筑波大学大学院博士課程

工学研究科修士論文

ビジュアルプログラミングシステムにおける
表現のカスタマイズ機能の実現

電子・情報工学専攻

著者氏名　　小川 徹

指導教官　　田中 二郎　　印

平成 12 年 2 月

要旨

ビジュアルプログラムにおいて、プログラムの視覚的な表現はシステムの仕様によって与えられる。一般的に、システムが与える表現はノードやエッジという単純な構造である。 プログラミングシステムの中で単純な表現を用いた場合、それが何を表しているのかを判断する手掛かりは、図形の形・構造ではなく、図形の中に書かれたラベルである。プログラムの意味が図形の形・構造に反映されていないため、プログラムを理解するには時間がかかる。形・構造という図形の特徴からプログラムを認識するほうが、よりインタラクティブなプログラミングが行えるはずである。プログラムが人によって様々な解釈をされるように、一つのプログラムに対する視覚化にも様々な表現が考えられる。表現が自由に変更可能なプログラミングシステムが望まれる。

本論文では、まず、 ビジュアルプログラミングシステム “CafePie” について述べる。 CafePie において、プログラムで使用されるオブジェクトはビジュアルな表現によって視覚化され、また、同じ表現を用いて実行の表示が行われる。プログラム上で視覚化されたオブジェクトは、ドラッグ アンド ドロップという直接操作を用いて編集される。本研究では、ビジュアルプログラミングシステムの特徴である「プログラム構造を直視することで認識し、直観的に理解することができる」ことを生かすために、表現のカスタマイズ機能について検討を行った。 さらに、 表現のカスタマイズ機能の実現方法を考察し、 CafePie 上に試作した。このカスタマイズ機能を利用することによって、図形的な意味をプログラムに反映することが可能となる。本システムでは、一つのプログラムに対して様々な表現が可能である。

目 次

1 はじめに **7**

2 準備 **9**

2.1 ビジュアルプログラミング 9

2.1.1 ビジュアルプログラミングの分類 9

2.1.2 実装におけるベース言語 10

2.2 代数的仕様記述言語 CafeOBJ 12

2.2.1 CafeOBJ のプログラム記述 12

2.2.2 CafeOBJ 言語におけるプログラム実行 15

3 ビジュアルプログラミングシステム "CafePie" **20**

3.1 システムによるプログラムの視覚化 22

3.2 ビジュアルプログラミングシステムにおける編集方法 24

3.2.1 直接操作 24

3.2.2 ドラッグ アンド ドロップ手法 25

3.2.3 ドラッグ アンド ドロップを用いたプログラムの編集 26

3.3 ビジュアルプログラミングシステムにおける実行表示 29

4 視覚化カスタマイズ機能 **32**

4.1 ユーザインタフェースの構築モデル 34

4.1.1 MVC モデル 34

4.1.2 プラガブル MVC モデル 35

4.2 ビジュアルプログラミングシステムにおけるカスタマイズ機能 . . . 36

4.2.1 ユーザの視点から見たカスタマイズ機能 37

4.2.2 書換えルールの編集における入力手法の強化 38

4.2.3 図形書換えルールの定義例 38

4.3 カスタマイズ機能の応用 41

4.3.1 ビューの省略表示の定義 43
4.3.2 ビューの切替え機能 44

5 関連研究 46

5.1 アルゴリズムアニメーション 46
5.2 ビジュアルプログラミングシステム 46

6 おわりに 48

謝辞 49

図一覧

2.1 CafeOBJ 言語による仕様記述例 13
2.2 CafeOBJ インタプリタによる仕様記述の実行 16
2.3 CafeOBJ トレーサによるトレース表示 18

3.1 CafePie の実行画面 21
3.2 CafePie におけるソートの視覚化 23
3.3 CafePie における項の表現 23
3.4 CafePie における演算の視覚化 23
3.5 CafePie における変数の視覚化 24
3.6 CafePie における等式の視覚化 24
3.7 ドラッグ アンド ドロップ手法による項の作成手順 28
3.8 インタプリタの URL 指定用ダイアログ 30
3.9 CafePie 上での実行の動的表示 30
3.10 CafePie 上での実行の静的表示 30

4.1 システムによるスタックの視覚化 32
4.2 CafePie によるスタック仕様の視覚化 33
4.3 MVC モデル 34
4.4 図形書換えルールの編集 37
4.5 演算 empty へのビューの定義 39
4.6 演算 push へのビューの定義 39
4.7 演算 push における図形書換えルールの編集 39
4.8 ドラッグ アンド ドロップ手法を用いた 2 つの図形間の編集 40
4.9 カスタマイズ後のスタックの視覚化 40
4.10 演算 pop と等式の視覚化 41
4.11 演算 empty へのビューの定義 その 2 41
4.12 演算 push へのビューの定義 その 2 42

4.13 カスタマイズ後のスタックの視覚化 その 2 42
4.14 スタックの要素／顔の表情 42
4.15 演算 empty の省略記法による視覚化 43
4.16 演算 push の省略記法による視覚化 43
4.17 ビューの切替え機能 44

第 1 章

はじめに

計算機の処理能力が飛躍的に向上しており、以前から用いられてきた伝統的なテキスト表現に代わってビジュアル表現が多く用いられるようになった。ビジュアル表現とは、アイコンという実世界にあるオブジェクトを抽象化して表現する小さなビットマップから OMT[1] のオブジェクト図などのように特定の用途に使われる図まで様々なものを指す。このような背景のもとで、 Windows や Macintosh のユーザインターフェースに代表されるように、アイコンやアニメーションなど人間がより理解しやすいビジュアルな表現を用いて人とコンピュータとのインタラクションが行われるようになってきた。

このような表現を用いてプログラミング処理を行う**ビジュアルプログラミング**が注目されている。これは「図形・アイコンなどの視覚的表現を用いてプログラムを表し、またそれらを操作することによってプログラムの編集や実行を行うための環境」として定義される [2, 3]。ビジュアルプログラミングシステムの特徴の一つは「構成要素が図形で表現されており、テキスト表現と比べてプログラム構造を直視することで認識し、直観的理解することができる」である。しかし、従来のビジュアルプログラミングシステムでは、システムが与えた楕円・長方形・線というような単純な表現であったり、ユーザが与えた単純なアイコンのみでしか表現できないなど、何かしらの制限があるという問題があった [4, 5]。システムが与える表現だけでは限界があるし、ユーザが毎回アイコンを容易するのは面倒である。

一般に、 1 つのプログラムの視覚化手法は必ずしも 1 つとは限らず、それを人がどのように認知するかによっても様々な視覚化が考えられる。例えば、日常で良く見掛ける交通信号は、今でこそ「青・黄・赤」という光信号が当たり前であるが、光信号がない場合には手信号のように人の身振りでも表現することができる。手信号は慣れないと戸惑うかもしれないが、手の向きなどで進むべき方向を指示してくれるため、色という人種などによって印象が異なる方法と比べるとある意味自然で

ある。その対象物に内在する本質的な機能とは別に、使用する場面や用途に応じてビジュアルに見える表現も変更する方がより自然であると考える。

我々は「プログラム構造を直視することで認識し、直観的理解することができる」というビジュアルプログラミングの特徴を生かすために、プログラム表現に図形的意味を与えられる仕組みが必要であると考えた。これはビジュアル表現のカスタマイズ機能で実現される。このカスタマイズ機能を利用することによって、図形的意味はプログラムに反映される。これにより、ユーザがプログラムを使用する場面に応じて、一つのプログラム表現に対し様々に視覚化部分を変更することができる。

本論文の構成は次のようになる。まず、第2章において本論文で必要となる知識の準備を行う。次に、第3章では上記の提案に基づいて実装したシステム CafePie について例を挙げながら説明し、第4章でビジュアル表現のカスタマイズ機能について述べる。第5章で関連研究について記述し、第6章でまとめる。

第 2 章

準備

本章では、まず、本論文を通して重要なキーワードとなるビジュアルプログラミングについて述べる。その後で、我々のシステムのベース言語／ CafeOBJ について簡単に説明する。

2.1 ビジュアルプログラミング

ビジュアルプログラミング (VP) は研究者によって様々な捉え方をされる。一般に、事物を表現するには図が最適であることが多く、回路図や設計図、地図などのように工学の世界では「図」を用いられることが多い。百聞は一見に如かず と古来より伝えられているとおり、具体的な事物を扱う計算機プログラムでは図的表現がテキスト表現に勝ることが多いし、抽象的な計算を行う場合でも式や関係などを図で表現した方が都合が良い場合が多くある。このような考えが VP の原点であると考えている。我々が定義する VP とは、**複雑な事柄をシンボル化し、コンピュータにより直接操作できるものに置換える技術**のことである。これは、従来の文字列や記号を中心としたインターフェースに代わって、アイコン／図形／アニメーション、という人間がより理解しやすいビジュアルな表現を用いて人間とコンピュータの相互作用を行なう技術のことであり、最近急速に世間の関心が高まっている。

2.1.1 ビジュアルプログラミングの分類

市川らによると、 VP は**ソフトウェアの視覚化**、**アルゴリズムアニメーション**、**グラフィカルプログラミング**の 3 つに分野に分類される [6, 7]。ソフトウェアの視覚化とは、その言葉通り、ソフトウェアを視覚化することである。ソフトウェアには、要求仕様、ソフトウェア設計、コーディング、テスト、運用、廃棄などのフェイズからなるライフサイクルがある。この各々のフェイズに対して視覚化が考えら

れる。アルゴリズムアニメーションは、プログラムの動作を与えるアルゴリズムについて、アニメーションを用いて抽象度の高い視覚化を行う。グラフィカルプログラミングは、最初にプログラミングを行うときから、テキストではなく図形を組合わせる形でプログラミングを行おうとするものである。我々の立場としては、グラフィカルプログラミングが最も近い。なお、市川の分類には含まれないが、最近、流行しているものとしてインターフェースの視覚化がある。これは、アプリケーションプログラムのインタフェース環境をユーザが視覚的に構築できるように従来のテキストプログラミングに取り入れた手法である。 Visual Basic や Visual C++ などのいわゆるインタフェースビルダと呼ばれるものがこれに含まれる。これは、我々が目指すビジュアルプログラミングとは全く異なるものである。

また、 VP の分類として、特に良く引用されるのが Myers の分類 [3] である。 Myers は、まず、従来において視覚化プログラム (Visual Programming) とプログラムの視覚化 (Program Visualization) との区別が曖昧であったことを指摘し、これら 2 つの総称として Visual Languages という言葉を提唱している。 Myers は分類基準として「例示」によるシステムかどうかということと「インタプリタかまたはコンパイラベースか」ということを挙げている。 1 つめの基準にある例示とは、その名の通り、プログラマが例を示すことであり、これによりシステムが自動的に推論を行ってプログラムを合成してくれる。ここで言う例とは、プログラムを実行したときに期待したときの入出力の対であったり、あるいは期待される実行トレースそのものであったりする。 2 つめの基準は、実装形態についての分類である。インタプリタベースの方がインタラクティブで柔軟にプログラミングを組むことができるが、その分、効率が悪い。それに比べ、コンパイラベースは効率的である。我々は、 VP においては効率の追求を求めるより柔軟にプログラミングを行うことを重視すべきであると考え、インタプリタベースの考え方を取入れている。インタプリティブなシステムとしては、 Pict[4] や HI-VISUAL[5] などがある。 1 つめの基準に照らし合わせると、 Pict が例示システムではない例、 HI-VISUAL が例示システムである例、と言える。今回のシステムではプログラムの視覚化における意味的付加を主眼においているためあえて取り挙げてはいないが、ユーザ支援を促すうえでも例示システムは必要であると考える。

