ISE-TR-88-71

On the relation between CCFG programs and logic programs

by

Yoshiyuki YAMASHITA and Ikuo NAKATA

June 1, 1988

INSTITUTE
OF
INFORMATION SCIENCES AND ELECTRONICS

UNIVERSITY OF TSUKUBA

On the relation between CCFG programs and logic programs

Yoshiyuki YAMASHITA* and Ikuo NAKATA**

University of Tsukuba,

Tsukuba-shi, Ibaraki-ken, 305, Japan.

Abstract

CCFG programming is a programming system based on the formal grammar: Coupled Context-Free Grammar (CCFG) [1]. A CCFG is regarded as a program, called a CCFG program, and its semantics is defined by the language, a set of tuples of derived terminal strings which it generates. CCFG programs are similar to logic programs, because their semantics domains are similar in the sense that both represent relations between input/output data objects. In this paper the relationship between CCFG programs and logic programs is discussed by using program transformations. The rules for these transformations are so simple and we can clearly understand the characteristics of CCFG programs by comparing them with those of corresponding logic programs.

---

* With the Doctoral Program in Engineering.

** With the Institute of Information Sciences and Electronics.

1.Introduction

CCFG programming is a programming system based on the formal grammar: Coupled Context-Free Grammar (CCFG) [1], which is an extension of context-free grammar. A CCFG is regarded as a program, called a CCFG program, and its semantics is defined by the language, a set of tuples of terminal strings which it generates. A CCFG program contains context-free grammars, which are interpreted as the representations of input/output data structures, and the program itself is interpreted as the representation of the relation between the data structures. For example, the following is the simple CCFG program which represents the relation between strings composed of terminal symbols "a" and "b" and strings composed of "c" and "d",

$$\{X\rightarrow\varepsilon,\quad Y\rightarrow\varepsilon\},$$

$$\{X\rightarrow aX,\ Y\rightarrow cY\},$$

$$\{X\rightarrow bX,\ Y\rightarrow dY\}.$$

Here each set of the above is called a rule-set. The empty string is denoted by ε. The derivation rule in CCFG is given as that all the production rules in a rule-set must be simultaneously applied. Therefore one of the derivations from the couple (X, Y) of nonterminal symbols is given as follows,

$$(X, Y) \Rightarrow (aX, cY) \Rightarrow (abX, cdY) \Rightarrow (abbX, cddY) \Rightarrow (abb, cdd).$$

The language is a set of couples of terminal strings derived from the start tuple (X, Y), that is

$$\{(\varepsilon, \varepsilon), (a, c), (b, d), (ab, cd), (ba, dc), (abb, cdd), ...\},$$

and it is called the Coupled Context-Free Language (CCFL). This gives the semantics of the program. The production rules X→ε, X→aX, and X→bX determine the structure of strings derived by X, and the production rules Y→ε, Y→cY, Y→dY determine the structure of strings derived by Y. The program on the whole represents the relation between the strings derived by X and Y. More complicated CCFG programs can be built by using the metasymbol "≈" which has almost the same meaning as the equality symbol in equational logic.

A logic program (a set of definite Horn clauses), as well-known, represents a relation. In this sense we expect that there is a relationship between CCFG programs and logic programs. For example, the following is a simple logic program,

```
p(nil, nil),
p(a(*X), c(*Y)):-p(*X, *Y),
p(b(*X), d(*Y)):-p(*X, *Y).
```

The semantics of the program for the predicate symbol: p is specified by the following set of ground atoms which can be proved by the program,

{p(nil, nil), p(a(nil), c(nil)), p(b(nil), d(nil)), p(a(b(nil)), c(d(nil))),
p(b(a(nil)), d(c(nil))), p(a(b(b(nil))), c(d(d(nil)))), ...}.

In the above two example of the CCFG program and the logic program, we find that both semantics are similar when we identify the terminal symbols "a", "b", "c" and "d" with the functors "a()", "b()", "c()" and "d()", respectively. Since a CCFG program and a logic program represent relations and their semantics are defined by sets of tuples, it is expected that both programs have similar features. The purpose of this paper is to clarify the relationship between them and the characteristics of CCFG programs by comparing them. More precisely, the purpose is to give the program transformation rules which preserve their semantics.

In section two, we define term-based CCFG (t-CCFG) programs in place of the original string-based one, in order to stand on the uniform data domain, the Herbrand universe [2] of CCFG and logic programs. Because strings can be regarded as the special terms which satisfy the associative law, that is (a:b):c=a:(b:c), we can easily establish the term-based one by discarding the associative law. The t-CCFG program corresponding to the string-based CCFG program described above is, for example, given as follows,

{X→nil, Y→nil},
{X→a(X), Y→c(Y)},
{X→b(X), Y→d(Y)},

The right-hand sides of production rules are the terms which are composed of constants, functors and nonterminal symbols. The semantics of t-CCFG programs are defined in the same way as that of string-based ones, by using the languages and the least fixpoints. In the above case, the semantics is given as

{(nil, nil), (a(nil), c(nil)), (b(nil), d(nil)), (a(b(nil)), c(d(nil))),
(b(a(nil)), d(c(nil))), (a(b(b(nil))), c(d(d(nil)))), ...},

and we find that it is similar to the semantics of the above logic program.

In section three, we briefly review the syntax and the least fixpoint semantics of a logic program. The semantics by the least Herbrand model and the operational semantics are not presented, because we are going to discuss by using only the least fixpoint semantics.

In section four, we define the equivalency of both semantics. This is the base of our

discussion to obtain the relationship. For example, we say that the above term-based CCFG program and logic program have the same meaning because both semantics are equivalent.

In section five, we give the program transformation rule from an arbitrary logic program into the t-CCFG program which is equivalent to the original logic program. This rule is so simple and we can automatically transform one definite clause in a logic program into one rule-set in a t-CCFG program. The correctness of the rule is proved by using the homogeneous form of the logic program, in which the equality symbol plays an important role.

In section six, according to the transformation rule described in section five, some typical logic programs are transformed into t-CCFG programs, and discuss the similarities and differences of both programs. We will show that the syntactic structures of both programs are *dual* to each other.

In section seven, the program transformation rule from an arbitrary t-CCFG to the equivalent logic program is defined. This is easy to understand because this is the inverse transformation rule described in section five.

2.Term-based CCFG programs

In this section we define the syntax of a term-based CCFG program and its semantics by the language and the least fixpoint.

A string-based CCFG program defined in [1] contains the production rules whose right-hand sides are strings, while a term-based one contains the production rules whose right-hand sides are terms. The data structure of terms is simpler than that of strings, because terms need not satisfy the associative law, and the definition of the term-based CCFG program is the same as that of the string-based one.

Definition 1 (term) Given a finite set N of nonterminal symbols, a finite set F of functors (constructors) and a set of variables, **terms** are defined as follows,

(1) A functor of the arity 0, called a constant, is a term.

(2) A nonterminal symbol is a term.

(3) A variable is a term.

(4) If $t_1, ..., t_n$ are terms and f is a functor of the arity n, $f(t_1, ..., t_n)$ is a term.

Terms which contain no nonterminal symbols and no variables are called **ground terms**, and the set of all the ground terms is denoted by U(F). The set of terms which contain no nonterminal symbols is denoted by U(F, V). The set of all the terms is denoted by U(N, F, V).

The above definition is extended by adding a **metasymbol** "≈" and the following condition,

(5) If t and u are terms, $t \approx u$ is a term.

Here the commutative and distributive laws for the metasymbols hold, i.e.

$$t \approx u = u \approx t,$$

and

$$f(..., t \approx u, ...) = f(..., t, ...) \approx f(..., u, ...).$$

For this extended definition of terms, U(F, ≈), U(F, V, ≈) and U(N, F, V, ≈) are defined in the same way.

Notice that nonterminal symbols are newly introduced to construct terms. The metasymbol "≈" has the same functions as described in [1].

In the following discussions, we suppose that the name of a nonterminal symbol begins with an upper case letter and the name of a variable begins with the star "*".

Definition 2 (t-CCFG) **A term-based Coupled Context-Free Grammar (t-CCFG)** is a quintuple (N, F, V, Ω, R) where N is a finite set of nonterminal symbols, F a finite set of functors, V a set of variables, Ω a n-tuple (n≥1) of start symbols, and R a finite set of **rule-sets**. A start symbol is a nonterminal symbol. A rule-set is a finite set of production rules. A production rule is either a context-free rule of the form $X \rightarrow t$ which has one nonterminal symbol X in its left-hand side and a term *t* in its right-hand side, or a **left-hand-omitted-rule** of the form of term *t*. Any two context-free rules in a rule-set must not have the same nonterminal symbols in their left-hand sides.

