ソフトウェア基礎論　37－7
プログラミング言語　27－7
（1990. 12. 13）

GHC プログラムの視覚的入力システム：FE '92

田中二郎　　太田　祐紀子
（富士通・国際研）　　（富士通SSL）
（電子メイル：jiro@iias.fujitsu.co.jp）

あらまし　並列論理型言語GHC のプログラム入力を、視覚的、図形的に支援するシステム FE '92 (Future Environment '92)を開発した．本システムにおいては、GHC のプログラム節の入力において、その図形的な表現を併用することができ、比較的難解とされる並列論理型言語GHC のプログラムを視覚的に表現し、編集することができる．またこのFE '92のインプリメンテーションはGHC でなされており、GHC の典型的なプログラミング・パラダイムであるプロセスとストリームを多用したプログラムとなっている．また、これにより、ユーザとのマルチ・ウインドウによるインタフェイスが、極めて自然な形で記述されている．

Visual GHC Program Input System: FE '92

Jiro Tanaka
IIAS-SIS, Fujitsu Limited
1-17-25 Shinkamata, Ota-ku, Tokyo 144

Yukiko Ohta
Fujitsu SSL Limited
1-6-4 Osaki, Shinagawa-ku, Tokyo 141

Abstract　FE '92 is a visual program input system which can graphically assist the input of a GHC program. In this system, a program clause of a GHC program can be expressed as a figure and this makes the understanding of the GHC program more easily. The implementation of the system has been carried out in GHC. The notion of processes and streams are used in an extensive manner. This language feature of GHC makes it possible to express the multi-window interface betwen the system and the user in a natural manner.

1. はじめに

並列論理型言語GHC は、第五世代コンピュータ・プロジェクトの核言語として提案されたものである［Ueda 85］．この言語は論理型言語に並列実行の機能を持たせたものである．GHC は、論理型言語Prologと比較した場合、以下の特徴を持つ．

①構文的には、各プログラム節に必ずコミット・オペレータを持つ．コミットより前の部分をガード部、後の部分をボディ部と呼ぶ．各プログラム節のガードはそのプログラム節が採用されるための条件を表している．また、各プログラム節のボディ・ゴールは並列に動作するプロセスと考えられる．

②それぞれのゴールは、ゴール間で共有される引数、すなわち共有変数を通じて、情報のやりとりを行う．それぞれのゴールの引数が入出力のどちらとして使われるかは、陽には記述しないが、プログラマは通常、引数の使い方についてあらかじめイメージを持っている．

③これは、Prologにも共通な特徴であるが、述語は、幾つかのプログラム節によって定義され、それぞれのプログラム節は独立して定義される．また定義される述語どうしにも階層構造はなく、モジュール構造を持たない．

こうしたGHC の言語的な特徴により、GHC のプログラム・ソースの可読性は必ずしも高くない．むしろGHC は熟練者にも難解で、プログラムを一見して、内容を即座に理解することを困難にしている．

2. GHC の可読性と図形的表現

GHC のプログラム・ソースの可読性が低い原因であるが、これをGHC プログラム節の図形的表現という観点からとらえなおしてみると以下のようにまとめられよう．

［ゴールの表示］各プログラム節のゴールは、ボディ部、ガード部ともにゴール列として一次元的に表示され、それらの間の相互関係や重要性を表現できない．そこで、これらのゴール列を二次元的に図形的表現で配置することができれば、ユーザは各プログラム節の意味を、ゴール列の二次元的な位置関係として、よりヒューマン・フレンドリに把握することが出来る．

［引数、入出力モードの表示］それぞれのゴールは、ゴール間で共有される引数を通じて、情報のやりとりを行うが、ゴール間の引数の共有関係は、通常『変数名』を共有することにより示される．また、それぞれのゴールでは、引数は通常入出力のどちらかとして使われるが、GHC ではそれを陽には記述しない．そこでユーザは各プログラム節を理解する場合、引数の入出力のモードを推定することが必要となる．

したがって、例えばゴールの引数については、例えば、入力を▼、出力を△というように図形的に表現し、引数間の共有関係を、あるゴールと他のゴールを結ぶ直線として表現すれば、こうした入出力関係や共有関係を極めて簡潔に表現することができる．

［統合的環境］ GHCでは、それぞれのプログラム節は独立に定義され、プログラムはモジュール構造を持たない．こうした言語の特徴は、ユーザがプログラムを見ながら、プログラムを理解することを困難にしている．これは、GHC プログラムの図形的表現という観点からは離れるが、一つの解決方法はGHC プログラムにモジュール構造を導入することである．しかしながら、これについては、いまだよいモジュール化の方法が見出されていない．もう一つの方法として、統合的なプログラミング環境で解決する方法がある．たとえば、ある述語についてすべてのプログラム節の定義を表示したり、さらにプログラム節に表れるサブ・ゴールの定義をたどったりすることが出来れば便利である．