2.1.2 実装におけるベース言語

ビジュアルプログラミングシステム (VPS) には様々あるが、我々のシステムではベースとなる言語を用いている。これは、ビジュアル言語のシステムをテキスト

ベースの世界から切り離して作るより、既存のテキストベースの処理系に寄生させた方が現実的であると考えるからである。プログラミング言語は大きく手続き型言語と宣言型言語に分類されるが、我々はベース言語として宣言的言語を選択している。これは以下の理由からである。

- 宣言型言語の方がプログラム要素数が少なくなる。
  VP は視覚的な表現が多いが、その分、同じプログラムを表現する場合にはテキストよりもかさばり量が多くなるということがある。これは、プログラムに加えて図形的要素が追加されるためであると容易に考えられる。したがって、なるべくビジュアル要素を少なくするような言語が好ましい。手続き型言語と宣言型言語の違いを、竹内 [8] は、「低レベル指令列の時間順序を保持したまま抽象化を進めたのが、いわゆる手続き型の言語である。これは指令の時系列という、いわばハウ (how) の抽象化である。これに対して、解きたい問題の記述の詳細化、すなわちホワット (what) の方向で抽象化を進めたのが宣言型プログラム言語である。」と述べている。一般にこの二つの言語の立場を明確にしたものは、 Backus のチューリング賞受賞講演 [9] であり、手続き型言語の大元のフォン・ノイマン型プログラムと宣言型言語の一つの関数型プログラムについて述べている。この一部に内積のプログラム例がある。 for 文と代入文からなる手続き型の記述例に対し、関数型は定義文一つで簡潔に記述することができる。これは特殊な例であるが、一般に宣言型言語は手続き型言語に比べて要素数が比較的少なくなると言う特徴を持つ。

- ビジュアル表現は宣言的である。
  プログラムをテキストで表記した場合、これを解釈する順序には、何を記さなくても「上から下へ」または「左から右へ」というように暗黙の了解がある。テキストを読むという作業をプログラム処理と見なすと、その処理における解釈は逐次的に処理が行われると言える。この点においてテキストは手続き的であると言える。一方で、図形を紙面に描いた場合を考えてみる。図形の記述は 1 次元のテキストと異なり解釈する順序が決まっていない [10]、と言われる。プログラムを図形で表現した場合、ある決まりを与えなければ、どの図形から解釈しても良い。この点において、図形解釈の処理は並列的に行われる。ここで少し見方を変えてみる。図形を紙に記述することは、単に「図形が空間的にその場所にある」と定義していることになる。これは言い換えると「図形を宣言している」と見て取れる。一般にビジュアル表現は宣言的と言われている。

このようなシステムの例として、並列処理言語を対象とした PP[11] や KLIEG[12] といった様々な VPS が試作されている。我々も宣言型言語をベース言語にすることを考え、その一つである代数的仕様記述言語を選択した。

2.2 代数的仕様記述言語 CafeOBJ

代数仕様の手法は形式仕様記述の一つであり、すでに 20 年近くにわたって研究が進められている [13]。形式仕様記述は、仕様を確定する段階で特定の数学モデルに基づく形式的言語を用い、記述の曖昧さを除去したり、ソフトウェア開発の上流工程における分析や検証を容易にすることにより、コストを削減したり信頼性を向上させたりする手法である。代数仕様はこのうち抽象代数をモデルとする手法を指す。例外処理や演算の多重定義に厳密な意味を与える順序ソート代数 [14] が意味論の基盤である。また、他の形式的仕様記述言語と異なり、記述した仕様を実行することができる。仕様の実行は項書換えによって実現される。

CafeOBJ[15][16] とは、代数的仕様記述言語の 1 つである。 CafeOBJ 言語を採用した理由としては、「CafeOBJ 言語には既にインタプリタ、トレーサなどの処理系が実装されていること」と「順序ソート等号理論を基盤とし、構造化プログラミングの概念を取り入れることでより高度な仕様を記述することが可能であること」などが挙げられる。

2.2.1 CafeOBJ のプログラム記述

以下ではこの CafeOBJ 言語を使って記述例を紹介する。

CafeOBJ 言語のトップレベルの構文要素はモジュールである。モジュールは合わせて定義すべきデータや操作のまとまりを意味する。このモジュールの中に代数言語の基本要素であるソート、演算、変数、等式の宣言が含まれる。この CafeOBJ 言語による仕様の記述例を**図 2.1**に示す。このモジュールは自然数 (厳密には Natural Number) とその上の足し算を定義したものである。 モジュールはキーワード **module** の後にモジュール名を書き、続いて “**{**” から “**}**” の間にモジュール要素を記述する。例では SIMPLE-NAT がモジュール名となる。

データの集合を意味するソートは “[” から “]” までの間に記述する。ソートを列挙する場合の区切りには記号 “,” を用いる。図 2.1の例に示すモジュール SIMPLE-NAT には、 Zero、 NzNat、 Nat の 3 個のソートがあり、ここでは各々 ゼロ、 ゼロ以外の自然数、 自然数 を意味する。これら 3 個のソートを単に列挙する場合、CafeOBJ では次のように書く。

```
module SIMPLE-NAT {
 [ Zero NzNat < Nat ]
  signature {
    op 0 : -> Zero
    op s : Nat -> NzNat
    op _+_ : Nat Nat -> Nat
  }
  axioms {
    var N : Nat
    var M : Nat
    eq [0] : 0 + N = N .
    eq [1] : N + s(M) = s(N + M)
  }
}
```

図 2.1: CafeOBJ 言語による仕様記述例

```
[ Zero , NzNat , Nat ]
```

CafeOBJ 言語は順序ソートを基盤としており各ソート間には順序関係を与えることができる。ソートをデータの集合と捉えことで、ある意味で順序関係を集合の包含関係に類似している。何かを含むソートを上位ソート、含まれるソートを下位ソートと呼ぶ。ソート間に上位／下位の関係がある場合、この下位ソートに属する項は、暗黙の了解により全て上位ソートにも属する。しかし、必ずしもその逆は成り立つというわけではない。 CafeOBJ におけるソート間の順序関係は、記号 “<” を使って表し、以下のように左から下位ソート、記号 “<”、上位ソートという順番で記述する。

下位ソート (列) < 上位ソート (列)

ここでの関係は Zero と Nat、 NzNat と Nat の間に関係を与える。 Zero や NzNat は Nat の下位ソート、逆に Nat は Zero や NzNat の上位ソートとする。 CafeOBJ 言語ではソート間における関係の定義を行うことで、同時にソート宣言も行う。

```
[ Zero < Nat , NzNat < Nat ]
```

このケースでは、上位ソート Nat が同じであるから、これらをまとめて

```
[ Zero NzNat < Nat ]
```

のように記述できる (これは図 2.1中にある記述と同じ)。

キーワード **signature** の後の “**{**” から “**}**” までの部分がシグニチャを表す。シグニチャとは、その意味 (著名、特徴、標識) からも推測できるとおり、このプログラム内での記述に使用する文字列を定義するために定義される。このシグニチャは主に演算の定義から成る。先ほどのソートもこの中で宣言することも可能である (別の見方をすれば、ソートも演算の定義に使われる記号の一部であり、シグニチャ内の要素とも言える)。演算の定義は、 **op** から始まる一文で表す。また、 CafeOBJ 言語の演算にはある一定の法則を満たすことを言明するために演算には属性を付けることができる。これには交換則・結合則などがある。

交換則:　`X + Y = Y + X`
結合則:　`(X + Y) + Z = X + (Y + Z)`

結合則を用いることによって、例えば $X + Y + Z$ と言う構文上は曖昧である項の意味を一意に定めることで構文解析を容易にすることが可能となる。演算属性は演算定義文の一番最後の “**{**” から “**}**” の間に記述する。演算属性は記述しなくても良い。演算には引数を表すアリティとその返却値を表すコアリティを持つ。以下に CafeOBJ 言語における演算の書式を記す。

```
op 演算名 : アリティ(列) -> コアリティ { 演算属性(列) }
```

例では、 0、 s、_+_という演算名を持った 3 個の演算が定義されている。演算 0 にはアリティが無く、コアリティは Zero である。演算 0 はソート Zero に属する定数を表していると言える。演算 s はアリティとしてソート Nat を持ち、コアリティはソート NzNat となる。この演算はサクセサ (+1) を表す。演算_+_は 2 つのアリティ (ソート Nat) とコアリティであるソート Nat から成り、ソート Nat 上の足し算を表している。演算名で + と_+_との違いは、一般にテキスト上で前置法 $(+(2,3))$ で記述するか中置法 $(2+3)$ で記述するかと言う違いによる。即ち後者の演算名_+_の中にある “_” という記号を項に置換えた結果の文字列はこの中置法による記述となる。また、演算_+_には属性 **assoc** と **comm** があり、各々、結合則と交換則を表している。

キーワード **axioms** の後の “**{**” から “**}**” までの部分が公理系を表す。シグニチャがモジュールで使用する文字列を定義するのに対し、公理系ではその記号列を使って動作 (アルゴリズム) を定義する。この公理系は主に等式の定義から成る。等式の定義を行う前に、等式の中で用いられる変数の定義を行う。変数を用いて項の書

換えパターンを省略して記述することができる。変数はキーワード **var** から始まる文で記述する。例ではソート Nat に属する 2 個の変数 N、 M を定義している。

```
var N : Nat
var M : Nat
```

同じソートに属する変数を列挙する場合はキーワード **vars** を使って書く。また、記号 “**:**” の後に変数の型となるソートを書く。従って以下のように記述しても同じ意味を持つ変数 M, N が定義できる。

```
vars N M : Nat
```

等式はキーワード **eq** で始まる文で記述する。左辺項と右辺項から成り、それらの間を記号 “=” で結ぶ。

```
eq 右辺項 = 左辺項
```

また、等式にはラベルを与えることができる。これは **eq** の直後の “[” と “]” の間に書く。その後に記号 “**:**” を置く。

```
eq [ ラベル ] : 右辺項 = 左辺項
```

等式 (1) は右辺項が $0+N$、左辺項が N である。 0、 $+$ が演算子、 N は変数である。これは項の中に $0+N$ と一致する部分があればその部分を変数 N と対応する部分項に書換えることを意味する。同様に、等式 (2) は $N+s(M)$ と $s(N+M)$ が各々左辺項、右辺項であり、 $N+s(M)$ と一致する部分を $s(N+M)$ に書換えることを意味する。

