ビジュアルプログラミングにおける
視覚情報のカスタマイズ手法に関する研究

工学研究科

筑波大学

2003 年 3 月

小川 徹

要旨

ビジュアルプログラミングでは，図式表現を用いてプログラムを記述したり，図式表現を媒介としてプログラムの編集や実行を行うことが広く行われている．しかし，図式表現の編集はテキストと比べて操作が複雑であり，テキストを用いたプログラミングに比べて手間が掛かるという操作における問題がある．また，全ての情報を図式表現により過不足なく表現すれば全てが理解しやすくなるというわけではない．逆に，ユーザに対して過大な情報を提供するという視覚化における問題が発生する可能性もある．

本論文では，ビジュアルプログラミングにおける作業を容易にするために，操作における問題と視覚化における問題を解決し，使いやすいビジュアルプログラミングを提案することを目的とする．まず，代数的仕様記述言語においてプログラム編集上のすべての操作を図式表現への直接操作によって実現できることを，システム CafePie を実装することによって示した．図式表現への直接操作には，ドラッグ＆ドロップ操作によるノード指向の操作により実現した．これによりテキストエディタと同じくらいの時間でプログラムの入力ができることを実験により確認した．また，情報の提示や把握における問題の解決のために使用する場面に応じて表示方法の変更を可能とする視覚情報のカスタマイズが必要となる．視覚情報のカスタマイズにおいては，操作結果を対話的にユーザに提示することにより直接操作を用いて視覚情報のカスタマイズが行えるようにした．この直接操作による視覚情報のカスタマイズをシステムに実装することで，直接操作を用いて手早く行えるようになったことを確認した．最後に，視覚情報のカスタマイズをプログラムをプログラム実行に応用することで，ソースコードに直接書く込むことなくプログラム実行における視覚情報を変更できることを，システムに実装することで確認した．また，プログラム実行表示にアニメーションの定義を視覚的に行えるようにした．アニメーション定義上で視覚情報を変更できるため，動きによる強調以外にも色を変更するような見た目の強調も可能であることが言える．

本論文で述べられている，ノード指向の直接操作，視覚情報のカスタマイズにおける直接操作を用いた配置方法は一般にも有用であると思われる．また，プログラム実行におけるアニメーションの設定は，前後置換ルールを用いて動作を定義するようなビジュアルプログラミングシステムにも適用可能である．

目次

1 序論 **1**

1.1 研究の背景 1

1.2 研究の目的 1

1.3 本論文の構成 2

2 準備 **3**

2.1 ビジュアルプログラミングとその応用範囲 3

2.1.1 ビジュアルプログラミングの分類 5

2.2 システム実装のための言語 6

2.3 代数仕様記述言語 CafeOBJ 7

2.3.1 CafeOBJ のプログラム記述 8

2.3.2 CafeOBJ 言語におけるプログラム実行 11

3 直接操作を用いたプログラム編集 **16**

3.1 操作における問題点 16

3.2 直接操作とドラッグ&ドロップ手法 17

3.3 ビジュアルプログラミングシステム “CafePie” 19

3.3.1 ビジュアルシンタックスの導入 21

3.3.2 ドラッグ&ドロップ手法への操作の割り当て 24

3.3.3 ラベルの扱い 25

3.4 ノード指向の操作 26

3.4.1 ソートにおける関連付け 27

3.4.2 変数の具体化による項の編集 28

3.4.3 項の編集における応用 30

3.5 CafePie におけるプログラムの作成例 30

3.5.1 ソートの作成 31
3.5.2 演算の作成 32
3.5.3 変数の作成 32
3.5.4 等式の作成 33
3.5.5 ドラッグ&ドロップ操作における考察 34
3.6 プログラム編集における評価実験 34
3.6.1 実験の被験者と実験環境 34
3.6.2 実験の手順 35
3.6.3 実験の結果と考察 35

4 視覚情報のカスタマイズ手法 38
4.1 視覚化における問題点 38
4.2 視覚情報のカスタマイズについて 38
4.2.1 視覚情報の実装モデル 39
4.2.2 カスタマイズ手法の実装方針 40
4.3 データ構造における視覚情報のカスタマイズ 42
4.3.1 データ構造のカスタマイズ 42
4.3.2 デフォルト・ビュー 43
4.3.3 ビューの対応画面 44
4.3.4 視覚情報のカスタマイズの作成例 44
4.3.5 カスタマイズ後における項の編集 46
4.3.6 視覚情報のカスタマイズ機能の適用 47
4.4 ドラッグ&ドロップ手法を用いたカスタマイズ 49
4.4.1 カスタマイズで行うこと 49
4.4.2 ドラッグ&ドロップ手法を用いる場合の問題点 50
4.4.3 カスタマイズの適用範囲 51
4.4.4 ビューのカスタマイズに関する関連研究 52
4.5 視覚情報のカスタマイズ手法における評価実験 52
4.5.1 実験の被験者と実験環境 53
4.5.2 実験の手順 53
4.5.3 実験の結果と考察 53

5 視覚情報のカスタマイズとプログラム実行 **55**
5.1 CafePie におけるプログラムの実行処理 55
5.2 プログラムの実行例 57
5.3 視覚情報のカスタマイズを用いたプログラム実行 58
5.3.1 プログラム実行における視覚情報の利用 59
5.3.2 プログラム実行のアニメーション表示 60
5.4 他の例題 62
5.4.1 新しい等式の作成 62
5.4.2 ゴールの作成とビューの変更 63
5.4.3 ビューの反映について 65
5.4.4 アニメーションの設定 66

6 結論 **69**

謝辞 **70**

参考文献 **71**

著者論文リスト **78**

図一覧

2.1 CafeOBJ 言語による仕様記述例 8
2.2 CafeOBJ インタプリタによる仕様記述の実行 12
2.3 CafeOBJ トレーサによるトレース表示 14

3.1 ドラッグ&ドロップ操作 18
3.2 CafePie の実行画面 20
3.3 CafePie におけるソートの視覚化 22
3.4 CafePie における項の表現 23
3.5 CafePie における演算の視覚化 23
3.6 CafePie における変数の視覚化 23
3.7 CafePie における等式の視覚化 24
3.8 ソート間の関係付け 28
3.9 項の作成手順 29
3.10 CafePie 上のソート定義例 31
3.11 CafePie 上の演算定義例 32
3.12 CafePie 上の変数定義例 33
3.13 CafePie 上の等式定義例 33
3.14 プログラム編集における評価 35

4.1 MVC モデル 40
4.2 演算 push に対する視覚化 41
4.3 システムによるスタックの視覚化 43
4.4 ビューの対応画面 44
4.5 スタックに対する新しい視覚化 45
4.6 演算 empty へのビューの定義 45
4.7 演算 push へのビューの定義 45

4.8 カスタマイズ後のビューを用いた項の編集（追加） 46
4.9 カスタマイズ後のビューを用いた項の編集（削除） 47
4.10 演算 empty へのビューの定義（人の並び） 47
4.11 演算 push へのビューの定義（人の並び） 48
4.12 カスタマイズ後のスタックの視覚化（人の並び） 48
4.13 スタックの要素／顔の表情 48
4.14 図式表現の関係の種類 50
4.15 ドラッグ＆ドロップ操作に応じたフィードバック表示 51
4.16 スペースを開けた配置 51
4.17 プログラム編集における評価 54

5.1 インタプリタの URL 指定用ダイアログ 56
5.2 動的な実行の視覚化 57
5.3 静的な実行の視覚化 58
5.4 ビューの切替えと等式の表現 60
5.5 カスタマイズ表現を用いた実行表示 60
5.6 アニメーションの補間設定 61
5.7 補間されたアニメーションの表示 61
5.8 ソーティング用の等式 62
5.9 ソーティングにおけるゴール 63
5.10 ソーティングでの演算 push 64
5.11 ソーティングでの演算 empty 64
5.12 等式のカスタマイズ結果 64
5.13 実行のアニメーション表示 65
5.14 ビューの反映 66
5.15 ソーティングにおけるアニメーション調整 66
5.16 演算 push に定義したマルチビュー 67
5.17 カスタマイズを利用した実行のアニメーション 67

表一覧

2.1　トレーサによる項書換えの実行 15

3.1　ドラッグ&ドロップ手法によるプログラム編集 24

第 1 章

序論

1.1 研究の背景

近年，コンピュータの情報処理能力は上がったが，人がコンピュータを使いこなせないという問題が表面化してきている．人の情報処理能力を拡張するためには，コンピュータ自体の情報処理能力を上げるだけでなく，人とコンピュータとのコミュニケーションを円滑に行いながら人が行いたいことをコンピュータに伝達するための手段が必要である．人とコンピュータとのコミュニケーションを円滑にしながら人が行いたい処理をコンピュータに伝達するための手段として，ビジュアルプログラミング [1, 2, 3, 4, 5, 6] が注目されている．

ビジュアルプログラミングでは，図式表現を用いてプログラムを記述したり，図式表現を媒介としてプログラムの編集や実行を行うことが広く行われている．しかし，図式表現の編集はテキストと比べて操作が複雑であり，テキストを用いたプログラミングに比べてプログラムの作成や編集に手間が掛かるという**操作における問題**がある．また，全ての情報を図式表現により過不足なく表現すれば全てが理解しやすくなるというわけではない．逆に，ユーザに対して過大な情報を提供するという**視覚化における問題**が発生する可能性もある．したがって，この操作における問題と視覚化における問題を解決し，プログラマの負荷を少なくするようなビジュアルプログラミングシステムを実現することが望まれている．

1.2 研究の目的

本論文では，使いやすいビジュアルプログラミングを提案することを目的とする．ビジュアルプログラミングにおける作業を容易にするために，操作における問題と視

覚化における問題を解決する．まず，操作における問題のために，メニューやダイアログボックスのような従来の Graphical User Interface（GUI）を用いなくて済むような，図式表現への直接操作を実現する．そして，実現した直接操作のみを用いたプログラム編集が可能であることを，実際にシステムを試作することで示す．次に，視覚化における問題を解決するために，状況に応じて適切に視覚化を行えるような視覚情報のカスタマイズ手法を提案する．また，提案した視覚情報のカスタマイズ手法がプログラム編集時だけでなくプログラム実行時にも利用できることを，実際にシステムに実装することで示す．

1.3 本論文の構成

本論文の構成は次のようになる．まず第 2章では，準備としてビジュアルプログラミングとその応用分野，本論文のシステムで用いている CafeOBJ 言語について述べる．次に第 3章において，直接操作を用いたプログラム編集について述べる．第 4章では，視覚情報のカスタマイズ手法について述べる．また，第 5章では，視覚情報のカスタマイズを利用したプログラムの実行について述べる．第 6章において本論文をまとめる．

第 2 章

準備

2.1　ビジュアルプログラミングとその応用範囲

ビジュアルプログラミングとは，**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．ビジュアルプログラミングではアイコンやアニメーションといった図式表現を用いてプログラムを記述すること，記述したプログラムを図式表現を用いてプログラムを編集すること，また，プログラムの実行過程や結果そのものを視覚的に表示することが行われる．一般に，事物を表現するには図で表現することが適している場合が多く，回路図や設計図，地図など「図」が用いられることが多い．具体的な事物を扱う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．

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というビジュアルプログラミングは，コンピュータの世界を実世界に近づけようとする試みであると言うことができる．

ビジュアルプログラミングの特徴として次のようなことが挙げられる．

- 具体性
  色や形などをで表現した方が好ましい図式表現は，図式表現として表現する方が良い．例えば，矩形や楕円や星型を Rect や Oval や Star といったテキスト表現で表すよりも具体的な図形として表示する方が実際の形が把握しやすい．

- 複雑な構造の表現
  テキスト言語は複雑な構造や関係を表現すること可能であるが，テキストは一

次元的であるためにテキストで記述した複雑な表現を人が理解することは困難である．図式表現を用いて複雑な構造や関係をより分かりやすく記述できるようになる．例えば，プログラムの構造を視覚化する手段としてフローチャートや NS チャート，データの関連や構造を視覚化した実体関連図などが知られている．

- データの流れの表現
  データフロー図のようにデータの処理をノードとしてデータの流れを矢印として表現することで，データの流れを明示的に表現することができる．視覚化を工夫することで，注目したいもの（データの流れ）の状態をうまく捉えることができるようになる．

- プログラムの動的な表示
  テキストを用いたプログラムは静的な記述に成らざるを得ないため，プログラマは頭の中にある実行時の振る舞いを想像しながらプログラミングを行う必要がある．一方，ビジュアルプログラミングでは，プログラムの実行を，例えばアルゴリズムアニメーションのように，アニメーションを用いて具体的に表示することによって，よりプログラムの理解が深まるようになる．

ビジュアルプログラミングを行うシステムとして以前から Pict[7] や Hi-VISUAL[8] などが試作されている．また，並列論理型言語 GHC[9] の構造および実行過程を視覚化するシステム PP[10] や並列論理型言語 Janus[11] を視覚化してその実行過程の視覚化を行うシステム Pictorial Janus[12] のようなシステムがある．また， BITPICT[13] や Visulan[14] などのようにはじめからビジュアル言語として設計されたものや，例示プログラミング環境としての KIDSIM[15] などがある．ソフトウェア仕様の分野では，コンポーネントベースようにしてビジュアルプログラミングが利用されている [16, 17]．計算機シミュレーションの分野でも，エンジニアのための計測制御／データ収集などを作成にビジュアルプログラミングを用いたシステムである LabVIEW[18] などが知られている．プログラムの動作をわかりやすく表示する手法として，プログラムビジュアライゼーションがある．プログラムの実行を視覚化したアルゴリズムアニメーションがあり， Zeus[19]， Pavane[20] などが試作されている．

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，ビジュアルプログラミングは**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 [1, 2]. 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，設計，コーディング，テスト，運用，保守のフェイズから成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．例えば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，ビジュアルプログラミングの分類として，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 Visual Programming と 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．一つ目の基準にある例示とは，プログラマが例を示すことであり，例示した結果を元にシステムが推論を行って自動的にプログラムを作成してくれる．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．例示システムではないものとして Pict[7]，例示システムである例として HI-VISUAL[8] が知られている．我々も必要に応じて例示システムの考え方を取り入れることが重要であると考えている．二つ目の基準は，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．ビジュアルプログラミング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り入れている．先ほど挙げた Pict や HI-VISUAL は

どちらもインタプリティブなシステムである．我々の立場もインタプリタベースのシステムにある．また，システムを実装するに当たっては既存の言語の処理系を利用することを考えている．既存の処理系がコンパイラベースやインタプリタベースであっても，ユーザにとってインタプリティブに行えることが重要であると考えている．

2.2 システム実装のための言語

ビジュアルプログラミングシステムには様々あるが，本論文中で扱うシステムではベースとなる言語を元にシステムを設計することを考えた．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たからである．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，ここで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．
  ビジュアルプログラミングは視覚的な表現を多く用いるが，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，なるべく図式表現を使った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 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 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 [21].

