ソフトウェア工学 96－8
（1994. 1. 21）

部品階層に基づくオブジェクト指向プログラミング トランスレータの検証

平松 耕太郎† 魚井 宏高† 萩原 兼一‡ 首藤 勝†

†大阪大学 基礎工学部
‡奈良先端科学技術大学院大学

筆者らは部品階層に基づくオブジェクト指向プログラミングを提案している．部品階層では，システムを構成するオブジェクト群が成す構造を階層的に定義することにより，プログラムの読みやすさを改善し，また，メッセージを配送する機能を提供することで，階層を構成するオブジェクト間で柔軟なメッセージの送信が可能となっている．この部品階層はObjective-C 言語の文法を拡張して実装されており，通常の Objective-C 言語のプログラムに変換するトランスレータも開発されている．本稿では，この部品階層に基づくプログラムをπ計算で記述するための変換規則を定義し，トランスレータによる変換前後のプログラムの動作を計算を用いて比較することにより，メッセージ配送のトランスレートについて検証を行なう．

Verifying the translator for aggregate based object-oriented programs

Kotaro HIRAMATU† Hirotaka UOI† Kenichi HAGIHARA‡ Masaru SUDO†

†Faculty of Engineering Science, Osaka University
‡Nara Institute of Science and Technology

A new programming paradigm AOOP, Aggregate based Object-Oriented Programming, is proposed. Aggregate is a kind of template for composite objects, which defines relations between each object as tree structure and provides a message delivering mechanism. By using this paradigm, it is possible to design systems more simply. In this paper, semantics for aggregate based object-oriented programming language is presented by translation into π-calculus, and then we verify translator which translates programs using aggregate into ones in normal Objective-C language, to show its correctness on message delivering mechanism.

1 はじめに

オブジェクト指向システムは，一般に複数のオブジェクトから構成され，動作時には，それらの間でメッセージのやり取りが行なわれる．このようなシステムを記述するために，従来の多くのオブジェクト指向言語では，各オブジェクトがメッセージ送信先のオブジェクトを指定するための識別子を持つことが必要となっている．したがって，これらのプログラミング言語では，システムの規模が大きくなるにつれて識別子を取得・保持するための記述が増大してしまう．また，オブジェクトが相互に識別子を持ち合うことにより，一般にオブジェクト間の参照関係が網状になるので，プログラムの理解が困難になると考えられる．

このような問題点に対して，筆者らは，オブジェクト群の成す構造を階層構造（部品階層）に統一して記述するプログラミングを提案し，プログラムの理解を助け，記述量の軽減を図っている．また，この階層構造では，オブジェクトの識別子を持たなくてもメッセージのやりとりが行なえるようにするためのメッセージの送信機能（メッセージ配送機構と呼ぶ）を提供している．さらに，この部品階層の概念をObjective-C言語を拡張して実装し，通常のObjective-C言語のプログラムに変換するトランスレータを開発している[4][5][6].

このトランスレータでは，部品階層の定義と，メッセージ配送機構を使用したメソッドを含むプログラムが，オブジェクト群の生成や初期化，メッセージ転送を実現するためのクラス群を追加したプログラムに変換される．そこで筆者は，部品階層に基づくプログラミングの言語処理系をより信頼性の高いものとするために，メッセージの配送に注目してトランスレータの検証を行う．検証するための枠組みとして，まず部品階層に基づくオブジェクト指向プログラミング言語に相当する言語$\mathcal{L}$の文法をπ計算に変換する変換規則を定義し，この言語の意味の定義を行う．そして，π計算への変換規則を用いてトランスレータによる変換される前のプログラムと，変換後のプログラムをπ計算で表し，実際に計算することでメッセージの配送について両者の比較を行う．メッセージ配送機構では3種類の配送方法が提供されているが，本稿ではそのうちの一つである相対・絶対表記を用いたオブジェクトの指定についてトランスレータを検証する．

本稿の構成は，2節で部品階層について説明し，3節で言語$\mathcal{L}$をπ計算で表す．そして，4節では，計算を用いてプログラムの等価性を定義し，相対・絶対表記によるオブジェクトの指定を用いたメッセージ配送についてトランスレータの検証を行う．

2 部品階層に基づくオブジェクト指向プログラミング

2.1 部品階層

システムの設計・構築は，トップダウン法やボトムアップ法などのように，一般的に階層的に行なわれることが多いと考えられる．しかし，システム動作時にオブジェクト群の成す構造は一般に階層構造にはならない．それは，各オブジェクトが階層上の直接の親・子だけでなく，兄弟あるいは親の兄弟などと相互作用を行なうことがあるためである．このため，プログラマは，プログラミング時に相互作用に着目して必要なオブジェクトの識別子を得るためのメソッドや変数を定義しなくてはならない．

そこで，筆者らは，システムの動作時にオブジェクト群が成す構造のうち静的に定まる部分を，階層構造に統一して定義することを考えた．そして，この階層単位での情報隠蔽・再利用を実現するため，階層の各ノード（オブジェクト）のクラスなどの定義は，従来のクラス定義とは別個に行うことにした．また，オブジェクトの識別子を得るためのメソッドや変数を定義せずに，階層上の直接の親・子・兄弟あるいは親の兄弟等にメッセージ送信を行えるよう，メッセージ配送機構を提供する．このような機構を持つ階層構造を部品階層と呼ぶ[4][5][6].

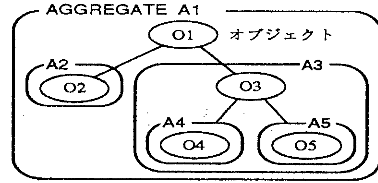

図 1: AG

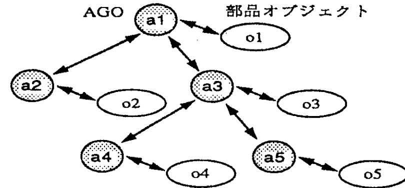

図 2: AGO の実装

メッセージ配送機構には，相対・絶対表記によるオブジェクトの指定，上昇伝播，放送という3通りの配送方法があるが，ここでは，相対・絶対表記によるオブジェクトの指定のみを述べる．これは，メッセージ送信先のオブジェクトを，UNIXのディレクトリ表記のように相対あるいは絶対表記を用いて指定する機能である．この機能を利用することにより，オブジェクトの識別子で指定しなくても，階層上の任意のオブジェクトに対してメッセージを送信できる．

2.2 AGGREGATE

筆者らは，2.1節で述べた部品階層を再利用・情報隠蔽の単位とするため，部品階層のひな型になる概念を導入し，AGGREGATE(AG)と呼んでいる．システムの動作時にはひな型であるAGをインスタンス化することで，部品階層を構築する．

