プログラミング言語 13－1
（１９８７・９・１６）

GはGMWのG
GMWウィンドウ・システムの拡張用言語Gについて

萩谷昌己

京都大学数理解析研究所

ＧＭＷウィンドウ・システムの拡張用言語であるＧについて述べる。ＧＭＷは、仮想機械サーバ方式のウィンドウ・システムであり、ウィンドウ・サーバ内で、Ｍと呼ばれるマルチ・タスクを行う仮想機械が稼働している。ユーザは、仮想機械のコードをダウン・ロードすることにより、サーバとクライアントの間のプロトコルを拡張することができる。Ｇは、ウィンドウ・サーバ内で走るアプリケーションを記述するための言語で、ＧのプログラムはＭの仮想コードにコンパイルされる。Ｇの特徴は、強い型付けの下で、オブジェクト指向プログラミングに必要な動的メソッド探索の機能をサポートしていることと、Ｍに基いたマルチ・タスク機能を持っていることである。本稿では、Ｇの型の体系とマルチ・タスク機能について述べる。

Language G
The Extension Language for the GMW Window System

Masami Hagiya

Research Institutue for Mathematical Sciences
Kyoto University
Sakyo, Kyoto 606, Japan

This paper describes the extension language for the GMW Window System. GMW is a virtual-machine-server-type window system in that a multi-tasking virtual machine, called M, is running inside the window server. A user of the window system can dynamically extend the protocol set between the server and the client by loading virtual code into the server. G is the language for describing applications that run inside the window server. A program written in G is compiled into virtual code of M. The characteristic features of G are: While it is a strongly-typed language, it supports the facility of dynamic method search needed for object-oriented programming. It also has the facility of multi-tasking that is based on M. This paper describes the type system and the multi-tasking facility of G.

1. はじめに

ＧＭＷ[萩谷87、Hagiya87]は、京都大学、立石電機、アステックの三者によって開発された、汎用ワークステーション（ＵＮＩＸ等の既存の汎用ＴＳＳをＯＳとするワークステーション）のためのウィンドウ・システムである。ＧＭＷの特徴は大きく次の三点にまとめられる。

- すべての描画操作が、ディスプレイ・オブジェクトと呼ぶ概念に基いて提供されているので、描画すべき実体に則した描画が可能である。
- ディスプレイ・デスクの機能を用いることにより、関係し合うウィンドウを一つにまとめ、一つのオブジェクトとして取り扱うことができる。
- 仮想機械サーバ方式を採用しているため、ウィンドウ・サーバとクライアントの間のプロトコルの変更や追加を容易に行うことができ、両者の間の通信の負荷を軽減することができる。

上記の三点のうち、本稿に関係するものは主として第三の特徴である。それについて述べるために、まず、汎用ワークテーション上のウィンドウ・システムの実現法について以下に簡単にまとめておく。

汎用ワークステーション上のウィンドウ・システムの実現法は、次の三種類に分類することができる。

- カーネル方式
  ＯＳのカーネル内にウィンドウ・システムの全機能を置き、アプリケーションはシステム・コールによってウィン ドウ・システムの機能を呼び出す。
- ユーザ・カーネル方式
  アプリケーション(の一つ一つ)にウィンドウ・システムの必要な部分をライブラリとして結合し、排他制御等の最小限の機能をＯＳのカーネル内に置く。
- サーバ方式
  ウィンドウ・システムのすべての機能をサーバと呼ばれる一つのユーザ・プロセスに置き、アプリケーション(サーバに対してクライアントと呼ばれる)は、サーバへプロセス間通信(リモート・プロシージャ・コール)を行うことによってウィンドウ・システムの機能を呼び出す。

仮想機械サーバ方式とは、サーバ方式を進めたもので、サーバ内に仮想コード・インタープリタを置き、サーバに仮想コード・プログラムをダウン・ロードすることにより、サーバとクライアントの間のプロトコルを拡張できるようにしたものである。仮想機械サーバ方式では、サーバとクライアントの間の通信は仮想コードの形式で行われる。

ＧＭＷの開発において、仮想機械サーバ方式を採用したのは、以下のような考察による。

サーバ方式のウィンドウ・システムは、カーネル方式に比べて、ＯＳのカーネルの変更を必要最小限に抑えることができるので、開発が容易であり移植性も高い。また、サーバがユーザ・プロセスとして実現されるので、メモリの使用等に関して柔軟性に富んでいる。ユーザ・カーネル方式は、アプリケーションのプロセスの一つ一つがウィンドウ・システム全体を背負っているようなものなので、ＯＳ全体のメモリ効率が非常に悪いが、サーバ方式では、プロセス間通信のインターフェースのみをライブラリとして結合すればよい。以上のような利点に対して、単純なサーバ方式の欠点は、サーバとクライアントの間のプロセス間通信のオーバヘッドが大きいことである。また、プロセス間通信のプロトコルが固定されているため、ライブラリによってしかウィンドウ・システムの機能を拡張することができないので、ライブラリが大きくなった場合、ユーザ・カーネル方式と同様の欠点が生じる。