  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 [22] であり，手続き型言語のおおもとのフォン・ノイマン型プログラムと宣言型言語の一つである関数型プログラム FP について述べている．この例として内積のプログラムがある． 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．これは一つの例であるが，一般に宣言型言語は手続き型言語に比べてプログラム要素数が比較的少なくなるという特徴を持つ．

- 図式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と言われている [23]．プログラムを図形で表現した場合には各図形は画面上にまばらに配置されるが，何かしらの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の解釈における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[10] や KLIEG[24] といった様々なビジュアルプログラミングシステムが試作されている．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仕様記述言語を選択した．

2.3 代数仕様記述言語 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 [25]．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 [26] が意味論の基盤である．また，形式的仕様記述言語は仕様を記述するために用いられるが， CafeOBJ 言語は記述した仕様を実行することも可能である．仕様の実行は項書換えによって行う．

CafeOBJ 言語 [27, 28, 29, 30] は，代数仕様記述言語の 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．

2.3.1 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．

```
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 [Eq1] : 0 + N = N .
    eq [Eq2] : N + s(M) = s(N + M)
  }
}
```

図 2.1: CafeOBJ 言語による仕様記述例

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 言語では次のように書く．

```
[ 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ることで，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た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のプログラム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 つのアリティであるソート 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 Eq1 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 Eq2 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．また，図 2.1のプログラムにはないが,

```
ceq [ ラベル ] : 右辺項 = 左辺項 if 条件項 .
```

のようにして条件付き等式を記述することができる．条件付き等式は，条件項の結果が true となった場合にのみ有効になる等式である．

2.3.2 CafeOBJ 言語におけるプログラム実行

代数的仕様記述言語で書かれた仕様は項の書換え（Term Rewriting）によって実行される． 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(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．

```
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

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. 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（リデックス）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．

2. 各等式の左辺と一致する部分が見つかれば，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．

3. 以上の 1，2 を処理が終了するまで繰り返す．

この書換え過程を順に実行していくと

```
s(s(0)) + s(s(s(0)))  =>   s(s(s(0)) + s(s(0)))
                      =>   s(s(s(s(0)) + s(0)))
                      =>   s(s(s(s(s(0) + 0))))
                      =>   s(s(s(s(s(0)))))
```

のようになる．最初の入力として与えられた項 $s(s(0)) + s(s(s(0)))$ は最終的に項 $s(s(s(s(s(0)))))$ となる．このとき項 $s(s(s(s(s(0)))))$ 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．

```
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(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

1>[簡約番号] 適用された等式から得られる（書換え）ルール
　　{ 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
$\underline{s(s(0)) + s(s(s(0)))}$		
$\Downarrow$	[Eq2] $N + s(M) = s(N + M)$	$N \Rightarrow s(s(0)), M \Rightarrow s(s(0))$
$s(\underline{s(s(0)) + s(s(0))})$		
$\Downarrow$	Eq2] $N + s(M) = s(N + M)$	$N \Rightarrow s(s(0)), M \Rightarrow s(0)$
$s(s(\underline{s(s(0)) + s(0)}))$		
$\Downarrow$	[Eq2] $N + s(M) = s(N + M)$	$N \Rightarrow s(s(0)), M \Rightarrow 0$
$s(s(s(\underline{s(s(0)) + 0})))$		
$\Downarrow$	[Eq1] $0 + N = N$	$N \Rightarrow s(s(0))$
$s(s(s(s(s(0)))))$		

表 2.1: トレーサによる項書換えの実行

第 3 章

直接操作を用いたプログラム編集

3.1　操作における問題点

ビジュアルプログラミングは，視覚的な手段を介してプログラミングを行う利用者からコンピュータへプログラミングに必要な情報が伝達されるというプログラム作成方法と捉えることができる．ビジュアルプログラミングにおいて，プログラマは図式表現を通してプログラムの状態を把握し図式表現を介してプログラムの修正を行う，という作業を繰返しながらインタラクティブにプログラムを作成していく．テキスト表現を用いたプログラミングでは，特別なツールが無くともテキストエディタを使用することで手軽にプログラムを記述することができる．しかし，図式表現を用いたプログラミングでは，ドローツールを使って図式表現を編集することも可能であるが，一般に，図式表現の編集には操作が複雑であり，テキスト表現を用いたプログラミングに比べて手間が掛かる．

既存の GUI を用いてシステムを作成すると「必要に応じて操作に必要な情報を提示でき，操作における負担を軽くなる」とされている．例えば，メニューやダイアログボックスを使う場合には，操作できることがメニューの中に一覧として表示されていたり，操作に必要な情報がすべてダイアログボックスの中に見えるようになっている．しかし，このような既存の GUI を用いてシステムを構築すると，一つ一つの操作に手間が掛かり，操作における負担が軽くならない場合もある．ドローツールなどで図形のサイズを調整する場合は以下のような操作手順となる．

1. 図形のサイズを調整する場合には，図形を選択してメニューから図形のプロパティの一覧を示したダイアログボックスを表示させる．この場合，ダイアログボックスが操作したい図形の上に重なってしまい，操作対象が見えなくなる場

合もある．

2. 図形のサイズはダイアログボックスの中に具体的な数値として表示されている．すなわち，操作に必要な情報はテキストとして表示されている．

3. 図形のサイズを選択して具体的な値を入力し，OK ボタンなどを押すことで操作の結果が図形に反映させる．この場合には，操作している具体的な値以外に実際に編集しようとしている図形のサイズも変化する．

このような操作方法では，サイズが思ったよりも小さすぎたりすると，もう一度同じ操作を繰り返す必要があり非常に手間がかかる．図形を直接選択したらそのまま図形の端をドラッグしてサイズを調整した方が手早く行えるし，何回も調整する場合に手間が掛からなくて良い．このような操作の増加を避けるためには，操作対象をそのまま操作できるようにすべきである．特に GUI を用いてプログラムを行う場合には，メニューやダイアログボックスなどの既存の GUI を用いないで済むような操作手法を考える必要がある．

3.2 直接操作とドラッグ＆ドロップ手法

直接操作（Direct Manipulation）[31] とは，例えば，対象となる図式表現をポインティングデバイスなどを用いて直接的に操作することによって，複雑な構文を持つコマンドを実行する手法である．直接操作を用いたシステムでは，ユーザがインタフェースを介して操作対象となるデータや実体に働き掛けることができる．この働き掛けは，データや実体にあたかも直接触れている感じをユーザに与え，その触れているという操作結果の反応が直接的にユーザに返っていく，というような 2 つのステップから成り立っており，ユーザは具体的な変化として操作を知覚できる．直接操作は GUI を扱うシステムにおいて重要な概念であり，図式表現の編集という手間の掛かる作業を効率良く行うためにも必要不可欠である．

マウスを用いて図式表現を操作する直接操作手法として，ドラッグ＆ドロップ手法 [32] が知られている．ドラッグ＆ドロップ操作は，図式表現を選択する操作とドラッグという図式表現を移動する操作とドロップという図式表現を手放す操作 3 つの操作を一連の動作として行う操作手法であり，ドロップ時に操作している図式表現を他の図式表現と重ね合わせることで 2 つの図式表現間の関連付けを定義できる．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て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では文章中の文字列やセルにある値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，編集に費やす思考が少なくなると考えられる．

ビジュアルプログラミングでは，アイコンを用いてプログラムを記述する．ビジュアルプログラミングにおいてドラッグ＆ドロップ操作（図 3.1）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 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 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あり， 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．

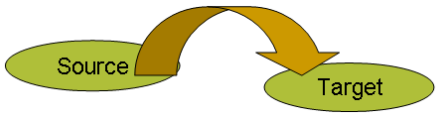


図 3.1: ドラッグ＆ドロップ操作

本研究では，このドラッグ＆ドロップ操作を用いて効率良くプログラムを作成することを考えた [33, 34, 35].

3.3 ビジュアルプログラミングシステム “CafePie”

ドラッグ&ドロップ手法を用いてプログラム編集が行えることを示すために，ビジュアルプログラミングシステム CafePie[33, 34, 35, 36, 37, 38] を試作した． CafePie は “Pictorial Interactrive Environment for CafeOBJ” の略である． CafePie は，代数仕様記述言語 CafeOBJ における各モジュールを視覚化／編集でき，また， CafeOBJ の一つの側面である項書換え系（Term Rewriting System， TRS）を視覚化することでプログラム実行を可能にするシステムである．ビジュアルプログラミングシステムとしての CafePie に期待される機能を整理すると，以下のようになる．

- 図式表現による入力
  ユーザは図式表現を組み合せる形で代数的仕様記述言語の各基本要素を入力／作成する．図式表現の組み合わせはマウスによる直接操作のみで編集できる．

- プログラムコードの自動生成
  図式表現でプログラムを編集した結果は，図式表現に対応した CafeOBJ プログラムのテキスト表現を生成することができる．

- 図式表現の自動生成
  逆に CafeOBJ プログラムのテキスト表現を入力することで，テキスト表現に対応する図式表現が自動生成され，画面上に表示されるこれは，例えば単に演算を定義する文字を入力することで，入力した文字をすぐに解釈して対応する図式表現を表示する機能である．

- 修正機能
  図式表現は，画面上で閲覧しながら修正できる．この機能を用うことで，ユーザはテキスト表現から自動生成された図式表現を修正し，プログラム編集を視覚的に行える．

- 保存／再生機能
  図式表現で編集したプログラムはファイルに保存でき，また，必要なときに呼び出すことができる．

- プログラム実行機能
  入力された図形式表現は，それを図形的に表示し，実行することができる．こ

のとき，実行の処理系には 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
  CafePie はプログラムの作成／編集／実行をビジュアル的な形で支援する統合環境である．

CafePie は当初 Java1.0 ベースで開発を進めていたが， Java1.1， 1.2 を経て，現在は Java 1.4.1 ベースへと移行している（ここで言う Java は， Sun Microsystems 社の $Java^{TM}2$ SDK, $Standard\ Edition$ を指す）． このシステムは Java のアプリケーションとして実装している． CafePie を起動すると，図 3.2に示すような画面が表示される．

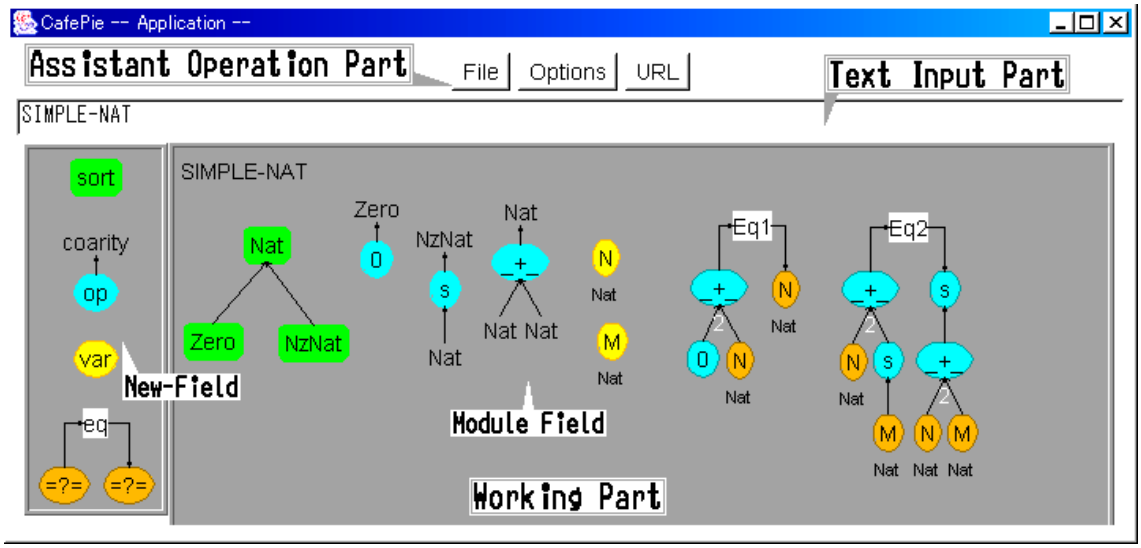


図 3.2: CafePie の実行画面

画面の半分以上を広く占有する部分が “Working Part” であり，プログラムの編集作業に用いられる． “Working Part” は “Module Field” と “New-Field” から構成される． “Module Field” では，現在編集しているプログラムのモジュールを図式表現を用いて視覚的に表示される．ユーザは “Module Field” の中に表示された図式表現を用いてプログラムのモジュールを視覚的に編集することができる．また “New-Field” は， “Module Field” に新しく図式表現を追加するために用いられる． “Working Part” の上部にある “Text Input Part” は， “Working Part” の中の図式表現に含まれるラベルの編集に用いられる．ユーザが図式表現を選択すると，選択中した

図式表現のラベルが “Text Input Part” の中に表示される．ユーザは “Text Input Part” で入力した文字を確認しながら自由に図式表現のラベルを変更することができる．“Text Input Part” の上部にある “Assistrant Operation Part” はボタン列から成っており，プログラム編集以外の補助的な操作を行うために用いられる．例えば プログラムをロードする場合には，画面上部のボタン列内の “File” ボタンを押し，ファイルを選択することで行う．選択するファイルは図 2.1で示したテキスト表現で書かれた CafeOBJ のプログラムファイルである．ファイルを選択すると， CafePie はそのテキスト表現で書かれたプログラムを解釈してシステムの内部表現に置き換え，“Working Part” 内の “Module Field” に表示する．結果として， “Module Field” には図 3.2に示すような図式表現で視覚化されたプログラムが表示される．

CafePie 内におけるプログラム編集に関するすべての操作は，メニューやダイアログボックスなど既存の GUI を用いないでも行えるような直接操作によって実現されている．プログラム編集を直接操作を用いて実現するために「**ノード指向による操作**」を提案している．このノード指向の操作を説明する前に，まず，操作を実現するための基盤となっている，「ビジュアルシンタックスの導入」，「ドラッグ＆ドロップ手法への操作の割り当て」，「ラベルの扱い」について説明する．

3.3.1 ビジュアルシンタックスの導入

テキスト表現で書かれたプログラムをビジュアルプログラミングシステム上で扱うためには，何らかの方法を用いてテキストからシステム上の要素である図式表現に変換する必要がある．

ここでは，視覚化されたプログラムの構成要素を表す図式表現のことを「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．つまり，「プログラミング言語における基本要素は，システム上で決められたアイコンを用いて視覚化される」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． CafePie ではこれらの 4 つの要素（ソート／演算／変数／等式）を視覚化の対象とし，それらに対してビジュアルシンタックスを定義している．また，単なる要素名を画面上に表示するのではなく，種類別に異なる形と色を与え，その中にラベルを付加したアイコンとして表示している．種類わけされたラベル付きアイコンを用いることで，各要素を視覚的に識別することが可能

になる．

各要素のビジュアルシンタックスは次のようになる．

ソート： ソートは CafeOBJ 言語における型を表す． これは， C 言語における整数

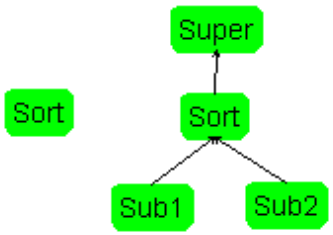


図 3.3: CafePie におけるソートの視覚化

型や文字型などに相当するものである．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 ではソートを 図 3.3 に示すように，ソートのラベルを頂点，ソート間の関係を有向線とした有向グラフで表す．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で結ぶ．図 3.3のテキスト表現は

