筑波大学大学院博士課程

システム情報工学研究科修士論文

3次元ビジュアルプログラミングにおける
実行アニメーションとデバッグ手法

岡村 寿幸

(コンピュータサイエンス専攻)

指導教官　田中 二郎

2004年1月

概要

本研究ではビジュアルプログラミングの手法を用いてプログラムを視覚化し，その視覚化されたプログラムの実行をアニメーションを用いて視覚化する手法を提案し，実装を行った．アニメーションはユーザが実行中の変化を掴みやすくするといったことに重点を置いている．また，プログラムの実行の進み具合をユーザが調節できるようになっており，ユーザが任意に実行状態を調節し，観察ができるような工夫を行っている．

また，本論文では実行アニメーションを利用してプログラムのデバッグを行う手法を提案する．本研究の手法では，プログラム実行中に実行をユーザがより細かく操作するための機能を追加している．また，正しい動作を行っている部分とそうでない部分をユーザが色分けするための機能を追加している．これらの機能により，バグの存在する場所を視覚的に絞り込んでいくことでデバッグを行う．

目次

第1章　序論 **1**

第2章　準備 **3**

2.1 3次元ビジュアルプログラミングシステム 3D-PP 3
2.1.1 並列論理型言語 GHC 3
2.1.2 ビジュアルプログラミングシステム PP と PP を拡張したシステム . . 5
2.1.3 3D-PP 8

第3章　3次元ビジュアルプログラミングにおける実行の視覚化 **10**

3.1 実行の過程をトレースするための手法 10
3.2 3次元ビジュアルプログラミングシステム Animation-3DPP 12
3.2.1 Animation-3DPP のプログラム表現 12
3.3 実行アニメーション 17
3.4 各オブジェクトの実行アニメーション 20
3.4.1 実行アニメーション – ゴールのリダクション 20
3.4.2 実行アニメーション – ファンクタのユニフィケーション 21
3.4.3 実行アニメーション – オペレータ 22
3.5 実装 23
3.6 実行アニメーションの例 23

第4章　3次元ビジュアルプログラミングにおけるデバッグ手法 **29**

4.1 バグ 29
4.2 デバッグのため機能と手法 29
4.2.1 実行の巻き戻し 29
4.2.2 プログラムの部分実行，ステップ実行 30
4.2.3 実行一時停止中の書き換え 30
4.2.4 色付きオブジェクト 31
4.3 デバッグ手法 35
4.4 デバッグの例 36

第5章　関連研究 **41**

5.1 実行の視覚化に関する研究 41
5.1.1 Pictorial Janus 41

5.1.2 SAM 41
5.2 デバッグに関する研究 42
5.2.1 Algorithmic Debugging of GHC 42
5.2.2 Instant Replay of Debugging 42

第 6 章 まとめと今後の課題 44

謝辞 45

参考文献 46

図目次

2.1 GHC プログラムの例 3
2.2 PP の実行画面 6
2.3 newPP の実行画面 7
2.4 PP を拡張したシステムの実行画面 7
2.5 3D-PP の実行画面 9

3.1 引数オブジェクト 13
3.2 オブジェクトの階層表示 14
3.3 Animation-3DPP のオブジェクト 15
3.4 グラフの例 (リスト [a,b,c,d,e]) 17
3.5 オペレータの例 (20 + 30) 17
3.6 グラフの例 (append([a,b,c],[d,e],Out)) 17
3.7 オペレータの実行アニメーション 19
3.8 並列に実行されるオペレータ 19
3.9 ゴールの実行アニメーション 21
3.10 ファンクタのユニファイの実行アニメーション 22
3.11 Animation-3DPP での reverse/2 24
3.12 実行アニメーションの例 – ゴール reverse/2 のリダクション 26
3.13 実行アニメーションの例 – ファンクタのユニフィケーション 27
3.14 実行アニメーションの例 – 図 3.13以降の様子 28

4.1 実行停止中の値の書き換え 31
4.2 実行時の色付きオブジェクトの例 1 32
4.3 実行時の色付きオブジェクトの例 2 33
4.4 実行の巻き戻し時の色付きオブジェクト 33
4.5 節の色付きオブジェクト 34
4.6 自動的に色が付けられたゴール (append/3) 35
4.7 バグのある GHC プログラムの例 37
4.8 バグのあるプログラムの実行結果 39
4.9 ゴールの動作の調査 – 正しく動作 (append/3) 39
4.10 ゴールの動作の調査 – 正しく動作 (wrongrev/2) 40
4.11 ゴールの動作の調査 – 間違いを発見 (wrongrev/2) 40

第1章　序論

ビジュアルプログラミングとは，図形や絵，アイコンなどを用いて，プログラムの要素を抽象化して表し，それらをマウスなどのデバイスによって直接操作することが可能なものにする手法である．ビジュアルプログラミングの手法を用いて図形を操作，編集することによってプログラムの作成，実行の過程などを視覚的に表現するシステムをビジュアルプログラミングシステムと呼ぶ．ビジュアルプログラミングシステムにはプログラムの要素間の関係，処理の流れ等をテキストを用いたプログラムの記述よりも直感的に表現できるという利点がある．ビジュアルプログラミングシステムの中で実行の視覚化を行っているシステムの例としては Pictorial Janus[1]，PP[2, 3, 4, 5] 等を挙げることが出来る．

近年はコンピュータのグラフィック描画能力の向上により，表現に用いられる図形も 2 次元の図形や線だけではなく 3 次元図形などが用いられるようになってきた．3 次元図形が用いられているビジュアルプログラミングシステムには，SAM[6]，Cube[7]，3D-Visulan[8] 等がある．また我々は，コンピュータ上の仮想的な 3 次元空間とそこに配置されるオブジェクトを用いてプログラムの視覚化を行う 3 次元ビジュアルプログラミングシステム 3D-PP[9, 10] の研究を進めている．

本研究では，ビジュアルプログラミングの手法を用いてプログラムの実行を視覚化する手法に関して考察した．プログラムの実行においては，記述されている定義や処理に基づいてデータが書き変わるなどのさまざまな変化が起こることによって実行が進められていく．よって，実行の過程を知るためには，プログラムの実行をユーザ自身が観察しやすいように操作できるようにする必要がある．それには，変化するデータの値を表示，現在実行されている処理の名称などを表示，プログラムの実行を任意の時点で止めることができることが重要である．また，これまでの実行を視覚化するシステムでは，プログラムの実行を図形を用いて視覚化することが出来た．しかし，実行中にユーザが実行状態を観察しやすいようにするために実行の速度などを操作することは出来なかった．そこで本研究では，以上のことを考慮してアニメーションを用いて実行を視覚化し，アニメーションを見る方向や動く速度を変えることによって実行をユーザ自身が操作できるような手法を提案する．また，この手法を 3 次元ビジュアルプログラミングシステム Animation-3DPP 上に実装した．実行アニメーションでは，プログラムの実行をビジュアルプログラミングシステム上で表すために，変化の前後を把握しやすいように表している．さらに，プログラムの実行の進み具合をユーザが実行状態を観察しながら調節できるようにしている．

また，本論文では実行アニメーション用いてプログラムのデバッグを行う手法を提案する．デバッグを行う手法では，実行アニメーションに実行状態の観察がしやすくするための手法

をとして実行の巻き戻し，部分実行，ステップ実行の機能を追加した．また，実行中にオブジェクトを塗りわける機能を追加した．この機能を本研究では，色付きオブジェクトと呼んでいる．色付きオブジェクトの機能によって，正しい動作を行っている部分とそうでない部分を色分けすることが可能となる．本研究では，これらの機能を実行アニメーションとともに用いることで，デバッグを行う．

本論文の構成は以下のようになっている．まず，第2章では準備として Animation-3DPP の基盤となったシステムである，ビジュアルプログラミングシステム PP と 3 次元ビジュアルプログラミングシステム 3D-PP について述べる．また，これらのシステムと Animation-3DPP のベースとなるプログラミング言語である並列論理型言語 GHC について述べる．第 3章では，本研究の提案手法である実行アニメーションについての解説と実行アニメーションによって視覚化された実行の例を挙げる．第 4章では，実行アニメーションを利用したデバッグ手法について述べる．第 5章では，本研究と関連する研究として，実行の視覚化を行っているシステムと並列論理型言語のデバッグを行うための手法を挙げる．そして，第 6章で結論と今後の課題について述べる．

第2章　準備

2.1　3次元ビジュアルプログラミングシステム 3D-PP

3次元ビジュアルプログラミングシステム 3D-PP は，並列論理型言語 GHC を図式表現を用いて視覚化するビジュアルプログラミングシステム PP をベースとして，3次元への拡張を行ったものである．本節では並列論理型言語 GHC と PP について述べる．また，3D-PP と 3D-PP に用いられている手法について述べる．

2.1.1　並列論理型言語 GHC

並列論理型言語 GHC(Guarded Horn Clauses)[11, 12] は並列実行に適したプログラム言語であり，論理型言語としての特徴を備えている．論理型言語のプログラムは，述語論理の公理を記述することによって表される．また，実行は述語論理をルールに従って書き換えるリダクション (簡約化) によって行われる．

図 2.1は GHC で書かれたプログラムの例である．図 2.1のプログラムでは，与えられたリストを逆順にする `reverse` が定義されており，`reverse` を用いてリスト `[a,b,c,d]` を逆順にしたリストを出力ストリームに出力するというプログラムが記述されている．`append` は 2 つのリストを結合するという述語の定義である．

```
main            :-  reverse([a,b,c,d],Out),
                    io:outstream([print(Out),nl]).

reverse(L,O)    :- L = []    | O = [].
reverse(L,O)    :- L = [H|T] | append(O2,[H],O),reverse(T,O2).

append(L1,L2,O):- L1 = []   | O = L2.
append(L1,L2,O):- L1 = [H|T]| O = [H|O2], append(T,L2,O2).
```

図 2.1: GHC プログラムの例

GHC プログラムは次のような要素から構成され，以下のように記述される．

- 節
  節はゴールのリダクションのルールを記述する役割をもっており，GHC プログラムの基本単位である．節は,

```
pred(Arg1,Arg2,...) :- guard | body.
```

と記述され，各部分には以下のような名前と意味がある．

– ヘッド
  冒頭から “:-” までの部分はヘッドと呼ばれる．ヘッドはこの節の定義がどの述語に関わるかを表し，この節においての引数の個数と引数として与えられているものを示している．“pred” は述語名であり，“Arg1”, “Arg2” はそれぞれ引数として与えられているものである．

– ガード
  “:-” から “|” までの部分はガードと呼ばれる．ガードはゴールのリダクションの際に節を選択するための条件が記述されている．ガードにはユニフィケーションの他に任意のゴールを書くことが出来る．

– ボディ
  “|” から “.” まではボディと呼ばれる．ボディには節が選択されたときにどのような定義に置き換えるかが記述されている．ボディにはユニフィケーションの他に，他のゴール，あるいは自分自身を呼び出すゴールを書くことが出来る．

節は述語名が同じであっても引数の数が異なる場合，別のものとして扱われる．よって節は述語名と引数から append/3，reverse/2 などの様に述語名の後，/とともに引数の総数を付けて呼ばれる．

• ゴール
  ゴールは図 2.1の例では記号 “:-” の右側に現れる “goal(X,Y,...)” という形をした物である．X,Y,... はゴールに与えられる引数である．ゴールは実行時の述語の呼び出しを表すものであり，0 個以上の引数を持っている．ゴールは与える引数によって異なった動作をし，引数を介して処理結果を返すことができる．図 2.1の例では，reverse(L,O) の L，O や append(L1,L2,O) の L1，L2，O が引数である．

• アトム，ファンクタ，コンス
  GHC のデータには内部構造を持たず，その値だけに意味があるアトムと，構造を持ったデータであるファンクタがある．また，ファンクタの一種としてコンスと呼ばれる構造を持ったデータが存在する．アトムには文字列アトム (“a”, “atom1” など)，数値アトム (1,2,100 など) がある．ファンクタは構造を持ったデータであり，

```
name(a,b,...)
```

  という形をしていて引数を持つことができる．引数はどのようなデータでも良く，また，入れ子構造をしていても良い．下の式はファンクタの例である．

```
f(g(a,b), h(c))
```

  コンスは [Head|Tail] の様に記述され，コンスを組み合わせることによってリストを作ることができる．リストは，Head を表す第 1 引数にリストを構成する要素を与え，Tail を表す第 2 引数に他のコンスまたはリストの終端を表すアトム “[ ]” を与えること

で構築される構造データである．終端を表すアトムは “nil” と記述される場合もある．また, アトム “[ ]” は空リストとも呼ばれる．リストは，例えば先頭から順に a,b,c の要素を持つリストの場合，[a|[b|[c|nil]]] と記述される．また，[a,b,c] の様に括弧を省略した記述方法も使用される．

- オペレータ
  GHC では本来ゴールの一種であるが，本論文では，四則演算 (加算，減算，乗算，除算) と剰余演算，および比較演算を行うものをオペレータと呼ぶこととする．GHC では，オペレータは 2 個のアトムの間に “+”(加算), “-”(減算), “=:=”(等号) などといった記号のがあるという形で表される．四則演算，剰余演算のオペレータは，例えば加算であるなら 2 つ数値アトムの値を足した数といったように演算の結果をアトムで出力する．比較演算は節のガードに現れ，入力引数などが他の値と等しい，等しくない，または，2 つの数値の大小比較などの条件を記述するために用いられる．

ユニフィケーション

GHC プログラムにおいて “=” 記号の両辺に値を書いた形をしたものと，節に存在する変数と同名の変数が同じ節の中に現れた形のものをユニフィケーション (Unification) と呼ぶ．ユニフィケーションは対応する 2 つのものが等しいという意味を表している．また，ユニフィケーションによって 2 つのものを等しいものにすることをユニファイ (Unify) と呼ぶ．ただし, “=” 記号によるユニフィケーションは，ガードに出てきたときと，ボディに出てきたときで意味が異なる．ガードでのユニフィケーションは両辺が等しいという条件を表す．ボディでのユニフィケーションは両辺の値を等しいものにするという意味を持つ．また，節に存在する変数と同名の変数が同じ節の中に現れた場合のユニフィケーションでは，同名の変数は全て同じものであるという意味を持つ．

