ISE-TR-88-72

Unfold/Fold Program Transformation in CCFG Programming

by

Yoshiyuki YAMASHITA and Ikuo NAKATA

June 1, 1988

Jan.16.1987

Unfold/Fold Program Transformation in CCFG programming

Yoshiyuki YAMASHITA and Ikuo NAKATA,

Institute of Information Sciences and Electronics,
University of Tsukuba.

Abstract

The program transformation rules, unfolding, folding and replacement, in CCFG programming are proposed. Because of the resembance between CCFG programming and logic programming, we can defined the rules in the same way as in logic programming. The correctness is also discussed.

1.Introduction

Unfolding and folding are central techniques to transform programs into effective ones. They are first proposed in the areas of functional programming by Burstall and Darlington [5], and also in the area of logic programming by some people [6],[7]. The correctness of the transformations are studied by Tamaki and Sato [8] and so on [9]. Such transformation techniques as unfolding, folding and replacement are known in most of program manipulation systems. We think that if one system stands on a strict mathematical foundation, such program transformation techniques can be well defined in the system as well as in functional and logic programmings.

CCFG programming is a grammatical programming system based on the formal grammar: Coupled Context Free Grammar (CCFG) [1],[2],[3],[4]. In CCFG programming, a CCFG is interpreted as a program and its execution is a derivation of tuples of terminal strings by the program. Since the semantics is denotated by a set of n-tupels of terminal strings, and the terminal strings can be considered as input/output objects, CCFG programs represent the relations of the input/output objects. Because of representing the relations, CCFG programming has similar feature to logic programming and there exists the transformation from an arbitrary logic program into the corresponding CCFG program preserving the equivalence of the meanings.

In this paper we propose the unfold, fold and replacement rules in CCFG programming. Because the mathematical structures of CCFG programming and logic programming are equivalent, as described above, we can construct the transformation rules in the same way as in logic programming. In this paper we mainly refer the studies by Tamaki and Sato[8]. They have proved the correctness of the rules and we expect to prove our transformation rules in CCFG programming in the same way.

The authors have proposed some ideas of the global strategies of CCFG program transformation in [3]. Context free grammars embedded in a CCFG program are interpreted as the representations of input/output data structures of the program, and such interpretation may give us new transformation strategies which are oriented to input/output data structures defined by context free grammars. The transformation rules presented in this paper become the mathematical basis of such strategies.

In section two, CCFG programming is briefly reviewed. Readers not familiar with CCFG programming are expected to read [1], [2] and [4].

In section three, some simple transformation rules are defined, which deal with only one rule-set at a time and transform it into another. These transformation rules play the auxiliary roles in the sequences of transformations. Their correctness is obvious.

In section four, unfolding, folding and application of replacement rules are defined in the same way as [8].

In section five, three examples of transformation are presented. The first one is to transform the program described in section two which defines an indirect relation between input and output objects using a intermediate file, into the program which defines a direct relation between them. The second one is to transform the program to calculate a squared integer n^2 (n is an integer) using the multiplier subprogram, into a more effective one . The last one is to transform the program to calculate a Fibonacci number, into a more effective one.

In section six, the correctness of the transformation rules are briefly discussed. The abstract line of the proof is given.

Glossary Symbols.

Nonterminal symbols are denoted by upper case letters A,B,...,X,Y,...

Terminal symbols are denoted by lower case letters a,b,...

Strings composed of nonterminal symbols, terminal symbols and a meta symbol "≈" are called *strings* and denoted by lower case greek letters $\alpha,\beta,...$

A n-tuple of strings is denoted by upper case greek letter Ω.

Terminal strings are denoted by italic lower case letters $x,y,...$

All the concatenation operators for strings and tuples of strings are denoted by a colon ":".

2.CCFG Programming

Here we briefly review CCFG programming in order to prepare for the discussions in the following sections.

In CCFG programming, a program is a quadruple to define a CCFG. For example, the following quadruple is a program to represent strings composed of symbols "a" and "b", and their reverse,

$$G_1=(N,T,\Omega_S,R),$$

$N=\{X,Y\},$
$T=\{a,b\},$
$\Omega_S=(X,Y),$
$R=\{\quad \{X\rightarrow\varepsilon,\ Y\rightarrow\varepsilon\},$
$\quad\quad \{X\rightarrow aX,\ Y\rightarrow Ya\},$
$\quad\quad \{X\rightarrow bX,\ Y\rightarrow Yb\}\},$

where N is the finite set of nonterminal symbols, T the finite set of terminal symbols, Ω_S the n-tuple of start symbols, and R the finite set of **rule-sets**. A rule-set is a set of rules, and it can be applicable to a tuple of sentential forms derived from Ω_S if the tuple contains all the nonterminal symbols which appear in the left-hand sides of rules in the rule-set. A derivation is a sequence of applications of rule-sets. One example of the derivations by the program is given bellow,

$$\begin{aligned}
\Omega_S=(X,Y) &\Rightarrow (aX,Ya) && \text{by } \{X\rightarrow aX,\ Y\rightarrow Ya\}\\
&\Rightarrow (abX,Yba) && \text{by } \{X\rightarrow bX,\ Y\rightarrow Yb\}\\
&\Rightarrow (abbX,Ybba) && \text{by } \{X\rightarrow bX,\ Y\rightarrow Yb\}\\
&\Rightarrow (abb,bba) && \text{by } \{X\rightarrow\varepsilon,\ Y\rightarrow\varepsilon\}.
\end{aligned}$$

The above derivation generates the couple (abb,bba) of terminal strings. The set of couples of terminal strings derived as above is the language generated by the program and it is also the meaning of the program. In this case, the language $L(G_1)$ is a set of the couples of terminal strings reversed each other, that is

$$L(G_1)=\{(\varepsilon,\varepsilon),(a,a),(b,b),(ab,ba),(ba,ab),(abb,bba),..\}.$$

Therefore we say that program G_1 reverses strings.

Context free grammars are often interpreted as the representations of data structures. If we consider the context free grammars embedded in a program, they can be also interpreted as the representations of the input/output data structures in the program. In the case of the above example, there are two context free grammars,

$$\{X\rightarrow\varepsilon,\ X\rightarrow aX,\ X\rightarrow bX\},$$

and

$$\{Y\rightarrow\varepsilon,\ Y\rightarrow Ya,\ Y\rightarrow Yb\},$$

embedded in program G_1. This interpretation is important to understand CCFG programs.

The above example is similar to a scheme of syntax directed translations [10]. More expressive powers of CCFG programming than that of syntax directed translations are given by the idea of **multiple data structures** by using a new **meta symbol** "≈". A multiple data structure is just like an intersection of context free languages. For example, in the following CCFG program G_2, the nonterminal symbol Q defines a multiple data structure,

$$
\begin{aligned}
&G_2=(\{Q,Y,Z\},\{a,b\},(Q),R),\\
&\quad R=\{\quad \{Q\rightarrow Y\approx Z\}\\
&\qquad\qquad \{Y\rightarrow\varepsilon\}, \qquad \{Z\rightarrow\varepsilon\},\\
&\qquad\qquad \{Y\rightarrow aYb\}, \qquad \{Z\rightarrow abZ\}\}.
\end{aligned}
$$

The derivation rule for the meta symbol "≈" is given that if a derived terminal string contai "≈" (for example $x{\approx}y$), and both sides are same (that is $x=y$), then we say the derivation is **successful** and generates a factor of the language. If a derived terminal string contains "≈" and both sides are different from each other (that is $x{\neq}y$), then we say that the derivation is **failed**. Both examples of successful and failed derivations by program G_2 are given as follows.

$$Q \Rightarrow Y{\approx}Z \Rightarrow aYb{\approx}Z \Rightarrow ab{\approx}Z \Rightarrow ab{\approx}abZ \Rightarrow ab{\approx}ab.$$

This is successful, because "ab"="ab".

$$Q\Rightarrow Y{\approx}Z \Rightarrow aYb{\approx}Z \Rightarrow ab{\approx}Z \Rightarrow ab{\approx}abZ \Rightarrow ab{\approx}ababZ \Rightarrow ab{\approx}abab.$$

This is failed, because "ab"≠"abab".

In this way, it is easily understood that

$$L(G_2)=\{x \mid Q\Rightarrow^* x{\approx}x\}=\{\varepsilon,ab\},$$

where $\Rightarrow^*$ means the reflective and transitive closure of $\Rightarrow$. In this case, $L(G_2)$ is equal to the set $\{x \mid Y\Rightarrow^* x\}\cap\{x \mid Z\Rightarrow^* x\}$.