2.2.2 CafeOBJ 言語におけるプログラム実行

はじめに書いたように代数的仕様記述言語で書かれた仕様は項の書換えによって実行する。 CafeOBJ 言語には既に実行を行うことが可能なインタプリタの処理系が実装されており、 CafeOBJ 言語で書かれた仕様を実際に実行することが可能である。

実際に図 2.1で与えた仕様例を CafeOBJ インタプリタで実行してみる。この結果を 図 2.2 に示す。この例では図 2.1の仕様を与えて $s(s(0))+s(s(s(0))) => s(s(s(s(s(0)))))$ という項の書換えを実行している。

図 2.1の左側に並んでいる数字は単に行数を示すための数字であり実際処理系が表示するものではない。この例の 1 行目で処理系を起動している。起動すると幾つ

```
 1   % cafeobj
 2   -- loading standard prelude
 3   Loading /opt2/cafe/cafeobj-1.3/prelude/std.bin
 4   Finished loading /opt2/cafe/cafeobj-1.3/prelude/std.bin
 5
 6                  -- CafeOBJ system Version 1.3.1 --
 7                  built: 1997 Oct 23 Thu 10:03:30 GMT
 8                         prelude file: std.bin
 9                                  ***
10                      1998 Jan 29 Thu 5:10:04 GMT
11                            Type ? for help
12                                  ---
13                      uses GCL (GNU Common Lisp)
14               Licensed under GNU Public Library License
15                 Contains Enhancements by W. Schelter
16
17   CafeOBJ> in simple-nat
18   -- processing input : ./simple-nat.mod
19   -- defining module SIMPLE-NAT....._..* done.
20
21   CafeOBJ> select SIMPLE-NAT
22
23   SIMPLE-NAT> red s(s(0)) + s(s(s(0))) .
24   -- reduce in SIMPLE-NAT : s(s(0)) + s(s(s(0)))
25   s(s(s(s(s(0))))) : NzNat
26   (0.000 sec for parse, 4 rewrites(0.030 sec), 10 match attempts)
27
28   SIMPLE-NAT>
```

図 2.2: CafeOBJ インタプリタによる仕様記述の実行

かのメッセージが表示されて “CafeOBJ>” というプロンプトが現れる。次に CafeOBJ 言語で書かれた仕様を示す拡張子 “mod” を持ったファイル “simple-nat.mod” を読込む。この操作は 17 行目の “in simple-nat” で行う。このファイルには図 2.1と同じ仕様が記述されている。読込んだ後に 21 行目の “select” からの文でモジュール SIMPLE-NAT を選択し、 23 行目の “red” から始まる文で、モジュール SIMPLE-NAT 上での項 $s(s(0)) + s(s(s(0)))$ の書換えを実行している。 25 行目から結果として NzNat ソートを型とする項 $s(s(s(s(s(0)))))$ が得られたことがわかる。

この実行は足し算を持つ自然数という仕様上で項 $2 + 3$ を規約項 5 に書換えていると捉えられる。自然数上の項 $2 + 3$ は図 2.1の仕様により $s(s(0)) + s(s(s(0)))$ のように記述する。

項の書換えは以下のように実行される。

1. 以下の (2)、 (3) を処理が終わるまで繰り返す。

2. 与えられた項の中で適用できる仕様上の各等式を探す。即ち、項の部分項の中で各等式の左辺と一致する部分を見つける。等式が複数適用可能である場合、適用できる部分項が複数ある場合も考えられるが、ここでは等式も部分項も一意に決まる。

3. 見つかればその等式に従って (等式の右辺のように) 項を書換える。見つからなければその項は規約項をとなるため、そこで処理を終了する。

この書換え過程を順に書いていくと項 $s(s(0)) + s(s(s(0)))$ は項 $s(s(s(s(s(0)))))$ となる。このとき $s(s(s(s(s(0)))))$ に適用できる等式がないので、これが規約項となり書換えが終了する。この $s(s(s(s(s(0)))))$ は自然数 5 に他ならない。

CafeOBJ インタプリタでは、この書換え過程 (トレース) を表示させることも可能である。この処理系はトレーサと呼ばれる。トレーサの実行を**図 2.3**に示す。この図は図 2.2の続きである。 28 行目でインタプリタの処理をトレース表示状態にしている (トレーサの起動)。図 2.2の 23 行目と同じように項の書換えを実行する。今度は結果以外に入力された項がトレースされていく過程を表示している。

32 行目から 34 行目、 35 行目から 37 行目、 38 行目から 40 行目、 41 行目から 43 行目でそれぞれ簡約が行われている。簡約は全部で 4 回行われていることがわかる。一回の簡約は以下のような形式で表示される。

1>[簡約番号] 適用された等式から得られる (書換え) ルール

```
28    SIMPLE-NAT> set trace on
29
30    SIMPLE-NAT> red s(s(0)) + s(s(s(0))) .
31    -- reduce in SIMPLE-NAT : s(s(0)) + s(s(s(0)))
32    1>[1] rule: eq N:Nat + s(M:Nat) = s(N:Nat + M:Nat)
33          { N:Nat |-> s(s(0)), M:Nat |-> s(s(0)) }
34    1<[1] s(s(0)) + s(s(s(0))) --> s(s(s(0)) + s(s(0)))
35    1>[2] rule: eq N:Nat + s(M:Nat) = s(N:Nat + M:Nat)
36          { N:Nat |-> s(s(0)), M:Nat |-> s(0) }
37    1<[2] s(s(0)) + s(s(0)) --> s(s(s(0)) + s(0))
38    1>[3] rule: eq N:Nat + s(M:Nat) = s(N:Nat + M:Nat)
39          { N:Nat |-> s(s(0)), M:Nat |-> 0 }
40    1<[3] s(s(0)) + s(0) --> s(s(s(0)) + 0)
41    1>[4] rule: eq 0 + N:Nat = N:Nat
42          { N:Nat |-> s(s(0)) }
43    1<[4] s(s(0)) + 0 --> s(s(0))
44    s(s(s(s(s(0))))) : NzNat
45    (0.010 sec for parse, 4 rewrites(0.070 sec), 10 match attempts)
46
47    SIMPLE-NAT>
```

図 2.3: CafeOBJ トレーサによるトレース表示

```
    { 等式上に出てくる変数から項への置換え }
1<[簡約番号] 等式上の変数に実際の項を与えた結果
```

“1 $>$” から始まる行がトレーサへの入力、“{” から “}” の間で等式上で用いられる変数の置換えを、“1 $<$” から始まる行がトレーサからの出力を示している。

最後に 44 行目でトレース結果が表示されている。図 2.3に示すトレース結果から以下のように簡約が進んでいることがわかる (**表 2.1**)。

項 (下線は簡約する部分)	簡約で用いられた等式	変数の置換え
$\underline{s(s(0)) + s(s(s(0)))}$		
$\Downarrow$	[1] $N + s(M) = s(N + M)$	$N \Rightarrow s(s(0)), M \Rightarrow s(s(0))$
$s(\underline{s(s(0)) + s(s(0))})$		
$\Downarrow$	[1] $N + s(M) = s(N + M)$	$N \Rightarrow s(s(0)), M \Rightarrow s(0)$
$s(s(\underline{s(s(0)) + s(0)}))$		
$\Downarrow$	[1] $N + s(M) = s(N + M)$	$N \Rightarrow s(s(0)), M \Rightarrow 0$
$s(s(s(\underline{s(0) + 0})))$		
$\Downarrow$	[0] $0 + N = N$	$N \Rightarrow s(s(0))$
$s(s(s(s(s(0)))))$		

表 2.1: トレーサによる項書換えの実行

第 3 章

ビジュアルプログラミングシステム “CafePie”

我々は VPS として CafePie[17, 18, 19] を作成した。VPS としての CafePie に期待される機能を整理すると、以下のようになる。

- 図形による入力
  ユーザは図形の組合せの形で代数的仕様記述言語の各基本要素を入力する。各図形はマウスによる直接操作で見たままのイメージで編集される。

- プログラムコードの自動生成
  図形表現形式でプログラムを編集すると、それに応じた CafeOBJ プログラムのテキスト表現を生成し、表示する。

- 図形表現の自動生成
  逆に CafeOBJ プログラムのテキストを表現を入力し、図形表現に自動生成され、表示される (これは、例えば単に演算を定義する文字を入力してすぐにそれを解釈しアイコンで表示する機能である。現 CafePie ではまだ実装されていないが、テキストを解釈することは現 CafePie でもファイルの読込みで行っているため実装は可能である)。

- 修正機能
  図形表現は、それをあとで修正できる。この機能を用い、テキストから自動生成された図形表現を修正し、ビジュアルなプログラム編集を行う。

- 保存／再生機能
  図形表現で編集したプログラムは、それをファイルに保存し、必要なときに呼出すことができる。現在の CafePie では、ビジュアル表現を CafeOBJ 言語に変換して保存する。図形の位置や大きさといったビジュアルな情報も保存も同時に行う。

- プログラム実行機能
  入力された図形表現は、それを図形的に実行することができる。このとき処理系には CafeOBJ インタプリタを使用する。 CafePie は項書換えシステムのビジュアルインタフェース部分を担当する。

- プログラミングの統合環境
  CafePie はプログラムの作成／編集／実行をビジュアルな形で支援する統合環境である。

CafePie は当初 Java1.0 ベースで開発が進めていたが、 Java1.1 を経て、現在は Java1.2 ベースへと移行している。このシステムは Java のアプリケーションとして実行される。 CafePie を起動すると、**図 3.1**のような画面が表示される。 画面の半分以

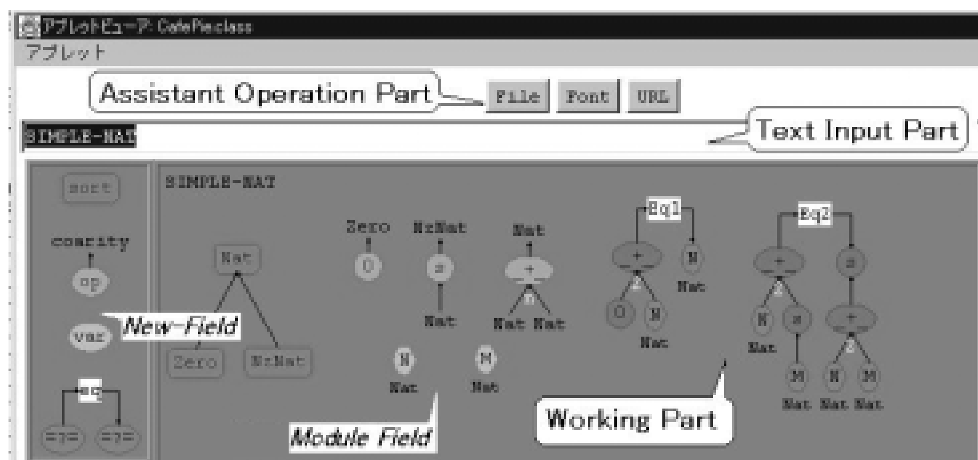

図 3.1: CafePie の実行画面

上を広く占有する部分が “Working Part” である。プログラムはここで視覚化される。また、この中でプログラムは視覚的に編集される。この中の要素を選択すると、その要素に書いてあるラベルを編集することができる。それが、 “Text Input Part” である。要素を選択すると、ここに現在のラベル中のテキストが表示されるので、ユーザがこれを自由に編集して、要素のラベルを変更することができる。 CafePie を起動しただけでは、図 3.1のような図形で視覚化されたプログラムは表示されない。これは、表示すべきプログラムがロードされていないためである。プログラムをロードするには、画面上部のボタン列 “Assistrant Operation Part” 内の “File” ボタンを押し、ファイルを選択する必要がある。選択するファイルは図 2.1のような CafeOBJ のプログラムファイルである。ファイルが選択されると、 CafePie は