Example 1 The following quintuple is a t-CCFG,

$\mathbf{fib.G} = (N_{fib.G}, F_{fib.G}, V_{fib.G}, \Omega_{fib.G}, R_{fib.G})$,

$N_{fib.G} = \{Fin, Fout, Fin', Fout', Plus_1, Plus_2, Plus_3, Num\}$,

$F_{fib.G} = \{0, s\}$,

$V_{fib.G} = \emptyset$,

$\Omega_{fib.G} = (Fin, Fout)$,

$R_{fib.G} = \{$ $\{Fin \rightarrow 0, \quad Fout \rightarrow s(0)\}$, ...(r.1)

$\{Fin \rightarrow s(0), \ Fout \rightarrow s(0)\}$, ...(r.2)

$\{Fin \rightarrow s(Fin) \approx s(s(Fin')), Fout \approx Plus_1, Fout' \approx Plus_2, Fout \rightarrow Plus_3\}$, ...(r.3)

$\{Fin' \rightarrow Fin, Fout' \rightarrow Fout\}$, ...(r.4)

$\{Plus_1 \rightarrow 0, Plus_2 \rightarrow Num, Plus_3 \rightarrow Num\}$, ...(r.5)

$\{Plus_1 \rightarrow s(Plus_1), Plus_2 \rightarrow Plus_2, Plus_3 \rightarrow s(Plus_3)\}$, ...(r.6)

$\{Num \rightarrow 0\}$, ...(r.7)

$\{Num \rightarrow s(Num)\}\}$. ...(r.8)

In this case, no variables appear. Here, if the functor: s is interpreted as the successor function for integer data, then the nonterminal symbol Num derives all the integers, $Plus_3$ (more precisely the values of ground terms generated by $Plus_3$) represents the sum of $Plus_1$ and $Plus_2$, and Fout represents the Fibonacci number of Fin. In the following examples we mainly refer this t-CCFG.

Example 2 The following quintuple is a t-CCFG,

car.G = ($N_{car.G}$, $F_{car.G}$, $V_{car.G}$, $\Omega_{car.G}$, $R_{car.G}$),

$N_{car.G}$ = {InputList, CarList},

$F_{car.G}$ = {nil, cons},

$V_{car.G}$ = {*X, *Y},

$\Omega_{car.G}$ = (InputList, CarList),

$R_{car.G}$ = { {InputList→nil, CarList→nil}, ...(r.9)

{InputList→cons(*X, *Y), CarList→*X}}. ...(r.10)

In this example, $V_{car.G}$ is not empty. This is the grammar to express the Lisp-like car operation.

Next we define the derivation rule for a t-CCFG. First we define numbered nonterminal symbols in the same way as in the string-based CCFG.

Definition 3 (numbered nonterminal symbols) For a nonterminal symbol X and an integer k ($k \geq 0$), the **nonterminal symbol numbered with** k is expressed as X[k]. For a term t, t[k] expresses the same term as t except that all the nonterminal symbols in t are numbered with k, and is said to be numbered with k. For a tuple $\Omega=(t_1, ..., t_n)$ of terms, Ω[k] expresses (t_1[k], ..., t_n[k]). For a set N of nonterminal symbols, N[k] expresses {X[k] | X∈N}.

Given a set N of nonterminal symbols, a set F of functors and a set V of variables, the set of all the numbered nonterminal symbols X[m] ($X \in N$, $m \geq 0$) is denoted by N[ω]. The sets U(N[ω], F, V) and U(N[ω], F, V, ≈) are defined in the same way as U(N, F, V) and U(N, F, V, ≈).

Definition 4 (substitutions) Given a t-CCFG G=(N, F, V, Ω, R), the substitution θ={X[m]/t} (X[m]∈N[ω], t∈U(N[ω], F, V)) is a mapping from U(N[ω], F, V) into U(N[ω], F, V) defined as follows,

(1) $c\theta = c$, if c is a constant.

(2) $*X\theta = *X$.

(3) $Y[n]\theta = t$, if X = Y∈N and m = n,

$Y[n]\theta = Y[n]$, otherwise.

(4) $f(u_1, ..., u_i)\theta = f(u_1\theta, ..., u_i\theta)$.

To avoid confusion of variable names, if necessary, some variables in t should be replaced by new ones, i.e. when θ={X[m]/t} is applied to u, there must be no common variables in u and t.

Extending this definition, the substitution $\theta=\{X[m]/t_1\approx...\approx t_n\}$ (t_1, ..., $t_n\in$ U(N[ω], F, V) , n≥1) is a mapping from U(N[ω], F, V) into U(N[ω], F, V, ≈) defined as follows,

(5) $u\theta = u\theta_1\approx...\approx u\theta_n$, if X[m] appears in u,

$u\theta = u$, otherwise,

where $\theta_i=\{X[m]/t_i\}$ (n≥i≥1). Further extending the definition, the substitution θ from U(N[ω], F, V, ≈) into U(N[ω], F, V, ≈) is defined as follows,

(6) $(u_1\approx...\approx u_n)\theta = u_1\theta\approx...\approx u_n\theta$,

where $u_i\in$ U(N[ω], F, V) (n≥i≥1). The substitution θ from a tuple $(u_1, ..., u_n)\in$ U(N[ω], F, V, ≈)×...×U(N[ω], F, V, ≈) into a tuple is defined as follows,

(7) $(u_1, ..., u_n)\theta = (u_1\theta, ..., u_n\theta)$,

where $u_i\in$ U(N[ω], F, V, ≈) (n≥i≥1). A substitution $\{X_1[m_1]/t_1\}...\{X_k[m_k]/t_k\}$ ($m_i\geq 0$, k≥i≥1) is expressed as $\{X_1[m_1]/t_1, ..., X_k[m_k]/t_k\}$ for the convenience if any two numbered nonterminal symbols $X_i[m_i]$ and $X_j[m_j]$ are not the same.

The substitution $\phi=\{*X/t\}$ ($*X\in V$, $t\in$ U(F)) of ground terms into variables is defined in the same way as above except the following two rules.

(2′) $*Y\phi = t$, if $*X = *Y\in V$,

$*Y\phi = *Y$, otherwise.

(3′) $Y[n]\phi = Y[n]$.

The substitution $\{*X_1/t_1\}...\{*X_k/t_k\}$ (k≥1) is expressed as $\{*X_1/t_1, ..., *X_k/t_k\}$. The set of all of such $\{*X_1/t_1, ..., *X_k/t_k\}$'s is denoted by Φ_G.

Definition 5 (tuples of sentential forms) Given a t-CCFG G=(N, F, V, Ω, R), a **tuple** Ω_k (k≥0) **of sentential forms** is defined as follows.

(1) The tuple Ω[0] numbered with zero is the start tuple Ω_0 of sentential forms.

(2) The tuple Ω_{k+1} is a tuple of sentential forms if Ω_k is a tuple of sentential forms and satisfies the relation $\Omega_k\Rightarrow_G\Omega_{k+1}$.

Here the relation $\Rightarrow_G$ is described in the next definition.

There are two derivation rules in t-CCFG. One is to rewrite nonterminal symbols in the same way as in the string-based CCFG, and the other is to rewrite variables.

Definition 6 (derivation rule-1) Given a t-CCFG G=(N, F, V, Ω, R) and a tuple of sentential forms $\Omega_k=(\omega_1, ..., \omega_n)$ (k≥0), let $N_{\Omega k}$ be the set of all the numbered nonterminal symbols which appear in Ω_k and $Left_r$ be the set $\{X_1, ..., X_p\}$ of all the nonterminal symbols which appear in the left-hand sides of a rule-set

$$r = \{X_1 \rightarrow t_1, ..., X_p \rightarrow t_p, u_1, ..., u_q\} \in R, (p \geq 1, q \geq 0),$$

where $u_1, ..., u_q$ are left-hand-omitted-rules. If it holds that for a certain integer m≥0,

$$Left_r[m] \subseteq N_{\Omega k},$$

then the relation $\Omega_k \Rightarrow_G \Omega_{k+1}$ is defined as follows,

$$\Omega_{k+1} = \Omega_k\{X_1[m]/t_1[k+1], ..., X_p[m]/t_p[k+1]\}:(u_1[k+1], ..., u_q[k+1]),$$

where $\{X_1[m]/t_1[k+1], ..., X_p[m]/t_p[k+1]\}$ is a substitution, and ":" is a concatenation operator for tuples.

Definition 7 (derivation rule-2) Given a t-CCFG G=(N, F, V, Ω, R) and a tuple of sentential forms Ω_k (k≥0), for the substitution $\phi \in \Phi_G$ the relation $\Omega_k \Rightarrow_G \Omega_{k+1}$ is defined as

$$\Omega_{k+1} = \Omega_k\phi.$$

We often abbreviate the relation $\Omega_k \Rightarrow_G \Omega_{k+1}$ as the relation $\Omega_k \Rightarrow \Omega_{k+1}$ if G is clear in the context. The sequence $\Omega_0 \Rightarrow \Omega_1 \Rightarrow ... \Rightarrow \Omega_k \Rightarrow ...$ is said to be a **derivation sequence**. The reflective and transitive closure of ⇒ is expressed as $\Rightarrow^*$.

Next we define the language generated by a t-CCFG just like a string-based Coupled Context-Free Language (CCFL) in [1]. In the usual sense of the theory of formal grammars, a language is defined as a set of terminal strings. In our case, however, it is a relation over ground terms, a subset of U(F)×...×U(F) where F is given in a t-CCFG G=(N, F, V, Ω, R).