```
[Sort, Sub1 Sub2 < Sort < Super]
```

となる．

項： 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木を用いる（ここでいう木とは，組織図などの階層構造を表すのによく用いられる根付き木を指す [39]）．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 3.4）．

演算： 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 3.5）．

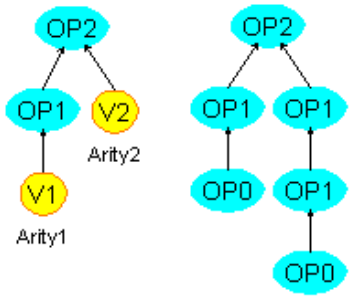


図 3.4: CafePie における項の表現

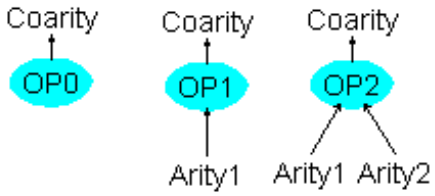


図 3.5: CafePie における演算の視覚化

変数： 変数は等式を記述するときにのみ使用されるが，演算のように具体的な値が決まっていない任意の項として捉えることができる．そこで，演算を項の構成要素である演算と同じ形で，かつ色違いというシンタックスを与える．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 3.6）．

図 3.6: CafePie における変数の視覚化

等式： 等式は項を書き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 3.7）．左辺から右辺に書き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．

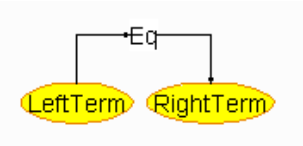


図 3.7: CafePie における等式の視覚化

CafeOBJ 言語において，プログラムはモジュールの集まりで表現される．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成る．モジュールは灰色の矩形で表し，その中にモジュールラベルとこれら 4 種類の基本要素を表示する．ユーザはこのモジュール上でプログラムの定義を行うことができる．

3.3.2 ドラッグ＆ドロップ手法への操作の割り当て

CafePie 上に表示されたアイコンは，マウスを用いて Module Field 内を自由に動かすことができる．プログラム編集におけるドラッグ＆ドロップ（DnD）へ対応は表 3.1のようになる．この対応表は，ドラッグ元である “Source” とドラッグ先である “Target” と重なったときに実行される動作 “Action” の組から成り，各々の組に動作名 “Action Name” がラベル付けされている．

表 3.1: ドラッグ＆ドロップ手法によるプログラム編集

Action Name	Source	Target	Action
ソート関係の作成	ソート	ソート	DnD 先を元の上位ソートとする
ソート関係の削除	ソート	ソート	ソート間の関係削除
引数の追加	ソート	演算	演算に引数を追加する
引数の変更	ソート	引数	引数を別のソートに変更する
引数の交換	引数	引数	引数の順番を入替える
項の生成	演算	変数	演算から項を生成し，変数を上書きする
項の追加	項	変数	項で変数を上書きする
変数名の変更	変数	変数	DnD 先のラベルを DnD 元と同じにする

CafePie 上でプログラムを作成する場合には， CafeOBJ プログラムのファイルを読み込む以外に一から編集することも可能である．その場合の手順は次のようになる．

(a) モジュール名の編集：
図 3.2の Module Field 上をクリックしてモジュール名を入力する．

(b) ソートの作成（図 3.2の Module Field 左端）：
New-Field 上のソートアイコンを移動し Module Field 上にコピーする．その後で各ソートの関連付けを作成する（表 3.1参照）．

(c) 演算の作成（図 3.2の Module Field 左中）：
New-Field 上の演算アイコンを移動し Module Field 上にコピーする．ソートアイコンを用いて 演算の型や引数を決める（表 3.1参照）．

(d) 変数の作成（図 3.2の Module Field 右中）：
New-Field 上の変数アイコンを移動し Module Field 上にコピーする．ソートアイコンを用いて 変数の型を決める（これは以前の操作手法であり，現在は対応するソートをダブルクリックすることで変数の作成とソートとの対応付けを同時に行うという手法を用いて変数を作成する）．

(e) 等式の作成（図 3.2の Module Field 右端）：
New-Field 上の等式アイコンを移動し Module Field 上にコピーする．右辺，左辺に当たる項を作成する．

(f) 必要に応じて各アイコンをクリック選択しラベルを変更する．

3.3.3 ラベルの扱い

従来のテキスト言語を用いたプログラミングでは，名前付けが重要となってくる．データを参照するには必ず名前を付けなければならない． しかし，簡潔で分かりやすく，しかも役割や属性を表す名前を付けるのは難しい．本システムでもラベル付きアイコンを扱っており，同じような名前付けに関する問題がある．アイコンの生成と同時にラベルの付加を強制するのでは，名前が決まっていないとプログラムが書けないということに成りかねない．この問題に対して，本システムでは必ずしも名前付けをはじめに行う必要はないようにしてある．本システムでは，アイコン生成時にあらかじめ適当な名前を与えておき，好きなときに修正できるようにして柔軟性を持たせている．

- **ラベルの自動付加：**キーボードから文字を入力しなくても，変数名やソート名などの全てのラベルはアイコン生成と同時に自動的に付加される．例えばソー

トのラベルは，生成した順に Sort1， Sort2， Sort3，… となる．また，システムは既に定義しているラベル情報を全て把握しているため，アイコンの作成時に既存のラベルを調べて，新しく生成するラベルが既存のラベルと重複しないようにしてある．このためキーボードを用いなくてもプログラミングが可能である．

- **ラベルの変更：** また，ユーザは必要に応じてキーボードを用いてラベルを変更することが可能である．ユーザが誤ってラベルを変更しない限りはラベルの重複は起こらない．キーボードから直接入力できるラベルは，モジュール上に定義されたソート，演算，変数，等式のみに限定している．他に演算の引数や返却値や変数のソート，等式の両辺の項に現れる演算や変数のラベルを変更することも考えられるが，スペルミスを防止するためにもキーボードからの直接編集はできないようにしている．演算の引数や変数のソートのラベルを変更する場合には，既存のソートをその上にドラッグ＆ドロップすることで行う．項に現れる変数のラベルを変更する場合には，モジュール上の変数や項の上の変数を用いて，ラベルを変更したい変数の上に変更元の変数をドラッグ＆ドロップすることで行う．項の上に現れる演算を直接変更することはできないようにしている．

- **ラベルの変更における副作用：** ユーザがラベルを変更すると，関係のある全てのラベルも同様に変更される．例えばソートのラベルを変更すると，演算の引数や返却値，変数のソートも同じように変更される．演算のラベルや変数のラベルを変更すると，項の中に現れる演算も同様に変更される．このようにラベル編集において，ユーザが事前に対応付けた図形間に対して自動的に副作用を与えることで，ユーザの負荷を軽減している．

3.4 ノード指向の操作

CafePie 内では，ビジュアルシンタックスで示したようにノードとエッジのグラフ構造によってプログラムを表記している．ビジュアルプログラミングにおいてノードとエッジによってプログラムを表現する例は多いが，すべてがエッジを操作することによってノード間の関連付けを行っている． dish[40] は GUI でシェルスクリプトを記述できるシステムである．ラベル付きアイコンを用いてノードをエッジで結線しながらスクリプトを記述することができる．ノード作成時に必ずラベルを入力する必要が

あり，I／O 線のようにエッジを操作して記述する必要がある．計算機シミュレーションの分野で使われているシステム LabVIEW[18] でも，アイコンの作成にダイアログボックスを用いたりエッジを操作を行う必要がある．本システムでは，つかみにくいエッジを操作しなくても済むようにノード指向の操作方法を取っている．

ノード指向の操作とは，エッジのような細かい対象を操作しないでも良い操作のことを指す．また，操作しにくい対象はあらかじめノードを操作しやすいような大きさにしておく．目安としては，選択するマウスカーソルを基準として，マウスカーソルよりも縦／横がそれぞれ半分以上くらいにしている．半分以下であると，操作中にマウスカーソルが操作対象を覆ってしまい場合によっては見えなくなる可能性があるからである．

3.4.1 ソートにおける関連付け

順序ソートは，図 3.3に示すような有向グラフによって表現することができる．通常，グラフエディタを使って有向グラフを編集する場合には，２つ以上のノードを作成して，各ノード間にエッジを描画しなければならない．２つのノード間にエッジを引く場合には，例えば，エッジ描画モードにしてから，片方のノードを選択した後に続けてもう片方のノードを選択する，というような操作を行う．ノードが 2，３個くらいならそれほど手間がかからないかもしれないが，ノードがもっと増えたり，頻繁に追加，削除を行わなければならない場合には，非常に手間が掛かる．また，ノード間に作成したエッジを削除する場合には，削除したいエッジを選択してから削除する．エッジはノードよりも選択範囲が狭いため，より詳細な操作を要求される．このため，より詳細な操作はユーザにとって非常に煩わしく感じる．

本システム上で２つのノード間に関連を作成する場合には，表 3.1に示すように関連元から関連先にドラッグ＆ドロップするようになっている．図 3.8の左側は，Sort1, Sort2 という２つのソートがありお互いに関係付けの無い状態である．関連元にする Sort1 から関連先にする Sort2 にドラッグ＆ドロップすることで関係付けが行える．この場合，図 3.8の右側に示すように関連元 Sort1 から関連先 Sort2 へ矢印が引かれる．また，エッジの削除についても，表 3.1に示すように関連付けられている２つのノードを片方からもう片方へドラッグ＆ドロップすることで行える．図 3.8の右側に示すような関係付けが行われている場合には，関連元 Sort1 から 関連先 Sort2 へドラッグ＆ドロップすることで，２つのノード間の関係付けの削除が行われ，結果として図 3.8の左側のようになる．

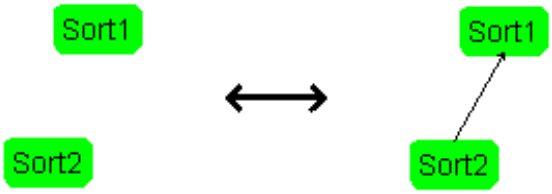


図 3.8: ソート間の関係付け

3.4.2 変数の具体化による項の編集

項の編集は，プログラムの動作を決定する等式の作成や実行時におけるゴールの作成など， CafePie 上では頻繁に行われる作業である．ここでは，ドラッグ＆ドロップ操作を用いた項の編集方法を述べる．

項の編集を簡潔に表現するために「変数の具体化」という概念を用いてモデル化した．この概念を利用し，項の追加を変数に対する具体化操作とみなす．従って，項の追加／削除は変数に対する操作として扱うことができる． CafePie では，項を木構造で表現し，演算をノード，演算の引数に対する具体化の関係をエッジで表している．

ドラッグ＆ドロップ操作を用いた項の編集は，**初期化**／**追加**／**削除**という 3 操作の繰返しで定義できる．

1. 変数を作成する（項の初期化）．

2. 変数の上に演算をドラッグして重ね合わせる（項の追加）．このとき変数は演算によって置き換わる．もし，演算に引数があるならば，その引数はそれぞれ変数に置き換わる．

3. 項にある演算のラベルをドラッグして編集空間の枠外に移動させる（項の削除）．演算のラベルを選択すると，演算のラベルとその引数にあたる部分項が選択状態になる．つまり，演算を移動させようとすると，演算に加えてその部分項が同時に移動する．したがって，この演算より下の項（部分項）が項から全て削除される．また，演算が削除された部分には変数に置き換わる．（移動させた演算をこの変数に追加することによって元に戻すことができる．）

項は演算と変数の組合わせによって表現されるが，この具体化操作は，変数に対してのみ行われる．それ以外の部分（演算）にはいっさい変更の操作をしないようにした．

ここで具体的な例を挙げる．モジュール SIMPLE-NAT （図 2.1）で記述された仕様の中の等式 “Eq1” （CafePie での表現は，図 3.2の画面中の右側に示すようになる）の左辺（ N:Nat + s(M:Nat) ）を作成してみる（図 3.9）．

1. まず， New Field を利用して 変数（V）を作成する．

2. 次に，変数（V）の上に演算（_+_）をドラッグ＆ドロップする．これにより変数（V）は演算（_+_）に上書きされる．また，演算に引数がある場合は，演算のビジュアルシンタックスにしたがって，引数をソートとする変数が追加される．変数のラベルはシステムにより自動的に付加される．ここでは V1, V2 としている．

3. 必要に応じて変数名 V1 を N に変更する．既に変数 N が定義されている場合には，その変数を用いて V1 にドラッグ＆ドロップすることでラベルをコピーする．そうでない場合にのみキーボードを利用する．特に変数名がそのままでもプログラムの実行は可能である．しかし，例えば図 3.2の等式 Eq1 のように，右辺と左辺の変数 N は同じものを表すが，ラベルは自動生成されるため別のラベルになる．この場合には，一方をもう一方にドラッグ＆ドロップし，ラベルのコピーを行うことで同じラベルにすることができる．

4. さらに変数（V2）の上に演算（s）を重ね合せる．これと同時に引数が変数（V3）に置き換わる．

5. 必要に応じて変数名 V3 というラベルを M に変更する．

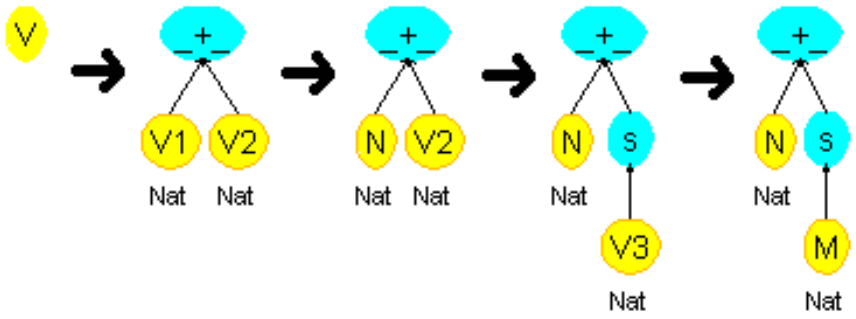


図 3.9: 項の作成手順

この「変数の具体化による項の編集」の仕組みを利用するためには，項を追加／削除に必要な「変数」があるだけで良い．このように変数の具体化によるモデル化を行うことで，項の編集を簡潔に表現することができる．

項においても変数の具体化による項の編集によってノード指向の編集が行えるようになった．この項の編集では，変数へのみ代入するという**操作対象の限定**を行うことによって，シンタックスエラーが起きないように，かつ手早く編集が行えるようになっている．

3.4.3　項の編集における応用

プログラミングの分野では，例えば，コンポーネントウェアに応用できると考えられる．コンポーネントには，これは演算と同じようにデータの入力と出力がある．入力が引数（アリティ），出力が返却値（コアリティ）に当たる．コンポーネントの種類によって描画するエッジの種類を変えるようにするなどの工夫を加えることによって，コンポーネントの種類によって関係を表すエッジを変更することもできる．しかし，同じコンポーネントを複数のコンポーネントで共有する場合には，そのままでは表現することができない．その場合には，コンポーネントのコピーを行うなどの工夫が必要となる．また， Java などで用いられるクラス継承図にも適用できると考えられる．一つのクラスの継承は，自分を木のルートとして親を自分の子することで，木構造で表現できる．しかし，クラス関係図を表現する場合には，同じクラスを複数のクラスで共有して継承する場合もあり，この場合にもコンポーネントウェアと同じように何かしらの工夫が必要である．

項の編集は，木構造を表現する場合にはそのまま適用できる．例えばファイルシステムなどディレクトリ構造の表現にはそのまま適用できる．しかし， UNIX システムなどで使われているシンボリックリンクなどを表現することは困難である． また Web のようにネットワークが入り組んだような複雑な構造を表現する場合には不向きである．

3.5　CafePie におけるプログラムの作成例

プログラムの編集を説明するために，図 2.1に示すモジュール SIMPLE-NAT の作成手順を紹介する．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，ソートを作成し，
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．

3.5.1 ソートの作成

モジュール SIMPLE-NAT の 3 つのソート（ Nat, Zero, NzNat）の CafePie 上での定義例を図 3.10に示す．

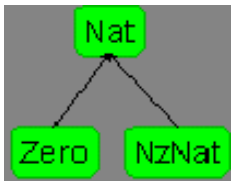


図 3.10: CafePie 上のソート定義例

ソートを作成する場合には， New Field 内のソートを選択し，ドラッグ&ドロップ操作を用いて 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ドラッグ&ドロップ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ドラッグ&ドロップ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．

3.5.2　演算の作成

モジュール SIMPLE-NAT の 3 つの演算（ 0,　s,　_+_）の CafePie 上での定義例を図 3.11に示す.

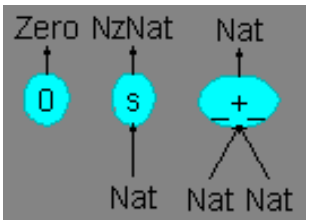


図 3.11: CafePie 上の演算定義例

演算を作成する場合は，まず New Field 内の演算を選択し，ドラッグ&ドロップ操作を用いて 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ドラッグ&ドロップ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．ここで “ op _+_ : Nat Nat -> Nat” の作成を考える．はじめに作成された演算 (これに名前を変更したもの) は，“ op _+_ : -> Nat” というように，引数がない形で与えられる．演算の定義を完成させるためには，更に引数を追加する必要がある．表 3.1を見ると，引数の追加はソートから演算へのドラッグ&ドロップ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．引数を挿入する順番であるが，既にある引数の一番最後の引数の後に追加される．これで 2 引数演算　_+_ が作成される．他の演算も同じようにして作成することができる.

3.5.3　変数の作成

モジュール SIMPLE-NAT の 2 つの変数（N,　M）の CafePie 上での定義例を図 3.12に示す.

変数を作成する場合は，まず New Field 内の変数を選択し，ドラッグ&ドロップ操作を用いて Module Field 内に作成する．変数名もデフォルトで与えられる．現在は，ソートをダブルクリックすることで変数を作成することもでき，このとき変数の作成

図 3.12: CafePie 上の変数定義例

と変数の型であるソートへの関連付けとを同時に行っている．変数は，変数名とその型であるソートから定義される．変数を作成したあとにソートとの対応付けを行う場合，ソートとの対応付けを忘れてしまった変数は，解釈することができずシンタックスエラーになる可能性がある．ソートから変数を作成することで，このようなシンタックスエラーを防ぐことができ，かつソートの入力を省くことで面倒な操作が不要になる [35]．この操作方法に従うと，“ var N : Nat” の作成はソート Nat を ダブルクリック することで行うことができ，同時にソート名 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できる．

3.5.4 等式の作成

モジュール SIMPLE-NAT の 2 つの等式（Eq1， Eq2）の CafePie 上での定義例を図 3.13に示す．

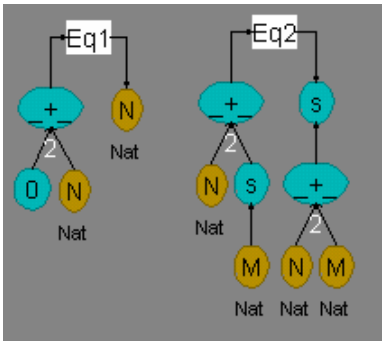


図 3.13: CafePie 上の等式定義例

等式を定義する場合は， New Field 内の等式を選択し，ドラッグ＆ドロップ操作を用いて 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 “ eq N + s(M) = s(N + M)” 上の左辺 “ N + s(M)” は，第 3.4.2節で示したような手順で作成する．

3.5.5 ドラッグ＆ドロップ操作における考察

ドラッグ＆ドロップ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．ドラッグ＆ドロップ手法の欠点として，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．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．ドラッグ＆ドロップ手法は，カット＆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＆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 [41] 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，移動途中で離して手裏剣のように投げる手法である．アイコンをつまんで移動させる．目的の方向に移動させて助走を付けて，移動途中でマウスを放す．通常，ドラッグ＆ドロップ手法ではマウスを放した所でドロップしたと認識されるが，アイコン投げ手法は，マウスを放した後も目的の方向へ移動し続け，移動途中で他のオブジェクトにぶつかると止まるようになる．この手法を応用することで，広い画面においても編集操作が可能になると考えられる．

3.6 プログラム編集における評価実験

CafePie のプログラム編集時における有効性を示すことを目的とする．

3.6.1 実験の被験者と実験環境

実験の被験者：本システムの評価のために，理工系の大学生・大学院生 11 人を対象に評価実験を行った．事前にアンケートを取り，全員マウスやキーボードを十分に使用したことがあることを確認した．また，被験者は，本システムや CafeOBJ 言語をあまり使用したことがない人を中心に選んだ．

実験環境：ハードウェアは， CPU が PentiumII 400MHz で 128MB の主記憶を搭載した DOS/V 機を用い， OS は WindowsMe を使用した．大きさが 15.4 インチで解像度が SXGA(1280x1024) の液晶ディスプレイに画面を表示し，被験者は全て同じマウスとキーボードを用いて実験を行った． CafePie は Java で記述されており，実験には JDK1.3.0 でコンパイルしたものを用いた．評価実験で用いるテキストエディタは emacs 互換の Meadow1.14 を用いた．

3.6.2　実験の手順

CafeOBJ のプログラム SIMPLE-NAT （図 2.1）を与え， CafePie を用いた場合とテキストエディタを用いた場合とでその編集に要した作業時間を測定した．

各実験を行う前に，本システムの使い方を説明してから 5 分程度使用してもらい，各被験者に基本的な作業を理解してもらった上で実験を行った．また，各プログラムの作成手順を予め指示しておき，同じような手順でプログラムを作成するという手順をとった．こうすることでプログラムの考察にかかる時間をなるべく排除し，プログラム編集にかかる時間のみを測定するように努めた．

測定は，まず， CafePie 上でドラッグ＆ドロップ（DnD）のみを用いてプログラムを記述し，最後にラベルを入力してもらった．その後，テキストエディタを用いて同等の CafeOBJ プログラムを編集するという手順をとった．

3.6.3　実験の結果と考察

プログラム編集における実験結果を図 3.14に挙げる．各グラフは，左から，「DnD のみを用いてラベル入力を行わない場合の CafePie による編集」，「DnD に加えラベル入力を加えた場合の CafePie による編集」，「テキストエディタによる編集」である．縦軸はプログラム編集に要した時間を示しており，各々，ソート，演算，変数，等式にわけて測定し，その平均値の総和をグラフにした．

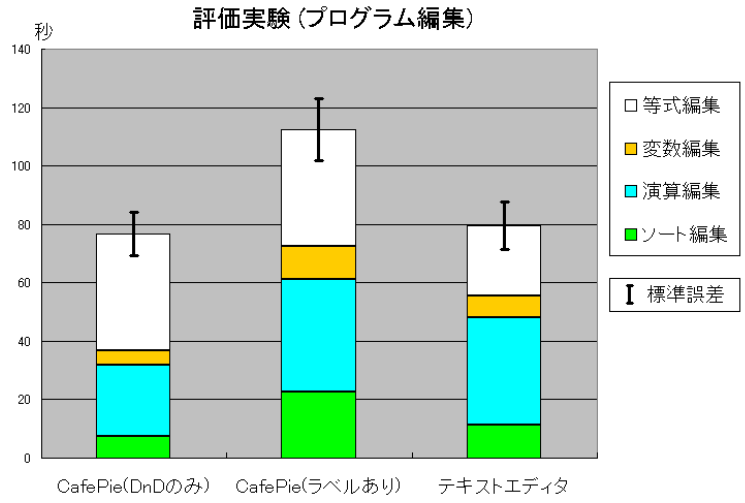


図 3.14: プログラム編集における評価

まず，プログラム編集に要した全体の時間を比較すると， CafePie （DnD のみ）の

方がテキストエディタによる場合よりも若干速くなっている．しかし，t 検定を行ってみたが，有意な差は認められなかった．次に，CafePie（ラベル入力あり）とテキストエディタによる場合とを比較すると，t 検定で 4.98（>3.17，危険率 5%，両側）となり，テキストエディタの方が速く編集できることがわかった．しかし，平均値の割合でみると約 1.41 倍と思ったよりも差は開かなかった．以上の結果により，テキストエディタを用いた場合と比較しても CafePie を用いてラベル入力をしない場合は同程度の時間で，ラベル入力をした場合でもさほど効率が悪くならないことがわかった．ビジュアルプログラミングの特徴である構造の把握を行いつつプログラム編集が行えるため，本手法は非常に有用であると言える．

また，各構成要素である，ソート／演算／変数／等式の場合を比較してみた．

ソート：　ソートの編集において，CafePie（DnD のみ）の方とテキストエディタによる場合と比較してみると，t 検定で 2.90（>2.23，危険率 5%，両側）となった．ドラッグ＆ドロップのみの場合だと，テキストエディタよりも速くなっている．しかし，CafePie（ラベル入力あり）の場合では，テキストエディタの方が速いという実験結果となった．CafePie 上でソートのラベル入力の量がテキストエディタでの入力に比べてほとんど差がないためであり，ドラッグ＆ドロップ操作にかかる時間の差がそのまま出ていると思われる．

演算：　演算の編集においては，CafePie（DnD のみ）の方とテキストエディタによる場合と比較してみると，t 検定で 2.64（>2.23，危険率 5%，両側）となり，CafePie（DnD のみ）の方が速いことがわかった．また，CafePie（ラベル入力あり）の場合では，テキストエディタと比べると若干時間が掛かるが両者にほとんど差がないという結果になった．演算の編集においては，CafePie（ラベル入力あり）の場合においてもテキストエディタと同等の性能が得られることがわかった．

変数：　変数の編集においては，CafePie（DnD のみ）の方とテキストエディタによる場合と比較してみると，t 検定で 3.10（>2.23，危険率 5%，両側）となり，CafePie（DnD のみ）の方が速くなっている．しかし，CafePie（ラベル入力あり）の場合では，テキストエディタの方が速いという結果となった．変数の定義に掛かる時間は，全体と比べると比較的短時間で行っている．変数のアイコンを定義してからラベル入力を行うために，マウスからキーボードへ切り替える作業に要する時間が影響しているものと思われる．

等式：　等式の編集においては，CafePie（DnD のみ）の方とテキストエディタによる場合と比較してみると，テキストエディタの方が速くなっている．（等式の編集

においては，テキストエディタと条件を同じようにするために，等式のラベルの入力は行っていない．）等式の場合には，両辺に項の入力に掛かる時間が大半である．図2.1のプログラムを見た場合，通常，変数にはその型となるソートを明記するのであるが，既にモジュール内に変数が宣言されている場合には，変数のソートを略記できるようになっている．しかし，CafePie においては，変数におけるソートも記述しており，その差が実験結果となっていると考えられる．また，CafePie においても，等式の編集はラベルの違いを確認しながら行うことができ，その試行錯誤のためにかかった時間が結果に影響しているものと考えられる．

実験のあとにアンケートをとってみた結果，半分以上がテキストと比べると視覚的に編集できるので分かりやすいという答えだった．図形で表記した方が特殊記号を入力する必要がなく概観が捉えやすくて良いという意見もあった．一方で図形が増えすぎるとかえって見えづらくなるという意見もあった．また，操作に関して，ドラッグ＆ドロップ操作中の図形の重なり判定が分かりにくいという意見もあった．操作に応じたガイドをユーザに提示するなどフィードバック面での改善が必要である．

第 4 章

視覚情報のカスタマイズ手法

4.1 視覚化における問題点

ビジュアルプログラミングは，図式表現を積極的に取り入れることが必要である．ボタンやスクロールバー，またはボタン内の文字列使われているフォントや色のように GUI における部品は具体的な図形を直接用いて表現した方が分かりやすい．図式表現で表示した方が好ましいものは，できるだけ図式表現で表示したい．

しかし，全ての情報を図形により過不足なく表現すれば理解しやすくなるというわけではない．逆に，ユーザに対して過大な情報を提供してしまう可能性がある．図式表現ではテキストのみの場合と比べると，この情報量の爆発が顕著に現れる．これは，同じ情報を表現した場合，テキストのみの場合と比べて図式表現は画面占有率が高くなってしまうという問題のためである．かといって，表示する量を減らせばわかり難くなる恐れもある．図式表現によってプログラムを視覚化する場合には，状況に応じて必要な視覚情報を適切に表示するような工夫が必要になる．システムが与えた図式表現だけで，視覚情報を積極的に使った視覚化を行いつつ，視覚情報を適切に把握できるような視覚化を行うには限界がある．

4.2 視覚情報のカスタマイズについて

本研究では，視覚情報を積極的に使った視覚化を行いつつ，視覚情報を適切に把握できるようにするために，システムが与えた図式表現を元にした視覚情報のカスタマイズを行うための手法を提案する．視覚情報のカスタマイズの意義は，視覚情報への積極的な対応付けを行うこと，視覚情報の変更における柔軟性の 2 つにある．前者は，テキストでの表現をより視覚的な情報に置き換えるためにカスタマイズを行うことで

ある．後者は，使用する場面に応じて表示方法を変更できるようにすると言うことであり，抽象度を高くして必要なものだけを表示することで必要な情報を取得しやすくなる，という抽象化の概念を実現する仕組みであると言える．

システムによって与えられた表現を変更するには，システム内部の視覚情報を定義している部分を書き直す必要がある．視覚情報の定義を書き直す手段として，ソースコードへ直接記述するということが考えられる．しかし，実装依存が強く視覚情報以外の部分と混在するために，視覚情報とそれ以外の部分との切り分けが難しくなる恐れがある．また，ソースコードという文字表現の部分に視覚的な情報を書き込んでいるためソースコードの可読性が悪くなる恐れがある．また，視覚情報の定義を変更するのには非常に手間が掛かることが予想される．やはり， 視覚的な情報は視覚的に記述する方が望ましい．

視覚情報の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支援する仕組みが必要である．ここでは，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の基本である MVC モデルについて取り挙げる．視覚情報のカスタマイズとは，「**MVC モデルにおける視覚情報（ビュー）を目的に応じて変更するための仕組みである**」と言える．

4.2.1 視覚情報の実装モデル

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手法が，オブジェクト指向言語の元祖である Smalltalk-80 [42, 43] 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 [44, 45] 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．

MVC モデルでは，ユーザインタフェースを以下の 3 つのオブジェクトの組を単位として構成していく（図 4.1）．

- **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．

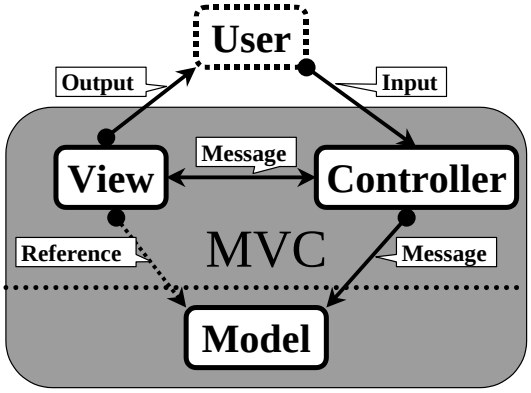


図 4.1: MVC モデル

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．

4.2.2 カスタマイズ手法の実装方針

MVC モデルを利用して，ビジュアルプログラミングシステム上での視覚情報のカスタマイズ手法を実現することを考えた． 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，ビューを必要に応じて変更可

能な仕組みは提供できるが，それとは別にユーザが新しいビューを作成する必要がある．新しいビューを作成するためには，

(1) モデルを記述する構成要素とビューで用いられる図式表現との対応関係を明示すること

(2) ビューで用いられる図式表現間の対応関係を明示すること

が必要である．(2) の「ビューで用いられる図式表現間の対応関係を明示すること」のために，Picture Layout Grammars [46] や Constraint Multiset Grammars [47] の中で用いられているような図形文法を用いて宣言的に記述することもできる．

図 4.2: 演算 push に対する視覚化

ここでは Picture Layout Grammars を用いて演算 push に図 4.2に示す積木構造のような視覚化を定義してみる．演算 push を CafeOBJ 言語を用いて次のように定義している．