そのテキストを解釈してシステムの内部表現に置換え、 Working Part 内の “Module Field” にこれを表示する。 Working Part 内にある “New Field” は、新しく要素を追加するために用いられる。

以下では、 CafePie の中で用いられる図形要素と直接操作を用いた編集操作、そして CafeOBJ インタプリタを利用した実行表示について記述する。

3.1 システムによるプログラムの視覚化

CafePie では CafeOBJ 言語のプログラム構造を視覚化するために、プログラムの各基本要素を図形で表している。前にも述べた通り Myers は、プログラムの視覚化 のように既に書かれたプログラムを視覚化する立場と視覚化プログラムのように最初から図形的にプログラムを作成する立場との違いを強調している。 我々は、視覚化プログラムという後者の立場にある。これを踏まえた上で、以下の話を進める。

我々は、視覚化されたプログラムの構成要素を表す図形のことを「アイコン」と呼ぶ。アイコンは「図像」であり、ウィンドウシステムなどの GUI 環境において、さまざまなオブジェクトを示すのに利用される小さなビットマップのことを意味している。単なる要素名を表示する代りにアイコンとして表示することで、視覚的に要素を識別することが可能になる。「CafeOBJ 言語における基本要素は CafePie 上で決められたアイコンで視覚化される」と言うことができる。

CafeOBJ 言語の基本要素には、データ構造を表すための要素であるソート・演算と、その振舞いを表すための変数・等式がある。これらの基本要素が CafePie における視覚化の対象となる。一般に、ここで言うデータを項と呼び、その振舞いを等式により記述する。等式でかかれた振舞いによる項書換え過程が CafeOBJ におけるプログラムの実行にあたる。

ソートは CafeOBJ 言語における型を表す。厳密ではないが、 C における整数型、文字型などと同じである。これらソートには順序関係を付けることができ、一般にこれを順序ソート (整数型に対して実数型を上位ソートなど) と呼ぶ。 CafePie ではこれを、ソートを頂点、関係を有向線とした有向グラフで表す (**図 3.2**)。ソートはラベル付きの緑の矩形とし、下位ソートから上位ソートへ有向線を引く。図 3.2のテキスト表現は $[Sort, Sub1Sub2 < Sort < Super]$ となる。

演算の視覚化を考える前に、演算と変数から構成される項の視覚化を考える。ここでは項を表現するために一般に良く用いられている木構造を用いる。これにより、その構成子である演算はノードとそれら要素間の下位 - 上位関係は下位から上位へ

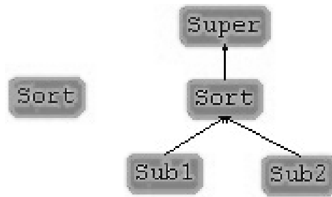

図 3.2: CafePie におけるソートの視覚化

の有向線によって表現される。演算はソートと区別するために楕円で表す。項の視覚化規則に合わせるため、演算の引数は楕円下方に返却値は上方に配置される (**図 3.3**)。

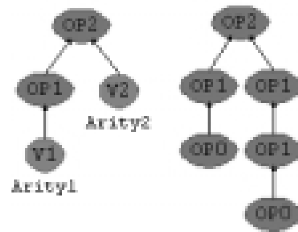

図 3.3: CafePie における項の表現

演算は演算名と幾つかの引数と 1 つの返却値から成る。引数と返却値は型を表すソートによって与えられる。演算は演算名をラベルとして持つ水色の楕円で表現し、引数を演算の下方に、返却値を上方に配置する。引数から演算、演算から返却値へと有向線を引く (**図 3.4**)。

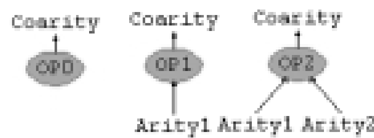

図 3.4: CafePie における演算の視覚化

変数は等式を記述するときに現れるが、項の構成要素であり演算の特別なものと見ることができる。変数は変数名とソートの型から成る。変数名をラベルとして持つ橙色の楕円で表され、その下方にソートを記す (**図 3.5**)。
等式は項を書換えるための規則を表す。 CafeOBJ において、等式は左辺と右辺の

図 3.5: CafePie における変数の視覚化

項、等式ラベルから成る。等式ラベルを中央に置き、その左下には左辺の項を、その右辺には右辺の項を配置する (**図 3.6**)。左辺から右辺に書換えるということを明確にするため左辺から (ラベルを通して) 右辺へと有向線を引く。

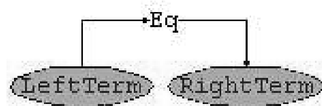

図 3.6: CafePie における等式の視覚化

CafeOBJ において、プログラムはモジュールの集まりである。モジュールはモジュール名と上述した各基本要素の集合からなる。モジュールは灰色の矩形で表し、その中にラベルとこれらの要素を表現する。ユーザはこのモジュールアイコン上でプログラムの定義を行う。

3.2 ビジュアルプログラミングシステムにおける編集方法

システムにより視覚化されたプログラムは全てアイコンにより表示されるが、それだけではプログラムを編集することはできない。我々は、プログラムをどのようにして作成していくか、即ちアイコンの操作が重要であると考え、以前から研究している [17, 18]。ここでは、現在一般に広く普及しているディスプレイ＋マウスという環境を想定している。

マウスを用いて操作する最大の利点は、「マウスカーソルで対象物を直接さわって操作しているという実感をユーザに与えること」である。我々は、単純な操作を用いて いかにユーザのプログラム編集を支援するかという問題を解決するために、直接操作 (Direct Manipulation) に注目した。

3.2.1 直接操作

直接操作は、 Shneiderma が提唱した操作法であり [20]、次のような条件を示している。

- 関連するオブジェクトが常に画面上に表示されていること。
- 決められた形式に従って命令 (コマンド) を入力するのではなく、マウスの移動のような物理的動作やボタンの押下による操作であること。
- 操作は高速、可逆的で、操作結果による変化が即座に見えること。

ここで言う直接操作は、メニューのようなものも含まれる。しかし、メニューは、項目が多くなると一つの操作を行うにも時間が掛かり迅速で快適な編集環境を実装するには不向きであるという欠点を持つ。我々は、メニューは補助的操作に用いることとし、主要となるプログラムの編集操作は以下で述べるドラッグ アンド ドロップ手法を用いている。

3.2.2 ドラッグ アンド ドロップ手法

ドラッグ アンド ドロップ (DND)[21] は、 GUI 環境におけるマウスによるアイコン操作手法の一つである。 DND 操作は、ドラッグからドロップまでの操作が一つの操作で行うことが可能である。例えば、ファイル操作では、マウスボタンを押してファイルを選択し、そのまま他のディレクトリの位置まで移動し、そこでボタンを離すことで、ファイルを移動することができる。また、アプリケーションのアイコン上に、そのアプリケーションで使うデータのアイコンを選択移動し、そこでアイコンを離すことで、アプリケーションを起動させることができる。同様に、ワードプロセッサや、スプレッドシートのソフトでは、データの移動などに利用されている。このような切れ目のない操作を利用することで、編集に費やす思考を少なくなると考えられる。 DND 操作の手順をまとめると、以下のようになる。

1. まず、マウスポインタを問題となる (Source) アイコンの上に移動させ、そこでマウスボタンを押す。
2. 次に、マウスボタンを押したままマウスを移動させることによって、アイコンを画面上で移動させる。この操作をドラッグ操作と言う。ユーザは、 Source アイコンを画面上の任意の場所にドラッグすることができる。
3. 最後に、適当な (Target) アイコンの上でマウスボタンを離すと、それに関連したアクションが実行される。この操作をドロップ操作と言う。このアクションは、予め Target アイコンに関連付けらているのが普通で、 Source アイコンの持つデータ情報をもとに処理される。ファイル操作においても「移動」というアクションが関連付けられていると見ることができる。もし、 Source

アイコンに関するアクションが定義されていなかったり Target アイコンにアクションが関連付けられていない場合には、何も行われない。

3.2.3 ドラッグ アンド ドロップを用いたプログラムの編集

プログラムの編集は、 DND 手法による 2 つのアイコンの重合わせという基本操作の繰返しによって行われる。これをまとめると**表 3.1**のようになる。

Event	Source	Target	Action
ソート関係	ソート	ソート	関係作成、削除
引数追加	ソート	演算	演算に引数追加
引数交換	引数	引数	同演算上の引数位置交換
演算型決定	ソート	返却値	演算の返却値を決定
項の作成	演算	変数	演算に置換、部分項の作成
項の置換	項	変数	項に置換
変数型決定	ソート	変数	変数のソートを決定

表 3.1: ドラッグ アンド ドロップによる編集一覧

アイコンは Module Field 内に作成されているという仮定のもとでプログラムの編集作業が行われる。アイコンを新しく作成する場合は、 New Field 内のアイコンを使うことで行われる。 New Field 内には基本要素である ソート・演算・変数・等式が既に定義されている。ユーザは作成したいアイコンの種類をこの中から選び、これを DND 手法を用いて Module Field 内にドロップすることで行われる。ドロップすることで、新しく作成すべきアイコンの位置がわかるため、システムはそこに新しくアイコンを作成し、表示する。また、削除する場合も DND 手法によって行われる。このアイコンの削除は、削除するべきアイコンを選択しこれを Module Field 内から枠外にドロップすることで行う。

プログラムの編集を説明するために、図 3.1の中に視覚化されているモジュール SIMPLE-NAT[16] の作成手順を紹介する。 Module Field 内の左側に ソート (Nat、 $Zero$、 $NzNat$) があり中央上部に演算 (0、 s、 $_+_$) がある。その下方に変数 (N、 M)、一番右側に書換え規則を示す等式 ($0 + N : Nat = N : Nat$、 $N : Nat + s(M : Nat) = s(N : Nat + M : Nat)$) がある。モジュール構造を記述する場合には、

