MLSwf ライブラリ

香川考司
香川大学工学部信頼性情報システム工学科
e-mail: kagawa@eng.kagawa-u.ac.jp

概 要

MLSwf は Objective Caml で SWF フォーマットを出力するためのライブラリである。SWF は**S**hock**W**ave **F**lash™ のことで、Macromedia 社の Flash の出力フォーマットとして利用されている。ベクターグラフィックスとアニメーションの WWW 上の事実上の標準として、多くの Web ブラウザで再生が可能となっている。

MLSwf は Objective Caml のラベル引数・オプショナル引数やクラスなどの機能を利用している。本稿では MLSwf ライブラリの目標と設計原理・概要を紹介し、現状と将来の課題を報告する。

1 はじめに

MLSwf は Objective Caml[6] で SWF フォーマットを出力するためのライブラリである。SWF は WWW 上でよく用いられるベクター形式の画像フォーマットである。

一般的に ML や Haskell などの関数型言語はリストや木などのデータ構造を扱うためのライブラリは充実しているものの、グラフィックスやネットワーク関係などのライブラリは充実しているとは言えない。このために、初学者用の教材は抽象的になりがちで、言語の習得の敷居を高くしている。初学者が学ぶプログラミング言語を選ぶときには、どのようなことに利用するか具体的に想定しているはずであり、いくらリストや木のような多相データ型を楽に扱えても、仕様が簡潔で優れていてもアピールが弱い。

Java がここまで普及したのは、AWT や Swing などをはじめとするグラフィックスやネットワーク関係のライブラリ関数（パッケージ）の充実に負うところが大きく、Perl が普及したのは、正規表現や HERE ドキュメントなど、プログラミング言語の仕様としては本質的ではないが、CGI を書くのに便利な特徴があったことによるところが大きいのはまちがいないであろう。

WWW が普及するにつれて、Web ページ上でグラフィックスやアニメーションが多用されるようになって来ている。WWW 上のベクターグラフィックスのフォーマットとしては、Macromedia 社の SWF（**S**hock**W**ave **F**lash™）が事実上の標準となっており、W3C で制定されている SVG や 3 次元グラフィックスの X3D（VRML を XML 化したもの）なども今後の普及が期待されている。

関数型言語のグラフィックスやアニメーションへの応用は、比較的早くから研究されている [5, 2]。しかし、Java のようにプラットフォーム共通の言語標準のライブラリとして整備されていないために、一般に普及していない。WWW で利用されている標準を利用すれば、各プラットフォームに対してライブラリを用意することが可能である。しかも WWW のサーバサイドプログラムにすぐに応用する事ができる。

現在、グラフィックスを CGI や Servlet のよ

うな Web サーバサイドプログラムで生成するために利用できるライブラリとしては、例えば Perl の GD, Perl-Fu（Gimp とのインターフェース）や、Java の Image I/O API（`javax.imageio` パッケージ）、 SVG を生成することができる Batik （`http://xml.apache.org/batik/`）などが挙げられる。これらは現在、Web サーバサイドプログラム用の言語として実績のある Perl や Java のライブラリとして提供されている。しかし、これまでのサーバサイドプログラムとして利用されているものは、おもにテキストを処理するプログラムであり、グラフィックス処理には別のプログラミング言語が適している可能性がある。例えば Perl のような静的な型チェックのない言語では、より多くの型を扱う可能性のあるグラフィックス処理では型エラーに悩まされる可能性がある。実際、Perl スクリプトのデバッグの難しさはテキスト処理が主流の現在でも、十分問題となっている。また、実行時の型チェックのために十分な効率が得られない可能性がある。一方、Java のようなオブジェクト指向言語では型を指定するためのコーディング量が増えてしまう恐れがある。初学者にとっては型を指定することによって増えるコーディング量や、型（クラス）名を調べるために資料を調べる手間は無視できない問題である。実際、Java をサーバサイドで使用するときはサーブレットなどの Java のプログラムを直接作成するよりも、JSP（Java Server Pages）のように Java を元にしたスクリプト言語を用いることが多いと思われる。JSP は HTML や XML のソースの一部に動的に生成される部分を Java のプログラムとして埋め込むものである。しかし、JSP の利点は画像の生成にはあまり活かされない。

そこで、ML のような型推論を行なう関数型言語の手軽さと表現力を活かして、Web 用の画像のデファクトスタンダードのフォーマットを操作するためのライブラリを作成すれば、これまでとは異なるユーザ層に関数型言語が普及する可能性がある。

ML には Standard ML と Objective Caml の 2 つの大きな方言がある。

このうち、Objective Caml は、ラベル引数・オプショナル引数 [3]・クラス [7] など実用的なライブラリを書くための機能が充実している。またラスターグラフィックスやフォント関係のライブラリなど、SWF 形式を扱う上で必要となる基盤も Objective Caml の方が充実している。特に、ラベル引数・オプショナル引数は SWF 形式においてオプションが多用されるために、本システムにおいて本質的である。

また ML ではないが、型推論を行なう関数型言語として、遅延評価を採用する Haskell もある。しかし、遅延評価型の言語はプログラミングの考え方があまりにも命令型言語とかけ離れている。さらに現実的には効率を良くすることが難しい上、デバッグがたいへん難しい。