The meta symbol "≈" also plays an important role to represent an indirect relation between objects, as seen in the following program G_3,

$$
\begin{aligned}
&G_3=(\{P,Q,R,X,Y,Z,W\},\{a,b,c,d,e\},(P,Q,R),R),\\
&\quad R=\{\quad \{P\rightarrow X, \qquad Q\rightarrow Y{\approx}Z, \qquad R\rightarrow W\},\\
&\qquad\qquad \{X\rightarrow\varepsilon,\quad Y\rightarrow\varepsilon\}, \qquad \{Z\rightarrow\varepsilon,\quad W\rightarrow\varepsilon\},\\
&\qquad\qquad \{X\rightarrow aX, Y\rightarrow cY\}, \qquad \{Z\rightarrow cdZ, W\rightarrow eW\},\\
&\qquad\qquad \{X\rightarrow bX, Y\rightarrow dY\}\}.
\end{aligned}
$$

Under the derivation rule for "≈", the language generated by program G_3 is obtained as follows,

$$
\begin{aligned}
L(G_3)&=\{(x,y,z) \mid (P,Q,R)\Rightarrow^*(x,y{\approx}y,z)\}\\
&=\{(\varepsilon,\varepsilon,\varepsilon),(ab,cd,e),(abab,cdcd,ee),\ldots\}\\
&=\{((ab)^n,(cd)^n,e^n) \mid n\geq 0\}.
\end{aligned}
$$

Program G_3 contains two subprograms. One subprogram is specified by the rule-sets

$\{X \to \varepsilon, Y \to \varepsilon\}$, $\{X \to aX, Y \to cY\}$ and $\{X \to bX, Y \to dY\}$, which means that the couple (X,Y) of the nonterminal symbols derives couples (x,y), where x is a terminal string composed of the terminal symbols "a" and "b" and y is the string obtained by changing "a" and "b" in x to "c" and "d" respectively. Another subprogram is specified by the rule-sets $\{Z \to \varepsilon, W \to \varepsilon\}$ and $\{Z \to cdZ, W \to eW\}$, which means that (Z,W) derives a couple $((cd)^n, e^n)$ for any $n \geq 0$. The connection of these two subprograms by "Y≈Z" restricts the derivations and makes a new indirect relation between the terminal strings derived by the start symbols P and R.

If we pay attention only to the indirect relation between P and R and have no interest in Q, then the left-hand side of the rule $Q \to Y \approx Z$ may be omitted and the right-hand side $Y \approx Z$ may be used as a rule. Such a rule is called a **left-hand-omitted-rule**. Notice that a left-hand-omitted-rule may not have a meta symbol "≈" in general. Then we can revise program G as follows,

$$
\begin{array}{lllll}
G_4=(\{P,R,X,Y,Z,W\},\{a,b,c,d,e\},(P,R),R), & & & & \\
\quad R=\{ & \{P \to X, & Y \approx Z, & & R \to W\}, \\
 & \{X \to \varepsilon, \; Y \to \varepsilon\}, & & \{Z \to \varepsilon, & W \to \varepsilon\}, \\
 & \{X \to aX, \; Y \to cY\}, & & \{Z \to cdZ, & W \to eW\}, \\
 & \{X \to bX, \; Y \to dY\}\}. & & &
\end{array}
$$

As described above, CCFG programs define the relation of input/output objects. In this sense, CCFG programming is similar to logic programming, and we can show that both programming systems stand on the same mathematical background, by showing the program transformation from an arbitrary logic program into the corresponding CCFG program preserving the equivalence (see section six and [2]). In this paper we can use this transformation in order to prove the correctness of program transformation rules (unfolding, folding and replacement) presented in section four.

CCFG programming has been briefly described. More discussions are presented in [1],[2],[3],[4].

3.Simple Program Transformation Rules

Although unfolding, folding and replacement are the central transformation rules, there are some other primitive transformation rules, which play the auxiliary roles in the sequences of transformations.

For example, the rule-set $\{X \to Y, a \approx a\}$ in a CCFG program can be said to be equivalent to the rule-sets $\{X \to Y\}$ because if there exists a successful derivation using the former rule-set, it is clear that there exists a successful derivation using the latter rule-set and if there exists a successful derivation using latter, so the former. For another example, the rule-set $\{X \to Y, a \approx b\}$ can be eliminated from the program because the derivation in which the rule-set is applied at least once, contains the sentential form "a≈b" in itself and such a derivation is not successful according to th derivation rule for "≈" by any means.

In this way, some transformation rules are found as follows. Their correctness is evident. we call them **simple rules.**

Definition(substring). Let α be an arbitrary string composed of terminal symbols, nonterminal symbols and the meta symbol "≈". The substrings of α are defined as follows,

(1) if α contains no meta symbol "≈" and $\alpha=\beta\gamma$, then β and γ are substrings of α.

(2) if $\alpha=\beta\alpha_1\gamma\approx\ldots\approx\beta\alpha_n\gamma$, then $\alpha_1\approx\ldots\approx\alpha_n$ is a substring of α.

(3) if $\alpha=\alpha_1\approx\ldots\approx\alpha_n$, then $\alpha_{f1}\approx\ldots\approx\alpha_{fi}$ $(n\geq fj\geq 1,\ i\geq j\geq 1)$ is a substring of α.

(4) if $\alpha=(\beta)$, then β is a substring of α, where "(" and ")" is the meta symbols which express a parenthesis and satisfy that for arbitrary strings α, β, γ and δ

$$\alpha(\beta\approx\gamma)\delta=\alpha\beta\delta\approx\alpha\gamma\delta.$$

(5) if β is a substring of α and γ is a substring of β, then γ is a substring of α.

The set of all the substrings of α is denoted by $\mathrm{Sub}(\alpha)$, and the set of all the substrings of $\alpha_1,\ldots,\alpha_n$ is denoted by $\mathrm{Sub}(\{\alpha_1,\ldots,\alpha_n\})$.

Simple rule-1. If a rule-set has a rule which has a substring $\alpha\approx\alpha$ in its right-hand side, the substring can be replaced by α.

Example The rule-set $\{X\rightarrow aZ,\ Y\rightarrow b\approx\underline{aZ\approx aZ},\ \underline{c\approx c}\approx\underline{T\approx T}\}$ can be transformed into the rule-set $\{X\rightarrow aZ, Y\rightarrow b\approx aZ,\ c\approx T\}$. Here the underlines are used only for highlighting.

Simple rule-2. If a rule-set has a left-hand-omitted-rule α and also has at least one other rules which has a substring α in its right-hand side, the left-hand-omitted-rule can be eliminated.

Example. The rule-set $\{X\rightarrow aZ,\ Y\rightarrow \underline{b\approx aZ},\ c\approx T,\ \underline{b\approx aZ}\}$ can be transformed into the rule-set $\{X\rightarrow aZ, Y\rightarrow b\approx aZ,\ c\approx T\}$ by eliminating the underlined left-hand-omitted-rule.

Simple rule-3. If a rule-set has a rule which has a substring α in its right-hand side, a left-hand-omitted-rule α can be added to the rule-set. Notice that the substrings α may or may not contain the meta symbol "≈".

This is the reverse rule of the simple rule-2.

Example. The rule-set $\{X\rightarrow aZ, Y\rightarrow \underline{b\approx aZ}, c\approx T\}$ can be transformed into the rule-set $\{X\rightarrow aZ, Y\rightarrow b\approx aZ, c\approx T, b\approx aZ\}$ by adding the underlined left-hand-omitted-rule. It can be also transformed into the rule-set $\{X\rightarrow aZ, Y\rightarrow b\approx aZ, c\approx T, \underline{aZ}\}$ because the underlined left-hand-omitted-rule appears in the original rule-set.

Simple rule-4. If a rule-set has a substring $\alpha \approx \beta$ and also has at least one other substring α in itself, the latter substring can be replaced by β or $\alpha \approx \beta$.

Example. The rule-set $\{X \rightarrow \underline{aZ},\ Y \rightarrow \underline{b \approx aZ},\ c \approx T\}$ can be transformed into the rule-set $\{X \rightarrow b,\ Y \rightarrow b \approx aZ,\ c \approx T\}$ or the rule-set $\{X \rightarrow b \approx aZ, Y \rightarrow b \approx aZ,\ c \approx T\}$ by replacing the underlined substring for the substring "b" or "b≈aZ", respectively.

Simple rule-5. If a rule-set has a left-hand-omitted-rule $x\alpha_1 y \approx \ldots \approx x\alpha_n y$, where x and y ($\in T^*$) are common prefix and postfix of all the side of the rule, respectively, the left-hand-omitted-rule can be replaced by $\alpha_1 \approx \ldots \approx \alpha_n$.

Example. The rule-set $\{X \rightarrow aZ,\ Y \rightarrow b \approx aZ,\ \underline{acd \approx aTd}\}$ can be transformed into the rule-set $\{X \rightarrow b,\ Y \rightarrow b \approx aZ,\ c \approx T\}$ by eliminating the common prefix "a" and postfix "d" from the underline rule.