1. まず、ソートを記述し、
2. それをもとに演算と
3. 変数を作成し、
4. 最後に等式を作成する

という流れになる。

ソートの作成: ここでは Nat、 $Zero$、 $NzNat$ という 3 つのソートを定義する。これは、 New Field 内のソートを選択し、 DND 操作を用いて Module Field 内に作成する。ソート名はデフォルトで与えられる。 Text Input Part を用いて各ソートのラベルを変更する。次にソート間に関係を定義する作業に移る。ここでは、 $Zero < Nat$ と $NzNat < Nat$ という 2 つの関係がある。ソート間の関係付けは、ソートからソートへの DND 操作によって定義する (表 3.1)。このとき Source が下位ソート、 Target が上位ソートとする。 $Zero < Nat$ という関係を作成する場合には $Zero$ が Source、 Nat が Target という具合いである。関係が追加されると下位ソートから上位ソートへ有向線が引かれる。ソートの関係を削除する時は、関連のあるソート間の一方をもう一方に DND 操作することで行なう (表 3.1)。このときはソートの関連付けと異なり、 Source と Target の区別は付けていない。

演算の作成: New Field 内の演算を選択し、 DND 操作を用いて Module Field 内に作成する。ソートと同じように新しい演算がモジュール内に定義される。演算は必ず一つの返却値 (コアリティ) を伴う。これは定義されているソート名を用いて定義される。返却値はソートと関連付けられるので、その名前の変更はソートから返却値への DND 操作で定義することができる (表 3.1)。演算名はデフォルトで与えられる。 $op_+_ : NatNat -> Nat$ の作成を考える。はじめにされた演算 (これに名前を変更したもの) は、 $op_+_ : -> Nat$ というように、引数がない形であるため、これに引数を追加する必要がある。表 3.1をみると、引数の追加はソートから演算への DND 操作で与えられている。したがって、引数となるべきソート Nat を演算 $_+_$ 上にドロップすることになる。この場合は 2 引数であるため、この操作も 2 回行う。これで 2 引数演算 $_+_$ が作成された。他の演算も同じようにして作成することができる。

変数の作成: New Field 内の変数を選択し、 DND 操作を用いて Module Field 内に作成する。変数名もデフォルトで与えられる。しかし、これは以前の実装方法 [17, 18] の名残であり、あまり奨励されない。現在は、ソートを Double Click することで作成する。変数は、変数名とその型であるソートから定義される。したがっ

て、ソートから作成することで、ソートの入力を省くことができ、面倒な操作が不要になる [19]。この操作方法に従うと、 $varN : Nat$ の作成はソート Nat を ダブル クリック することで行われる。ソート名には Nat が与えられることになる。変数名は Text Input Part を利用することで変更される。

等式の作成: 等式を定義する場合は、 New Field 内の等式を選択し、 DND 操作を用いて Module Field 内に作成する。等式のラベルはデフォルトで与えられる。左辺、右辺には名前、型の決まっていない変数が配置されるので、これらの上に項を作成する必要がある。項の作成は、既にある変数上に項の作成／置換を再帰的に行うことで実現されている (表 3.1)。例えば等式 $eqN + s(M) = s(N + M)$ 上の左辺 $N + s(M)$ は、次のような手順で作成される。

1. 等式が作成されるとその左辺部分には変数が作成される。この変数名を仮に V とする。

2. この変数上に、既に作成されている演算 _+_ を代入する感覚で重ね合せる。変数はこの演算に置換わる。またこの演算は引数を 2 つ持つため、新たに 2 つの変数 ($V1$ と $V2$) が作成される。

3. 片方の変数の名前 ($V1$) を変更する。これには既に定義した変数 N を用いる。

4. もう片方の変数 $V2$ に 1 引数演算 s を重ね合わせる。変数はこの演算に置換わり、新たな変数 ($V3$) ができる。

5. 最後にこの変数の名前 ($V3$) を、変数 M を利用して変更する。

この作業によるアイコンの変化は**図 3.7**のようになる。

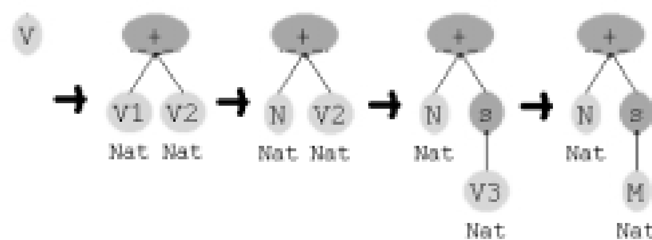

図 3.7: ドラッグ アンド ドロップ手法による項の作成手順

DND 手法のみによる編集操作は魅力的であるが、この応用範囲は広く、まだ幾つかの問題が残されている。 DND 手法の欠点として、遠隔操作には不向きであると

いうことが挙げられる。ドラッグしてドロップするまでの間が長すぎると、マウスでの移動が困難になるためである。また、画面がスクロールする場合には、この問題が顕著に現れる。 DND 手法は、カット アンド ペースト手法の応用であり、マウスを使って切れ目のない操作を実現したものである。切れ目のない操作を断念し、カット アンド ペースト手法との併用することも考えられる。また、これを解決する手法の 1 つとして、アイコン投げ手法 [22] が提案されている。これはアイコンをドラッグする代りに、手裏剣のように投げる手法である。アイコンをつまみ、助走を付けてから目的の方向で離す。通常、 DND 手法では離した所でドロップとされるが、この手法の場合は目的の方向へ移動し、他のオブジェクトにぶつかって止まる。この手法を応用することで、変化に飛んだ編集操作を実現することが可能であると考えられる。

3.3 ビジュアルプログラミングシステムにおける実行表示

次に CafePie におけるプログラムの実行について述べる。
プログラムの実行は以下の手順で行われる。

1. ユーザは、 Module Field 内で項を編集する。この項はゴールと呼ばれ、作成したモジュール動作をテストするために用いられる。項の編集は、等式の右辺や左辺に現れる項と同じように DND 操作により編集することができる。

2. ゴールを編集したあとに、プログラムは評価される。評価されるプログラムは、現在 Module Field に表示されているプログラムである。この Module Field 内に作成したゴールをモジュール名の上に DND 操作で重ね合わせることによって、プログラムの評価が開始される。

3. CafeOBJ インタプリタがネットワーク上で動作していると仮定する。 Assistant Operation Part 内の Options ボタンによって、 CafeOBJ インタプリタがどこで動作しているのかという情報を、ユーザが明示的に指定することができる (**3.8**)。 CafePie は、まず、 CafeOBJ インタプリタに接続を試みる。接続が可能であれば、インタレクティブモードで接続する (“*interactive*” メッセージを CafeOBJ インタプリタに送信する)。

4. 接続された後で、 CafePie は、モジュール情報を CafeOBJ インタプリタに送信する (視覚化されたモジュールを CafeOBJ プログラムに変換し、これをインタプリタに送信する)。

URL: インタプリタの位置を明示するために IP アドレスまたは URL を入力する

PORT: インタプリタとのソケット通信に使われるポート番号を明示する

図 3.8: インタプリタの URL 指定用ダイアログ

5. 続いて、ゴールを送信する。 CafeOBJ インタプリタにトレースモードでの実行を開始するように命令を出す (“*red in* **module_name** *:* **goal_term** *.*” をインタプリタに送信する)。

6. インタプリタはゴールが書換えられる過程をテキスト情報で CafePie に返すので、 CafePie でこれを解釈し、視覚化する。

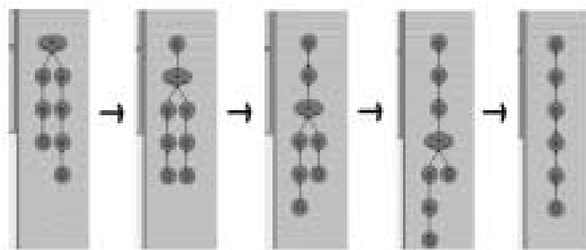

図 3.9: CafePie 上での実行の動的表示

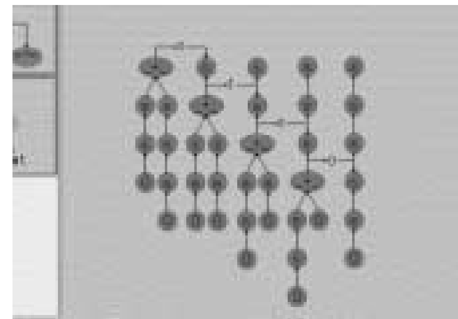

図 3.10: CafePie 上での実行の静的表示

CafePie は、以下に示す 2 つの方法により項書換えの過程を視覚化する。 1 つはアニメーションのように書換え過程を動的に表示する方法である (**図 3.9**)。もう 1 つは書換え過程で出現した項を横に並べて帯状に表示する方法である (**図 3.10**)。前者は、インタプリタの出力に従って、与えたゴールがその場で変化する。これにより、ユーザは項が書きかわっていく過程を大まかに捉えることができる。これに対し、後者は、書換えの行程を詳細に見る場合に適している。ただし、書換え回数が多くなると、画面に入りきらないため、将来、省略した表示方法が必要となる。

CafeOBJ インタプリタからの出力を受取ると、まず、書換え過程を動的に表示する。 SIMPLE-NAT のゴールである “`s(s(0)) + s(s(s(0)))`” は、図 3.9 に示

すようにその場で次々と変化していく。ゴールが“s(s(s(s(s(0)))))”まで書き変り書換えが終了すると、その書換え過程全体を示すために静的に表示される (図 3.10)。書換え過程の情報は CafePie 上に保存されるため、一度実行した書換えはインタプリタと通信しなくてもその場で再現することが可能である。ユーザが、静的に表示された項の一部をダブルクリックすることによって、再びアニメーションを使ってプログラム実行を視覚化することできる。

第 4 章

視覚化カスタマイズ機能

プログラムを視覚化する利点は、プログラムにおける記述力の向上／プログラム動作の理解支援／人に対しより親しみ易くすること、など様々である。しかし、 VPS において、テキストによる表現をグラフのように単純な表現に置換えるだけでは不十分である。プログラミングシステムの中で単純な表現を用いた場合、それが何を表しているのかを判断する手掛かりは、図形の形・構造ではなく、図形の中に書かれたラベルである。プログラムの意味が図形の形・構造に反映されていないため、プログラムを理解するには時間がかかる。

例えば、 CafePie においてプログラム表現は主に木構造によって視覚化される。**図 4.1**はスタックの仕様 (**図 4.2**) を元に項を視覚化した例である。 今まで他の仕様を元にプログラミング作業を行っていたユーザにとって、図 4.1がスタック構造を視覚化したものであると瞬時に判断するのは難しい。なぜ判断が難しいかというと、グラフ中のノードの形・構造を見ただけではプログラムをイメージできないからである。ノードの形は丸であっても四角であっても良い。プログラムの判断にお

