ページ記述言語におけるプログラム変換

伊知地 宏† 窪寺隆行‡ 森田雅夫†

富士ゼロックス株式会社
†システム・コミュニケーション研究所
‡ドキュメントシステム開発センター
E-mail: hiro@ksp.fujixerox.co.jp

Abstract

文書構造を表すプログラミング言語の一つにページ記述言語がある．ページ記述言語は，それぞれ基本構造に類似性はあるものの詳細な部分で違いが多く，異なるページ記述言語を統一的に扱うことは難しかった．我々はページ記述言語 Interpress に操作的意味を与え，その意味にしたがって Interpress のプログラムを PostScript のプログラムに変換する方法を確立した．さらにその技法を用いて，Interpress から PostScript へ自動的にプログラム変換を行なうソフトウェア ip2ps を作成した．これにより，二つのページ記述言語を融合することが出来た．

Program Transformations in Page Description Languages

Hiroshi Ichiji† Takayuki Kubodera‡ Masao Morita†

†Systems & Communications Laboratory
‡Document Systems Development Center
Fuji Xerox, Co, Ltd.
E-mail: hiro@ksp.fujixerox.co.jp

Abstract

A page description langage is a programming languages for representations of document structures. Many page description langages have similar syntax, but their semantics are diffrent in many points. So, it was difficult to interpret two or more page description langages with same computational environments. This paper gives the operational semantics for Interpress and the techniques of transformation from Interpress to PostScript which based on this semantics. The ip2ps which is derived from these techniques is an automatic program transformation software from Interpress program to PostScript ones. And finally, Interpress and PostScript can be interpreted with same computational environments.

1 はじめに

ページ記述言語は, 印刷物を出力装置に依存することなく記述するためのプログラミング言語であり, 代表的なものとして Interpress[14] と PostScript[1][3] がある. ビットマップディスプレイとウインドウシステムの普及により, 最近ではページ記述言語がユーザインタフェースの道具として使われることも多くなってきた. しかし, 人が直接プログラミングする機会は少なく, 多くの場合にはアプリケーションソフトウェアがプログラムを生成する. 例えば, Interpress のプログラムを生成するソフトウェアとしては GloabalView(JStar) が, PostScript については Macintosh の各種ソフトウェアが知られている. この頃では, TEX[6] で書いた文書も最終的に PostScript に変換して出力することが多くなっている.

ところで, Interpress と PostScript は構造的には非常に似ているが, 意味的にはかなりの違いがあるため, 両言語の両立が難しかった. この問題に対して, 両言語の比較を行ない, プログラム変換の可能性を検討した. Interpress に操作的意味を明確に与え, その意味を PostScript のプログラムで記述し, これを用いて Interpress から PostScript へのプログラム変換ソフトウェア ip2ps[9] の作成を試みた.

2 ページ記述言語

2.1 ページ記述言語の構造

ページ記述言語は, 印刷物などのページのイメージを表現するためのプログラミング言語であり, そのプログラムは, 印刷イメージの動的なフォーマットであると言える.

ページ記述言語は, 基本的にデータと命令 (手続き) で構成されている. データはデータ型を持ち, 整数などの基本データ型と配列などの複合データ型がある. 命令には, 基本言語と呼ばれる汎用的なプログラミング言語が持つような命令と, イメージ命令とかグラフィック命令と呼ばれるページ記述言語の特徴である画像処理のための命令とが用意されている. 一般には逆ポーランド記法を採用したプログラミング言語である.

ページ記述言語のプログラミングスタイルは次のようなものである. まず最初に環境の設定や命令の定義などの初期化のプログラムを記述する. この初期化プログラムを Interpress ではプリアンブル [13], PostScript ではプロローグと呼ぶ. その後にページイメージを表現するプログラムが書かれる. この部分を Interpress ではページボディ, PostScript ではスクリプトと言う. ページ記述言語において基本的なプログラミングスタイルは共通である.

しかし, プログラムの意味論的にはページ記述言語毎にいろいろ異なっている. 一番大きな違いはページ独立と呼ばれる概念に起因する. ページ独立とは, 異なるページ間ではプログラムの実行による環境の変化の影響が及ばないことを意味している. つまり, ページ独立なページ記述言語では, 各ページ毎に完全に独立したプログラムを書くことが出来る. 一方, ページ独立でないページ記述言語では, 前のページの影響を気にしながらプログラムを書かないといけない. しかし, ページ独立でない言語では, 前のページの影響を受けないようにプログラミングが出来るようなメカニズムが用意されていることが多い.