Definition 8 (t-CCFL) Given a t-CCFG G=(N, F, V, Ω, R), where Ω is a n-tuple, the **term-based Coupled Context-Free Language (t-CCFL)** L(G) is a set of n-tuples of ground terms defined as follows,

$$L(G) = \{(t_1, ..., t_n) \mid \Omega_0 \Rightarrow^* (\omega_1, ..., \omega_n, ..., \omega_m),\ \omega_i = t_i \approx ... \approx t_i,\ t_i \in U(F),\ m \geq i \geq 1\},$$

where the term $t_i \approx ... \approx t_i$ expresses one of t_i, $t_i \approx t_i$, $t_i \approx t_i \approx t_i$, ... We call an element in L(G) a **solution** of G. If a derivation sequence generates a solution, we call it a **successful** one. Otherwise it is a **failed** one.

Now we define t-CCFG programs.

Definition 9 (t-CCFG program) A t-CCFG G is a **t-CCFG program**, and its semantics SF_L[G] is defined by the t-CCFL L(G).

Example 3 (continued) The t-CCFG **fib.G** is a t-CCFG program. One example of the successful derivations by **fib.G** is given as follows,

(Fin[0], Fout[0])

$\Rightarrow$(s(Fin[1])$\approx$s(s(Fin´[1])), $Plus_3$[1],

Fout[1]$\approx$$Plus_1$[1], Fout´[1]$\approx$$Plus_2$[1]) by (r.3)

$\Rightarrow$(s(s(Fin[2]))$\approx$s(s(s(Fin´[2])))$\approx$s(s(Fin´[1])), $Plus_3$[1],

$Plus_3$[2]$\approx$$Plus_1$[1], Fout´[1]$\approx$$Plus_2$[1],

Fout[2]$\approx$$Plus_1$[2], Fout´[2]$\approx$$Plus_2$[2]) by (r.3)

$\Rightarrow$(s(s(s(0)))$\approx$s(s(s(Fin´[2])))$\approx$s(s(Fin´[1])), $Plus_3$[1],

$Plus_3$[2]$\approx$$Plus_1$[1], Fout´[1]$\approx$$Plus_2$[1],

s(0)$\approx$$Plus_1$[2], Fout´[2]$\approx$$Plus_2$[2]) by (r.2)

$\Rightarrow$(s(s(s(0)))$\approx$s(s(s(Fin[4])))$\approx$s(s(Fin´[1])), $Plus_3$[1],

$Plus_3$[2]$\approx$$Plus_1$[1], Fout´[1]$\approx$$Plus_2$[1],

s(0)$\approx$$Plus_1$[2], Fout[4]$\approx$$Plus_2$[2]) by (r.4)

...

$\Rightarrow^*$(s(s(s(0)))$\approx$s(s(s(0)))$\approx$s(s(s(0))), s(s(s(0))),

s(s(0))$\approx$s(s(0)), s(0)$\approx$s(0), s(0)$\approx$s(0), s(0)$\approx$s(0)).

This derivation is successful and the solution is the couple (s(s(s(0))), s(s(s(0)))) of ground terms which can be interpreted as that the third element of the Fibonacci sequence is three. In the same way, the language L(**fib.G**), that is the semantics SF_L[**fib.G**], is obtained as

L(**fib.G**) = {(0, s(0)), (s(0), s(0)), (s(s(0)), s(s(0))), (s(s(s(0))), s(s(s(0)))), ...}.

An element in L(**fib.G**) is (s^m(0), s^n(0)) for m$\geq$0, where the numbers m and n satisfy that the m-th number of the Fibonacci sequence is n. Namely this program means the computation of the Fibonacci sequence.

Example 4 (continued) For the t-CCFG **car.G**, the start tuple is rewritten as

(InputList[0], CarList[0])⇒(nil, nil), by (r.9)

or

(InputList[0], CarList[0])⇒(cons(*X, *Y), *X). by (r.10)

The former gives a simple solution. The latter can further derive the solutions, such as (cons(nil, nil), nil), (cons(cos(nil,nil),nil), cons(nil,nil)), by applying the derivation rule-2. In this way, we see that the derived couple (x, y) of two ground terms satisfies that **car**$(x)=y$ over U({nil, cons}), where **car** is a Lisp's function.

Next we define the least fixpoint semantics of a t-CCFG program as well as that of the string-based one.

Definition 10 (ground substitution) For a t-CCFG program G=(N, F, V, Ω, R), the **ground substitution** $\Theta=\{X_1/t_1, ..., X_k/t_k\}$ ($X_i\in N$, $t_i\in U(F)$, $k\geq i\geq 1$) over G is a mapping from U(N, F, V, ≈) into U(N, F, V, ≈). The definition of the ground substitution is the same as the definition 4 except that the nonterminal symbols are not numbered. If $X_1, ..., X_m, Y_1, ..., Y_n$ ($m,n\geq 1$) are distinct nonterminal symbols, the two ground substitutions $\{X_1/t_1, ..., X_m/t_m\}$ and $\{Y_1/u_1, ..., Y_n/u_n\}$ are said to be **disjoint.**

Definition 11 Given a t-CCFG program G=(N, F, V, Ω, R), let B_G be a set of all the ground substitutions over G, and $P(B_G)$ be the set of all the subsets over B_G. The mapping T_G over $P(B_G)$ is defined as follows,

$$
\begin{aligned}
T_G(A) = \{\{X_1/x_1, ..., X_p/x_p\} \mid\ & \\
& \{X_1\rightarrow t_1, ..., X_p\rightarrow t_p, u_1, ..., u_q\}\in R, \\
& \exists\Theta_1, ..., \Theta_k\in A\ (k\geq 0), \text{ any two } \Theta_i \text{ and } \Theta_j\ (1\leq i, j\leq k) \text{ are disjoint}, \\
& \exists\phi\in\Phi_G, \\
& t_i\Theta_1...\Theta_k\phi = x_i\approx...\approx x_i, \quad x_i\in U(F) \quad \text{for all } 1\leq i\leq p, \\
& u_j\Theta_1...\Theta_k\phi = y_j\approx...\approx y_j, \quad y_j\in U(F) \quad \text{for all } 1\leq j\leq q\},
\end{aligned}
$$

where $A\in P(B_G)$.

The domain $P(B_G)$ is a complete lattice when we define the order $A\leq B$ ($A, B\in P(B_G)$) as the

inclusion relation $A \subseteq B$. The least upper bound of X ($\subseteq P(B_G)$) is the union of all the subsets in X. The top element of this lattice is the total set B_G itself, and the bottom element is the empty set Ø. It can be easily proved that the mapping T_G is continuous. Therefore there exists the least fixpoint of T_G and it is equivalent to $T_G \uparrow \omega$, that is

$$T_G \uparrow \omega = \emptyset \cup T_G(\emptyset) \cup T_G^2(\emptyset) \cup ...,$$

where $T_G^n(\emptyset)$ denotes Ø for n=0 and $T_G(T_G^{n-1}(\emptyset))$ for $n \geq 1$.

Definition 12 (the least fixpoint semantics) The semantics $SF_F[G]$ of a t-CCFG program G=(N, F, V, (S_1, ..., S_n), R) is given as follows,

$$SF_F[G] = \{(t_1, ..., t_n) \mid \{S_1/t_1, ..., S_n/t_n\} \in T_G \uparrow \omega\}.$$

Example 5 (continued) For the t-CCFG **fib.G**,

$T_{fib.G}(\emptyset)$ = {{Num/0}, {Fin/0, Fout/s(0)}, {Fin/s(0), Fout/s(0)}},

$T_{fib.G}^2(\emptyset) = T_{fib.G}(\emptyset) \cup$ {{Num/s(0)}, {$Plus_1$/0, $Plus_2$/0, $Plus_3$/0},
{Fin´/0, Fout´/s(0)}, {Fin´/s(0), Fout´/s(0)} },

$T_{fib.G}^3(\emptyset) = T_{fib.G}^2(\emptyset) \cup$ {{Num/s(s(0))}, {$Plus_1$/0, $Plus_2$/s(0), $Plus_3$/s(0)},
{$Plus_1$/s(0), $Plus_2$/0, $Plus_3$/s(0)}},

$T_{fib.G}^4(\emptyset) = T_{fib.G}^3(\emptyset) \cup$ {{Num/s(s(s(0)))}, {$Plus_1$/0, $Plus_2$/s(s(0)), $Plus_3$/s(s(0))},
{$Plus_1$/s(0), $Plus_2$/s(0), $Plus_3$/s(s(0))},
{$Plus_1$/s(s(0)), $Plus_2$/0, $Plus_3$/s(s(0))}},

...

Therefore

SF_F[**fib.G**] = {(0, s(0)), (s(0), s(0)), (s(s(0)), s(s(0))), (s(s(s(0))), s(s(s(0)))), ...}.

Theorem 1 (identicalness) For a t-CCFG program G, it holds that $SF_L[G] = SF_F[G]$.

This proof is omitted because it is rather lengthy though it can be carried out straightforwardly.

3.Overview of logic programs

In this subsection we overview the syntax of a logic program (a pure Prolog program) and its least fixpoint semantics [2][3]. Other semantics are omitted since we do not use them in the following sections.

Definition 13 (atom) Given a set F of functors and a set of V of variables, if p is an n-ary predicate symbol and t_1, ..., t_n are terms in U(F, V), then p(t_1, ..., t_n) is an atomic formula or, more simply, called an **atom**. If t_1, ..., t_n are ground terms, the atom is called a **ground atom**.

In this paper, we do *not* consider the predicate symbols whose arities are *zero*.