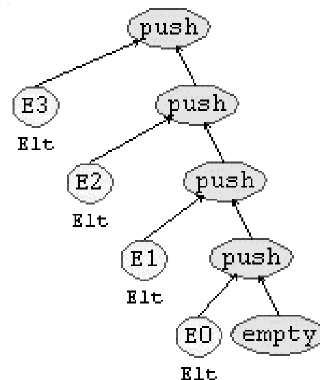

図 4.1: システムによるスタックの視覚化

いてはノード中のラベルのみが重要であり、図形の形・構造とプログラムとの関係は何もない。

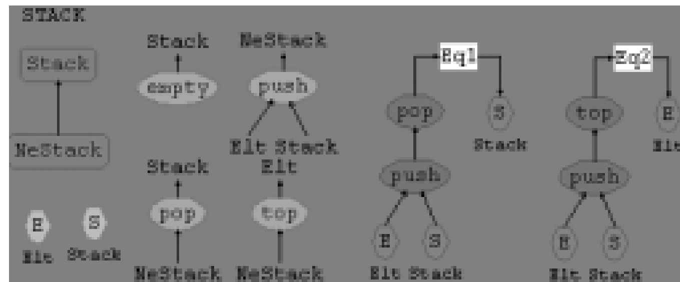

図 4.2: CafePie によるスタック仕様の視覚化

一般に、人が物を認知するときは、まずその形から判断する。我々は、プログラムの持つ意味を各図形の形・構造に反映させることが重要であり、 VPS 上の図形を効率良く識別することに有効であると考えた。そこで、図形の形・構造という特徴を生かした VPS の実現のために、システムが与えた表現をユーザが自由に編集できるような表現のカスタマイズ機能を VPS 上で実現する [23, 24]。

VPS における視覚化のカスタマイズ機構を実現するにあたって、我々は、以下のことを目標にした。

- **以前 CafePie における操作手法との互換性**
  カスタマイズ機構を付ける以前で用いられた操作手法や表示方法はそのまま用いることができる。以前の操作手法とは、 DND 手法による直接操作であり、カスタマイズを編集する操作もこの手法を取入れる。

- **カスタマイズした表現の切替機能**
  カスタマイズ機構によりプログラムの視覚的な表現部分が変更されるばかりでなく、視覚化表現が自由に変更でき、また元の表現に容易に戻すことが可能である。

以下では、カスタマイズ機能を実現するために、その背景となるユーザインタフェースの構築モデルである MVC モデルについて取り挙げる。その後で、実際に VPS として我々の開発しているシステムを取り挙げ、具体的な例を交えながら説明する。

4.1　ユーザインタフェースの構築モデル

システムによって与えられた表現を自由に変更するには、システム内部の視覚化を定義している部分を書き直す必要がある。しかし、動的にシステムの内部コードを書換えるのは危険であるし、できたとしても手間がかかり面倒である。柔軟なカスタマイズ機能を実現するためには、ユーザインタフェース部分のみのカスタマイズを行う必要がある。

4.1.1　MVC モデル

ユーザインタフェースをオブジェクト指向プログラミングで構築するときに広く用いられているのが、オブジェクト指向言語の元祖である Smalltalk で対話型のユーザインタフェースを構築するために考案された **MVC(model-view-controller) モデル** [25] である。 MVC モデルは、アプリケーションとユーザインタフェースを分離するための機構である。

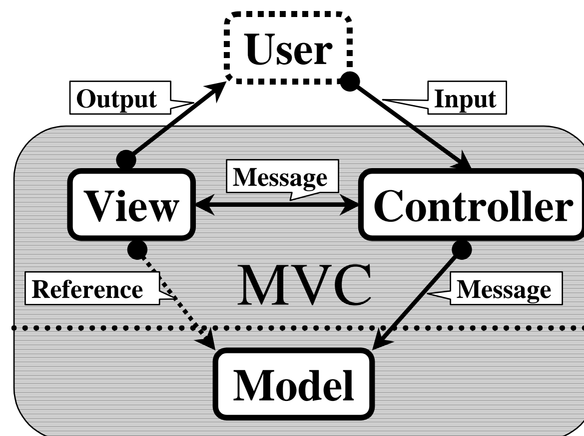

図 4.3: MVC モデル

MVC モデルでは、ユーザインタフェースを以下の三つのオブジェクトの組を単位として構成していく (**図 4.3**)。

- **モデル** (model)
  アプリケーションの扱うデータと、そのデータに関する操作を担当する。ユーザインタフェースに対する依存性は低い。

- **ビュー** (view)
  モデルを画面上に表示する手段を提供する。表示対象となるモデルを 変数と

して保持している。自分が画面上のどの領域に描くのか、どうやって表示するのかを知っているオブジェクト。

- **コントローラ (controller)**
  ユーザからの入力 (キーボードやマウス) を解釈して、モデルやビューに適切なメッセージを送ることを担当する。システムから制御を割り当てられると、入力に関するメッセージを獲得し、モデルとビューに分配する。

通常の MVC モデル の動作は次のようになる。まず、コントローラがユーザからの入力を受け取り、ビューとモデルにメッセージを送信する。モデルは受け取ったメッセージに従って必要な処理を行う。モデルの内容が変化した場合には、自分に依存しているビューに対してその旨を通知する。これによりビューが現在の表現を更新し、ユーザは内容が変化したことを認知できる。

モデル自身は、ユーザインタフェースへの関連を極力排除した形で実現し、ビューとコントローラに表示・対話手段をカプセル化する。これにより、同じモデルに対して異なったビュー・コントローラを用いることができる。また、同じモデルに対して複数のビュー・コントローラの組を設定することで、ひとつのモデルを異なった観点から操作することも可能である。

プログラム表現のカスタマイズ機能を MVC モデルの観点から考えると、「ビューの変更」と言うことができる。我々は、ビューのカスタマイズはプログラム編集中に自由に変更可能な環境を想定している。自由に変更可能であるためには、ビューのカスタマイズが頻繁に行われた場合でも簡単に対応することが要求される。従って、ビューとコントローラが強く結び付いている通常の MVC モデルでの実装は現実的でない。簡単にビューが変更できるようにするために、プラガブル MVC の考え方を用いる。

4.1.2 プラガブル MVC モデル

通常の MVC モデルでは、ユーザインタフェース部分とモデルを切り離して実装できることが重要であり、ビューとコントローラの結び付きには柔軟性がないという問題があった。このような実装形態の場合には、次のような問題が起こる。

- **非常に似通ったクラスを別々に定義する必要がある**
  頻繁にビューが変更されると、そのたびにビューとコントローラの似通ったクラスの組を定義する必要があり、現実的でない。

- **再表示のタイミングが同じとは限らない**
  ビューとコントローラの結び付きが強いため、例えば一つのコントローラに一つのモデルと複数のビューが対応している場合に問題が起こる。コントローラによりモデルを変更する場合を考える。モデルに対応したビューは複数あるが、通常の MVC モデルではどのビューに対しての変更なのかを区別する手段がないので、モデルに対応する全てのビューを更新する必要がある。ビューの数が少ない場合にはあまり問題にならないが、その数が増加すると現実的に問題が起きてくる。

これらの問題を解決し、ビューとコントローラに種々のモデルに対応できるようにした機構が、**プラガブル・ビュー** (pluggable view)[26] と呼ばれるものである。プラガブル形式による MVC モデルは、モデルとビュー間の対話を標準化し、モデルに対する様々なビューを手軽に変更できるようなインタフェースを提供している。

このプラガブルな MVC モデルは、次の二つの概念から成り立っている。

- アスペクト
  一つのモデルを幾つかのサブビューに表示するために、各サブビューに機能を表す名前 (アスペクト) を付け、モデルの変化に従ってそのビューを書き換えるかを指示して不必要な再描画の遅延を避けている。

- アダプタ
  ビューがモデルにアクセスするときに必要なメッセージのやり取りを変換器 (アダプタ) により吸収し、直接アクセスしないようにしている。これにより、モデルに依存しない標準化されたビューをいつでも追加・削除することが可能になる。

このプラガブル MVC モデルを使った場合も、ビューを変更してもモデルには影響を与えずに交換することができる。また、単なる依存性を用いた MVC の欠点を解決し、効率良く実装が行える利点を持つ。

4.2 ビジュアルプログラミングシステムにおけるカスタマイズ機能

我々は、このプラガブル MVC モデルを利用して、 VPS 上での視覚化カスタマイズ機能を実現することを考えた。プラガブル MVC モデルは、一般のユーザインタフェースモデルの実装のために考え出された手法であるため、このモデルを VPS 上で利用するためには次のようなことを解決する必要がある。

- ビューを自由に変更可能な仕組みは提供できるが、それとは別に新しいビューをユーザが用意する必要がある。
- 新しくビューを定義するには、適当なアイコンを用意するだけでは不十分である。以前のビューとの対応付けを明確に行う必要がある。
- 新しくビューを定義するのは時間が掛かるため、効率の良い編集の仕組みが必要となる。

これらの問題を解決するために、我々は次のようなカスタマイズ用の編集機能を実現した。ビューをプログラム編集と同様に、しかもプログラミングの段階で自由に編集可能にする。また、以前とビューとの対応をユーザに明示的に示し、インタラクティブな編集を可能にするために、 DND 手法の直接操作による単純な編集の枠組みを提供している。以降、ユーザレベルの視点からの話を進める。

4.2.1 ユーザの視点から見たカスタマイズ機能

ユーザは VPS によって既に与えられているビューを見ながら新しいビューを定義する。例えば、 VPS により与えられた表現 (ビュー) がある (**図 4.4**左側の OLD VIEW) とすると、ユーザはその右側 (図 4.4の NEW VIEW) に新しい表現を定義する。 このように元のビューとそれを置換えるビューとを同時に表示することで、

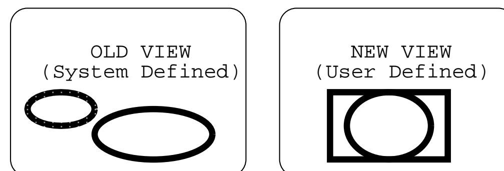

図 4.4: 図形書換えルールの編集

例えば、片方のビュー内の 1 つのオブジェクトを選択すると、もう片方のビュー内のそれに対応するオブジェクトをフォーカスさせることができるようになる。これにより、図形的な対応を直接見てすぐ把握することができ編集を効率的に行うことが可能になる。この状況をユーザの視点から見ると、 VPS 表現からユーザ表現への図形書換えルールを定義するというように捉えることができる。

4.2.2　書換えルールの編集における入力手法の強化

我々の目標は、ユーザによる操作は大雑把でも、完全ではないが大体の作業が行えるようにすることである。マウスを使った図形の編集を考えた場合、 VP のプログラム [17, 18, 19] と同様に、その全てを直接操作 (DND 手法など) で扱うことが可能である [23, 24]。この操作手法は、細い線を選択する、図形をちょっと拡大する、など微妙な動きを必要とする作業には向かない。そのため、操作を単純にする、対象物にある程度の大きさを持たせる、というような工夫が必要となる。我々は、図形書換えルールの編集にこの手法を適用した。操作の対象物はある程度の大きさを持ったオブジェクトとして表現される。そして、これらのルールはドローツールのように描くのではなく、図形オブジェクトを組立てるという操作で編集することになる。見たままにオブジェクトをマウスでつまんで好きなところに移動させて配置してする。ただ、それだけの概念で行えるようにする。