このような理由で、本システムではプログラミング言語として Objective Caml を採用した。

本稿では、MLSwf のデザインと概要を紹介し、現状と将来の課題について述べる。第 2 節では Objective Caml について他の言語と比較しながら簡単に説明する。第 3 節では SWF 形式について例を用いて説明する。第 4 節では MLSwf ライブラリが提供している API を低レベル API と高レベル API にわけて紹介する。第 5 節では将来の課題について述べる。最後に第 6 節では本ライブラリのデザインを通じて Objective Caml および関数型言語全体について感じた点を述べる。

2 Objective Caml

Objective Caml は、フランスの INRIA を中心に開発が進められている ML の方言の一つである。ここでは、読者に一般的な ML の知識を仮定し、Objective Caml に特有の事柄 — 特にラベ

ルつき変数とオプショナル変数 — について説明する。これらはともに Version 3.00 で導入された機構である。

2.1 ラベルつき引数

Objective Caml では関数を定義する時、引数にラベルをつけることができる。

```
let mk_pt ~x:x0 ~y:y0 = (x0, y0);;
```

~x:と~y:がラベルであり、そのうしろのx0とy0が対応する引数である。ラベル名と変数名が同一で構わない時は、省略して次のように書くこともできる。

```
let mk_pt ~x ~y = (x, y);;
```

この関数の型は次のように表される。

```
val mk_pt : x:'a -> y:'b -> 'a * 'b
```

この関数を使用する時、ラベルつきの引数は好きな順番に並べることができる。

```
# mk_pt ~x:2 ~y:3;;
- : int * int = (2, 3)
# mk_pt ~y:1 ~x:2;;
- : int * int = (2, 1)
```

また、次のように部分適用することさえ可能である。

```
# mk_pt ~y:3;;
- : x:'a -> 'a * int = <fun>
```

2.2 オプショナル引数

オプショナル引数を定義する時は、ラベル名の~を?に変える。

```
let mk_cpt ~x ~y ?c () = match c with
     None     -> (x, y, "black")
   | Some str -> (x, y, str);;
```

なお、オプショナル引数を使用する場合は、その後ろに必ずラベルのない引数が必要である。これがなければ、呼出し時に引数が省略されたかどうかを判断できないからである。

オプショナル引数は関数定義の中では、次に示すoption 型:

```
type 'a option = None | Some of 'a
```

のオブジェクトとして扱われる。

```
# mk_cpt ~y:2 ~x:1 ~c:"red" ();;
- : int * int * string = (1, 2, "red")
```

のようにオプショナル引数を省略しなかった場合は、仮引数c にはSome "red"が渡され、

```
# mk_cpt ~x:1 ~y:2 ();;
- : int * int * string = (1, 2, "black")
```

のように省略した場合には、仮引数c にNone が渡される。

関数定義時に既定値を指定することも可能である。

```
let mk_cpt ~x ~y ?(c = "black") ()
                  = (x, y, c);;
```

この場合は、仮引数c はoption 型でなく、既定値の型（この例の場合string 型）を持つものとして扱われる。

型を示す時にはオプショナル引数は次のように?付きのラベルで示される。

```
val mk_cpt : x:'a -> y:'b -> ?c:string
          -> unit -> 'a * 'b * string
```

3 SWF 形式

SWF 形式は、ベクター形式のグラフィックスおよびアニメーションのためのフォーマットであり、次のような目的を持って設計されている。

- 帯域幅に制限のあるネットワークで配送できること

- ストリーミングによって配送される時に、インクリメンタルに再生できること
- フォントを含む外部の資源に依存しないこと

つまり、ネットワークや再生プログラムに対して “軽量” であることが大きな目標である。

SWF 形式には、はじめにファイルの長さ、フレームのサイズ、1 秒あたりのフレーム数、トータルのフレーム数などの情報を含む**ヘッダ**がある。そしてヘッダの後にタグつきのデータブロックがいくつか続くという形になっている。各データブロックは、ブロックの種類を表すタグ番号とブロック長ではじまり、そのあとにタグ固有のパラメータが続く。

SWF 形式の再生プログラムは理解できないタグをもつデータブロックをスキップしても良いことになっている。ブロック長はこのために必要である。

SWF 形式は Flash バージョン 4 時点で 0〜48 まで[1]の番号を持つタグを使用する。

SWF タグは、図形やビットマップ、テキストなど（SWF ではまとめて**キャラクタ**と呼ぶ）を定義（define）する**定義タグ**と、それを配置（place）したり、移動したりする**コントロールタグ**にわけることができる。SWF では、必ずまず定義タグを用いてキャラクタを作成してから、コントロールタグで配置するという順番をたどる。

例えば、PlaceObject2 というキャラクタを配置または移動・変形するコントロールタグでは、データブロックは表 1 のような形式になっている。なお、MATRIX, CXFORM などの基本データ型は別に形式が定められている。多くのパラメータがオプションとなっていることがわかる。例えば Ratio はシェイプトゥイーン（モーフィング）のとき、Name はスクリプト（ActionScript）からコントロールされるとき、Clip はクリッピング（他に配置されたキャラクタの一部だけを見せること）のときだけに使用される。