一つの部品階層はさらにいくつかの子の部品階層から構成されるため，AGは階層的に定義される(図1)．部品階層を木構造と考えたとき，ひとつのAGは，

(1) このAGをひな型とする部品階層の根となるオブジェクト　と

(2) (1)の子オブジェクトを根とするいくつかの部品階層

から構成される．以後，このAGをスーパーAG, (1)をこのAGに属するオブジェクト，(2)のひな型のAGをサブAGと呼ぶ．

ひな型であるAGをインスタンス化すると，そのAGに対応するオブジェクトが生成される．これをAGオブジェクト(以下，AGO)と呼ぶ．AGOは各AG毎に生成され，そのAGに属するオブジェクト，サブAGのAGO，およびスーパーAGのAGOが相互に参照できる木構造として部品階層を構築する．2.1節で述べたメッセージ配送機構は，このAGOの機能として実現される．例えば，図2に示す部品階層において部品オブジェクト“$o5$”から“$o2$”を指定

```
AG A1 has
        o1 : CLASS C1;
        a2 : AG A2 has
                o2 : CLASS C2
        end;
        a3 : AG A3 has
                o3 : CLASS C3;
                a4 : AG A4 has
                        o4 : CLASS C4
                end;
                a5 : AG A5 has
                        o5 : CLASS C5
                end;
        a6 : AG A2
        end
end
```

図 3: AG "A1"

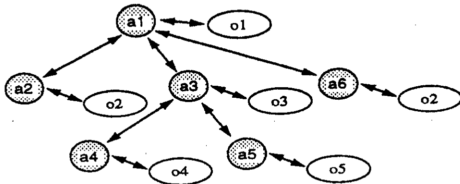

図 4: インスタンス化された AG "A1"

する場合，相対表記ならば "../../a2/o2"，絶対表記ならば "/a2/o2" と書く．ここで，"/" は AGO $a1$ を表す．このAGO $a1$ をルート AGO と呼ぶ．

先に述べたように，AG は部品オブジェクトといくつかのサブ AG から階層的に定義される．このとき，入れ子状に定義された AG をそのスーパー AG の宣言中でのみ再利用することができる．この場合のスコープの考え方は，Pascal 言語における関数・手続きのスコープと同様である．例えば，図 3 の部品階層においては，AG "a6" として AG "A2" が再利用されている．AG "A1" がインスタンス化されたときには，図 4 に示す部品階層が生成される．

AGの再利用を行うと，一つの相対表記に対して指定されるオブジェクトが複数存在する場合が生じる．例えば，図 3 中のクラス "C2" で相対表記を用いたメッセージ式 "[./o2 *msg*]" が定義されているとする．このとき，AG "A2" に属するオブジェクト "o2" からこの表記で指定されるオブジェクトは，絶対表記 "/a2/o2" で指定されるオブジェクトと同じである．ところが，AG "A6" に属するオブジェクト "o2" からこの表記で指定されるオブジェクトは，絶対表記 "/a6/o2" に相当するオブジェクトである．

3 π計算による部品階層に基づくオブジェクト指向言語の表現

前節で述べたような文法の拡張を行なった Objective-C 言語をπ計算で表現するために，この言語に相当するオブジェクト指向言語 $\mathcal{L}$ を考える．さらに，その言語をπ計算を用いて意味を記述する．

3.1 オブジェクト指向言語の文法

部品階層を使用するための宣言を持つオブジェクト指向言語 $\mathcal{L}$ を定義する．ここでは，言語の文法を簡略化するために，クラスの定義では継承が完了していることを仮定する．つまり，サブクラスはスーパークラスで定義された変数やメソッドをすべて持っているものと考える．また，ここでは各々のメソッドが引数と返り値を 1 つずつ持つものとして表現するが，引数の数については容易に規則を拡張できる．言語 $\mathcal{L}$ の式 E，文 S を次のように定義する．ここで，X は変数を表す．

$$
\begin{array}{llll}
E & ::= & X & \\
 & | & k & (k = 0,\ 1,\ 2, \ldots) \\
 & | & b & (b = \textbf{true},\ \textbf{false}) \\
 & | & \textbf{nil} & \\
 & | & \textbf{self} & \\
 & | & E_1 + E_2 & \\
 & | & E_1 = E_2 & \\
 & | & \textbf{not}\ E & \\
 & | & E_1\ \textbf{and}\ E_2 & \\
 & | & [\,E_1\ M : E_2\,] & (M\text{はメソッド名}) \\
 & | & D & (D\text{は相対・絶対表記}) \\
S & ::= & X = E & \\
 & | & X = [\,C\ \textbf{new}\,] & (C\text{はクラス名}) \\
 & | & [\,E_1\ M : E_2\,] & (M\text{はメソッド名}) \\
 & | & S_1\ ;\ \ S_2 & \\
 & | & \textbf{if}\ E\ \textbf{then}\ S_1\ \textbf{else}\ S_2 & \\
 & | & \textbf{while}\ E\ \textbf{do}\ S &
\end{array}
$$

変数，メソッド，クラス，部品階層，プログラムの宣言は，次のように定める．C はクラス名，$M_i (i = 1 \ldots max\{k, l\})$ はメソッド名，A は AG 名，O は AG オブジェクト名を表し，これらの名前は一意に定められたものとする．また，j, k, l, m, n は自然数を表す．

$$
\begin{array}{lll}
Pdec & ::= & Adec_1, \ldots,\ Adec_m, Cdec_1, \ldots,\ Cdec_n,\ Main \\
Adec & ::= & \textbf{AG}\ A\ \textbf{has}\ O : \textbf{CLASS}\ C\ Parts \\
Parts & ::= & \textbf{AG}\ A_1, \ldots,\ A_j \\
Cdec & ::= & \textbf{interface}\ C\ \{\ Vdec\ \}\ Mlist\ Mdec \\
Vdec & ::= & t_1\ X_1;\ \ldots\ t_n\ X_n;\quad (t_i = \textbf{int}, \textbf{bool}, \textbf{id}) \\
Mlist & ::= & M_1;\ \ldots\ M_k; \\
Mdec & ::= & \textbf{implementation} \\
 & & \quad M_1(X_1, Y_1)\ \{\ S_1;\ \textbf{return}\ Y_1;\ \} \\
 & & \quad \ldots\ M_l(X_l, Y_l)\ \{\ S_l;\ \textbf{return}\ Y_l;\ \} \\
Main & ::= & \textbf{main}\ \{\textbf{AG}\ X; Vdec\ S\ \}
\end{array}
$$

プログラム $Pdec$ は，有限個のクラス $Cdec_i$ と部品 $Adec_j$ $(i = 1 \ldots n,\ j = \ldots m)$，メイン関数 $Main$ から構成される．ここで，$Adec_1$ ではルート AG の定義を行なうものとする．$Main$ では，プログラムを初期化し，計算を開始するためのコードを記述する．