2.2 Interpress

ここでは Interpress の言語仕様について簡単に述べる. なお言語仕様の詳細については文献 [14] を, 概念については文献 [13] の Interpress についての解説を, プログラミングについては文献 [7] を, Interpress の処理系の実現法については文献 [8] を参照願いたい.

Interpress の基本言語にはデータ型として, Number, Identifier, Mark, Vector, Body, Operator, Any がある. Number は有理数の型, Identifier は識別子の型, Mark はエラー検出用の特別の型, Vector は Any 型に属するデータを並べた列を表すデータ型, Body は"{"で始まり"}"で終るブロックを表す型, Operator は命令を表す型, そして Any は Body と Mark 以外の前記の型を包含する型である. また, Number の部分型として Cardinal という非負整数型もある.

Interpress は普通のプログラミング言語にあるような変数を持っていない. その代わりにフレームというデータの格納場所を持っており, その数は 50 個である. また, 印刷表示環境用の変数としてイメージ変数があり, 現在の位置 (X-Y 座標で表される) や線の太さなどの情報が保存される.

次に Interpress の基本言語が持つ命令について説明する. 基本言語の命令は, ベクトル命令, フレーム

命令, 新たな命令の作成と実行のための命令, スタック命令, 制御命令とテスト命令の 6 つに分けられる.

ベクトル命令は Vector 型に関係する命令である. フレーム命令はフレームに値を入れたり, フレームから値をとる命令で, 例えば FGET は, 与えられた整数 j を引数にしてフレームの j 番目の値 x を返す命令である. このような命令の動作を図式で表すと

$$\langle j : Cardinal\rangle \mathrm{FGET} \rightarrow \langle x : Any\rangle$$

となる.

制御命令はプログラムの制御を行なう命令で IF などがある. IF は i が 0 でないときに b を実行する.

$$\langle i : Cardinal\rangle\langle b : Body\rangle \mathrm{IF} \rightarrow \langle\rangle$$

画像処理のために Intepress では X-Y 直交座標系を想定している. そのもとでイメージ命令と呼ばれるページ記述言語独特の手続きが用意されている. イメージ命令は, 座標変換命令, 位置命令, ピクセル配列命令, カラー命令, マスク命令, 文字とフォントの命令とスペース調整命令の 7 つに分けられる.

座標変換命令には拡大縮小, 移動, 回転やそれらを組合せる命令があり, 文字とフォントの命令にはフォントの指定や大きさの指示の命令がある. ここではマスク命令のうち軌跡の指定の命令について述べる.

$$\langle x : Number\rangle\langle y : Number\rangle \mathrm{MOVETO} \rightarrow \langle t : Trajectory\rangle$$
$$\langle t_1 : Trajectory\rangle\langle x : Number\rangle\langle y : Number\rangle \mathrm{LINETO} \rightarrow \langle t_2 : Trajectory\rangle$$

軌跡は線分または曲線が端点でつながった 1 本の線である. MOVETO で始点を指定し LINETO で次々に接続点を指定することで軌跡を決定する. ベジェ曲線, 円錐曲線, 円弧を扱う命令もある. なお Trajectory は軌跡を表すデータ型である.

Interpress のプログラムは通常 Interpress マスタ (Interpress master) と呼ばれる. Interpress マスタは以下に示すように BEGIN で始まり END で終了する.

```
BEGIN
{..プリアンブル..}{..ページボディ 1}
{..ページボディ 2..}...{..ページボディ n..}
END
```