他のタグについてはここでは示さないが、やはり 1 つのタグが多くのパラメータを持ち、その中の多くがオプショナルになっている。

SWF 形式の仕様の詳細について興味のある読者は OpenSWF.org[1] などの WWW 上の情報を参考にされたい。

4 MLSwf の API

前節で紹介したような特徴を持つ SWF 形式を生成するために、まず SWF のタグに（ほぼ）一対一に対応する関数が必要である。ここではそのような関数群を低レベル API と呼ぶことにする。低レベル API に属する関数のデザインはある意味単純である。基本的には関数名としては SWF 仕様の中のタグの名前を、ラベル名として SWF 仕様の中のフィールド名をそのまま使用する。ただし、単語の区切りに大文字を使用する FooBar のような名前は、Objective Caml で一般的な、アンダースコアを区切りに利用する `foo_bar` のようなスタイルに変更する。SWF 仕様の中でオプショナルなパラメータは MLSwf でもオプショナルである。

低レベル API だけでは、人手でおもしろいアニメーションを作成することはたいへん難しい。そこで、単純に SWF のタグの 1 つに変換されるのではない高レベルな API が必要となる。現時点の MLSwf では高レベル API は充実しているとは言いがたいが、いくつかの簡単な機能は提供している。

以下では MLSwf の提供する関数群を低レベルと高レベルにわけて紹介する。

[1] ただし一部欠番あり。

フィールド名	型	説明
タグ番号	10bit	PlaceObject2 の場合は 26
ブロック長	6bit	ただし`0x3F` の時は、つづく 32bit がブロック長
Flags	8bit	特定のパラメータがあるかどうかを示すフラグ
	（Flags&0x01==1 のときはすでに配置されたキャラクタを移動することを示す）	
Depth	16bit	キャラクタを配置する深さ
CharacterId	Flags&0x02==1 のときは 16bit	配置すべきキャラクタの番号
Matrix	Flags&0x04==1 のとき MATRIX	変形・移動を表す行列
Cxform	Flags&0x08==1 のとき CXFORM	色の変換（Cxform = Color Transform）
Ratio	Flags&0x10==1 のとき 16bit	モーフィングの割合
Name	Flags&0x20==1 のとき STRING	名前
Clip	Flags&0x40==1 のとき 16bit	クリップする深さ

表 1: PlaceObject2 タグ

4.1 低レベル関数

低レベル関数は、基本的には SWF フォーマットの中のタグに一対一に対応するものであるが、SWF ファイルの詳細をプログラマから隠すという重要な役割を持っている。

これらの低レベル関数は`Swftype` と`Swfutil` というモジュールに定義されているため、使用する時には

```
open Swftype
open Swfutil
```

の 2 行が必要である。

前節で SWF 形式が、ネットワークや再生プログラムの負担を軽くするために設計されていることを紹介した。その結果として、SWF 形式は次のような特徴も持っている。

- データ量を小さくするためにデータの配置がバイト境界に必ずしも従っていない
- ブロックを読み飛ばせるように、ブロック長をブロックのはじめに持っている

例えば、ビット単位の出力命令は標準ライブラリに用意されていないので、バッファを用意しなければいけないし、ブロック長を記録する位置を記憶しておいて、データブロックの長さがわかった時点でいったん巻き戻してブロック長を記録する必要がある。

`swf_obj` 型はこのように SWF フォーマットを出力するために必要な処理を自動的におこなってくれる出力ストリームの型である。この型のオブジェクトは、出力の何**ビット**目までを使用したか、データブロックのサイズは何バイトになっているか、次にキャラクタの id として未使用の整数はいくつか、などの情報を保持する。この型のオブジェクトは `new_swf_obj` 関数によって生成される。

```
type swf_obj
val new_swf_obj : unit -> swf_obj
```

例えば、PlaceObject2 タグを出力する関数の型は次のようになっている。

```
val place_object2 :
  ?move:bool -> ?depth:int
  -> ?character:int -> ?matrix:matrix
  -> ?cxform:cxform -> ?ratio:int
  -> ?name:string -> ?clip:int
  -> swf_obj -> int
```

swf_obj 型の引数だけが必須となっている。あとのパラメータはすべてオプショナルである。move と depth も SWF タグのパラメータとしては必須（move は 1bit なので当然であるが）だが、MLSwf では省略した時には、それぞれfalse（move）とその時に使われていない最小の整数（depth）が使われるようになっている。place_object2 の戻り値は、このdepth の値そのものである。

この関数は、図 1 の例のように用いられる。最初の place_object2 は図形を配置し、**for** ループの中の place_object2 はこの図形を移動するために使用している。（ちなみに SWF の中の長さの単位は TWIPS と呼ばれ、1TWIPS は 1/20 ピクセルに相当する。上の例では 1 フレーム毎に 100TWIPS=5 ピクセル移動している。）

その他の命令に対応する低レベル関数には図 2 のようなものがある。また、フォント・テキスト関連の低レベル関数は、別に図 3 に示す。この中で使われているデータ型については、このすぐ後で説明する。

ラベル付き引数のおかげで、多くの場合関数の型がそのままある程度のドキュメントになっている。なお、ここでは、いくつかのあまり使用しないオプショナル引数は省略している場合があることをお断りしておく。define_shape**3**, define_bits_JPEG**2** などと数字のつく命令が多いのは、Flash のバージョンが上がるにしたがって、少しずつ意味が強化されたタグが追加されていったからである。define_shape**2**, define_bits など、旧バージョンのタグに対応する命令も MLSwf には存在するが、ここでは説明を省略する。