まず、長方形、円などの良く使われる基本的図形は用意しておき、必要に応じてイメージなどをファイルから呼出すことにしている。また、前のビューの情報 (図4.4の OLD VIEW) を利用して、直接的にどこに配置するかを決める。ユーザが行う作業は、用意された図形や既に作成した図形を再利用して、プログラムの主要な構成要素である各演算上にこの図形書換えルールを定義することである。このときの操作は全て DND 手法を用いた直接操作で行う。この DND 手法を図 4.4を用いて考えてみる。書換えルールの編集時において、 OLD VIEW は既に与えられている表現である。よって編集の対象となるのは、 NEW VIEW となる。ユーザはこの中に書換えルールを定義する。 NEW VIEW にオブジェクトが重ね合わせられると、そのオブジェクトはコピーされ、 VIEW 内に挿入される。このときシステムが与えたレイアウト手法に基づいて自動的に配置が決定される。レイアウトをシンプルなものにすることで簡潔に行う。以下で具体的な例を使ってこれを見てみる。

4.2.3　図形書換えルールの定義例

CafePie 上に表示するプログラムは、システムから与えられた規則に従い視覚化される。例として図 4.1で示されるスタックの再視覚化を考える。スタックの要素となる演算には主に empty と push がある。 empty はスタックが空であることを示す。これは長方形で表すことにする (**図 4.5**)。 また、 push はスタックに 1 つの要素を積んだという状態を表す。これを何かスタックがあるときにその上に要素を積むという表現に変換する (**図 4.6**)。

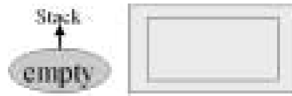

図 4.5: 演算 empty へのビューの定義

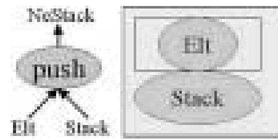

図 4.6: 演算 push へのビューの定義

この演算の編集は**図 4.7**に示すように、 DND 操作を繰り返すことで作成する。この図はスタックの push 演算に対し書換えルールを定義の手順を示している。 そ

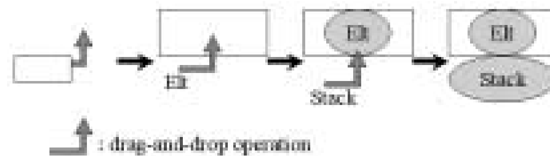

図 4.7: 演算 push における図形書換えルールの編集

の手順は次のようになる。まず、長方形の図形を用意する。次に、システムによって与えられた push の引数である Elt を利用し、それを DND 操作によって、その長方形の中に配置する。このときシステムよってある程度、自動的に制約が与えられる。引数 Elt は文字ではなくそれをラベルとして持つ楕円に自動的に変換される(引数は、項における変数に対応しているため、このようなラベル付き楕円にする)。また、土台となる長方形は中に要素を挿入可能なオブジェクトである。この場合は、このオブジェクトの中央に挿入した図形は、そのオブジェクト内に納めるという制約が自動的に付加される。したがって、要素 Elt は長方形の枠内に入れることにより、枠内の中に配置されるように自動的にレイアウトされる。最後に今度は push のもう一つの引数である Stack を利用して配置する。このとき Stack は要素 Elt の下に配置するため、要素 Elt の下方に DND 操作する。あとは、自動的にレイアウトされ、ちょうど要素 Stack が要素 Elt の下に合うように配置される。オブジェクトの近くに配置する場合には、システムがデフォルトで密着して配置する、幅を揃

えるよという制約が自動的に付加する。

システムが自動的にレイアウトすると述べたが、このレイアウトには様々な方法がある。現段階では、以下に示すような単純なレイアウト機能を実装している。**図 4.8**において、図形 A は図 4.4の NEW VIEW 内にある現在編集しているものである。また、図形 B は DND 操作でユーザが操作しているものである。今、図形 B を図形 A に追加する場合を考える。ユーザは図形 B を操作し、図形 A との関係を定義しようとしている。 図形 B を図形 A の上に重なると、システムは図形 A のどの

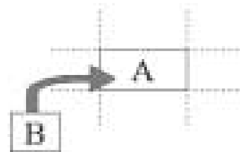

図 4.8: ドラッグ アンド ドロップ手法を用いた 2 つの図形間の編集

当りに図形 B を配置するかという目安を与えるために、図 4.8に示すような点線を表示する。ここでは、図形 A の周りとその中心の 9 個所に配置させることができる。マウスボタンを離し、図形 B を図形 A の上に重ね合わせると、大まかな配置を行うためにシステムは自動的にレイアウトを行う。例えば、図形 A の

上下に配置するときには図形 B の横幅をあわせ、

左右に配置する場合には図形 B の高さを合わせる。

中心に配置する場合には図形 A の中に収まるように図形 B を小さくし、

また、斜めに配置する場合には図形 B を半分重ねて配置する

という具合いである。今考えているビュー (図 4.7右) の編集の場合には、このようなレイアウト方式で十分対応可能である。

これらの作業で、前に示したスタックの項 (図 4.1) は**図 4.9**のようにコンパクトで分かりやすい図で再視覚化できる。 単に表示方法を変更するだけではなく、さ

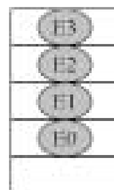

図 4.9: カスタマイズ後のスタックの視覚化

らにこれらの書換えルールで定義された新しいビジュアル表現を利用して等式を記述することができる。各等式にはそれを引き起こす演算があり、これを対応付けて表記する。例えば、スタックからデータを１つ取る作業を行う演算

```
eq pop( push( E:Elt, S:Stack ) ) = S .
```

に対応する演算は pop であり、**図 4.10**のように視覚化できる。 このように等式で

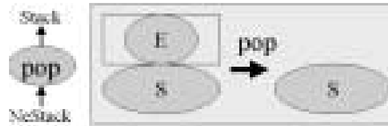

図 4.10: 演算 pop と等式の視覚化

も再視覚化した表現がそのまま利用できるということは、書換えによる実行表示にもこの表現をそのまま用いることが可能であることを示唆している。

4.3 カスタマイズ機能の応用

視覚化は１通りである必要はない。図形書換えルールを用いる利点はプログラム表現を自由に変更できるという点にある。例えば、スタックの例を用いて、他の視点から見た視覚化を考えてみる。

一般に待ち行列はキュー (FIFO) と呼ばれスタック (LIFO) と対峙するデータ構造を意味する。このれらの違いはデータを取り出す順番にあり、視覚的表現に見るだけならば実は大差はない。スタックが空の状態には**図 4.11**のように、行止りを示す表現に置換える。 人の順番待ちは先に並んだ人を優先的に処理するということからも分かるように一般にこれはキューと呼ばれる。 しかし、先に並んだ人ではなく後から並んだ人から処理されるような場合には、これはスタック構造と見る

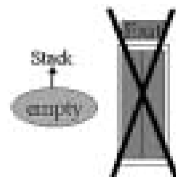

図 4.11: 演算 empty へのビューの定義 その 2

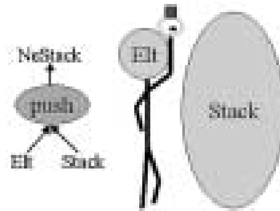

図 4.12: 演算 push へのビューの定義 その 2

ことができる。この人の並びをスタックで実現する場合を考えてみる。以前定義したスタックの視覚化 (図 4.5や図 4.6) の代りに、各演算に対し以下のような視覚化を行う。スタックのデータ構造には、**図 4.12**のようにして人の並びを与える。ここでは、スタックを積む順番をスタックの左側とすることで、スタックを左から右への人の並びとしている。 視覚化方法におけるスタックの要素に人の顔を用いる

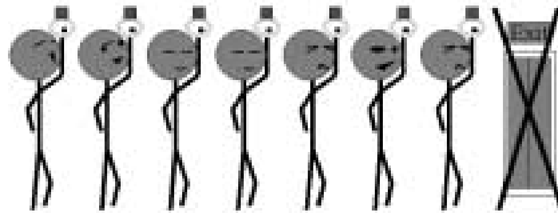

図 4.13: カスタマイズ後のスタックの視覚化 その 2

図 4.14: スタックの要素／顔の表情

ことで**図 4.13**のような視覚化が行える。このとき人の顔はスタックの要素集合として別に定義されているとする (**図 4.14**)。以前の積み木構造によるスタック構造とは異なった表現を行うことが可能になる。ここで変更されるのはビジュアルな情報だけであり、プログラムの内容は変更されない。従ってプログラムの解釈は同じように行うことが可能である。また前に述べたように、ビジュアル表現には、それぞれ CafeOBJ 言語の要素が対応している。このため、同じ表現であっても異なる処理を与えることが可能である。

このように、１つのプログラムをビジュアル的に複数の視点からにとらえることができるため、プログラムの本質的な意味の知る、といった学習などにも役立つ。

4.3.1 ビューの省略表示の定義

ビジュアルプログラミングの１つの問題として、図形が多くなると見ずらくなるという問題がある。これを解決する方法の１つに省略化による方法がある。これは、図形書換えルールを用いることでこれが容易に実現できる。

スタックはデータ構造を格納する箱と考えられるが、実際にプログラミングの最中には、データが入っているということだけが重要したい場合がある。そのスタックという概念やその中身までは問題にしない。そこでスタックの中身は表示せずにその外形のみを表示するようにビューを編集することを考えてみる。

以前定義した図 4.5や図 4.6の代りに、各演算に対し以下のような視覚化を行う。演算 empty は**図 4.15**のように長方形に置換える (今回は単なる長方形ではなく“empty”というラベルを付加している)。

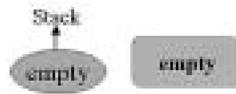

図 4.15: 演算 empty の省略記法による視覚化

演算 push は**図 4.16**右のように省略記法を用いて定義する。この場合、演算 push の引数要素である Elt ／ Stack は 使わない。また、ここでは演算 empty と区別するために、ラベルを “non-empty” とし、箱の右上に装飾を施している。 これによっ

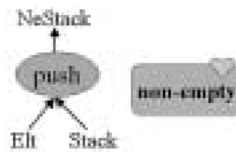

図 4.16: 演算 push の省略記法による視覚化

て、本来は項の一部である演算 push の引数以下にどのような項があっても、同じように表現される。

この書換えルールに基づいて項が表示されると、スタックが空のときは図 4.15に示すルールによって単なる箱でスタックは視覚化される。また、スタックに何かの

要素が入っているときには、図 4.16に示すルールに従ってそれが分かるように装飾された箱が表示される。スタックの中に要素が ある／ない の区別は、図 4.15と図 4.16に示すように箱の左上に装飾があるかないかによって区別することができる。例えば、次のような項