Simple rule-6. If a rule-set has a left-hand-omitted-rule $\alpha_1 \approx \ldots \approx \alpha_n$, the rule can be replaced by $x\alpha_1 y \approx \ldots \approx x\alpha_n y$ for arbitrary terminal strings x and y.

This is the reverse rule of the simple rule-5.

Example. The rule-set $\{X \rightarrow aZ,\ Y \rightarrow b \approx aZ,\ \underline{c \approx T}\}$ can be transformed into the rule-set $\{X \rightarrow b,\ Y \rightarrow b \approx aZ,\ acd \approx aTd\}$ by appending the prefix "a" and postfix "d" to the underlined rule.

Simple rule-7. If a rule-set has a left-hand-omitted-rule $x \approx \ldots \approx x$, where x is a terminal string, it can be eliminated from the rule-set.

Example. The rule-set $\{X \rightarrow aZ, Y \rightarrow b \approx aZ,\ \underline{cd \approx cd}\}$ can be transformed into the rule-set $\{X \rightarrow b, Y \rightarrow b \approx aZ\}$ by eliminating the underlined rule.

Simple rule-8. If a rule-set in a program has the right-hand side $\alpha_1 \approx \ldots \approx \alpha_n$ of a rule, and some α_i and α_j ($n \geq i,j \geq 1$) are **not unifiable**, the rule-set can be eliminated from the program.

Non-unifiability between strings α_i and α_j holds if $\alpha_i\theta \neq \alpha_j\theta$ for any substitutions θ of strings into nonterminal symbols which appear in α_i and α_j.

The reason why the simple rule-8 can be stated is because when the rule-set has been applied, the derived tuple of sentential forms always contains the substring $\alpha_i \approx \alpha_j$ and such a derivation is not successful by any means under the derivation rule for the meta symbol "≈". Since the meaning of a CCFG program depends on only successful derivations, the rule-set which always generates failed derivations may be eliminated from the program and the meaning of the transformed program is equivalent to that of the original one.

As described in the following sections, the simple rule-8 is used in order to eliminate the useless rule-sets which are derived after an unfolding operation.

Example. If there exists a rule-set {X→aZ, Y→b≈aZ, <u>c≈d</u>}, this rule-set can be eliminated from the program by simple rule-8 because the rule-set has the non-unifiable substring "c≈d".

Definition(anonymous nonterminal symbol). The symbol "*" is called an **anonymous nonterminal symbol** to derive any terminal strings, that is

$$\{x \mid * \Rightarrow^{*} x\} = T^{*}.$$

Anonymous nonterminal symbols always appear in the CCFG program transformed from a logic program. They correspond to variables in the logic program.

Simple rule-9. If a rule-set in a program has a substring X≈β and the program contains the rule-set {X→*}, all the nonterminal symbols X which appear in the right-hand sides of the rules in the rule-set can be replaced by β.

Example If there exist rule-sets {X→<u>A</u>Z, Y→b≈<u>A</u>Z, c≈T, <u>A≈a</u>}, and <u>{A→*}</u>, considering the nonterminal symbol A, the rule-sets can be transformed into the rule-set {X→aZ, Y→b≈aZ, c≈T, a≈a}. And further it can be transformed into the rule-set {X→aZ, Y→b≈aZ, c≈T} by the simple rule-7.

Simple rule-10. If a rule-set in a program has a substring α, the substring can be replaced by a new nonterminal symbol X which does not appear in the original program provided, the left-hand-omitted-rule X≈α is added to the rule-set and the rule-set {X→*} is added to the program.

Example If there exists a rule-set {X→aZ, Y→<u>b</u>≈aZ, c≈T} in a program, it can be transformed into {X→aZ, Y→<u>A</u>≈aZ, c≈T, b≈<u>A</u>} and the rule-set {A→*} can be added to the program.

The above ten simple rules are applied in the way between the tranformation sequences of unfolding, folding and replacement presented in the next section and play the auxiliary roles in the sequences.

4.Program Transformation Rules

Here we propose CCFG program transformation rules so called unfolding, folding and replacement. These rules have been well known in functional and logic programming. Especially we refer to the corresponding rules in logic programming studied by Tamaki and Sato[5] in order to construct those in CCFG programming. The correctness of the application of the rules is discussed in section six.

Definition(set of nonterminal symbols). For a string α, $N(\alpha)$ denotes the set of all the nonterminal symbols which appear in α.

For a set $S=\{\alpha_1,\dots,\alpha_i\}$ of strings, $N(S)$ denotes the set of all the nonterminal symbols in the strings in S, $N(\alpha_1)\cup\dots\cup N(\alpha_i)$.

For a rule-set $R=\{X_1\rightarrow\alpha_1,\dots,X_i\rightarrow\alpha_i,\beta_1,\dots,\beta_j\}$, $N_{left}(R)$ denotes the set of all the nonterminal symbols in the left-hand sides of the rule in R, $\{X_1,\dots,X_i\}$, and $N_{right}(R)$ denotes the set of all the nonterminal symbols which appear in the right-hand sides of the rules in R, $N(\alpha_1)\cup\dots\cup N(\alpha_i)\cup N(\beta_1)\cup\dots\cup N(\beta_j)$.

Definition(rule-set application). Let R_1 and $R_2=\{X_1\rightarrow\alpha_1,\dots,X_i\rightarrow\alpha_i,\beta_1,\dots,\beta_j\}$ $(i\geq 1, j\geq 0)$ be two rule-sets. If $N_{right}(R_1)$ contains $N_{left}(R_2)$, that is

$$N_{right}(R_1)\supseteq N_{left}(R_2),$$

then the rule-set

$$R=(R_1\{X_1/\alpha_1,\dots,X_i/\alpha_i\})\cup\{\beta_1,\dots,\beta_j\},$$

is said to be the result of applying R_2 to R_1, where $R_1\{X_1/\alpha_1,\dots,X_i/\alpha_i\}$ means the simultaneous substitutions of α_k $(i\geq k\geq 1)$ into X_k in the right-hand sides of R_1.

If the names of the nonterminal symbols which originally appear in R_1 and are not replaced by the substitution conflict with the names of the nonterminal symbols which appear in the right-hand sides of the rules in R_2, that is

$$(N_{right}(R_1)-N_{left}(R_2))\cap N_{right}(R_2)\neq\phi,$$

then we suffix numbers to the names of the substituted nonterminal symbols so that such nonterminal symbols may be identified.

Example. The result of applying the rule-set $\{Z\rightarrow\varepsilon, W\rightarrow\varepsilon\}$ to the rule-set $\{P\rightarrow X, Y\approx Z, R\rightarrow W\}$ is $\{P\rightarrow X, Y\approx\varepsilon, R\rightarrow\varepsilon\}$. The result of applying the rule-set $\{Z\rightarrow X, W\rightarrow Y\}$ to the rule-set $\{P\rightarrow X, Y\approx Z, R\rightarrow W\}$ is $\{P\rightarrow X, Y\approx X_2, R\rightarrow Y_2\}$. In the second case, because the name conflicts appear, the suffixed names X_2 and Y_2 of the nonterminal symbols are used for X and Y.

Definition(unfolding). Let G be a CCFG program, R be a rule-set in G, M be a set of nonterminal symbols, and $R_1,\dots,R_n$ in G be all the rule-sets which are applicable to R and satisfy that

$$N_{left}(R_1)= \ldots = N_{left}(R_n)=M.$$

Let R'_i ($n \geq i \geq 1$) be the result of applying R_i to R. We call the set $\{R'_1,\ldots,R'_n\}$ the result of unfolding R by $R_1,\ldots, R_n$.

Example. In program G_2, the result of unfolding the rule-set $\{P \rightarrow X, Y \approx Z, R \rightarrow W\}$ by the rule-sets $\{Z \rightarrow \varepsilon, W \rightarrow \varepsilon\}$ and $\{Z \rightarrow Zab, W \rightarrow Wc\}$ is the set $\{\{P \rightarrow X, Y \approx \varepsilon, R \rightarrow \varepsilon\}, \{P \rightarrow X, Y \approx Zab, R \rightarrow Wc\}\}$.