少し補足すると、図 2 と図 3 の関数は、show_frame と set_background_color, do_action, define_font_info の 4 つを除けば、すべてキャラクタを定義する関数である。このような関数については、戻り値はすべて定義されたキャラクタの ID である。通常はその時に未使用の最小の整数が使用されるが、オプショナル引数id で明示的に与えられた場合は、それをそのまま使用する。do_action はフレームが表示された時に実行するスクリプトを定義する関数である。define_bits_JPEG2, define_bits_lossless2, define_sound の引数の中の bin_data はバイナリデータを表す MLSwf の型で、ユーザは通常はこれらの関数を直接使用することはなく、あとで紹介する画像・サウンドを表すデータ型を引数として渡せる関数を使用する。

define_button2 の button_record 型の引数はボタンのアップ、ダウン、オーバー、ヒットテストの各状態のときに表示するキャラクタを指定するためのデータである。さらに、この関数の最後から 2 番目の引数は action_condition list とaction list の組のリストの型になっているが、この action_condition はアクションを対応させるイベントを表す。この部分には ‘Over_down_to_idle, ‘Idle_to_over_down, ‘Out_down_to_idle, ‘Out_down_to_over_down, ‘Over_down_to_out_down, ‘Over_down_to_over_up ‘Over_up_to_over_down, ‘Over_up_to_idle, ‘Idle_to_over_up を使用することができる。さらに、‘Key_code of int という形式によってキーが押されたというイベントを指定することもできる。

define_font_info と define_font2 の flags に使用できるフラグ用のヴァリアントとしては、‘Unicode, ‘SJIS, ‘Ansi, ‘Italic, ‘Bold が、define_edit_text の flags に使用できるヴァリアントとしては、‘Disable_editing, ‘Password, ‘Multi_line, ‘Word_wrap, ‘Use_outlines, ‘Draw_box, ‘Disable_select がある。define_font_info の意味については後述する。

このようにして作成された swf_obj は、最終

```
let swfo = new_swf_obj () in
let shape = ... (* 何か図形の定義 *) in
let d = place_object2 ~character:shape swfo in
show_frame swfo;
for x=1 to 10 do
  let matrix = mk_matrix ~trans:(x*100, x*100) () in
  place_object2 ~move:true ~depth:d ~matrix swfo;
  show_frame swfo
done;
```

図 1: place_object2 の使用例

```
val show_frame : swf_obj -> unit  (* フレーム（ページ）を表示する *)
val set_background_color : color -> swf_obj -> unit (* 背景色を定義する *)
val do_action : action list -> swf_obj -> unit (* スクリプトを定義する *)

val define_shape3 :  (* 2 次ベジエ曲線で表現された‘‘形’’を定義する *)
  ?id:int -> bounds:rect -> shape_with_style -> swf_obj -> int
val define_morph_shape : (* シェイプトゥイーン（モーフィング）を定義する *)
  ?id:int -> bounds1:rect -> bounds2:rect -> morph_shape_with_style ->
  swf_obj -> int

val define_bits_JPEG2 :  (* JPEG 形式で圧縮された Bitmap データを定義する *)
  ?id:int -> bin_data -> swf_obj -> int
val define_bits_lossless2 : (* PNG（に似た）形式の Bitmap データを定義する *)
  ?id:int -> format:int -> int -> int -> bin_data -> swf_obj -> int
val define_sound : (* サウンドを定義する *)
  ?id:int -> format:bool -> rate:int ->
  size:bool -> stereo:bool -> count:int -> bin_data -> swf_obj -> int

val define_sprite : (* ムービークリップ（ムービーの中のムービー）を定義する *)
  ?id:int -> ?count:int -> f:(swf_obj -> unit) -> swf_obj -> int
val define_button2 : (* ボタンを定義する *)
  ?id:int -> button_record list -> (action_condition list * action list) list ->
  swf_obj -> int
```

図 2: その他の低レベル関数

```
type font_shape_table (* フォントシェイプ情報 *)
type font_code_table  (* フォントの文字コードの対応情報 *)

val define_font : (* フォントシェイプを定義する *)
  ?id:int -> font_shape_table -> swf_obj -> int
val define_font_info : id:int -> name:string ->
  flags:font_flag list -> codes:font_code_table -> swf_obj -> unit
val define_text : (* テキストを定義する *)
  ?id:int -> bounds:rect -> ?matrix:matrix -> text_record list ->
  swf_obj -> int

val define_font2 : (* デバイスフォントを定義する *)
  ?id:int -> ?flags:font_flag list -> name:string -> swf_obj -> int
val define_edit_text : (* テキスト入力フィールドを定義する *)
  ?id:int -> bounds:rect -> ?flags:edit_text_flag list -> ?font:int*int ->
  ?color:color -> ?length:int -> ?layout:int*int*int*int*int ->
  name:string -> ?init:string -> swf_obj -> int
```

図 3: フォント・テキスト関係の低レベル関数

的に次の関数でファイルに出力される。

```
val write_swf_file : ?version:int ->
  ?xmin:int -> ?ymin:int ->
  int -> int ->
  ?rate:int -> ?count:int ->
  name:string -> swf_obj -> unit
val write_html_file :
  ?before:string -> ?after:string ->
  int -> int ->
  html:string -> swf:string -> unit
```

