NP–completeness results concerning the transformation of logic programs into attribute grammars

Markus Lohrey
Grundlagen der Programmierung, Institut für Softwaretechnik I,
Fakultät Informatik, Technische Universität Dresden,
D–01062 Dresden, Germany,
e–mail: lohrey@orchid.inf.tu-dresden.de

November 24, 1997

Abstract

Attribute grammars and logic programs are two well investigated formalisms, which were related in [DM85] for the restricted class of simple logic programs. In this paper we define the more restricted class of very simple logic programs and we prove that the problem of deciding, whether a given logic program is (very) simple, is NP–complete.

1 Introduction

Attribute grammars were introduced in [Knu68] as a formalism for the specification of the semantics of programming languages. [DJ90] and [AM91] give an overview of current research trends. One of the most significant features of attribute grammars are their declarative programming style and the existence of efficient attribute evaluation methods. For the last point, see e.g. [Alb89].

Another declarative programming paradigm are definite program clauses. Their ability to specify computations was first recognized in [Kow74]. For a modern and exhaustive introduction to the wide area of logic programming see [Apt96].

In [DM85] a strong relationship between attribute grammars [1] and definite program clauses has been established. The investigation of this relationship is attractive, because one can try to apply a great number of techniques, which were originally developed for attribute grammars, to logic programs. For instance, dependency graphs can be used to study strictness, necessity of the occur–check or other run–time properties of logic programs. Moreover, attribute evaluation methods can be used for giving alternative operational semantics to logic programs.

It has been shown in [DM85] that every attribute grammar, whose function symbols are interpreted as term constructors, can be transformed into a semantically equivalent logic program. However, the reverse construction, i.e., the transformation of a logic program into a semantically equivalent attribute grammar is not always possible. In [DM85] the restricted class of simple logic programs has been defined and a construction has been presented, which transforms a simple logic program into an equivalent attribute grammar. In general, the semantic domain of the resulting attribute grammar is not a free term algebra. In Section 3 we will define the even more restricted class of very simple logic programs. For a very simple logic program the construction in [DM85] produces an attribute grammar, whose semantic domain is a free term algebra. It should be noted that the class of simple logic programs is still Turing–complete whereas very simple logic programs are no longer Turing–complete (in fact, attribute grammars with a free term algebra as semantic domain are not Turing–complete).

For a given logic program it can be decided whether this logic program is (very) simple or not. Since many techniques, developed for attribute grammars, can be applied to simple logic

[1] The kind of attribute grammars we use in this paper are called functional attribute grammars in [DM85].

programs and even more to very simple logic programs, see the note above and [DM85], [DM93] for more details, it would be useful to have efficient algorithms for these problems. In this paper we show that the problem to decide, whether a given logic program is (very) simple, is in fact NP–complete. Therefore it is unlikely to find efficient algorithms for these problems.

This paper is organized as follows. Section 2 gives some basic definitions concerning attribute grammars and logic programs. In Section 3 we define the class of simple and very simple logic programs and show how a simple logic program can be transformed into an attribute grammar. Finally in section 4 we prove the NP–completeness results mentioned above.

2 Preliminaries

First we recall some basic notions from universal algebra. Throughout this paper we assume that there is a countable infinite set V of variables.

Definition 2.1. A *ranked alphabet* Γ is a finite set of symbols together with a mapping $rank_\Gamma : \Gamma \to \mathbb{N} = \{0,1,2,\ldots\}$. $\Gamma^{(n)} = \{f \in \Gamma \mid rank_\Gamma(f) = n\}$ is the set of all symbols of rank n. The set of all Γ*–terms* with variables in V, denoted by $T_\Gamma(V)$, is the smallest set such that (i) $V \subseteq T_\Gamma(V)$ and (ii) $f \in \Gamma^{(n)}, t_1, \ldots, t_n \in T_\Gamma(V)$ implies $f(t_1,\ldots,t_n) \in T_\Gamma(V)$. For $t \in T_\Gamma(V)$, $V(t)$ denotes the set of all variables, which occur in t. If $x \in V(t)$, we say that *x occurs in t*.

Definition 2.2. A *semantic domain* $D = (\Omega, \Gamma, \Pi, \varphi)$ consists of a set Ω, two ranked alphabets Γ and Π of *function* and *predicate symbols*, respectively, such that $\Gamma \cap \Pi = \emptyset$, and an *interpretation function* φ, i.e., for every $n \geq 0$, $f \in \Gamma^{(n)}$, and $q \in \Pi^{(n)}$, $\varphi(f)$ is a partial function $\varphi(f) : \Omega^n \to \Omega$ and $\varphi(q)$ is a relation $\varphi(q) \subseteq \Omega^n$. An *assignment* is a function $val : V \to \Omega$. Every assignment val can be lifted to a function $\widehat{val} : T_\Gamma(V) \to \Omega$ by (i) $\widehat{val}(x) = val(x)$ for $x \in V$ and (ii) $\widehat{val}(f(t_1,\ldots,t_n)) = \varphi(f)(\widehat{val}(t_1),\ldots,\widehat{val}(t_n))$ for $n \geq 0$, $f \in \Gamma^{(n)}$ and $t_1,\ldots,t_n \in T_\Gamma(V)$ [2].

Note that we allow partial functions. For the rest of the paper we fix two ranked alphabets Γ and Π of function and predicate symbols, respectively, such that $\Gamma \cap \Pi = \emptyset$.

Definition 2.3. The *free Γ–term algebra (generated by V)*, denoted by $\mathcal{T}_\Gamma(V)$, is the semantic domain $(T_\Gamma(V), \Gamma, \emptyset, \varphi)$, where $\varphi(f)(t_1,\ldots,t_n) = f(t_1,\ldots,t_n)$ for $n \geq 0$, $f \in \Gamma^{(n)}$, and $t_1,\ldots,t_n \in T_\Gamma(V)$. Thus, a free Γ–term algebra does not contain relations and every function symbol $f \in \Gamma$ is interpreted by itself. A *substitution* is an assignment $\theta : V \to T_\Gamma(V)$ such that $\{x \in V \mid \theta(x) \neq x\}$ is finite. If $\theta(x_i) = t_i$ for $1 \leq i \leq n$ and $\forall x \in V \backslash \{x_1,\ldots,x_n\} : \theta(x) = x$, we use the notation $t\{t_i/x_i \mid 1 \leq i \leq n\}$ rather than $\widehat{\theta}(t)$ for $t \in T_\Gamma(V)$.