以上の理由に加えて、仮想機械サーバ方式を採用したのは、ウィンドウ・システムが次のような特徴を持っているからである。

ウィンドウ・システムは、そのアプリケーションも含めて、オブジェクト指向プログラミングの枠組の中で構築されることが多い。(Smalltalk-80[Goldberg83]やLisp Machine[Weinreb81]がその例である。) これには様々な理由が考えられるが、その一つは、ウィンドウ・システムが極めて多くのライブラリから成り、それらが何重もの階層を構成しているということである。しかも、アプリケーションは、そのどの階層にも自由にアクセスできることが望ましい。また、アプリケーション自身も、ライブラリと同様の複雑な構造を成すことが一般的である。以上のようなライブラリやアプリケーションを統一的に扱い、見通しよく整理・統合するには、オブジェクト指向プログラミングの枠組が優れていることはいうまでもない。

ウィンドウ・システムとそのアプリケーションのもう一つの特徴は、並列処理の必要性である。ウィンドウ・システム自身は、マウスやキーボードからの非同期的な入力を処理すると同時に、複数のウィンドウへ同時に描画を行わなくてはならない。また、ウィンドウ・システム上のアプリケーションも、ウィンドウ・システムから送られる様々なイベントを処理しながら、多くのウィンドウを管理しなくてはならない。従って、ウィンドウ・システムのアプリケーションは、本質的に複数のプロセスから構成されることになる。ところが、アプリケーションの中のウィンドウを管理する部分は、多くのデータを共有し複雑に関係し合うことが一般的なので、ＵＮＩＸにおけるような完全に独立したアドレス空間を持つプロセスに分割することは容易ではない。Smalltalk-80、Lisp Machine等、成功したウィンドウ・システムの多く(特に初期のもの)は、同一のアドレス空間の中でプロセスが走るものである。また、ウィンドウ・システムとそのアプリケーションにおいては、動的にプロセスが必要になることが多くある。例えば、メニューやパネルが作られたとき、その一つ一つを別々のプロセスで管理することは極めて自然なことである。ところが、ＵＮＩＸ等のＴＳＳのプロセスは非常に重たいリソースであり、プロセス生成のオーバーヘッドが大きく、プロセスを使い捨てにすることは、一般に不可能であるか非現実的である。オブジェクト指向プログラミングの観点から見ると、ウィンドウ・システムとそのアプリケーションは、並列オブジェクト指向のパラダイムが適している分野であるということができる。

一方、単純なウィンドウ・サーバであっても、動的に生成・消滅する膨大なデータを効率よく管理するために、ゴミ集めを含めたメモリ管理の機能を持っている。また、複数のクライアントからのリクエストを同時に実行するために、最小限のコンテキスト・スイッチの機能がなくてはならない。

以上の考察から、ウィンドウ・サーバを拡張して仮想機械コードのインタプリタを埋め込み、ウィンドウ・システムとそのアプリケーションに適した仮想機械を、そうでないＯＳの上に実現することはすることは、極めて意味のあることと思われた。このような考えに基いて設計・実現されたのが、ＧＭＷの仮想機械Ｍであり、その上のコードを生成する高級言語Ｇである。

しかし、仮想機械が実現すべき機能は、本来、ＯＳ自身が持っているべき機能であり、上述の機能を既存のＯＳに付加するか、あるいは、上述の機能を持ったＯＳを新たに設計・実現し、その上にウィンドウ・システムを構築するのが正しい道筋である。しかし、ウィンドウ・システムやそれに類似したシステムがどのような機能を必要としているかを、仮想機械を用いて検証することは、新たなＯＳを構築するためにも、極めて意義のあることと考えられる。

Ｇは、ウィンドウ・システムに代表される、上述のような特徴を持つシステムを記述するための言語として、現在開発中のプログラミング言語である。仮想機械ＭはＧのプログラムを実行するための特殊な命令を持ち、ＧのプログラムはＭ上の仮想コードにコンパイルされるが、Ｇのプログラムはのみで実行可能というわけではない。Ｍが重要なのは、そのメモリ管理機能とマルチ・タスク機能である。逆に、Ｍのメモリ管理機能とマルチ・タスク機能は、Ｇのプログラムの実行を前提にして設計されている。

Sun MicrosystemsのNeWS[SUN86、SUN87]も同様のアプローチを採っているウィンドウ・システムである。NeWSは、PostScriptを拡張してウィンドウ・システムに必要な機能を実現している。従って、ユーザは、仮想コードであるPostScriptプログラムを手で書くことになる。NeWSのPostScriptは様々な拡張がなされ、オブジェクト指向的な機能も導入されているが、やはり可読性に問題がある。これに対して、Ｇは通常の汎用言語として設計されたプログラミング言語である。