この［ゴールの表示］、［引数、入出力モードの表示］、［統合的環境］は、前章、①、②、③に、それぞれ対応している．これらの考察に基づき、われわれは、GHC のプログラム節の入力を視覚的に支援する統合的な環境である FE '92 (Future Environment '92)を開発した．こうしたプログラムを視覚的に表現する研究には、関数型言語で［Keller 81 ］、［Nunokawa 89 ］、［Fukamura 90 ］、またGHC に関して［Fujimura 85 ］などがあるが、それを統合的な環境として実現した例はない．

以下では、まずユーザ・インタフェイス設計のキー・コンセプトにおいて考察を行ったあと、 FE '92 の目標とする機能、技術的問題点について述べ、また、FE '92のGHC によるインプリメンテーションについて紹介する．

3. ユーザ・インタフェイス設計のキー・コンセプト

FE '92 のようなシステムはインタラクティブに実現される必要がある．こうしたインタラクティブなシステムの設計にとって核となるのは、システムのユーザとのインタフェイスをどうするかという問題である．

本システムでは、ハードウェア環境として、マルチ・ウインドウ・インタフェイスを持ち、マウス、キーボードなどを持つ標準的なワークステーション環境を想定する．われわれはこういった環境でのユーザ・インタフェイス設計のキー・コンセプトとして、MVC アーキテクチャ、因果結合およびモードレス・デザインの三つに着目する．

［MVC アーキテクチャ］ これはSmalltalk-80で導入されたユーザ・インタフェイスの構築法である［Kamiya 87 、TanakaY 90］．ここでM はモデル、V はビュー、C はコントローラの意味である．すなわち、モデルとはデータを格納するデータベースやその処理プログラムのことであり、ビュー

とは画面にデータをある見方に基づいて表示するプログラム、またコントローラとは、端末のキーボードやマウスからの入力をモデルやビューに送出するプログラムのことである．MVC アーキテクチャの特徴は、この三つの概念を区別することにより、柔軟なユーザ・インタフェイスを提供することにある．すなわち、例えば、一つのモデルに対して、デジタル表示とアナログ表示の二つのビューを対応させることも可能である．

われわれは、想定するマルチ・ウインドウ・インタフェイスのもとで、特にモデルをビューやコントローラから切り離して定義することに留意する．またビューとコントローラは、一緒に一つのウインドウとして実現する．

［因果結合］これは、リフレクションの研究［Smith 84、Tanaka 90］でよく論議される概念である．すなわち、あるオブジェクトと他のオブジェクトが因果結合しているとは、一方のオブジェクトを変更したときに、他方のオブジェクトも変更され、また逆に、他方のオブジェクトを変更したときに、一方のオブジェクトも変更される状態であることをいう．例えば、リアルタイム・システムなどにおいては、実世界と計算機の中のモデルが因果結合されている必要があるし、ユーザ・インタフェイスにおける直接操作原理においては、計算機の中のモデルとビューが因果結合されていることを要求する．

［モードレス・デザイン］使いやすいユーザ・インタフェイスの設計技法として、モードレス・デザインという考え方がある．これは『人間は、自分の属している実行モードを忘れやすいので、なるべく実行モードの階層を少なくしよう』という考え方である．たとえば、エディタについても、文書を入力したり、修正するときに、入力モードや修正モードにいちいち切り換えるのではなく、じかにテキストを打ち込んでやればそれが入力になることが望ましい．また漢字かな混じり文の入力でも、漢字モード、ひらがなモード、カタカナ・モードなどを用いるのではなく、ただテキストを打ち込めばそれが漢字かな混じり文になることが望ましい．

われわれは、このモードレス・デザインを、想定するマルチ・ウインドウ・インタフェイスのもとで、カーソルがあるウインドウや領域の中に入るとシステムのモードが自動的にその領域の持つモードに切り換わるような、自然な形で実現することを試みる．

4. FE '92の機能、実現イメージ

前章に挙げた［MVC アーキテクチャ］、［因果結合］、［モードレス・デザイン］の三つは、FE '92におけるユーザ・インタフェイスの基本思想と考えることができる．従ってFE '92の設計にあたっては、このコンセプトに基づき、2 章に挙げた、［ゴールの表示］、［引数、入出力モードの表示］、［統合的環境］を如何に実現していくかが鍵となる．

さて、FE '92の目標とする機能、実現イメージであるが、これらは、以下のようにまとめらる．

［図形による入力］FE '92では、 GHCのプログラム節を、図的表現の形式でも入力できる．プログラム節を入力するときの画面イメージを図に示したのが図 1 である．まずシステムを起動するとディスプレイ上には二つのウインドウが開く．一つはプログラム節を格納するプログラム・ビュー・ウインドウであり、これには最初はなにも書かれていない．もう一つはプログラム節の図的表現を格納するグラフィック・ビュー・ウインドウである．ユーザはこのグラフィック・ビュー・ウインドウから図形の組み合わせの形で、GHC のプログラム節を入力できる．システムを起動すると、最初に、このウインドウには、二つに区切られた矩形が表れ、その左上には述語名を入れるべき小さな矩形スペースがある．ユーザは、小さな矩形スペースに述語名を、二つに区切られた矩形の上部に定義すべきプログラム節のガード部を、矩形の下部に定義すべきプログラム節のボディ部を、それぞれ入力する．述語名はマウスで位置を選択してからキーボードにより入力し、ガード部、ボディ部はそれぞれ図形の組み合わせで入力する．ここで楕円はゴール、▼は入力、△は出力をそれぞれ意味するので、マウスによって図形を思いのままに入力し、共有される引数は直線で結んでいけばよい．