Next we have to introduce the machinery of attribute grammars. For the simulation of logic programs it is not necessary to include terminals and start symbols in the grammatical part of attribute grammars. Therefore we just introduce the notion of an abstract context–free grammar (see [GTWW77]), which is equivalent to the notion of a many–sorted signature.

Definition 2.4. An *abstract context–free grammar*, or briefly *abstract cfg*, $G_0 = (N, L, P)$ consists of a finite set N of *nonterminals*, a finite set L of labels and a finite set $P \subset L \times N \times N^*$ [3] of *productions* such that for every $(l, X, \alpha), (l', X', \alpha') \in P$, $l = l'$ implies $(l, X, \alpha) = (l', X', \alpha')$, i.e., different productions are labeled by different labels. The production (l, X, α) will be denoted by $l : X \to \alpha$.

Derivations of abstract cfgs will be represented by trees. Nodes of a tree are specified by means of the well–known Dewey notation, i.e., a tree node x is a string $x = i_1.i_2\ldots i_n$ with $i_j > 0$ for $1 \leq j \leq n$. Intuitively, this string indicates the path from the root of the tree to x. Thus, the root itself is denoted by the string ϵ of length 0. Furthermore, for technical reasons it is useful to define $x.0 = x$.

[2]We assume that $\varphi(f)(a_1,\ldots,a_n)$ is undefined if one of its arguments a_i is undefined.

[3]As usual, N^* denotes the set of all finite sequences of elements of N, including the empty sequence, which will be denoted by ϵ.

Definition 2.5. Let $G_0 = (N, L, P)$ be an abstract cfg. A *syntax tree* of G_0 is a finite tree s whose nodes are labeled by nonterminals from N such that the following condition holds:

For every node x there is a production $(l : X_0 \to X_1 \dots X_n) \in P$ with $n \geq 0$ such that x is labeled by X_0, x has exactly n successor nodes $x.1, \dots, x.n$, and $x.i$ is labeled by X_i for $1 \leq i \leq n$. In this situation we say that the production $l : X_0 \to X_1 \dots X_n$ is applied at the node x.

Definition 2.6. An attribute grammar, or briefly *ag*, $G = (G_0, D, B, R, C)$ consists of the following components:

- an abstract cfg $G_0 = (N, L, P)$
- a semantic domain $D = (\Omega, \Gamma, \Pi, \varphi)$
- an *attribute description* $B = (Inh, Syn, inh, syn)$. Inh and Syn are finite disjoint sets of *inherited* and *synthesized attributes*, respectively. inh and syn are functions $inh : N \to 2^{Inh}$ [4] and $syn : N \to 2^{Syn}$, respectively. $Att = Inh \cup Syn$ is the set of *attributes* of G.
- a set $R = \{R(p) \mid p \in P\}$ of finite sets $R(p)$ of *semantic rules*. To every production $p \in P$ of the form $l : X_0 \to X_1 \dots X_n$ with $n \geq 0$ we assign the set

  $$in(p) = \{\langle \gamma, i\rangle \mid (\gamma \in syn(X_0) \wedge i = 0) \vee (\gamma \in inh(X_i) \wedge 1 \leq i \leq n)\}$$

  of *inside attribute occurrences* of p and the set

  $$out(p) = \{\langle \gamma, i\rangle \mid (\gamma \in inh(X_0) \wedge i = 0) \vee (\gamma \in syn(X_i) \wedge 1 \leq i \leq n)\}$$

  of *outside attribute occurrences* of p . $att(p) = in(p) \cup out(p)$ is the set of *attribute occurrences* of p. For every $\langle \gamma, i\rangle \in in(p)$, the set $R(p)$ contains exactly one semantic rule of the form $\langle \gamma, i\rangle = t$ with $t \in T_\Gamma(out(p))$. Nothing else belongs to $R(p)$.
- a set $C = \{C(p) \mid p \in P\}$ of finite sets $C(p)$ of *semantic conditions*. For every $p \in P$, very element of $C(p)$ has the form $q(t_1, \dots, t_n)$ with $q \in \Pi^{(n)}$ and $t_i \in T_\Gamma(out(p))$ for $1 \leq i \leq n$.

G is called *free* iff there is a ranked alphabet Γ such that D is the free Γ–term algebra $\mathcal{T}_\Gamma(V)$. In particular, a free ag does not contain semantic conditions.

The semantic of an ag is the set of all (correctly) decorated syntax trees of the underlying abstract cfg, defined as follows.

Definition 2.7. Let $G = (G_0, D, B, R, C)$ be an ag with an underlying abstract cfg $G_0 = (N, L, P)$ and a semantic domain $D = (\Omega, \Gamma, \Pi, \varphi)$. Furthermore let s be a syntax tree of G_0 and x be a node of s, which is labeled by $X \in N$. To x we assign the set $inh(x) = \{\langle \gamma, x\rangle \mid \gamma \in inh(X)\}$ of *inherited attribute instances* of x and the set $syn(x) = \{\langle \gamma, x\rangle \mid \gamma \in syn(X)\}$ of *synthesized attribute instances* of x. $inst(x) = inh(x) \cup syn(x)$ is the set of *attribute instances* of x. The set $inst(s)$ of all attribute instances of s is $\bigcup\{inst(x) \mid x \text{ is a node of } s\}$. A *decoration* of s is a function $val : inst(s) \to \Omega$. A decoration val is called *valid* iff for every node x of s the following condition holds:

Assume that production p is applied at node x. Let $\theta : out(p) \to inst(s)$ be the substitution, defined by $\theta(\langle \gamma', j\rangle) = \langle \gamma', x.j\rangle$ for every $\langle \gamma', j\rangle \in out(p)$ (recall that we defined $x.0 = x$), where we assume w.l.o.g. that $out(p)$ is contained in the set V of variables. Then for every semantic rule $\langle \gamma, i\rangle = t$ in $R(p)$, the equation $val(\langle \gamma, x.i\rangle) = \widehat{val}(\widehat{\theta}(t))$ has to hold in D. Furthermore, for every semantic condition $q(t_1, \dots, t_n) \in C(p)$ we must have $(\widehat{val}(\widehat{\theta}(t_1)), \dots, \widehat{val}(\widehat{\theta}(t_n))) \in \varphi(q)$.

