resultant

$$\underline{(D \to E)} \to (\underline{A_1}\&...\&\underline{A_i}\&...\&\underline{A_m} \xrightarrow[\lambda \underline{g a_1}...\underline{a_i}...\underline{a_m}.h(\underline{g},\underline{a_i},f(\underline{a_1},...,\underline{a_i},...,\underline{a_m}))]{} H)$$

then the proof of $\underline{(D \to E)}$ may use $\underline{A_1}, ..., \underline{A_i}, ..., \underline{A_m}$ as assumptions. This causes a more complex form of resultants.

Unfortunately, derivation of conditional resultants with conditional computability statements cannot be defined in our framework. In this case we would have come up with more then one level of nested implications in the resulting formulae. This is out of the scope of the language defined in the Sect. 2.

Correctness and completeness for derivations of resultants with conditional computability statements are defined similarly to those in Sect. 3., this subject is out of the scope of this paper.

4 Partial Deduction tactics

PD is a very simple principle and its practical value is limited without defining appropriate strategies. These are called tactics and refer to the selection and stopping criteria. We describe them informally.

4.1 Selection criteria

Selection functions (computation rules) define which formula should be tried next for derivation of resultant. We consider the following possible selection criteria.

1. *Unconditional computability statements are selected first and if there is no unconditional computability statements that can be applied to derivation of resultant then a conditional statement (if any) is selected.*

   This is the main selection criterion which is implemented in the NUT system. This criterion keeps simplicity of derivation when it is possible. However it does not say which unconditional/conditional statement to choose next if more then one statement is applicable at the same time.

2. *Priority based selection criteria.*

   These criteria require binding priorities to computability statements. They should be done by user. Synthesis with priorities was considered in some work on SSP. The most reasonable base for priorities is an estimated computational cost of function/program which implements computations specified by computability statements. Whenever it is possible to obtain such costs, the criterion allows to synthesize more efficient program. This criterion allows to make decision when the same objects can be computed by different functions. Until now the criterion was implemented only experimentally.

3. *Only a specified set of computability statements is used in the derivation.*

   This criterion requires partitioning of the set of statements. It is not realistic to ask the user to point out particular computability statements which could be used in derivation. However, such partitioning can be done on the level of the specification language. For example, the user can specify that s/he would like to perform computations on objects of class NAND only and then only computability statements derived from this class description are considered for the derivations. This criterion is implemented and widely used in the NUT system and it is a feature of object-orientation rather than of the logical part of the system (message sending to an object means that the context of computations/ derivations is restricted by this object).

4. *A mixture of forward and backward derivations.*

   This is quite interesting but a less investigated criterion. The idea is to develop derivation forms (1) and (2) (see Sect. 2) concurrently or in parallel. It means that computability statements are selected to support derivations of both resultants in some order. It can be strict alteration or some other criterion. We think that such criteria should be investigated in more detail in the

context of distributed computations. Another application of this approach was investigated in the context of debugging of specifications [14].

We would like to notice that the above criteria are not mutually exclusive but rather complementary to each other.

4.2 Stopping criteria

Stopping criteria define when to stop derivation of resultants. They can be combined freely with all the selection criteria above. We suggest the following stopping criteria.

1. *Stepwise*: The user is queried before each derivation which of the possible derivations s/he wants to perform. Actually, this stopping criterion can be of interest in debugging and providing the user with traces of the derivation process.
2. *Restricted*: A goal is given together with an indicator of the maximum depth of the derivation. This criteria is of a similar virtue as the previous one.
3. *Goal-based:* A derivation stops when resultant is equal to the goal. This criterion allows to synthesize a program when the goal is completely defined. A more interesting case is specialization of this criterion to a mixture of forward and backward derivations. In this case, a derivation can stop when bodies of resultants of derivation form (1) (backward derivation) are included in the union of bodies and heads of resultants of derivation form (2) (forward derivation).
4. *Exhaustive:* A derivation stops when no new resultants are available. This is the case when goals are not specified completely (Sect. 3) or when problem is not solvable (derivation of the goal can not be done).

The last stopping criteria is very important for PD and makes possible deriving programs from incompletely specified goals. Notice that the *Goal – based* criterion allows to derive programs in case of completely specified goals.

5 Concluding remarks

The notion of partial deduction has been transferred from logic programming to the framework of SSP. Our main results are a definition of partial deduction in this framework, a proof of completeness and correctness of PD in the framework and a set of selection criteria for utilizing them together with a set of stopping criteria.