```
op  push :  Nat Stack -> Stack
```

演算 push は Nat, Stack という 2 つの引数と返却値として Stack を持つ．視覚化は演算 push と各引数に対して定義する．例えば，次に示すような 4 つの制約規則を記述する必要がある．

```
(1) rectA.x + rectA.w/2 == push.ar1.x + push.ar1.w/2
(2) rectA.y + rectA.h/2  == push.ar1.y + push.ar1.h/2
(3) rectA.y + rectA.h == push.ar2.y
(4) rectA.x + rectA.w/2 == push.ar2.x + push.ar2.w/2
```

ここで rectA は長方形を示し，rectA.x, rectA.y はその右上の x 座標と y 座標を，rectA.w, rectA.h はその幅と高さを表す．また，push.ar1 は 演算 push の第一引数 Nat を，push.ar2 は 演算 push の第二引数 Stack を表す．同じように push.ar1.x, push.ar1.y で演算 push である第一引数 Nat の右上の x 座標と y 座標を，push.ar1.w, push.ar1.h 幅と高さを表す．上の 2 つの規則によって，「rectA

と push.ar1 の 中心を一致させる」という制約を記述している．また，3 番目の規則により「rectA と push.ar2 の上下を一致させる」という制約を，最後の規則により「rectA と push.ar2 の x 軸方向の中心を一致させる」という制約を記述している．

単にテキストを記述するだけであれば，比較的容易に記述できるかもしれない．しかし，各図形が何を表しているのか具体的なイメージが把握しにくい．イメージが把握し辛いのは，視覚情報という 2 次元的な情報をテキスト表現という 1 次元的な情報として記述しているためである．具体的なイメージの把握が困難であるから，予想される結果が分かりにくく，編集にも非常に手間が掛かる．

また，この 4 つの宣言的な記述で行っていることは，図形間に決めた制約の関係を明示しているに過ぎなく「モデルを記述する構成要素とビューで用いられる図式表現との対応関係を明示すること」ことは行っていない．例えば rectA.ar1 が Nat であることは 4 つのテキスト記述からは判断することができない．モデルを記述する構成要素との対応関係を明示するために，他で予め定義しておく必要がある．

ビジュアルプログラミングシステム上でカスタマイズ機能を実現するに当たって，以下の 2 点が重要である考えた．

- 新しくビューを定義するには，視覚情報のカスタマイズに必要なアイコンを用意するだけでは不十分である．モデルを記述する構成要素との対応付けを行いつつ，テキスト表現ではなく視覚的に明示しながら各アイコン間の対応関係を変更できるようにする．

- 視覚情報のカスタマイズの作業は本来のプログラミング作業よりも手間が掛からない方が良い．厳密に記述できなくても良いから，プログラミング作業と比べて手早く簡単にカスタマイズできることが望ましい．

4.3 データ構造における視覚情報のカスタマイズ

4.3.1 データ構造のカスタマイズ

プログラムの中でも特にデータ構造を表す項は，プログラムの動作を与える等式やプログラム実行の表示に頻繁に使用される．視覚情報のカスタマイズの対象は，プログラム編集や実行において最も重要となる項（データ構造）である．

例えばデータ構造における視覚情報のカスタマイズが行えるシステムとして GELO [48] がある． GELO は，データやソースコードなどに図的なビューを提供することが

できるツールであり，他のソフトウェアツールのライブラリとして使用される．しかし，カスタマイズの方法はダイアログボックスを用いて図形の種類やパラメタを決めるという手法をとっており，視覚情報のカスタマイズを手早く行うことができない．

ビジュアルプログラミングにおいて，視覚情報のカスタマイズに費やす時間をなるべく短くして，プログラミングの作業をじっくり行いたい．そこで，視覚情報のカスタマイズを手早く行うために既に表示されている情報を用いることで，画面を見ながら視覚的に行うことを考えた．

4.3.2 デフォルト・ビュー

ここでは， CafeOBJ 言語のモデルを過不足なく記述できることから，テキスト表現で書かれた内容をプログラムの**モデル**と呼ぶことにする．そして，プログラムモデルに対応した視覚情報を表示する部分を**ビュー**と呼ぶことにする．例えば，スタックについて表記してみると，項のモデルは