Definition(folding). Let

$$R_1=\{X_1 \rightarrow \alpha_1,\ldots,X_i \rightarrow \alpha_i, \beta_1,\ldots, \beta_j\}, (i \geq 1, j \geq 0),$$

and

$$R_2=\{Y_1 \rightarrow \gamma_1,\ldots,Y_k \rightarrow \gamma_k, \delta_1,\ldots, \delta_m\}, (k \geq 1, m \geq 0),$$

be two rule-sets in a CCFG program G. If the following three conditions hold,

$$Sub(\{\alpha_1,\ldots,\alpha_i\}) \cup Sub(\{\beta_1,\ldots,\beta_j\}-\{\delta_1,\ldots, \delta_m\}) \supseteq \{\gamma_1,\ldots,\gamma_k\},$$

$$\{\beta_1,\ldots,\beta_j\} \supseteq \{\delta_1,\ldots, \delta_m\},$$

and

$$N_{right}((R_1-\{\delta_1,\ldots,\delta_m\})\{\gamma_1/\varepsilon,\ldots,\gamma_k/\varepsilon\}) \cap N_{right}(R_2-\{\delta_1,\ldots,\delta_m\})=\phi,$$

then the following rule-set

$$R=(R_1-\{\delta_1,\ldots, \delta_m\})\{\gamma_1/Y_1,\ldots,\gamma_k/Y_k\},$$

is said to be the result of folding R_1 by R_2. The rule-set R_1 is called the folded rule-set, and R_2 the folding rule-set.

If no rule-set R'_2 in the program except R_2 saitsfies that

$$N_{left}(R'_2)= N_{left}(R_2),$$

the folding operation is said to be **reversible.**

Example. The result of folding the rule-set $\{P \rightarrow baX, Y \approx Z, R \rightarrow Wc\}$ by the rule-set $\{P \rightarrow X, Y \approx Z, R \rightarrow W\}$ is $\{P \rightarrow baP, R \rightarrow Rc\}$.

In the contrary, the rule-set $\{P \rightarrow X, Y \approx Z, R \rightarrow W\}$ is not foldable by the rule-set $\{P \rightarrow baX, Y \approx Z, R \rightarrow Wc\}$, because the first condition in the definition does not hold, that is,

$$baX \notin Sub(\{X,W\}) \cup Sub(\{Y,Z\}-\{Y,Z\}),$$

and

$$Wc \not\in Sub(\{X,W\}) \cup Sub(\{Y,Z\}-\{Y,Z\}).$$

The rule-set $\{P \to baX, Y \approx Z, R \to Wc, Q \to X\}$ is not also foldable by the rule-set $\{P \to aX, Y \approx Z, R \to W\}$, because the third condition does not hold, that is,

$$N_{right}((\{P \to baX, Y \approx Z, R \to Wc, Q \to X\} - \{Y \approx Z\})\{aX/\varepsilon, W/\varepsilon\}) = \{X\},$$

and

$$N_{right}(\{P \to aX, Y \approx Z, R \to W\} - \{Y \approx Z\}) = \{X,W\},$$

then

$$\{X\} \cap \{X,W\} \neq \phi.$$

The third condition in the above definition means that any nonterminal symbols which appear in $N_{right}(R_2 - \{\delta_1, ..., \delta_m\})$ must not remain in the result of folding. It matters little because even if the condition does not hold, the applications of the simple rule-10 and rule-6 can make such a non-foldable rule-set foldable as shown in the following example.

Example. In the previous example, the rule-set $\{P \to baX, Y \approx Z, R \to Wc, Q \to X\}$ is not foldable by the rule-set $\{P \to aX, Y \approx Z, R \to W\}$. Adding the left-hand-omitted-rule "$A \approx X$" to the former rule-set by the simple rule-10, and further adding the prefix "a" to the rule "$A \approx X$" by the simple rule-6, the former is transformed into $\{P \to baX, Y \approx Z, R \to Wc, Q \to A, aA \approx aX\}$, where we assume that $\{A \to *\}$ has been added to the progam,. This rule-set is foldable and the result of folding is $\{P \to bP, R \to Rc, Q \to A, aA \approx P\}$.

Definition(application of a replacement rule). A replacement rule is a simultaneous substitution $\theta = \{\alpha_1/\beta_1, ..., \alpha_n/\beta_n\}$ $(n \geq 1)$. For a rule-set

$$R = \{Y_1 \to \gamma_1, ..., Y_k \to \gamma_k, \delta_1, ..., \delta_m\}, (k \geq 1, m \geq 0),$$

and a replacement rule $\theta = \{\alpha_1/\beta_1, ..., \alpha_n/\beta_n\}$, if the following condition holds,

$$Sub(\gamma_1) \cup ... \cup Sub(\gamma_k) \cup Sub(\delta_1) \cup ... \cup Sub(\delta_m) \supseteq \{\alpha_1, ..., \alpha_n\},$$

then the rule-set $R\theta$ is said to be the result of applying the replacement rule to R.

Definition(correctness of a replacement rule). A replacement rule $\theta = \{\alpha_1/\beta_1, ..., \alpha_n/\beta_n\}$ is said to be correct with respect to a CCFG program $G = (N, T, \Omega_S, R)$ if the following two CCFG programs

$$G' = (N \cup \{S_1, ..., S_n\}, T, (S_1, ..., S_n), R \cup \{\{S_1 \to \alpha_1, ..., S_n \to \alpha_n\}\}),$$

and

$$G''=(N\cup\{S_1,...,S_n\},T,(S_1,...,S_n),R\cup\{\{S_1\to\beta_1,...,S_n\to\beta_n\}\}),$$

generate the same language, that is $L(G')=L(G'')$, where we suppose that $S_1,...,S_n$ does not appear in the original program.

Examples of replacement are presented in section five.

Definition(transformation sequence). Let Sr_0 be a set of rule-sets in a CCFG program $G=(N,T,\Omega_S,Sr_0)$ and Θ be a set of replacement rules. A (finite or infinite) sequence of sets of rule-sets $Sr_0,...,Sr_n,...$ is said to be a transformation sequence with the input (Sr_0,Θ), if for each Sr_n $(n\geq1)$ in the sequence one of the followings holds,

(1) *(application of simple rule)* $Sr_n=(Sr_{n-1}-\{R\})\cup\{R'\}$, where R is a rule-set in Sr_{n-1} and R' is the result of applying the simple rule-1,rule-2,..., or rule-10 in section three to R.

(2) *(unfolding).* $Sr_n=(Sr_{n-1}-\{R\})\cup\{R_1,...,R_k\}$, where R is a rule-set in Sr_{n-1}, and $\{R_1,...,R_k\}$ is the result of unfolding R by some rule-sets in Sr_{n-1}.

(3) *(folding)* $Sr_n=(Sr_{n-1}-\{R\})\cup\{R'\}$, where R is a rule-set in Sr_{n-1} and R' is the result of reversibly folding R by a rule-set in Sr_0.

(4) *(replacement)* $Sr_n=(Sr_{n-1}-\{R\})\cup\{R'\}$, where R is a rule-set in Sr_{n-1} and R' is the result of applying some replacement rule in Θ to R.

5.Examples

Here some examples are given in order to illustrate the definitions in the previous sections.

5.1.Elimination of Intermediate file

CCFG program G_4 in section two represents the **indirect** relation between the couples of terminal strings derived from the nonterminal symbol P and R, connecting two subprograms by the intermediation of "$Y\approx Z$". Since the two data structures specified by P and R have no structure cla in the meaning of [11], this program can be transformed into the program which represents the **direct** relation between two data structures. The following sequence indicates such a transformation.

First the initial set of rule-sets in G_4 is

$$
\begin{aligned}
Sr_0=\{\ & \{P\to X,\ Y\approx Z,\ R\to W\}, && ...(R.1)\\
& \{X\to\varepsilon,\ Y\to\varepsilon\}, && ...(R.2)\\
& \{X\to aX,\ Y\to cY\}, && ...(R.3)\\
& \{X\to bX,\ Y\to dY\}, && ...(R.4)\\
& \{Z\to\varepsilon,\ W\to\varepsilon\}, && ...(R.5)
\end{aligned}
$$

$\{Z \rightarrow cdZ, W \rightarrow eW\}\}$. ...(R.6)

Unfolding (R.1) at {Z,W} by (R.5) and (R.6),

$Sr_1=\{$ $\{P \rightarrow X, Y \approx \varepsilon, R \rightarrow \varepsilon\}$, ...(R.7)
$\{P \rightarrow X, Y \approx cdZ, R \rightarrow eW\}$, ...(R.8)
(R.2),(R.3),(R.4),(R.5),(R.6)}.

Unfolding (R.7) at {X,Y} by (R.2), (R.3) and (R.4),

$Sr_2=\{$ $\{P \rightarrow \varepsilon, \varepsilon \approx \varepsilon, R \rightarrow \varepsilon\}$, ...(R.9)
$\{P \rightarrow aX, cY \approx \varepsilon, R \rightarrow \varepsilon\}$, ...(R.10)
$\{P \rightarrow bX, dY \approx \varepsilon, R \rightarrow \varepsilon\}$, ...(R.11)
(R.8), (R.2),(R.3),(R.4),(R.5),(R.6)}.