BEGIN と END の間は, プリアンブル, ページボディ 1, ページボディ 2,..., ページボディ n で構成されている. Interpress はページ独立な言語であり, 各ページボディはプリアンブル以外の他のページボディの影響を受けずページブロック単位で独立性を保つプログラムになっている. Interpress マスタの実例を以下に示す.

```
BEGIN
{ /* プリアンブル */
Identifier "Xerox" Identifier "XC1-1-1"
Identifier "Classic" 3 MAKEVEC FINDFONT
7620/18 -7620/18 SCALE2 MODIFYFONT
0 FSET /* フォントの生成 */
....(省略)....
Identifier "Xerox" Identifier "XC1-1-1"
Identifier "Modern" 3 MAKEVEC FINDFONT
7620/18 -7620/18 SCALE2 MODIFYFONT
4 FSET
}
{....(1 ページ目省略)....}
{
0.00001 SCALL CONCATT
0 29704 TRANSLATE CONCATT
1 -1 SCALL2 CONCATT
....
1 SETGRAY 4 SETFONT
3031 25011 SETXY
String "Interpress" SHOW
/* Interpress と印字される */
....
70 15 ISET 1 16 ISET
2540 24554 MOVETO 5150 24554 LINETO
MASKSTROKE        /* 線分の表示 */
....
70 15 ISET
6779 23725 MOVETO 6294 23853 LINETO
6482 23901 LINETO
6779 23725 MOVETO 6482 23901 LINETO
6434 24089 LINETO
2 MAKEOUTLINE MASKFILL
/* 閉領域の塗りつぶし */
....
}
.... (以下省略) ....
END
```

2.3 PostScript

PostScript は広く知れ渡っているのでここではあまり細かいことは述べない. PostScript の言語仕様の詳細については文献 [1][3] を, プログラミングについては文献 [2] を参照願いたい.

PostScript も型を持った言語であり, 汎用言語とグラフィックス命令で構成されていて, 変数を使えるようになっている. PostScript は基本的にはページ独立な言語でないが, いくつかの命令を使ってペー

ジ毎に独立させることも可能である.

PostScript の最大の特徴は, 辞書という構造を持つことである. 辞書にはユーザが定義した命令や変数などが置かれるが, さらに PostScript 言語自体の命令も辞書上に存在している. このため, PostScript システムが提供している命令を, ユーザがダイナミックに定義し直して PostScript の動作環境を変更することが出来る. ip2ps においてもそうであるが, この辞書の機能が PostScript によるプログラミングの可能性を大きくしている.

2.4 Interpress と PostScript の違い

Interpress と PostScript の大きな違いを以下に示す.

- 変数　Interpress には変数がなく, その代わりにフレームという Interpress のオブジェクトを 50 個置ける領域がある. PostScript では, 配列や辞書などでいくつでも変数を扱える.
- 線の描画　Interpress では, 軌跡 (trajectory) と呼ばれるパス (path) を作り, そのパスに対して線を引く. 軌跡は, Interpress のオブジェクトであり, フレームやスタック上に保存することが出来る. PostScript では, カレントパスに対して線を引く. カレントパスは, PostScript のオブジェクトでなく状態なので, スタックや変数で保存することが出来ない.
- 描画状態変数　Interpress にはイメージ変数, PostScript にはグラフィックス状態という描画のためのパラメータ領域がある. 両者のパラメータの意味は微妙に異なる.
- ページ独立性　Interpress はページ独立な構造を持つ. 一方, PostScript はページ独立な構造を持たない.
- 文字位置補正　Interpress には, 文字位置の補正を行なう CORRECT という命令がある. PostScript には, 文字位置の補正機能はない.
- アンダーライン　Interpress には, 文字にアンダーラインをつける機能があるが, PostScript にはそのような機能はない.

この違いが両言語を両立させるための問題となる. 他に細かい点ではまだまだいろいろな違いがあるのだが説明は省略する.

3 Interpress のモデル

3.1 文法

ここでは形式的に Interpress の文法を定義する. なお, 見やすくするために, 文法を実際よりは若干簡単にしてある.