本稿では、まず、Ｇの設計方針と構文について簡単に述べた後、オブジェクト指向的な機能も含めて、Ｇの型について説明する。次に、Ｇのマルチ・タスク機能について述べ、最後に、実際のプログラムの例をあげる。

2. Ｇの設計方針

Ｇの設計目標は次の二点である。

- オブジェクト指向プログラミングのための機能を持つ。
- マルチ・タスクの機能を持つ。

しかし、どちらの機能も、多くのバリエーションに富んでおり、個々のアプリケーションによって必要とされる機能が異なるものである。従って、Ｇは、完成された言語ではなく、上記の目標を達成するための土台とすべきだと考えた。例えば、オブジェクト指向プログラミングに関しては、複雑なインヘリタンスの機能を始めから提供するのではなく、そのような機能を実現するための基本的な枠組を提供しようと考えた。

複雑な機能を組み立てていく土台としては、各データ構造や制御構造が実際にどのようにして実現されるかを、ユーザがある程度把握できることが重要である。なぜなら、基本的な枠組の効率は、その上に組み立てられた機能の効率を大きく左右するからである。仮想機械Ｍは、そのためのモデルと考えることができる。

Ｇが実行されるアーキテクチャとしては、既存の汎用計算機のアーキテクチャを前提とした。仮想機械Ｍも、既存のアーキテクチャの上で効率よく実現されることを前提としている。従って、LispやSmalltalkのようなシステムとは異なり、Ｍ上のすべてのデータが型情報を持っているわけではない。従って、型情報をコンパイル時に検出する必要がある。また、型の機構は、効率のよい実行だけでなく、プログラムの信頼性や可読性を向上させる。以上のような考察から、Ｇは強く型付けされた言語であるとした。従って、強い型付けの機能と、動的なメソッド探索を基礎にしたオブジェクト指向の機能とを、どのように融合させるかが課題の一つとなった。

オブジェクト指向とマルチ・タスクを組み合わせた機能として、並列オブジェクト指向の機能が考えられるが、これも、基本的な枠組を用いて組み立てるべきものと考えている。

3. 構文について

Ｇのコンパイラは、現在、プロトタイプがCommon Lisp[Steele84]上に実現されている。このために、Ｇの構文はLispと同様のS式によって表現される。また、Lisp用のエディタを流用したりするために、各構文を表すシンボルも、Lisp(特にCommon Lisp)に従っている。しかし、構文上の類似はあくまで便宜的なものであり、ＧとLispとは全く異なる言語である。

3.1. プログラム

Ｇのプログラムは、次のような構文(トップ・レベル式と呼ぶ)の並びである。

- ストラクチャ定義
```
(defstruct ストラクチャ名
  (型 メンバ名) … (型 メンバ名))
```
- 型定義
```
(deftype 型名 型)
```
- 関数定義
```
(defun 値の型 関数名
    ((型 仮引数) … (型 仮引数))
  式 … 式)
```
- メソッド定義
```
(defmethod 値の型
      (ストラクチャ名 . メソッド名)
    ((型 仮引数) … (型 仮引数))
  式 … 式)
```

● 大域変数宣言
(defvar 型 変数名 [初期値式])

3.2. 式

定数としては、

```
整数            浮動点小数
文字            文字列
#t              #f
NIL
'名前
@型名
```

等がある。#tは真、#fは偽を表す。'名前は名前の引用である。式には、次のようなものがある。

```
定数
変数
(関数 式 … 式)          (関数呼び出し)
(式 . メンバ名)          (メンバ参照)
((メソッド名 . 式) 式 … 式)
                (メソッド呼び出し)
(vref 式 式)
                (ベクタのエレメントの参照)
(vlength 式)            (ベクタの長さ)
(new)
                (弱いストラクチャの生成)
(vnew 式)               (弱いベクタの生成)
#(式 … 式)
                (ストラクチャ、ベクタ)
(setq 式 式)            (代入)
(eq 式 式)              (等式)
(the 型 式)             (コアーション)

(progn 式 … 式)
(let ((型 局所変数 [初期値式])
      …
      (型 局所変数 [初期値式]))
  式 … 式)
(with 式 式 … 式)
(if 式 式 [式])
(when 式 式 … 式)
(unless 式 式 … 式)
(cond ((式 式 … 式) … (式 式 … 式))
(loop 式 … 式)
(do ((型 局所変数 [初期値式] [更新値式]))
      …
     (型 局所変数 [初期値式] [更新値式])))
  式 … 式)
(dotimes (変数 式) 式 … 式)
(lreturn [式])          (loopからのリターン)
(freturn [式])          (関数からのリターン)
(catch 式 式 … 式)
(throw 式 [式])
```