$Adec$ では，部品階層を構成する各 AG がそれぞれ定義される．サブ AG を持たない場合には，$Parts$ の部分が省略される．

$Mdec$ 中で定義されるメソッドはそれぞれが X, Y という変数を持つ．変数 X は引数を保持するために，変数 Y は返り値を保持するために使用する．

3.2 π計算の文法

π計算とは，通信機構を持つシステムを自然に表現できる計算である [2][3]．

計算システムはそれぞれが独立した agent の集合として表現される．それらのうちのいくつかは，名前の付けられたリンクを共有しており，通信を行なうことができる．そして，2 つの agent P, Q 間で相互作用が行なわれるとは，それらが共有するリンク x を通して，別のリンクを表す名前 y が送受信されることをさす[1]．ただし，通信が行われるのは，あるリンクに対して送信を行う agent と受信を行う agent が並列に動作している場合に限られる．

x, y を名前とし，A を agent 識別子（非負の引数を持つ）とする．このとき，π計算の agent（P, Q で表す）は，次の

[1] 共有するリンク x に対してリンク名 x を送信することもできる．

ように定義される．

$$
\begin{array}{rcll}
P & ::= & \sum_{i\in I} P_i & \textit{summation} \\
 & | & \alpha.P & \textit{prefix} \\
 & | & P \mid Q & \textit{composition} \\
 & | & (y)P & \textit{restriction} \\
 & | & [x=y]P & \textit{match} \\
 & | & A(\tilde{y}) & \textit{agent identifier}
\end{array}
$$

ここで，I は有限集合を，α は$\overline{x}y, x(y), \tau$ のいずれかを，また$\tilde{y}$ は A の引数を表しており，$\tilde{y} = y_1 \dots y_n$とする．

このとき，それぞれの agent は次のように動作する．

- summation
  P_i のうちのどれか一つとして動作する agent を表す．何も実行できない agent を 0 と表記する．
- prefix
  $\overline{x}y.P$ は，リンク x に名前 y を送信した後，P の動作を行う．
  $x(y).P$ は，リンク x から名前を受信した後，$P\{z/y\}$ として動作する．ここで，$P\{z/y\}$ とは，P 内の束縛されていないすべての y を z で置き換えることである．
  $\tau.P$ は，内部動作τを行った後，P の動作を行う．
- composition
  agent P, Q が並列に動作する．ただし，これらの agent の間で通信が行なわれる可能性があり，ここでの通信は内部動作として扱う．
- restriction
  括弧内に表わされた名前 y は，agent P 内の局所的な名前であるという制限を表す．したがって，P 以外のどんな agent も，このリンクを用いて P と通信することができない．
- match
  $x = y$ のとき，P の動作を行い，それ以外の場合，0 である．
- agent identifier
  $A(\tilde{x}) = P$ （ここで，agent P は異なる変数 x_i を持ち，P 中の束縛されていない変数が $\tilde{x}$ に含まれている）と定義されているとき，$A(\tilde{y})$ は $P\{\tilde{y}/\tilde{x}\}$ として動作する．

さらに，次のような省略記法を用いる．

- 再帰的に定義された agent
  $\alpha * P = \alpha.\ (\ P \mid \alpha * P\ )$
- match 文を用いた分岐表現
  $x(z).\ ([z = y_1]P_1 + [z = y_2]P_2 + \dots)$
  を次にように省略して表現する．
  $x : [\ y_1 \Rightarrow P_1,\ y_2 \Rightarrow P_2,\ \dots\ ]$

agent の動作は，遷移システム $\langle P, \alpha, \{\xrightarrow{\mu} \mid \mu \in \alpha\}\rangle$ で与えられる．ここで，$\alpha = \{\overline{x}y, x(y), \tau, \overline{x}(y)\}$，$\xrightarrow{\mu} \subseteq P \times P$ であり，$\overline{x}(y)$ とは，リンク x に対して局所的な名前 y を送信する動作を表す．主な動作には次のものがあるが，詳細は [2] に述べられている．

$$\frac{-}{\tau.\ P \xrightarrow{\tau} P} \qquad \frac{-}{\overline{x}y.\ P \xrightarrow{\overline{x}y} P}$$

$$\frac{-}{x(z).\ P \xrightarrow{x(w)} P\{w/z\}} \qquad \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$$

$$\frac{P \xrightarrow{\overline{x}y} P' \quad Q \xrightarrow{x(z)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'\{y/z\}} \qquad \frac{P \xrightarrow{\overline{x}(w)} P' \quad Q \xrightarrow{x(w)} Q'}{P \mid Q \xrightarrow{\tau} (w)(P' \mid Q')}$$

3.3 π計算による言語の表現

本節では，3.1 節で定めた言語 $\mathcal{L}$ について，その意味をπ計算で記述する．言語 $\mathcal{L}$ は言語 $\mathcal{L}_2$[3] を拡張して定義されており，部品階層の宣言が追加され，クラスの宣言などが変更されている．したがって，これらの追加・変更されている文法を中心にπ計算を用いて表す．変換規則が定義されていない文法については，言語 $\mathcal{L}_2$ に対してπ計算で定められた変換規則と同一である．

ここで，意味の記述とは，言語 $\mathcal{L}$ の文法に基づいて作成されるプログラムに対して，そのプログラムが実行されたときのプログラムの動作をπ計算を用いて表現することである．任意のプログラムの動作を形式的に表現するために，ここではプログラミング言語の文法上の規則をπ計算の agent に対応させる変換規則を定義する．変換規則を定義する際には，変換の対象が構文単位 rであることを明示するために，次のような記法を使用する．

$$[\![\ r\ ]\!] \stackrel{\text{def}}{=} \pi\text{計算の}\ agent$$

π計算では，agent が相互作用を行なうために，リンクを通じて名前を通信する必要がある．そこで，$[\![\ r\ ]\!](v)$ という記法で他の agent との通信に使用するリンクが vであることを表す．agent が通信に使用するリンクには，主に c, a, m, v, w を使用している．プログラムで宣言されるクラスや AG，メソッドの数は有限であることから，それらに相当する agent には一意に定められたリンク c_i, a_j, m_k (i, j, k は自然数) をそれぞれ割り当てている．また，文や式を表す agent ではリンク v, wが使用されており，値を表す名前を通信する場合にはリンク vを，参照を表す名前を通信する場合はリンク wを使用する．

π計算で意味を表すときには，定数に相当する名前を使用する．非負整数を表す場合には名前 ONE, ZERO を，また真偽値を表す場合には TRUE, FALSE を使用する．オブジェクトへの参照を保持していることを表すために REF を，逆に保持していないことを表すために NIL を用いる．さらに，ディレクトリ表記で使用する “/”, “..”, “.” を表すための名前として ROOT, UP, SELF を，agent 間で同期をとるときに通信する名前として $done$ を用いる．