$\langle Operator\rangle$ ::= ABS|...|UNMARK0
$\langle Alphabet\rangle$ ::= $A|...|Z|a|...|z$
$\langle Digit\rangle$ ::= 0|...|9
$\langle Sign\rangle$ ::= +|−
$\langle Letter\rangle$ ::= $\langle Digit\rangle|\langle Alphabet\rangle$
$\langle Cardinal\rangle$ ::= $\langle Digit\rangle^+$
$\langle Integer\rangle$ ::= $\langle Cardinal\rangle$
$|\langle Sign\rangle\langle Cardinal\rangle$
$\langle Rational\rangle$ ::= $\langle Digit\rangle^+/\langle Digit\rangle^+$
$|\langle Sign\rangle\langle Digit\rangle^+/\langle Digit\rangle^+$
$\langle Real\rangle$ ::= $\langle Digit\rangle^*.\langle Digit\rangle^*$
$|\langle Sign\rangle\langle Digit\rangle^*.\langle Digit\rangle^*$
$\langle Number\rangle$ ::= $\langle Integer\rangle|\langle Rational\rangle$
$|\langle Real\rangle$
$\langle Identifier\rangle$ ::= $\langle Alphabet\rangle$
$|\langle Alphabet\rangle\langle Letter\rangle$
$\langle Elements\rangle$ ::= $\langle Any\rangle|\langle Any\rangle,\langle Elements\rangle$
$\langle Vector\rangle$ ::= $[\langle Elements\rangle]$
$\langle Any\rangle$ ::= $\langle Number\rangle|\langle Identifier\rangle$
$|\langle Vector\rangle|\langle Operator\rangle$
$\langle Expression\rangle$ ::= $\langle Any\rangle$
$|\langle Any\rangle\langle Expression\rangle$
$|\langle Body\rangle\langle Expression\rangle$
$\langle Body\rangle$ ::= $\{\langle Expression\rangle\}$
$\langle Preamble\rangle$ ::= $\langle Body\rangle$
$\langle Pageblock\rangle$ ::= $\langle Body\rangle$
$\langle Pagebody\rangle$ ::= $\langle Pageblock\rangle^*$
$\langle Program\rangle$ ::= $BEGIN\langle Preamble\rangle$
$\langle Pagebody\rangle END$

$\langle Operator\rangle$, $\langle Number\rangle$, $\langle Identifier\rangle$, $\langle Vector\rangle$, $\langle Body\rangle$ なるデータは, それぞれ Operator, Number, Identifier, Vector, Body という型を持つ. さらに, $\langle Cardinal\rangle$ なるデータは Cardinal という型を持つ.

以下では, データ a が型 t を持つということを, $a:t$ で表すことにする.

3.2 操作的意味

本節では Interpress の操作的意味 [12] を記述する. 本稿では以下のように表すことにする.

$$\frac{\Sigma_1 ... \Sigma_n}{\{a_1 ... a_m \ op \ \Pi\}[F,I,G] \triangleright \{b_1 ... b_k \ \Pi\}[F',I',G']}$$

ここで, $\Sigma_1, ..., \Sigma_n$ を仮定 (条件) とし, $a_1, ..., a_m$ と $b_1, ..., b_k$ を Any 型または Body 型のデータ, op を Operator 型のデータ, Π をプログラムの残り, F, I, G (および F', I', G') をそれぞれフレーム, イメージ変数, 画像の状態, そして $\triangleright$ を操作による推移と解釈すると意味は明らかであろう. 以下では便宜上 Π の部分を省略して記述する.

また, F の n 番目の要素を $F(n)$ で表し, $F[x/j]$ を以下のように定義する.

$$F[x/j](n) = \begin{cases} x & \text{if } n = j \\ F(n) & \text{otherwize} \end{cases}$$

今, 簡単な例として命令 ADD の操作的意味を記述してみよう.

[ADD]

$$\frac{a, b, c : Number \quad c = a + b}{\{a \ b \ \text{ADD}\}[F,I,G] \triangleright \{c\}[F,I,G]}$$

上から分かるように, ADD はプログラムの実行において加算を行なうのみで, フレーム, イメージ変数や画像に影響を与えない.

ページ数に限りがあるので, 全ての命令の操作的意味を記述することができない. ここでは, 次の節で使う命令に限って操作的意味を記述することにする.

[FGET]

$$\frac{j : Cardinal \ \ x : Any \ \ x = F(j)}{\{j \ \text{FGET}\}[F,I,G] \triangleright \{x\}[F,I,G]}$$

[FSET]

$$\frac{j : Cardinal \ \ x : Any}{\{x \ j \ \text{FSET}\}[F,I,G] \triangleright \{\}[F[x/j],I,G]}$$

[IGET]

$$\frac{j : Cardinal \ \ x : Any \ \ x = I(j)}{\{j \ \text{IGET}\}[F,I,G] \triangleright \{x\}[F,I,G]}$$