GHC の実行

GHC プログラムの実行は，節が定める書き換えルールに従ってゴールを書き換えるリダクションによって進められる．ゴールはこれ以上書き換えることのできないゴール “true” になった時点でリダクションを終了する．GHC プログラムの実行はすべてのゴールが “true” に書き換えられた時点で終了する．

2.1.2 ビジュアルプログラミングシステム PP と PP を拡張したシステム

ビジュアルプログラミングシステム PP[13] と PP を拡張したシステムでは，GHC プログラムの要素を平面上の円や線などの 2 次元の図形からなるグラフを用いて表現している．PP では，テキストで記述された GHC のプログラムを図式表現に変換することが出来る．また，図形を編集することによって GHC の節を定義することが出来る．また，GHC の実行の視覚化

を行うことが出来る．PP を拡張したシステムには，newPP[3]，viewPP[4]，LivePP[5] がある．PP を拡張したシステムの研究では，プログラムの表現方法や編集方法，実行に関するさまざまな研究が行われた．例としては，節の定義のための入力方式，グラフ描画，アニメーション表示等の改良 (newPP)，GHC のサブセットをロードしてその実行を可視化する手法 (viewPP) である．また，プログラムの編集と実行を単一のビューの上で行なえるようにする手法の研究 (LivePP) もある．これらの手法のうち，単一のビューの上で編集と実行の両方を行う手法は後述する 3D-PP，Animation-3DPP にも受け継がれており，3D-PP，Animation-3DPP では全ての操作，表示は単一のビュー上で行うことが出来る．

図 2.2は PP の実行画面である．また，図 2.3，図 2.4は PP を拡張したシステムの実行画面である．

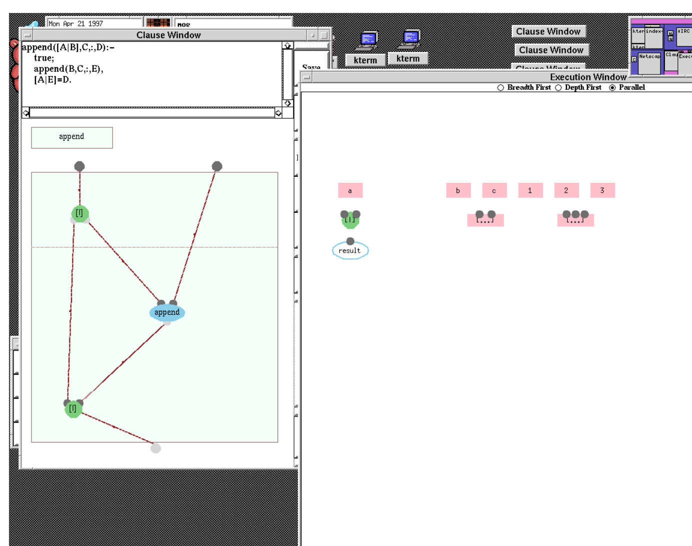


図 2.2: PP の実行画面

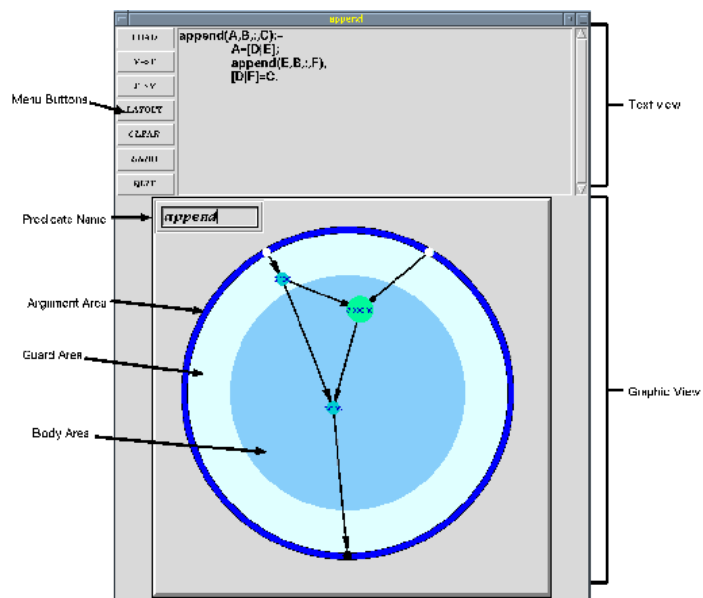
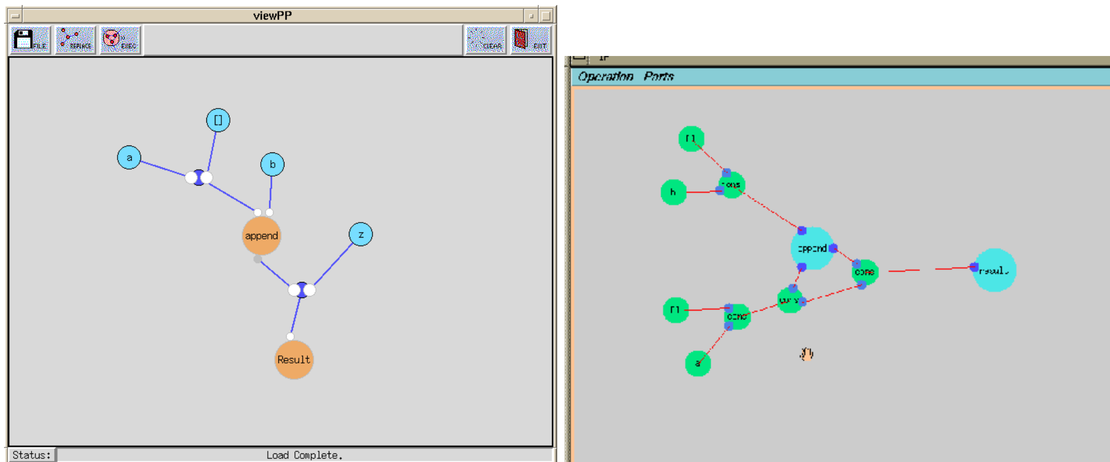


図 2.3: newPP の実行画面



(a) viewPP の実行画面

(b) LivePP の実行画面

図 2.4: PP を拡張したシステムの実行画面

2.1.3　3D-PP

3D-PP[9, 10] はビジュアルプログラミングシステム PP を 3 次元オブジェクトを扱えるように拡張したものである．図 2.5は 3D-PP の実行画面である．3D-PP では，視覚化にコンピュータ上の仮想的な 3 次元空間とそこに配置される 3 次元オブジェクトを用いている．また，3 次元オブジェクト同士をエッジ (線) で結線することによって作られる 3 次元グラフを用いている．3D-PP が 3 次元空間を用いた背景には以下のような目的がある．

- 大規模プログラミングへの対応
  ビジュアルプログラミングシステムは図形を用いてプログラムを表すため，図形が大きな場所を占めてしまうことがある．プログラムが大きくなり，扱うプログラム要素が増えてくると 2 次元では図形が配置しきれなくなってしまう．そこで，図形を配置する次元を 1 つ増やし，3 次元空間を利用することで配置可能な図形の数の上限を大きくする．

- レイアウト自由度の向上
  平面上では，図形同士が重なりあってしまったり，オブジェクト同士を線で繋ぐ場合には 2 本以上の線が交差してしまったりすることが避けにくい．3 次元空間を利用することによって，図形の重なりや線の交差を防止する．また，2 次元図形には出来ないようなより自由度の高い配置が可能となる．

- リアリティの向上
  我々の住んでいる世界は 3 次元であるので，3 次元でビジュアルな表現を行うことにより，より直感的な表現が出来る．

3D-PP ではマウスを用いて 3 次元オブジェクトを操作，編集することによってプログラム編集を行う．そのため，3 次元オブジェクトをユーザが簡単に操作し，見やすく配置するための手法が研究され実装されている．具体的には，拡張ドラッグ&ドロップ手法 [14]，強化された直接操作手法 [15]，自動レイアウト [16] といった手法が用いられている．また，3 次元メニュー [17]，オブジェクトのグループ化 [18] といった研究も行われた．

拡張ドラッグ&ドロップの手法は，平面でのドラッグ&ドロップと同様の操作で 3 次元空間上の 3 次元のオブジェクトを操作することを可能にする手法である．強化された直接操作手法とは，3 次元空間内のオブジェクトに付加的な情報を加えることによってユーザが 3 次元オブジェクトの位置を把握しやすくし，操作しやすくするという手法である．3D-PP では，地面のオブジェクトと地面にオブジェクトの影を表示する手法が用いられている．地面のオブジェクトによって，ユーザは 3 次元オブジェクトの位置を把握しやすくなる．また，地面のオブジェクトはオブジェクトを移動させるときの目安になる．オブジェクトの影は地面のオブジェクトとオブジェクトとの位置関係を表し，こちらもオブジェクトを移動させるときの目安となる．自動レイアウト [16] の手法は，画面上の複数のオブジェクトを互いに重なり合ず，離れすぎない適度な位置に自動的に配置する手法である．これによって，ユーザは自動的に見やすい配置を得ることが出来る．3 次元メニューでは，オブジェクトのコピー&ペーストなどの操作を行うための項目を 3 次元オブジェクトによって表し，3D-PP と同じ画面上に

配置するという手法である．3 次元メニューは他のオブジェクトと同じように 3 次元空間内の好きな位置に配置できる．オブジェクトのグループ化は，複数のオブジェクトを一度に操作するための手法である．画面上のオブジェクトをいくつかのグループに分け，グループ単位での表示，非表示の切り替えやグループ単位での移動などの操作を行うことが出来る．

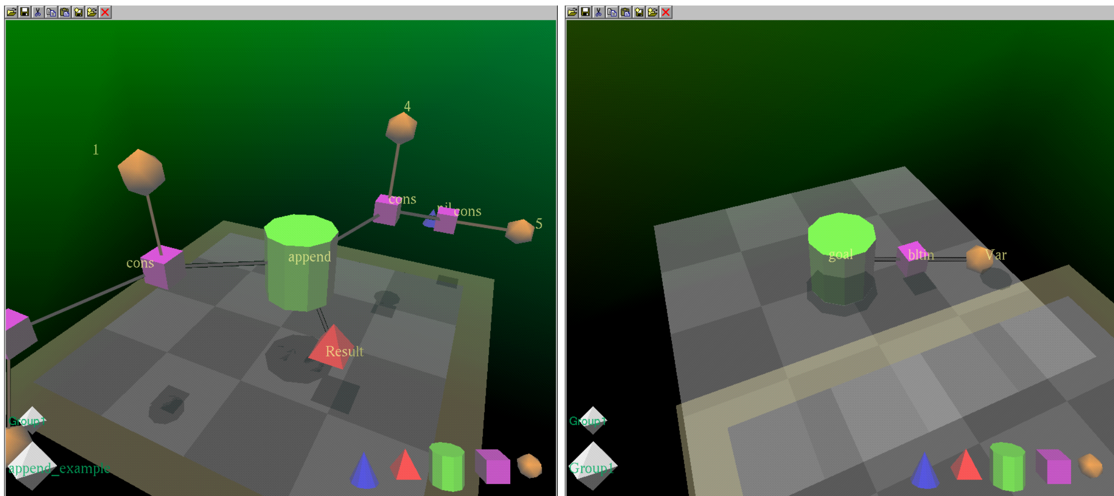


図 2.5: 3D-PP の実行画面

第3章　3次元ビジュアルプログラミングにおける実行の視覚化

本研究では，プログラムの実行をアニメーションで視覚化することを提案する．実行アニメーションの目的としては，プログラムの実行を見る事によって理解できること，編集したプログラムの実行をユーザが何度でもトレースし，テストできるようにすることが挙げられる．以下に実行アニメーションについて述べる．

3.1　実行の過程をトレースするための手法

テキストベースのプログラムであってもビジュアルプログラムであっても，プログラムの実行においては記述，表現されている処理に基づいてデータなどが書き変わるなどのさまざまな変化が起こることによって実行が進められていく．これらの実行時のデータ変化の様子や，ある時点で実行されている処理がどれであるかといった情報を知るためには次のようなことが重要である．

1. 変化するデータの値を出力または表示する (値の表示)
2. 実行される処理の名称，機能，与えられた引数などを表示する (処理の表示)
3. プログラムの実行を任意の時点で止めたり，少しずつ実行したりしてプログラムの実行状態を確認する (一時停止，ステップ実行)

テキストベースのプログラムで上記のことを示すための方法としては次のようなものがある．1，2 ではコンソール上やウィンドウ上にデータの値や，実行されている処理の名称などが出力されるという方法が用いられている．これらの出力はプログラマが確認のために出力命令などを用いて出力をさせる場合が該当する．また，C 言語および C++のデバッガである gdb が行う変数の値の表示なども該当する．3では，gdb 等のデバッガのブレークポイント機能やステップ実行機能などをあげることが出来る．デバッガではプログラム中に実行を一時停止する地点としてブレークポイントを設定する．その後，ステップ実行を行い 1 ステップずつ実行を進めることによって処理の過程を表示している．

ビジュアルプログラミングシステムにおいても 1，2はコンソール上や別ウィンドウにデータ値の出力，現在行われている処理の出力を行えば，実行時の様子をユーザに知らせることが出来る．しかし，ビジュアルプログラミングシステムでは，プログラムをビジュアルな表現で表している．よって，コンソール上などにテキストで出力を行った場合，ユーザは表示

されてくるテキストとビジュアルな表現を理解し，結びつける必要がある．別々のウィンドウに表示されているビジュアルな情報とテキストによる情報を結びつけるのはオブジェクトが少ない場合には容易である．しかし，オブジェクトが増えてくると困難となり，ユーザにとって大きな負担となる．この問題を解決するためにはビジュアルシステム上のいかなる時点でも同じビジュアルな表現で表す必要がある．