マルチ・タスク関係の構文は後に述べる。

4. 型

本節では、Gの型について述べる。メソッド定義等、オブジェクト指向的な機能については次節で詳しく述べる。

4.1. Mのオブジェクト

仮想機械Mは、動的に生成・消滅するデータ（例えば、ピクセル・バッファ、フォント)を、オブジェクト配列とヒープを用いて管理している。各データに対して、オブジェクト配列の一つのエントリ(エレメント)が対応する。オブジェクト配列の各エントリは、

```
型
ディレクトリ
ヒープ上のデータへのポインタ
ヒープ上のデータの大きさ
```

の四つフィールドからなる。型は、データの形式に関する情報である。ディレクトリは、データの割り当てと保護に関する情報である。データの本体はヒープ上に割り当てられ、オブジェクト配列のエントリの残りの二つのフィールドに、ヒープ上のデータへのポインタとデータの大きさが入る。各データは、常にオブジェクト配列のエントリへのポインタによって参照される。

4.2. 強い型と弱い型

Gの扱うデータは、大きく二つに分かれる。一つは、Mのオブジェクトによって実現され、ポインタによって参照されるデータである。もう一つは、ポインタでは表現されないデータで、整数や文字等の基本的なデータがこれに属する。また、C言語のストラクチャやPascalのレコードに相当するものや、長さの定まったベクタも後者に属する。C言語やPascalにおいてポインタによって表現されるデータは、Gではすべて Mのオブジェクトによって実現される。前者のデータを強いオブジェクトと呼び、後者を弱いオブジェクトと呼ぶ。

強いオブジェクトの型を強い型といい、弱いオブジェクトの型を弱い型という。強い型に関しては、データ自身が型情報を持っていないので、変数や式の型が静的(コンパイル時)に決定しなくてはならない。これに対して、弱い型に関しては、データ(Mのオブジェクト)が型情報を持っているので、動的な型チェックが可能である。

4.3. 基本型

いくつかの型が基本型として用意されている。現在、

```
void
bool
int
u-int(unsigned int)
char
float
```

がある。これらはすべて強い型である。voidは値がないことを示すための型(値を返さない関数の型)である。また、弱い型の全体(合併)を意味する型として、

```
*
```

がある。

4.4. 型の生成

基本型から、以下の操作によって、新たに型を作ることができる。

弱いストラクチャ型
強いストラクチャ型
弱いベクタ型
強いベクタ型
関数型

弱いストラクチャ型、弱いベクタ型、関数型は弱い型であり、強いストラクチャ型、強いベクタ型は強い型である。弱いストラクチャ型(単にストラクチャ型ともいう)は、C言語におけるストラクチャへのポインタ、Pascalにおけるレコードへのポインタに相当するものである。ストラクチャは、ストラクチャ定義によって新たに定義される。例えば、

```
(defstruct point (int x) (int y))
```

とすると、xとyという名の二つのメンバを持つpointというストラクチャ型が定義される。(従って弱いストラクチャ型には必ず名前がつく。)メンバの型は両方ともintである。弱いストラクチャ型Sに対して、

```
(strong S)
```

は、Sに対応した強いストラクチャ型である。強いストラクチャ型は、C言語のストラクチャ、Pascalのレコードに相当するものである。例えば、

```
(strong point)
```

は、pointに対応した強いストラクチャ型である。以上のように、Gでは、C言語やPascalとは逆に、まずポインタ型に相当するものが先に定義され、次に、それが指す本体の型が作られる。Gには、

C言語やPascalにおけるような一般的なポインタ型は存在しない。

Tを任意の型としたとき、

```
(vector T)
```

はエレメントの型をTとする弱いベクタ型である。弱いベクタ型のデータ(弱いベクタ)は長さが不定である。

```
(strong vector T n)
```

は、エレメントの型をTとする強いベクタ型である。強いベクタに対しては、大きさが定まっている。

T、Ti∈Tに対して、

```
(function T (T1 … Tn))
```

は、T1…Tnの元をもらってTの元を返す関数の型である。

deftypeによって型に名前をつけることができる。例えば、

```
(deftype s-point (strong point))
```

とすると、(strong point)の代わりにs-pointを用いることができる。

ストラクチャ型は、名前が違えば違う型であると考える。その他の型に関しては、構造的な比較が行われる。また、deftypeで定義された型は単なる型の省略形であるとされ、deftypeによって新たに型が作られるわけではない。

4.5. 型の包含とコアーション

*は弱い型の全体を表わす型である。これを基底として、型の間に包含関係≦を定義することができる。T≦Uは型Tが型Uに含まれることを意味する。

- T ≦ *
- T≦U ⇒
  (vector T) ≦ (vector U)
  (strong vector T n)
  ≦ (strong vector U n)
- Ui≦Ti, T≦U ⇒
  (function T (T1 … Tn))
  ≦ (function U (U1 … Un))

式の型Tが期待される型Uに含まれない場合、すなわち、T≦Uが成り立たない場合は、コアーション(型の強制変換)が行われる。コアーションは、

```
(the 型 式)
```