```
push(E3,push(E2,push(E1,empty)))
```

のように表現できる．システム CafePie において，このモデルに対応するビューは，図 4.3のような木構造で表現している． また，ここでシステムで定義されている，モ

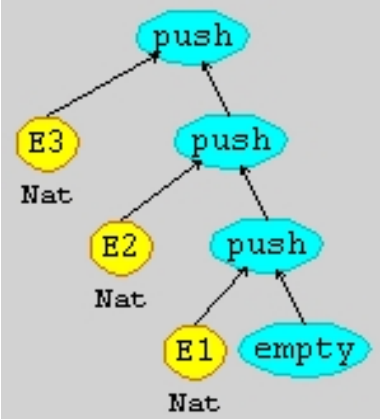


図 4.3: システムによるスタックの視覚化

デルを視覚化するためのビューのことをデフォルト・ビューと呼ぶことにする．CafePie においては，「木構造というデフォルト・ビューを用いることにより項を視覚化している」と言える．

4.3.3 ビューの対応画面

変更する部分の対応関係を視覚的に確認しながら新しいビューを編集することができるようにするために，ビューの対応画面を使用することを考えた．

ビューの対応画面を 図 4.4 に示す． 図 4.4の左側に示すビュー（OLD VIEW）は，

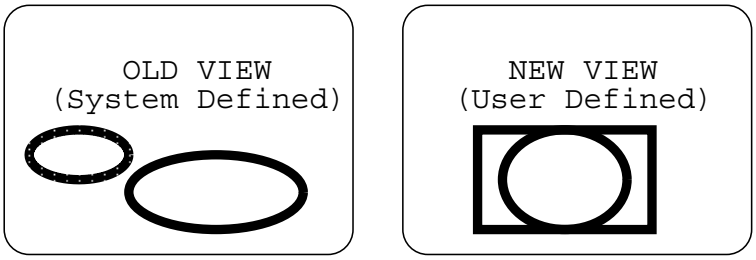


図 4.4: ビューの対応画面

カスタマイズ前の視覚情報である． 図4.4の右側に示すビュー（NEW VIEW）は，ユーザが新しく定義するカスタマイズ後の視覚情報である． CafePie 上での視覚情報のカスタマイズにおいては，デフォルト・ビューが OLD VIEW に当たる．

4.3.4 視覚情報のカスタマイズの作成例

項のデフォルト・ビューに対して視覚化を考える場合には，項の構成要素である各演算に対しての視覚化を考える．項を構成する各演算に対して視覚情報のカスタマイズを定義することで項全体の視覚化を作成するというようなボトムアップ的に構築していく方法を取っている．ここでは，例として図 4.3で示されるスタックにおける視覚情報のカスタマイズを例として取り挙げる．図 4.3に示すデフォルト・ビューの木構造による視覚化の代わりに，ここでは図 4.5に示すような積木構造による視覚化を考えてみる．

スタックの要素となる演算には主に empty と push がある．演算 empty はスタックが空であることを示す．システムが与えたデフォルト・ビューでは，図 4.6の左に示すようにラベル empty 付きの楕円の上に演算の返却値（演算の型を表すソート）のラベル Stack を配置し，楕円 empty からラベル Stack へ有向線を引く，というビューになる．ラベル付楕円が演算 empty のビューを表している．ここでは，演算 empty の楕円部分を，図 4.6の右に示すような矩形で表すことにする．

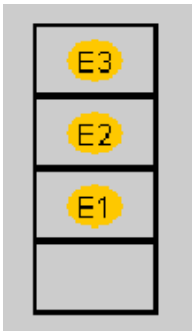


図 4.5: スタックに対する新しい視覚化

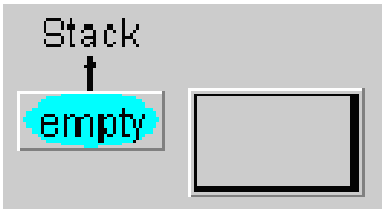


図 4.6: 演算 empty へのビューの定義

また，演算 push はスタックに 1 つの要素を積んだという状態を表す．演算 push は 2 引数の演算であり，デフォルト・ビューは，図 4.7の左に示すように，ラベル push 付きの楕円の下に演算の引数ソートである Nat と Stack のラベルを配置し，楕円の上に演算の返却値のラベル Stack を配置する．さらに，各引数を示すラベルから楕円へ，楕円から返却値のラベルへと有向線を引く．演算 push のように引数付き演算の場合には，演算を示す楕円と各引数を含めたビュー考える．ここでは，演算 push の楕円部分と各引数の代わりに，図 4.7の右に示すように演算 push の中に 引数 Nat を配置し，その下に引数 Stack を配置するというビューで表す．図 4.7は，演算を「スタックがあるときにその上に要素を積む」という視覚的な表現を表している．

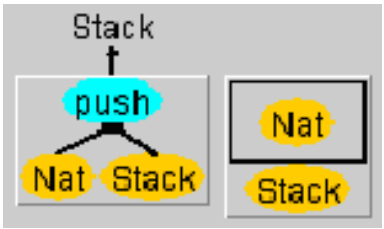


図 4.7: 演算 push へのビューの定義

4.3.5　カスタマイズ後における項の編集

第 3.4.2節で説明したように，変数の具体化による項の編集の仕組みを利用するためには，項の追加／削除に必要となる変数があれば良い．これはカスタマイズを行った後においても変数がそのまま利用できるため，項の編集は以前と同じような枠組みで行うことができる．

図 4.8に，カスタマイズ後のビューをそのまま利用した場合における，項の追加を行う場合の例を示す．まず，図 4.8の左は，項 push(E3, push(E2, push(E1, Stack))) の変数 Stack の部分に，矩形（演算 empty）を追加することを示している．細い矢印はドラッグ＆ドロップ操作を意味している．図 4.8の右は，追加の操作を行った後の結果を示している．変数 Stack の部分に矩形が上書きされ，結果として項 push(E3, push(E2, push(E1, empty)) を得る．

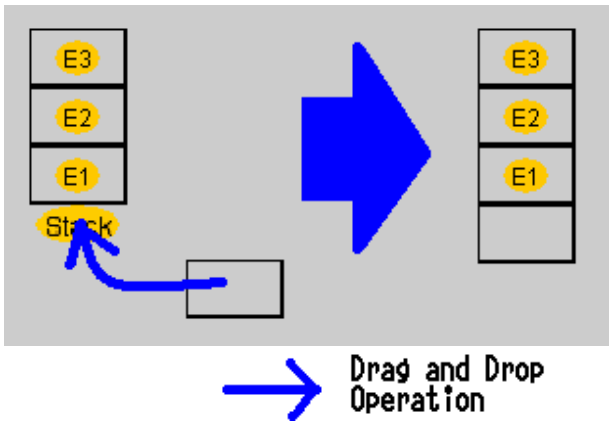


図 4.8: カスタマイズ後のビューを用いた項の編集（追加）

同様に，図 4.9に，カスタマイズ後のビューをそのまま利用した場合における，項の削除を行う場合の例を示す．図 4.9の左は，図 4.8で作成した項 push(E3, push(E2, push(E1, empty)) がある．この項の上から 3 番目の矩形（演算 push）を選択する．このとき，演算の引数である変数 E1 と 演算 empty も同時に選択状態になる．矩形をドラッグ＆ドロップし任意の場所に移動させると，項 push(E3, push(E2, push(E1, empty)) から項 push(E1, empty) が削除される．削除された部分は，変数 Stack で置き換わる．この結果として，図 4.9の右側のように項 push(E3, push(E2, Stack)) と項 push(E1, empty) とに分かれる．

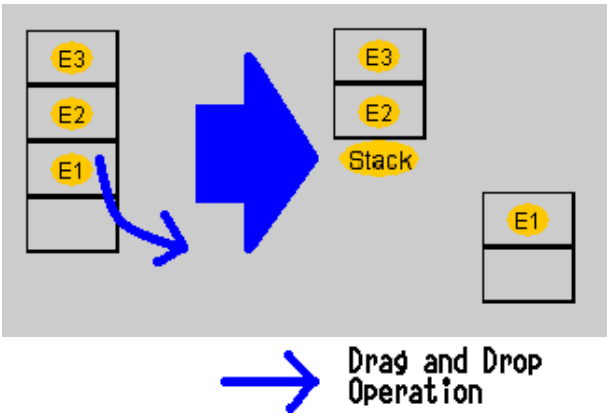


図 4.9: カスタマイズ後のビューを用いた項の編集（削除）

4.3.6 視覚情報のカスタマイズ機能の適用

視覚化は一通りである必要はない．カスタマイズ機構を用いる利点は，プログラム表現を目的に応じて自由に変更できるという点にある．

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．これらの違いはデータを取り出す順番にあり，視覚的な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 4.10 のように，行止りを示す表現に置換える．

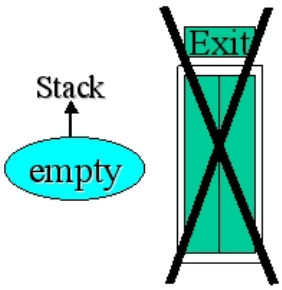


図 4.10: 演算 empty へのビューの定義（人の並び）

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．以前に定義したスタックの視覚化（図 4.6や図 4.7）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

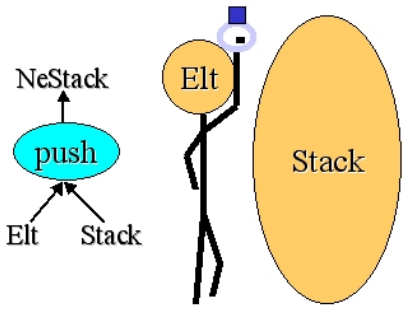


図 4.11: 演算 push へのビューの定義（人の並び）

には，図 4.11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

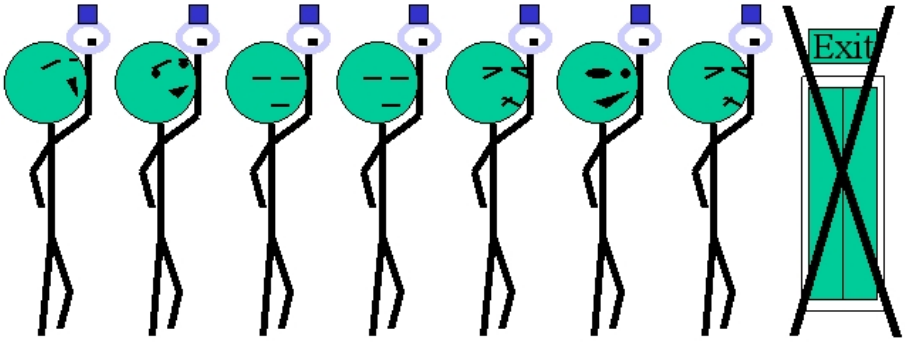


図 4.12: カスタマイズ後のスタックの視覚化（人の並び）

図 4.13: スタックの要素／顔の表情

おけるスタックの要素に人の顔を用いることで 図 4.12 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 4.13）．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，

CaefOBJ 言語では同じテキスト表現であっても異なる視覚化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を図的に考察する」といった学習などにも役立つ．

4.4 ドラッグ＆ドロップ手法を用いたカスタマイズ

ビューの対応画面は，対応関係を明示しているに過ぎない．視覚情報のカスタマイズを手早く簡単に行えるようにするためには，操作の負荷を軽減するための操作手法を考える必要がある．このために，直接操作を用いてインタラクティブにカスタマイズを行えるようにすることを考えた．インタラクティブな編集に当たっては以前のシステムにおける操作手法との互換性を保つために，**ドラッグ＆ドロップ手法を用いたカスタマイズ**を実現することを考えた [49, 50, 51, 52, 38] ．

4.4.1 カスタマイズで行うこと

視覚情報のカスタマイズで行うことは，

- 図式表現を再定義すること，
- 各図式表現の間の位置関係を再定義すること，

の 2 つである．前者の図式表現の再定義は，デフォルト・ビューで用いられた図式表現の代わりに，複数の図形オブジェクトを組み合わせて新しいビューを作成することである．また，カスタマイズで扱う図形オブジェクトの種類は，変数以外に，矩形，丸矩形，楕円とユーザ定義によるものがある．また，各図形は色／大きさ／位置などの属性を持つ．後者の各図式表現の間の位置関係の再定義は，各図式表現間の配置方法を決めることである．配置方法を決める場合には，各図式表現における関係（制約）を元にする．図式表現間の関係の種類には，

- 離れている位置に各図式表現を配置する（図 4.14の左側），
- 各図式表現の上下左右のいずれかを付着させる（図 4.14の中側），
- 各図式表現の縦または横の中心を一致させる（図 4.14の右側），

などが考えられる．

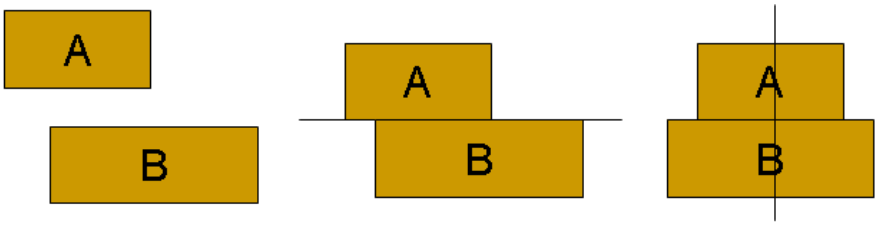


図 4.14: 図式表現の関係の種類

4.4.2 ドラッグ＆ドロップ手法を用いる場合の問題点

ユーザは基本的に，**図形の移動** と **拡大／縮小**を使って編集を行う．図形は，マウスカーソルで選択し，それをドラッグすることで自由に移動することができる．また，図形を拡大／縮小する場合は，図形の端をマウスカーソルで摘みドラッグ＆ドロップ操作することで行う．

ドラッグ＆ドロップ手法を用いて視覚情報のカスタマイズを行う場合，ドラッグ中に操作の結果が分かりにくいという問題がある．これに対して，図形オブジェクトをドラッグし，図形をきれいに揃えて配置するためのスナッピングという手法が良く用いられる．スナッピングは，操作／移動中の図形がグリッドや他の図形に揃うように配置する場合に，操作中の，もしくは操作した図形を強制的に移動させる手法である．しかし，レイアウトを行う場合には，近い位置でも異なるレイアウトが考えられる．この場合，強制的に移動させることは，操作対象を少し移動させただけで注目すべき操作対象が頻繁に移動することに成り兼ねない．この場合，ユーザに不快感を与えてしまう可能性がある．したがって，視覚情報のカスタマイズを行う場合には，移動途中では強制的に移動させないようにする方が望ましい．移動途中で図形を強制的に移動させない場合には，スナッピングの適用範囲が分かりにくい．同じように，図形間の配置を作成する場合には，ドラッグ操作中において操作結果の情報欠如という問題が起こりえる．ドラッグ操作中にでも，現在行っている操作の結果がどのようになるか，状況の把握できるような仕組みが必要である．

そこで，ユーザが図形を移動している途中でも，その図形の位置に応じて操作結果がどのようになるか分かるように，フィードバックを提示して操作結果をユーザに示唆することを考えた． 図 4.15に，演算 push のビューを作成する場合（図 4.7）のドラッグ＆ドロップ操作に応じたフィードバック表示例を示す．ユーザのドラッグ操作に応じて，システムは現在の図形間の関係を示したフィードバックを図形の上に重ね

図 4.15: ドラッグ＆ドロップ操作に応じたフィードバック表示

て提示する．システムはフィードバックを提示するために，重ね合わせた 2 つの図形の情報，図形の種類や図形の属性（位置や大きさなどの視覚的な情報）を元に計算している．ユーザは，その提示に応じて現在の状況を認識することができ，スムーズに図形間の関係を定義できる．

4.4.3 カスタマイズの適用範囲

- 重ならない図形間の関係付け

  通常，ドラッグ＆ドロップ手法は 2 つの図形を重ねることで 2 つの図形間の関係を定義する．したがって，カスタマイズを行う場合には図形が重なっていない場合には通常，図形間に関係付けを行うことができない．例えば，図 4.16の右側に示すように，図 4.7の右側と比べて Stack を長方形の下方に少しスペースを開けて配置したい場合を考えてみる．単にドラッグして移動するだけで，見た目上はうまく配置を行うことはできる．しかし，どの図形に対してどれだけ離して配置したいのかを明示することは困難である．

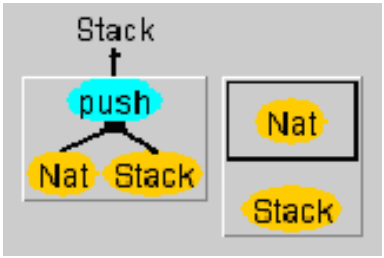


図 4.16: スペースを開けた配置

重ならない図形間に関係付けを行えるようにするために，一度でも重なった場合には関係をシステムが記憶するようにした．例えば，図 4.16の引数 Stack を

矩形の下に放して配置したい場合には，一旦，引数 Stack を矩形に重ねてシステムに 2 つの図形を関係付けるように明示してから，引数 Stack を下に移動させて目的に位置に配置することができる．

- 2 次元的な配置について

  ビューのカスタマイズは，スタックのようにデータを一列に並べるという 1 次元的な配置には向いているが， 2 次元以上の配置を行う場合には困難になることが予想される． CafePie では項を木構造によって表現しているが，これは 2 次元的な配置であると言える．項の構成要素である演算についてビューを定義するが， CafePie では次のような方法で 2 次元的な配置の問題を解決している．

  – 演算における各引数を一まとめにグループ化をして 1 次元的に配置する．

  – グループ化した引数の上に演算を配置する．

  しかし，グラフ構造のように，グループ化ができない場合にはこの手法は適用できない．グラフ構造を表現する場合には，画面上を自由に動くことができる特殊なエッジを用いるなどの工夫が必要になると思われる．

4.4.4 ビューのカスタマイズに関する関連研究

Chimera[53] は図形エディタの編集履歴を操作してマクロを生成するツールである．本手法との類似点として，マクロを表示する場合に操作変化を具体的な図式表現を使って提示していることが挙げられる．しかし，マクロの表示は静的な表示であり対話的に表示してはいない．また， IMAGE[54] は制約指向の宣言的図形配置システム Trip2a[55] の後継したシステムである．本手法との類似点として，配置制約を適用した結果の例を対話的に提示していることが挙げられる．相違点は， IMAGE が配置制約から推論した結果を図的に表示するのではなくテキストとして表示している点が挙げられる．また，本手法はインタラクティブ性を重視しており，操作結果から提示するのではなく，ドラッグ操作中という操作の完了前において結果を提示している．

4.5 視覚情報のカスタマイズ手法における評価実験

ビューのカスタマイズ時における有効性を示すことを目的とする．

4.5.1 実験の被験者と実験環境

実験の被験者：我々は，視覚情報のカスタマイズの評価のために，理工系の大学生・大学院生 11 人を対象に評価実験を行った．事前にアンケートを取り，全員マウスやキーボードを十分に使用したことがあることを確認した．また，被験者は，本システムや CafeOBJ 言語をあまり使用したことがない人を中心に選んだ．

実験環境：ハードウェアは， CPU が PentiumII 400MHz で 128MB の主記憶を搭載した DOS/V 機を用い， OS は WindowsMe を使用した．大きさが 15.4 インチで解像度が SXGA(1280x1024) の液晶ディスプレイに画面を表示し，被験者はすべて同じマウスとキーボードを用いて実験を行った． CafePie （視覚情報のカスタマイズ実装版）は Java で記述されており，実験には JDK1.3.0 でコンパイルしたものを用いた．

4.5.2 実験の手順

CafePie におけるスタックの演算 push における積木構造（図 4.7）を表すビューをカスタマイズし，その編集に要した時間を測定した．各実験を行う前に，作成手順を示し，その後で約 1 分間 CafePie を使用してから実験を行った．ビューのカスタマイズについて同じ操作を 3 回行い，その時間を測定した．

4.5.3 実験の結果と考察

ビューカスタマイズにおける実験結果を図 4.17に載せる． 各被験者がビューのカスタマイズに要した時間をグラフに表示した．横軸がビューのカスタマイズに要した作業時間である．各グラフは，測定した作業時間の平均値を表している．作業時間は平均で 9.42 秒（標準偏差 1.78）となった．これは，図 4.2のための 4 つの制約規則をテキストで記述する場合に比べて早く記述できるのは明らかである．また，第 3.6節で述べたプログラム編集における評価実験で行った，簡単なプログラム編集にかかる時間と比較してもほんの 1 割程度にすぎない． CafePie を使い始めてまもないにも関わらず，ビューのカスタマイズに要する時間は非常に少ないことがわかる．更にアンケートでも，操作中に現在の状態がガイドで表示されるため非常にわかりやすいという意見も得られた．このことから視覚情報のカスタマイズはそれほど時間を掛けずに手早く行うことができると言える．

11 人中 6 人は失敗せずに一度で編集できたが，その一方で，残りの人達は一度では編集できず修正をする必要があった．その人たちのアンケートには，操作する対象が

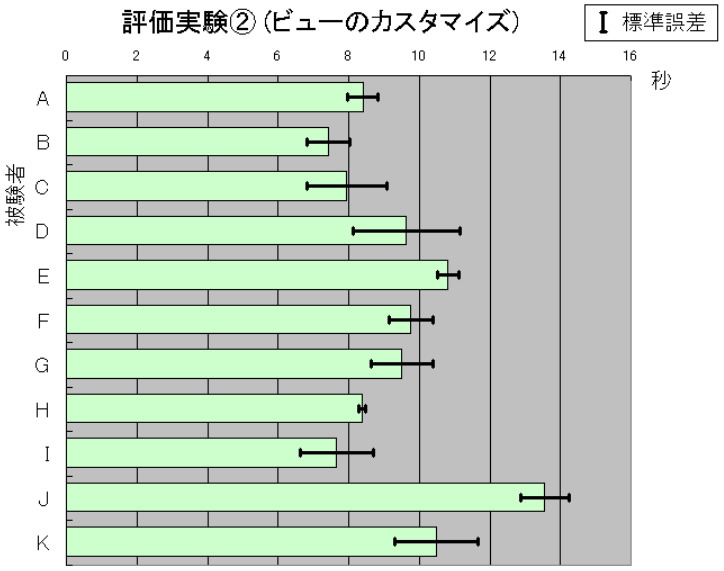


図 4.17: プログラム編集における評価

小さすぎて微調整が難しいなどの意見があった．対象物を大きくするなどの工夫が必要である．

第 5 章

視覚情報のカスタマイズとプログラム実行

5.1 CafePie におけるプログラムの実行処理

ここでいうプログラムの実行とは項書換えを意味する．項書換え系（Term Rewriting System, TRS）は，項の書換えのみで計算が進む計算モデルであり，その形式の単純さと直観に優れた代数的意味論が特徴である．TRS は書換え規則（等式）の集合であり，書換え対象である項は計算過程の 1 つの状態を表す．

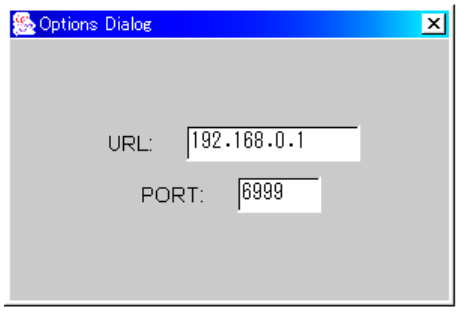
既存システムである CafeOBJ の処理系（CafeOBJ インタプリタ）は，cafemaster と呼ばれるネットワークサーバを介して利用することができる．cafemaster は，TCP/TP ソケットを用いて実装されたシステムであり [56]，接続要求を受け取ると受け取ったポート番号に応じて予め決められたサービスを実行することができる．cafemaster のことを，ここでは CafeOBJ サーバと呼ぶことにする．CafePie は TCP/IP ソケット通信を用いて CafeOBJ サーバにアクセスし，CafeOBJ サーバを介して CafeOBJ インタプリタを利用することで，項書換えの処理を実現している．

プログラム実行においてユーザが行う作業は，書換え対象となる項（ゴール）を作成し，ゴールを CafeOBJ インタプリタに渡すことである．

ゴールの作成： ゴールの作成は，プログラム編集における項の作成と同じようにして行う．ユーザは，Module Field 上に変数を作成し，ドラッグ＆ドロップ操作を用いて変数の上に演算を重ね合せることによって作成していく．

CafeOBJ の処理系への受渡し： 処理系に必要な情報は，書換え対象のゴールと，書換えの処理に必要なプログラムである．ユーザは，ドラッグ＆ドロップ操作を用いて作成したゴールを Module Field の左上にあるモジュールラベルの上に重ね合せることで，処理系への受渡しを行う．

プログラムの実行は，CafeOBJ インタプリタが行う．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．図 5.1のようなダイアログボックスを用いて，CafeOBJ インタプリタがネットワーク上の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．



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 5.1: インタプリタの URL 指定用ダイアログ

CafePie が行う処理は，(1) CafeOBJ インタプリタの接続要求，(2) データのテキスト化処理，(3) CafeOBJ サーバへの実行要求，(4) 実行結果の視覚化処理である．

CafeOBJ インタプリタの接続要求： CafePie は，まず，CafeOBJ サーバを介して CafeOBJ インタプリタに接続を試みる．インタプリタとの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)．

データのテキスト化処理： 接続された後で，CafePie は，実行を開始するために，現在，編集しているモジュールと実行を行うゴールをテキスト形式に変換する．

CafeOBJ サーバへの実行要求： CafePie は，まず，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)．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 CafeOBJ インタプリタに送信する)．

実行結果の視覚化処理と表示： ゴールが正しく解釈されれば，CafeOBJ インタプリタは項を書き換えた結果を返す．この結果は，図 2.3に示すように書き換え前と書き換え後などの情報からなる簡約の組から構成される．CafePie でこの実行結果をを解釈して視覚化する．

5.2　プログラムの実行例

ここでは，モジュール SIMPLE-NAT （図 2.1）上で項 “ s(s(0)) + s(s(s(0)))” をゴールとして与えた場合の実行結果について示す．

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 図 5.2 に示すように，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（紙芝居表示）．もう 1 つは 図 5.3 に示すように，書換え過程で出現した項を横に帯状に並べて静的に表示する方法である（絵巻物表示）．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換わっていく過程を大まかに把握できる．これに対し，後者は，書換えの行程の前後関係を詳細に観察する場合に適している．

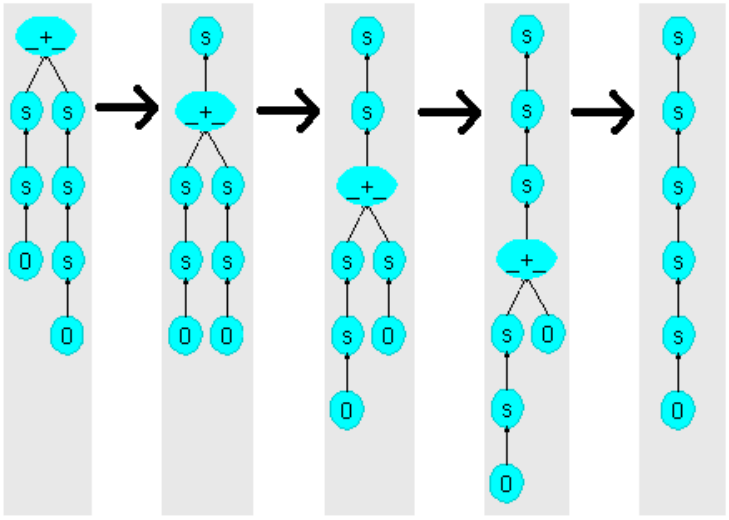


図 5.2: 動的な実行の視覚化

CafePie は CafeOBJ インタプリタからの出力を受け取ると，その都度，その簡約手順に従って視覚化処理を施し，ゴールをその場で上書きして表示する．これによりユーザには，あたかもゴールが動的に変化したかのように見える．つまり，ユーザの視点で見ると，書き換え過程を紙芝居表示を使って動的に表示しているように見える．SIMPLE-NAT のゴールである “ s(s(0)) + s(s(s(0)))” は，図 5.2 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))))) ” まで書き換わってそれ以上書き換えられなくなると，書き換えが終了してその書き換え過程の全体を表示するために図 5.3に示すように静的に表示される．書き換え過程の情報は CafePie 上に保存さ

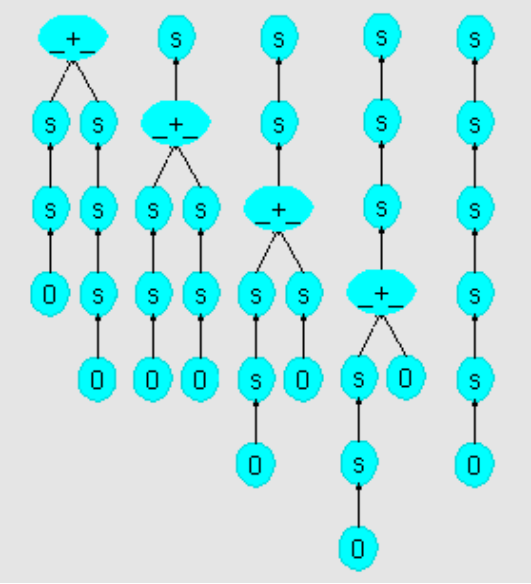


図 5.3: 静的な実行の視覚化

れるため，一度実行した書き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．また，書換えが終了したゴールは，編集し直すことができる．この切替えは，処理系への受渡しと同じように，項をモジュールラベルの上に重ね合せることで行う．

5.3 視覚情報のカスタマイズを用いたプログラム実行

プログラムを実行するに当たって，プログラム実行に必要な情報を視覚的に抽出して効果的にプログラムを表示したいが，従来のビジュアルプログラミングシステムでは，プログラム実行の表示形式があらかじめ決まっており，データの視覚的な特徴を利用した表示が困難であった． Zeus[19] は，同じデータをいろいろな側面から可視化できるアルゴリズムアニメーションシステムであり，表示されたデータを操作することによりマルチビューエディタとして使用することができる．しかし，視覚化はプログラム中に埋め込むという手法を採っている．また，アルゴリズムアニメーションシステム Pavane[20] や汎用的なアニメーションシステム Trip2a[55] では宣言的記述で記述できるが，テキストベースで記述している．

そこで，視覚情報のカスタマイズを利用することで，プログラムの実行時において

も表示形式を変更できるようにすることを考えた [57, 52, 37, 38, 58] .

スタックにおける実行を考えてみる．例えば，スタックにおけるゴールのモデルを，