本研究では，ビジュアルプログラミングを用いて変化をアニメーションで表すのがよいと考えた．理由としては実行の視覚化には以下のことが重要であり，これらを実現するにはアニメーションが適当であると考えたからである．

- 実行による変化，出力はオブジェクトの変化で表す．
  これは，プログラム要素の変化はオブジェクトの変更で表すということである．変化のあったプログラム要素のオブジェクトをプログラム要素の変化後を表すように変更することで変化を見れるようにする．また，変化を見て分かるようにする．

- 表示されているプログラムの変化をなめらかに変化の前後が掴みやすい様に表す．
  オブジェクトが突然移動したり，いきなり大きく形を変えるような急激な変化は実行を観察する人がその変化に対してついていけなくなる恐れがある．すなわち，元々のものが何であったのか実行が進んだことによってどのような変化が生じたのかを見失ってしまう恐れがある．よって，オブジェクトは出来るだけ滑らかに移動させる必要がある．また，形や大きさが変わる際にはこの先どのように変化するのかを予想できるような仕組みを作る必要がある．

- プログラムを編集するときと同じオブジェクト，同じ表現で行う．
  プログラム編集時と実行時においてプログラム表現を変えた場合，ユーザは場面場面によって違ったプログラム表現を覚えなくてはならない．また，それらの情報を理解し結びつける必要が出てくる．これを避けるためにはプログラム実行の時のプログラム表現もプログラム編集時と同じ表現を用いる必要がある．

- プログラムの実行の進み具合などをユーザ自身が手軽に調節できるようにする．
  実行にともなう変化は簡単なプログラムでは少なく，複雑なプログラムになればなるほど多くなる．また，一つのプログラム中でも変化が多く連続して起こる部分とあまり変化せず間隔が空くような部分が存在する．これらの部分でユーザが自分に都合がよい速さで実行を進めることが出来るようにすることが重要である．プログラム実行の進み具合を調節できるようになるとユーザは変化のないところは速くする，変化のあるところはゆっくりにするといったことが出来るようになる．このようにすることでユーザが自分が実行を観察しやすいように速度を調節することが出来るようになる．これは，プログラムの実行を理解するための助けとなる．

そこで，本研究ではビジュアルプログラミングシステムの実行時の変化をオブジェクトが次々と書き換わって行くアニメーションで表す実行アニメーションの手法を提案する．

3.2　3次元ビジュアルプログラミングシステム Animation-3DPP

本研究では，3D-PP をベースとして，プログラムをより詳細に表現するための手法と実行アニメーションを追加した新しいシステム Animation-3DPP を実装した．プログラムを詳細に表現するための手法は後述する引数オブジェクトとオブジェクトの階層表示の手法である．ここでは実行アニメーションに関して述べるための準備として Animation-3DPP における GHC プログラムの視覚化の方法に関して述べる．

3.2.1　Animation-3DPP のプログラム表現

Animation-3DPP では 3D-PP と同様に GHC プログラムの要素を表した 3 次元オブジェクトを用いている．また，それらをエッジで結線して作成する 3 次元グラフを用いて GHC プログラムの視覚化を行う．

Animation-3DPP では GHC プログラムを視覚化する際に引数対応を明確にするために引数オブジェクトを用いる．また，同じヘッド (同じ述語名と同じ個数の引数) を持つ節を明確にすることと，節のガードとボディの定義の対応を明確にすることを目的として，オブジェクトの階層表示の手法を用いている．以下に引数オブジェクトとオブジェクトの階層表示の手法の解説を行う．

引数オブジェクト

GHC プログラムにおいて，ファンクタ，オペレータ，ゴールは引数にどのような値が与えられているかによって，意味や実行時の動作が違ってくる．よって，引数にどのような値が与えられているかは，プログラムやその実行に関して非常に重要な意味を持っている．また，節においては，引数の数が節を識別するために必要となる．Animation-3DPP では，引数を持つ要素を表すオブジェクトには，引数を表すオブジェクトを付加している．この引数を表すオブジェクトを引数オブジェクトと呼ぶ [19, 20]．引数オブジェクトは付加される位置でその引数が入力として扱われるのか，出力として扱われるのかを表している．また，互いに違った色を付けることによって，入力引数同士，出力引数同士の区別を行っている．図 3.2(a)の実行画面上のプログラムにおいても，ゴール，コンスに引数オブジェクトが付加されている．引数オブジェクトとアトム，ファンクタなどを結線することによって，引数とアトム，ファンクタとのユニフィケーション，すなわち，引数としてそれらを与えることを意味する．図 3.1は引数オブジェクトが付加されたオブジェクトの例である．

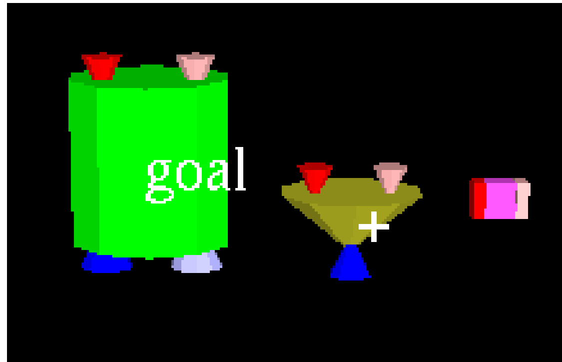
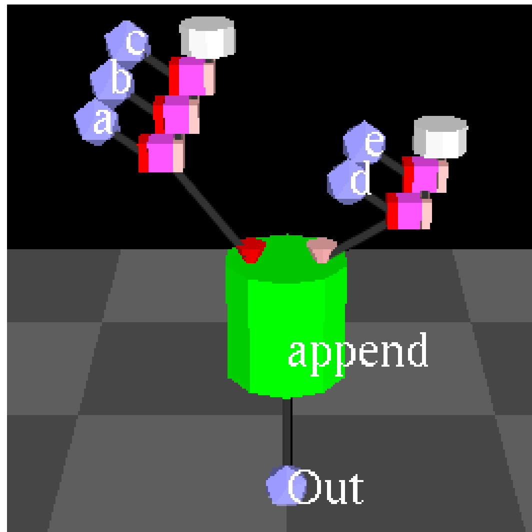


図 3.1: 引数オブジェクト

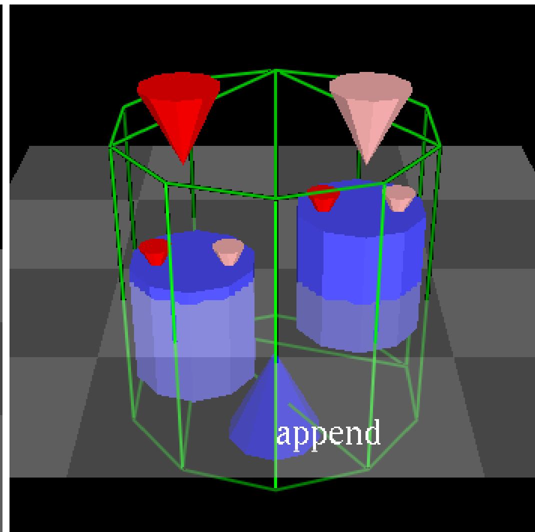
オブジェクトの階層表示

オブジェクトの階層表示は，オブジェクトの中にオブジェクトを入れて階層的にオブジェクトを配置する手法である [19, 20]．Animation-3DPP ではオブジェクトの階層表示を利用して，GHC の節を表現している．節のオブジェクト (図 3.2(b)の青色の円柱状のオブジェクト) は節のヘッドを表しており，その内部には節の定義であるガードと，ボディの定義するためのグラフを入れている．また，同じヘッドを持つ節のオブジェクトを，ゴールの中に入れている．プログラムを編集する際には，オブジェクトの階層表示を利用して，同じヘッドを持つ節を近い位置にまとめて配置することでプログラマの理解を助ける．

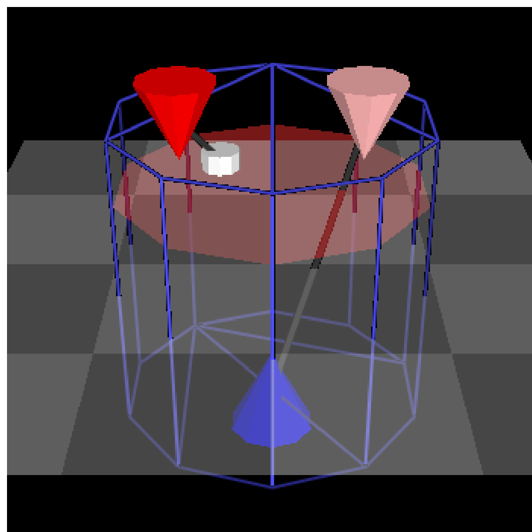
図 3.2はオブジェクトの階層表示の例である．図 3.2(b)は，図 3.2(a)にあるゴール append/3 を拡大したものである．ゴール append/3 を表すオブジェクトの中に，図 3.2(b)のように，節のオブジェクトが配置されている．図 3.2(c)は図 3.2(b)にある節のオブジェクトの内，左側の節を拡大したものである．また，図 3.2(d)は，右側の節を拡大したものである．節のオブジェクトの中に，図 3.2(c)，図 3.2(d)の様にガードとボディの定義するためのグラフが入っている．



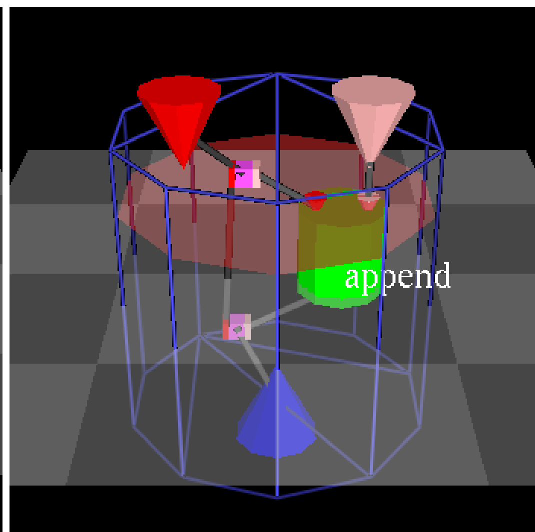
(a) ゴール



(b) 節の階層表示



(c) 節の定義の階層表示 1



(d) 節の定義の階層表示 2

図 3.2: オブジェクトの階層表示

GHC プログラムとの対応

図 3.3は，Animation-3DPP で用いられるプログラム要素のオブジェクトを並べた図である．図 3.3の 3 次元オブジェクトは左から，ゴール (緑色の円柱)，オペレータ (黄色の逆三角錐)，ファンクタ (白い円盤)，コンスセル (紫の立方体)，アトム (水色の球) を表している．また，エッジは GHC におけるユニフィケーションを表している．また，以下に GHC プログラムと，オブジェクトによるビジュアルな表現との対応を述べる．

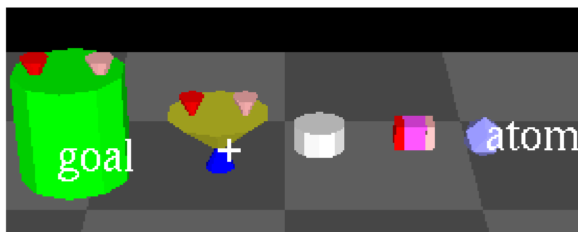


図 3.3: Animation-3DPP のオブジェクト

- 節
  Animation-3DPP では，節は青い円柱のオブジェクトで表される．節のオブジェクトは，ゴールのオブジェクトの内部にオブジェクトの階層表示の手法を用いて配置されている．また，節のヘッド，ガード，ボディは以下の様に表されている．
  - ヘッド
    ヘッドは，節のオブジェクトと，それに付加される引数オブジェクトによって表される．節のオブジェクトはゴールのオブジェクトの内部に配置されており，そのゴールに付けられているラベルが述語名を表している．また，付加されている引数オブジェクトが引数を表している．例えば，前出の図 3.2(b)の青い円柱のオブジェクトの一つ一つは，節 `append/3` のヘッド `append(L1,L2,O)` を表している．
  - ガード
    ガードの定義は，3 次元グラフで表され，節のオブジェクトの内部にオブジェクトの階層表示の手法を用いて配置される．節のオブジェクトの内部にある 3 次元グラフのうち，赤い円盤型のオブジェクトよりも上に配置されているものが，ガードを表すグラフである．赤い円盤のオブジェクトは GHC プログラムにおける “|” に相当する．例えば，前出の図 3.2(c)では，ガードにある，白い円盤状のオブジェクト (空リストを表す) は，入力引数オブジェクトの片方と結線されている．これは，結線されている入力引数が空リストである，という条件を表している．GHC プログラムでは，ガードにある，`L = []` という記述に相当する．
  - ボディ
    ボディの定義は，3 次元グラフで表され，節のオブジェクト内にオブジェクトの階層表示の手法を用いて配置される．節のオブジェクト内にある 3 次元グラフのう

ち，赤い円盤型のオブジェクトより下に配置されているものが，ボディを表すグラフである．例えば，前出の図 3.2(d)では，ボディには，再帰呼び出しを表すゴール append/3 とコンスが存在する．これは，リダクションの際に，置き換えるものはゴール append/3 とコンスであることを表している．GHC プログラムでは，節のボディに，O = [H|O2]，append(T,L2,O2) という記述があることに相当する．

- ゴール
  Animation-3DPP では，ゴールはラベルの付ついた緑色の円柱のオブジェクトで表される．また，引数の数に相当する引数オブジェクトが付けらている．ラベルは実行時に呼び出される述語名を表している．また，引数オブジェクトはゴールに与えられる引数を表している．例えば，前出の図 3.2(a)では，append/3 というゴールに対して，[a,b,c,d]，[e,f] という 2 つのリストが与えられている．Out という出力に結線されているのは，ゴールの出力を Out という出力ストリームに出力する，ということを表している．GHC プログラムでは，append([a,b,c,d],[e,f],Out) という記述に相当する．

- アトム
  Animation-3DPP では，アトムは水色の球のオブジェクトで表されている．オブジェクトにはアトムの値を表すラベルが付けられている．ラベルは，“a”, “atom1” 等の文字列または，1,2,100 等の数値が付けられる．

