STUDIA UNIV. "BABEŞ–BOLYAI", INFORMATICA, Volume XLIII, Number 2, 1998

WELL-TYPEDNESS VERIFICATION IN LOGIC PROGRAMMING WITH TYPES

DOINA TĂTAR AND GABRIELA ŞERBAN

ABSTRACT. In this paper we show how the relationship between attribute grammars and logic programs, established in [6], [7], can be extended to logic programs with type. In this order the concept of " well-typedness " is introduced. The examples for typed logic programs are done in Turbo Prolog and one application for well-typedness verification in Pascal 7.0 is supplyed.

1. Introduction

As observed by Deransart and Maluszynski [6], [7], there is an intimate relationship between the notion of a definite program and the notion of an attribute grammar, because a proof tree [4] of a definite program can be seen as a parse tree of this grammar. This observation made possibly a transfer of methods from the area of attribute grammars to the area of logic programming, as a complementary approach to the more utilised perspective of the first-order logic and theory resolution. The paper studies the attribute dependency scheme related with the logic programs with types by a suitable modification of the methods introduced in [6], [7]. The notion of well-typedness is applied to Turbo Prolog programs and a procedure (realised in Pascal) to check up the well-typedness is developed.

2. Definite clause programs

According to [1], [2], [3], a *definite clause* is a finite set of atomic formulas (atoms) $\{h, a_1, \cdots, a_n\}$ written as

$$h \leftarrow a_1, \cdots, a_n.$$

If $n = 0$ then the formula is called a *unit clause* or a *fact*. The language considered here is essentially that of the first-order predicate logic. Let:

- P be a set of predicates.
- F be a set of functors.
- C be a set of constants.
- V be a set of variables.

An atom is of the form

$$p(u_1, \cdots, u_n), n \geq 0$$

where $p \in P$ and his arrity is n. Each u_j is a term over $C \cup V \cup F$.

Definition 2.1

A definite clause program P (shortly DCP) is a sequence of definite clauses, $h \leftarrow a_1, ..., a_n$ where h and $a_1, ..., a_n$ are atomic formulas, the comma is the logic operation "and", and the sign $\leftarrow$ is "if" or reverse of the logical implication.

We refer to the left (h) and right-hand side ($a_1, ..., a_n$) of a clause as its *head* and *body*. A clause is logically interpreted as the universal closure of the implication $a_1 \wedge \cdots \wedge a_n \rightarrow h$.

Let us observe that this definition considers only the class of positive logic programs(all atoms in all clauses are positive).

Definition 2.2 A goal g consists of a conjunction of atoms, and is denoted by: $\leftarrow b_1, \cdots, b_t$

3. Attribute grammars and attribute dependency schemata

The original motivation for introducing attribute grammars has been to simplify compiler specification and construction [10]. This idea brings a declarative notion, as a parse tree of a context- free grammar, with the operational notion of dependency relation. More exactly, we give a general definition :

Definition 3.1 [7] A relational attribute grammar is a 7-tuple

$$G = (N, P, S, Attr, L, \phi, I)$$

where:

•N is a finite set of nonterminal symbols .

•P is a set of context-free production rule.

•S is a set of sorts.

•$Attr$ is a finite set of attributes, such that each nonterminal X has associated a set of attributes $Attr(X)$, each attribute a has a sort $s(a) \in S$, and each attribute occurence of a has also the sort $s(a)$.

•L is an S-sorted logical language, whose variables include all attribute occurrences of attributes in $Attr$.

•ϕ is an assignment of a logic formula ϕ_r of L to each production rule r in P. The free variables of ϕ_r are attribute occurrence.

•I is an interpretation of L.

A *functional* attribute grammar is a *relational* attribute grammar such that the set $Attr$ is the union of two disjoint sets: Inh (inherited attributes) and Syn (synthesized attributes) for each noonterminal X. For a production rule r the set of the attribute occurrences is partitioned into

• the set of input attributes, $Input(r)$, which contains the attribute inherited by the nonterminal head of the rule r, and the attributes synthesized by the nonterminals in the body of the rule r, and

• the set of output attributes, $Output(r)$, which are obtained in opposites cases.