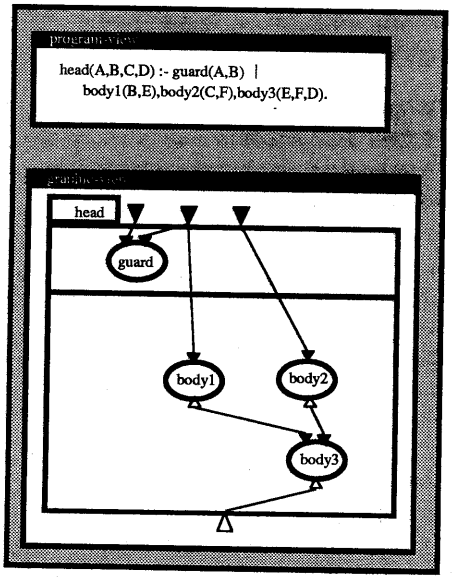

図 1　FE '92の画面イメージ

［プログラム・コードの自動作成］こうしてプログラム節を、図的表現の形式で入力すると、それに応じて、プログラム・コードは、プログラム・ビュー・ウインドウに自動的に作成され、表示される．いつ、図的表現からコードへの変換を行うかというタイミングについては、因果結合の考え方を用い、グラフィック・ビュー・ウインドウに変化があると、常に変換のトリガーがかかり、再計算がおこなわれ、プログラム・ビュー・ウインドウには、常に図的表現に対応する最新のコードが示される．

［図形表現の自動作成］逆にプログラム・コードをプログラム・ビュー・ウインドウから入力し、グラフィック・ビュー・ウインドウに図的表現を自動的に作成することも出来る．この場合、グラフィック・ビュー・ウインドウの中のガード部やボディ部におけるゴールの配置などは、あるアルゴリズムに従って自動的に定められる．

［修正機能］入力されたプログラム・コードや図的表現は、それをあとで修正することができる．この機能を用いれば、自動作成された図的表現を修正し、例えばゴールの位置を移動して、ユーザのイメージどうりの配置にすることもできる．また既に定義したプログラム節の引数の数を変更したり、ガード部やボディ部のゴールを変更したりすることもできる．

［保存・再生機能］プログラム・コードと図的表現は、それをファイルに保存し、必要なときに再生することができる．図的表現については、引数、ゴールの位置を座標点の位置で記憶する．プログラム・コードと図的表現は一緒に保存される．

［統合的環境］FE '92は、GHC プログラムの作成、理解、修正を支援するような統合的なプログラミング環境でなければならない．本システムでは、一つのプログラム節だけでなく、ある述語に関するプログラム節をすべて一緒に扱うことが可能であり、それらはプログラムの作成、理解、修正の過程を統合的に扱うことを可能する．また、ある述語を定義している途中に、ほかの述語をついでに定義したり、参照したりしたいような事態はしばしば起こるが、本システムでは、ある述語の定義中に、ほかのサブ・ゴールの述語定義を行ったり、また元に戻ったりすることが可能である必要がある．

これらからもあきらかなように、本システムでは、プログラム・コードと図形的な表現が、双方併用される．ここで、すべてを図形的な表現のみで扱わないのは、図形的な表現とプログラム・コードはそれぞれ長短があるからである．すなわち、可読性では二次元的に情報を持つ図形的な表現がまさるが、一般には、プログラム・コードの表現のほうがより簡潔であり、複雑なデータ構造を表す場合には、プログラム・コードの方が優る場合が多い．

これをMVC アーキテクチャという観点から述べれば、本システムは、モデルにGHC のプログラム節を格納し、それを二つのビュー、すなわち、プログラム・ビューとグラフィック・ビューの二つで表示するシステムである．それぞれのビューには、それぞれコントローラが組み込まれており、ユーザはこのプログラム・ビューとグラフィック・ビューのどちらからでも、GHC のプログラム節を入力することができる．またモデルと二つのビューは互いに因果結合されているので、どちらから入力をしても、それらはすぐにモデルに反映され、それはただちにもう一方のビューにも反映される．

5. 技術的問題点

前章に示したようなFE '92を実現しようとすると、幾つかの技術的な問題点をクリヤーする必要がある．以下に、FE '92実現のための技術的な問題点と、それに対するわれわれの解決策をまとめる．

5.1 入出力モードの問題

ここでは、実行ゴールやGHC プログラムの入出力のモードについて考える．まず、ゴールについては、それがシステム定義述語である場合とユーザ定義述語である場合の二とおりがある．システム定義述語の場合、引数の入出力のモードは既に定まっている、と考える．例えば、plus(X,Y,Z) というゴールであれば、第一引数と第二引数は入力、第三引数は出力である．