- ファンクタ，コンス
  ファンクタ，コンスは，紫色の立方体のオブジェクトで表される．また，コンスには，2 つの引数オブジェクトが付加されている．コンスの引数オブジェクトの内，第 1 引数 (濃い赤色の引数オブジェクト) は Head を表している．また，第 2 引数 (ピンク色の引数オブジェクト) は Tail を表している．これは，GHC のコンス [Head|Tail] と対応している．また，Animation-3DPP でも，コンスを組み合わせてリストを作ることができる．リストは図 3.4の様に，コンスとアトムを組み合わせて作成する．図 3.4はリスト [a,b,c,d,e] を表している例である．

- オペレータ
  Animation-3DPP では，オペレータは黄色の逆さ円錐のオブジェクトで表される．オペレータは 2 個のアトムの四則演算や比較を演算行うため，必ず 2 個の入力の引数オブジェクトを持っている．また，四則演算を表すオペレータには，演算結果を出力する引数オブジェクトが，オブジェクトの下方に付加される．オペレータの例は，図 3.5の様になる．図の例では，入力に数値アトム “20”，“30” が結線されており，$20 + 30$ という演算を表している．

図 3.4，3.6は 3 次元グラフの例であり，図 3.4はリスト [a,b,c,d,e] を，図 3.6は 2 つのリストの結合の処理を行うゴール append/3 に引数として，リスト [a,b,c] とリスト [d,e] と出力ストリーム Out を与えたプログラムである．

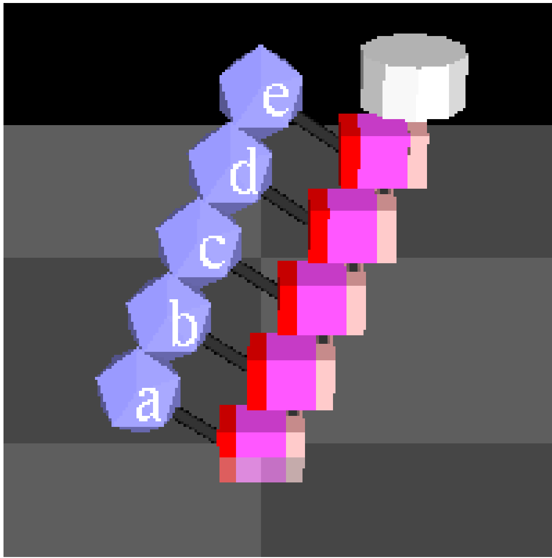


図 3.4: グラフの例 (リスト [a,b,c,d,e])

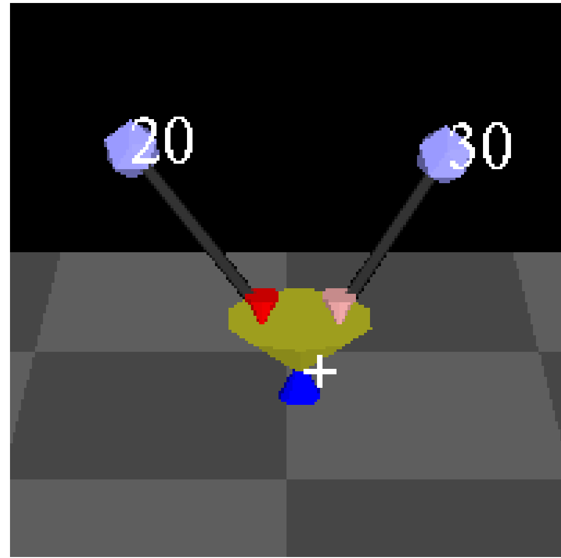


図 3.5: オペレータの例 (20 + 30)

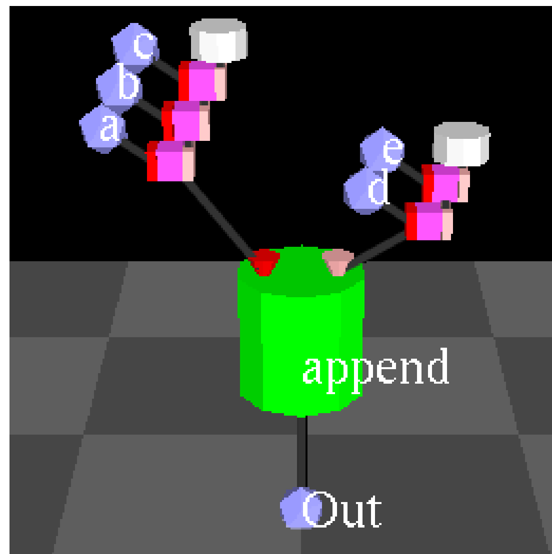


図 3.6: グラフの例 (append([a,b,c],[d,e],Out))

3.3 実行アニメーション

実行アニメーションでは，編集したプログラムの実行をその編集したプログラムで用いられたオブジェクト自身が形を変え，描き換わっていくアニメーションで表現する．また，ア

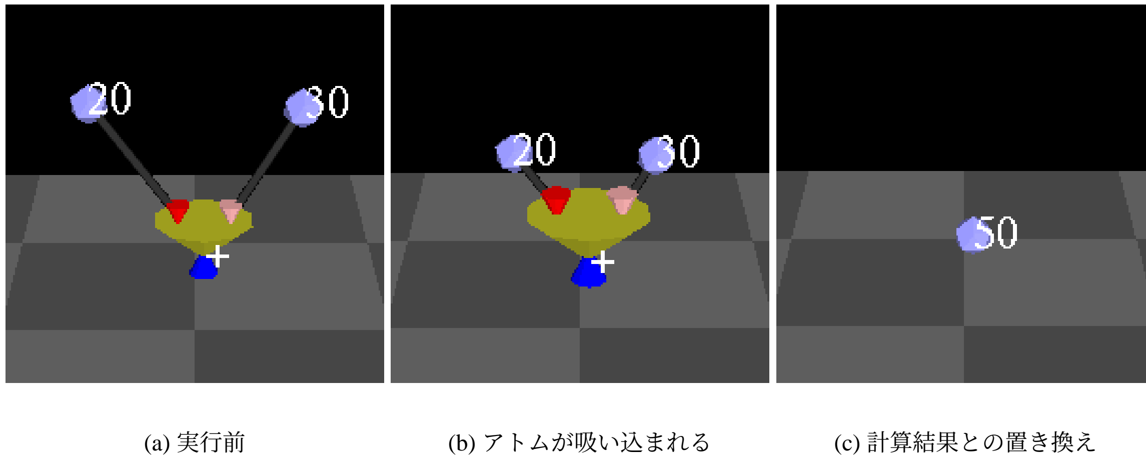
ニメーションを見る方向やアニメーションが動く速度をユーザが任意に変更することが出来る．このようにアニメーションを見る方向や動く速度を変える事を本研究ではアニメーションを操作すると呼ぶことにする．

アニメーションは観察する人が変化についていけなくならないように，オブジェクトの形や位置の変化は飛び飛びにではなく滑らかに変化するように行われる．例えば，図 3.7の加算演算のオペレータの例では加算のオペレータに 2 つの数値アトムが入力の引数として与えられている．このオペレータの実行は図 3.7のように，まず，オペレータの引数オブジェクトに向かって 2 つのアトムが滑らかに移動し，吸い込まれていくアニメーションが行われる．その後，2 つのアトムとオペレータのオブジェクトが計算結果のアトムのオブジェクトと置き換わる．

また，実行アニメーションでは，プログラム実行時に行われる実際の処理と同じ順にそれぞれのオブジェクトのアニメーションが行われる．

さらに，ユーザはアニメーションが行われている最中であってもパン，ズーム，視点回転といった視点位置の変更を行うことが出来る．実行アニメーションでは，実行を観察するユーザは好きなときにアニメーションを一時停止することが出来る．アニメーションが一時停止した状態からはアニメーションを再開すること以外にもオブジェクトの移動や，視点位置の変更といった操作を編集時と同様の操作で行うことが出来る．

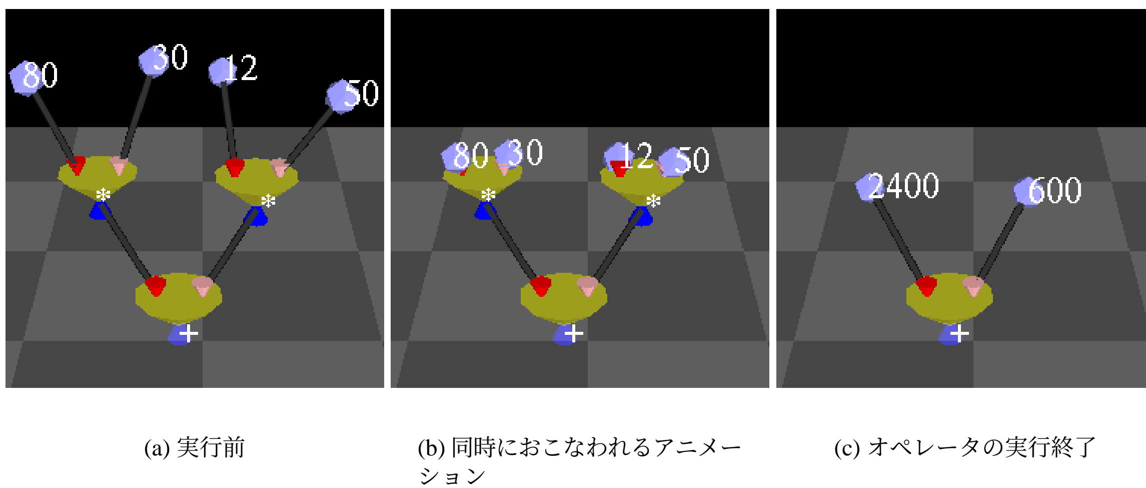
実行アニメーションでは，これから実行される処理を表しているオブジェクトやどの節が選ばれたかなどを点滅する，拡大する等のアニメーションを用いて強調して表示する．これによって，これからどの要素が実行され，アニメーションで表示されるのかを分かりやすくする．また，計算やゴールのリダクションを表す実行アニメーションを行った後にはオブジェクトの拡大，縮小などによってオブジェクトが移動する．そのため，オブジェクトが重なり合うなどして見えにくくなっている場合がある．これを見やすいような配置にするために，計算やリダクションのアニメーションを行った後にはオブジェクトの自動的な再配置を行っている．また，オブジェクトの自動的な再配置は実行アニメーションと同様にアニメーションを用いて行っている．さらに，実行アニメーションでは，ユーザが自分が観察しやすいように，オブジェクトが拡大する場合，どこまで拡大するのかといったことを調節することが出来る．また，計算や処理のアニメーションを行った後のレイアウトに用いるアルゴリズムを自由に変更することが出来るようになっている．



(a) 実行前　(b) アトムが吸い込まれる　(c) 計算結果との置き換え

図 3.7: オペレータの実行アニメーション

実行アニメーションでは，実行可能であるオブジェクトが複数存在する場合にはそれらのアニメーションを同時に行う．これによって並列に実行されるプログラムもアニメーションで表現することが可能である．図 3.8は複数のオペレータの計算を並列に行っている例である．図 3.8(a)では，画面上に存在する 2 つのオペレータ “`80* 30`”，“`12*50`” はどちらも実行が可能である．よって，2 つのオペレータの計算は同時に実行され，アニメーションも同時に行われる (図 3.8(b))．



(a) 実行前　(b) 同時におこなわれるアニメーション　(c) オペレータの実行終了

図 3.8: 並列に実行されるオペレータ

3.4 各オブジェクトの実行アニメーション

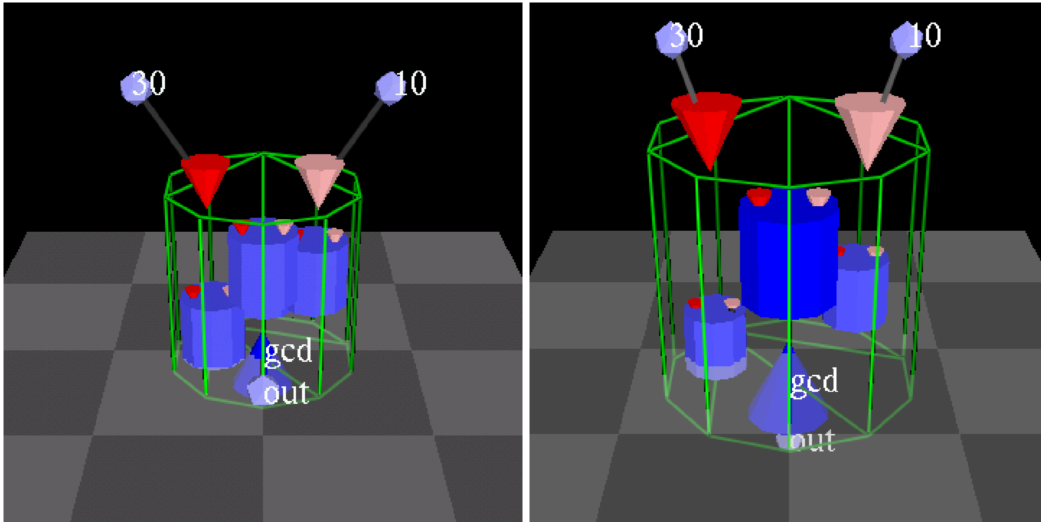
以下にゴールのリダクション，ファンクタのユニフィェーション，オペレータの実行アニメーションでどのようなアニメーションが行われるかを解説する．