We denote by $Pos(r)$ the set $Input(r) \cup Output(r)$ The formula ϕ_r is a conjunction of the form

$$\bigwedge_{w \in Output(r)} \omega = t_\omega$$

where t_ω are terms of L whose only variables are elements of $Attr(r)$.

With every r in P is associated a binary relation of dependency D_r . The family of binary relation defines the notion of *attribute dependency scheme* (ADS).

Definition 3.2 [6] An ADS is a 4-tuple S=$(N, P, Attr, D)$ where $N, P, Attr$ are defined as in definition 3.1 and D is a family of binary relations $\{D_r\}, r \in P$ defined on $Pos(r)$, such that

$$\{x | y D_r x, y \in Pos(r)\} \subseteq Output(r).$$

4. Logic programs and associated attribute grammars

Attribute grammars and definite programs can be compared with respect to the declarative semantics. The study of dependency relation for definite programs is an abstraction related to information flow through the parameters of the predicates.

Definition 4.1 [7] For a given definite program P the relational attribute grammars G is defined to be the 7-tuple G= $(N, P, S, Attr, L, \phi, I)$ where:

•N is the set of predicates symbols .

•P is a set of context-free production rule of the form:

$$p_0 \rightarrow p_1, \cdots, p_m$$

iff

$$p_0(t_{01}, \cdots, t_{0n_0}) \leftarrow p_1(t_{11}, \cdots, t_{1n_1}), \cdots, p_m(t_{m1}, \cdots, t_{mn_m})$$

is a clause c of P.

•S is a singleton

•$Attr$ is a finite set of attributes, denoting the arguments of predicates. The j-th argument of the predicate p corresponds to the attribute denoted pj.

•L is a first-order logical language, whose variables include all attribute occurrences of attributes of $Attr$.

•ϕ is an assignment of a logic formula ϕ_r of L to each production rule r in P. The free variables of ϕ_r are attribute occurences of r. If the rule r is associated with a clause c of the above form, then the formula ϕ_r is the following formula:

$$\exists V(c) \bigwedge_{k=0}^{m} \bigwedge_{j=1}^{n_k} (pj(k) = t_{kj})$$

where $\exists V(c)$ denotes existential quantification over all variables of the clause c , $pj(k)$ is the k-th occurrence of the attribute pj of the nonterminal p, n_k is the arrity of predicate p.

Example 1.

Consider the classical *append* program:

r_1: append([], X, X).
r_2: append([H : T], X, [H : Y]):-append(T, X, Y).

The set $Attr$(append) is {append1, append2, append3}. The set $Pos(r_1)$ is {append1(0), append2(0), append3(0)} and the set $Pos(r_2)$ is {append1(0), append2(0), append3(0), append1(1), append2(1), append 3(1)}. The corresponding terms t_{kj} in definitiôn 4.1 are:{[], X, X} for r_1 and {[H : T], X, [H : Y], T, X, Y} for r_2.

A splitting of the set $Attr$ in Inh and Syn can be obtained if we consider the flow of data in a definite clause program. In connection with this fact we introduce the following definition:

Definition 4.2[1], [2], [6], [7]. Given a DCP, P, a *direction assignement* or briefly *d*-**assignement** is a mapping of the arguments of each predicate symbol into the set {+, - }.
We will call an argument assigned to + "inherited" , and an argument assigned to - , "synthesized".

In the presence of a splitting we can define the attribute dependency schema (ADS) associated with a DCP as follows:

Definition 4.3[6], [7] Given a DCP with a *d*-assignement d, the associated ADS, $S = (N, P, Attr, D)$ defined as follows:

- $N, P, Attr$ are defined as in definition 4.1.
- D is a family of binary relations $\{D_r\}, r \in P$ defined on $Pos(r)$, such that aD_rb iff

(i) $a \in Input(r)$ and $b \in Output(r)$
(ii) the terms corresponding to these positions have a common variable.

Example 1 (continued):
If d(append)={+, +, -}, then $Input(r_1) = \{append1(0), append2(0)\}$ and $Input(r_2) = \{append1(0), append2(0), append3(1)\}$.

Definition 4.4
An ADS S is *well-formed* (or non-circular) if the tranzitive closure of the relation D is a partial order.