また、ユーザ定義述語の場合、その引数の入出力のモードは、ユーザが定義したプログラムによって決まる．たとえば、いまmulti という述語を以下のように定義する．

```
multi(X,Y,Z) :-  X = 1 | Z = X.
multi(X,Y,Z) :-  X≠1 |
        Y1 = Y - 1, multi(X,Y1,Z1), Z = Z1 + X.
```

ただし、ここでY1 = Y - 1は minus(Y,1,Y1)の、Z = Z1 + Xは plus(Z1,X,Z) のそれぞれ略記である．minus やplusの場合、第一引数と第二引数は入力、第三引数は出力であるので、述語multi は、第一引数と第二引数が入力、第三引数が出力になる．

一般に、ゴールの引数の入力、出力については、いろいろな定義の仕方があるが、ここでは以下のように考える．

①入力引数とは、ゴールが実行される時に、変数でなく、すくなくともトップレベルは具体化されている引数のことである．
②出力引数とは、ゴールの実行が開始される時には変数であり、ゴールの実行が終了したときには、すくなくともトップレベルは具体化されている引数のことである．

しかしながら、こういった手続きにより、GHC プログラム

のゴールの引数の入出力のモードを完全に決定できるかというとそうではない．例えば、

```
goal(X) :-  cond(X) | compute(X).
```

という、プログラム節を考えてみると、X はガード部分に表れるので入力のようにも思われるし、ボディ部をみるとX は計算結果を格納する出力のようにも思われる．また、

```
goal(H,T) :-  H = T | H =  [] .
```

の場合も、H は入力のようであり、同時に出力のようにも見える．このように、一般に、GHC プログラムでは、ゴールの引数の入出力のモードを完全に決定することは出来ない．
一般に、ユーザはゴールの引数について、通常は、ある思い入れをもってプログラムを書くのが普通であり、図形表示においても、ゴールの引数の入出力のモードがあったほうが、より理解が容易になると考えられる．そこで、FE '92では、ゴールの引数の入出力モードについては、『ユーザが自由に指定出来る付加情報』であると定義することにした．すなわち、述語やゴールの引数の入出力モードを、プログラム・ビュー、グラフィック・ビューに関して、以下のように指定する．

[プログラム・ビュー]
述語やゴールの引数を必ず入力、出力の順に書くことにして、その間にはダミーの引数 ":"をはさむ．例えば、

```
append( [A | X ] ,Y,Z) :-  true |
        append(X,Y,Z1), Z= [A | Z1] .
```

であれば、

```
append( [A | X ] ,Y,:,Z) :-  true |
        append(X,Y,:,Z1), Z= [A | Z1] .
```

と表現するようにする．（図1で示した画面イメージでは、説明の都合上 ":"を省いているが、実際にはプログラム・ビューに表示されるプログラム節は ":"を含む．）

[グラフィック・ビュー]
前述したように、図形表示では、▼は入力、△は出力を意味する．述語の引数の場合、入力は全体の矩形の上部に、出力は矩形の下部に、これら引数を表す三角形を置く．またゴールの引数の場合も、入力はゴールを表す楕円の上部に、出力は楕円の下部に、これら引数を表す三角形を置く．

ここでは、述語やゴールの引数の『入出力モード』はプログラムの図的表示のための付加情報であり、プログラム実行では無視される点に留意したい．すなわち、ユーザは自分の思い入れどうりに入出力を指定でき、そのモードは実行時には、何ら影響をおよぼさない．

5.2 変数名の扱い
一般に、プログラム・ビューでは、プログラム節の定義で変数名が使用されるが、グラフィック・ビューでは、変数名という概念はいままでのところ使用しなかった．すなわち、プログラム・ビューでは変数名を共有することにより、情報の流れを示したが、グラフィック・ビューでは、それを直接、引数から引数への直線として表現するので変数名の必要がなかったのである．
しかしながら、一般にはプログラム：コードの変数名は、その変数についての思い入れを含め名称をつける場合もある．例えば、リストとして使われる変数をListと命名したり、重リストとして使われる対の変数をHead, Tailと命名したりする場合である．したがってFE '92では、二種類の変数、すなわち無名変数、名前付き変数、を用意する．ここで無名変数とは変数名が重要でない変数のことで、システムが自動的に変数名を管理し、変数名は保存されない．また名前付き変数とは変数名が重要な変数のことで、システムは変数名を保存する．
無名変数は、変数名が大文字だけ、もしくは大文字と数字だけからなる文字列で表現される．また名前付き変数は、変数名に小文字を含む文字列で表現される．無名変数、名前付き変数ともに、通常のGHC と同様、変数名は大文字で始まる必要がある．
FE '92では、この二種類の変数がうまく使い分けられている．すなわち、ユーザが最初にグラフィック・ビューから図形入力をした場合、とくに変数名の指定を行わなければ、プログラム・ビューに表示されるプログラム節の変数にはA,B,C,... といった無名変数が自動的に割り当てられる．またプログラム・ビューからプログラム節の入力を行った場合、グラフィック・ビューでは無名変数は変数名を表示せず、名前付き変数の変数名のみを表示する．またプログラム・ビューのプログラム節の表示も、無名変数は、ユーザが指定したものでなく、システムは自動合成したものに置き換わる．