Definition 14 (logic program) A **definite Horn clause** is a formula of the form

$$p(t_1, ..., t_i) :- q_1(u_{1,1}, ..., u_{1,f1}), ..., q_n(u_{n,1}, ..., u_{n,fn}) \quad (i \geq 1, n \geq 0, n \geq j \geq 1, fj \geq 0),$$

where p, q_1, ..., q_n are predicate symbols, t_1, ..., t_i, $u_{1,1}$, ..., $u_{n,fn}$ are terms. A **logic program** is defined by the quintuple (P, F, V, s, D), where P is a finite set of predicate symbols, F a finite set of functors, V a set of variables, s is a predicate symbol in P, and D a finite set of definite Horn clauses.

Here the definition of a logic program is different from the usual one, because our purpose is to clarify the relationship between t-CCFG programs and logic programs. We relate the quituple of a t-CCFG program with the quintuple of a logic program in section five.

Example 6 The following quintuple is a logic program,

fib.L = ($P_{fib.L}$, $F_{fib.L}$, $V_{fib.L}$, fib, $D_{fib.L}$),

$P_{fib.L}$ = {fib, plus, num},

$F_{fib.L}$ = {0, s},

$V_{fib.L}$ = {*W, *X, *Y, *Z, ...},

$D_{fib.L}$ = { fib(0, s(0)), ...(d.1)

fib(s(0), s(0)), ...(d.2)

fib(s(s(*X)), *Y):-fib(s(*X), *Z), fib(*X, *W), plus(*Z, *W, *Y), ...(d.3)

plus(0, *X, *X):-num(*X), ...(d.5)

plus(s(*X), *Y, s(*Z)):-plus(*X, *Y, *Z), ...(d.6)

num(0), ...(d.7)

num(s(*X)):-num(*X)} ...(d.8)

We will show later that the set $D_{fib.L}$ in **fib.L** corresponds to the set $R_{fib.G}$ of rule-sets in **fib.G**. The reason why we do not define the clause (d.4) in $D_{fib.L}$ is because the corresponding rule-set (r.4) is only an auxiliary one in **fib.G**.

Definition 15 (substitution) Given a logic program L=(P, F, V, s, D), the substitution mapping $\phi=\{*X_1/t_1, ..., *X_k/t_k\}$ ($*X_i \in V$, $t_i \in U(F)$) from U(F, V) into U(F, V) is defined in the same way as $\phi \in \Phi_G$ in the definition 4. The set of all of such ϕ is denoted by Φ_L.

Definition 16 Given a logic program L=(P, F, V, s, D), let B_L be the set of all the ground atoms over L, and $P(B_L)$ be the set of all the subsets over B_L. The mapping T_L over $P(B_L)$ is defined as follows,

$$T_L(A) = \{\ p(t_1, ..., t_i)\phi \mid\ p(t_1, ..., t_i) :- q_1(u_{1,1}, ..., u_{1,f1}), ..., q_n(u_{n,1}, ..., u_{n,fn}) \in D,$$
$$\exists_{\phi \in \Phi_L},$$
$$p(t_1, ..., t_i)\phi \in B_L,$$
$$q_i(u_{i,1}, ..., u_{i,fi})\phi \in A, \text{ for all } n \geq i \geq 1\},$$

where $A \in P(B_L)$.

In the same way as $P(B_G)$ in the previous section, the domain $P(B_L)$ is a complete lattice when we define the order $A \leq B$ ($A, B \in P(B_L)$) as the inclusion relation $A \subseteq B$. The mapping T_L is continuous, and there exists the least fixpoint $T_L \uparrow \omega$ similar to $T_G \uparrow \omega$ in the previous section.

Definition 17 (the least fixpoint semantics) The semantics $SF_F[L]$ of a logic program L=(P, F, V, s, D) is given as follows,

$$SF_F[L] = \{(t_1, ..., t_n) \mid s(t_1, ..., t_n) \in T_L \uparrow \omega\}.$$

Example 7 (continued) For the logic program **fib.L**,

$T_{fib.L}(\emptyset)$ = {num(0), fib(0, s(0)), fib(s(0), s(0))},

$T_{fib.L}^{2}(\emptyset) = T_{fib.L}(\emptyset) \cup \{num(s(0)), plus(0, 0, 0)\}$,

$T_{fib.L}^{3}(\emptyset) = T_{fib.L}^{2}(\emptyset) \cup \{num(s(s(0))), plus(0, s(0), s(0)), plus(s(0), 0, s(0))\}$,

$T_{fib.L}^{4}(\emptyset) = T_{fib.L}^{3}(\emptyset) \cup \{num(s(s(s(0)))), plus(0, s(s(0)), s(s(0))),$

$plus(s(0), s(0), s(s(0))), plus(s(s(0)), 0, s(s(0)))\}$,

...

Then

SF_F[**fib.L**] = {(0, s(0)), (s(0), s(0)), (s(s(0)), s(s(0))), (s(s(s(0))), s(s(s(0)))), ...}.

We omit the details of logic programs (see [2] for the details), since our major objective in this paper is to clarify the characteristics of CCFG programs through discussing the relationship between CCFG and logic programs rather than to describe the details of logic programs.

4.Equivalence of semantics

In the previous two section we define the least fixpoint semantics of a t-CCFG program and a logic programs. Here we define the equivalence of them by comparing their semantics.

Definition 18 The correspondence between an n-ary predicate symbol p and an n-tuple (P_1, ..., P_n) of nonterminal symbols is expressed by the following naming convention

$$\langle p,1\rangle = P_1, \ldots, \langle p,n\rangle = P_n.$$

The symbol $\langle p,i\rangle$ ($n \geq i \geq 1$) is identified with the symbol P_i.

Given a logic program L and a t-CCFG program G, let A be a set ($\in P(B_L)$) of ground atoms and A´ be a set ($\in P(B_G)$) of ground substitutions. If it holds that

$$A' = \{\{\langle p,1\rangle/t_1, \ldots, \langle p,n\rangle/t_n\} \mid p(t_1, \ldots, t_n) \in A\},$$

or that

$$A = \{p(t_1, \ldots, t_n) \mid \{\langle p,1\rangle/t_1, \ldots, \langle p,n\rangle/t_n\} \in A'\},$$

we say that A and A´ are **equivalent** each other, and it is expressed as $A \cong A'$.

Definition 19 (Equivalence of programs) Given a logic program L and a t-CCFG program G, we say that G is **equivalent** to L if it holds that $SF_F[L]=SF_F[G]$.

Example 8 (continued) The t-CCFG program **fib.G** is equivalent to the logic program **fib.L**, because SF_F[**fib.L**]=SF_F[**fib.G**].

5.Transformation from logic programs into t-CCFG programs

In this section we give the transformation rule from a logic program into the t-CCFG program which preserve their semantics, and prove the correctness of the transformation rule.

5.1.Transformation rules

First the preprocessing procedure for logic programs is defined.

Transformation rule $\mathcal{P}$ (preprocessing) For every definite Horn clause d in a logic program L=(P, F, V, s, D), if a predicate symbol q∈ P appears more than once in the body of the clause as in

$$d = \text{p(...):-..., q(...), ..., q(...), ..., q(...), ...,}$$

then suffix these q's with the numbers 2, ..., k (k≥2) as follows in order to distinguish them clearly.

$$\mathcal{P}(d) = \text{p(...):-..., q(...), ..., } q_2\text{(...), ..., } q_3\text{(...), ...,}$$

Otherwise $\mathcal{P}(d)=d$.

The preprocessed logic program is expressed as $\mathcal{P}$(L), which is a quintuple (P∪P′, F, V∪V′, s, D′∪D′′), where

$P' = \{q_j \mid q_j$ is a suffixed predicate symbol in a $\mathcal{P}(d)\}$,

$D' = \{\mathcal{P}(d) \mid d \in D\}$,

$D'' = \{q_j(*X_1, ..., *X_n)\text{:-}q(*X_1, ..., *X_n) \mid q_j \in P'\}$,

and V′ is a set of newly introduced variables in D′′.

Example 9 (continued) For every definite clause d in **fib.L**, it holds that $d=\mathcal{P}(d)$, except the clause (d.3) which is transformed into

$$\text{fib(s(s(*X)), *Y):-fib(s(*X), *Z), } \text{fib}_2\text{(*X, *W), plus(*Z, *W, *Y).} \quad ...\mathcal{P}(\text{d.3})$$

Then introducing the following definite clause

$$\text{fib}_2\text{(*X, *Y):-fib(*X, *Y),} \quad ...(\text{d.4})$$

$\mathcal{P}$(**fib.L**) is obtained as the quintuple ($P_{fib.L}$∪{fib_2}, $F_{fib.L}$, $V_{fib.L}$, fib, (D-(d.3))∪{$\mathcal{P}$(d.3), (d.4)}).