さらに、write_html_file は SWF を読込むためのタグを含む HTML ファイルを出力する関数である。

4.2 基本データ型に関する関数

これまで紹介したような低レベル関数に付随して必要となる関数群に color, rect, matrix, cxform（Color Transform）, fill_style, line_style, shape_record, button_record, action, text_record などの基本データ型に関する関数がある。これらのなかでフォント・テキスト関連以外を図 4 に紹介する。mk_button_record の flags はキャラクタを表示すべきボタンの状態を指定するためのもので、‘Hit_test, ‘Down, ‘Over, ‘Up を指定できる。

フォント・テキスト関連のデータ型に関しては少し詳しい説明が必要である。SWF ファイルの中では、文字は define_font に渡す font_shape_table の中の添字で指定する。これと通常の UNICODE などの文字コードとの対応をとるのが define_font_info に渡す font_code_table 型の引数である。通常の string 型の文字列から text_record を作成するためには、この font_code_table が必要である。さらに、text_record の中ではフォントの幅の情報が必要になる。mk_text_record() 関数の引数の font_metrics_table はこのフォントの幅を含むメトリック情報である。これらのテーブルをすべて用意するために、mk_font_info という関数を用意している。この関数の第 2 引数は、SWF ファイル中でそのフォントを利用する使用

```
val mk_RGB : ?r:int -> ?g:int -> ?b:int -> ?a:int -> unit -> color
val mk_HSB : hue:float -> sat:float -> bright:float -> color
val mk_rect : xmin:int -> xmax:int -> ymin:int -> ymax:int -> rect
val mk_matrix : ?scale:float*float -> ?rotate:float*float ->
                ?trans:int*int -> unit -> matrix
val mk_cxform : ?mult:color -> ?add:color -> unit -> cxform
type fill_style = Solid of color | Tiled_bitmap of int * matrix
                | Clipped_bitmap of int * matrix | ...
val mk_line_style : ?color:color -> int -> line_style
val straight : int * int -> shape_record
val curved : int * int * int * int -> shape_record
val mk_non_edge :
  ?move_to:int*int -> ?fill_style0:int -> ?fill_style1:int ->
  ?line_style:int -> ?new_styles:fill_style array * line_style array ->
  unit -> shape_record
type shape_with_style =
       fill_style array * line_style array * shape_record list

val mk_button_record : character:int -> flags:button_record_flag list ->
  ?depth:int -> ?matrix:matrix -> ?cxform:cxform -> unit -> button_record
type action = Next_frame | Previous_frame | Play | Stop
            | Goto_frame of int | ...
```

図 4: 基本データ型に関する関数

```
type swf_glyph_metrics  (* フォントのメトリック情報 *)
type font = int -> shape_record list * swf_glyph_metrics
type font_metrics_table

val mk_font : Freetype.t -> string -> font (* open Ftutil が必要 *)
val mk_text_record1 : ?font:int*int -> ?color:color ->
  ?xoff:int -> ?yoff:int -> unit -> text_record
val mk_text_record0 : fontht:int ->
  font_code_table*font_metrics_table -> string -> text_record*int
val mk_font_info : font -> int array ->
  font_shape_table * (font_code_table * font_metrics_table)
```

図 5: フォント・テキスト関連のデータ型に関する関数

```
let library = Freetype.init () in
let fontf = mk_font library fontfile in
let fontht = 30*20 in
let line ="Hello!" in
let used = Array.of_list (List.map Char.code ['H'; 'e'; 'l'; 'o'; '!']) in
let (font, fi) = mk_font_info fontf used in

let font_id = define_font font swfo in
define_font_info ~id:font_id ~name:"some font name" ~flags:[‘Ansi]
                 ~codes:(fst fi) swfo;
let tr1 = mk_text_record1 ~font:(font_id, fontht) ~color:red () in
let (tr0, wd) = mk_text_record0 ~fontht fi line in
let text_id = define_text ~bounds:(mk_rect2 (wd, fontht)) [tr1;tr0] swfo in
let dp = place_object2 ~matrix:(mk_matrix ~trans:(100,100) ())
                       ~character:text_id swfo in
show_frame swfo;
```

図 6: 典型的なテキスト・フォント関係の関数の使用例

するすべての文字の文字コードのリストである。mk_text_record0 は text_record 型のオブジェクトとその幅を組として返す。典型的なテキスト・フォント関係の関数の使用例は図 6 のようになる。

さらに、ラスターグラフィックスやサウンドなどに関する関数群も必要である。これらを図 7 に紹介する。

JPEG や PNG などのラスターグラフィックスの読込みには CamlImages ライブラリ[2]を利用する。TrueType フォント（Windows などで利用されている）の読込みには FreeType ライブラリ[3]を利用している。ただし、JPEG や PNG 形式の画像は SWF 形式内では通常と少し異なる形式で格納されており、また FreeType ライブラリには CamlImages ライブラリ経由で利用可能な関数群もあるが、MLSwf ではフォントのアウトライン情報[4]が必要となるので、これらの部分に関しては独自のインターフェースを用意している。

Image.t は CamlImages ライブラリの中で定義されている画像のデータタイプで、次の関数により画像ファイルから作成可能である。

```
load : string -> load_option list -> t
```

load の第 1 引数の string は画像ファイル名である。

4.3 高レベル関数

ここで高レベルと呼んでいるのは、SWF の命令に単純に一対一で対応しないという意味である。通常の基準では十分低レベルと言えるだろう。

例えば、SWF フォーマットのなかでは、組込みのフォント（SWF ファイルのなかにアウトライン情報が含まれるフォント）とデバイスフォント（_typewriter, _serif などプラットフォームに用意されているフォント）を利用する場合では、