[ISET]

$$\frac{j : Cardinal \ \ x : Any}{\{x \ j \ \text{ISET}\}[F,I,G] \triangleright \{\}[F,I[x/j],G]}$$

[MASKVECTOR]

$$\frac{x_1, x_2, y_1, y_2 : Number}{\{x_1 \ y_1 \ x_2 \ y_2 \ \text{MASKVECTOR}\}[F,I,G] \triangleright \{x_1 \ y_1 \ \text{MOVETO} \ x_2 \ y_2 \ \text{LINETO MASKSTROKE}\}[F,I,G]}$$

[MOVETO]

$$\frac{x, y : Number}{\{x \ y \ \text{MOVETO}\}[F,I,G] \triangleright \{< (\varphi(x,I), \varphi(y,I)) >\}[F,I,G]}$$

[LINETO]

$$\frac{< (\chi, \psi) > : Trajectory \ \ x, y : Number}{\{< (\chi \ \psi) > \ x \ y \ \text{LINETO}\}[F,I,G] \triangleright \{< (\chi, \psi)\{line\}(\varphi(x,I), \varphi(y,I)) >\}[F,I,G]}$$

[MASKSTROKE]

$$\frac{< (\chi_1, \psi_1)\{line\}(\chi_2, \psi_2) > : Trajectory}{\{< (\chi_1, \psi_1)\{line\}(\chi_2, \psi_2) > \ \text{MASKSTROKE}\}[F,I,G] \triangleright \{\}[F,I,G\$(line \ \chi_1 \ \psi_1 \ \chi_2 \ \psi_2 \ I)]}$$

ここで φ は座標変換の関数であり, $Trajectory$ は軌跡を表す型である. また $G\$H$ は, G への H の追加を意味するものとする.

4 PostScript による記述

4.1 MASKVECTOR

MASKVECTOR は線分を引く命令である. 以下のプログラムは (x_1, y_1) から (x_2, y_2) へ線分を引く:

$x_1 \ y_1 \ x_2 \ y_2$ MASKVECTOR

MASKVECTOR を操作的意味論に基づいて PostScript で記述するために, まず MOVETO, LINTO, MASKSTROKE を検討しよう.

MOVETO を以下のように定義する.

```
/moveto load null 4 1 roll 4 array astore
dup 2 array astore
```

上のプログラムは, 3.2節で与えた*Trajectory*型のデータ$<(\varphi(x,I),\varphi(y,I))>$を作るプログラムである.

またLINETOは次のようになる.

```
/lineto load null 4 1 roll 4 array astore
2 copy exch 1 get exch 0 exch put 1 index
exch 1 exch put
```

このプログラムは技巧的なので細かいことは述べないが, スタック上にある*Trajectory*型データを使って, 新しい同じ型のデータを作っており, 3.2節で与えた操作的意味を実現している.

MASKSTROKEの実現は,

```
gsave newpath 0 get {
cvx exec dup null eq {pop exit} if} loop
stroke flattenpath grestore
```

であり, strokeより前の部分はTrajectory型のデータからパス(path)を作り出すための手続きである.

MASKVECTORは上記の3つのプログラムをつなげることで出来る. しかし, これらのプログラムの大部分はTrajectory型のデータの作成と解釈の作業であり, MASKVECTORではTrajectory型のデータを保存する必要はないので, PostScriptのパスの機能を使って, もっと簡単なプログラムを作成できる. この方針でプログラムの変形を行なうと, MASKVECTORのプログラムは以下のようになる.

```
gsave newpath moveto lineto stroke grestore
```

これでMASKVECTORを表現でき, 同時にMOVETO, LINETO, MASKSTROKEも得れた.

4.2 FGET, FSET

FGETとFSETは, 操作的意味からすぐ分かるように, フレームの値の取り出しと設定をするための命令である. まずフレームの実現は非常に簡単で, 以下のようにフレームとしてframedictを作る.

```
/framedict 50 dict def
```

フレームからの値の取り出しFGETは,

```
framedict exch get
```

フレームへの値の設定FSETも

```
framedict 3 1 roll exch put
```

となる. これからわかるように, フレームの実現は非常に容易である.