A *decorated tree* of G is a pair, consisting of a syntax tree s of G_0 and a valid decoration of s.

In most investigations only the class of so called non–circular attribute grammars is considered, since non–circularity of an ag G is sufficient for the construction of an attribute evaluator for G. For our purpose this restriction is not necessary. Next we give some basic definitions concerning logic programs.

[4] For every set A, 2^A denotes the powerset of A.

Definition 2.8. The set of all (Γ, Π)–*atoms* with variables in V, denoted by $A_{\Gamma,\Pi}(V)$, is the set $\{q(t_1, \ldots, t_n) \mid q \in \Pi^{(n)} \wedge t_1, \ldots, t_n \in T_\Gamma(V)\}$. $A_{\Gamma,\Pi}(\emptyset)$ is the set of all *ground* (Γ, Π)–atoms. $V(q(t_1, \ldots, t_n)) = \bigcup_{i=1}^{n} V(t_i)$ is the set of all variables, which occur in the atom $q(t_1, \ldots, t_n)$. A *definite* (Γ, Π)*–clause*, or just *clause*, is a formula of the form $\forall x_1, \ldots, x_n(A_1 \wedge \ldots \wedge A_m \rightarrow A_0)$, where $m \geq 0$, $A_0, \ldots, A_m \in A_{\Gamma,\Pi}(V)$ and $\{x_1, \ldots, x_n\} = \bigcup_{i=0}^{m} V(A_i)$. In the sequel we will use the abbreviation $A_0 \leftarrow A_1, \ldots, A_m$ for this clause. A *definite logic program*, or briefly *dlp*, is a triple $H = (\Gamma, \Pi, U)$, where U is a finite set of definite (Γ, Π)–clauses.

Definition 2.9. Let θ be a substitution. The atom $\theta(A) = q(\hat{\theta}(t_1), \ldots, \hat{\theta}(t_n))$ is called an *instance* of the atom $A = q(t_1, \ldots, t_n)$. The clause $\theta(A_0) \leftarrow \theta(A_1), \ldots, \theta(A_m)$ is called an instance of the clause $A_0 \leftarrow A_1, \ldots, A_m$.

There are several ways to assign a semantic to a dlp $H = (\Gamma, \Pi, U)$. Probably the way, which is familiar to most of the readers, is the least Herbrand model of H (see [Apt96]). The least Herbrand model of H consists of the set of all ground atoms that are logical consequences of U. This set can be generated by an operational method as well.

Definition 2.10. Let $H = (\Gamma, \Pi, U)$ be a dlp. A *proof tree* of H is a finite tree s whose nodes are labeled by atoms from $A_{\Gamma,\Pi}(V)$ such that the following condition holds:

For every node x of s there is an instance $A_0 \leftarrow A_1, \ldots, A_n$ $(n \geq 0)$ of a clause in U such that x is labeled by A_0, x has exactly n successor nodes $x.1, \ldots, x.n$, and $x.i$ is labeled by A_i for $1 \leq i \leq n$.

As shown in [Cla78] the least Herbrand model of H consists of all ground atoms that are the root of a proof tree of H.

3 Transformation of definite logic programs into attribute grammars

In the rest of the paper we fix a dlp $H = (\Gamma, \Pi, U)$.

The material in this section is taken from [DM85]. Our aim is to translate the dlp H into a semantically equivalent ag G. What it means precisely that the dlp H and the ag G are semantically equivalent, will be defined at the end of this section. Intuitively, the predicate symbols in Π will become the nonterminals of G. The clauses in U will be translated into productions of G by simply removing the argument positions of the predicate symbols in Π. To store the values of the argument positions of $q \in \Pi^{(n)}$, we assign n attributes $q1, \ldots, qn$ to q, one for each argument position $j \in \{1, \ldots, n\}$ of q. Now the formalism of definite logic programs gives no hint of how to split this attribute set into inherited and synthesized attributes. Therefore we have to enrich H with some additional information.

Definition 3.1. A *direction assignment*, or briefly *d–assignment*, for H is a function d, mapping every pair (q, j) with $q \in \Pi^{(n)}$ and $1 \leq j \leq n$ to an element of the set $\{\uparrow, \downarrow\}$.

$d(q, j) = \uparrow$ means that qj becomes a synthesized attribute, whereas $d(q, j) = \downarrow$ means that qj becomes an inherited attribute. In the rest of this section, let d be a d–assignment for the dlp H. Since the value of an inside attribute occurrence must be calculated by use of the outside attribute occurrences, we have to impose a restriction on d.

Definition 3.2. Let

$$u = (q_0(s_{0,1}, \ldots, s_{0,n_0}) \leftarrow q_1(s_{1,1}, \ldots, s_{1,n_1}), \ldots, q_m(s_{m,1}, \ldots, s_{m,n_m})) \in U. \qquad (1)$$

$POS(u) = \{\langle q_i j, i\rangle \mid 0 \leq i \leq m \wedge 1 \leq j \leq n_i\}$ is the set of *positions* of u (this set will become the set of attribute occurrences of the translation of the clause u). For $0 \leq i \leq m$, $1 \leq j \leq n_i$, the term $s_{i,j}$ is denoted by $t(\langle q_i j, i\rangle, u)$. For $pos \in POS(u)$, we say that $t(pos, u)$ *appears at the* *position* pos *of* u. The set $POS(u)$ is partitioned into the sets $in_d(u)$ and $out_d(u)$ of *inside* and *outside positions under* d *of* u, respectively, defined as follows:

$$in_d(u) = \{\langle q_i j, i\rangle \mid (i = 0 \wedge d(q_0, j) = \uparrow) \vee (1 \le i \le m \wedge d(q_i, j) = \downarrow)\}$$

$$out_d(u) = \{\langle q_i j, i\rangle \mid (i = 0 \wedge d(q_0, j) = \downarrow) \vee (1 \le i \le m \wedge d(q_i, j) = \uparrow)\}$$

d is called *safe for* H iff

$$\forall u \in U\, \forall pos \in in_d(u)\, \forall x \in V(t(pos, u))\, \exists pos' \in out_d(u) : x \in V(t(pos', u)).$$