5.3 構成子、引数束縛、ユニフィケーションの扱い
図形表示では、引数は▼や△などの記号、ゴールはシステム定義述語、ユーザ定義述語ともに楕円で表される．また、関数記号、リスト構成オペレータについても楕円で示される．本システムでは、リスト構成オペレータとして、図2 に示すよう、リストを構成するものと分解するものの二種類を用意している．
次に引数であるが、プログラム節の定義においては引数は変数であるとは限らず、定数や関数記号などで束縛されている可能性がある．この場合、『引数はすべて変数で受けることにして、あとはユニフィケーションで表現する』、たと

えば、

```
append( [A | X ] ,Y,Z) :-  true |
        append(X,Y,Z1), Z=  [A | Z1] .
```

であれば、

```
append(A1,Y,Z) :-  A1=  [A | X ] |
        append(X,Y,Z1), Z=  [A | Z1] .
```

と解釈するのも可能ではあるが、その場合、図形表示は複雑になる．そこで、FE '92では、引数は、直接、定数や関数記号などで束縛することができるものとする．

図3 に、引数への束縛の例を示す．ここでは、引数に基低項(ground term) が束縛した場合には、引数を示す▼や△などの記号の左横に、その基低項を示し、構成子(list constructor)や関数記号が変数を含む場合には、その構成子や関数記号を含む楕円で束縛を示す．図2 の左は、入力変数が [] に束縛されている例である．右は"X = [true | Y] "に対応する図形表現である．（ここでは、出力引数がtrueに束縛されているが、入力と出力の違いは、あくまで便宜的なものであることに注意されたい.）

ユニフィケーションの扱いであるが、FE '92では、ユニフィケーションを陽に扱うことはない．二つの引数を結べばそれがユニフィケーションを表すことになる．

また、前にも述べたが、例えば、Z = X + Y は plus(X,Y,Z)の略記と考えるし、同様に、Z= [X | Y] のようにユニフィケーションの方向がはっきりしているときには、引数の束縛と考える．

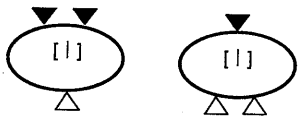

図2　リスト構成オペレータ

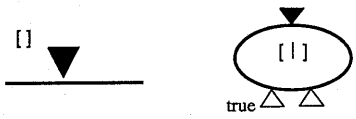

図3　引数束縛の例

6.　GHC によるFE '92のインプリメンテーション

これらの考察に基づいて、並列論理型言語GHC を用い、FE '92のインプリメンテーションを行った．実際のシステムは、かなり複雑なものであるので、本論文では、FE '92の概略を示すため、簡単化したコードの上位の記述のみを示すことにする．（実際のシステムの詳細については [Tanaka 90b] を参考にされたい.）

まず、FE '92のトップレベルの記述であるが、これは以下のように書けよう．

```
fe92( [make | In] ,Out) :- true |
     graphic__view(Gin,Gout),
     program__view(Pin,Pout),
     merge( [Gout,Pout] ,Min),
     model__server(Min,Gin,Pin, [] ),
     fe92(In,Out).
```

すなわち、『fe92』述語は二つの引数を持っており、最初の引数はシステムからFE '92への入力、二つ目の引数は、FE '92からシステムへの出力である．（二つ目の引数の用途は、この簡単化したコードでは、省略されているが、終了処理、エラー処理などに使われる.）

まず、入力引数にmakeというメッセージを入力すると、二つのビュー『graphic__view』、『program__view』と、モデル『model__server』が起動される．ここで二つのビューは、いずれも二つの引数を持っている．最初の引数はビューへの入力、二つ目の引数は、ビューからの出力である．ビューとモデルとは、双方向に入出力が結合されており、二つのビューからの出力GoutとPoutは、merge で結合されてモデルへの入力となる．

graphic__viewは、以下のように記述される．

```
graphic__view(Gin,Gout):- true |
     create__window(In,Out),
     g__controller(In,Gout, [] ,write),
     g__view(Gin,Out).
```

すなわち、graphic__viewは、まず『create__window』でウインドウを開く．ここでcreate__windowのInには、ユーザからウインドウへ入力されたデータが流れてくる．またcreate__windowのOut には、ウインドウへ表示したいデータを流す必要がある．

そこで、ウインドウからの入力Inについては、『g__controller』で変換してGoutに格納し、graphic__viewからの出力として外に送られる．またgraphic__viewへの入力Gin については、『g__view』で変換してからウインドウの出力Out に送るようにする．

g__controllerは4 つの引数を持っているが、三番目の引数は、前の入力データのバッファとして、最後の引数はシス

テムの現在の状態(writeモード、linkモードなど) を示すのに使われる.

```
g__controller( [click(r,__) | In] ,
    Gout,Buf,__):- true |
    %%%% r==mode change
    show__menu(Menu),
    read__new__mode(Menu,Mode),
    close__menu(Menu),
    g__controller(In,Gout, [] ,Mode).

g__controller( [click(Type,Position) | In] ,
    Gout,Buf,Mode):- Type≠r |
    mouse__click__parser(Mode,Type,Position,
       Buf,Buf1,Command),
    send(Command,Gout,Gout1),
    g__controller(In,Gout1,Buf1,Mode).
```

このコードでもあきらかなように、g__controllerは、ウインドウからの (マウス) 入力が、モードの変更を示す右クリックであれば、メニューを開き、変わるべきモードを読み込んで、それで指定されたモードになる. また、ウインドウからの (マウス) 入力が、右クリック以外であれば、(場合によってはバッファの情報を使い) それをコマンドに変換してからgraphic__viewの出力に送るようにする.