We assume in the following that a DCP is *augmented* with a *goal* . To simplify the construction we assume the the goal clause is an additional clause whose left-hand side is a special nullary predicat *goal* , which does not occur in the clauses of the program. An *augmented* DCP P will be denoted (P, g), where

$$g : goal \leftarrow b_1, \cdots, b_s.$$

is the additional clause.

Definition 4.5 Let (P, g) be an *augmented* DCP . A *d*-assignement d is said to be a *proper* *d*-assignement for (P, g) iff the associated ADS of the augmented DCP is well-formed.

5. Logic programs with types

We consider in the following the case of Turbo Prolog, as example of language of logic programming with types. Let a set of lines from an first author's automated theorems prover :

Example 2.

domains

term=**var**(symbol); **con**(symbol); **cmp**(symbol, terml)

terml=term*

form=s(symbol); and(form, form); or(form, form); impl(form, form); neg(form); atom(symbol, terml); all(term, form); ex(term, form)

forml=form*

predicates

arg(integer, terml, term)

conj(form)

disj(form)

member(form, forml)

append(forml, forml, forml)

From the section **domains** results that a **term** is constructed from the variables (denoted by **var**(symbol)), constantes (denoted by **con**(symbol)) and that a composed term is constructed with : the functor **cmp** , the name of term, and the list of his arguments:

Example : f(a, x) , where a is a constant and x is a variable, is introduced as **cmp**(f, [**con**(a), **var**(x)]).

Observation: The set of all terms *correct* constructed , can be considered as a set which we denote the **type term.**

A declaration in section **predicates** as of predicate "arg(integer, terml, term)" means that the third argument of this predicate must be contained in the **type term.**

(The predicate "arg" selects the "integer"-th element of a list "terml" of terms.)

Analogously, in this program exists the **type form** . The following formula: $and(atom(p, [var(x), con(b)]), atom(q, [cmp(f, [var(y)])]))$ is in the **type form** and represents the usully writted logical formula :

$$p(x, b) \wedge q(f(y)).$$

We can now define the notion of **type**:

Definition 5.1

The **type** T_i of the i-th arguments of a predicate p is the set of all values which this argument can get. (This set is established in the declarations as **domains** in

Turbo Prolog).

A **declaration of type** is a construction of the form $p(T_1, \cdots, T_k)$ (as in section **predicates** of a program in Turbo Prolog), where T_i is a **type**. Let consider that a predicate p can have only one declaration of type.

When a d-assignement is done, the set of arguments of a predicat p is divided in two set of arguments: the set of "input " (associated with +) arguments, and of "output" (associated with -) arguments.

Definition 5.2 Let $p(t_1, \cdots, t_k)$ be an atom and $p(T_1, \cdots, T_k)$ the declaration of type for this. Let $I \subseteq \{1, \cdots, k\}$ the set of indices of input positions (accordingly with a d-assignement) , and O the set of indices of output positions. The atom $p(t_1, \cdots, t_k)$ is **input well-typed** , briefly **input w-t** , if for every $j \in I$ results that $t_j \in T_j$. Analogously is introduced the notion of **output w-t.**

Let observe that the **well-typedness** is a notion related with a d-assignement. Hence, we assume anywhere in the following that a d-assignement is done.

Definition 5.3

- A goal (query)

$$b_1, \cdots, b_n$$

is called **well-typed** if from e $b_1, \cdots, b_{j-1}$ **output w-t** results b_j **input w-t**, for each $j \in [2, n]$.

- A clause

$$h \leftarrow a_1, \cdots, a_m$$

is called **well-typed** if

(i)from h **input w-t** , and $a_1, \cdots, a_{j-1}$ **output w-t** results:
a_j **input w-t**, for each $j \in [2, m]$, and

(ii)from h **input w-t** , and $a_1, \cdots, a_m$ **output w-t** results:
h **output w-t**.

- A program P is **well-typed** if every clause of it is.

- An augmented program (P, g) is **well-typed** if both P and g are.

In particular an atomic goal is **well-typed** if it is **input w-t** and a unit clause $h \leftarrow$. is **well-typed** if from h **input w-t** results h **output w-t**.

Example 3

The following is a non **well-typed** query for the program in *Example 2* .

goal: ← member(X, [s(a), s(b), s(c)]), append(X, X, Y)

(let remember that in Turbo Prolog the variables are written with the upper leters). Here from output well-typedness of the atom member(X, [s(a), s(b), s(c)]) not results input well-typedness of the atom append(X, X, Y).