以下では，言語 $\mathcal{L}$ のπ計算への変換規則を定義する．まず，プログラムの宣言 $Pdec$ ，メイン関数 $Main$ は次のように表される．

$$
\begin{array}{l}
[\![\ Pdec\ ]\!](w) \\
\quad \stackrel{\text{def}}{=} (\ a_1, \dots,\ a_m,\ c_1, \dots,\ c_n\ ) \\
\qquad (\ [\![\ Main\ ]\!](w) \mid [\![\ Adec_1\ ]\!](a_1) \mid \dots \mid [\![\ Adec_m\ ]\!](a_m) \mid \\
\qquad\qquad [\![\ Cdec_1\ ]\!](c_1) \mid \dots \mid [\![\ Cdec_n\ ]\!](c_n)\ ) \\
[\![\ Main\ ]\!](w) \stackrel{\text{def}}{=} (N)(Loc_X \mid [\![\ Vdec\ ]\!] \mid \\
\qquad\qquad a_1(u).\ \overline{u}u.\ \overline{u}u.\ u(u').\ \overline{w_X}\mathsf{REF}.\ \overline{w_X}u.\ Done \\
\qquad\qquad before\ [\![\ S\ ]\!](w)) \\
\quad N = \{r_X, w_X\} \cup \{r_Y, w_Y \mid Y\text{は}\ Vdec\ \text{で宣言された変数}\}
\end{array}
$$

$$
\begin{array}{l}
Done \stackrel{\text{def}}{=} \overline{done}\ done.\ 0 \\
P\ before\ Q \stackrel{\text{def}}{=} (done)(P \mid done(z).\ Q) \\
\qquad (z\text{は agent}\ Q\ \text{中で使用されていない名前とする})
\end{array}
$$

agent $[\![\ Pdec\ ]\!]$ は，クラスと AG，メイン関数を表す agent を並列に動作させたものとして表現される．リンク $a_i (i = 1 \dots m)$, $c_j (j = 1 \dots n)$ はすべてのクラスや AG に対して一意に割り当てられており，このリンクを通じてそれぞれの agent と通信することができる．

agent $[\![\ Main\ ]\!]$ は，ルート AGへのリンクを保持するための変数に相当する agent Loc_X を持ち，$a_1(u).\ \overline{u}u.\ \overline{u}u.\ u(u')$ の部分で部品階層が生成されたとき，ルート AG へのリンクが保持される．その後，agent $[\![\ S\ ]\!]$ で部品階層が初期化され，計算が開始される．以下では，agent を逐次に動作させるために，$Done, before$ を使用する．

クラスの定義 $Cdec$，オブジェクトを生成する **new** 文は，次のように表される．

$$
\begin{aligned}
&\llbracket Cdec \rrbracket(\mathrm{c}) \stackrel{\mathrm{def}}{=} \overline{\mathrm{c}}(x) * C(x)\\
&\quad C(x) \stackrel{\mathrm{def}}{=} (N)(Loc_X \mid \llbracket Vdec \rrbracket \mid\\
&\qquad\qquad \llbracket Mlist \rrbracket(x) \mid \llbracket Mdec \rrbracket)\\
&\quad N = \{r_X, w_X\} \cup \{r_Y, w_Y \mid Y \text{は} Vdec \text{で宣言された変数}\}\\
&\llbracket [\, C\ \mathbf{new}\,] \rrbracket(w) \stackrel{\mathrm{def}}{=} c(z).\ \overline{w}\mathsf{REF}.\ \overline{w}z.\ Done
\end{aligned}
$$

クラス定義 $Cdec$ は，オブジェクトに相当する agent $C(x)$ を再帰的に生成する agent として表現される．このオブジェクトとの通信に使用されるリンク x には，他の agent と共有されていない，局所的な名前が割り当てられる．

オブジェクトを表す $C(x)$ は，変数の集合を表す agent $\llbracket Vdec \rrbracket$，メソッドの集合を表す $\llbracket Mdec \rrbracket$，AGO へのリンクを保持する変数の agent Loc_X，Loc_X の初期化やメソッドの選択・実行を行なう $\llbracket Mlist \rrbracket$ を並列に動作させたものとして表される．

クラスの生成 $\llbracket [C\ \mathbf{new}] \rrbracket$ は，クラスを表す agent から新しいオブジェクトへのリンクを受信し，参照を表す定数 REF と共に，そのリンクを w に送信する．

変数宣言 $Vdec$ と変数は次のように表される．

$$
\begin{aligned}
&\llbracket Vdec \rrbracket \stackrel{\mathrm{def}}{=} \llbracket t_1\ X_1;\ \ldots\ t_n\ X_n; \rrbracket\\
&\qquad\quad \stackrel{\mathrm{def}}{=} \llbracket t_1\ X_1 \rrbracket \mid \ldots \mid \llbracket t_n\ X_n \rrbracket\\
&\llbracket t\ X \rrbracket \stackrel{\mathrm{def}}{=} Loc_X(y) \qquad\qquad (t = \mathbf{int}, \mathbf{bool})\\
&\quad Loc_X \stackrel{\mathrm{def}}{=} w_X(z).\ Reg_X(z)\\
&\quad Reg_X(y) \stackrel{\mathrm{def}}{=} \overline{r_X}y.Reg_X(y) + w_X(z).Reg_X(z)\\
&\llbracket t\ X \rrbracket \stackrel{\mathrm{def}}{=} Loc_X(y) \qquad\qquad (t = \mathbf{id})\\
&\quad Loc_X \stackrel{\mathrm{def}}{=} \overline{r_X}\mathsf{NIL}.\ Loc_X\\
&\qquad\quad + w_X : [\ \mathsf{NIL} \Rightarrow Loc_X,\\
&\qquad\qquad\quad \mathsf{REF} \Rightarrow w_X(y).\ Reg_X(y)]\\
&\quad Reg_X(y) \stackrel{\mathrm{def}}{=} \overline{r_X}\mathsf{REF}.\ \overline{r_X}y.\ Reg_X(y)\\
&\qquad\quad + w_X : [\ \mathsf{NIL} \Rightarrow Loc_X,\\
&\qquad\qquad\quad \mathsf{REF} \Rightarrow w_X(z).\ Reg_X(z)]
\end{aligned}
$$

変数の表現では，変数を表す agent X それぞれに対して r_X, w_X というリンクを使用する．この agent から読み出しをするときにはリンク r_X を，書き込むときには w_X をそれぞれ使用する．変数が int, bool 型の値を保持する場合には，値を表す agent へのリンクが保持される．また，id 型という参照を保持する変数の場合には，NIL または，REF とリンクが保持される．