3.4.1 実行アニメーション – ゴールのリダクション

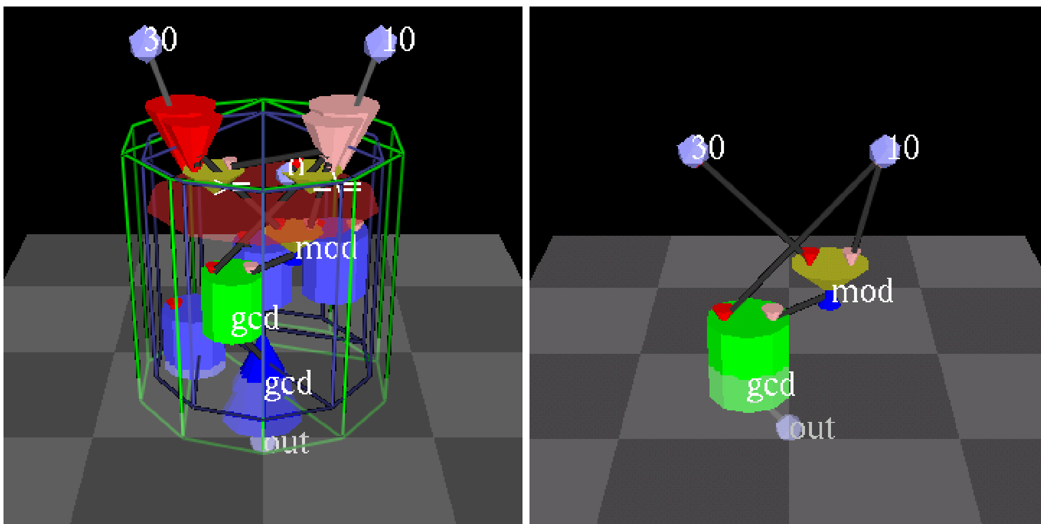
ゴールのリダクションの実行アニメーションは実行ゴールの強調，節の選択，ゴールと節の定義の置き換えの各アニメーションからなっている．実行ゴールの強調では，実行されるゴールを徐々に拡大するアニメーションを行う．拡大する際にゴールのオブジェクトはワイヤフレーム表示となる (図 3.9(a))．オブジェクトの階層表示によって，ゴールのオブジェクトの内部には節のオブジェクトが配置されている．よって，内部から節のオブジェクトが現れてくることになる．

節の選択では，条件に合う節が 1 つ，以下の様に選択される．ゴールの内部にある節のガード部分と入力引数をチェックし，どの節がガード部の定義する条件にマッチするのかを調べる．条件に合う節が見つかった場合はその節をゴールとの置き換えのために選択する．見つからなかった場合はエラーであるので実行を停止する．複数の節が条件を満たした場合，条件を満たした節の中から 1 つの節を無作為に選択する．また，選択された節のオブジェクトは点滅するアニメーションによって強調される．図 3.9(b)では，ゴールのオブジェクトの内部にある節のうち，図中で一番手前にある節が選択されている．一番手前にある節は点滅のアニメーションで強調されているため，図 3.9(b)では通常の節の色より濃い青色をしている．

ゴールと節の定義の置き換えでは，まず，図 3.9(c)の様に，選ばれた節のオブジェクトが徐々に拡大していくアニメーションが行われる．同時に，選ばれた節以外の節のオブジェクトは徐々に縮小されるアニメーションが行われ，画面上から消える．拡大されるときに節のオブジェクトはワイヤフレーム表示となる．従って，オブジェクトの階層表示によって節のオブジェクトの内部に配置されている節の定義のオブジェクトが現れてくる．選択された節のオブジェクトは，ゴールのオブジェクトと同じ大きさになるまで拡大される．節がゴールと同じ大きさになった時点で節の定義とゴールのオブジェクトの置き換えが行われる (図 3.9(d))．置き換えでは，内部のグラフをゴールに与えられている引数と結線する．節の引数オブジェクトに結線されているものはゴールの引数オブジェクトに繋ぐ．例えば，節の 1 番目の入力引数オブジェクトにボディのオブジェクトが結線されていた場合，ゴールの 1 番目の入力引数オブジェクトに結線されていたオブジェクトとそのボディのオブジェクトとが結線される．



(a) 実行ゴールの強調　　(b) 節の選択の強調



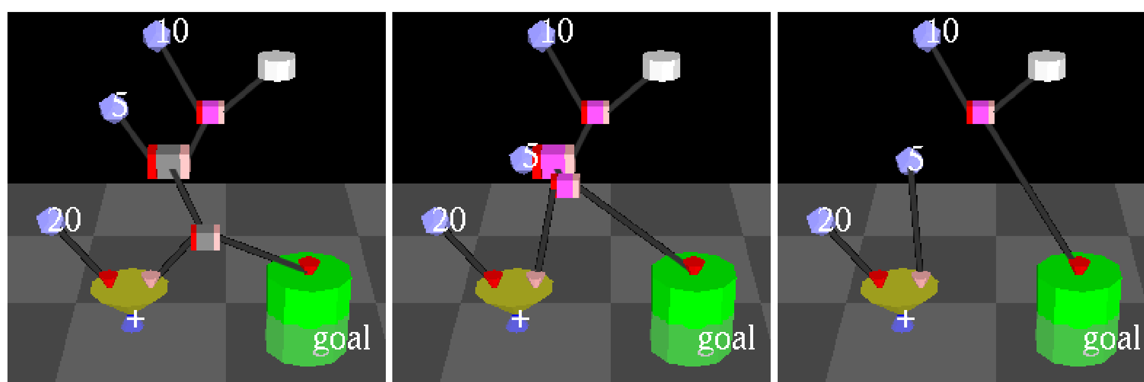
(c) ゴールと，節の定義の置き換え　　(d) リダクション終了

図 3.9: ゴールの実行アニメーション

3.4.2　実行アニメーション – ファンクタのユニフィケーション

ファンクタのユニファイを行うアニメーションは図 3.10の様になる．ユニファイされるファンクタ同士は必ず同じ数の引数を持っている．まず，図 3.10(a)の様に実行でユニファイされ

るファンクタが点滅のアニメーションによって強調される．その後，ファンクタのオブジェクト同士がゆっくりと近づいていき，全てのファンクタが同じ位置に重なる (図 3.10(b))．その後，ファンクタの引数に結線されているもの同士の結線が行われる．ファンクタの第 1 引数に結線されているものはユニファイされるファンクタの第 1 引数に結線されているものと結線される．また，第 2 引数以外でも第 1 引数と同様に引数に結線されているもの同士の結線が行われる (図 3.10(c))．図 3.10の例で言うと次の様になる．図 3.10(a)でユニファイされるファンクタ (灰色になっている 2 つのファンクタ) のうち，図中で上の位置にあるものをファンクタ A，下の位置にあるものをファンクタ B とする．A の第 1 引数にはアトム “5” が，第 2 引数には，今回はユニファイされないファンクタ C が結線されている．また，B の第 1 引数にはオペレータ “+” の入力引数オブジェクトが，第 2 引数にはゴール `goal/2` の入力引数オブジェクトがそれぞれ結線されている．ユニフィケーションの実行アニメーションの結果として A,B の第 1 引数に結されているもの同士，すなわちアトム “5” とオペレータ “+” の入力引数オブジェクトが結線される．また，A，B の第 2 引数に結されているもの同士，すなわち，ファンクタ C とゴール `goal/2` の入力引数オブジェクトが結線される (図 3.10(c)).



(a) ユニファイするファンクタを強調　(b) ファンクタ同士が重なるように移動　(c) ユニファイが完了

図 3.10: ファンクタのユニファイの実行アニメーション

3.4.3 実行アニメーション – オペレータ

前出の図 3.7はオペレータの実行アニメーションの例である．オペレータの実行の場合には以下のようなアニメーションで表される．まず，引数として与えられている 2 つのアトムのオブジェクトがオペレータのオブジェクトに吸い込まれる (図 3.7(b))．そのあと，重なった 2 つのアトムとオペレータのオブジェクトが消え，オペレータのオブジェクトと計算結果の値を持つアトムのオブジェクトと置き換わる (図 3.7(c))．実行結果として新しく現れたアトムはオペレータの出力引数オブジェクトと結線されていたオブジェクトと結線される．

3.5 実装

Animation-3DPP は Windows XP 上で実装を行った．プログラミング言語は C++言語を用いており，ステップ数は約 50,000 である．また，3 次元図形表示およびマウス，キー操作の処理には OpenGL[21] と GLU，GLUT[22] ライブラリを使用した．

実行アニメーションでは，GLUT ライブラリの機能であるアイドル・コールバック機能を用いてアニメーションを行っている．アイドル・コールバック機能ではウィンドウシステム・イベントが何もないアイドル時に登録された関数が呼び出される．

実行アニメーションの実装では，アイドル・コールバックに登録した関数内で Animation-3DPP が起動してからの時間を計測している．そして，一定時間が経過するごとにオブジェクトの移動，作成，削除などの描画処理を行っている．一定時間経過ごとに描画処理を行うため，Animation-3DPP 上では滑らかにアニメーションをしているように見える．また，実行アニメーションでは GoalAnimation, OperatorAnimation, UnificationAnimation といった各プログラム要素に応じたアニメーションのためのクラスを定義している．ゴールの実行アニメーションが行われるときは GoalAniamtion クラスのインスタンスが作成され，そのインスタンスがアニメーションを行う．同様に，オペレータの実行アニメーションのときは OperatorAnimation クラスの，ファンクタのユニフィケーションの実行アニメーションのときには UnificationAnimation クラスのインスタンスがそれぞれ作成され，アニメーションが行われる．

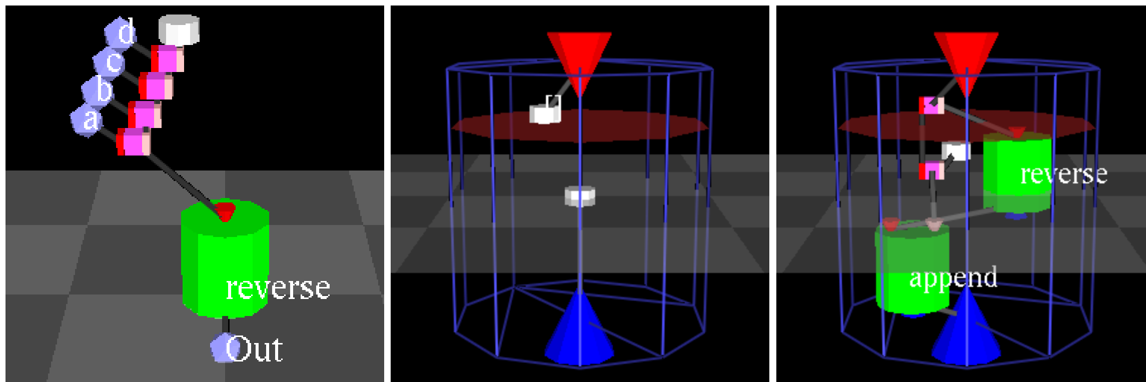
3.6 実行アニメーションの例

以下に Animation-3DPP 上で作成したプログラムの実行を実行アニメーションの手法を用いて表示する例を示す．この例で実行されるプログラムは図 2.1 で示されているプログラムである．図 2.1の `reverse/2` は１つのリストを入力として受け取り，そのリストを逆順にしたリストを出力する．`append/3` は 2 つのリストを入力として受け取り，第 1 引数に与えられたリストの後ろに第 2 引数のリストを結合したものを第 3 引数に出力する．図 2.1で示されているプログラムは Animation-3DPP では図 3.11(a)のように表わされる．

`reverse/2` は 2 つの節からなり，図 3.11(b)は図 2.1にある 1 番目の節の定義に対応する．図 3.11(c)は図 2.1にある 2 番目の節の定義に対応する．

節 1 のガードには入力に与えられたものが空リストであるという条件 (`L=[]`) が定義されている．ボディには出力引数 `O` に空リストを出力するという処理が定義されている．Animation-3DPP では，図 3.11(b)の様に，カードに入力引数オブジェクトが空リストを表す白い円盤上のオブジェクトと結線されている．ボディの定義では，出力引数オブジェクトと空リストを表すオブジェクトと結線されており，空リストを第 3 引数に出力することを表している．

節 2 のガードには入力に与えられたものがコンスの組み合わせで作られるリストであるという条件 (`L=[H|T]`) が定義されている．また，同時に H，T という変数を用いることでリストの先頭の要素 (Head) と先頭の要素を除いた要素からなるリスト (Tail) とを分けている．ボディには，Tail を入力引数としたゴール `reverse/2` の再帰呼び出しがある．また `reverse/2`



(a) ゴール reverse/2 とリスト　(b) reverse/2 の節 1　(c) reverse/2 の節 2

図 3.11: Animation-3DPP での reverse/2

の再帰呼び出しから得られる出力 O2 と Head のみを要素として持つリスト ([H]) とをゴール append/3 で結合している．節 2 は Animation-3DPP では図 3.11(c)の様に表される．カードの条件は入力引数オブジェクトがガードにあるコンスと結線されている．これによって，入力がコンスの組み合わせで作られるリストであるという条件を表している．入力に，ガードにあるコンスと同じ形のものが与えられたときに条件を満たしたことになる．同時に，ガードにあるコンスは，入力に与えられたリストの Head と Tail とをコンスに付加されている引数オブジェクトによって分けている．ボディには Head のみを要素として持つリスト ([H]) を作るためのコンスとゴール append/3 とゴール reverse/2 を表すオブジェクトがある．append/3 の第 1 引数には Head のみを要素として持つリストが結線されている．第 2 引数には reverse/2 の出力が結線されている．これらの結線でこの 2 つのリストを結合することを表している．また，reverse/2 の入力には Tail が結線されており，Tail が reverse/2 の入力として与えられていることを表している．

Animation-3DPP 上で reverse/2 のプログラムを実行した時の実行アニメーションは図 3.12，図 3.13，図 3.14のスクリーンショットの様になる．

1. 実行アニメーションの開始
   例の場合では，ゴール reverse/2 にリスト [a,b,c,d] を入力として与えている．また，reverse/2 の結果は，reverse/2 の出力の引数オブジェクトと結線されている Out と結線される．ここで，実行アニメーションは画面上に存在するプログラム要素のオブジェクトをチェックし，どのプログラム要素が実行可能であるかを調べる．

2. ゴール reverse/2 の実行
   図 3.11(a)には，入力の引数にリスト [a,b,c,d] という具体的な値が結線されているゴール reverse/2 が存在する．また，節のガードの条件を満たしているためリダクションが可能である．よって，ゴール reverse/2 のリダクションを表すアニメーショ