In [9] is proved the following result :

Theorem 5.4 Let (P, g) be an augmented DCP.If d is a proper d-assignment of (P, g) and if P is **well-typed** ,then g,is **well- typed**.

For the above example of not **well-typedness** of the goal (example 3) it can be checked that d is not a proper d-assignment for (P, g).

6. Verification of well-typedness of a logic program

The application is written in Turbo Pascal 7.0. Having as input a Prolog program, given as a text file, it verifies if the input program is well-typed.The application will be send at request by the second author. In the following we will describe shortly this application.

We assume that the Prolog program works with two domains: the first named **form** (as a symbol, integer or character) and the second **forml** (as a list of forms).

The input data are read from the text file named **logic.txt**, which contains the Prolog program, in fact the sections **domains, predicates, goal** and **clauses**. In this file, in the description of the clauses, the comma, representing the logic operation "and" and the sign ":-" representing "if" (the reverse of the logical implication) are replaced with a blank.

Data types:

```
functie=function (x:string):boolean
```

defines the type of a criterion function, which verifies if its argument (given as a string) is a symbol, an integer or a character

```
    nod=record
nume:string[20];
c:integer;
     end
```

where nume is the name of a predicate or the name of the domain corresponding to an argument of the predicate, c is the number of a predicate arguments or the type of a predicate argument (0 if it's an input argument and 1 if it's an output argument)

```
variabila=record
nume:string[20]; - the name of a variable
tip:string[20]; - the name of the domain corresponding to the variable
     end
```

defines the type of a variable from a clause

```
sir_var=array[1..20] of variabila
```

represents the type corresponding to the array of the variables from a clause

```
sir=array[1..20] of nod
```

is the type used to represent a predicate with its arguments; the first record from the array contains the name of the predicate and the number of its arguments and the rest of the records from the array contains for each argument of the predicate the name of the argument's domain and the argument's type (0 if it's an input argument and 1 if it's an output argument)

```
sird=array[1..20] of sir
```

is the type used to represent the array of the predicates from the Prolog program

```
sirs=array[1..20] of string[50]
```

is the type used to represent the clauses from the program; each clause is retained as a string; the first element from the array is the goal

Global variables:

* dom - a variable of type functie; represents the type corresponding to a form
* pred - a variable of type sir representing the array of the predicates from the Prolog program
* Notations

pred[i][1].nume - the name of the i-th predicate

pred[i][1].c - the number of arguments of the i-th predicate

pred[i][j].nume - the name of the domain of the j-th argument of the i-th predicate

pred[i][j].c - 0 if the j-th argument of the i-th predicate is an input argument and 1 if it's an output argument

* np - an integer variable representing the number of predicates
* clauze - a variable of type sirs representing the clauses, including the goal
* nc - an integer variable representing the number of clauses, including the goal

The algorithm consists of:

* read of the input data from the text file and identification of the predicates, the clauses and the goal
* verification if an atom is well-typed, in fact if it is input well-typed and output well-typed
* verification if a clause is well-typed
* verification if the goal is well-typed
* if all the clauses, including the goal, are well-typed (corresponding to the definitions from the theory part), then the program is well-typed, otherwise no

Subprograms used:

(F) *e_symbol(x : string) : boolean*

- verifies if the argument is a symbol

(F) *e_integer(x : string) : boolean*

- verifies if the argument is an integer

(F) *e_var(x : string; var cap, coada : string) : boolean*

- verifies if the argument is a variable (simple or a list); if it's a list, then *cap* and *coada* returns the head and the tail of the list

(F) $e\_lista(x : string; f : functie) : boolean$
- verifies if the argument is a list with elements verifying the criterion function f

(P)$citire$
- read the input data from the text file *logic.txt*

(F) $elimin(x : string) : string$
- eliminates the blanks from the beginning of a string and returns the result

(P) $construire(x : string; var\ y : sir)$
- forms the predicate y from the string x

(P)$identificare(x : string; var\ n : integer; var\ y : sirs; var\ m : integer)$;
- processes the string x representing an atom and identifies the name of the corresponding predicate and its arguments
- n represents the index of the predicate in the array pred
- y represents the array of the predicate arguments (as an array of strings)
- m represents the number of the predicate arguments