を用いて明示的に行うこともできる。Gでは次のようなコアーションが定義されている。

- voidへのコアーション
  任意の型はvoidへ変換できる。
- 強いストラクチャ型へのコアーション
  (弱い)ストラクチャ型Sは、(strong S)へ
  変換できる。
- *から弱い型へのコアーション
  *は任意の弱い型へ変換できるが、
  その際に動的な型チェックが行われる。
  例えば、eの型が*のとき、

  ```
  (the point e)
  ```

  とすると、eの型がpointかどうかが
  動的にチェックされる。

以上の他に、intとu-intの間のコアーションのように、特定の型の間のコアーションがある。

4.6. 型オブジェクトとメタ型

Mのオブジェクト配列の四つフィールドの中の型とは、型情報を表わすMのオブジェクトである。例えば、ストラクチャの場合、次のような三つ組からなるテーブルになる。

メンバ名　　オフセット　　型

または

メソッド名　　関数　　型

オフセットは、メンバのヒープ上における位置である。関数は、メソッドを実現する関数である。

このような型を表わすオブジェクトを型オブジェクトといい、型オブジェクトの型をメタ型という。メタ型は必ずストラクチャ型である。各型に対し

て、そのメタ型は次のように規定されている。

ストラクチャ型	structure
強いストラクチャ型	strong$structure
ベクタ型	vector
強いベクタ型	strong$vector
関数	function
*	**

各メタ型は以下のように定義される。

```
(deftype name (vector char))

(defstruct structure$member
  (name name)
  (int offset)
  (* type))

(defstruct structure$method
  (name name)
  (* method)
  (* type))

(defstruct structure
  (int size)
  ((vector (strong structure$member))
   members)
  ((vector (strong structure$method))
   methods))

(defstruct strong$structure
  (* structure))

(defstruct vector
  (* element-type))

(defstruct strong$vector
  (* element-type)
  (int number-of-elements))

(defstruct function
  (* result-type)
  ((vector *) argument-types))

(defstruct **)
```

メタ型は通常は意識する必要がないが、6節で述べたように、ユーザが型の拡張を行う場合に重要になる。このために、G(及びM)では、メタ型の仕様を定めている。

5. メンバとメソッド

5.1. メンバ参照

pの型がpointのとき、pのメンバxとメンバyは、それぞれ、

```
(p . x)      (p . y)
```

で参照される。メンバ参照の一般形は、

```
(式 . メンバ名)
```

である。例えば、

```
(defun void move-point
    ((point p) (int dx) (int dy))
  (setq (p . x) (+ (p . x) dx))
  (setq (p . y) (+ (p . y) dy)))
```

によって、pをx方向にdx、y方向にdy移動させる関数が定義される。

5.2. メソッド定義

move-pointの代わりに、pointに対してmoveという名前のメソッドを定義するには、

```
(defmethod void (point . move)
    ((int dx) (int dy))
  (setq x (+ x dx))
  (setq y (+ y dy)))
```

とする。メソッド内では、そのメソッドのターゲットのメンバ名を変数として用いることができる。また、ターゲットはselfという疑似変数によってアクセスできる。従って、上のメソッドは、

```
(defmethod void (point . move)
    ((int dx) (int dy))
  (setq (self . x) (+ (self . x) dx))
  (setq (self . y) (+ (self . y) dy)))
```

と等価である。

メソッドは関数によって実現される。メソッドを実現する関数は、ターゲットを第一引数とし、メソッドの引数を第二引数以降の引数とする関数である。

5.3. メソッド呼び出し

上で定義されたメソッドを起動するには、

```
((p . move) dx dy)
```

とする。メソッドの呼び出しの一般形は、

```
((式 . メソッド名) 式 … 式)
```

である。

メソッド定義の中では、他のメソッド名を関数名のように用いることができる。例えば、

```
(defmethod void (point . move-vertically)
    ((int y))
  (move 0 y))
```

は、

```
(defmethod void (point . move-vertically)
    ((int y))
  ((self . move) 0 y))
```

と等価である。

以上の例の中のメソッド呼び出しやメンバ参照においては、コンパイル時にターゲットとなるストラクチャの型が定まっている。このような場合、Gでは、メソッド呼び出しとメンバ参照は、それぞれ、メソッドを実現する関数呼び出しとオーフセットによるメンバの参照にコンパイルされる。

5.4. 動的メソッド探索

次の関数は、pをx方向とy方向に同じだけ移動させる関数である。

```
(defun void foo ((point p) (int d))
  ((p . move) d d))
```

この関数を、pointだけでなく、moveという名前のメソッドを持つ任意のストラクチャに対して定義したいときは、

```
(defun void foo ((* p) (int p))
  (declare (void move (int int)))
  ((p . move) d d))
```