[2] http://pauillac.inria.fr/camlimages/

[3] http://www.freetype.org/

[4] なお、SWF フォーマットでは TrueType のヒント情報は利用しないので、パテントに触れる部分は使用していない。

```
(* open Bitslossless が必要*)
val define_bits_lossless2_from_image :?id:int -> Image.t -> swf_obj -> int
(* open Bitsjpeg が必要 *)
val define_bits_JPEG3_from_image : ?id:int -> Image.t -> swf_obj -> int
(* 以下の 2 つはopen Sound が必要 *)
val parse_WAV : in_channel -> Sound.t
val define_sound_uncompressed : ?id:int -> Sount.t -> swf_obj -> int
```

図 7: イメージ・サウンドなどに関する関数

使用する命令が define_font と define_font2 というように異なる。テキストを表示する時もそれぞれ define_text と define_edit_text を利用する。そこで高レベル API でこれらの差を吸収できるように設計している。また、低レベル関数では、図形や文字列を定義する時にフォントや色やサイズをいちいち指定しなければならないが、これらも直前と同じであれば省略できることが望ましい。さらに SWF では“かたち”は 2 次ベジエ曲線として表現しなくてはならない。これに対して、長方形・楕円などよく使用するかたちを手軽に提供できる必要がある。

このような関数は Java の AWT に用意されている関数と機能的にほとんど同じなので、ML-Swf でも AWT と同じような名前で用意した（図 8）。graphics は swf_obj に加えて、フォントや色の情報を保持する出力ストリームのようなものである。new_embeded_font と new_device_font の flags は、define_font_info や define_font2 の flags と同じである。

例えば、図 9 のようなプログラムで 図 10 のような図形が生成される。

この他に、簡易テキストレイアウト関数として、図 11 のような関数を用意している。このような関数を利用すると、図 12 のような配置は比較的簡単に作成することができる。

さらに、簡易テキストレイアウト関数の手に負えないような複雑なレイアウトは、TEX で作成した DVI ファイルを読込む関数を用意することで対応している。

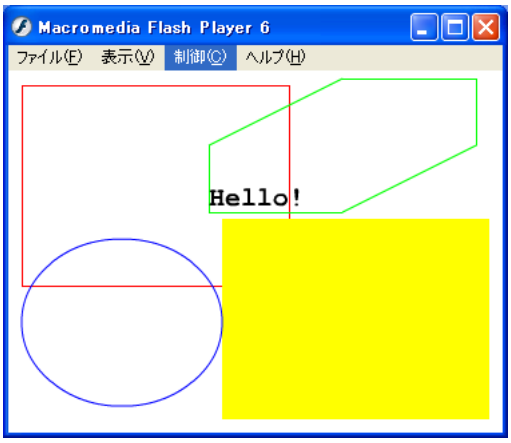

図 10: graphics の使用例（結果）

DVI 読込み部分については、ここでは詳しく説明できないが、DVI フォーマットの読込み部分は Active-DVI[5]からコードを借用している。また LATEX のスタイルフォーマットも Active-DVI のものを流用できるようにしている。また、Active-DVI 用の special 命令の一部を解釈することが可能で、これによって、LATEX のソースの中に命令を追加しておくことで、アニメーション効果を挿入する事が可能である。

[5]http://pauillac.inria.fr/advi/

```
type u_font
val new_device_font : ?flags:font_flag list -> string -> u_font
val new_embeded_font : string -> ?flags:font_flag list -> Swftype.font -> u_font

type graphics
val new_graphics : swf_obj -> graphics
val draw_rect : int -> int -> int -> int -> graphics -> int
val draw_oval : int -> int -> int -> int -> graphics -> int
val draw_line : int -> int -> int -> int -> graphics -> int
val draw_polyline : (int * int) list -> graphics -> int
val draw_polygon : (int * int) list -> graphics -> int
val draw_arc : int -> int -> int -> int -> int -> int -> graphics -> int
val draw_bezier :  close:bool -> (int * int) list -> graphics -> int
val fill_rect : int -> int -> int -> int -> graphics -> int
val fill_oval : int -> int -> int -> int -> graphics -> int
val fill_polygon : (int * int) list -> graphics -> int
val fill_arc : int -> int -> int -> int -> int -> int -> graphics -> int
val fill_bezier :  (int * int) list -> graphics -> int
val get_color : graphics -> color
val set_color : color -> graphics -> unit
val get_font : graphics -> u_font
val set_font : u_font -> graphics -> unit
val get_size : graphics -> int
val set_size : int -> graphics -> unit
```

図 8: JavaAWT 風の関数

```
let g = new_graphics swfo in
set_color red g;
let s1 = draw_rect 200 200 4000 3000 g in
let d1 = place_object2S ~character:s1 g in
...
let library = Freetype.init () in
let fontf = mk_font library fontfile in
let font = new_embeded_font "courbd" fontf in
set_font font g; set_size 400 g; set_color black g;
let s5 = draw_string "Hello!" 3000 2000 g in
let d5 = place_object2S ~character:s5 g in
show_frame g;
```

図 9: graphics の使用例

```
type text_layout
type text_element
val new_text_layout : swf_object -> text_layout
type justify = Left | Center | Right
val set_justify : justify -> text_layout -> unit
val add_string : ?color:color -> ?size:int -> ?font:u_font ->
            ?sameas:text_element -> string -> text_layout -> text_element
val line_break : text_layout -> unit
```