```
pop(push(E1,push(E3,push(E2,pop(pop(
  push(E3,push(E4,push(E1,empty)))))))))
```

のように与える．このモデルを木構造で表現した場合，スタックにおける基本的な構造部分と動作とが入り交じっているために視認性が損なわれる恐れがある．プログラムの実行において，ゴールをその場で書き換えることでゴールの変化を動的に提示することでその動作を視覚的に捉えることができるが，スタックのように，プログラムの種類によっては動作を理解するのが困難な場合もある． CafePie では，ゴールで用いられているビューを変更することによって，プログラムの視認性を向上することができる．

5.3.1 プログラム実行における視覚情報の利用

等式は，実行時のアルゴリズムを定義しており， CafePie においても，プログラム実行の動作において視覚化を行う際に利用している．

プログラムの実行において視覚情報を利用する場合には，等式におけるビューを変更する．図 5.4 の左側には，スタックにおける演算　pop における等式を定義している． pop は 1 引数の演算であり，この図が示す等式では，スタックの要素から第一番目の要素を取り除いた残りのスタックを返すという動作を定義している．等式の左辺に現れる演算　push を選択して既に定義されているビュー（図 4.7の右側）を呼び出し，ビューの切り替えを行う．演算　push のビューの変更を行うと，等式の視覚情報は図 5.4の右側のように変更される．

演算　pop の等式におけるビューを変更したあとにプログラムを実行すると 図 5.5のようになる．従来の実行と同じように，まず，図 5.5の左端にある図形をゴールとして入力する．ゴールは，ビューを図 4.7のようにカスタマイズした演算　push と演算 pop を用いて編集する．次に，ゴールの一番上の演算　　pop を選択し移動して，モジュールラベルの上に重ね合わせ，実行を開始する． CafeOBJ インタプリタで処理が行われたあとに図 5.5の右側から左側の方へ順に書き換えが行われ，最後に図 5.5の右端に示すように