とする。

Gは強く型付けされた言語である。ところが、動的にメソッドが探索された場合、探索によって得られたメソッドの型と、期待される型が異なる可能性がある。後述するように、Mの動的メソッド探索命令では、型の整合性が動的にチェックされるようになっている。

現在のGでは、動的なメソッド探索が必要な場合に、期待されるメソッドの型をユーザが明示的に指定するようになっている。上の場合、

```
(declare (void move (int int)))
```

として、moveの型を宣言しなければならない。moveの期待される型は、

```
(function void (int int))
```

である。これは、型の推論を容易にすると同時に、メソッド名を誤って書いた場合に、無差別的に動的メソッド探索にコンパイルされるのを防ぐためである。

メンバに対しても、動的メンバ探索を行うことができる。

```
(defun void bar ((* p))
  (declare (int x))
  (setq (p . x) (+ (p . x) 1)))
```

関数barは、pのxというint型のメンバを1増やす関数である。

動的なメソッド呼び出しは次のようにして行われる。(メンバの参照・更新もこれに準ずる。) スタックに、メソッド呼び出しの引数と、ターゲットとなる弱いストラクチャをプッシュする。そして、メソッド名と期待される型に対して、メソッド呼び出しの機械語命令を実行する。

メソッド呼び出しの機械語命令は、ターゲットの型(structureストラクチャ型のストラクチャ)を取り出し、メソッド名を探索する。メソッド名が見つからなければエラーである。メソッド名が見つかった場合は、メソッドの実際の型(T)と期待される型(U)と比較する。T≦Uならば、メソッドを実現する関数を呼び出す。T≦Uでなければ、型エラーが発生する。

6. 型の拡張について

現在、Gにはインヘリタンスの機能がない。インヘリタンスを含めて、型に関する豊富な機能を実現するために、Common Loops[Bobrow86]に従った型の拡張を考慮中である。ただし、Gでは、式に対して静的な型が定まっているという点が大きく異なる。

新しい型の種類を定義するために、任意のストラクチャを型オブジェクトとして許す。すなわち、任意のストラクチャ型をメタ型として用いる。ただし、メタ型として用いられるには、いくつかのメソッドが定義されていなければならない。ここでは、

```
dom
search-member
search-method
```

というメソッドを考える。domの型は、

```
(function bool (*))
```

である。domは、引数がその型に入っているかどうかをチェックするメソッドである。例えば、従来のストラクチャ型に対しては、

```
(defmethod bool (structure . dom)
    ((* x))
  (eq (type-of x) self))
```

と定義することができる(type-ofはオブジェクトの実際の型を与える)。search-memberの型は、

```
(function (strong structure$member)
          (name *))
```

である。第一引数はメンバ名、第二引数は期待されるメンバの型である (nameはメンバ名も含めた名前の型)。例えば、structureのsearch-memberは、

```
(defmethod (strong structure$member)
           (structure . search-member)
    ((name m) (* type))
  (dotimes (i (vlength members))
    (when (eq ((vref members i) . name)
              m)
      (unless
        (or (eq type NIL)
            (type<= type
                    ((vref members i)
                     . type)))
        (error))
      (freturn (vref members i))))
  #(NIL 0 NIL))
```

とmeta-circularに定義することができる(type<=は型の包含関係)。同様に、search-methodの型は、

```
(function (strong structure$method)
          (name *))
```

である。

型オブジェクトは、実際のオブジェクトの型になるとは限らない (例えば、*を型に持つオブジェクトはない)。実際にオブジェクトの型になるためには、メタ型に、

```
size
```

というintのメンバがあることが必要であるとする (実際にstructureはそうなっている)。

新しいメタ型としては、例えば、

```
(defstruct inheritable-structure
```

```
    ((vector *) supers)
    (int size)
    ((vector (strong structure$member))
     members)
    ((vector (strong structure$method))
     methods))

  (defstruct or
    ((vector *) disjuncts))
```

のようなものが考えられる。

コンパイラは、メソッド呼び出し

```
  ((e . m) a1 … an)
```

をコンパイルする場合、最初に式eの(静的な)型Tに対して、

```
  ((T . search-method) 'm NIL)
```

というメソッド呼び出しを行う。('mはmというメソッド名を表し、NILは期待する型がないことを示す。) もしメソッドが見つかれば、メソッド呼び出しを、メソッドを実現する関数の呼び出しにコンパイルする。メソッドが見つからなければ、動的メソッド呼び出しにする。(動的メソッド呼び出し対しても、静的な場合と同様に、オブジェクトの型のsearch-methodが呼び出される。) すなわち、メソッド探索おいては、静的な型が動的な型に優先する。型Tのメソッドmを探索して見つかればそれを返し、見つかれなければ型Uのメソッドmを探索することを、

```
  method2(T, U, m)
```