図 11: テキストレイアウト用の関数

図 12: テキストレイアウト関数の使用例（結果）

4.4 ツール

MLSwf で生成中にはエラーが起こらないのに、生成した SWF 形式を再生しようとすると、エラーが出て再生できない時がある。このような時には、MLSwf ライブラリに付属しているデバッグ用のツールの`swfparse` を利用する。`swfparse` は SWF 形式を人間に理解できるテキスト形式に変換するプログラムである。この形式は Objective Caml のプログラムで、コンパイルして実行すれば、もとの SWF ファイルが再生成されるはずである。もし`swfparse` が生成された SWF ファイルを読もうとしてエラーを出力すれば、おそらく MLSwf のバグである。そうでない場合は、`swfparse` の出力をさらに詳細に調査する必要がある。

5 現状と将来の課題

現在は “高レベル” 関数でも JavaAWT 相当であり、実用的な作品を作成するためには、さらに抽象度の高い関数を用意する必要があると思われる。

応用例 これまでは、MLSwf による作品として、ハノイの塔・ソートアルゴリズム・演算子順位法による構文解析の説明などの教材用のアニメーションを作成している。応用例の作成もまだまだ始まったばかりであり、今後もっと多くの作品を作成して、MLSwf の設計にフィードバックする必要がある。

関連ライブラリ Caml 以外の言語で SWF を出力するためのライブラリとして、Ming（`http://ming.sourceforge.net/`）や JavaSWF2 （`http://www.anotherbigidea.com/javaswf/`）など、Java, PHP などの言語用のものがある

これらのライブラリとの表現力の比較を行ない、良いところを採り入れ、MLSwf をさらに使いやすくする必要がある。

多相ヴァリアントの利用 MLSwf ライブラリでは、Objective Caml（Ver 3.0 以降）の特徴であるラベル付き引数、オプショナル引数をとくに多用しているが、もう一つの特徴である多相ヴァ

リアントはフラグ引数として少し利用しているだけである。多相ヴァリアントはTcl/Tk用のライブラリに利用され、ひじょうに有効であることが確認されている[4]。

多相ヴァリアントはMLSwfではフラグ以外の引数としても、もっと有効に利用できるはずである。例えば、いくつかの関数には座標として値そのものを受取るバージョンと、あとで必要な時に評価されるthunkを受取るバージョンが存在する。(後者は複雑なレイアウトをしたい時に必要になる。)これらのバージョンの違いについては、多相ヴァリアントを利用して一つの関数にまとめることが考えられる。

しかし、このような利用法では関数を多相ヴァリアントを利用して変更すると、呼出し側もそれにあわせてすべて書換えなければならない。それに対してオプショナル引数は呼出し側を変更することなく関数に機能を追加することが可能である。オプショナル引数のこのような柔軟性はライブラリの進化中にひじょうに有効であった。例えば、まず最小限の機能を実装し、そのあと追加の機能を実装した時はオプショナル引数として追加するなどである。そうすると、既存の呼出し側のコードは変更することなく引続き利用することが可能である。

関数のラベルと一体化して、関数を定義しなおしても呼出し側を書き直さなくても良いような仕組みがあれば、多相ヴァリアントをより活用するできると思われる。

クラスの利用　Objective Camlのオブジェクト指向関係の機能(クラス)は今のところ、`rect`など基本的なオブジェクトの型として利用している。それによって`move`などのメソッドが多重定義されている。また、`swf_obj`や`graphics`, `text_layout`などの出力ストリームもクラスとして定義している。これによって、`swf_obj`を引数として受け取ることができる関数は`graphics`や`text_layout`のオブジェクトも引数として受け取ることができる。これらのクラスはコンストラクタに引数として渡される`swf_obj`にほとんどのメッセージをそのまま渡す、いわゆるdelegationスタイルである。

この他の部分であまりクラスを利用していない理由の一つに、クラスを使用した時に型エラーのメッセージがひじょうに読みにくくなってしまうという点が挙げられる。これは初学者にとっては実際に大きな問題となるだろう。

MLSwfではあるデータ型がクラスなのかレコード型なのか、外にはできるだけわからないようにしている。現在のObjective Camlは、レコードは“`.`”を、オブジェクトは“`#`”を用いてメンバにアクセスする。このため、これらのアクセス方法を利用していると、最初はレコードとしてデザインして、あとでクラスに書き換えることが簡単にできない。そこでラッパ関数を用意して、できるだけ一般ユーザはこれらのアクセス方法を利用しなくても良いようにしている。

Objective Camlのクラスの定義の仕方はまだこなれていないところがあるように思われる。例えば、値のレベルの記述と型のレベルの記述が入り混じってしまう点、単に`val`で宣言した変数を返すだけの`method`をたくさん定義しなければならない点などである。

読込みフォーマット　既存のアプリケーションと連携するために、より多くの画像フォーマットを読込めることが望ましい。このうちラスター形式の画像はCamlImagesがかなり対応しているが、ベクター形式は独自に対応する必要がある。現在のところ、MLSwfが対応しているのはベクター形式はSWF形式のみで、他のアプリケーションで作成したベクター形式の画像を読み込むためには、Macromedia FlashやAdobe LiveMotionなどの有償のソフトウェアを利用してSWF形式に変換する必要がある。