```
push(E3,push(E2,push(E1,empty)))
```

というゴールを得る．

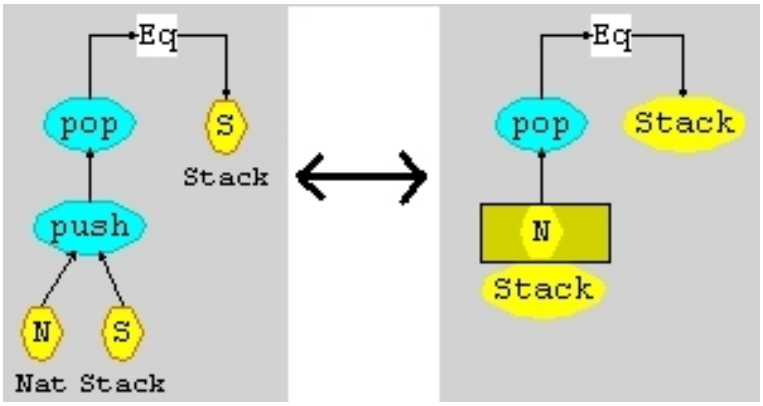


図 5.4: ビューの切替えと等式の表現

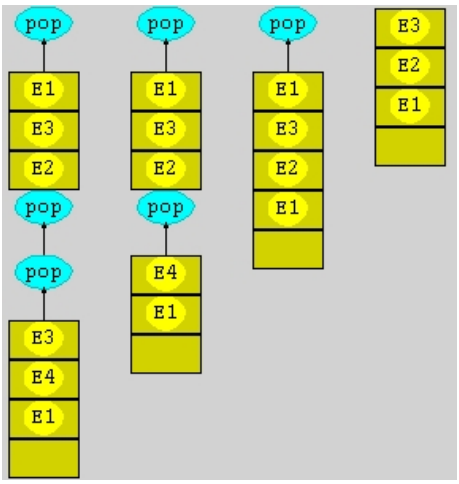


図 5.5: カスタマイズ表現を用いた実行表示

5.3.2 プログラム実行のアニメーション表示

実行を動的に表示した場合，図 5.5の左端のユーザが与えたゴールはその場で書き換えられる．そして，スナップショットを一定の間隔で次々に表示させていき，アニメーションのように実行される（紙芝居表示）．最後に図 5.5の右端のようになるまで書き換えられると紙芝居表示が終了し，絵巻物表示に切り替わる．

しかし，スナップショット形式で表示しているため，同じような演算が重なっていたりすると，どの部分に対して書き換えられたのか判断にしくい．例えば，図 5.5左端には pop を重ねて表示している箇所がある．これから図 5.5左中に書き換わる場合，どちらの pop を書き換えたのか判別しにくい．図形の書き換えをスムーズに見せる仕組みが必要である．

そこで，図形の書き換えルールと見なせる等式の間に，**スナップショット間の状態を補間できるようにする**ことでアニメーションを定義できるようにした（図 5.6） [37, 38, 58] .

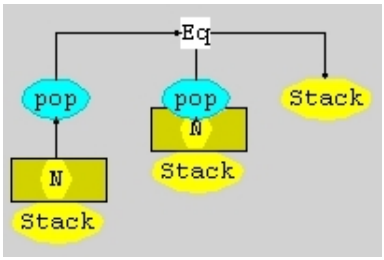


図 5.6: アニメーションの補間設定

図 5.6の左側と右側の間に，スナップショットをスムーズに見せるために補間した図（図 5.6の中心）が挿入してある．これは，左側を評価して右側に書き換える途中に，全体を 1 とすると 0.5 のタイミングで表示することを意味している．これにより，プログラムの実行は図 5.7のようにスナップショット間が補完されてよりスムーズなアニメーションとして表示される．

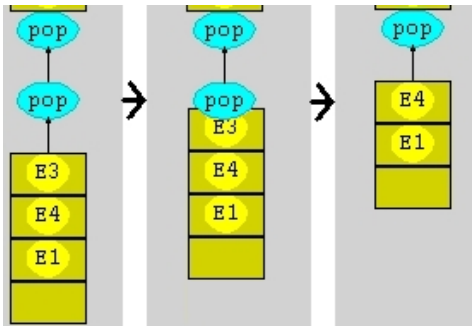


図 5.7: 補間されたアニメーションの表示

図 5.6の横軸の（ラベル Eq の左右に伸びる）線は，補間する図を表示するタイミングを表す．中心の図をドラッグすることで左側に近づければより早いタイミングで，右側に近づければより遅いタイミングで，中心の図を表示することになる．等式以外の所にドロップすることで補間する図形を削除することができる．また，ユーザが等式の左（または右の図形）を中心の方にドラッグ＆ドロップすることでそのコピーが作成される．ユーザはこの図形の一部を移動させたりして間の図形を編集する．スナッ

プショット間に補間する数を，２個，３個というように増やすことによって，よりスムーズなアニメーションを作成することができる．

5.4 他の例題

他の例題として，ここではソーティングプログラムを挙げる．

5.4.1 新しい等式の作成

スタックの演算 push の意味を変更することでソーティングのアルゴリズムを定義している．意味を変更とは，動作を定義してる等式を追加したり変更したりすることである．図 5.8 に示すような，演算 push に関する新しい等式を追加した． 図 5.8の

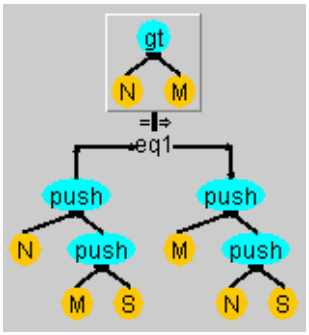


図 5.8: ソーティング用の等式

等式を CafeOBJ 言語で記述すると，

```
eq [eq1] : push(N, push(M,empty))
             = push(M, push(N, empty)) if (N gt M) .
```

のようになる．ここで， eq1 は等式のラベルであり，システムが自動的に与えたものである． N， M はともにソート Nat の変数である．また，キーワード “if” から “.” までが等式の条件式を示している．したがって，この等式は条件式付きの等式であることがわかる．条件式で使われている演算 gt は，第 1 引数が第 2 引数とり大きい場合には真となり true を返す，という意味を表している（このための等式が別に定義されている）．したがって，この等式の意味は，スタックの任意の隣り合う 2 つの要素（ N， M）を取ってきて，もし左側の要素 N が右側の要素 M よりも大きい場合には，その 2 つの要素を入れ替えるという意味になる．

5.4.2　ゴールの作成とビューの変更

- **ゴールの作成**

  まず，初期状態 (ゴール) として，

```
push( s(s(s(s(s(0))))),
push( s(0),
push( s(s(s(s(0)))),
push( 0,
push( s(s(s(0))),
push( s(s(0)),          empty))))))).
```

  を与える．これは，ビュー変更前の演算　push を使って編集すると，通常，図5.9のような木構造で与えられる．　このゴールの意味は，　(5, 1, 4, 0, 3, 2) を昇

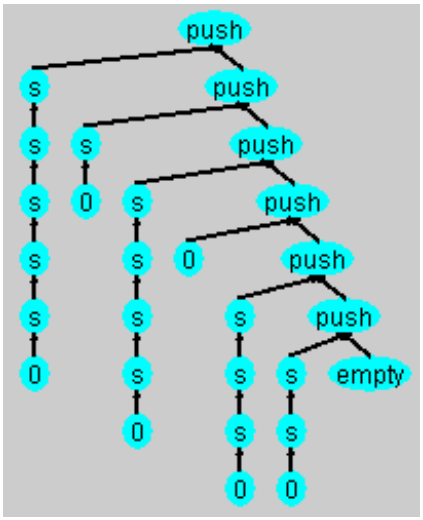


図 5.9: ソーティングにおけるゴール

  順にソートすることを示している．

- **演算におけるビューの作成**

  ソーティングの実行の際に，データ同士を横一列に並べたような表示を行うようにするために，演算　push と　empty に対して次に示すような新たなビューを作成する．演算　push は，図 5.10に示すようになる．この図 5.10は，何かスタックがあったらその左側に要素を並べるという意味を示している．また，演

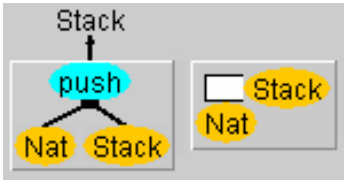


図 5.10: ソーティングでの演算 push

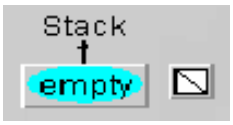


図 5.11: ソーティングでの演算 empty

算　empty は，図 5.11に示すような正方形に斜め線が入った図として表示することで，データの最後という状態を示している．

- **等式へのビューの反映**

  次に，演算　push に定義した等式に対して，演算　push のビューを反映させる．この反映のやり方は，図 5.8に示す等式　eq1 の左辺の一番上の演算　push を選択して演算　push に定義したビューを呼び出すことで行う．また，等式の右辺についても同様の操作を行う．結果として，等式　eq1 は 図 5.12 のようになる（図 5.12では条件式の表示を省略している）．

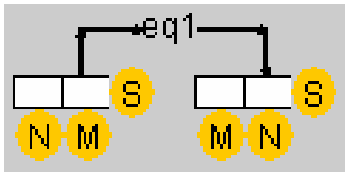


図 5.12: 等式のカスタマイズ結果

- **ゴールへのビューの反映と実行表示**

  演算　push と　empty に対して新しいビューを作成した後に，図 5.9の一番上にある演算　push を選択して演算　push に定義したビューを呼び出すことで，ゴールは図 5.13の左側に示すような視覚化を行うことができる．

  ゴールへのビューの反映を行った後で実行を開始すると，ゴール（図 5.13の左側）は，書き換えられていく．書き換わる過程が随時画面上のスナップショットとして表示され，実行のアニメーション表示が行われる．最後に，図 5.13の右側のような結果を得る．　この結果をテキスト表記すると，

```
push( 0,
push( s(0),
```

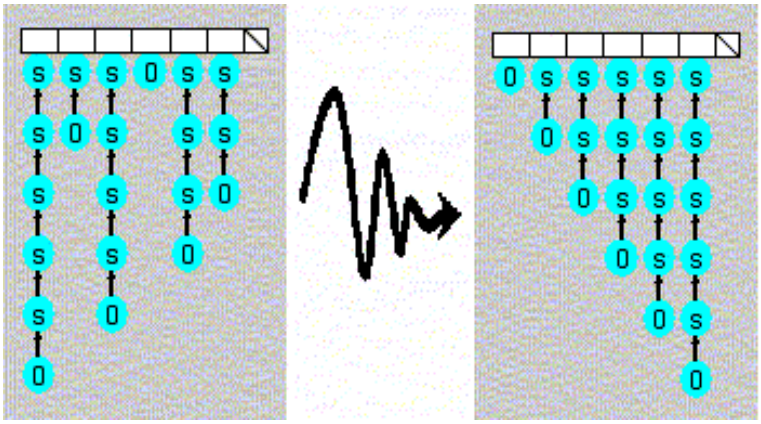


図 5.13: 実行のアニメーション表示