反対に、『g__view』はgraphic__viewへの入力Gin をウインドウの画面出力用のコントロール・メッセージの列に変換し、ウインドウの出力Out に送る機能を持っている.

```
g__view( [Msg | Gin ] ,Out):-
    Msg=write(Type,Position,Attribute) |
    g__translate(Msg,Gmsgs),
    send(Gmsgs,Out,Out1),
    g__view(Gin,Out1).

g__view( [Msg | Gin ] ,Out):-
    Msg=remove(Type,Position,Attribute) |
    g__translate(Msg,Gmsgs),
    send(Gmsgs,Out,Out1),
    g__view(Gin,Out1).

g__view( [Msg | Gin ] ,Out):-
    Msg=move(Type,Position1,Position2,Attribute) |
    Msg1=remove(Type,Position1,Attribute),
    g__translate(Msg1,Gmsgs1),
    send(Gmsgs1,Out,Out1),
    Msg2=write(Type,Position2,Attribute),
    g__translate(Msg2,Gmsgs2),
    send(Gmsgs2,Out1,Out2),
    g__view(Gin,Out2).
```

このg__viewの三つの定義節は、いずれも同じような処理を行っているが、三番目の定義節では、moveという命令を、removeとwrite に分解してからコントロール・メッセージの列に変換していることに注意されたい.

次に、program__viewは、以下のように記述される.

```
program__view(Pin,Pout):- true |
    create__emacs __window(In,Out),
    p__controller(In,Pout),
    p__view(Pin,Out).
```

program__viewは、まず『create__emacs__window』でテキストの入出力に適したウインドウを開く. ウインドウからの入力Inについては、p__controllerを経由してprogram__viewの出力Poutに送る. またprogram__viewの入力Pin については、p__viewでウインドウの画面出力用のコントロール・メッセージの列に変換し、ウインドウの出力Out に送る.

```
p__controller( [term(Clause) | In] ,Pout):- true |
    Pout= [change(Clause) | Pout1 ] ,
    p__controller(In,Pout1).

p__view( [display(Clause) | Pin ] ,Out):- true |
    p__translate(display(Clause),Pmsgs),
    send(Pmsgs,Out,Out1),
    p__view(Pin,Out1).
```

次に、モデルであるが、『model__server』は、4 つの引数を持っており、第1 引数は入力、第2 引数は、グラフィック・ビューへの出力、第3 引数は、プログラム・ビューへの出力、第4 引数は、モデルを格納するデータベースとなっている. すなわち、『model__server』は、グラフィック・ビューやプログラム・ビューからの情報を第1 引数で受け取り、第4 引数のデータベースを更新したあと、必要に応じて、第2 引数や第3 引数を使って、グラフィック・ビューやプログラム・ビューへ状態変更メッセージを送る.

まずmodel__serverの、第1 引数に、ある位置に図形を書けというwrite メッセージが来た時の処理は以下のように記述できる.

```
model__server( [write(Position) | In] ,
    Gin,Pin,Map) :- true |
    decide__component(Position,Map,Type),
    create__component(Type,Position,Input,
        Output,Gin1),
    register__map(Type,Input,Map,Map1),
    collect__code(In1,In2,Map1,Map2,Pin1),
```

```
    merge( [Output,In] ,In1),
    merge( [Gin1,Gin2] ,Gin),
    merge( [Pin1,Pin2] ,Pin),
    model__server(In2,Gin2,Pin2,Map2).
```

最初にdecide__component 述語を実行し、書くべき図形が何であるかのタイプ(すなわち、ゴール、入力引数、出力引数のどれであるか) を決める. 前述したように、本システムはモードレス・デザインをめざしており、ユーザによりポイントされた位置と、これまで組み立てた図形のデータベースよりこれは完全に決定できる. 次にcreate__component により、指定された部品が作られ (この部品は入力変数Input と出力変数Outputを持っている.) 、それと同時にGin1を経由して、グラフィック・ビューへ描かれる. register__map により、作られた部品は、そのタイプと入力変数がmap に登録される. また、こうして入力された図形を含む最終的なデータベースMap1からテキスト・コードを作り、プログラム・ビューに出力するのがcollect__codeである.