$empty$ ／ $push(3, empty)$ ／ $push(2, push(1, push(0, empty)))$

の視覚化は、それぞれ、図 4.15右／図 4.16右／図 4.16右、のようになる。ここで注目していただきたいのは、項 $push(3, empty)$ と項 $push(2, push(1, push(0, empty)))$ とが同じ表現で表示されることである。このように、図形書換えルールを定義するときに図を配置 する／しない かの切分けにより、簡単に省略化を実現することが可能になる。

4.3.2 ビューの切替え機能

このような省略記法を適用する場合には、元のスタックの中身が表示された以前のビューに戻したい場合がある。これを実現するのが、ビューの切替え機能である。

図 4.17は表示切替を簡単に示した図である。図の中央上部の "rule editing" は、図 4.4の図形書換えルールを示している。図 4.4の右側には OLD VIEW が、右側には NEW VIEW が対応付けられている。ここには基本図形である演算に対する図形の書換えルールが定義されている。図形書換えルールを定義・編集するときのみ両方のビューが表示される (rule editing) が、通常は演算のデフォルトのビューである OLD VIEW(図 4.17の左下) のみを表示する、というようにどちらか一方を表示する。各データ構造に表れる演算部分の視覚化は、この書換えルールのうち、どちらが表示されているかによって決定される。

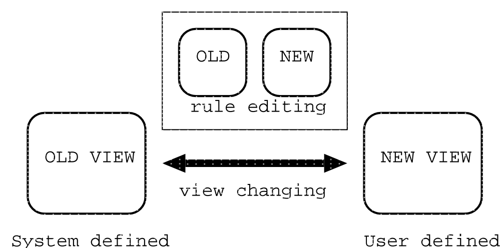

図 4.17: ビューの切替え機能

演算上に定義されているビューを OLD VIEW から NEW VIEW (図 4.17の左下) に切替えることよって、その演算のビューは NEW VIEW に切替わる。このように図形書換えルールが定義された演算のビューが切替えられると、この変更は瞬

時にその演算を持つ各項に反映され、これに対応する全ての演算が NEW VIEW に示すビュー表現に置換えられる。

第 5 章

関連研究

この章では、本研究に関連のある研究について記述する。

5.1 アルゴリズムアニメーション

プログラムを単に視覚化する場合は、アルゴリズムアニメーションに代表されるプログラムビジュアライゼーションの分野がある。プログラムを視覚化するときの問題の 1 つとして、ソフトウェアプローブと呼ばれる視覚化手続きの挿入の手間がある。Pavane[27] では宣言的記述を用いてこの問題を解決しているが、一つ一つ記述して定義するということがまだ手間がかかる。また、Zeus[28] などは 1 つのアルゴリズムに対しダイアログボックスのパラメータを変更することによって複数の表現を適用できることができる。オブジェクトを直接操作して視覚化方法を決めるというよりは、幾つかある視覚化の方法から選ぶ形でその表現を変更する形をとるため柔軟性に欠る。

5.2 ビジュアルプログラミングシステム

Visual[29] は、ビットマップ置換を用いた VPS である。画面上のビットマップ領域に置換規則を表すビットマップ (プログラム) と書換えの対象となるビットマップ (データ) ビットマップをそれぞれ定義する。プログラムの実行は、データビットマップを定義された置換規則を適用して書換えることによって行われる。

ToonTalk[30] は、プログラミングについての知識を持たない人でも、人から教わることなくプログラミングを行なうことができるようにすることを目的とした VPS である。プログラミングは全て漫画の世界で行われる。演算や操作といった要素を漫画のキャラクタに見立て、それらが動く課程がプログラムの実行となる。

現実世界の持つ並列性に着目し、並列言語をモデルにすることでプログラムの表現力向上を図っている。また、プログラムを表現する手段そのものとしてアニメーションを用いたり、ソーシャルインタフェースによってユーザがプログラミング環境に没入できるように工夫している。

TERSE[31] は、視覚的に実現された項書換えのための支援環境である。項の書換え過程の視覚化以外に書換え回数など付加的情報も同様に視覚的に表示することで直観的な解析を工夫し、項書換えの検証システムを実現している。

第 6 章

おわりに

ドラッグ アンド ドロップ手法という直接操作を用いたビジュアルプログラミングシステム CafePie を開発した。プログラム構造はアイコンの組合せによって表現され、項書換えというプログラムの実行においても同じアイコンを用いて視覚化される。単純な操作のみでプログラム編集を行うことで、操作のための学習を減らし直観的な作業を実現した。

　また、ビジュアルプログラミングシステムの特徴である「プログラム構造を直視することで認識し、直観的に理解することができる」ことを生かすために、ビジュアル表現のカスタマイズ機能を実現方法を考察し、 CafePie 上に試作した。このカスタマイズ機能を利用することによって、図形的な意味をプログラムに反映されることが可能になる。一つのプログラムに対して様々な視覚化が可能であるということを例を用いて示した。

謝辞

本研究を進めるにあたり、終始ご指導下さった主査の田中 二郎 教授、および副査の狩野 均 助教授、山本 幹雄 助教授に心から感謝いたします。

情報処理振興協会 (IPA) の CafeOBJ プロジェクトにおいて、二木 厚吉教授や中川 中さん、その他メンバの方々には、大変に貴重なアドバイスをいただきました。また、田中研究室 OB の遠藤浩通さんをはじめとして、研究室のみなさんからはプログラムの視覚化に関して多くの貴重な助言をいただきました。ここに感謝の意を表します。

参考文献

[1] J. Rumbaough, W. Premerlani M. Blaha, F. Eddy and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall International, 1991.

[2] E. J. Golin and S. P. Reiss. The specification of visual language syntax. *Journal of Visual Languages and Computing*, Vol. 1, No. 2, pp. 141–157, 1990.

[3] B. A. Myers. Taxonomies of Visual Programming and Programming Visualization. *Journal of Visual Languages and Computing*, Vol. 1, No. 1, pp. 97–123, 1990.

[4] E. Glinert and S. Tanimoto. PICT: An Interactive Graphical Programming Environment. *IEEE Computer*, Vol. 17, No. 11, pp. 7–25, 1984.

[5] M. Hirakawa, M. Tanaka and T. Ichikawa. An Iconic Programming System, HI-VISUAL. *IEEE Transaction on Software Engineering*, Vol. 16, No. 10, pp. 1178–1184, 1990.

[6] 市川忠男, 平川正人. ビジュアル・プログラミング. *bit*, Vol. 20, No. 5, pp. 404–412, 1988.

[7] 紫合治ほか. パネル討論会 視覚的プログラミング環境 昭和 61 年後期第 33 回全国大会報告. 情報処理, Vol. 29, No. 5, pp. 485–504, 1988.

[8] 竹内郁雄. 「作ってなんぼ」のナラトロジ. 日本記号学会誌, April 1994.

[9] J. Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *ACM Turing Award Lecture*, 1977.

[10] 増井俊之. ビジュアル言語のすすめ. *bit*, Vol. 30, No. 1, pp. 17–19, January 1998.

[11] J. Tanaka. PP : Visual Programming System For Parallel Logic Programming Language GHC. *Parallel and Distributied Computing and Networks '97*, pp. 188–193, August 11-13 1997. Singapore.

[12] M. Toyoda, B. Shizuki, S. Takahashi, S. Matsuoka and E. Shibayama. Supporting Design Patterns in a Visual Parallel Data-flow Programming Environment. In *Proceedings of the 1997 IEEE Symposium on Visual Languages*, 1997.

[13] J. A. Goguen, J. W. Thather, E. G. Wagner and J. B. Wright. Abstract Data Types as Initial Algebras and the Correctness of Data Structure, 1975.

[14] J. A. Goguen and J. Meseguer. Order-Sorted Algebra 1: Equational Deducation for Multiple Inheritance, Polymorphism, Overloading and Partial Operations. Technical report, SRT International, 1989.

[15] R. Diaconescu and K. Futatsugi. *CafeOBJ Report*. World Scientific, 1998.

[16] A. T. Nakagawa, T. Sawada and K. Futatsugi. *CafeOBJ User's Manual*. IPA, 1997.

[17] 小川徹. Drag and Drop に基づく代数的仕様記述言語のための視覚的プログラミング環境, 1997. 筑波大学卒業論文.

[18] 小川徹, 田中二郎. Drag and Drop 手法を用いた代数的仕様記述言語における視覚的プログラミング環境. 日本ソフトウェア科学会 第 15 回大会論文集, pp. 165–168, 1998.

[19] T. Ogawa and J. Tanaka. Double-Click and Drag-and-Drop in Visual Programming Environment for CafeOBJ. In *Proceedings of International Symposium on Future Software Technology (ISFST'98)*, pp. 155–160, Hangzhou, October 28-30 1998.

[20] B. Shneiderman. Direct Manipulation: A Step Beyond Programming Languages. *IEEE Computer*, Vol. 16, No. 8, pp. 57–69, 1983.

[21] A. Wagner, P.Curran and R. O'Brien. Drag Me, Drop Me, Treat Me Like an Object. In *Proceedings of CHI'95: Human Factors in Computing Systems*, pp. 525–530, 1995.

[22] 久野靖, 大木敦雄, 角田博保, 粕川正充. 「アイコン投げ」ユーザインターフェース. コンピュータソフトウェア, Vol. 13, No. 3, pp. 38–48, 1996.

[23] T. Ogawa and J. Tanaka. Realistic Program Visualization in CafePie. In *Proceedings of IDPT-99*, 1999. (to appear).

[24] 小川徹, 田中二郎. CafePie: 図形の組合せを用いた CafeOBJ の視覚化. 日本ソフトウェア科学会 第 16 回大会論文集, pp. 65–68, 1999.

[25] ソニー・テクトニクス (株) AI 事業部抄訳, 訳校閲 竹内郁雄. Smalltalk-80 によるアプリケーションプログラムの作り方. *bit*, Vol. 18, No. 4, pp. 379–395, April 1984.

[26] 青木淳. 使わないと損をする Model-View-Controller -Smalltalk の設計指針 -. 富士ゼロックス情報システム株式会社、資料, 1988.

[27] K. C. Cox and G.-C. Roman. Visualizing Concurrent Computations. In *Proceedings of the 1991 IEEE Workshop on Visual Languages*, pp. 18–24, 1991.

[28] M. H. Brown. Zeus: A System for Algorithm Animation and Multi-View Editing. In *Proceedings of the 1991 IEEE Workshop on Visual Languages*, pp. 4–9, October 1991.

[29] K. Yamamoto. Visulan: A Visual programming Language for Self-Changing Bitmap. In *Proceedings of International Conference on Visual Information Systems*, pp. 88–96, Melborune, 1996.

[30] K. Kahn. ToonTalk(TM) – An Animated Programming Environment for Children. *Journal of Visual Languages and Computing*, June 1996.

[31] 河口信夫, 坂部俊樹, 稲垣康善. 項書き換え系の解析・検証・変換のための視覚的支援方法. コンピュータソフトウェア, Vol. 13, No. 1, pp. 23–36, 1996.