This condition says that for every clause $u \in U$, every variable that occurs in a term that appears at an inside position of u occurs also in a term that appears at some outside position of u. d is called *very safe for* H iff d is safe for H and the following additional condition holds:

$$\forall u \in U\, \forall pos, pos' \in out_d(u) : t(pos, u), t(pos', u) \in V \;\wedge\; (pos \ne pos' \;\Rightarrow\; t(pos, u) \ne t(pos', u))$$

This condition says that for every clause $u \in U$, the terms that appear at the outside positions of u are different variables. H is called *simple* iff a safe d–assignment for H exists. H is called *very simple* iff a very safe d–assignment for H exists.

Example 3.3. Let $H = (\{nil, cons\}, \{app\}, \{u_1, u_2\})$ be the well–known append–dlp, where $u_1 = (app(nil, L, L) \leftarrow\ )$, $u_2 = (app(cons(X, L_1), L_2, cons(X, L_3)) \leftarrow app(L_1, L_2, L_3))$, and X, L_1, L_2, L_3 are variables. Let a d–assignment d for H be given by $d(app, 1) = d(app, 2) = \downarrow$, $d(app, 3) = \uparrow$. d is a safe but not very safe d–assignment for H. In fact, H is not very simple.

Example 3.4. Simple dlps are Turing–complete since every two–counter machine can be simulated by a simple dlp. A two–counter machine M consists of a finite set Q of states, an initial state $q_0 \in Q$, a set $F \subseteq Q$ of final states, and a finite set R of statements. Every $r \in R$ has the following form, where q denotes the current state, x and y are two registers whose values range over $\mathbb{N}$, and $i, j \in Q$.

- if $q = i$ then $x := x + 1$ and $q := j$ (analogously for the register y)
- if $q = i$ then $x := x - 1$ and $q := j$ (analogously for the register y)
- if $q = i$ and $x = 0$ then $q := j$ else $q := k$ (analogously for the register y)

A calculation of M is defined in the obvious way. M can be simulated by the dlp $H(M) = (\{0, s\}, \{P_i \mid i \in Q\}, \{P_i(x, y) \leftarrow\ \mid i \in F\} \cup \bigcup_{r \in R} U_r)$, where U_r consists of the following clauses, depending on the form of the statement r.

- $P_i(x, y) \leftarrow P_j(s(x), y)$ (analogously for y)
- $P_i(s(x), y) \leftarrow P_j(x, y)$ (analogously for y)
- $P_i(0, y) \leftarrow P_j(0, y)$, $P_i(s(x), y) \leftarrow P_k(s(x), y)$ (analogously for y)

The d-assignment d for $H(M)$, which is given by $d(P_i, 1) = d(P_i, 2) = \downarrow$ for every $i \in Q$ is a safe d–assignment for $H(M)$. On the other hand, if for instance there exist two statements of the form (if $q = k$ then $x := x + 1$ and $q := i$) and (if $q = i$ then $x := x - 1$ and $q := j$) then $H(M)$ is not very simple. In fact, it is not difficult to see that very simple dlps are not Turing–complete.

Example 3.5. Let $H = (\{0, s\}, \{plus\}, \{u_1, u_2\})$ be the well–known dlp for adding natural numbers, where $u_1 = (plus(0, x, x) \leftarrow\ )$ and $u_2 = (plus(s(x), y, s(z)) \leftarrow plus(x, y, z)$. Let a d–assignment d for H be given by $d(app, 1) = d(app, 3) = \uparrow$, $d(app, 2) = \downarrow$. d is a very safe d–assignment for H.

Since for every given dlp there are only finitely many different d–assignments, it is decidable whether a dlp is simple or very simple, respectively. We call the corresponding computational problems SIMPLE and VERY–SIMPLE, respectively. A problem instance of (VERY–)SIMPLE consists of a dlp H. The question is whether H is (very) simple or not.

In the rest of this section we will show that a (very) simple dlp H together with a (very) safe d–assignment d for H can be transformed into a semantically equivalent (free) ag $G(H, d)$. This result is not necessary for understanding our main results in Section 4 but we present it for completeness.

If the d–assignment d is very safe for H, then for every clause $u \in U$, the terms in $\{t(pos, u) \mid pos \in in_d(u)\}$ can be constructed from the terms in $\{t(pos, u) \mid pos \in out_d(u)\}$ (which are variables in this case) and the function symbols in Γ. If d is only safe but not very safe, we need additional special selector functions $s_i\text{–}f$, defined as follows:

Definition 3.6. For every $f \in \Gamma^{(n)}$ and $i \in \{1, \dots, n\}$, $s_i\text{–}f$ is a partial function on $T_\Gamma(V)$, defined by (i) $s_i\text{–}f(t) = t_i$ if $t = f(t_1, \dots, t_n)$ and (ii) $s_i\text{–}f(t) =$ undefined otherwise. $s_i\text{–}f$ is called a *selector function*.

In order to refer to selector functions, for every $s_i\text{–}f$ we introduce a new function symbol $sel_i\text{–}f$. Of course we assume that $sel_i\text{–}f \notin \Gamma$. We introduce the new ranked alphabet $\Gamma' = \Gamma \cup \{sel_i\text{–}f \mid f \in \Gamma^{(n)}, n \geq 0, 1 \leq i \leq n\}$, where (i) $rank_{\Gamma'}(f) = rank_\Gamma(f)$ if $f \in \Gamma$ and (ii) $rank_{\Gamma'}(f) = 1$ otherwise.