で表わすと、型Tの式eのメソッド探索は、静的な探索と動的な探索を合わせて、

```
  method2(T, type-of(e), m)
```

によって行われることになる。同様に、member2(T, U, m)も考える。

すると、型の包含関係T≦Uは、

- ((T . dom) x)=#t ⇒ ((U . dom) x)=#t
- 任意の((T . dom) x)=#tとなるxに対して、
  member2(T, type-of(x), m)
  = member2(U, type-of(x), m)
  method2(T, type-of(x), m)
  = method2(U, type-of(x), m)

と定義することができる(vector、function等に関しては従来通り)。すなわち、メンバ探索とメソッド探索に対する振る舞いが、Tの元とみなしたときも、Uの元とみなしたときも同じということである。

*に対して、

```
  (defmethod bool (** . dom) ((* x)) #t)
```

と定義し(**は*の型)、*にはメンバやメソッドが一つもない、すなわち、

```
  (defmethod (strong structure$member)
        (** . search-member)
     ((name m) (* type))
   #(NIL 0 NIL))

  (defmethod (strong structure$method)
        (** . search-member)
     ((name m) (* type))
   #(NIL NIL NIL))
```

とすると、上の定義を用いても、従来通りS≦*となる。

型Tへのコアーションは、domによる動的なチェックとして実現される。

ストラクチャ型に対して、dom、search-member、search-method等を定義すれば、新しい種類のメタ型を定義したことになる。しかし、≦は、dom、search-member、search-methodから実際に計算することができないので、何らかの方法で別に定義する必要がある。従って、その妥当性が問題となる。

現在、以上の機能を用いた、インヘリタンス、UNION型、(静的な)部分型等の実現方法を考察中である。

7. マルチ・タスク機能

7.1. タスクの生成

タスクを新しく作るには、

```
  (task (式 . メソッド名) 式 … 式)
  (task 関数 式 … 式)
```

という二種類の構文を用いる。これによってタスクが新しく作られ、そのタスクの下でメソッド呼び出し、もしくは、関数呼び出しが評価される。(task …)の値は、新しいタスクである。

タスクを生じるもとになったメソッド呼び出しのターゲットを、そのタスクの基底オブジェクトという。基底オブジェクトは、タスクの動的な環境を与えると考えることができる。基底オブジェクトは

```
  (myself)
```

によって参照される。関数呼び出しによってタスクが生成された場合は、新しいタスクは元のタスクの基底オブジェクトを継承する。

タスク自身は

```
  (mytask)
```

によって参照される。タスクを終了させるには、

```
  (exit)
```

とする。

7.2. 同期

タスク間の同期は、相互排除ストラクチャとガード付きメソッドによって行う。このために、ストラクチャ定義を次のように拡張する。

```
  (defexstruct ストラクチャ名
    (型 メンバ名)
       …
```

(型 メンバ名))

defexstruct によって定義されたストラクチャ型を、相互排除ストラクチャ型という。例えば、

```
(defexstruct buffer
  (int length)
  (int count)
  (int begin)
  (int end)
  ((vector char) body))
```

とすると、bufferストラクチャは相互排除ストラクチャになる。

相互排除ストラクチャ型に対しては、ガード付きメソッドを定義することができる。例えば、bufferに対しては、次のようなガード付きメソッドを定義することができる。

```
(defgmethod char (stream . read) ()
  (when (> count 0)
    (let ((char c (vref body begin)))
      (setq count (- count 1))
      (setq begin
            (mod (+ begin 1) length))
      c)))
```

同一の相互排除ストラクチャのガード付きメソッドの呼び出しは、同時に一つのタスクしか行うことができない。また、いうまでもなく、whenの後のガートが成り立たない限り、メソッドの本体は実行されない。

ガード付きメソッドは次のようにして定義される。

```
(defgmethod 型
      (ストラクチャ名 . メソッド名)
    ((型 仮引数) … (型 仮引数))
  (when ガード式 式 … 式))
```

概念的には、一つの相互排除ストラクチャには、二つのタスク・キューが存在する。一つは、ガードの評価を待っているタスクのキューで、これを白紙キューという。もう一つは、ガードの評価に一度失敗したタスクのキューで、これを失敗キューという。白紙キューにタスクがないときにガード付きメソッドを呼び出したタスクは、直ちにそのガードに評価に入る。もし、ガードの評価に失敗したならば、そのタスクは失敗キューに加えられる。この時点で白紙キューにタスクがあれば、タスクが一つ白紙キューから外されガードの実行に入る。タスクがガードの評価に成功した場合は、直ちにメソッドの本体の実行に入る。メソッドの本体の実行が終ると、まず、失敗キューのタスクがすべて白紙キューにつなぎかえられる。そして、白紙キューからタスクが一つ外されガードの実行に入る。

相互排除ストラクチャとガード付きメソッドは、抽象度が十分に高く、しかも、容易に実現することができる。また、他の同期のプリミティブも、相互排除ストラクチャとガード付きメソッドを用いて簡単に実現することができる。