Eliminating (R.10) and (R.11) by the simple rule-8 and eliminating the left-hand-omitted-rule "$\varepsilon \approx \varepsilon$" in (R.9) by the simple rule-7,

$Sr_3=\{$ $\{P \rightarrow \varepsilon, R \rightarrow \varepsilon\}$, ...(R.12)
(R.8), (R.2),(R.3),(R.4),(R.5),(R.6)}.

Unfolding (R.8) at {X,Y} by (R.2), (R.3) and (R.4),

$Sr_4=\{$ $\{P \rightarrow \varepsilon, \varepsilon \approx cdZ, R \rightarrow eW\}$, ...(R.13)
$\{P \rightarrow aX, cY \approx cdZ, R \rightarrow eW\}$, ...(R.14)
$\{P \rightarrow bX, dY \approx cdZ, R \rightarrow eW\}$, ...(R.15)
(R.12), (R.2),(R.3),(R.4),(R.5),(R.6)}.

Eliminating (R.13) and (R.15) by the simple rule-8, and further unfolding (R.14) at {X,Y} by (R.2), (R.3) and (R.4),

$Sr_5=\{$ $\{P \rightarrow a, c \approx cdZ, R \rightarrow eW\}$, ...(R.16),
$\{P \rightarrow aaX, ccY \approx cdZ, R \rightarrow eW\}$, ...(R.17),
$\{P \rightarrow abX, cdY \approx cdZ, R \rightarrow eW\}$, ...(R.18),
(R.12), (R.2),(R.3),(R.4),(R.5),(R.6)}.

Eliminating (R.16) and (R.17) by the simple rule-5 and eliminating the prefix "cd" of "$cdY \approx cdZ$" in (R.18) by the simple rule-5,

$Sr_6=\{$ $\{P \rightarrow abX, Y \approx Z, R \rightarrow eW\}$, ...(R.19),
(R.12), (R.2),(R.3),(R.4),(R.5),(R.6)}.

Folding (R.19) by (R.1) in Sr_0,

Sr_7={ {P→abP, R→eR}, ...(R.20),
(R.12), (R.2),(R.3),(R.4),(R.5),(R.6)}.

This folding operation is reversible.

Since the nonterminal symbols W, X, Y, and Z do not appear in any tuple of sentential forms derived by the tuple of start symbols, the rule-sets (R.2), (R.3),..., and (R.6) are applied no longer and we obtain the final program G'_4 as follows,

G'_4=({P,R},{a,b,c},(P,R),R),
R={ {P→ε, R→ε},
{P→baP, R→Rc} }.

The above program represents the direct relation between the data structures specified by the nonterminal symbols P and R.

5.2.Calculation of Squared Interger

The program to calculate a squared integer is given as the followings in functional style,

SQR(X)=MUL(X,X), ...(F.1)
MUL(X,Y)=if X=0 then 0 else Y+MUL(X-1,Y), ...(F.2)

where SQR is the function to calculate a squared integer and MUL is a multiply. More effective program can be given as follows,

SQR(X)=if X=0 then 0 else SQR(X-1)+(X-1)+(X-1)+1, ...(F.3)

because for an integer $n \geq 1$, $n^2=(n-1)^2+2*(n-1)+1$. Assuming that the execution cost of additions and subtractions can be less than that of multiplications, the latter program is more effective than the former.

In this subsection we show that a CCFG program which is equivalent to the former program can be transformed into that of the latter.

The initial CCFG program is given as follows,

SQR=({SQR_i,SQR_o,M1,M2,Mo,X},{@},(SQR_i, SQR_o),Sr_0),
Sr_0={ {SQR_i→M1≈M2, SQR_o→Mo}, ...(R.1)
{M1→ε, M2→X, Mo→ε}, ...(R.2)
{M1→@M1, M2→M2, Mo→M2Mo}, ...(R.3)

$\{X\to*\}\}$. ...(R.4)

Since the integer data type is not defined in CCFG programming, in the above program we use strings of "@" for representing integers. We suppose that the string with length n means the integer n. Then the meanings of the rule-sets (R.1), (R.2), and (R.3) are same as those of (F.1), the **then**-part of (F.2), and the **else**-part of (F.2), respectively. Because we use strings of "@" for integers, the addition of two integers is expressed by the concatenation of two strings. For example, 2+3 is expressed by "@@":"@@@". Therefore the right-hand side "M2Mo" of the third rule in the rule-set (R.3) means that M2+Mo.

First unfolding (R.1) by (R.2) and (R.3),

$$\begin{aligned} Sr_1=\{\ & \{SQR\_i\to\varepsilon\approx X,\ SQR\_o\to\varepsilon\}, && \text{...(R.5)}\\ & \{SQR\_i\to @M1\approx M2,\ SQR\_o\to M2Mo\}, && \text{...(R.6)}\\ & (R.2),(R.3),(R.4)\}. \end{aligned}$$

Replacing X in (R.5) for ε by the simple rule-9, and further applying the simple rule-1,

$$\begin{aligned} Sr_2=\{\ & \{SQR\_i\to\varepsilon,\ SQR\_o\to\varepsilon\}, && \text{...(R.7)}\\ & (R.6),(R.2),(R.3),(R.4)\}. \end{aligned}$$

Applying the replacement rule {M1/M2,M2/M1,Mo/Mo} to (R.6),

$$\begin{aligned} Sr_3=\{\ & \{SQR\_i\to @M2\approx M1,\ SQR\_o\to M1Mo\}, && \text{...(R.8)}\\ & (R.7),(R.2),(R.3),(R.4)\}. \end{aligned}$$

This replacement rule is correct because informally the rule is equivalent to the commutative law of multiplication, M1*M2=M2*M1=Mo, and precisely the language L(G') generated by the following grammar

$$\begin{aligned} G'=&(\{S_1,S_2,S_3,SQR\_i,SQR\_o,M1,M2,Mo\},\{@\},(S_1,S_2,S_3),R),\\ &R=\{\{S_1\to M1,\ S_2\to M2,\ S_3\to Mo\}\}\cup Sr_0, \end{aligned}$$

and the language L(G") generated by the following grammar

$$\begin{aligned} G''=&(\{S_1,S_2,S_3,SQR\_i,SQR\_o,M1,M2,Mo\},\{@\},(S_1,S_2,S_3),R),\\ &R=\{\{S_1\to M2,\ S_2\to M1,\ S_3\to Mo\}\}\cup Sr_0, \end{aligned}$$

are equal to each other.

Unfolding (R.8) by (R.2) and (R.3),

$$Sr_4=\{\ \{SQR\_i\to @X\approx\varepsilon,\ SQR\_o\to\varepsilon\}, \qquad \text{...(R.9)}$$

{SQR_i→@M2≈@M1, SQR_o→@M1M2Mo}, ...(R.10)
(R.7),(R.2),(R.3),(R.4)}.

Eliminating (R.9) by the simple rule-8, and substitute the substring "M2≈M1" of the right-hand side "@M2≈@M1"="@(M2≈M1)" of the first rule in (R.10) to the substrings "M1" and "M2" of the right-hand side "@M1M2Mo" of the second rule by the simple rule-4,

Sr_5={ {SQR_i→@(M2≈M1), SQR_o→@(M2≈M1)(M2≈M1)Mo}, ...(R.11)
(R.7),(R.2),(R.3),(R.4)}.

Folding (R.11) by (R.1) in Sr_0,

Sr_6={ {SQR_i→@SQR_i, SQR_o→@SQR_iSQR_iSQR_o}, ...(R.12)
(R.7),(R.2),(R.3),(R.4)}.

The rule-sets (R.7) and (R.12) have same meanings as the **then**-part and the **else**-part of (F.1), respectively. The final program is

SQR'=({SQR_i,SQR_o},{@},(SQR_i, SQR_o),Sr),
Sr={ {SQR_i→ε, SQR_o→ε},
{SQR_i→@SQR_i, SQR_o→@SQR_iSQR_iSQR_o}}.

5.3.Calculation of Fibonacci Number

The program to calculate Fibonacci numbers is often used for the exmaple of program transformations[5],[12]. Here we can construct the same transformation in CCFG programming.

The initial program is given as follows,

Fib=({Fin,Fout,G1,G2,G3},{@},(Fin,Fout),Sr_0),
Sr_0={ {G1→Fin, @Fin≈Fin_2, G2→$Fout_2$, G3→Fout$Fout_2$}, ...(R.1)
{Fin→ε, Fout→@}, ...(R.2)
{Fin→@, Fout→@}, ...(R.3)
{Fin→@Fin≈@@Fin_2, Fout→Fout$Fout_2$}}. ...(R.4)

The nonterminal symbols Fin_2 and $Fout_2$ suffixed with 2 are introducted so that name conflicts may be avoided, and the meanings of Fin_2 and $Fout_2$ are equivalent to those of Fin and Fout.