For a term $t \in T_\Gamma(V)$ and a variable $x \in V$, we denote by $Sel(t, x)$ the set of all terms $s = sel_{i_1}\text{–}f_1(\dots sel_{i_k}\text{–}f_k(x)\dots)$ such that the term $s\{t/x\}$ evaluates the variable x when every $sel_i\text{–}f$ is interpreted by $s_i\text{–}f$. In other words, $i_k.i_{k-1}.\dots.i_1$ specifies a path from the root of t to an occurrence of the variable x in t. This is formalized by the following definition.

Definition 3.7. Let $t \in T_\Gamma(V)$ and $x \in V$. The set $Sel(t, x)$ is defined by (i) $Sel(x, x) = \{x\}$, (ii) $Sel(y, x) = \emptyset$ for $y \in V \backslash \{x\}$, and (iii) $Sel(f(t_1, \dots, t_n), x) = \bigcup_{i=1}^{n} \{s\{sel_i\text{–}f(x)/x\} \mid s \in Sel(t_i, x)\}$ for $n \geq 0$, $f \in \Gamma^{(n)}$ and $t_1, \dots, t_n \in T_\Gamma(V)$.

For instance, $Sel(g(f(x), g(x, y)), x) = \{sel_1\text{–}f(sel_1\text{–}g(x)), sel_2\text{–}g(sel_2\text{–}g(x))\}$.

In addition to the selector functions, semantic conditions are also necessary for the simulation of the dlp H, if d is safe but not very safe. This is because of two reasons. Firstly, if there exist an $u \in U$ and a $pos \in out_d(u)$ such that the term $t(pos, u)$ is not just a variable, we have to express the fact that the value of the attribute occurrence pos is an instance of $t(pos, u)$. This will be done with the help of a predicate symbol $instance_{t(pos,u)}$ of rank one. Secondly, if there exist $pos, pos' \in out_d(u)$ and $x \in V$ such that $x \in V(t(pos, u)) \cap V(t(pos', u))$, we must express the fact that the corresponding subterms of the values of the attribute occurrences pos and pos' are equal. Therefore we have to include the syntactic equality of terms into the semantic domain of the simulating ag. Note that both situations cannot occur if d is very safe for H.

Construction 3.8. Let d be a safe d–assignment for H. The ag $G(H, d) = (G_0, D, B, R, C)$ is defined as follows:

- $G_0 = (\Pi, U, P)$, where for every clause $u \in U$ of the form shown in (1) we put the production $p(u) = (u : q_0 \to q_1, \dots, q_m)$ into P. Nothing else belongs to P.
- $D = (T_\Gamma(V), \Gamma', \{=\} \cup \{instance_t \mid \exists u \in U\, \exists pos \in out_d(u) : t = t(pos, u) \notin V\}, \varphi)$, where
  - φ interprets every $f \in \Gamma$ as in the Γ*–term algebra* $\mathcal{T}_\Gamma(V)$,
  - $\varphi(sel_i\text{–}f) = s_i\text{–}f$ for $f \in \Gamma^{(n)}$ and $1 \leq i \leq n$,
  - $\varphi(instance_{t_1})(t_2)$ iff t_2 is an instance of t_1 for $t_1, t_2 \in T_\Gamma(V)$,
  - $\varphi(=)$ is the syntactic equality on $T_\Gamma(V)$.
- $B = (Inh, Syn, inh, syn)$, where for every $q \in \Pi^{(n)}$, $inh(q) = \{qi \mid 1 \leq i \leq n, d(q, i) = \downarrow\}$ and $syn(q) = \{qi \mid 1 \leq i \leq n, d(q, i) = \uparrow\}$. $Inh = \bigcup_{q \in \Pi} inh(q)$ and $Syn = \bigcup_{q \in \Pi} syn(q)$.

- For every clause $u \in U$, $R(p(u)) = \{pos = t_{pos,u} \mid pos \in in_d(u)\}$, where for $pos \in in_d(u)$ the term $t_{pos,u} \in T_{\Gamma'}(V)$ is defined as follows:

  For every $x \in V(t(pos,u))$ let $pos_x \in out_d(u)$ such that $x \in V(t(pos_x,u))$. Since d is safe, such a pos_x must exist for every $x \in V(t(pos,u))$. Of course there may be several choices for pos_x. Therefore the construction is nondeterministic. Now let $t_x \in Sel(t(pos_x,u),x)$. Again there may be several choices for t_x. We set $t_{pos,u} = t(pos,u)\{t_x\{pos_x/x\}/x \mid x \in V(t(pos,u))\} \in T_{\Sigma'}(out_d(u))$.

- For every clause $u \in U$, $C(p(u)) = C_1(p(u)) \cup C_2(p(u))$, where

  - $C_1(p(u)) = \{instance_{t(pos,u)}(pos) \mid pos \in out_d(u), t(pos,u) \notin V\}$ and
  - $C_2(p(u)) = \{t\{pos/x\} = t'\{pos'/x\} \mid pos, pos' \in out_d(u), x \in V, t \in Sel(t(pos,u),x),$ $t' \in Sel(t(pos',u),x)\}$. Of course, equations of the form $t = t$ can be omitted in $C_2(p(u))$.

Example 3.9. Let H be the append program from Example 3.3 and let d be the d–assignment for H from the same example. The ag $G(H,d) = (G_0, D, B, R, C)$ has the following components.

- $G_0 = (\{app\}, \{u_1, u_2\}, \{u_1 : app \to app, u_2 : app \to\})$
- $B = (\{app1, app2\}, \{app3\}, inh, syn)$, where $inh(app) = \{app1, app2\}$ and $syn(app) = \{app3\}$.
- $R = \{R(u_1), R(u_2)\}$, where $R(u_1) = \{\langle app3, 0\rangle = \langle app2, 0\rangle\}$ and

$$\begin{aligned} R(u_2) = \{&\langle app3,0\rangle = cons(s_1\text{–}cons(\langle app1,0\rangle), \langle app3,1\rangle),\\ &\langle app1,1\rangle = s_2\text{–}cons(\langle app1,0\rangle), \langle app2,1\rangle = \langle app2,0\rangle\}.\end{aligned}$$

- $C = \{C(u_1), C(u_2)\}$, where

$$C(u_1) = \{instance_{nil}(\langle app1,0\rangle)\} \quad \text{and} \quad C(u_2) = \{instance_{cons(X,L_1)}(\langle app1,1\rangle)\}.$$

If d is very safe, then $C(p(u)) = \emptyset$ for every $u \in U$. Moreover, the function symbols $sel_i\text{–}f$ do not appear in the semantic rules of $G(H,d)$. Therefore we can omit them in the semantic domain D of $G(H,d)$. In this way $G(H,d)$ becomes a free ag.