8. プログラム例

次のプログラムは、ＧＭＷにおける簡単なメニュー・パッケージである。simple-menuというストラクチャが定義され、それに対するinit、handle-event、popというメソッドが定義されている。動的メソッド呼び出しはないが、event$queueのイベントに対して、動的メンバ参照が行われている。また、event$queueは相互排除ストラクチャで、dequeueはガード付きメソッドである。

```
(!defstruct simple-menu
  (display$object menu)
  (fragment fr)
  (sheet sh)
  (event$queue q))

(!defmethod void (simple-menu . init) (((vector v-char) labels))
  (let ((int w 0))
    (dotimes (i (vlength labels))
      (let ((int x (- (vlength (vref labels i)) 1)))
        (when (> x w) (setq w x))))
    (setq menu (menu-object-create "MENU" NIL NIL NIL 8 16 255))
    (dotimes (i (vlength labels))
      (menu-object-set-menu-item-text menu (vref labels i)))
    (setq fr (fr-create menu (* w 8) (* (+ (vlength labels) 2) 16)
                        RETAINED NIL ROP-CLEAR 255))
    (menu-object-draw menu fr)
    ((fr . set-frame) FRAME-NORMAL -1 2 2 2 2)
    (setq sh (sh-create 1))
    ((fr . put-sheet-group) sh 0 0 0)
    (setq q (new-event-queue))
    ((fr . set-event-behavior) EV-RIGHT-RELEASED #t q)
    ((fr . set-event-behavior) EV-CURSOR-MOVE #t q)))
```

```
(!defmethod int (simple-menu . handle-event) ((int id) (ev-cursor$move ev))
  (cond ((eq id EV-CHANGE)
         (menu-object-draw menu (ev . where))
         -1)
        ((eq id EV-REDRAW)
         (menu-object-draw menu (ev . where))
         -1)
        ((eq id EV-CURSOR-MOVE)
         (menu-object-display menu ((ev . locate) . y-coord))
         -1)
        ((eq id EV-RIGHT-RELEASED)
         (menu-object-get-current-item menu))
        (#t -1)))

(!defmethod int (simple-menu . pop) ((ev-cursor$move ev))
  (let ((fragment full-screen-fragment (get-full-screen-fragment))
        ((strong pix$locate) l)
        ((strong event$behavior) eb)
        (int item -1))
    (setq l (((ev . where) . fctofc) ((ev . locate) . x-coord)
                                     ((ev . locate) . y-coord)
                                     (get-full-screen-fragment)))
    (setq l ((full-screen-fragment . fctodobjc) (l . x-coord) (l . y-coord)))
    (menu-object-display menu 0)
    ((sh . set-group-position) 0 (l . x-coord) (l . y-coord))
    ((sh . put-desk) (get-full-screen-desk))
    (setq eb ((full-screen-fragment . get-event-behavior) EV-RIGHT-RELEASED))
    ((full-screen-fragment . set-event-behavior) EV-RIGHT-RELEASED #t q)
    (loop (let ((* e (q . dequeue)))
            (declare (int event-id) (fragment where))
            (when (and (eq (type-of e) @ev-cursor$move)
                       (eq (e . where) fr))
              (setq item (handle-event (e . event-id)
                                       (the ev-cursor$move e)))))
          (when (>= item 0)
            (!return)))
    ((sh . remove-desk))
    ((full-screen-fragment . set-event-behavior)
        EV-RIGHT-RELEASED (eb . catch) (eb . queue))
    item))

(!defun int menu-test ()
  (let ((event$queue q (new-event-queue))
        (message m (new))
        (simple-menu sm (new))
        (int item))
    ((sm . init) #("Sunday" "Monday" "Tuesday" "Wednesday"
                   "Thursday" "Friday" "Saturday"))
    ((m . init) #("Hello." "This is a menu test.") #(400 400))
    (((m . fr) . set-event-behavior) EV-RIGHT-PRESSED #t q)
    (let ((* ev (q . dequeue)))
      (setq item ((sm . pop) (the ev-cursor$move ev))))
    ((m . destroy))
    item))
```

文献

[萩谷87] 萩谷昌己: GMWウインドウ・システムについて, bit 1983年3月.

[Hagiya87] M. Hagiya: Introduction to the GMW Window System, RIMS-565, 1987.

[Goldberg83] A. Goldberg: Smalltalk-80, the interactive programming environment, Addison-Wesley, 1983.

[Weinreb81] D. Weinreb, D. Moon: Lisp Machine Manual, MIT AI Lab., 1981.

[Steele84] G. L. Steele: Common Lisp: the language, Digital Press, 1984.

[SUN86] Sun Microsystems: NeWS Preliminary Technical Overview, 1986.

[SUN87] Sun Microsystems: NeWS Manual, 1987.

[Bobrow86] D. G. Bobrow et al.: Common Loops, Merging Lisp and Objected-Oriented Programming, OOPSLA '86, 1986.