4.3 IGET, ISET

IGETとISETも, 操作的意味からすぐ分かるように, イメージ変数の値の取り出しと設定である. フレームと同様に, まずイメージ変数imagedictを作る.

```
/imagedict 30 dict def
```

イメージ変数の取り出しIGETも, FGETとほぼ同様にできるが, 実際のグラフィックス状態はPostScriptの環境にあるので, IGETを適用するたびにimagedictを最新の状態に更新する. このための命令をImageinitとすると, IGETのプログラムは次のようになる.

```
Imageinit imagedict exch get
```

イメージ変数に値を設定するISETのプログラムもFSETとほぼ同様であるが, PostScriptのグラフィックス状態に影響を与えるための工夫がいる.

```
isetdict exch get exec
```

ここで, isetdictはプログラムの配列であり, 以下のようになっている.

```
isetdict begin
0  % DCScpx
    {currentpoint transform exch pop
    itransform moveto} bind def
...(一部省略)...
23 % strokeJoin
    {dup 1 eq {pop 2} {dup 2 eq
    {pop 1} if} ifelse
    setlinejoin} bind def
24 % clipper
    {pop} bind def
end
```

4.4 CORRECT

文字位置の補正を行なう命令CORRECTの操作的意味は3.2節で示さなかったが, Interpressの表現の中で一番複雑なものである. 直観的に言えば, 与えられた文字列を与えられた幅の中に文字の間隔を調整して配置する命令である. CORRECTのプログラムは次のようになる.

```
DOSAVESIMPLEBODY dup
 1 19 ISET currentpoint
 /CpY exch def /CpX exch def
 /MaskCount 0 def
 /SumX 0 def /SumY 0 def exec %PASS 1
 /TargetX CpX 2 IGET add def
 /TargetY CpY 3 IGET add def
 /MaskX 0 def /MaskY 0 def
 /MaskCount dup load 1 sub def
 /SpaceX TargetX currentpoint pop sub def
 /SpaceY TargetY currentpoint exch
 pop sub def
 SpaceX SpaceY Length SumX SumY Length
 20 IGET mul gt
 CpX CpY TargetX TargetY Distance CpX
 CpY currentpoint Distance lt
 and
 {/MaskX 20 IGET SumX mul SpaceX add def
 /MaskY 20 IGET SumY mul SpaceY add def
 /SpaceX SpaceX MaskX sub def
 /SpaceY SpaceY MaskY sub def
 }if
 DORESTORESIMPLEBODY
 DOSAVESIMPLEBODY
 2 19 ISET CpX CpY moveto exec %PASS 2
 0 19 ISET TargetX TargetY
 currenptpoint Distance
 21 IGET 22 IGET Length gt {(error) =} if
 TargetX TargetY moveto
 DORESTORESIMPLEBODY
```

上の Length, Distance, DOSAVESIMPLEBODY, DORESTORESIMPLEBODY は, PostScript で別に書いたプログラムである. なお, CORRECT の実行コードを合計すると, 上記のプログラムの 2 倍以上になる.

5 ip2ps

ip2ps は, Interpress マスタを PostScript のプログラムに変換するソフトウェアである. 4 節で与えたような Interpress の PostScript による記述を用いて実現し, Interpress マスタをほぼそのままの形で PostScript のプログラムとして解釈できるように変換している.

ip2ps は 2 フェーズで構成されていて, 第 1 フェーズでは, Interpress マスタを, コード化された表現から対応する ASCII 表現に変更する. また, Interpress と PostScript ではデータの表現方法が異なるので, データ型の変換もこのときに行なう. 第 2 フェーズでは, Interpress の操作的意味を PostScript で定義したファイル IP.proc を, プロローグとして第 1 フェーズで作成したファイルの先頭につける. この方法で, Interpress マスタは PostScript のプログラムに変換される.

IP.proc では, Interpress の命令に PostScript で操作的意味を与えるために, PostScript の関数定義を用いている. 例えば, カレントポイントを移動する命令 MOVE は次のように定義されている.

```
/MOVE { currentpoint translate } bind def
```

すなわち, MOVE という文字列にマッチしたら, PostScript の組み込み命令 currentpoint と translate をこの順に実行することを意味している. 前に述べた MASKVECTOR, FGET, FSET, IGET, ISET, CORRECT も同様に定義している.

6 考察

プログラム変換として, いろいろな方法が研究されている. 本節では, 我々の方法と他の研究との関連について述べる.