次に，メソッド定義 $Mdec, Mlist$ とメッセージ送信文は次のように表される．

$$
\begin{aligned}
&\llbracket Mdec \rrbracket(m_1, \ldots, m_l)\\
&\quad \stackrel{\mathrm{def}}{=} \llbracket M_1(X_1, Y_1)\{S_1\} \rrbracket(m_1) \mid\\
&\qquad\qquad \ldots \mid \llbracket M_l(X_l, Y_l)\{S_l\} \rrbracket(m_l)\\
&\llbracket M(X,Y)\{S\} \rrbracket(m) \stackrel{\mathrm{def}}{=} \overline{m}(z) * M(z)\\
&\quad M(z) \stackrel{\mathrm{def}}{=} (N)(Loc_X \mid Loc_Y \mid\\
&\qquad z(w).\ z(x).\ \overline{w}_X x.\ (\llbracket S \rrbracket(w)\ before\ r_Y(v').\ \overline{z}v'))\\
&\quad N = \{r_X, w_X, r_Y, w_Y\}\\
&\llbracket Mlist \rrbracket(w) \stackrel{\mathrm{def}}{=} \llbracket M_1, \ldots, M_k \rrbracket(w)\\
&\qquad\quad \stackrel{\mathrm{def}}{=} w(u).\ \overline{w_X}u.\ Done\ before\ w(v) * M(v)\\
&\quad M(v) \stackrel{\mathrm{def}}{=} v : [m_i \Rightarrow v(v').\ m_i(z).\ \overline{z}w.\ \overline{z}v'.\\
&\qquad\qquad z(v'').\ \overline{v}v''.\ Done]\\
&\qquad\qquad\qquad (i = 1, \ldots, k)\\
&\llbracket [\, E_1\ M : E2\,] \rrbracket(v)\\
&\quad \stackrel{\mathrm{def}}{=} (v_3)\ ((v_1)(\llbracket E_1 \rrbracket(v_1) \mid\\
&\qquad v_1(u).\ [u = \mathsf{REF}]v_1(w).\ Done)\\
&\qquad before\ ((v_2)(\llbracket E_2 \rrbracket(v_2) \mid Eval(v_2, v_3))\\
&\qquad before\ (u)\overline{w}u.\ \overline{u}m.\ \overline{u}v_3.\ u(v').\ v'(v'').\ Copy(v'', v)))
\end{aligned}
$$

$M(z)$ は，引数，返り値を保持するための変数の agent Loc_X，Loc_Y を持つ．メソッドが実行されると，引数が Loc_X に代入された後，メソッドの本体 S が実行され，最後に Loc_Y に保持されているリンクを返り値として返すという動作を行なう．

agent $\llbracket Mlist \rrbracket$ では，まず AGO へのリンクを受信し，Loc_X にそのリンクを保持する．その後はリンク w からメッセージセレクタ m_i と引数を受信して，該当するメソッドの agent を実行させる動作を繰り返す．

メッセージの送信 $\llbracket [E_1\ M : E_2] \rrbracket$ では，まずレシーバを表す式 E_1 が評価され，オブジェクトへの参照であることが確かめられる．その後，引数を表す式 E_2 が評価され，レシーバにメッセージセレクタを表す名前 m_i と引数を表す agent へのリンクが送信されて，メソッドが実行される．最後にメソッドからの返り値を表す agent へのリンクを受信し，このリンクを返す動作をする．

AG の定義 $Adec$ とディレクトリ表記 D は，次のように表される．

$$
\begin{aligned}
&\llbracket Adec \rrbracket(a) \stackrel{\mathrm{def}}{=} \overline{a}(x) * A(x)\\
&\quad A(x) \stackrel{\mathrm{def}}{=} (N)(\ \underline{x(lr)}.\ \underline{x(lp)}.\\
&\qquad a_1(la_1).\ \overline{la_1}lr.\ \overline{la_1}x.\ la_1(la'_1) \ldots\\
&\qquad a_j(la_j).\ \overline{la_j}lr.\ \overline{la_j}x.\ la_j(la'_j)\\
&\qquad c(lo)\ .\ \overline{lo}x.\ \overline{x}done.\ Done\ before\ x(u) * B(x, u)\\
&\quad N = \{lr, lp, lo, la_1, \ldots, la_j\}
\end{aligned}
$$

$$
B(x,u) \stackrel{\mathrm{def}}{=} u : [\
\begin{array}{lll}
\mathsf{ROOT} & \Rightarrow & \overline{x}lr,\\
\mathsf{UP} & \Rightarrow & \overline{x}lp,\\
\mathsf{SELF} & \Rightarrow & \overline{x}x,\\
\mathsf{O} & \Rightarrow & \overline{x}lo,\\
\mathsf{A}_i & \Rightarrow & \overline{x}la_i]
\end{array}
\qquad (i = 1, \ldots, j)
$$

$$
\begin{aligned}
&\llbracket D \rrbracket(v) &&\stackrel{\mathrm{def}}{=} r_X(v_1).\ \llbracket D \rrbracket(v_1, v)\\
&\llbracket /D \rrbracket(v_i, v) &&\stackrel{\mathrm{def}}{=} \overline{v_i}\mathsf{ROOT}.\ v_i(v_{i+1}).\ \llbracket D \rrbracket(v_{i+1}, v)\\
&\llbracket ./D \rrbracket(v_i, v) &&\stackrel{\mathrm{def}}{=} \overline{v_i}\mathsf{SELF}.\ v_i(v_{i+1}).\ \llbracket D \rrbracket(v_{i+1}, v)\\
&\llbracket ../D \rrbracket(v_i, v) &&\stackrel{\mathrm{def}}{=} \overline{v_i}\mathsf{UP}.\ v_i(v_{i+1}).\ \llbracket D \rrbracket(v_{i+1}, v)\\
&\llbracket A/D \rrbracket(v_i, v) &&\stackrel{\mathrm{def}}{=} \overline{v_i}A.\ v_i(v_{i+1}).\ \llbracket D \rrbracket(v_{i+1}, v)\\
&\llbracket O \rrbracket(v_i, v) &&\stackrel{\mathrm{def}}{=} \overline{v_i}O.\ v_i(v_{i+1}).\ \overline{v}\mathsf{REF}.\ \overline{v}v_{i+1}.\ Done
\end{aligned}
$$

agent $\llbracket Adec \rrbracket$ では，AG を表す $A(x)$ が再帰的に生成される．$A(w)$ では，ルート AG，スーパー AG，サブ AG，部品オブジェクトへのリンクが，名前 $lr, lp, la_i, lo (i = 1 \ldots k)$ にそれぞれ置き換えられる．その後，ディレクトリ表記に対応する名前に応じてこれらのリンクを返す動作を行う agent $B(w, u)$ として，再帰的に動作する．

ディレクトリ表記は，“/” 毎に分割し，順に AGO を表す agent に問い合わせていく形式で解析される．例えば，相対表記 “./a/o” の場合は，まず “.” が SELF に変換され，AGO から SELF に相当する agent1 へのリンクを受信する．次に agent1 から名前 “a” に相当する agent2 へのリンクを受信する．最後に agent2 から名前 “o” に相当する agent を受信し，この agent へのリンクを解析結果としてリンク v へ出力する．絶対表記の場合，最初の “/” はルートを表すので，このときだけは名前 ROOT に変換される．

4 トランスレータの検証

本節では，部品階層に基づくプログラムのためのトランスレータの検証を行なう．まず，検証方法を説明し，続いてメッセージ配送機構の相対・絶対表記を用いたメッセージ送信についてトランスレータの変換の正当性を示す．

4.1 検証の方法

部品階層を利用したプログラムは，トランスレータを使用することにより，通常の Objective-C 言語のプログラムに変換される．変換されたプログラムはメッセージ配送機構を実現するクラスを含んでいるため，変換される前のプログラムと単純には比較できない．そこで，変換される前のプログラ