Transformation rule $\mathcal{R}$ For a preprocessed logic program L = (P, F, V, s, D), let ‹p,j›

($i \geq j \geq 1$) be the nonterminal symbol corresponding to the j-th argument of an i-ary predicate symbol $p \in P$. Transform the following definite Horn clause,

$$d = p(t_1, ..., t_i) :- q_1(u_{1,1}, ..., u_{1,f1}), ..., q_n(u_{n,1}, ..., u_{n,fn})$$

into the following rule-set,

$$\mathfrak{R}(d) = \{\langle p,1\rangle \rightarrow t_1, ..., \langle p,i\rangle \rightarrow t_i, \ \langle q_1,1\rangle \approx u_{1,1}, ..., \langle q_1,f1\rangle \approx u_{1,f1},$$
$$...,$$
$$\langle q_n,1\rangle \approx u_{n,1}, ..., \langle q_n,fn\rangle \approx u_{n,fn}\}.$$

And transform L into the following t-CCFG program $\mathfrak{R}(L)$,

$$\mathfrak{R}(L) = (N, F, V, \Omega, R),$$
$$N = \{\langle p,j\rangle \mid m \geq j \geq 1, \text{ m is the arity of } p \in P\},$$
$$\Omega = (\langle s, 1\rangle, ..., \langle s, n\rangle)$$
$$R = \{\mathfrak{R}(d) \mid d \in D\}.$$

Example 10 (continued) The clauses in the logic program $\mathfrak{P}$(**fib.L**) are transformed into the following rule-sets,

$\{\langle fib,1\rangle \rightarrow 0, \langle fib,2\rangle \rightarrow s(0)\}$, ...$\mathfrak{R}$(d.1)

$\{\langle fib,1\rangle \rightarrow s(0), \langle fib,2\rangle \rightarrow s(0)\}$, ...$\mathfrak{R}$(d.2)

$\{\langle fib,1\rangle \rightarrow s(s(*X)), \langle fib,2\rangle \rightarrow *Y,$
$\langle fib,1\rangle \approx s(*X), \langle fib,2\rangle \approx *Z, \langle fib_2,1\rangle \approx *X, \langle fib_2,2\rangle \approx *W,$
$\langle plus,1\rangle \approx *Z, \langle plus,2\rangle \approx *W, \langle plus,3\rangle \approx *Y\}$, ...$\mathfrak{R}(\mathfrak{P}$(d.3))

$\{\langle fib_2,1\rangle \rightarrow *X, \langle fib_2,2\rangle \rightarrow *Y, \langle fib,1\rangle \approx *X, \langle fib,2\rangle \approx *Y\}$, ...$\mathfrak{R}$(d.4)

$\{\langle plus,1\rangle \rightarrow 0, \langle plus,2\rangle \rightarrow *X, \langle plus,3\rangle \rightarrow *X, \langle num,1\rangle \approx *X\}$, ...$\mathfrak{R}$(d.5)

$\{\langle plus,1\rangle \rightarrow s(*X), \langle plus,2\rangle \rightarrow *Y, \langle plus,3\rangle \rightarrow s(*Z),$
$\langle plus,1\rangle \approx *X, \langle plus,2\rangle \approx *Y, \langle plus,3\rangle \approx *Z\}$, ...$\mathfrak{R}$(d.6)

$\{\langle num,1\rangle \rightarrow 0\}$, ...$\mathfrak{R}$(d.7)

$\{\langle num,1\rangle \rightarrow s(*X), \langle num,1\rangle \approx *X\}$. ...$\mathfrak{R}$(d.8)

Then the logic program $\mathfrak{P}$(**fib.L**) can be transformed into the following t-CCFG,

$$\mathfrak{R}(\mathfrak{P}(\textbf{fib.L})) = (N_{fib.L}, F_{fib.L}, V_{fib.L}, \Omega_{fib.L}, R_{fib.L}),$$
$$N_{fib.L} = \{\langle fib,1\rangle, \langle fib,2\rangle, \langle fib_2,1\rangle, \langle fib_2,2\rangle, \langle plus,1\rangle, \langle plus,2\rangle, \langle plus,3\rangle, \langle num,1\rangle\},$$
$$\Omega_{fib.L} = (\langle fib, 1\rangle, \langle fib, 2\rangle)$$
$$R_{fib.L} = \{\mathfrak{R}(d.1), \mathfrak{R}(d.2), \mathfrak{R}(\mathfrak{P}(d.3)), \mathfrak{R}(d.4), \mathfrak{R}(d.5), \mathfrak{R}(d.6), \mathfrak{R}(d.7), \mathfrak{R}(d.8)\}.$$

We see that this program is equivalent to **fib.G**, and the rule-sets $\mathfrak{R}$(d.1), ..., $\mathfrak{R}$(d.8) correspond to the (r.1), ..., (r.8), respectively.

In this way, an arbitrary logic program can be transformed into a t-CCFG program. These rules are simple and give the basis to discuss the relationship between logic programs and CCFG programs.

5.2.Correctness

The correctness of the transformation rule $\mathfrak{P}$ is obvious. Therefore its proof is omitted.

The correctness of the transformation rule $\mathfrak{R}$ is proved here by using the least fixpoint semantics by induction.

Lemma 1 Given a logic program L and a t-CCFG program G, it holds that SF_F[L]=SF_F[G] if $T_L\uparrow\omega \cong T_G\uparrow\omega$.

The proof is trivial.

Theorem 2 For every preprocessed logic program L, the transformation rule $\mathfrak{R}$ preserves the semantics. Namely it holds that

$$SF_F[L] = SF_F[\mathfrak{R}(L)].$$

proof We prove that $T_L^{\,n}(\emptyset) \cong T_{\mathfrak{R}(L)}^{\,n}(\emptyset)$ $(n \geq 0)$ by induction.

Basic step: It is clear that $T_L^{\,0}(\emptyset) = T_{\mathfrak{R}(L)}^{\,0}(\emptyset) = \emptyset$. Therefore $T_L^{\,0}(\emptyset) \cong T_{\mathfrak{R}(L)}^{\,0}(\emptyset)$.

Inductive step: We assume that $T_L^{\,k}(\emptyset) \cong T_{\mathfrak{R}(L)}^{\,k}(\emptyset)$ for a certain k. The preprocessed logic program L contains the following clause,

$$p(t_1, ..., t_i) :- q_1(u_{1,1}, ..., u_{1,f1}), ..., q_n(u_{n,1}, ..., u_{n,fn}),$$

if and only if the t-CCFG program $\mathfrak{R}$(L) contains the following rule-set derived by the transformation rule $\mathfrak{R}$,

$$\{\langle p,1\rangle \rightarrow t_1, ..., \langle p,i\rangle \rightarrow t_i,\ \langle q_1,1\rangle \approx u_{1,1}, ..., \langle q_1,f1\rangle \approx u_{1,f1},$$

$$...,$$

$$\langle q_n,1\rangle \approx u_{n,1}, ..., \langle q_n,fn\rangle \approx u_{n,fn}\}.$$

From the assumption, for a substitution $\phi \in \Phi_L = \Phi_{\mathcal{R}(L)}$, $T_L{}^k(\emptyset)$ contains the ground atom $q_k(u_{k,1}, ..., u_{k,fk})\phi$ $(n \geq k \geq 1)$ if and only if $T_{\mathcal{R}(L)}{}^k(\emptyset)$ contains the ground substitution $\Theta_k = \{\langle q_k, 1\rangle / u_{k,1}\phi, ..., \langle q_k, fk\rangle / u_{k,fk}\phi\}$. In this case it holds that

$$(\langle q_k, j\rangle \approx u_{k,j})\Theta_k\phi = u_{k,j}\phi \approx u_{k,j}\phi. \qquad (n \geq k \geq 1,\ f_k \geq j \geq 1)$$

We can always obtain ϕ such as $t_j\phi$ is a ground term for all $i \geq j \geq 1$. Therefore, from the definitions of the least fixpoint semantics of both programs, $T_L{}^{k+1}(\emptyset)$ contains $p(t_1, ..., t_i)\phi$ if and only if $T_{\mathcal{R}(L)}{}^{k+1}(\emptyset)$ contains $\{\langle p,1\rangle / t_1\phi, ..., \langle p,i\rangle / t_i\phi\}$. Hence $T_L{}^{k+1}(\emptyset) \equiv T_{\mathcal{R}(L)}{}^{k+1}(\emptyset)$.

Thus it holds that $T_L{}^n(\emptyset) \equiv T_{\mathcal{R}(L)}{}^n(\emptyset)$ for any n. Therefore $T_L \uparrow \omega \equiv T_{\mathcal{R}(L)} \uparrow \omega$. At last, from the lemma 1, $SF_F[L] = SF_F[\mathcal{R}(L)]$.

In other words, every logic program L can be transformed into the equivalent t-CCFG program $\mathcal{R}(\mathcal{P}(L))$.

6.Intuitive relationship between CCFG programs and logic programs

Although the correct transformation from a logic program into a t-CCFG program has been defined, it does not clearly show us the intuitive relationship between CCFG and logic programs yet.

In subsection 6.1 we introduce some simple transformation rules on CCFG programs, which eliminate the redundancies of the CCFG programs. For a logic program L, we may find some redundancies of the transformed t-CCFG program $\mathfrak{R}(\mathfrak{P}(L))$. For example, the text size of the transformed rule-set $\mathfrak{R}(\mathfrak{P}(d.3))$ is bigger than that of its corresponding rule-set (r.3), because $\mathfrak{R}(\mathfrak{P}(d.3))$ has the variables *W, *X, *Y and *Z which mean arbitrary ground terms. Such variables express only the indirect bindings between nonterminal symbols, for example, the right-hand side s(s(*X)) in $\mathfrak{R}(\mathfrak{P}(d.3))$ is the same as s(‹fib,1›)≈s(s(‹fib_2,1›)) because there exist the left-hand-omitted-rules ‹fib,1›≈s(*X) and ‹fib_2,1›≈*X. In this subsection we obtain such transformation rules that {...→s(s(*X)), ‹fib,1›≈s(*X), ‹fib_2,1›≈*X, ...} automatically derives {...→s(‹fib,1›)≈s(s(‹fib_2,1›)), ...}.