プログラム変換の代表的なものとして, 展開/畳み込み変換がある. 初期の研究として, Burstall と Darlington による再帰プログラムのための展開/畳み込み変換システムがある [4]. 簡単にいえば, Burstall らのシステムは, 関数型プログラムにおいて, 元のプログラムから効率的な再帰プログラムをつくり出すための項書換え系になっている. また, 論理プログラミングの領域においても, 玉木と佐藤による研究成果がある [11]. 玉木らのシステムでは, 論理プログラムにおける非決定性を, 展開/畳み込み変換で枝刈りを行なって, 効率的なプログラムを抽出する. 我々のシステムでも, MASKVECTOR のプログラムで見たように, 操作的意味を与えた Interpress の仕様から PostScript のプログラムを生成する際に, 実行

効率を良くするために展開/畳み込み変換を行なっている. すなわち, PostScript の意味論に基づいて, より効率の良い PostScript 命令へ変換している.

プログラムの実行効率を上げるための方法としては, 他に部分計算がある. 理論的な研究成果として二村のものがあり, プログラミング言語インタプリタからコンパイラが機械的に生成できることが示されている [5]. 部分計算の実装はいくつかのシステムで行なわれており, 最近では Standard ML のコンパイラにおける部分計算が知られている. 我々のシステムでも, 操作的意味で与えた仕様からプログラムへの変換において, 部分計算を行なって実行効率を上げている.

類似したプログラミング言語間のプログラム変換としては, 佐藤と玉木の第一階コンパイラの研究がある [10]. これは第一階論理プログラミング言語を, Prolog 上に作られたインタプリタへコンパイルするものである. 我々の ip2ps は, ページ記述言語における同種のプログラム変換といえる. さらに佐藤らとの違いは, ip2ps が Interpress マスタの意味解釈を全く行なわずに文法のみでプログラム変換を行なっていることである. この実現においては, PostScript の命令定義と辞書の機能の活用がキーポインとになっている.

7 まとめ

Interpress に操作的意味を与え, それに基づいて Interpress の PostScript による表現を示した. また Interpress から PostScript へのプログラム変換ソフトウェア ip2ps について説明した. これにより, Interpress を PostScript に変換できることを示した. 本稿で示した方法は, 他のページ記述言語におけるプログラム変換でも有用であると思う.

現在, ip2ps は我々の日常的な道具となり, 多くの人に利用されている. 初期のバージョンを使って, さまざまなアドバイスを下さった方々に感謝する.

参考文献

[1] Adobe Systems: *PostScript Reference Manual*, Addison-Wesley, 1985.

[2] Adobe Systems: *PostScript Language Tutorial and Cookbook*, Addison-Wesley, 1985.

[3] Adobe Systems: *PostScript Language Reference Manual Second Edition*, Addison-Wesley, 1990.

[4] Burstall, R.M., Darlington, J.: A Transformation System for Developing Recursive Programs, *J. ACM*, Vol.24, No.1 (1977), pp.44-67.

[5] Futamura, Y.: Partial computation of programs, *LNCS* 147 (1983).

[6] Knuth, D.E.: *The TEXbook*, Addison-Wesley, 1984.

[7] Harrington, S.J., Buckley, R.R.: *Interpress, The Source Book - The Documnet and Page Description Language for Performance Printing* –, Brady, 1988.

[8] 伊知地宏, 芳賀進, 清水一善: UNIX 上のページ記述言語 Interpress の処理系とプレビューア, SSE 研究集会, 1990.

[9] 伊知地宏, 窪寺隆行, 森田雅夫: Interpress から PostScript へのプログラム変換, 富士ゼロックステクニカルリポート No.7 (1993).

[10] 佐藤泰介, 玉木久夫: 第一階コンパイラ, コンピュータソフトウェア, Vol.5, No.2(1988), pp.69-80.

[11] Tamaki, H., Sato, T.: Unfold/fold Transformation of Logic Programming, *Proc. 2nd International Conference*, 1984, pp.127-138.

[12] Tannent, R.D.: *Semantics of Programming Languages*, Prentice Hall International, 1991.

[13] 上谷晃弘 (編著): 改訂2版ローカルエリアネットワーク, 丸善, 1989.

[14] Xerox: *Interpress Electronic Printing Standard Version 3.0*, 1986.