ムと変換後のプログラムのそれぞれを，前節で定義した変換規則に基づいてπ計算のコードに変換する．そして，それらのコードを実際に計算し，計算が進む過程を観測することにより，メッセージ配送機構を用いたメッセージ送信の動作を比較することができる．トランスレータにより変換される前後のプログラムがメッセージの送信に関して等価であれば，トランスレータが正しく変換していると結論する．

ここで，検証の対象となるトランスレータについて，次のことを仮定する．

仮定 1 宣言等の構文が正しく解析され，また識別子に依存した誤りは起こらないものとする．

仮定 2 ディレクトリ表記を解析するとき，任意の "/" で分割し，"/" の前後の表記に分けて解析することができるものとする．ただし，絶対表記を先頭の "/" で分割する場合には，"/" と残りの表記に分割する．

例えば，相対表記 "$../A/B/b$" は表記 "$..$" と "$A/B/b$" に分割できる．そして，表記 "$..$" で AGO が正しく指定され，かつその AGO が表記 "$A/B/b$" に相当する部品オブジェクトを正しく指定できるとき，これらを "/" で結合した表記 "$../A/B/b$" は正しくオブジェクトを指定できるものとする．また，絶対表記 "$/A/B/b$" の場合は，"/" と "$A/B/b$" に分解でき，ルート AGO が表記 "$A/B/b$" に相当するオブジェクトを正しく指定できるとき，この絶対表記も正しく指定できるものとする．

トランスレータは部品階層の定義を解析し，その情報を木の形で保持している．この木の探索をすることにより相対・絶対表記の解析を行なうので，あるディレクトリ表記によりオブジェクトが正しく指定されるときには，表記を解析するために探索した部分が正しく構成されていると言える．したがって，仮定 2のようにディレクトリ表記を分割して解析した結果，それぞれが正しい表記であれば，元の表記も正しい表記となることが仮定できる．また，本節では，相対・絶対表記によるオブジェクトの指定を用いたメッセージ送信をトランスレータが正しく解析・変換していることを調べるので，仮定 1に示すようにトランスレータが構文や識別子に対して誤りを起こさないことを仮定する．

π計算に変換されたプログラムの計算の過程を観測するために，前節で定義した変換規則を一部変更する．まず，次のようなπ計算のコードを追加する．

$$GetInfo(\mathsf{OUT}) \stackrel{\mathrm{def}}{=} \mathsf{OUT}(v).\ GetInfo(\mathsf{OUT})$$

この agent をプログラムの agent と並行に動作させ，リンク OUT を通して受信される名前を等価性の判定に使用する．また，このリンク OUT に対してメッセージ送信先のオブジェクトに相当する agent の名前とメッセージ名が送信されるように，メッセージ配送機構を用いたメッセージ式の変換規則を改める．さらに，生成されるオブジェクトに対して，一意に定められた名前が割り当てられるように，クラス定義の変換規則も次のように改める．

$$\begin{aligned}
&[\![\,[\,D\ M : E\,]\,]\!](v) \\
&\quad \stackrel{\mathrm{def}}{=} (v_3)\ ((v_1)([\![\,D\,]\!](v_1)\ | \\
&\qquad v_1(u).\ [u = \mathsf{REF}]v_1(w).\ Done) \\
&\qquad before\ \overline{\mathsf{OUT}}w.\ \overline{\mathsf{OUT}}m.\ Done \\
&\qquad before\ ((v_2)([\![\,E\,]\!](v_2)\ |\ Eval(v_2, v_3)) \\
&\qquad before\ (u)\overline{w}u.\ \overline{u}m.\ \overline{u}v_3.\ u(v').\ v'(v'').\ Copy(v'', v)))
\end{aligned}$$

$$[\![\,Cdec\,]\!](c) \stackrel{\mathrm{def}}{=} \overline{c}(\mathsf{C}_1).\ (C(\mathsf{C}_1)\ |\ \overline{c}(\mathsf{C}_2).\ (C(\mathsf{C}_2)\ |\ \ldots\ )$$

ここで，クラス定義の変換規則の変更は再帰的に新しい名前が必要とされる形式に記述されているが，プログラム中で使用されるオブジェクトの数は有限であるので，必要なだけの名前を割り当てることができるものと考える．トランスレータにより変換されたプログラムをπ計算で表したコードに対しても同様にして，メッセージ送信先に指定されたオブジェクト名とメッセージ名をリンク OUT に対して出力するようにコードを挿入する．

次に，双模倣関係を定めて，π計算上でのプロセスの等価を定義する．

定義 1 P, P' を $agent$, μ を名前とし，どんな遷移もできないような終了状態に相当する $agent$ を P_{End} と表す．

$$\begin{aligned}
&(1)\quad P \stackrel{\mu}{\Longrightarrow} P' \text{ とは } P\ (\stackrel{\tau}{\longrightarrow})^{*} \stackrel{\overline{\mathsf{OUT}}\mu}{\longrightarrow} P' \\
&(2)\quad P \stackrel{\tau}{\Longrightarrow} P' \text{ とは } P\ (\stackrel{\tau}{\longrightarrow})^{+} P' \quad \text{ただし}, P' = P_{End}
\end{aligned}$$

定義 2 二つの $agent$ P, Q 上の関係 R がメッセージ送信に関して双模倣関係であるとは，次の 2つの条件が成立することである．ここで，$\alpha = \{\mu, \tau\}$ である．

$$\begin{aligned}
&(1)\quad P \stackrel{\alpha}{\Longrightarrow} P' \text{ のとき，}\quad Q \stackrel{\alpha}{\Longrightarrow} Q' \text{ かつ } (P', Q') \in R \\
&(2)\quad Q \stackrel{\alpha}{\Longrightarrow} Q' \text{ のとき，}\quad P \stackrel{\alpha}{\Longrightarrow} P' \text{ かつ } (P', Q') \in R
\end{aligned}$$

このとき，双模倣関係 R の最小の集合を等価関係 $\approx$ とする．

この等価関係を用いて，プログラムの等価性を調べることができる．部品階層を用いたプログラムを S を用意し，これをそのままπ計算に変換したものを P，トランスレータで Objective-C 言語に変換した後，π計算に変換したものを Q とする．このとき，P, Q を実際に計算し，リンク OUT を通して観測される名前の系列が同じであれば，等価関係が成立する．

$$\begin{gathered}
P \stackrel{\alpha_1}{\Longrightarrow} P_1 \stackrel{\alpha_2}{\Longrightarrow} P_2\ \ldots\ \stackrel{\alpha_n}{\Longrightarrow} P_n \stackrel{\tau}{\Longrightarrow} P_{End} \\
Q \stackrel{\alpha_1}{\Longrightarrow} Q_1 \stackrel{\alpha_2}{\Longrightarrow} Q_2\ \ldots\ \stackrel{\alpha_n}{\Longrightarrow} Q_n \stackrel{\tau}{\Longrightarrow} Q_{End} \\
\text{このとき，} P \approx Q \text{ が成立する．}
\end{gathered}$$