In addition to the theoretical interest partial SSP defines a method for synthesis of programs in the case of incompletely specified goal. One specific application of the partial deduction technique in connection with SSP is to support debugging of $LL1$-specifications. In case of a non-solvable problem (a propositional goal cannot be derived) PD of an $LL1$ specification contains the set of

all possible derivations and is a good starting point for reasoning about possible inconsistency and/or incompleteness of the specification [14].

This paper has focused on partial deduction of the unconditional computability statements. We have only pointed out how the approach can be extended to the case of conditional computability statements. This and tactics supporting a mixture of backward and forward derivations of resultants will be investigated in future.

References

1. M. Z. Ariola and Arvind. A syntactic approach to program transformation. In *Proceedings of the symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'91)*, pages 116–129. ACM press, 1991.
2. D. Bjørner, A. P. Ershov, and N. D. Jones, editors. *Partial Evaluation and Mixed Computation*, Gammel Avernæs, October 18–24 1987. North-Holland.
3. R.L. Constable, S.F. Allen, H.M. Bromley et al. Implementing mathematics with the Nurpl Proof development system Prentice-Hall, 1986.
4. H-M. Haav and M. Matskin. Using partial deduction for automatic propagation of changes in OODB. In H. Kangassalo et al, editor, *Information Modeling and Knowledge Bases IV*, pages 339–352. IOS Press, 1993.
5. N. D. Jones, C. K. Gomard, and P. Sesoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, Englewood Cliffs, NJ, 1993.
6. S. Kleene. Introduction to metamathematics. Amsterdam, North-Holland, 1952.
7. J. Komorowski. *A Specification of An Abstract Prolog Machine and Its Application to Partial Evaluation.* PhD thesis, Department of Computer and Information Science, Linköping University, Linköping, Sweden, 1981.
8. J. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language: a theory and implementation in the case of Prolog. Proc. of the ACM Symp. Principles of Programming Languages, ACM, pp. 255–267, 1982.
9. J. Komorowski. A Prolegomenon to partial deduction. *Fundamenta Informaticae*, 18(1):41–64, January 1993.
10. I. Krogstie and M. Matskin. Incorporating partial deduction in structural synthesis of program. Technical Report 0802-6394 5/94, IDT, NTH, Trondheim, Norway, June 1994.
11. J. W. Lloyd. Foundations of logic programming. Springer Verlag, second edition, 1987.
12. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. , Journal of Logic Programming, 1991:11:217-242, also: Technical Report CS-87-09 (revised 1989), University of Bristol, England, July 1989.
13. Z. Manna, R. Waldinger. A Deductive approach to program synthesis. ACM Trans. on Programming Languages and Systems, 2(1):294:327, Jan, 1980.
14. M. Matskin. Debugging in programming systems with structural synthesis of programs (in Russian). *Software*, 4:21–26, 1983.
15. M. Matskin and J. Komorowski. Partial deduction and manipulation of classes and objects in an object-oriented environment. In *Proceedings of the First Compulog-Network Workshop on Programming Languages in Computational Logic*, Pisa, Italy, April 6-7 1992.

16. G. Mints and E. Tyugu. Justification of structural synthesis of programs. *Science of Computer Programming*, 2(3):215–240, 1982.
17. J. Pahapill. Programmpaket zur modeliering der hydromachinen systeme. 6. Fachtagung Hydraulik und Pneumatik, Magdeburg, pp. 609-617, 1985.
18. D. A. Schmidt. Static properties of partial evaluation. In Bjørner et al. [2], pages 465–483.
19. E. Tyugu. The structural synthesis of programs. In *Algorithms in Modern Mathematics and Computer Science*, number 122 in Lecture Notes in Computer Science, pages 261–289, Berlin, 1981. Springer-Verlag.
20. E. Tyugu. *Knowledge-Based Programming.* Turing Institute press, 1988.
21. E. Tyugu. Three new-generation software environments. *Communications of the ACM*, 34(6):46–59, June 1991.
22. B. Volozh, M. Matskin, G. Mints, E. Tyugu. Theorem proving with the aid of program synthesizer Cybernetics, 6:63-70, 1982.
23. T. Uustalu, U. Kopra, V. Kotkas, M. Matskin and E. Tyugu. The NUT Language Report. The Royal Institute of Technology (KTH),TRITA-IT R 94:14, 51 p., 1994.