In the same way as in the previous subsection we also suppose that integers are expressed by strings of "@"s. The relation between the integers Fin and Fout means that the Fin-th value of Fibonacci sequence is Fout, that is **fib**(Fin)=Fout, where **fib**(n) is a function to calculate n-th value of the Fibonacii sequence.Then the rule-set (R.2) means that

$$\mathbf{fib}(0)=1,$$

the rule-set (R.3) means that

$$\mathbf{fib}(1)=1,$$

and the rule-set (R.4) means that

$$\mathbf{fib}(n)=\mathbf{fib}(n-1)+\mathbf{fib}(n-2),$$

for n≥2. The relation between G1, G2 and G3 means that

$$G2=\mathbf{fib}(G1+1)$$

and

$$G3=\mathbf{fib}(G1)+\mathbf{fib}(G1+1)=\mathbf{fib}(G1+2).$$

Although we find that Fibonacci numbers can be computed even if the rule-set (R.1) is never applied, however by introducing the rule-set, the program to compute the numbers more effectively can be transformed from the above original program.

First unfolding (R.1) at $\{Fin_3,Fout_3\}$ by (R.2),(R.3) and (R.4),

$Sr_1=\{$ $\{G1\to Fin, \varepsilon\approx @Fin, G2\to @, G3\to Fout@\}$, ...(R.5)
$\{G1\to Fin, @\approx @Fin, G2\to @, G3\to Fout@\}$, ...(R.6)
$\{G1\to Fin, @Fin_2\approx @@Fin_3\approx @Fin, G2\to Fout_2Fout_3, G3\to FoutFout_2Fout_3\}$, ...(R.7)
(R.2),(R.3),(R.4)$\}$.

The nonterminal symbols Fin_3 and $Fout_3$ suffixed with the integer 3 are introduced so that name conflicts may be avoided, and the meanings of Fin_3 and $Fout_3$ are equivalent to those of Fin and Fout.

Eliminating (R.5) by the simple rule-8, and unfolding (R.6) at {Fin,Fout} by (R.2),(R.3) and (R.4),

$Sr_2=\{$ $\{G1\to\varepsilon, @\approx @, G2\to @, G3\to @@\}$, ...(R.8)
$\{G1\to @, @\approx @@, G2\to @, G3\to @@\}$, ...(R.9)
$\{G1\to @Fin\approx @@Fin_2, @\approx @@Fin\approx @@@Fin_2, G2\to @, G3\to FoutFout_2@\}$, ...(R.10)
(R.7),(R.2),(R.3),(R.4)$\}$.

Eliminating (R.9) and (R.10) by the simple rule-8 and eliminating "@≈@" in (R.8) by the simple rule-7,

$Sr_3=\{$ $\{G1\to\varepsilon, G2\to @, G3\to @@\}$, ...(R.11)
(R.7),(R.2),(R.3),(R.4)$\}$.

Eliminating the prefix "@" of the second left-hand-omitted-rule in (R.7) by the simple rule-5,

Sr_4={ {G1→Fin, Fin_2≈@Fin_3≈Fin, G2→$Fout_2Fout_3$, G3→Fout$Fout_2Fout_3$},...(R.12)
(R.11),(R.2),(R.3),(R.4)}.

Applying the replacement rule θ_1={Fin≈Fin_2/Fin, Fout/Fout, $Fout_2$/Fout} to (R.12),

Sr_5={ {G1→Fin, Fin≈@Fin_3, G2→Fout$Fout_3$, G3→FoutFout$Fout_3$} , ...(R.13)
(R.11),(R.2),(R.3),(R.4)}.

The replacement rule is correct because informally the rule means that if m=n, then **fib**(m)=**fib**(n) for any integers m and n, and formally the language L(G') generated by the following program

G'=({S_1,S_2,S_3,Fin,Fout,G1,G2,G3},{@},(S_1,S_2,S_3),R),
R={{S_1→Fin≈Fin_2, S_2→Fout, S_3→$Fout_2$}}∪Sr_0,

and the language L(G") denerated by the following program

G"=({S_1,S_2,S_3,Fin,Fout,G1,G2,G3},{@},(S_1,S_2,S_3),R),
R={{S_1→Fin, S_2→Fout, S_3→Fout}}∪Sr_0,

are equal to each other.

Replacing the right-hand side "Fin" of the first rule by "@Fin_3" by the simple rule-4,

Sr_6={ {G1→@Fin_3, Fin≈@Fin_3, G2→Fout$Fout_3$, G3→FoutFout$Fout_3$}, ...(R.14)
(R.11),(R.2),(R.3),(R.4)}.

Applying the replacement rule θ_2={Fin/Fin, Fin_3/Fin_3, Fout$Fout_3$/$Fout_3$Fout} to (R.14),

Sr_7={ {G1→@Fin_3, Fin≈@Fin_3, G2→$Fout_3$Fout, G3→Fout$Fout_3$Fout}, ...(R.15)
(R.11),(R.2),(R.3),(R.4)}.

This replacement rule is correct because of the commutative law of addition, that is

fib(m)+**fib**(n)=**fib**(n)+**fib**(m)

for any integers m and n.

Folding (R.15) by (R.1) {G1→Fin_3,Fin≈@Fin_3,G2→Fout,G3→$Fout_3$Fout} in Sr_0,

Sr_8={ {G1→@G1, G2→G3, G3→G2G3}, ...(R.16)
(R.11),(R.2),(R.3),(R.4)}

This folding operation is reversible.

Next adding the left-hand-omitted-rule "$@Fin \approx @@Fin_2$" to (R.4) by the simple rule-3,

Sr_9={ {$Fin \to @Fin \approx @@Fin_2$, $Fout \to FoutFout_2$, $@Fin \approx @@Fin_2$}, ...(R.17)
(R.16),(R.11),(R.2),(R.3)}.

Replacing the right-hand side "$@Fin \approx @@Fin_2$" of the first rule in (R.17) by "$@@Fin_2$" by the simple rule-4,

Sr_{10}={ {$Fin \to @@Fin_2$, $Fout \to FoutFout_2$, $@Fin \approx @@Fin_2$}, ...(R.18)
(R.16),(R.11),(R.2),(R.3)}.

Eliminating the prefix "@" from the left-hand-omitted-rule "$@Fin \approx @@Fin_2$" in (R.18) by the simple rule-4,

Sr_{11}={ {$Fin \to @@Fin_2$, $Fout \to FoutFout_2$, $Fin \approx @Fin_2$}, ...(R.19)
(R.16),(R.11),(R.2),(R.3)}.

And further adding the left-hand-omitted-rule "$Fout_2$" to (R.19) by the simple rule-3,

Sr_{12}={ {$Fin \to @@Fin_2$, $Fout \to FoutFout_2$, $Fin \approx @Fin_2$, Fout}, ...(R.20)
(R.16),(R.11),(R.2),(R.3)}.

Applying the replacement rule θ_2={Fin/Fin, Fin_2/Fin_2, $FoutFout_2/Fout_2Fout$} to (R.20),

Sr_{13}={ {$Fin \to @@Fin_2$, $Fout \to Fout_2Fout$, $Fin \approx @Fin_2$, Fout}, ...(R.21)
(R.16),(R.11),(R.2),(R.3)}.

Folding (R.21) by (R.1) {$G1 \to Fin_2$,$Fin \approx @Fin_2$,$G2 \to Fout$,$G3 \to Fout_2Fout$} in Sr_0,

Sr_{14}={ {$Fin \to @@G1$, $Fout \to G3$, G2}, ...(R.22)
(R.16),(R.11),(R.2),(R.3)}.

At last, the final program is obtained as follow,

```
Fib'=({Fin,Fout,G1,G2,G3},{@},(Fin,Fout),R),
        R={   {Fin→ε,              Fout→@},
              {Fin→@,              Fout→@},
              {Fin→@@G1,  Fout→G3, G2} ,
              {G1→ε,      G2→@,  G3→@@},
              {G1→@G1,   G2→G3, G3→G2G3}}.
```

6.The Brief View of the Correctness

We have defined the program transformation rules in section three and four. In this section we discuss the correctness of the rules.

There are two methods to prove their correctness.

One is to prove straightforwardly. Because the mathematical background of CCFG programming is similar to that of logic programming, we can prove in the same way as [8].