In subsection 6.2 some typical examples of transformations from definite Horn clauses into rule-sets are given. Through the discussions on the result obtained by eliminating the redundancies, we can clearly understand the intuitive relationship between the structures of both programs.

6.1.Elimination of redundancies

Definition 20 (subterm) The subterm u of a term t is defined as follows,

(1) The term t is a subterm of t itself.

(2) if $t = f(u_1, ..., u_n)$, then u_i $(n \geq i \geq 1)$ is a subterm of t.

(3) if $t = u \approx v$, then u and v are subterms of t.

(4) if $t = f(u_1, ..., u_n) \approx f(v_1, ..., v_n)$, then $u_i \approx v_i$ $(n \geq i \geq 1)$ is a subterm of t.

(5) if u is a subterm of t and v is a subterm of u, then v is a subterm of t.

Simplifying rule-1 If a rule-set has a left-hand-omitted-rule $t \approx u$ and also has at least one other term which contains the subterm t, the latter t can be replaced by $t \approx u$, and the original left-hand-omitted-rule can be eliminated from the rule-set. Namely

$$\{ ..., ...t..., ..., t \approx u\} \Rightarrow \{..., ...t \approx u..., ...\}.$$

Simplifying rule-2 If a rule-set has a left-hand-omitted-rule of the form *X≈t, , all the variables *X in the rule-set can be replaced by t, and the left-hand-omitted-rule can be eliminated from the rule-set. Namely

$$\{..., ...*X..., ..., *X\approx t\} \Rightarrow \{..., ...t..., ...\}.$$

It is obvious that the above rules preserve the meanings of rule-sets.

There is no rule-set which the above rules can be endlessly applied to because the number of left-hand-omitted-rules in a rule-set is finite and it decreases every time applying the rules. The transformation procedure to eliminate the redundancies of a program is finitely terminating.

Example 11 (continued) In $\mathcal{R}(\mathcal{P}(\textbf{fib.L}))$, the rule-set $\mathcal{R}(\mathcal{P}(\text{d.3}))$ is transformed into the following rule-set by applying the simplifying rule-2 to the left-hand-omitted-rules ‹fib,2›≈*Z, ‹fib_2,1›≈*X, ‹fib_2,2›≈*W and ‹plus,1›≈*Z.

{‹fib,1›→s(s(‹fib_2,1›)), ‹fib,2›→‹plus,3›,

‹fib,1›≈s(‹fib_2,1›), ‹plus,1›≈‹fib,2›, ‹plus,2›≈‹fib_2,2›}. ... $\mathcal{R}(\mathcal{P}(\text{d.3}))'$

further applying the simplifying rule-1 to the left-hand-omitted-rule ‹fib,1›≈s(‹fib_2,1›) and the subterm s(‹fib_2,1›) of the right-hand side s(s(‹fib_2,1›)), the above rule-set is transformed into

{‹fib,1›→s(‹fib,1›)≈s(s(‹fib_2,1›)), ‹fib,2›→‹plus,3›,

‹plus,1›≈‹fib,2›, ‹plus,2›≈‹fib_2,2›}. ... $\mathcal{R}(\mathcal{P}(\text{d.3}))''$

In the same way, the rule-sets $\mathcal{R}$(d.5) and $\mathcal{R}$(d.6) are transformed into the following rule-sets,

{‹plus,1›→0, ‹plus,2›→‹num,1›, ‹plus,3›→‹num,1›}, ...$\mathcal{R}(\text{d.5})'$

{‹plus,1›→s(‹plus,1›), ‹plus,2›→‹plus,2›, ‹plus,3›→s(‹plus,3›)}. ...$\mathcal{R}(\text{d.6})'$

We find that the t-CCFG program $\mathcal{R}(\mathcal{P}(\textbf{fib.L}))$ with the above transformed rule-sets is the same as the t-CCFG program **fib.G** when the variable names ‹fib,1›, ‹fib,2›, ‹fib_2,1›, ‹fib_2,2›, ‹plus,1›, ‹plus,2›, ‹plus,3› and ‹num,1› are replaced by Fin, Fout, Fin´, Fout´, Plus_1, Plus_2 Plus_3 and Num, respectively.

6.2.Typical examples

The program transformation has interesting characteristics as described below. Here we discuss three kinds of the transformations.

Case 1

One of the most simple definite clauses is an unit clause which has empty body as follows,

$$p(t_1, .., t_n).$$

This is transformed into the rule-set of the form

$$\{\langle p,1\rangle \rightarrow t_1, ..., \langle p,n\rangle \rightarrow t_n\}.$$

We see that an unit clause is transformed into the rule-set which contains only context-free production rules, and contains no metasymbols $\approx$ and no left-hand-omitted-rules. If no variables appear in the unit clause, then no variables appear in the the transformed rule-set. Otherwise, some variables appear and they can not be eliminated because there are no left-hand-omitted-rules in the rule-set and the simplifying rule-1 and -2 can not be applied.

Example 12 The unit clauses composed of only ground terms as follows are called *facts*,

likes(joe, fish),

likes(joe, mary),

likes(mary, book),

likes(john, book).

These are directly transformed as follows,

{‹likes,1›→joe, ‹likes,2›→fish},

{‹likes,1›→joe, ‹likes,2›→mary},

{‹likes,1›→mary, ‹likes,2›→book},

{‹likes,1›→john, ‹likes,2›→book}.

The right-hand sides in the transformed rule-sets are composed of only ground terms.

Example 13 The unit clauses which express Lisp's primitive function **car** are given as follows,

car(nil, nil),

car(cons(*X, *Y), *X).

These are transformed into

{‹car,1›→nil, ‹car,2›→nil},

{‹cdr,1›→cons(*X, *Y), ‹cdr,2›→*X}.

These are already shown in the example in section two.

Case 2

In the following definite clause,

$$p(t_1, ..., t_i):-q_1(u_{1,1}, ..., u_{1,f1}), ..., q_n(u_{n,1}, ..., u_{n,fn}),$$

if every $u_{j,k}$ ($n \geq j \geq 1$, $fj \geq k \geq 1$) is a subterm of a term t_m ($i \geq m \geq 1$), it is transformed into the rule-set which has no left-hand-omitted-rules but may have metasymbols $\approx$ by replacing such a subterm $u_{j,}$ of t_m by $\langle q_j,k \rangle$. If at least two term $u_{j,k}$ and $u_{j',k'}$ is subterms of t_m, the right-hand side of the production rule $\langle p,m \rangle \rightarrow$... contains at least one metasymbol "$\approx$".

Example 14 The following clause may possibly appear in a program in the area of stream processing and satisfies the condition of the above case,

stream([*X|*Y]):-job1(*X), job2(*X), stream(*Y).

Here *X and *Y are subterms of [*X|*Y]. This is transformed as follows,

{‹stream,1›→[*X|*Y], ‹job1,1›≈*X, ‹job2,1›≈*X, ‹stream,1›≈*Y}.

Applying the simplifying rule-2, it is further transformed into the clause

{‹stream,1›→[‹job1,1›≈‹job2,1›|‹stream,1›]},

or the equivalent one

{‹stream,1›→[‹job1,1›|‹stream,1›]≈[‹job2,1›|‹stream,1›]}.

Case 3

In the following definite clause,

$$p(t_1, ..., t_i):-q_1(u_{1,1}, ..., u_{1,f1}), ..., q_n(u_{n,1}, ..., u_{n,fn}),$$

if there exists at least one term $u_{j,k}$ which appears in the body but does not appear in the head of the clause as a subterm, there is at least one left-hand-omitted-rules in the transformed rule-set which can not be eliminated from the rule-set by applying the simplifying rule-1 or -2. The reason is easy to understand when we compare this case to the previous case in which there are no left-hand-omitted-rules.

Example 15 The term *F appears in the body of the following clause as an argument of atoms father(*C, *F) and father(*F,*G), but does not appear in the head,

grandFather(*C,*G):-father(*C,*F),father(*F,*G).

This clause is transformed into

{‹grandFather,1›→*C, ‹grandFather,2›→*G,

‹father,1›≈*C, ‹father,2›≈*F, ‹father$_2$,1›≈*F, ‹father$_2$,2›≈*G}.

Applying the simplifying rule-2, it is further transformed into

{‹grandFather,1›→‹father,1›, ‹grandFather,2›→‹father$_2$,2›, ‹father,2›≈‹father$_2$,1›}.

The left-hand-omitted-rule ‹father,2›≈‹father$_2$,1› can never be eliminated from the rule-set.

7.Transformation from t-CCFG programs into logic programs

We have shown that an arbitrary logic program can be transformed into the equivalent t-CCFG program by applying the rule $\mathfrak{R}$ described in the section five. Here we discuss the inverse transformation rule $\mathfrak{R}^{-1}$ from a t-CCFG program into the equivalent logic program.