Let Φ_H be the function, defined as follows. Φ_H maps a proof tree s of the dlp H to a pair, consisting of a syntax tree s' of G_0 (the underlying abstract cfg of ag $G(H,d)$) and a decoration val of s'. s' results from s by replacing for every node x of s the label $q(t_1, \ldots, t_n)$ of x by the label q. val is defined by $val(\langle qi, x\rangle) = t_i$ for every node x of s with label $q(t_1, \ldots, t_n)$ and $1 \le i \le n$.

Theorem 3.10. Let $H = (\Gamma, \Pi, U)$ be a simple dlp and let d be a safe d–assignment for H. Furthermore, let $G(H,d)$ be the ag as constructed in Construction 3.8. Then Φ_H is a bijective mapping between the set of all proof trees of H and the set of all decorated trees of $G(H,d)$.

The proof of this theorem can be found in [DM85].

4 Complexity of SIMPLE and VERY–SIMPLE

In this section we show that the computational problems SIMPLE and VERY–SIMPLE are NP–complete problems. The NP–completeness of SIMPLE will be proved by a reduction from SAT, which is the satisfaction problem for boolean expressions (see [Coo71] and [GJ79]), to SIMPLE. For the corresponding result for VERY–SIMPLE we will use a variant of SAT, called ONE–SAT, that will be introduced later.

Theorem 4.1. SIMPLE is NP–complete.

Proof. Since we can guess a d–assignment for the dlp H and check in polynomial time, whether this d–assignment is safe for H, SIMPLE is in NP. To prove that SIMPLE is NP–hard, we will construct a polynomial time reduction from SAT to SIMPLE. Firstly, we recall the definition of SAT. A problem instance S of SAT is a boolean expression, which can be assumed to be in conjunctive normal form, i.e.,

$$S = C_1 \wedge C_2 \wedge \ldots \wedge C_n, \tag{2}$$

where C_i is a nonempty disjunction of literals $A_{i,1}, \ldots, A_{i,l_i}$ $(l_i > 0)$ with

$$A_{i,j} = \begin{cases} \alpha_{i,j} & \text{or} \\ (\neg\alpha_{i,j}), & \end{cases} \tag{3}$$

where the $\alpha_{i,j}$ are boolean variables. To ease notation we will write the disjunction C_i as a set [5], i.e.,

$$C_i = \{A_{i,1}, \ldots, A_{i,l_i}\} \neq \emptyset. \tag{4}$$

Finally, let

$$\{\alpha_1, \ldots, \alpha_m\} = \{\alpha_{i,j} \mid 1 \leq i \leq n, 1 \leq j \leq l_i\} \tag{5}$$

be the set of all boolean variables that occur in S. Given such a problem instance S the question is, whether there exists a truth assignment for S which satisfies S, i.e., whether there exists a function

$$w : \{\alpha_1, \ldots, \alpha_m\} \rightarrow \{true, false\} \tag{6}$$

such that

$$\forall i \in \{1, \ldots, n\} \exists j \in \{1, \ldots, m\} : (\alpha_j \in C_i \wedge w(\alpha_j) = true) \vee ((\neg\alpha_j) \in C_i \wedge w(\alpha_j) = false).$$

In [Coo71], this question was shown to be NP–complete.

In order to construct a polynomial time reduction from SAT to SIMPLE let S be the problem instance of SAT given by (2) to (5). We construct a dlp

$$H(S) = (\{a\}, \{q\}, \{U(C_1), \ldots, U(C_n)\}).$$

The only function symbol a is of rank 0. The only predicate symbol q is of rank m, which is the number of different boolean variables appearing in S, see (5). For every $1 \leq i \leq n$ the clause $U(C_i)$ is of the form

$$q(s_{i,1}, \ldots, s_{i,m}) \leftarrow q(t_{i,1}, \ldots, t_{i,m}).$$

We fix a variable $x \in V$. For $1 \leq j \leq m$ the terms $s_{i,j}$ and $t_{i,j}$ are defined by

$$s_{i,j} = \begin{cases} x & \text{if } \alpha_j \in C_i \\ a & \text{otherwise} \end{cases} \quad \text{and} \quad t_{i,j} = \begin{cases} x & \text{if } (\neg\alpha_j) \in C_i \\ a & \text{otherwise.} \end{cases} \tag{7}$$

Let w be the truth assignment for S given in (6). We define a corresponding d–assignment d_w of $H(S)$ by

$$d_w(q, j) = \begin{cases} \uparrow & \text{if } w(\alpha_j) = false \\ \downarrow & \text{if } w(\alpha_j) = true \end{cases} \tag{8}$$

for $1 \leq j \leq m$. We will use the abbreviation $d_w(j)$ for $d_w(q, j)$.

[5]Therefore in each disjunction a literal can only appear once. Of course this is not a real restriction.

Example 4.2. Let

$$S = (\alpha_1 \vee \alpha_2 \vee \alpha_3) \wedge (\neg\alpha_2 \vee \neg\alpha_3 \vee \alpha_4) \wedge (\alpha_1 \vee \alpha_3 \vee \neg\alpha_4).$$

Then $H(S)$ consists of the following clauses:

$$q(x,x,x,a) \leftarrow q(a,a,a,a)$$

$$q(a,a,a,x) \leftarrow q(a,x,x,a)$$

$$q(x,a,x,a) \leftarrow q(a,a,a,x)$$

Let w be the following truth assignment:

$$w(\alpha_1) = w(\alpha_3) = w(\alpha_4) = \mathit{false}, w(\alpha_2) = \mathit{true}$$

In fact w satisfies S. The corresponding d–assignment d_w is

$$d_w(1) = d_w(3) = d_w(4) = \uparrow, d_w(2) = \downarrow .$$

d_w is a safe d–assignment for $H(S)$. The following claim shows that this is not just a coincidence.