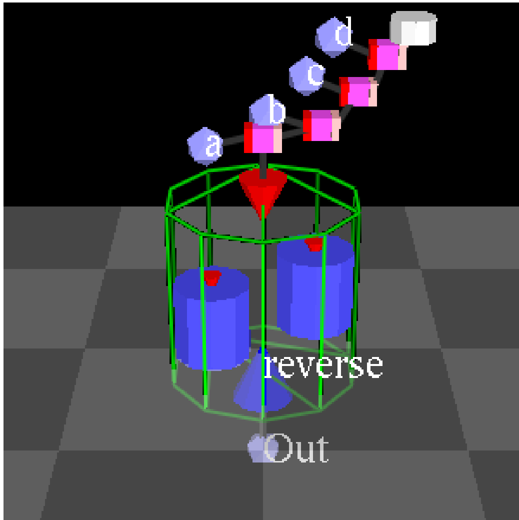
ンが行われる．実際のアニメーションでは，ゴール reverse/2 のオブジェクトが条件にマッチする節の定義のグラフと置き換えられる．まず，ゴール reverse/2 のオブジェクトが拡大され，現在はこのゴールのリダクションが行われていることを示す (図 3.12(a))．その後，引数と各節のガードの条件とのチェックが行われる．例の場合では，reverse/2 の節 2(図 2.1 にある 2 番目の節 reverse/2) には入力がコンスで作られるリストであるという条件が存在する．ゴール reverse/2 の入力に結線されているオブジェクトはリストであり，また空リストではないので節 2 の条件にマッチする．よって，節 2 が選択される (図 3.12(c))．選択された節がゆっくりと拡大していき，ゴール reverse/2 のオブジェクトと置き換わり，図 3.12(d)のような新しいグラフが作成される．

3. ファンクタのユニフィケーション
新しく出来グラフには図 3.13(a)(図中で灰色になっている 2 個のファンクタ) のように，ユニファイ可能なファンクタが存在する．それらはリスト [a,b,c,d] の最初の要素 a を引数に持つ [a|[b,c,d]] とゴール reverse/2 を展開したときにガードにあった [H|T] である．ここではこれら 2 個のファンクタをユニファイするアニメーションが行われる．ユニファイされる 2 個のファンクタが点滅のアニメーションで強調されたあと，2 個のファンクタが移動して重なる．その後，図 3.13(b)のようにそれぞれのファンクタの引数に結線されていたもの同士がエッジで結線される．

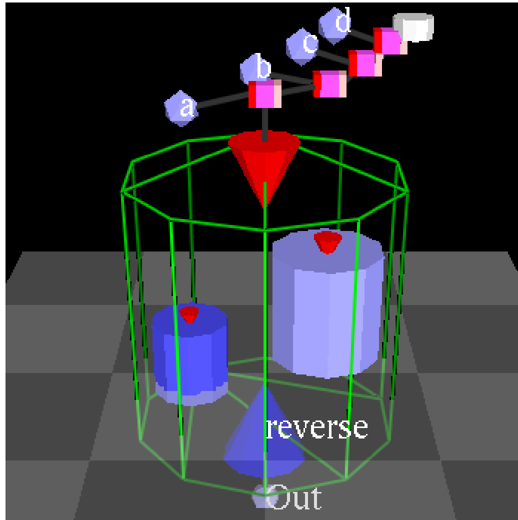
4. 新しく構築されたグラフ中のゴールの実行
ファンクタのユニファイの後，グラフにある reverse/2 が入力の引数が決定したため，ガードの条件を満たし，リダクション可能となっている．よって，この reverse/2 の実行アニメーションが 3.12(a) – 3.12(d)と同様のアニメーションで表される．置き換えで現れてくるゴール reverse/2 の実行と，ファンクタのユニファイを最初の reverse/2 の入力に与えられていたリストにあったコンスの数だけ繰り返すと図 3.14(b)のようなグラフとなる．ここでは，ゴール reverse/2 は存在せず，ゴール append/3 が 4 個存在し，そのうちの 1 個 (図の中央のゴール) が入力の引数が 2 個とも決定している状態である．よって，この append/3 の実行アニメーションが行われる．

   この後，全ての appned/3 がそれがリダクション可能になったときにアニメーションが順々に行われていく．

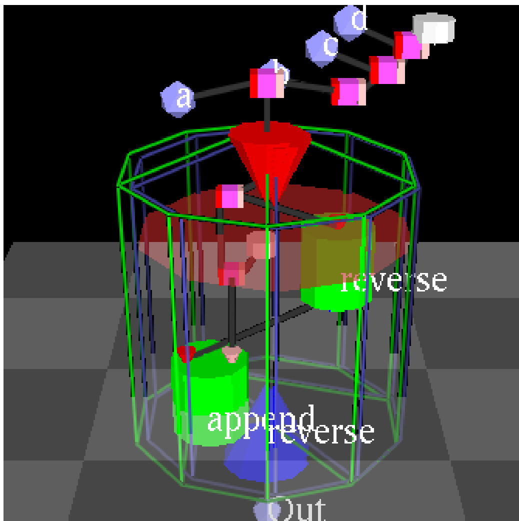
5. 実行の終了
こうして画面上から全てのリダクション可能なゴール，ユニファイ可能なファンクタ，計算可能なオペレータがなくなるまで実行が続けられ，アニメーションで表示される．図 3.11(a)のプログラムの最終的な実行結果は図 3.14(d) となる．



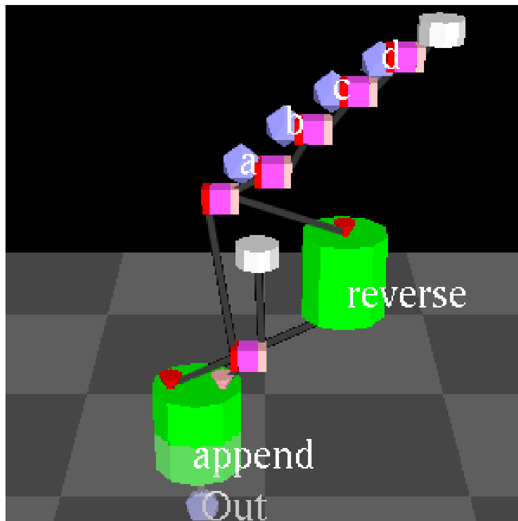
(a) リダクションされるゴール reverse/2 の強調



(b) 選択された節 2 の強調

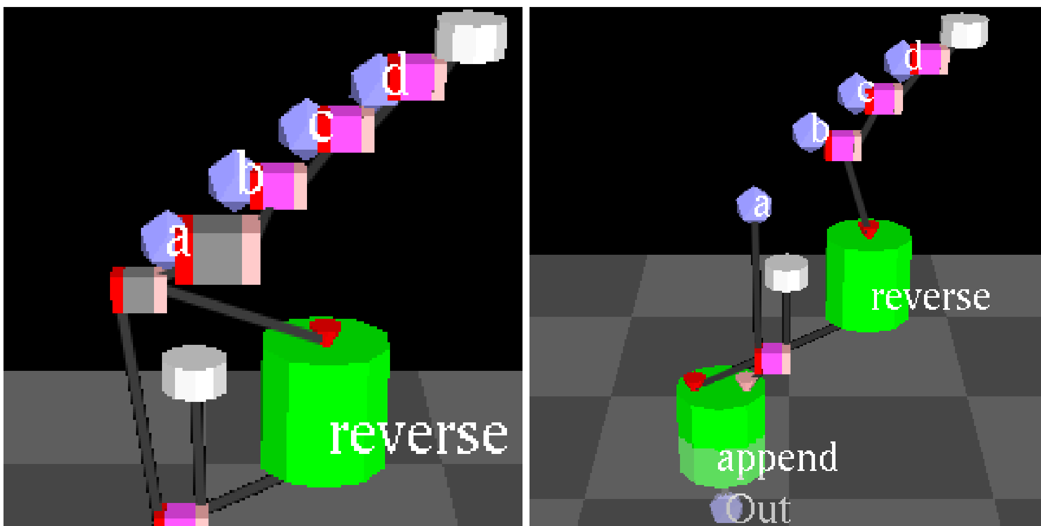


(c) 定義の置き換え



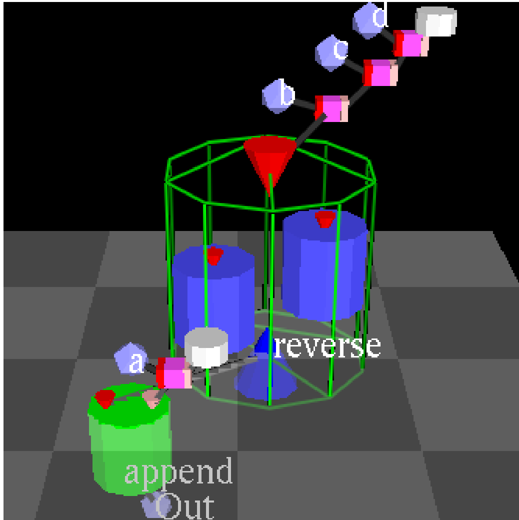
(d) ゴール reverse/2 のリダクション後

図 3.12: 実行アニメーションの例 – ゴール reverse/2 のリダクション

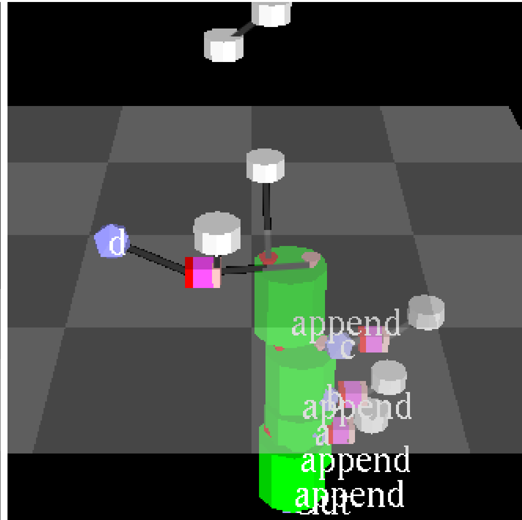


(a) ユニファイされるファンクタの強調　　(b) ユニフィケーションの完了

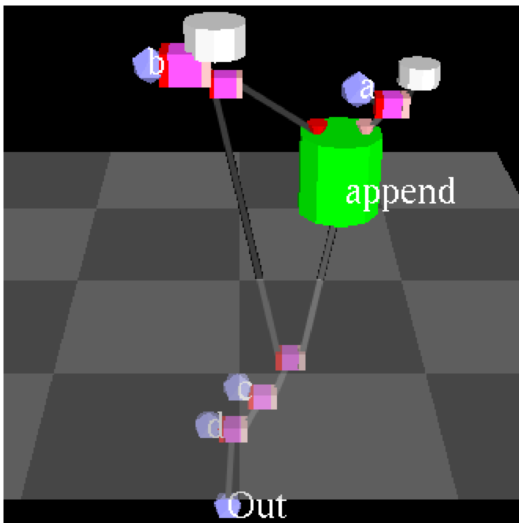
図 3.13: 実行アニメーションの例 – ファンクタのユニフィケーション



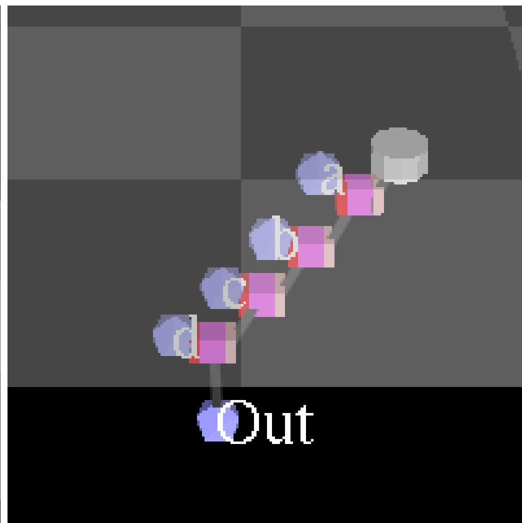
(a) ゴール reverse/2 のリダクション



(b) append/3 とリストのみからなるグラフ



(c)



(d)

図 3.14: 実行アニメーションの例 – 図 3.13以降の様子

第4章　3次元ビジュアルプログラミングにおけるデバッグ手法

4.1　バグ

プログラムの実行において，ユーザが記述したプログラムがユーザの意図しない動作を行うことがある．このようなプログラムにはバグと呼ばれる誤りが含まれている．ここでは，実行の最後に出力される実行結果がユーザが意図したものとは違ったものになるということを引き起こすバグについて考える．この種類のバグは間違った答えの出力 (incorrect answer) と呼ばれている [23]．論理型言語において，間違った答えの出力は述語の定義に誤りがあることによって起こる．述語の定義の誤りのため，実行の過程で本来実行されるはずのない計算が実行されたり，述語に与える引数が間違っていたりすることによってゴールから間違った結果が出力される．

4.2　デバッグのため機能と手法

ビジュアルプログラミングシステム上でデバッグを行うために以下のような機能を Animation-3DPP 上に実装した．

- 実行の巻き戻し
- プログラムの部分実行
- プログラムのステップ実行
- 実行の一時停止中のデータの書き換え
- 色付きオブジェクト

4.2.1　実行の巻き戻し

実行の巻き戻しは，プログラム実行の一時停止中，または実行の終了後にアニメーションをビデオの逆再生のように巻き戻す機能である．実行の巻き戻しでは，それまでに行われたアニメーションの逆回しとなるアニメーションが行われる．その結果，オブジェクトの位置の変更，オブジェクトの大きさの変更，オブジェクトの置き換えなどの実行アニメーション

によって行われた全てのオブジェクトの変化が巻き戻され，もとの状態に戻っていく．全てのオブジェクトが元の状態に戻るため，プログラムの実行状態もそれにあわせて戻っていくことになる．また，巻き戻しのアニメーション中にも実行アニメーションと同じように一時停止することができ，実行の任意の時点に巻き戻した後，その時点から実行を再開することができる．この機能では，アニメーションを巻き戻すことにより実行を任意の時点まで巻き戻して観察を行うことが可能となる．また，巻き戻しを実行時とちょうど逆のアニメーションで表しているため，ユーザは巻き戻っていく実行状態を簡単に追うことが出来る．