Another is to prove indirectly by using the program transformation from logic programs into CCFG programs. Let L be a logic program and R(L) be the logic program transformed from L by applying a rule R in [8]. We have already known their equivalence if the transformation satisfies the assumptions A1, A2 and A3 in [8]. Let T(L) and T(R(L)) be the CCFG programs transformed from the logic program L and R(L), respectively, where T is a program transformation from an arbitrary logic program into the corresponding CCFG program which preserves the equivalence. Because the meanings of L and R(L) are equivalent to each other, so are the meanings of T(L) and T(R(L)). If the CCFG program T(L) can be transformed into R'(T(L)) by applying the rules R' presented in the previous sections, the application of R' can be stated to preserve the equivalence of the meanings. We say that R' in CCFG programming is the equivalent transformation rule as R in logic programming. That is illustrated bellow.

$$\begin{tikzcd}
 & \underline{\text{original program}} & \underline{\text{transformed program}} \\
\text{logic program} & L \arrow[r, "R"] \arrow[d, "T"] & R(L) \arrow[d, "T"] \\
\text{CCFG program} & T(L) \arrow[r, "R'"] & T(R(L))=R'(T(L))\ ?
\end{tikzcd}$$

In this paper we trace the brief line of the above contents by a example. The strict and precise discussions will be given in other papers.

For the simplicity of our discussion, the **canonical form** of a definite Horn clause is introduced.

Definition(canonical form). A definite Horn clause of the form

$$p(P_1,\dots,P_i)\text{:-}q(Q_1,\dots,Q_j),\dots,r(R_1,\dots,R_k),P_1=s_1,\dots,P_i=s_i,t_1=u_1,\dots,t_n=u_n.,$$

is said to be of the canonical form where $P_1,\dots,R_k$ are variables and $s_1,\dots,u_n$ are terms. The name of a variable P_n can be uniquely identified with the name of a predicate symbol p and the occurence position n of the variable in an atom $p(\dots,P_n,\dots)$. A set of definite clauses of the cannonical form is said to be a logic program of the cannonical form. Here the predicate "=" is defined by the Horn clause, X=X:-true.

It is clear that an arbitrary definite clause can be transformed into one of the canonical form, and that an original logic program (a set of definite clauses) can be trasnformed into the logic program (a set of definite clause of the canonical form) which has the same meaning as the original one. We consider logic programs of the canonical form in the following discussions.

Definition(transformation). A Horn clause C of the canonical form

$$p(X_1,...,X_i):-q(Y_1,...,Y_j),...,r(Z_1,...,Z_k),X_1=s_1,...,X_i=s_i,t_1=u_1,...,t_n=u_n.,$$

is translated into the rule-set T(C),

$$T(C)=\{X_1\to i(s_1),...,X_i\to i(s_i),i(t_1)\approx i(u_1),...,i(t_j)\approx i(u_k)\},$$

and a set of some other completementary rule-sets c-T(C),

$$c\text{-}T(C)=\{\{i(V_1)\to *\},...,\{i(V_n)\to *\}\},$$

where $V_1,...,V_n$ are all the variables which appear in $s_1,...,u_k$, and function ***i*** is a mapping from a term to a string, defined as follows,

(1) if α is a variable, $i(\alpha)$ is the corresponding nonterminal symbol.

(2) if α is a constant, $i(\alpha)$ is the corresponding terminal symbol.

(3) if α is a term $f(\alpha_1,..., \alpha_n)$ constructed by a n-ary functor f and terms $\alpha_1,...,\alpha_n$, $i(\alpha)$ is the corresponding string "f(":$i(\alpha_1)$:",": ... :",":$i(\alpha_n)$:")", where "f","(",")", and "," are terminal symbols and ":" is a concatenation operator.

Let L be a logic program. When we consider the meaning for a n-ary predicate symbol p in L, T is a transformation from the logic program L into the CCFG program $T(L)=(N,T,\Omega_S,R)$, where Ω_S is a n-tuple $(i(P_1),...,i(P_n))$, R is a set $\{T(C)\cup c\text{-}T(C)|c$ is a clause in L$\}$ of rule-sets, N is a set of all the nonterminal symbols which appear in R, and T is a set of all the terminal symbols which appear in R.

Proposition. Let L be a logic program of the canonical form and T(L) be the CCFG program transformed from L. If there exists a CCFG program transformation $T(L)\Rightarrow^* R'(T(L))$, then there exists a logic program transformation $L\Rightarrow^* R(L)$, and the following condition holds,

$$T(R(L))=R'(T(L)).$$

Because the total correctness of the transformation $L\Rightarrow^* R(L)$ has been proved in [8], then the total correctness of the transformation $T(L)\Rightarrow^* R'(T(L))$ holds.

In this paper we do not present the proof of the above proposition, because it requires much more preliminaries and more considerations. It will be given in other papers. Here we show it with the following example.

The following set is a logic program

```
L0={  p(A,B):-q(A,C),r(C,B).,
      q(nil,nil).,
      q(cons(a,A),cons(c,B)):-q(A,B).,
      q(cons(b,A),cons(d,B)):-q(A,B).,
      r(nil,nil).,
      r(cons(c,cons(d,A)),cons(e,B)):-r(A,B).},
```

where nil, "a", "b","c" and "d" are constants and "cons" a functor. Considering the predicate symbo p, the meaning for the predicate symbol p is the follwoing set of ground atoms,

$M(p,L_0)$={p(nil,nil),p(cons(a,nil),cons(c,nil)),p(cons(b,nil),cons(d,nil)),p(cons(a,cons(b,nil)),co ns(c,cons(d,nil))),...}

Program L_0 is transformed into the program of the canonical form as follows,

L_1={	p(P1,P2):-q(Q1,Q2),r(R1,R2).,P1=Q1,P2=R2,Q2=R1.,	...(C.1)
	q(Q1,Q2):-Q1=nil,Q2=nil.,	...(C.2)
	q(Q1,Q2):-q(Q1$_2$,Q2$_2$),Q1=cons(a,Q1$_2$),Q2=cons(c,Q2$_2$).,	...(C.3)
	q(Q1,Q2):-q(Q1$_2$,Q2$_2$),Q1=cons(b,Q1$_2$),Q2=cons(d,Q2$_2$).,	...(C.4)
	r(R1,R2):-R1=nil,R2=nil.,	...(C.5)
	r(R1,R2):-r(R1$_2$,R2$_2$),R1=cons(c,cons(d,R1$_2$)),R2=cons(e,R2$_2$),.}.	...(C.6)

Program L_1 for the predicate symbol p is translated into the following CCFG program,

$T(L_1)$=({P1,P2,Q1,Q2,R1,R2,A,B,C},{a,b,c,d,e,cons,(,),,},(P1,P2) ,R),
R={ {P1→Q1,P2→R2,Q2≈R1},
{Q1→nil, Q2→nil},
{Q1→cons(a,Q1),Q2→cons(c,Q2)},
{R1→nil,R2→nil},
{R1→cons(c,cons(d,R1)),R2→cons(e,R2)}}.

If we identify a term "cons(X,Y)" with a string "X:Y", and the atom "nil" as an empty string "ε", we find that program $T(L_1)$ is equivalent to program G_4. In the same way as in subsection 5.1, it is easily understood that program $T(L_1)$ can be transformed into the more effective CCFG program as follows,

$R'(T(L_1))=(\{P1,P2\},\{a,b,e,cons,(,),,\},(P1,P2),R),$
$\{P1\rightarrow nil, P2\rightarrow nil\},$
$\{P1\rightarrow cons(a,cons(b,P1)), P2\rightarrow cons(e,P2)\}.$

Acoording to the above proposition, there exists such a logic program $R(L_1)$ as satisfies that

$$R'(T(L_1))=T(R(L_1)).$$

For an example of the above proposition, we can show the same program transformation from L_1 into $R(L_1)$ as in the subsection 5.1.

The initial program of this transformation is $Sc_0=L_1$. First unfolding (C.1) by (C.5) and (C.6),

```
Sc1={ p(P1,P2):-q(Q1,Q2),R1=nil,R2=nil,P1=Q1,P2=R2,Q2=R1.,              ...(C.7),
      p(P1,P2):-q(Q1,Q2),r(R12,R22),R1=cons(c,cons(d,R12)),R2=cons(e,R22),
              P1=Q1,P2=R2,Q2=R1.,                                      ...(C.8),
      (C.2),(C.3),(C.4),(C.5),(C.6)}.
```

The clauses (C.7) and (C.8) are corresponding to the rule-set (R.7) and (R.8) in subsection 5.1, and in the same way a clause (C.n) appearing bellow is corresponding to the rule-set (R.n) in subsection 5.1 for $20\geq n\geq 1$. We see that the transformed rule-set from a cluase (C.n) by the transformation rule T described above is equivalent to the rule-set (R.n) in subsection 5.1.

Because we use the canonical form, we need not to pay attention to applicability of (C.5) and (C.6) to r(R1,R2) in (C.1). The clauses (C.7) and (C.8) are obviously equivalent to the following clauses,

```
p(P1,P2):-q(Q1,Q2),P1=Q1,P2=nil,Q2=nil.,                               ...(C.7'),
p(P1,P2):-q(Q1,Q2),r(R12,R22),P1=Q1,P2=cons(e,R22),
        Q2=cons(c,cons(d,R12)).,                                       ...(C.8').
```

Sc_1 is corresponding to Sr_1, since $T(Sc_1)$ is equivalent to Sr_1.