7.1.Transformation rule for the programs of canonical form

Since the semantic domains of a logic program and t-CCFG program is similar to each other and the transformation rule $\mathfrak{R}$ is simple, the inverse rule $\mathfrak{R}^{-1}$ is almost trivial except one problem. The following quintuple **ncf.G** is, for example, a t-CCFG program,

$$\mathbf{ncf.G} = (N_{nfc.G}, F_{nfc.G}, \emptyset, \Omega_{nfc.G}, R_{ncf.G}),$$

$$N_{nfc.G} = \{A, B, C\},$$

$$F_{nfc.G} = \{0, 1, 2, a, b, c, d, e\},$$

$$\Omega_{nfc.G} = (A, B, C),$$

$$R_{ncf.G} = \{ \quad \{A\rightarrow 0, B\rightarrow 1\}, \qquad \text{...(n.1)}$$

$$\{C\rightarrow 2\}, \qquad \text{...(n.2)}$$

$$\{A\rightarrow 0, B\rightarrow 0, C\rightarrow 0\}, \qquad \text{...(n.3)}$$

$$\{A\rightarrow d(A), B\rightarrow e(B)\}, \qquad \text{...(n.4)}$$

$$\{A\rightarrow a(A), B\rightarrow b(B), C\rightarrow c(C)\}\}. \qquad \text{...(n.5)}$$

This program generates the following set of triplets,

$$\{(a^i(0), b^i(0), c^i(0)) \mid i\geq 0\}\cup\{(a^i(d^j(0)), b^i(e^j(1)), c^i(2)) \mid i,j\geq 0\},$$

where $a^i(\alpha)$ express α for i=0 and $a(a^{i-1}(\alpha))$ for $i\geq 1$. The reason why the triplet of the form (d(a(...)), e(b(...)), c(...)) can not be derived is because the derivation proceeds with nonterminal symbols *numbered with the same integer* (see the definitions in section two). The start triplet (A[0], B[0], C[0]) of numbered nonterminal symbols can derive the triplet (d(A[1]), e(B[1]), C[0]) by applying the rule-set (n.4). In the next derivation step, A[1] and B[1] can be simultaneously replaced, but A[1], B[1] and C[0] can not be simultaneously replaced because their numbers are different. Then the rule-set (n.5) can not be applied to the triplet. We must pay particular attentions to such a confusing program.

Definition 21 (canonical form) The set of all the nonterminal symbols appearing in the left-hand sides of a rule-set *r* is denoted by left_set(*r*), and the tuple composed of all the symbols in

left_set(r) is denoted by left_tuple(r). We assume that for any two rule-sets r_1 and r_2

$$\text{left\_tuple}(r_1) = \text{left\_tuple}(r_2) \quad \text{if} \quad \text{left\_set}(r_1) = \text{left\_set}(r_2).$$

A t-CCFG program (N, F, V, Ω, R) is said to be of **canonical form** if either it holds that for any two rule-sets r_1 and r_2 in R, it holds either

$$\text{left\_set}(r_1) = \text{left\_set}(r_2) \quad \text{or} \quad \text{left\_set}(r_1) \cap \text{left\_set}(r_2) = \emptyset.$$

Remarks We have implicitly assumed that the arity of every predicate symbol in a logic program is fixed. It is easily seen that such a logic program is transformed into a t-CCFG program of the canonical form. And further even if the arity is not fixed, we can transform an arbitrary logic program into a t-CCFG program of the canonical form by treating a predicate symbol p with arity m and with arity n (m≠n) as distinct predicate symbols. For example, if a logic program has the following clauses,

p(d(*X), e(*Y)):-p(*X,*Y).

p(a(*X), b(*Y), c(*Z)):-p(*X, *Y, *Z).,

these can be transformed into

{‹p,1›→d(‹p,1›), ‹p,2›→e(‹p,2›)},

{‹p´,1›→a(‹p´,1›), ‹p´,2›→b(‹p´,2›), ‹p´,3›→c(‹p´,3›)}.

In this way all the logic programs can correspond to t-CCFG programs of the canonical form, and in this sense the canonical form is an important concept for the study of CCFG programs.

The program transformation rule $\mathfrak{R}^{-1}$ from a t-CCFG program of the canonical form into the equivalent logic program is gievn as follows. It is easier than the rule from an arbitrary t-CCFG program which is given in the next subsection.

Transformation rule $\mathfrak{R}^{-1}$ Given a t-CCFG program G=(N, F, V, Ω, R) of the canonical form, we can assume that for every left_tuple(r) for a rule-set r in G there is a corresponding predicate symbol p whose arity is the same as the size of left_tuple(r), and that the predicate symbols corresponding to any two rule-sets r_1 and r_2 are the same if left_tuple(r_1)=left_tuple(r_2).

Let

$$r = \{X_1 \rightarrow t_1, \ldots, X_i \rightarrow t_i, u_1, \ldots, u_j\}, \quad (i \geq 1, j \geq 0),$$

be a rule-set in G, where the right-hand sides t_1, ..., t_i, u_1, ..., u_j contain the nonterminal symbols Y_1, ..., Y_m. Since the program G is of the canonical form, if the set $\{Y_1, ..., Y_m\}$ can be decomposed into the union: left_set(r_1)∪...∪left_set(r_n), the decomposition is unique. If the set can not be decomposed, the rule-set is not transformed into a definite clause (such a rule-set contains *bugs*!). Let p, q_1, ..., q_n be the predicate symbols corresponding to the tuples left_tuple(r)=(X_1, ..., X_i), left_tuple(r_1)=(Y_1, ..., Y_{g1}), left_tuple(r_2)=(Y_{g1+1}, ..., Y_{g2}), ..., left_tuple(r_n)=($Y_{g(n-1)+1}$, ..., Y_m) ($1 \le g1 < g2 < ... < m$, $n \ge k \ge 1$), respectively. Transform the rule-set r into the following definite Horn clause,

$$\mathfrak{R}^{-1}(r) = p(*X_{1,h}, ..., *X_{i,h}) :- q_1(*Y_1, ..., *Y_{g1}), ..., q_n(*Y_{g(n-1)+1}, ..., *Y_m), *X_{1,h} \approx t_1', ..., *X_{i,h} \approx t_i' \; u_1', ..., u_j'.$$

where $*X_{1,h}$, ..., $*X_{i,h}$, $*Y_1$, ..., $*Y_m$ mean the variables corresponding to the nonterminal symbols X_1, ..., X_i, Y_1, ..., Y_m, and the terms t_1', ..., t_i', u_1', ..., u_j' contain $*Y_1$, ..., $*Y_m$ in place of Y_1, ..., Y_m.

Without the loss of generality we can assume that every left-hand-omitted-rule u_k has at least one metasymbol "≈", because we may replace it by $u_k \approx *Z$ if u_k has no metasymbol, where *Z is a new variable which does not originally appear in the rule-set r.

The metasymbol "≈" in the rule-set is re-interpreted as the **equality symbol** ≈ in the definite clause. The value of the ground atom $t \approx u$ is true if t and u are the same ground term, and false otherwise.

The transformed logic program $\mathfrak{R}^{-1}$(G) is defined as the quintuple (P, F, V∪V', s, D}, where P is a finite set of predicate symbols corresponding to all left_tuple(r), V' is the set of variables corresponding to all the nonterminal symbols, the predicate symbol: s corresponds to Ω, and D is the set of the transformed definite clauses.

Example 16 (continued) The t-CCFG program **fib.G** is of the canonical form. Here we know that

left_tuple((r.1)) = left_tuple((r.2)) = left_tuple((r.3) = (Fin, Fout),

left_tuple((r.4)) = (Fin', Fout'),

left_tuple((r.5)) = left_tuple((r.6)) = ($Plus_1$, $Plus_2$, $Plus_3$),

left_tuple((r.7)) = left_tuple((r.8)) = (Num).

Supposing that (Fin, Fout) is corresponding to the predicate symbol fib, (Fin', Fout') to fib',

($Plus_1$, $Plus_2$, $Plus_3$) to plus, and (Num) to num, respectively, each rule-set is transformed as follows,

$fib(*Fin_h, *Fout_h)$:-$*Fin_h \approx 0$, $*Fout_h \approx s(0)$ ...$\mathcal{R}^{-1}$(r.1)

$fib(*Fin_h, *Fout_h)$:-$*Fin_h \approx s(0)$, $*Fout_h \approx s(0)$ ...$\mathcal{R}^{-1}$(r.2)

$fib(*Fin_h, *Fout_h)$:-$fib(*Fin, *Fout)$, $fib'(*Fin', *Fout')$, $plus(*Plus_1, *Plus_2, *Plus_3)$,
$*Fin_h \approx s(*Fin) \approx s(s(*Fin'))$, $*Fout_h \approx *Plus_3$, $*Fout \approx *Plus_1$,
$*Fout' \approx *Plus_2$, ...$\mathcal{R}^{-1}$(r.3)

$fib'(*Fin'_h, *Fout'_h)$:-$fib(*Fin, *Fout)$, $*Fin'_h \approx *Fin$, $*Fout'_h \approx *Fout$, ...$\mathcal{R}^{-1}$(r.4)

$plus(*Plus_{1h}, *Plus_{2h}, *Plus_{3h})$:-$num(*Num)$, $*Plus_{1h} \approx 0$, $*Plus_{2h} \approx *Num$,
$*Plus_{3h} \approx *Num$, ...$\mathcal{R}^{-1}$(r.5)

$plus(*Plus_{1h}, *Plus_{2h}, *Plus_{3h})$:-$plus(*Plus_1, *Plus_2, *Plus_3)$, $*Plus_{1h} \approx s(*Plus_1)$,
$*Plus_{2h} \approx *Plus_2$, $*Plus_{3h} \approx s(*Plus_3)$, ...$\mathcal{R}^{-1}$(r.6)

$num(*Num_h)$:-$*Num_h \approx 0$, ...$\mathcal{R}^{-1}$(r.7)

$num(*Num_h)$:-$num(*Num)$, $*Num_h \approx s(*Num)$. ...$\mathcal{R}^{-1}$(r.8)

By eliminating redundancies, the above definite clauses can be further transformed into the same clauses as $\mathcal{P}$(**fib.L**) not the exact **fib.L**.