Claim 4.3. w satisfies S iff d_w is a safe d–assignment for $H(S)$.

Proof of the claim. w satisfies S iff

$$\forall i \in \{1,\dots,n\} \exists j \in \{1,\dots,m\} : (\alpha_j \in C_i \wedge w(\alpha_j) = \mathit{true}) \vee ((\neg\alpha_j) \in C_i \wedge w(\alpha_j) = \mathit{false}).$$

Because of (7) and (8) this is equivalent to

$$\forall i \in \{1,\dots,n\} \exists j \in \{1,\dots,m\} : (s_{i,j} = x \wedge d_w(j) = \downarrow) \vee (t_{i,j} = x \wedge d_w(j) = \uparrow). \tag{9}$$

We claim that (9) is equivalent to the statement that d_w is a safe d–assignment for $H(S)$.

Assume that (9) holds and consider a clause $U(C_i)$. (9) says that x appears at an outside position $\mathit{pos} \in \mathit{out}_{d_w}(U(C_i))$ of $U(C_i)$. Since x is the only variable that occurs in $U(C_i)$ (and therefore is the only variable y that occurs in a term that appears at an inside position of $U(C_i)$), this shows that d_w is a safe d–assignment for $H(S)$.

Now assume that d_w is a safe d–assignment for $H(S)$ and consider a clause $U(C_i)$ of $H(S)$. We claim that x appears at an outside position of $U(C_i)$. Since $C_i \neq \emptyset$, there exists a $j \in \{1,\dots,m\}$ such that either $s_{i,j} = x$ or $t_{i,j} = x$. Thus, x appears at some position pos of $U(C_i)$. If pos is an outside position of $U(C_i)$, we are ready. Thus, assume that pos is an inside position of $U(C_i)$. Since d_w is a safe d–assignment for $H(S)$, there exists an outside position $\mathit{pos}' \in \mathit{out}_{d_w}(U(C_i))$ of $U(C_i)$ such that x occurs in the term that appears at position pos' of $U(C_i)$. Since $H(S)$ does not contain function symbols of rank greater than zero, this implies that x appears at the outside position pos' of $U(C_i)$, which proves (9). □

Since the function that maps a truth assignment w for S to the d–assignment d_w of $H(S)$ is bijective, the claim shows that we have constructed a correct reduction from SAT to SIMPLE. Since this reduction can be done in polynomial time, the proof of the theorem is complete. □

The proof above shows that for the class of logic programs with only one constant symbol (and no other function symbol) and only one predicate symbol, SIMPLE is already NP–complete.

Theorem 4.4. VERY–SIMPLE is NP–complete.

Proof. Of course, VERY–SIMPLE is also in NP. To prove that VERY–SIMPLE is NP-hard, we use a variant of SAT, called ONE–SAT (see [Sch78], where the problem is called ONE-IN-THREE 3SAT). Again a problem instance of ONE–SAT is a disjunctive normal form S. The question is whether there exists a truth assignment w for S such that in every disjunction C_i *exactly one* literal $A_{i,j}$ becomes true under w. This problem is known to be NP–complete as well. Let S be the disjunctive normal form S given in (2) – (5) and let w be the truth assignment for S given in (6).

We will construct a dlp $H'(S)$ that is a mild variant of the dlp $H(S) = (\{a\}, \{q\}, \{U(C_1), \ldots, U(C_n)\})$ constructed in the proof of Theorem 4.1. It is easy to see that the dlp $H(S)$ and the d–assignment d_w given in (8) satisfy the following fact:

> w satisfies exactly one literal $A_{i,j}$ in every disjunction C_i iff there is exactly one $pos \in out_{d_w}(U(C_i))$ for every clause $U(C_i)$ such that $x \in V(t(pos, U(C_i)))$.

Therefore, if w satisfies exactly one literal $A_{i,j}$ in every disjunction C_i, the d–assignment d_w is almost very safe for $H(S)$. The only problem that may arise is that there may exist a clause $U(C_i)$ and an outside position $pos \in out_{d_w}(U(C_i))$ such that $t(pos, U(C_i)) = a$, i.e., a term, which is not a variable, appears at an outside position of $U(C_i)$. We solve this problem by replacing every occurrence of the constant a in $U(C_i)$ by a new variable (different from x and all other new variables that will be introduced). Call the resulting clause $U''(C_i)$. Now it is possible that one of these new variables appears at an inside position $pos \in in_{d_w}(U''(C_i))$ of $U''(C_i)$ and thus has to appear also at an outside position of $U''(C_i)$. To compensate this, we collect all new variables of the clause $U''(C_i)$ (i.e., all variables different from x that occur in $U''(C_i)$) in the argument positions of a new predicate symbol p_i and attach the resulting atom to the body of $U''(C_i)$. The resulting clause will be called $U'(C_i)$. The d–assignments for the new predicate symbol p_i are chosen in such a way that every new variable appears exactly once at an outside position $pos \in out_{d_w}(U'(C_i))$ of $U'(C_i)$ and once at an inside position $pos' \in in_{d_w}(U'(C_i))$ of $U'(C_i)$. A formal presentation of this idea follows.