以下では，トランスレータで変換される前後のプログラムそれぞれをπ計算に変換したものを記号 P, Q で表す．

4.2 相対・絶対表記によるオブジェクトの指定

ここでは，部品階層のメッセージ配送機構で提供されている，相対・絶対表記によるオブジェクトの指定のみを用いてメッセージ送信を行なうプログラムが，トランスレータにより正しく変換されることを示す．

トランスレータによって変換された後のプログラムでメッセージ送信が正しく動作するためには，相対・絶対表記が正しく解析され，メッセージ送信先のオブジェクトが部品階層中の特定のオブジェクトとして正しく定められてなければならない．したがって，ここでは，トランスレータがメッセージ送信を行なうプログラムを正しく変換することを示すために，変換後のプログラムにおいて相対・絶対表記が部品階層中のオブジェクトを正しく指定できていることを調べる．

以下では，部品階層の構造を表現するために高さという表現を使用し，n 段の入れ子に定義された部品階層のことを高さ n の部品階層と呼ぶ．

まず，次のような AG の再利用を行なわない部品階層 1 の定義を考える．

(部品階層 1)

```
AG Aag has
        a : CLASS Acls;
        B : AG Bag has
                b : CLASS Bcls
        end;
        C : AG Cag has
                c : CLASS Ccls
        end
end
```

部品階層中で使用されているクラス“Acls”には次のようなプログラムを用いる．また，クラス“Bcls”, “Ccls”は，メソッド“begin”を持たないことを除くと，クラス“Acls”と同じ定義とする．

// Acls.h の定義	// Acls.m の定義

#import ⟨ objc/Object.h ⟩
@interface Acls : Object
{ }
- begin;
- test;
@end
	@implementation Acls
	- begin {
	[/a test] ;
	return self;
	}
	- test { return self; }
	@end

このプログラムとトランスレート後のプログラムをπ計算に変換し，リンク OUT を通じて計算の過程を観測すると次のように遷移する．

$$P \overset{\text{Acls}_1}{\Longrightarrow} P_1 \overset{\text{test}}{\Longrightarrow} P_2 \overset{\tau}{\Longrightarrow} P_{End}$$
$$Q \overset{\text{Acls}_1}{\Longrightarrow} Q_1 \overset{\text{test}}{\Longrightarrow} Q_2 \overset{\tau}{\Longrightarrow} Q_{End}$$

したがって，$P \approx Q$ が成り立ち，これらのプログラムはメッセージ送信について等価である．すなわち，絶対表記“/a”により正しくメッセージ送信先のオブジェクトが指定されている．

クラス“Acls”のメッセージ配送機構を用いたメッセージ送信の部分を変更し，部品階層中の各部品オブジェクトにメッセージ送信を行なうプログラムについて，同様の手順でその等価性を調べる．その結果，次の表に示す相対・絶対表記を含むプログラムはすべて等価となる．

	絶対指定	相対指定
a	[/a test] [/B/b test] [/C/c test]	[./a test] [./B/b test] [./C/c test]

さらに，クラス“Bcls”, “Ccls”のプログラムを変更し，次の表に示すオブジェクトの指定を用いたメッセージ送信を行なうプログラムについて，その等価性を調べる．その結果，それらのプログラムはどれもトランスレータにより変換されたものと等価となる．

	絶対指定	相対指定
b	[/a test] [/B/b test] [/C/c test]	[../a test] [./b test] [../C/c test]
c	[/a test] [/B/b test] [/C/c test]	[../a test] [../B/b test] [./c test]

以上の結果から，この部品階層で定義されている部品オブジェクトの間では，相対・絶対表記に対してオブジェクトが正しく指定されていることが結論できる．

補題 1 *AG* の再利用が行なわれていない，高さ *1* の *2* 分木を構成する部品階層では，部品オブジェクト間で，相対・絶対表記を用いてオブジェクトを正しく指定できる．

次に，部品階層 1 の定義をそのままサブ AG として使用している高さ 2 の部品階層を考える．

(部品階層 2)

```
AG Gag has                        D : AG Dag has
   g : CLASS Gcls;                    d : CLASS Dcls;
   A : AG Aag has                     E : AG Eag has
      a : CLASS Acls;                    e : CLASS Ecls
      B : AG Bag has                  end;
         b : CLASS Bcls               F : AG Fag has
      end;                               f : CLASS Fcls
      C : AG Cag has                  end
         c : CLASS Ccls            end
      end;                        end
```

この部品階層中で，相対・絶対表記を用いて正しくオブジェクトの指定が行なえることを確かめた．つまり，任意の二つの部品オブジェクト間でオブジェクトの指定を用いたメッセージ送信を行なうプログラムをそれぞれ用意し，それらすべてについてトランスレータによって変換される前後でプログラムが等価であることを確かめた．ただし，ここで調べた表記は，あるオブジェクトの AG から送信先の部品オブジェクトまでの最短の経路を指定したものである．例えば，オブジェクト“b”から“c”を指定する場合には，相対表記で“../C/c”を，また絶対表記では“/A/C/c”を用いている．以下では，このような表記を用いてオブジェクトを指定することを最短パスによる指定と呼ぶ．

この結果から，この部品階層内のすべての部品オブジェクトから他の部品オブジェクトを正しく指定できる．よって，次の補題が得られる．

補題 2 *AG* の再利用を行なっていない場合，高さ *2* の *2* 分木を構成する部品階層においては，相対・絶対表記によりオブジェクトが正しく指定される．

補題 1, 2 より，絶対表記を解釈する際に，階層中のどの部品オブジェクトからもルート AG が正しく指定できることが結論できる．

補題 3 部品階層中の任意の部品オブジェクトからルート *AG* が正しく指定できる．

部品階層 2 では，部品階層 1 で定義されている“Aag”がサブ AG として使用されている．したがって，“Aag”に含まれる部品オブジェクト“a”, “b”, “c”の間での相対表記によるオブジェクトの指定は，部品階層 1 でオブジェクトの指定に使用したものと同一である．このことから，高さ 1 の 2 分木を成す AG をサブ AG として使用しても，スーパー AG の存在によらずに，相対表記は正しく行なわれることが結論できる．したがって，次の補題が得られる．

補題 4 *AG*の再利用を行なっていない部品階層において，高さ *1* の *2*分木を成す *AG*をサブ *AG*として使用する場合，サブ *AG*内の部品オブジェクト間では相対表記を用いて正しくオブジェクトが指定できる．

ここで，高さ k ($k \geq 2$) の 2 分木を成す部品階層に対してトランスレータが正しく変換を行ない，相対・絶対表記を用いて正しく部品オブジェクトが指定できることを仮定する．高さ k+1 の 2 分木を成す部品階層は，高さ k の位置にある AG に，サブ AG として部品オブジェクトのみを持つ AG をいくつか追加して定義される．ここで，追加されたサブ AG のうちの任意の二つの AGO を S, T，属するオブジェクトを s, t，また，そのスーパー AGO を $\tilde{S}, \tilde{T}$，ルート AGO を R とする．さらに，高さ $i (1 \leq i \leq k)$ の位置にある任意の部品オブジェクトを u，その AGO を Uとする．このとき，オブジェクト s から t, u を指定する場合を考える．s からこれらのオブジェクトが正しく指定されるためには，次の表に示す表記により，t, u が指定されなくてはならない．ここで，オブジェクト s がオブジェクト t を指定することを $s \rightarrow t$ と表す．また，最短のパスによる指定は，ディレクトリ表記の部分を省略して表現しており，省略部分は記号δ_iで表す．