4.2.2　プログラムの部分実行，ステップ実行

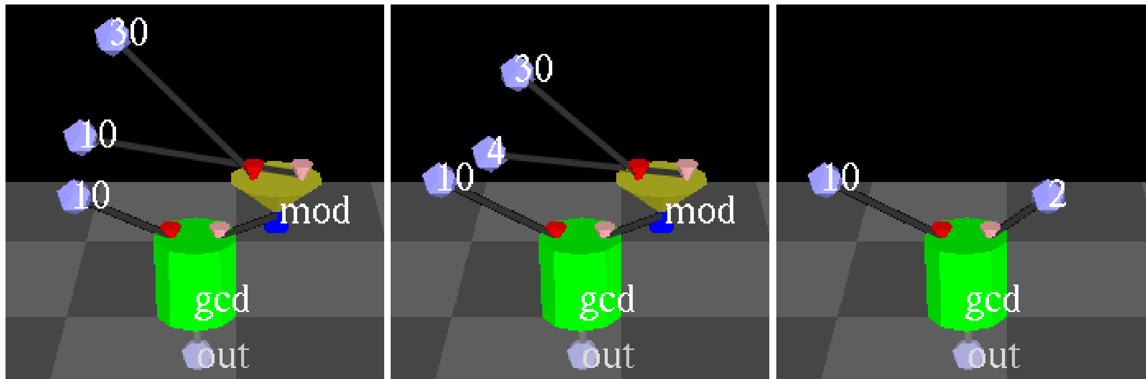
実行アニメーションでは，リダクション可能なゴール，ユニファイ可能なファンクタ，計算可能なオペレータが画面上に同時に複数存在する場合にはそれらのアニメーションを同時に行う．プログラムの部分実行は，このように同時実行可能な処理が複数存在する場合に，それらの中からユーザが選択した処理だけを実行する機能である．同時にたくさんの処理が行われると，データなどの変化が多くなる．また，アニメーションで表しているオブジェクトの変化も増える．プログラムの部分実行ではこれを観察しやすくするために，ユーザが注目したいプログラム要素だけを指定することで，指定した要素の実行だけを行うことが出来る．

プログラムのステップ実行では，ゴールのリダクション，ファンクタのユニファイ，オペレータによる計算といった処理を単位として，処理を 1 つずつ行っていくことが出来る機能である．ステップ実行はユーザが任意にオン，オフを選択することができる．ステップ実行がオンの場合には，一つの要素の処理，たとえばゴールのリダクションが終わったときに自動的に実行アニメーションが一時停止の状態になる．

4.2.3　実行一時停止中の書き換え

実行一時停止中の書き換えは実行を一時停止しているときにアトムの値や，エッジ結線などの変更を可能にする機能である．書き換えた後に実行を再開した場合，書き換えた後のグラフを用いて実行が進められ，計算や節の選択などが行われる (図 4.1)．ただし，ここでの変更は一時的なものであり節の定義には影響を及ぼさない．例えば，ゴールがリダクションされた直後にゴールと置き換えられたグラフに対して変更を加た場合でも，その変更はその場限りのものである．今後，今回のゴールと同名のゴールが現れ，同じ節が選択されたとしても変更が加えられる前の定義と置き換えられる．

この機能を用いることによって，ゴールに引数に別の値が与えられていたらどのような処理が行われたのかということをテストすることが可能となる．また，書き換えたアトムの値やエッジ結線などはアニメーションの巻き戻しを使うと元に戻る．よって，一時停止中に変更を加えても元の定義に戻すことができ，以前の実行の過程が変化することはない．



(a) 書き換え前　(b) 書き換え後　(c) 書き換え後に実行

図 4.1: 実行停止中の値の書き換え

4.2.4　色付きオブジェクト

色付きオブジェクトはオブジェクトに色で印をつけることによって，そのオブジェクトを他のオブジェクトと区別する機能である．また，そのオブジェクトが実行においてどのような影響を与えるのかを見られるようにするための機能である．色付きオブジェクトには，アトムの色付きオブジェクトと，節の色付きオブジェクトがある．

アトムの色付きオブジェクト

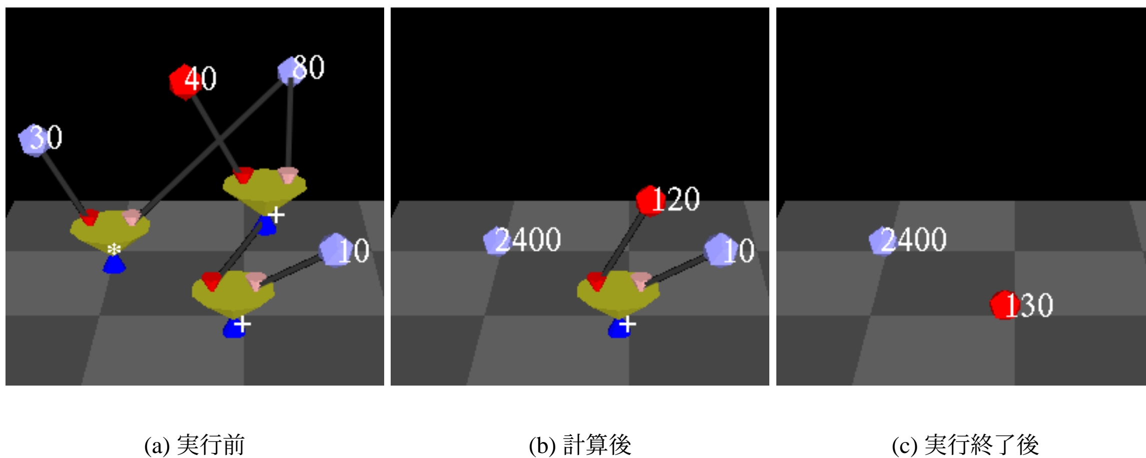
アトムの色付きオブジェクトでは，画面上にあるアトムに対して通常のアトムの色とは違う色を付けることが出来る．色を付けられたアトムが四則演算の引数に用いられた場合，計算が実行された後でも結果として出力されるアトムには同系統の色が自動的に付けられる (図 4.2)．これによって，色を付けられたアトムの計算結果を表すオブジェクトがどれであるかを示す．プログラムの実行が進んでも色を付けられているアトムの実行結果には全て色が付けられ，ユーザが最初に色を付けたデータの影響を受けていることを示す．また，データに色を付けて実行の巻き戻しを用いた場合，入力のデータにも同系統の色が自動的に付けられる．これを利用して，間違いのあった答えに色を付けてから実行の巻き戻し を用いることによって，関連のあるデータを絞り込むことが出来るようになる．

図 4.2はアトムの色付きオブジェクトの例である．図 4.2(a)は加算を表すオペレータの引数として与えられているアトム “40” に赤い色を付けた例である．図 4.2(b)は色を付けた状態での実行の図を示している．“40” と “80” の加算の実行後に出力されるアトム “120” にも，図 4.2(a)で付けた色と同じ色が付けられているのが分かる．色が付けられていることによって，このアトム “120” は図 4.2(a)で色を付けたアトム “40” の影響を受けていることが分かる．図

4.2(c)は，このプログラムの実行が終了した状態の図である．アトム “130” に赤い色が付けられており，このアトム “130” も アトム “40” の影響を受けていることを表している．

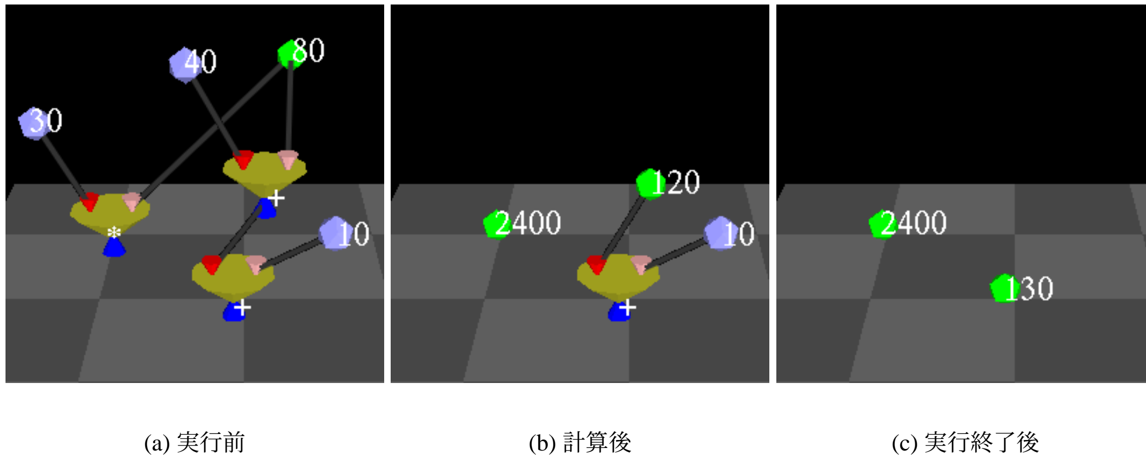
図 4.3は図 4.2と同じプログラムで色を付けるアトムを変えた時の実行結果を示している．ここではアトム “80” に緑色を付けている．アトム “80” は出力される 2 つの実行結果に影響を与えているアトムである．よって，図 4.3(c)の様に，2 つのアトム両方にアトム “80” に付けた色と同じ緑色が付けられる．

図 4.4色付きオブジェクトを用いた状態で実行を巻き戻した場合の例である．ここでは図 4.4(a)の様に実行結果のアトム “130” に青い色を付けた．これを実行の巻き戻しで巻き戻すと，図 4.4(b)，4.4(c) の様に，色を付け実行結果を出力したオペレータの 2 つの引数にも同じ色が付けられる．これによって，色を付けた実行結果 (アトム “130”) に影響を与えたアトムは “40”，“80”，“10” の 3 つであることが図 4.4(c)のアトムの色から判断することが出来る．



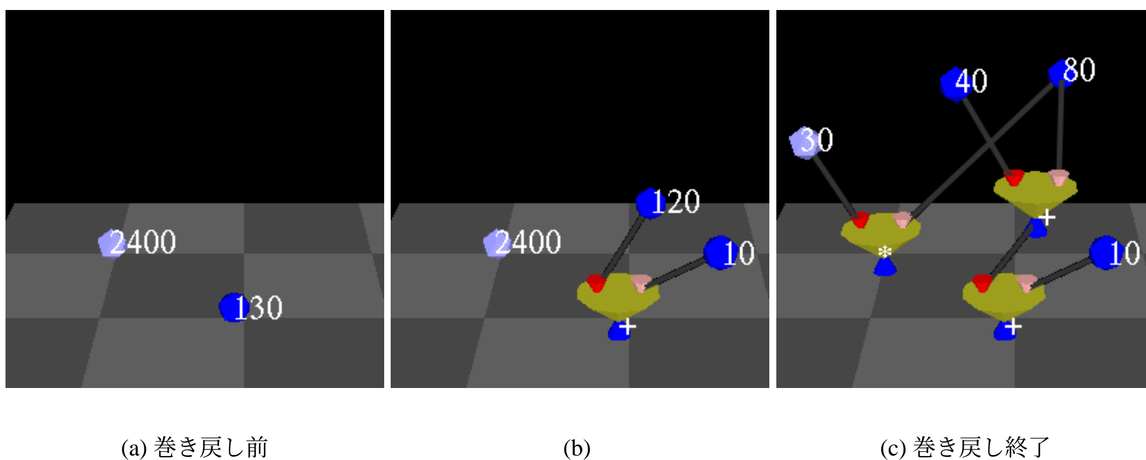
(a) 実行前　(b) 計算後　(c) 実行終了後

図 4.2: 実行時の色付きオブジェクトの例 1



(a) 実行前　　(b) 計算後　　(c) 実行終了後

図 4.3: 実行時の色付きオブジェクトの例 2



(a) 巻き戻し前　　(b)　　(c) 巻き戻し終了

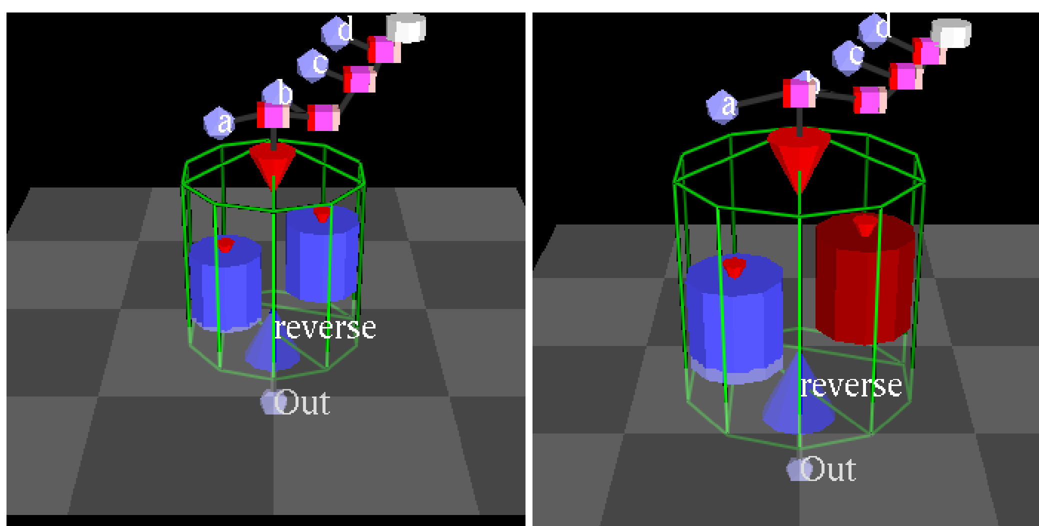
図 4.4: 実行の巻き戻し時の色付きオブジェクト

節の色付きオブジェクト

節の色付きオブジェクトでは，ゴールのリダクションの際に選択された節に色を付けることが出来る．節の色付きオブジェクトは実行を観察するときにユーザが意図した意図した正しい動作を行ったゴールに対して用いる．ゴールのリダクションにおいてゴールが正しい動作を行った場合，一旦実行を巻き戻し，正しい動作を行ったゴールを指定する．指定されたゴールでは，リダクションの際に選択された節の定義に対して色が付けられる．ある節の定義に色を付けた場合，その節の定義が実行の過程に現れる時には，節のオブジェクトに通常