As motivated, every clause C_i is translated into a clause $U'(C_i)$ of the form

$$q(s'_{i,1}, \ldots, s'_{i,m}) \leftarrow q(t'_{i,1}, \ldots, t'_{i,m}), p_i(u_{i,1}, \ldots, u_{i,m_i}).$$

The number m_i will be specified later. Let $Y = \{y_1, y_2, \ldots\} \subset V$ and $Z = \{z_1, z_2, \ldots\} \subset V$ be two disjoint sets of new variables different from x. The terms $s'_{i,j}$ and $t'_{i,j}$ are defined by

$$s'_{i,j} = \begin{cases} x & \text{if } \alpha_j \in C_i \\ y_j & \text{otherwise} \end{cases} \quad \text{and} \quad t'_{i,j} = \begin{cases} x & \text{if } (\neg\alpha_j) \in C_i \\ z_j & \text{otherwise} \end{cases}$$

(note the small difference to the terms $s_{i,j}$ and $t_{i,j}$ in the proof of Theorem 4.1). Fix an $i \in \{1, \ldots, n\}$. Let $K = \{j \mid 1 \le j \le m \wedge s'_{i,j} \in Y\} = \{k_1, \ldots, k_{\#K}\}$ [6] and $L = \{j \mid 1 \le j \le m \wedge t'_{i,j} \in Z\} = \{l_1, \ldots, l_{\#L}\}$ be the set of all argument positions of q, where variables from Y respectively Z appear in $U'(C_i)$. W.l.o.g. we assume that $k_i < k_j$ and $l_i < l_j$ for every $i < j$. Define $m_i = \#K + \#L$. The terms $u_{i,j}$ are defined by

$$u_{i,j} = \begin{cases} y_{k_j} & \text{if } 1 \le j \le \#K \\ z_{l_{j-\#K}} & \text{if } \#K < j \le m_i. \end{cases}$$

For a given truth assignment w for S, the d–assignments $d_w(q, j)$ for the positions of the predicate symbol q are defined as in (8). The d–assignments $d_w(p_i, j)$ for the argument positions of the new predicate symbol p_i are defined by [7]

$$d_w(p_i, j) = \begin{cases} d_w(q, k_j) & \text{if } 1 \le j \le \#K \\ d_w(q, l_{j-\#K})^{-1} & \text{if } \#K < j \le m_i. \end{cases}$$

[6] $\#A$ denotes the number of elements in the finite set A.

[7] We define $\uparrow^{-1} = \downarrow$ and $\downarrow^{-1} = \uparrow$.

Example 4.5. Let S be the conjunctive normal form given in Example 4.2. $H'(S)$ consists of the following clauses:

$$q(x,x,x,y_4) \leftarrow q(z_1,z_2,z_3,z_4), p_1(y_4,z_1,z_2,z_3,z_4)$$
$$q(y_1,y_2,y_3,x) \leftarrow q(z_1,x,x,z_4), p_2(y_1,y_2,y_3,z_1,z_4)$$
$$q(x,y_2,x,y_4) \leftarrow q(z_1,z_2,z_3,x), p_3(y_2,y_4,z_1,z_2,z_3)$$

Let w be the truth assignment given in Example 4.2. w satisfies exactly one literal in each disjunction C_i. The corresponding d–assignment d_w is

- $d_w(q,1) = d_w(q,3) = d_w(q,4) =\uparrow, d_w(q,2) =\downarrow$
- $d_w(p_1,1) = d_w(p_1,3) =\uparrow, d_w(p_1,2) = d_w(p_1,4) = d_w(p_1,5) =\downarrow$
- $d_w(p_2,1) = d_w(p_2,3) =\uparrow, d_w(p_2,2) = d_w(p_2,4) = d_w(p_2,5) =\downarrow$
- $d_w(p_3,2) = d_w(p_3,4) =\uparrow, d_w(p_3,1) = d_w(p_3,3) = d_w(p_3,5) =\downarrow$.

This is a very safe d–assignment for $H'(S)$.

The following claim completes the proof. Its proof is very similar to the proof of the corresponding claim, made in the proof of Theorem 4.1, and it is therefore omitted.

Claim 4.6. w satisfies exactly one literal $A_{i,j}$ in every disjunction C_i of S iff d_w is a very safe d–assignment for $H'(S)$.

□

The proof above shows that for the class of logic programs with an empty set of function symbols, VERY–SIMPLE is already NP–complete.

5 Conclusion

In this paper, we have proved that the problem of deciding, whether a given logic program is simple, is NP–complete. A simple logic program H can be transformed into an attribute grammar G. In general G has a semantic domain, which is not a free term algebra, i.e., G is not free. We have defined the more restricted class of very simple logic programs. Very simple logic programs can be transformed into equivalent free attribute grammars. We proved that the problem of deciding, whether a given logic program is very simple, is NP–complete as well.

References

[Alb89] H. Alblas. Attribute evaluation methods. *Memoranda Informatica*, 89-20, University of Twente, Enschede, 1989.

[AM91] H. Alblas and B. Melichar. *International Conference SAGA, Prague*, volume 545 of *Lect. Notes Comput. Sci.* Springer–Verlag, Juni 1991.

[Apt96] K. Apt. *From Logic Programming to Prolog.* Prentice-Hall, January 1996.

[Cla78] K. L. Clark. Predicate logic as a computational formalism. Report Res. Mon. 79/59, Imperial College, London, 1978.

[Coo71] S.A. Cook. The complexity of theorem–proving procedures. In *Proc. 3rd IEEE Symp. on the Foundations of Computer Science*, pages 151–158, 1971.

[DJ90] P. Deransart and M. Jourdan. *Attribute Grammars and Their Applications, International Conference WAGA, Paris*, volume 461 of *Lect. Notes Comput. Sci.* Springer–Verlag, September 1990.

[DM85] P. Deransart and J. Maluszynski. Relating logic programs and attribute grammars. *J. Logic Programming*, 2:119–155, 1985.

[DM93] P. Deransart and J. Maluszynski. *A Grammatical View of Logic Programming*. MIT Press, Cambridge MA, 1993.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP–completeness*. Freeman, San Francisco, 1979.

[GTWW77] J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright. Initial algebra semantics and continuous algebras. *J. ACM*, 24:68–95, 1977.

[Isa91] T. Isakowitz. Can we transform logic programs into attribute grammars. *Journal of Theoretical Informatics and Applications*, 15:499–543, 1991.

[Knu68] D.E. Knuth. Semantics of context–free languages. *Math. Systems Theory*, 2:127–145, 1968.

[Kow74] R. Kowalski. Predicate logic as a programming language. *Inform. Process. Letters*, 74:569–574, 1974.

[Sch78] T.J. Schaefer. The complexity of satisfiability problems. In *Proc. 10th Ann. ACM Symp. on Theory of Computing*, Association for Computing Machinery, pages 216–226, New York, 1978.