同様に、model__serverの、第1 引数にlink、moveなどのメッセージが来た時の処理は以下のように記述できる.

```
model__server( [link(Position1,Position2) | In] ,
    Gin,Pin,Map):- true |
    send(Position1,Map,Map1,link(LinkV)),
    send(Position2,Map1,Map2,link(LinkV)),
    create__component(link,(Position1,
        Position2),Input,Output,Gin1),
    register__map(link,(Position1,Position2),
        Map2,Map3),
    collect__code(In1,In2,Map4,Map5,Pin1),
    merge( [Output,In] ,In1),
    merge( [Gin1,Gin2] ,Gin),
    merge( [Pin1,Pin2] ,Pin),
    model__server(In2,Gin2,Pin2,Map5).

model__server( [move(Position1,Position2) | In] ,
    Gin,Pin,Map):- true |
    send(Position1,Map,Map1,move(Position2)),
    collect__code(In,In1,Map1,Map2,Pin1),
    merge( [Gin1,Gin2] ,Gin),
    merge( [Pin1,Pin2] ,Pin),
    model__server(In2,Gin2,Pin2,Map2).
```

ここで、指定されたタイプの部品をプロセスとして作るのがcreate__component 述語である. ここでは入力引数が作られるときのcreate-componentの定義と、作られる部品に相当するarg の定義を以下に記述する.

```
create-component(input__arg,Position,
    Input,Output,Gin):- true |
    Gin= [write(arg,Position,input) | Gin1]
    arg(Input,Output,Position,Value,input,Gin1).

arg( [move(Position2) | Input ] ,Output,Position1,
    Value,Attribute,Gin):- true |
    Gin= [move(arg,(Position1,Position2))
      | Gin1] ,
    arg(Input,Output,Position2,Value,Attribute,Gin1).

arg( [link(LinkV) | Input ] ,Output,Position,
    Value,Attribute,Gin):- true |
    LinkV=Value,
    arg(Input,Output,Position,Value,Attribute,Gin1).

arg( [collect__code(Done,Done1) | Input ] ,Output,
    Position,Value,Attribute,Gin):- true |
    Output= [send(goal,arg(Position,
        Value,Attribute,Done,Done1) | Output1 ] ,
    arg(Input,Output1,Position,Value,Attribute,Gin1).
```

作られた部品であるarg は入力変数Input と出力変数Outputを持っているプロセスで、Input からメッセージを入力すると自己の内部状態を変更したり、答えをOutputから出力したりする. arg の最後の定義節は、collect__code述語の実行のために必要である.

次にcollect__code述語であるが、これはInとMap を与えられ、これにメッセージを流すことによって、現在の図形入力に対応するコードをPin に出力する述語である. 計算の過程で、InとMap を使用するので、それらは計算終了後にはNewIn とNewMapに更新される. 以下にcollect __code述語の定義を示す.

```
collect__code(In,NewIn,Map,NewMap,Pin)
    :-true |
    send__collect__arg(Map,Map1,Flag),
    collect__arg(Flag,In,NewIn,Map1,NewMap,Pin).

collect__arg(end,In,NewIn,Map,NewMap,Pin)
    :-true |
    send__collect__guard(Map,Map1,Flag1),
    collect__guard(Flag1,In,NewIn,Map1,NewMap,Pin).
collect__arg(Flag, [send(Type,Position,Msg) | In] ,
    NewIn,Map,NewMap,Pin):-true |
    send(Type,Position,Msg,Map,Map1),
    collect__arg(Flag,In,NewIn,Map1,NewMap,Pin).

collect__guard(end,In,NewIn,Map,NewMap,Pin)
    :-true |
```

```
    send__collect__body(Map,Map1,Flag2),
    collect__body(Flag2,In,NewIn,Map1,NewMap,Pin).
collect__guard(Flag1, [send(Type,Position,Msg)
    | In] ,NewIn,Map,NewMap,Pin):-true |
    send(Type,Position,Msg,Map,Map1),
    collect__guard(Flag1,In,NewIn,Map1,NewMap,Pin).

collect__body(end,In,NewIn,Map,NewMap,Pin)
    :-true |
    send__collect__clause(Map,Map1,Head),
    collect__clause(Head,In,NewIn,Map1,NewMap,Pin).
collect__body(Flag2, [send(Type,Position,Msg)
    | In] , NewIn,Map,NewMap,Pin):-true |
    send(Type,Position,Msg,Map,Map1),
    collect__body(Flag2,In,NewIn,Map1,NewMap,Pin).

collect__clause(clause(Clause),In,NewIn,
    Map,NewMap,Pin):-true |
    Pin= [display__clause(Clause)] ,
    NewIn=In,
    NewMap=Map.
```

この定義でわかるように、collect __code述語は、図形に対応するコードを（入出力）引数⇨ガード・ゴール⇨ボディ・ゴール⇨クローズの順で構成する．これは、send__collect__xxxxという述語により、それぞれの部品であるプロセスに、コード回収メッセージcollect__codeを流すことにより得られる．　次にmodel__serverの第1 引数に、draw(Clause)という、コードから図形を書くメッセージが来た時の処理であるが、これは以下のように記述できる．

```
model__server( [draw(Clause) | In] ,
    Gin,Pin,Map) :- true |
    draw__graph(Clause,In,In1,Gin1,
        Map,Map1),
    collect__code(In1,In2,Map1,Map2,Pin1),
    merge( [Gin1,Gin2] ,Gin),
    merge( [Pin1,Pin2] ,Pin),
    model__server(In2,Gin2,Pin2,Map2).

draw__graph(Head :- GuardL | BodyL),In,NewIn,
    Gin1,Map,NewMap) :- true |
    draw__head(Head,In,In1,Map,Map1,Gin1),
    draw__guard(GuardL,In1,In2,Map2,Map3,Gin2),
    draw__body(BodyL,In2,NewIn,Map3,NewMap,Gin3),
    merge( [Gin1,Gin2,Gin3 ] ,Gin).
```