7.2.Transformation rule for the programs of non-canonical form

The transformation rule for t-CCFG programs of the non-canonical form is obtained by revising the transformation rule for programs of the canonical form.

In a program of the canonical form, the set of all the nonterminal symbols which appear in the right-hand sides of a rule-set can be *uniquely* decomposed into the union of sets just like $left\_set(r_1) \cup ... \cup left\_set(r_n)$. In a program of the non-canonical form, on the other hand, it may be decomposed in *several* ways. The idea of the revision is that if there are several decompositions, a rule-set is transformed into several definite Horn clauses corresponding to the decompositions.

Transformation rule $\mathcal{R}^{-1}$(revised version) Let r be the same rule-set as in the definition of the previous transformation rule. We suppose that the union: $left\_set(r_1) \cup ... \cup left\_set(r_n)$ is one of the decompositions of the set $\{Y_1, ..., Y_m\}$. Then the rule-set r is transformed into the same definite Horn clause $\mathcal{R}^{-1}(r)$ as in the previous rule. This transformation is applied for all the

possible decompositions.

Example 17 (continued) The t-CCFG program **ncf.G** described in the previous subsection is not of the canonical form. Let the tuples (A, B, C), (A, B) and (C) correspond to the predicate symbols: p, q and r, respectively. Then each rule-set is transformed as follows,

$q(*A_h, *B_h)$:-$*A_h \approx 0$, $*B_h \approx 1$, ...$\mathfrak{R}^{-1}$(n.1)

$r(*C_h)$:-$*C_h \approx 2$, ...$\mathfrak{R}^{-1}$(n.2)

$p(*A_h, *B_h, *C_h)$:- $*A_h \approx 0$, $*B_h \approx 0$, $*C_h \approx 0$, ...$\mathfrak{R}^{-1}$(n.3)

$q(*A_h, *B_h)$:-$q(*A, *B)$, $*A_h \approx d(*A)$, $*B_h \approx e(*B)$, ...$\mathfrak{R}^{-1}$(n.4)

$p(*A_h, *B_h, *C_h)$:- $p(*A, *B, *C)$, $*A_h \approx a(*A)$, $*B_h \approx b(*B)$, $*C_h \approx c(*C)$, ...$\mathfrak{R}^{-1}$(n.5)

$p(*A_h, *B_h, *C_h)$:- $q(*A, *B)$, $r(*C)$, $*A_h \approx a(*A)$, $*B_h \approx b(*B)$, $*C_h \approx c(*C)$ ...$\mathfrak{R}^{-1}$(n.5)

Note that (n.5) is transformed into two kind of clauses. These are simplified as

q(0,1),

r(2),

p(0,0,0),

q(d(*A), e(*B)):-q(*A, *B),

p(a(*A), b(*B), c(*C)):-p(*A, *B, *C),

p(a(*A), b(*B), c(*C)):-q(*A, *B), r(*C).

The correctness of the transformation rule $\mathfrak{R}^{-1}$ can be proved in the same way as the rule $\mathfrak{R}$. One difficulty is that the equality symbol which appears in a transformed definite clause acts like a meta-functional symbol. Then in order to preserve the precise equivalence between $T_G{}^k(\emptyset)$ and $T_{\mathfrak{R}^{-1}(G)}{}^k(\emptyset)$ for any $k \geq 0$, the mapping T_L for a logic program L is revised as follows.

Definition 22 (revised version) Given a logic program L=(P, F, V, s, D), the mapping T'_L of $P(B_L)$ is defined as follows,

$$T'_L(A) = \{\ p(...)\phi \mid\ p(...):\text{-}q_1(...), ..., q_n(...), v_1 \approx ... \approx w_1, ..., v_k \approx ... \approx w_k \in D,$$
$$\exists \phi \in \Phi_L,$$
$$p(...)\phi \in B_L,$$
$$q_i(...)\phi \in A, \text{ for all } n \geq i \geq 1,$$
$$(v_j \approx ... \approx w_j)\phi = x_j \approx ... \approx x_j,\ x_j \in U(F) \text{ for all } k \geq j \geq 1\},$$

where $A \in P(B_L)$.

Theorem 3 For every t-CCFG program G, the transformation rule $\mathfrak{R}^{-1}$ preserves the semantics. Namely it holds that $SF_F[G] = SF_F[\mathfrak{R}^{-1}(G)]$.

In other words, every t-CCFG program G can be transformed into the equivalent logic program $L=\mathfrak{R}^{-1}(G)$. The relationship between the t-CCFG program G and the logic program L is similar to the relationship between the logic program L and the t-CCFG program $\mathfrak{R}(L)$ described in subsection 6.2, because G and $\mathfrak{R}(L)$ $(=\mathfrak{R}(\mathfrak{R}^{-1}(G)))$ have the same program structure.

8.Discussions

First, we have defined term-based CCFG programming system. Strings the original CCFG programming system can treat are special terms which satisfy the associative law. Term-based one has been defined in the same way as in [1] without the associative law.

Second, we have shown the program transformation rule from an arbitrary logic program (t-CCFG program) into the equivalent t-CCFG program (logic program). Each definite clause in the logic program (rule-set in the t-CCFG program) is transformed into the corresponding rule-set (definite clause). These transformation rules are so simple that it is easy to construct an automatic program transformation system. Through these transformation rules, the close relationship between the syntactic structures of both programs has been shown. It can be said that a logic program and the corresponding t-CCFG program are in the *dual* relation. Since logic programs and t-CCFG programs can be mutually transformed, one can solve problems in both programming systems. However, one should solve the problems which can be naturally expressed as logical deductions in logic programming system whereas one should solve the problems in which the data structures of objects can be naturally expressed as (term-based) context-free grammars in CCFG programming system.

This transformation rule has some effects on our studies of CCFG programming which we have begun for the purpose of establishing a data-structure directed programming system.

First one is to translate the fruitful results that many researchers have obtained in the area of logic programming into the corresponding ones in the area of CCFG programming. For example, some unified theories of logic and functional programmings have been already known. From these theories we will be able to deduce the unified theories of CCFG and functional programmings straightforwardly. For one more example, we hardly know the theory of negation in the area of formal grammars, while it has been well studied in the area of logic and logic programming. By translating the latter into the area of CCFG programming, the negation may also be established in the area of extended CCFG programming.

Second one is to establish the unified theory of CCFG and logic programs. The syntax and the (least fixpoint) semantics of the mixed programs containing definite Horn clauses and rule-sets can be well defined because the semantic domains of CCFG programs and logic programs are similar and they can be easily extended for the mixed programs.

The authors are now studying these subjects besides the design and implementation of a

programming language based on CCFG.

Acknowledgments

The authors would like to thank Professor M. Sassa for his useful and helpful advice.

References

[1] Yamashita, Y. and Nakata, I. :Programming in Coupled Context-Free Grammars, to be submitted.

[2] Lloyd, J. W. :*Foundation of Logic Programming*, Springer-Verlag (1984).

[3] Clocksin, W. F. and Mellish, C. S. :*Programming in Prolog*, Springer-Verlag (1981).

INSTITUTE OF INFORMATION SCIENCES AND ELECTRONICS
UNIVERSITY OF TSUKUBA
SAKURA-MURA, NIIHARI-GUN, IBARAKI 305 JAPAN

REPORT DOCUMENTATION PAGE

REPORT NUMBER
ISE-TR-88-71

TITLE

On the relation between CCFG programs and logic programs

AUTHOR(S)

Yoshiyuki YAMASHITA and Ikuo NAKATA

REPORT DATE
Jun. 1, 1988

NUMBER OF PAGES
33

MAIN CATEGORY
Programming Languages

CR CATEGORIES
D.3, F.3, F.4

KEY WORDS
Logic Program, Context-Free Grammar, Coupled Context-Free Grammar, Program Transformation

ABSTRACT

CCFG programming is a programming system based on the formal grammar: Coupled Context-Free Grammar (CCFG) . A CCFG is regarded as a program, called a CCFG program, and its semantics is defined by the language, a set of tuples of derived terminal strings which it generates. CCFG programs are similar to logic programs, because their semantics domains are similar in the sense that both represent relations between input/output data objects. In this paper the relationship between CCFG programs and logic programs is discussed by using program transformations. The rules for these transformations are so simple and we can clearly understand the characteristics of CCFG programs by comparing them with those of corresponding logic programs.

SUPPLEMENTARY NOTES