需要が大きいと考えられるのは、整形文書のWWW上のデファクトスタンダードとなっているPDFと、Windowsで広く用いられているWMF（Windows Meta Format）である。これらのフォーマットの仕様は公開されているので、基本的には手間をかければ、読込みライブラリの作成は可能である。

書出しフォーマット　SWFの他にWWW上で今後普及が見込まれているベクターグラフィックスフォーマットに、SVG（Scalable Vector Graphics）がある。SVGはW3C（`http://www.w3.org/`）でXMLの一種として規格が制定されている。

SVGはSWFと比較すると、パラメータを連続的に変化させるいわゆる“宣言的なアニメーション”の他は、アニメーションの機能は内部でのECMAScript（いわゆるJavaScript）の使用に依存している。このため、他の言語ではアニメーションをコントロールしづらいところがある。しかし、静止画部分についてはSWFと共通化することも可能であろう。

アニメーションのプラグイン化　`dviswf`はMLSwfのDVI形式読込み機能を利用したDVIからSWFへの変換プログラムである。しかし、Microsoft PowerPointに用意されているような、さまざまなアニメーション効果を用意することはできていない。利用者が簡単にアニメーション効果を追加できるようなプラグインの仕組みが必要であると考えられる。

6　おわりに

MLSwfの作成で一番手間がかかったところは、SWFやJPEG, TrueTypeなどの規格を調べ、入出力のための関数群を作るところであり、その点ではC言語やJavaなど他の言語を用いた時と大差はない。Objective Camlという言語を用いた利点はこのライブラリを使用する時に現れるはずである。本稿の時点では、ライブラリを使用した経験がまだ少なく、この点を報告できるには至っていない。

ここでは、本ライブラリのデザインを通じてObjective Camlおよび関数型言語全体について感じたところを述べる。

関数型言語の特徴である高階関数は、ユーザの目に見える範囲では`define_sprite`や`font`関係で利用しているだけである。しかし、これはJavaのように内部クラスなどといった大がかりなものを使わなくて良いので、目立っていないだけと言えるかも知れない。また、ライブラリを利用してユーザがプログラムを作成する時には高階関数はひじょうに有効になると思われる。

ライブラリの作成時にも型推論はもちろん有用で多くのミスを事前に発見できた。ただ、ライブラリを初心者が使用する時にはObjective Camlの式の逐次実行の区切りの“`;`”を忘れてしまった時のエラーメッセージが問題となるかもしれない。このようなエラーメッセージを出力する際には、レイアウト情報を考慮する必要があると思われる。

本ライブラリは、Objective Camlを非手続き的な — つまり破壊的な代入などの副作用を使用しない — プログラミングをするために採用したのではない。むしろ副作用は積極的に利用している。このような目的の場合、Objective Camlで`for`文のループ変数が`int`型でしか使えないのは問題である。リストなど他のデータ型に関する繰返しにも`for`文が利用できることが望まれる。

また、通常の整数（`int`）型が32bitでないところは、予想以上に面倒の種だった。本ライブラリのなかでは31bitの`int`型とスタンダードライブラリの中の`Int32.t`型を必要に応じて使いわけている。使いわけ方はかなりアドホックである。

とくに Objective Caml では+などの四則演算子が多重定義されていないため、当初int 型を使用していたところをInt32.t 型に書き換えるなどの作業はたいへん手間がかかった。32bit 整数の取扱いはいまなお関数型言語の未解決な課題である。しかし、+, -などの演算子が定義されるデータ型や数値リテラルが Lisp のように、いわゆる Bignum（Objective Caml ではNum.num 型）— 無限精度の有理数 — であった方が便利ではないかと考えられる。

最後に、本ライブラリの作成は結果がアニメーションとなるため、やりがいのある面白い仕事だった。本ライブラリによって Objective Caml などの関数型言語に興味を持つ人がより多く出てきてくれれば、作者にとってこれに勝る喜びはない。

ダウンロード

MLSwf ライブラリは http://guppy.eng.kagawa-u.ac.jp/~kagawa/MLSwf/ からダウンロード可能である。

謝辞

Objective Caml のラベル付き・オプショナル引数の部分を設計した京都大学数理解析研究所の Jacques Garrigue 氏、INRIA の古瀬 淳氏に感謝する。古瀬氏は Active-DVI と CamlImages ライブラリの作者の一人でもある。両氏にはライブラリの作成中に、さまざまな助言をいただいた。また、SWF 形式の謎を解明するのに貢献された多くの方々に感謝する。

本研究は科研費・若手研究（B）13780243 の支援を受けている。

参考文献

[1] OpenSWF.org. http://www.openswf.org/.

[2] Guy Cousineau and Michel Mauny. *The Functional Approach to Programming*. Cambridge University Press, 1998.

[3] Jun P. Furuse and Jacques Garrigue. A label-selective lambda-calculus with optional arguments and its compilation method, October 1995. RIMS Preprint 1041, Research Institute for Mathematical Sciences, Kyoto University.

[4] Jacques Garrigue. Programming with polymorphic variants. In *ML Workshop*, September 1998.

[5] Peter Henderson. Functional geometry. In *ACM Symposium on Lisp and Functional Programming*, pages 179–187, 1982.

[6] Xavier Leroy, Damien Dolegez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system, release 3.06 Documentation and user's manual*. INRIA, August 2002.

[7] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1), 1998.