こうした記述からも明らかなように、GHC のプログラムは、プロセスとストリームを多用したプログラムとなっている．こうしたプログラミング・スタイルは [Tanaka 88] などのシステム・プログラミングでも使われた典型的なGHC のプログラミング・スタイルである．

7.　まとめ

われわれは、並列論理型言語GHC のプログラム入力を、視覚的、図形的に支援するシステム FE '92 (Future Environment '92)を開発した．実際のFE '92でappendプログラムを入力したときの入力例を図4 に示す．

本論文では、まず、GHC の言語的特徴に基づいて、GHC の可読性と図形表現について考察をおこなった．次にユーザ・インタフェイス設計のキー・コンセプトにおいて考察を行ったあと、FE '92の目標とする機能、技術的問題点について述べた．本システムは、GHC のプログラム節の入力において、図形的な表現を併用することができ、比較的難解とされる並列論理型言語GHC のプログラムを視覚的に表現し、編集することができる．

また、FE '92のGHC によるインプリメンテーションについても、簡単なコードの概略を紹介した．このプログラムは、GHC の典型的なプログラミング・パラダイムであるプロセスとストリームを多用したプログラムとなっており、これにより、ユーザとのマルチ・ウインドウによるインタフェイスが、極めて自然な形で記述されている．

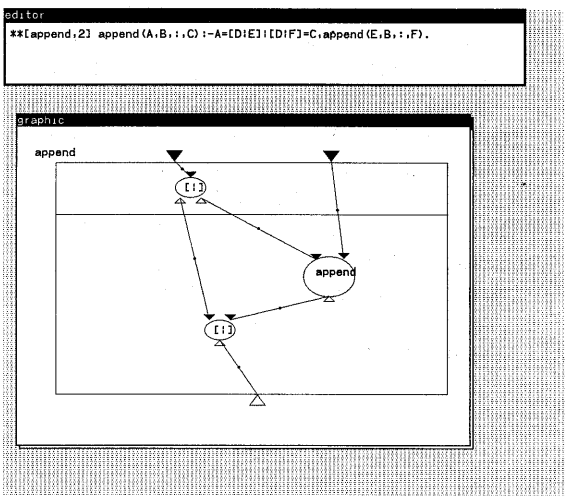

図4　FE '92 によるappendプログラムの入力例

［謝辞］

本研究は第5 世代コンピュータ・プロジェクトの一環として行われたものである．本研究に関連して日頃ディスカッションの相手になってくれる国際研の同僚である神田陽治、前田宗則の両氏に感謝する．また富士通SSL の的野文夫氏に感謝する．

［参考文献］

[Fujimura 85] 藤村考、栗原正仁、加地郁夫: データフローに基づく並列論理型言語のソフトウェア設計、日本ソフトウェア科学会第二回論文集、pp.141-144, 1985.

[Futamura 90] 二村良彦、野木兼六、高野明彦: ラムダ図、ラムダ式の図的表現、bit 、vol.22, No.12, pp.11-19, 1990.

[Kamiya 87] 上谷晃弘編著: 統合化プログラミング環境—Smalltalk-80とInterlisp-D —, 丸善, 1987.

[Keller 81] R.M. Keller: Applications of Feedback in Functional Programming, 1981 Conference on Functional Programming and Computer Architecture, ACM, pp.123-130, 1981.

[Nunokawa 89] 布川博士、富樫敦、野口正一: 図式をシンタックスに持つ関数型言語、コンピュータ・ソフトウェア、Vol.6, No.2, pp.135-147, 1989.

[Smith 84] B.C. Smith: Reflection and Semantics in Lisp, 11th. POPL, Salt Lake City, Utah, pp.23-35, 1984.

[Tanaka 88] J. Tanaka: A Simple Programming System Written in GHC and Its Reflective Operations, The Logic Programming Conference '88, ICOT, pp.143-149, 1988.

[Tanaka 90a] J. Tanaka, Y. Ohta, F. Matono: An Overview of Experimental Reflective Programming System: ExReps, Fujitsu Scientific and Technical Journal, Vol.26, No.1, pp.86-97, 1990.

[Tanaka 90b] 田中二郎: メタ・リフレクション実験環境: FE '92、第5 世代コンピュータに関するシンポジウム、デモンストレーション資料、ICOT、1990.6.12-13.

[Ueda 85] K. Ueda: Guarded Horn Clauses, ICOT Technical Report TR-103, 1985.

[TanakaY 90] 田中譲: IntelligentPad—シンセティックダイナミックメディア—, Computer Today, pp.45-54, No.38, 1990.7.