Unfolding (C.7') at {X,Y} by (C.2), (C.3) and (C.4),

```
Sr2={ p(P1,P2):-Q1=nil,Q2=nil,P1=Q1,P2=nil,Q2=nil.,                   ...(C.9),
      p(P1,P2):-q(Q12,Q22),Q1=cons(a,Q12),Q2=cons(c,Q22),P1=Q1,P2=nil,Q2=nil.,
                                                                       ...(C.10),
      p(P1,P2):-q(Q12,Q22),Q1=cons(b,Q12),Q2=cons(d,Q22),P1=Q1,P2=nil,Q2=nil.,
                                                                       ...(C.11),
      (C.8'), (C.2),(C.3),(C.4),(C.5),(C.6)}.
```

Eliminating (C.10) and (C.11) because the bodies do not succeed by any means, and transforming (C.9) into the following equivalent clause,

Sr_3={ p(P1,P2):-P1=nil,P2=nil., ...(C.12),
(C.8'), (C.2),(C.3),(C.4),(C.5),(C.6)}.

Unfolding (C.8') by (C.2), (C.3) and (C.4),

Sr_4={ p(P1,P2):-Q1=nil,Q2=nil,r($R1_2$,$R2_2$),P1=Q1,P2=cons(e,$R2_2$),
Q2=cons(c,cons(d,$R1_2$))., ...(C.13),
p(P1,P2):-q($Q1_2$,$Q2_2$),Q1=cons(a,$Q1_2$),Q2=cons(c,$Q2_2$),r($R1_2$,$R2_2$),P1=Q1,
P2=cons(e,$R2_2$),Q2=cons(c,cons(d,$R1_2$))., ...(C.14),
p(P1,P2):-q($Q1_2$,$Q2_2$),Q1=cons(b,$Q1_2$),Q2=cons(d,$Q2_2$),r($R1_2$,$R2_2$),P1=Q1,
P2=cons(e,$R2_2$),Q2=cons(c,cons(d,$R1_2$))., ...(C.15),
(C.12), (C.2),(C.3),(C.4),(C.5),(C.6)}.

Eliminating (C.13) and (C.15) because the bodies do not succeed by any means, and (C.14) is transformed into the following equivalent clause,

p(P1,P2):-q(Q1,Q2),r(R1,R2),P1=cons(a,Q1),P2=cons(e,R2),
cons(c,Q2)=cons(c,cons(d,R1)). ...(C.14').

Further unfolding (C.14') by (C.2), (C.3) and (C.4),

Sr_5={ p(P1,P2):-Q1=nil,Q2=nil,r(R1,R2),P1=cons(a,Q1),P2=cons(e,R2),
cons(c,Q2)=cons(c,cons(d,R1))., ...(C.16),
p(P1,P2):-q($Q1_2$,$Q2_2$),Q1=cons(a,$Q1_2$),Q2=cons(c,$Q2_2$),r(R1,R2),
P1=cons(a,Q1),P2=cons(e,R2),cons(c,Q2)=cons(c,cons(d,R1)).,
...(C.17),
p(P1,P2):-q($Q1_2$,$Q2_2$),Q1=cons(b,$Q1_2$),Q2=cons(d,$Q2_2$),r(R1,R2),
P1=cons(a,Q1),P2=cons(e,R2),cons(c,Q2)=cons(c,cons(d,R1)).,
...(C.18),
(C.12), (C.2),(C.3),(C.4),(C.5),(C.6)}.

Eliminating (C.16) and (C.17) because the bodies do not succeed by any means, and transforming (C.18) into the following equivalent clause,

Sr_6={ p(P1,P2):-q(Q1,Q2),r(R1,R2),P1=cons(a,cons(b,Q1)),P2=cons(e,R2),Q2=R1.,
...(C.19),
(C.12), (C.2),(C.3),(C.4),(C.5),(C.6)}.

The molecule "q(Q1,Q2),r(R1,R2),Q2=R1" is an L_1-expansion of p(Q1,R2) because for the substitution θ={P1/Q1,R2/P2} the following condition holds,

(p(P1,P2):-q(Q1,Q2),r(R1,R2),P1=Q1,P2=R2,Q2=R1.)θ=
p(Q1,R2):-q(Q1,Q2),r(R1,R2),Q2=R1.

Then folding "q(Q1,Q2),r(R1,R2),Q2=R1" of (C.19) into p(Q1,R2), the following clause is obtained,

Sr_7={ p(P1,P2):-p(Q1,R2),P1=cons(a,cons(b,Q1)),P2=cons(e,R2)., ...(C.20),
(C.12),(C.2),(C.3),(C.4),(C.5),(C.6)}.

This folding operation is reversible. Variables in (C.20) can be renamed as follows,

p(P1,P2):-p($P1_2$,$P2_2$),P1=cons(a,cons(b,$P1_2$)),P2=cons(e,$P2_2$)., ...(C.20'),

At last the final program is obtained as follows,

$R(L_1)$={p(P1,P2):-P1=nil,P2=nil.,
p(P1,P2):-p($P1_2$,$P2_2$),P1=cons(a,cons(b,$P1_2$)),P2=cons(e,$P2_2$).}.

We find that $T(R(L_1))=R'(T(L_1))$. Becausethe above transformation satisfies the assumption A.1, A.2 and A.3 in [8], the correctness of the above logic program transformation $L_1 \Rightarrow^* R(L_1)$ holds. So does the correctness of the CCFG program transformation $T(L_1) \Rightarrow^* R'(T(L_1))$. And to say further, the transformation $G_4 \Rightarrow^* G_4'$ is correct.

7.Discussion

In this paper, we have discussed the program transformation rules in CCFG programming, mainly the unfolding and folding rules.

The following problems to solve the next step remain,

(1) to precisely prove the total correctness of the rules.

(2) to well-define the transformation strategies proposed in [3] by using our transformation rules.

Especially the problem (2) is important to construct an automatic program transformaiton system.

References

[1] Yamashita,Y. and Nakata,I.:An extension of context free grammar and the computation model based on the extended grammar, JIPS Working Group Report of Software Foundation SF15-5(1986) (in Japanese).

[2] Yamashita,Y. and Nakata,I.:Coupled Context Free Grammar and its Interpretation as a Programming Language, RIMS Symposia Report on Software Science and Engineering, Kyoto University(1986).

[3] Nakata,I. and Yamashita,Y.:Program transformation in Coupled Context Free Grammar, JIPS Working Group Report of Software Foundation SF17-3(1986) (in Japanese).

[4] Yamashita,Y. and Nakata,I.:The execution method of the programming language based on Coupled Context Free Grammar, JIPS Proc. 28th Programming Simposium(1987) (in Japanese).

[5] Burstall,R.M. and Darlington,J.:A transformation system for developing recursive programs, J.ACM 24(1977),pp.44-67.

[6] Clark,K.L.:Predicate logic as a computational formalism, Imperial College Reserch Monograph 79/59 TOC(1979).

[7]Hogger,C.J.:Derivation of logic programs, J.ACM 28(1981) pp.372-392.

[8] Tamaki,H. and Sato,T:A generalized correctness proof of the unfold/fold logic program transformation, Technical Report No.86-4, Ibaraki University(1986).

[9] Kanamori,T. and Fujita,H.:Unfold/fold transformation of logic programs with counters, ICOT Technical Report(1986).

[10] Aho,A.V. and Ullman,J.D.:"The theory of parsing,translation and compiling Vol 1:Parsing", Prentice-Hall(1972).

[11] Jackson,M.A.:"Principles of program design", Academic Press(1975).

[12] Tamaki,H. and Sato.T:A transformation system for logic programs which preserves equivalence, ICOT Technical Report-018(1982).

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

REPORT DOCUMENTATION PAGE	REPORT NUMBER ISE-TR-88-72

TITLE

Unfold/Fold Program Transformation in CCFG programming

AUTHOR(S)

Yoshiyuki YAMASHITA and Ikuo NAKATA.

REPORT DATE Jun. 1, 1988	NUMBER OF PAGES 26
MAIN CATEGORY Programming Languages	CR CATEGORIES D.3, F.3, F.4

KEY WORDS

Program Transformation, Unfolding, Folding,
Coupled Context-Free Grammar

ABSTRACT

The program transformation rules, unfolding, folding and replacement, in CCFG programming are proposed. Because of the resembance between CCFG programming and logic programming, we can defined the rules in the same way as in logic programming. The correctness is also discussed.

SUPPLEMENTARY NOTES