```
push( s(s(0)),
push( s(s(s(0))),
push( s(s(s(s(0)))),
push( s(s(s(s(s(0))))),  empty))))))).
```

のようになる．この項の意味は (0, 1, 2, 3, 4, 5) であり，ソーティングが正常に終了したことがわかる．

5.4.3 ビューの反映について

項の構成要素である演算を選択して，演算に定義しているビューを呼び出すことで項に演算のビューを反映させることができるが，この反映の仕方には，次の 4 通りがある [58].

1. 選択した演算のみ
2. 選択した演算とその引数に含まれるすべての演算
3. 選択した演算を含む項全体
4. 選択した演算モジュール全体

何も指定しない場合には “2” が選択される．例えば，図 5.9に示す項の中に含まれる演算 push について見てみる．例題では，項の一番上にある演算 push を選択してビューの反映を行った．この場合，特に指定しなかったので “2” が選択され，一番上の演算 push 以下の全ての演算，すなわち，項全体に含まれる演算 push に対

してビューが反映される．この場合 “3” を選択しても同じ結果となる．また，仮に “1” を選択したとする．すると，一番上の演算 push のみにビューが反映されるだけで，項に含まれる残りの演算 push は何も影響がない．“1” を選択した結果として図 5.14のようになる． 演算 push に定義したビュー（図 5.10）を見れば分かるとお

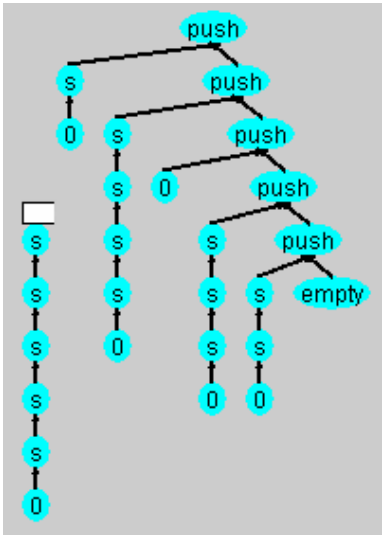


図 5.14: ビューの反映

り，このビューでは，ある Stack の左側に配置して，さらにその下側に要素を置くという配置を行っている．この場合 Stack とは， 1 番上の演算 push の第 2 引数に含まれる部分項全体となる．部分項に含まれるすべての演算についてレイアウトを計算した後で，一番上の演算 push の配置を計算するというボトムアップ的にレイアウトの計算を行っているため，結果として図 5.14のようになる．

5.4.4 アニメーションの設定

図 5.15にソーティングにおけるアニメーションの設定例を示す． ここでは，ソー

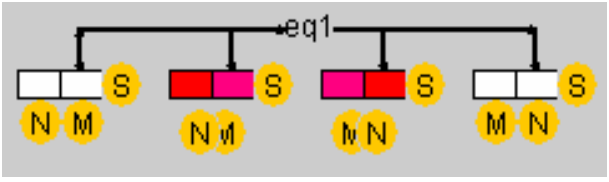


図 5.15: ソーティングにおけるアニメーション調整

ティングの入れ替え時に注目するために交換する要素を含む演算 push を色付けし，

また，２つの要素の位置を少し下にずらした状態でかつ交差するように配置している．このアニメーションの設定は，図 5.6と同じように行える．違いは，図 5.6では，等式の左辺と右辺の間に１つしか補間となる表現を挿入しなかったが，ここでは２つの表現を挿入している点である．デフォルトの設定では，左辺から右辺まで１秒のタイミングで書き換えるようになっている．したがって，図 5.15のように２つの表現を挿入した場合には，アニメーションの各フレームは１／３秒で表示されるようになる．

図 5.6上にある演算　push には，図 5.16 で示すように全部で３つ （デフォルト・

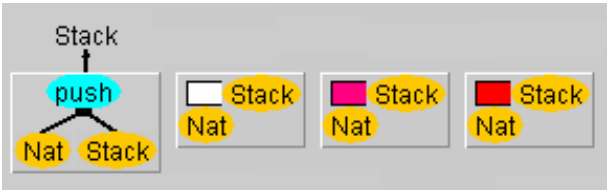


図 5.16: 演算 push に定義したマルチビュー

ビューを入れれば４つ）のビューが定義されている．このように，一つの演算に対して複数のビュー（マルチビュー）を定義することができる．この機能を使って，あらかじめ演算　push に色別のビューを図 5.16 のように定義しておけば，図 5.6のようなアニメーションの設定も，その定義したビューを切り替えるだけで簡単にビューの切り替えを行うことができる．

図 5.17にアニメーションの設定後のプログラム実行例を示す． 図 5.17は，ソーティ

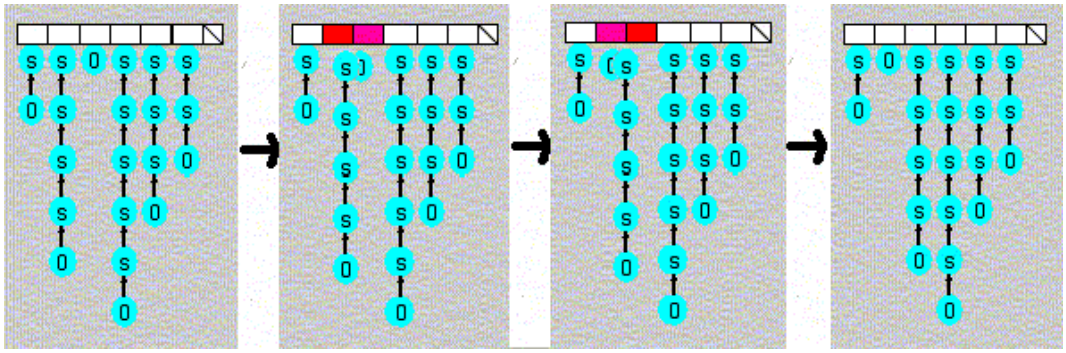


図 5.17: カスタマイズを利用した実行のアニメーション

ングの書換え過程の１つのリダクションをを示している．ソーティングアルゴリズムにおいて，データを交換する箇所を色付けと位置調整により強調することで，現在動作しているリダクションの過程を分かりやすく表示できるようになった．

ビジュアルプログラミングにおいて，ビューを用いてプログラムの実行過程を視覚的に捉えることができることは，デバッグの補助に役立ち，また，言語の初心者にとっては理解の助けとなる．

第 6 章

結論

本論文では，ビジュアルプログラミングにおいてプログラム編集上のすべての操作が図式表現への直接操作により実現できることを述べた．この図式表現への直接操作とは，メニューやダイアログボックスのような既存の GUI ををを用いないでも済むような操作のことであり，主にドラッグ＆ドロップ手法により実現されている．このドラッグ＆ドロップ手法を使ったプログラム編集のためにノード指向の操作を提案した．また，代数仕様記述言語で最も頻繁に使われる項においても，変数の具体化による項の編集によってノード指向の編集が行えることを示した．更にこの項の編集では，変数という操作対象の限定を行うことによって，シンタックスエラーが無いように手早く編集が行えるようになる．更に，代数仕様記述言語のためのビジュアルプログラミングシステム CafePie を試作した．このシステム上でノード指向の操作手法を実装することにより，プログラム編集上のすべての操作が実現できることを示した．また，テキストエディタと同じくらいの時間でプログラムの入力ができることを実験により確認した．ビジュアルプログラミングの特徴である構造の把握を行いつつプログラム編集が行えるため，本手法は非常に有用である．また，情報の提示や把握における問題の解決のために視覚情報のカスタマイズの重要性について述べた．操作途中において結果をユーザに対話的に提示することで，直接操作による視覚情報のカスタマイズを実現した．この直接操作による視覚情報のカスタマイズをシステムに実装し評価実験を行うことにより，視覚情報のカスタマイズが手早く行えるようになることを確認した．最後に，視覚情報のカスタマイズをプログラム実行に適用することを行った．プログラム編集だけでなくプログラムの実行表示においても視覚情報を変更が可能であることをシステムに実装することにより示した．また，プログラム実行表示にアニメーションの定義を視覚的に行えるようにした．アニメーション定義上で視覚情報を変更

できるため，動きによる強調以外にも，色を変更するような見た目の強調も可能であることが言える．

本論文で述べられている，ノード指向の直接操作，変数の具体化による項の編集，視覚情報のカスタマイズにおける直接操作を用いた配置方法は一般にも有用であると思われる．また，プログラム実行におけるアニメーションの設定は，前後置換ルールを用いて動作を定義するようなビジュアルプログラミングシステムにも適用可能である．また，代数仕様記述言語のためのビジュアルプログラミングシステムを作成したこと，項の視覚情報を変更可能にし、項書換え系の表示に利用できることを示したこと，によって代数仕様記述言語へ貢献したと言える．本論文で述べた，メニューやダイアログボックスのような従来の GUI を用いないような操作手法と状況に応じて適切に視覚化を行えるようにする視覚情報のカスタマイズ手法を応用することで，プログラマの負荷を少なくするような使いやすいビジュアルプログラミングシステムを構築できるものと思われる．

今後の展望としては，視覚情報のカスタマイズを 2 次元的な広がりを持つグラフにも対応できるように拡張することや，項の視覚化を仕様の検証に適用することなどが挙げられる．また，項の編集をより大きく捉えてシグニチャの編集まで行えるようにすることは非常に重要である．シグニチャの編集，すなわちシステムのインタフェースデザインまで拡張することによって，ソフトウェア設計という大枠を考察できるようなシステムを構築できる可能性を秘めているからである．プログラミングというと，C や Java という手続き型言語を用いたコーディング主体で捉えることが多いように思われる．コーディングよりもインタフェースデザインというシステムの設計にこそビジュアルプログラミングを用いる意義が見出せる可能性があると考えている．

謝辞

大学の卒業研究から約 6 年間にわたりご指導下さった筑波大学電子・情報工学系の田中二郎教授に厚く御礼を申し上げます．田中教授には，本研究を進める上で厳しいながらも適切な助言を数多くいただきました．また，筑波大学電子・情報工学系の西原清一教授，福井幸男教授，北川博之教授，そして，北陸先端科学技術大学院大学の二木厚吉教授には，本論文の審査にあたって数多くの貴重な助言・討論をいただきました．深く感謝いたします．

情報処理振興協会（IPA） の CafeOBJ プロジェクトにおいて，二木教授，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．筑波大学電子・情報工学系志築文太郎助手，同三浦元喜助手には，研究について多くの助言をいただきました．また，糸賀裕弥氏，飯塚和久氏，亀山裕亮氏，酒寄保隆氏， Simona Mirela Vasilache さんをはじめとする田中研究室の皆様及び，遠藤浩通氏， Ali Jauhar 氏，宮城幸司氏，奥村穂高氏，神谷誠氏，大芝崇氏，野口隆佳氏，宮下貴史氏，西名毅氏をはじめとする田中研究室 OB の皆様には，データ収集や実験，研究を進める上での議論に参加・協力していただいき，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．

また，大学院の体育・水泳を担当して下さいました，坂田勇夫氏，高橋伍郎氏とその他の関係者の皆様，水泳の楽しさを教えて下さいまして本当にありがとうございました．そして，喫茶店タートルの東川夫妻，いつも美味しい食事をありがとうございました．最後に，今まで温かい目で見守ってくださいました父章と母美代子，そして，いつも心の支えに成ってくださいました最も尊敬する兄建には，感謝を尽くしたくても尽くしようがありません．今日に至るまでの温かいご支援とご配慮，本当にありがとうございました．

参考文献

[1] 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.

[2] 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.

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

[4] Eric J. Golin and Steven P. Reiss. The Specification of Visual Language Syntax. *Journal of Visual Languages and Computing*, Vol. 1, No. 2, pp. 141–157, 1990.

[5] Margaret M. Burnett and Marla J. Baker. A Classification System for Visual Programming Languages. *Journal of Visual Languages and Computing*, Vol. 5, No. 3, pp. 287–300, 1994.

[6] 田中二郎. ビジュアルプログラミング. ビジュアルインタフェース - ポスト GUI を目指して -, bit 別冊, pp. 65–78, 1996.

[7] Ephraim P. Glinert and Steven L. Tanimoto. PICT: An Interactive Graphical Programming Environment. *IEEE Computer*, Vol. 17, No. 11, pp. 7–25, 1984.

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

[9] Kazunori Ueda. Guarded Horn Clauses. ICOT Technical Report TR-103, Institute for New Generation Computer Technology (ICOT), 1985. (Also in Concurrent Prolog: Collected Papers, Vol. 1, The MIT Press, pp.140-156, 1987).

[10] Jiro Tanaka. PP : Visual Programming System For Parallel Logic Programming Language GHC. In *Parallel and Distributied Computing and Networks '97*, pp. 188–193, August 1997.

[11] Vijay A. Saraswat, Ken Kahn, and Jacob Levy. Janus: A Step Towards Distributed Constraint Programming. In *Proceedings of the North American Conference on Logic Programming*, pp. 431–446. MIT Press, October 1990.

[12] Kenneth M. Kahn and Vijay A. Saraswat. Complete visualizations of concurrent programs and their execution. In *Proceedings of the 1990 IEEE Workshop on Visual Languages (VL'90)*, pp. 7–15, October 1990.

[13] George W. Furnas. New Graphical Reasoning Models for Understanding Graphical Interfaces. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'91)*, pp. 71–78, April 1991.

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

[15] Alan Cypher and David Canfield Smith. KidSim: End User Programming of Simulations. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'95)*, pp. 27–34, May 1995.

[16] Michael P. Stovsky and Bruce W. Weide. Building Interprocess Communication Models Using STILE. In *Proceedings of the 21st Hawaii International Conference on Systems Sciences*, Vol. 2, pp. 639–647, January 1988.

[17] David C. Smith and Joshua Susser. A Component Architecture for Personal Computer Software. In Brad A. Myers, editor, *Languages for Developing User Interfaces*, pp. 31–56. Jones and Bartlett Publishers, 1992.

[18] Michael G. Vose and Gregg Williams. LabVIEW: Laboratory Virtual Instrument Engineering Workbench. *BYTE*, Vol. 11, No. 9, pp. 84–92, September 1986.

[19] Marc H. Brown. Zeus: A System for Algorithm Animation and Multi-View Editing. In *Proceedings of the 1991 IEEE Workshop on Visual Languages (VL'91)*, pp. 4–9, October 1991.

[20] Kenneth Cox and Gruia-Catalin Roman. Visualizing Concurrent Computations. In *Proceedings of the 1991 IEEE Workshop on Visual Languages (VL'91)*, pp. 18–24, 1991.

[21] 竹内郁雄. 「作ってなんぼ」のナラトロジ. 記号学研究 14, 日本記号学会編, 東海大学出版会, pp. 105–118, 1994.

[22] John Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of ACM*, Vol. 21, No. 8, pp. 613–641, August 1978. (1977 ACM Turing Award Lecture).

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

[24] Masashi Toyoda, Buntarou Shizuki, Shin Takahashi, Satoshi Matsuoka, and Etsuya Shibayama. Supporting Design Patterns in a Visual Parallel Data-flow Programming Environment. In *Proceedings of the 1997 IEEE Symposium on Visual Languages (VL'97)*, pp. 76–83, April 1997.

[25] Joseph A. Goguen, James W. Thather, Eric G. Wagner, and Jesse B. Wright. Abstract Data Types as Initial Algebras and the Correctness of Data Structure. In *Computer Graphics, Pattern Recgnition and Data Structure*, pp. 89–93. IEEE, May 1975.

[26] Joseph A. Goguen and Jose Meseguer. Order-Sorted Algebra 1: Equational Deducation for Multiple Inheritance, Polymorphism, Overloading and Partial Operations. Technical Report, SRI International, 1989.

[27] Razvan Diaconescu and Kokichi Futatsugi. Logical Semantics for CafeOBJ. Technical Report IS-RR-96-22S, JAIST, 1996.

[28] Kokichi Futatsugi and Ataru Nakagawa. An Overview of CAFE Specification Environment – An Algebraic Approach for Creating, Verifying, and Maintaining Formal Specifications over Networks –. In *Proceedings of the 1st Interna-*

tional Conference on Formal Engineering Methods (ICFEM'97), pp. 170–181, November 1997.

[29] Ataru T. Nakagawa, Toshimi Sawada, and Kokichi Futatsugi. *CafeOBJ User's Manual*. IPA, Tokyo, Japan, 1997.

[30] Razvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report – The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification –*, Vol. 6 of *AMAST Series in Computing*. World Scientific, 1998.

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

[32] Annette Wagner, Patrick Curran, and Robert O'Brien. Drag Me, Drop Me, Treat Me Like an Object. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'95)*, pp. 525–530, May 1995.

[33] 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, 1998. 筑波大学 第三学群 情報学類 卒業論文.

[34] 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.

[35] Tohru Ogawa and Jiro 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, October 1998.

[36] Tohru Ogawa and Jiro Tanaka. CafePie: A Visual Programming System for CafeOBJ. In Kokichi Futatsugi, Ataru T. Nakagawa, and Tetsuo Tamai, editors, *Cafe: An Approach to Industrial Strength Algebraic Formal Methods*, pp. 145–160. Elsevier Science, 2000.

[37] Tohru Ogawa and Jiro Tanaka. CafePie – A Visual Programming Environment for CafeOBJ. *Journal of INFORMATION*, Vol. 4, No. 3, pp. 343–362, 2001.

[38] 小川徹, 田中二郎. ドラッグ&ドロップを用いたビジュアルプログラミングシステム. 情報処理学会論文誌 プログラミング, Vol. 43, No. 1 (PRO13), pp. 36–47, 2002.

[39] 杉山公造. グラフ自動描画法とその応用 – ビジュアルヒューマンインタフェース –. 社会法人 計測自動制御学会編, コロナ社, July 1993.

[40] 早野浩生. ドラッグ&ドロップでスクリプトの記述が可能なシェル. コンピュータソフトウェア, Vol. 16, No. 5, pp. 68–71, 1999.

[41] 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.

[42] Adele Goldberg and David Robson. *Smalltalk-80 : the language and its Implementation*. Addison-Wesley, 1983.

[43] Adele Goldberg. *Smalltalk-80 : the Interactive Programming Environment*. Addison-Wesley, 1984.

[44] 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 1986.

[45] Glenn E. Krasner and Stephen T. Pope. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System, 1988.

[46] Eric J. Golin. Parsing Visual Languages with Picture Layout Grammars. *Journal of Visual Languages and Computing*, Vol. 2, No. 4, pp. 371–394, 1991.

[47] Kim Marriott. Constraint multiset grammars. In *Proceedings of the 1994 IEEE Symposium on Visual Languages (VL'94)*, pp. 118–125, October 1994.

[48] Steven P. Reiss, Scott Meyers, and Carolyn Duby. Using GELO to Visualize Software Systems. In *Proceedings of the 2nd ACM Symposium on User Interface Software and Technology (UIST'89)*, pp. 149–157, November 1989.

[49] 小川徹. 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, 2000. 筑波大学 博士課程 工学研究科 電子・情報工学専攻 修士論文.

[50] Tohru Ogawa and Jiro Tanaka. Realistic Program Visualization in CafePie. In *Proceedings of the fifth World Conference on Integrated Design and Process Technology (IDPT'99)*, June 2000. (CD-ROM), 8 pages, Copyright (c) 2000 by the Society for Design and Process Science, (SDPS).

[51] 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.

[52] 小川徹, 田中二郎. データ構造の視覚的カスタマイズとプログラム実行の視覚化. インタラクティブシステムとソフトウェア VIII, 日本ソフトウェア科学会 WISS2000, pp. 85–90. 近代科学社, 2000.

[53] David Kurlander and Steven Feiner. Inferring constraints from multiple snapshots. *ACM Transactions on Graphics (TOG'93)*, Vol. 12, No. 4, pp. 277–304, October 1993.

[54] Ken Miyashita, Matsuoka Satoshi, Takahashi Shin, and Akinori Yonezawa. Interactive Generation of Graphical User Interfaces by Multiple Visual Examples. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'94)*, pp. 85–94, 1994.

[55] Satoshi Matsuoka, Shin Takahashi, Tomihisa Kamada, and Akinori Yonezawa. A general Framework for Bi-directional Translation between Abstract and Pictorial Data. *ACM Transactions on Information System (TOIS)*, Vol. 10, No. 4, pp. 408–437, 1992.

[56] 二木厚吉, 森彰, 萩谷昌己, 澤田寿実, 瀬尾明志, 加藤賢次郎, 石黒正揮. 開放型分散環境でのソフトウェア部品検証システムの研究開発. 情報処理振興事業協会 (IPA), 次世代基盤技術 平成 13 年度 成果報告集, 2002. (CD-ROM), 8 pages.

[57] 小川徹, 田中二郎. CafePie: プログラムビューのカスタマイズ機能によるプログラム実行の視覚化. 日本ソフトウェア科学会 第 17 回大会論文集, 2000. (CD-ROM), 4pages.

[58] Tohru Ogawa and Jiro Tanaka. Visualization of Program Execution via Customized View. In *Proceedings of 5th Asia Pacific Conference on Computer Human Interaction (APCHI2002)*, Vol. 2, pp. 823–832, November 2002.

著者論文リスト

【本研究に関する論文】

[1] Tohru Ogawa and Jiro Tanaka. CafePie – A Visual Programming Environment for CafeOBJ. *Journal of INFORMATION*, Vol. 4, No. 3, pp. 343–362, 2001.

[2] 小川徹, 田中二郎. ドラッグ&ドロップを用いたビジュアルプログラミングシステム. 情報処理学会論文誌 プログラミング, Vol. 43, No. 1 (PRO13), pp. 36–47, 2002.

[3] Tohru Ogawa and Jiro 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, October 28-30 1998.

[4] Tohru Ogawa and Jiro Tanaka. Realistic Program Visualization in CafePie. In *Proceedings of the fifth World Conference on Integrated Design and Process Technology (IDPT'99)*, June 4-8 2000. (CD-ROM), Copyright (c) 2000 by the Society for Design and Process Science, (SDPS), 8 pages.

[5] Tohru Ogawa and Jiro Tanaka. Visualization of Program Execution via Customized View. In *Proceedings of 5th Asia Pacific Conference on Computer Human Interaction (APCHI2002)*, Vol. 2, pp. 823–832, November 1-4 2002.