(P) $prelucrez(x : string; t : string; var\ cont : boolean; var\ l\_var : sir\_var; var\ nv : integer)$
- processes a variable from a clause (the variable is named x and it's domain is t) and point out if the variable was found once more in the clause, but with an inadequate domain
- l_var represents the array of variables from the clause
- nv represents the number of variables from the clause
- the variable *cont* is true if a contradiction (shown above) is found

(P) $atom\_i\_o\_w\_t(z : string; var\ l\_var : sir\_var; var\ nv : integer; var\ a\_i\_w\_t, a\_o\_w\_t : boolean)$
- verifies if the atom z (as a string) is well-typed, corresponding to the Definition 5.2
- l_var represents the array of variables from the clause
- nv represents the number of variables from the clause
- the variable $a\_i\_w\_t$ is true if the atom is input well-typed
- the variable $a\_o\_w\_t$ is true if the atom is output well-typed

(F) $goal\_w\_t(na : integer; i\_w\_t, o\_w\_t : sirb) : boolean$
- verifies if the *goal* is well-typed, corresponding to the Definition 5.3
- na represents the number of atoms in the goal
- i_w_t is an array of boolean values specifying if the atoms of the goal are input well-typed
- o_w_t is an array of boolean values specifying if the atoms of the goal are output well-typed

(F) $clause\_w\_t(na : integer; i\_w\_t, o\_w\_t : sirb) : boolean$
- verifies if a clause is well-typed, corresponding to the Definition 5.3

(F) $clauza\_w\_t(i : integer) : boolean$

- verifies if the i-th element of the array clauze is well-typed (this element could be the goal or a clause)

(F) *program_w_t : boolean*

- verifies if the Prolog program is well-typed, corresponding to the Definition 5.3

Examples

1. If the input file "logic.txt" contains the following Prolog program

```
        domains
   form=symbol
predicates
   member(form,forml)
   -+
   append(forml,forml,forml)
   ++-
goal
   member(X,[a,b,c]) append(X,X,Y).
clauses
   member(X,[X|_]).
   member(X,[_|L]) member(X,L).
   append([],X,X).
   append([H|T],X,[H|Y]) append(T,X,Y).
```

the application's result is "The program is not well-typed"

2. If the input file "logic.txt" contains the following Prolog program

```
domains
   form=symbol
predicates
   member(form,forml)
   -+
   append(forml,forml,forml)
   ++-
goal
   member(X,[a,b,c]) append([X],[X],W).
clauses
   member(X,[X|_]).
   member(X,[_|L]) member(X,L).
   append([],X,X).
   append([H|T],X,[H|Y]) append(T,X,Y).
```

the application's result is "The program is well-typed"

References

[1] K.R. Apt, M.H.van Emden: *Contribution to the theory of logic programming* J. of ACM, vol.29, 1982, pg.841-862.

[2] K.R. Apt, D. Pedreschi: *Studies in pure Prolog: termination*, CWI Report CS-R9048, September, 1990.

[3] M.P.Bonacina, J.Hsiang: *On rewrite programs: semantics and relationship with Prolog*, J. Logic Programming, 1992, vol.14, pp.155-180.

[4] K.L.Clark: *Predicate Logic as a computational formalism*, Res.Mon. 79/59 TOC, Imperial College, london, 1979.

[5] S.Debray, P.Mishra: *Denotational and operational semantics for PROLOG*, J. of Logic Programming, vol.5, nr.1, 1988, pp.33-61.

[6] P.Deransart, J.Maluszynski: *Relating logic programs and attribute grammars*, J.of Logic Programming, nr.2. 1985, pp.119-155.

[7] P.Deransart, J.Maluszynski: *A grammatical view of logic programming*, The MIT Press, 1993.

[8] D.Tatar: *Logic grammars as formal languages tool for studying logic programs* Studia Univ. "Babes-Bolyai", 1993, nr.3.

[9] D.Tatar: *Attribute grammar and Logic Programs with types*, Proceedings ROSYCS'96, Iasi, mai 1996, pp 57-69.

"Babeş-Bolyai" University of Cluj-Napoca, Department of Computer Science
E-mail address: {dtatar,gabis}@cs.ubbcluj.ro