指定	表記法	最短パスによる指定	
$s \rightarrow u$	相対表記	$../\delta_1/U/u$	(1)
	絶対表記	$/\delta_2/U/u$	(2)
$s \rightarrow t$	相対表記	$../\delta_3/\tilde{T}/T/t$	(3)
	絶対表記	$/\delta_4/\tilde{T}/T/t$	(4)

まず，表記 (1) は "..", "$\delta_1/U/u$" と分割でき，

- 補題4より，表記 ".." で $s \rightarrow \tilde{S}$ が成り立つ．
- 高さ k の部品階層で指定が正しく行なわれることから，表記 "$\delta_1/U/u$" で $\tilde{S} \rightarrow u$ が成り立つ．

よって，表記を分割して解析できるというトランスレータについての仮定から，表記 (1) で $s \rightarrow u$ が成り立つ．

表記 (2) から (4) についても，同様に表記を分割することにより，次のように正しくオブジェクトが指定される．

表記	分割された表記	成立する指定	帰結
(2)	"/" "$\delta_2/U/u$"	$s \rightarrow R$ (a より) $R \rightarrow u$ (b より)	$s \rightarrow u$
(3)	".." "$\delta_3/\tilde{T}$" "T/t"	$s \rightarrow \tilde{S}$ (c より) $\tilde{S} \rightarrow \tilde{T}$ (b より) $\tilde{T} \rightarrow t$ (c より)	$s \rightarrow t$
(4)	"/" "$\delta_4/\tilde{T}$" "T/t"	$s \rightarrow R$ (a より) $R \rightarrow \tilde{T}$ (b より) $\tilde{T} \rightarrow t$ (c より)	$s \rightarrow t$

a. 補題 3

b. 高さ k の部品階層についての仮定

c. 補題 4

以上のことから，(1) から (4) の表記はすべて正しくオブジェクトを指定できる．したがって，高さ k についての帰納法から次の定理が成り立つ．

定理 1 *AG* の再利用を行なわない高さ *n* の *2* 分木を成す部品階層において，相対・絶対表記を用いて階層中の任意の部品オブジェクトが正しく指定される．

次に再利用を行なった部品階層を考える．2.2節で述べたように，部品階層の定義では AG を再利用することができる．トランスレータは AG の定義が行なわれる毎に AG 名と AG 定義の対応を保持し，AG が再利用されている場合には，その AG 名にしたがって既出の AG 定義に展開するという方法を用いて，部品階層の定義を解析している．したがって，AG の再利用を行なった部品階層 A とあらかじめ AG の再利用を元の AG 定義に置き換えておいた部品階層 B をトランスレータで変換し，変換結果を比較することで，この方法の正しさを調べることができる．

次に示す，部品オブジェクトのみを持つ AG の再利用を行なった高さ 1 の部品階層 A と再利用されている AG を展開した部品階層 B を用意する．これらの部品階層において，相対・絶対表記を用いたメッセージ送信を行なうプログラムをトランスレータで変換すると，変換後のプログラムは AGO 名を除いて等しいものとなる．

(部品階層 A)	(部品階層 B)
AG Aag **has** a : **CLASS** Acls; B : **AG** Bag **has** b : **CLASS** Bcls **end**; C : **AG** Bag **end**	**AG** Aag **has** a : **CLASS** Acls; B : **AG** Bag **has** b : **CLASS** Bcls **end**; C : **AG** Cag **has** b : **CLASS** Bcls **end** **end**

高さ 1 の AG を再利用をした部品階層についても変換されたプログラムは等しくなる．また，部品階層 A の AG "Aag" のように，すでに AG の再利用を行なっている AG をさらに再利用している，2 段階に再利用を行なう部品階層に対しても同様の結果が得られる．定理 1 より，AG を展開して再利用をなくした部品階層は正しく変換されることから，トランスレータは再利用された AG を正しく展開し解析していることが結論できる．

補題 5 *AG* の再利用を行なった部品階層においては，再利用された *AG* は既出の *AG* 定義に展開され，相対・絶対表記によりオブジェクトが正しく指定される．

以上のことから，部品の再利用を行なったプログラムにおいても，相対・絶対表記を用いて正しくオブジェクトを指定できる．したがって次の定理が得られる．

定理 2 *AG* の再利用を行った高さ *n* の *2* 分木を成す部品階層において，相対・絶対表記によるオブジェクトの指定が正しく行なわれる．

この定理から，相対・絶対表記によるオブジェクトの指定についてトランスレータの変換の正当性が示された．この定理 2 は 2 分木の部品階層について確かめたものであるが，補題を示す段階で n 個の AG を持つ部品階層を調べることにより，より一般的な n 分木の部品階層について確かめることが可能である．

5 おわりに

本研究では，部品階層に基づくオブジェクト指向プログラミング言語の意味を π 計算で表した．さらに，この π 計算への変換規則を用いて部品階層を用いたプログラムを記述し，相対・絶対表記によるオブジェクトの指定を用いたメッセージ配送について，部品階層に基づくプログラミングのためのトランスレータを検証した．これにより，同プログラミング言語処理系の信頼性を高めることができた．

今後の課題として，メッセージ配送機構のその他の配送方法である上昇伝播や放送について，また，静的な部品階層を動的に生成・消滅させて構造が変化する部品階層についてもトランスレータを検証することが挙げられる．

謝辞

日頃ご討論いただく首藤研究室の皆様に感謝いたします．

参考文献

[1] Cox,B.J. "Object Oriented Programming : An Evolutionary Approach". Addison-Wesley Publishing Company, Reading, 1886.

[2] Parrow,J. Milner,R. and Walker,D. "A Calculus of Mobile Processes, Part I and II". *Information and Computation*, Vol. 100, pp.1-pp.77, 1992.

[3] Walker,D. "π-Calculus Semantics of Object-Oriented Programming Languages". *Theoretical Aspects of Computer Software*, pp.532-pp.546, September 1991.

[4] 市川, 魚井, 萩原, 首藤. "オブジェクト指向言語における部品階層の記述の問題点とその改良について". 情報処理学会研究報告（記号処理・プログラミング合同研究報告）, 91-SYM-60-91-PRG-2-5, June 1991.

[5] 市川, 魚井, 野島, 萩原, 首藤. "部品階層に基づくオブジェクト指向プログラミングのための開発環境". オブジェクト指向コンピューティング '92, 田中・西尾編, 近代科学社, July 1993.

[6] 市川, 香川, 野島, 魚井, 萩原, 首藤. "部品階層に基づくオブジェクト指向プログラミング〜その評価とメッセージ配送機構の実装方法について〜". 第 *9* 回オブジェクト指向計算ワークショップ（*WOOC'93*）, March 1993.