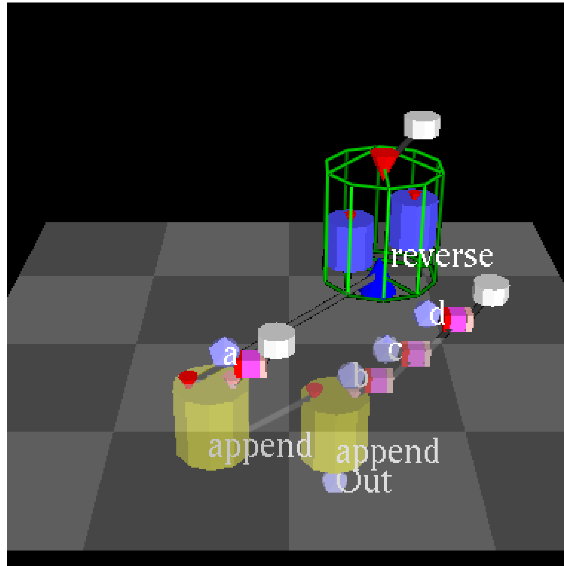
の節のオブジェクトの色 (青色) とは違った色が付けられている．すなわち，節の定義に色を付けた後，同じヘッドを持つ節が実行の他の時点で現れた場合でも，既に色を付けられた節には同じ色が付けられている．正しい動作を行ったときに選択されていた節の定義に通常の節のオブジェクトの色とは違った色を付けていくことで，バグのないと思われる節とそうでない節を分けていく．

また，節の色付きオブジェクトでは全ての節にその動作が正しかったとして色が付けられた場合，ゴールには通常のゴールのオブジェクトとは違った色 (濃い黄色) が自動的に付けられる (図 4.5)．例えば，図 2.1では節 append/3 は 2 個存在するが，色付きオブジェクトによってこの 2 個の節を表しているオブジェクトの両方に色が付けられた場合，今後画面上に現れるゴール append/3 のオブジェクトには図 4.6のように濃い黄色が自動的に付けられる．これによって，ユーザはどのゴールの動作をチェックしたのかチェックしていないのかを知ることが出来る．このように正しい動作を行った節に色を付けながらゴールのリダクションをチェックしていくことによってバグが含まれている可能性がある節を絞り込んでいく．



(a) (b)

図 4.5: 節の色付きオブジェクト



(a)

図 4.6: 自動的に色が付けられたゴール (append/3)

4.3 デバッグ手法

ここでは，間違った答えに対するデバッグ手法の手順について解説を行う．手順は以下の様になっている．

1. プログラムを実行させて結果を表示させる．
   まず，実行アニメーションを用いてプログラムの実行を行い，最終的な結果がユーザの意図したものであるかどうを確かめる．

2. 実行の巻き戻し，再度実行を行ってゴールの動作のチェックを行う．
   実行の結果がユーザの意図したものと違った場合，どこかにバグがあるのでそれを調査する．間違った答えがアトムで表されているならそのアトムに対してアトムの色付きオブジェクトを用いて色を付け，間違った答えであることを表す．また，以下のような手順でゴールの動作のチェックを行う．

   (a) 実行をゴールのリダクション前まで巻き戻し，再度実行することによってゴールがリダクションされたことによる結果が正しいかどうかをチェックする．

   (b) 実行結果が正しければ節の色付きオブジェクトを利用して節の定義に色を付ける．

(c) まだ色が付けられていないゴールに関して動作をチェックしていく．

2aでは，実行の巻き戻しを用いて，実行をゴールのリダクションが行われるよりも前に巻き戻し，その時点から実行を再開することによってそのゴールがユーザが意図したような動作を行っているかのチェックを行う．

2bでは，節の色付きオブジェクトを用いて，ユーザが意図した通りの動作をした節に対してチェック済みであることを示す色を付ける．一度色を付けられた節は実行のいつの段階でその節が現れたとしても同じ色が付けられているので，ユーザはこれからチェックするべき節と既にチェックされて動作が確認された節とを見分けることが出来る．

その後，2cの様に，まだ色が付けられていないゴールに対して動作のチェックを行う．節の色付きオブジェクトでは，全ての節に色が付けられたゴールには通常のゴールのオブジェクトとは違った色が自動的に付けられる．これによって，これから動作をチェックすべきゴールに関しても絞りこんでいくことができ，ユーザが動作をチェックする範囲を狭める助けとなる．

3. バグのあった節の定義を修正したあと，実行を行い，結果が意図したものとなるかどうかを確かめる．
   2で，バグのある節が特定できた場合，その節の定義を修正する．どのように修正すればよいかは実行アニメーションで一時停止中の時にアトムの値や，エッジ結線のされ方などを変え，試しに実行してみることが手助けとなる．

4.4 デバッグの例

図 4.7はバグを含むプログラムの例である．プログラムに記述されている内容は図 2.1と似ているが，一部にコードの違いがあり，図 4.7の節 `wrongrev/2` は図 2.1の `reverse/2` とは違った動作をし，違った結果を出力する．具体的には，`reverse/2` では第 1 引数として与えられたリスト `[a,b,c,d]` の要素が逆順となったリスト `[d,c,b,a]` が出力されるが，`wrongrev/2` では第 1 引数として与えられたリストと全く同じリストが出力されてしまう．以下に図 4.7に存在するバグを実行アニメーションとデバッグのための追加機能を用いてどの節にあるのかを示す例を以下に述べる．ユーザは，`wrongrev/2` は与えたリストを逆順にするゴール，`append/3` は 2 つのリストの結合を行うゴールであると考えているものとする．

```
main              :-  wrongrev([a,b,c,d],Out),
                      io:outstream([print(Out),nl]).

wrongrev(L,O)     :- L = []    |O = [].
wrongrev(L,O)     :- L = [H|T] |append([H],O2,O),wrongrev(T,O2).

append(L1,L2,Out):- L1 = []    |Out = L2.
append(L1,L2,Out):- L1 = [H|T]|Out = [H|O2], append(T,L2,O2).
```

図 4.7: バグのある GHC プログラムの例

1. 間違った答えの出力
   まず，図 4.7を視覚化した図 4.8(a)を実行する．結果として，図 4.8(b)のようなグラフが出力される．この出力はリスト [a,b,c,d] であり，ユーザが意図していた結果である [d,c,b,a] と違う．この時点でユーザは結果が意図した結果と違うことに気づく．

2. 巻き戻し，再実行による動作の確認
   次に，実行の巻き戻しを行い，どこの時点で間違いが起こっているのか，どのゴールが間違った出力をしたのかを特定をする．それは，実行の巻き戻しと巻き戻した後の再度実行を観察することで行う．

   (a) append/3 の動作をチェックをする例
   まず，実行の巻き戻しで図 4.8(b)の状態から巻き戻して一番初めに現れるゴール append/3 の動作をチェックをする例を示す．append/3 の展開前は図 4.8(b)であり，これを巻き戻したものは図 4.9(b)となる．append/3 はリストの結合を行うゴールであるので，空リスト ([]) とリスト [b,c,d] の出力結果は [b,c,d] となるはずである．意図した結果では，これが出力に結び付けられているリスト [a] と結線される．図 4.8(b)では，まさに意図した通りの結果が出ている．よって今回の append/3 は正しい動作をしている．ここで，節の色付きオブジェクトを用いて今回のリダクション際に選択された節に対して濃い赤い色を付ける．この色を付けることによって，今回チェックした節はチェック済みであることを示し，まだ動作のチェックを行っていない他の節と区別が出来るようにする．

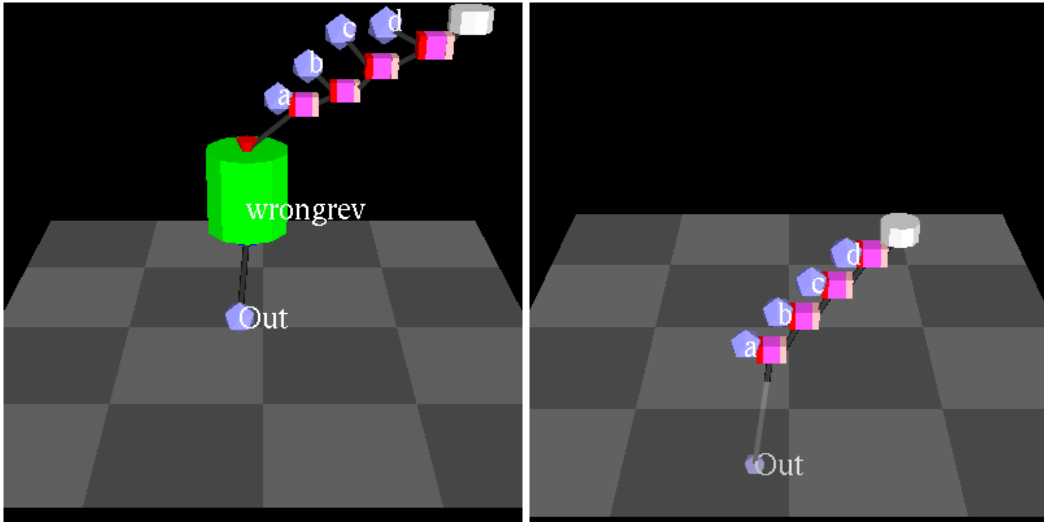
   上記の append/3 と同様に正しい出力をした場合には逐次色を付けていく．このようにすることによって正しい動作をした節とまだチェックを行っていない節を分けていく．また，節の色付きオブジェクトでは，全ての節に動作が正しかったとして色が付けられたゴールには他のゴールとは違った色 (濃い黄色) が自動的に付けられる．よって，巻き戻し，実行を繰り返している中で，色が付けられた節が展開される時や色が付けられたゴールのリダクションが行われるときには正しい動作が行われていると考えられる．

   (b) ゴール wrongrev/2 の動作のチェックの例 1
   図 4.10(a)では，ゴール wrongrev/2 の動作のチェックをしている．このときの

wrongrev/2 は入力として空リストが与えられている．また，出力にはゴール append/3 の第 2 入力引数が結線されている．この時点では，これまでの巻き戻しのチェックの結果として append/3 には，全ての節が正しい出力をしたことを示す濃い黄色が付けられているとする．この図の状態から実行を再開すると図 4.10(b)となる．空リストを反転したリストは空リストであるので，wrongrev/2 の出力が結線されていた append/3 の引数に正しく空リストが結線されたことが分かる．よって，今回の wrongrev/2 の出力は正しいことが分かる．従って，今回リダクションされた節にもチェックした印として濃い赤い色を付ける．

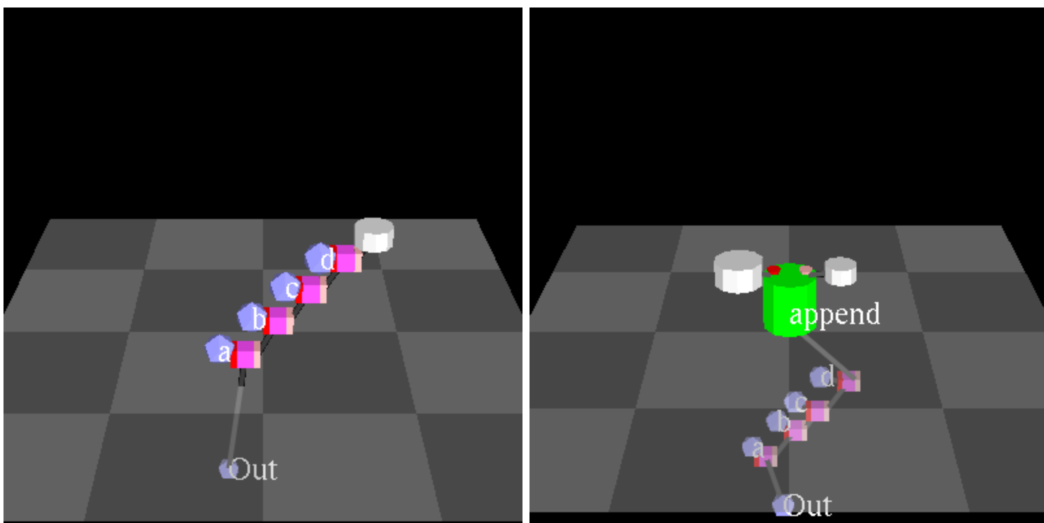
(c) ゴール wrongrev/2 の動作のチェックの例 2
さらに実行を巻き戻し，今度は図 4.11(a)の状態を調べる．このときの wrongrev/2 は，入力引数にリスト [a,b,c,d] が，出力引数にアトム Out がそれぞれ与えられている．wrongrev/2 の節のうち一つは前述のチェックによって正しい動作をしたので，色が付けられている (図 4.11(a)中の左側の節を参照)．これを実行すると図 4.11(b)の様になる．図 4.11(b)では，リスト [b,c,d] が入力として与えられたゴール wrongrev/2 とリスト [a とゴール wrongrev/3 の出力を入力引数としたゴール append/3 がある．append/3 は以前の調査で正しく動作することが分かっている．また，ユーザは wrongrev/3 の出力はリストを逆順にしたものであると考えている．よって，ユーザはこの状態からの実行結果は [a,d,c,b] となることが予想できる．従って，この予想される実行結果は，ユーザの意図した出力とは異なっている．よって，今回の wrongrev/2 のリダクションは意図した wrongrev/2 の動作とは違うことが分かる．すなわち，バグは wrongrev/2 のまだ色が付けられていないほうの節 (図 4.11(a)中の右側の節) の定義にあるのではないかと予想できる．

3. バグのある節の定義の修正
2cでバグを含む可能性のある節が発見されたので，プログラムが実行を開始する前まで巻き戻し，節の定義の修正をする．



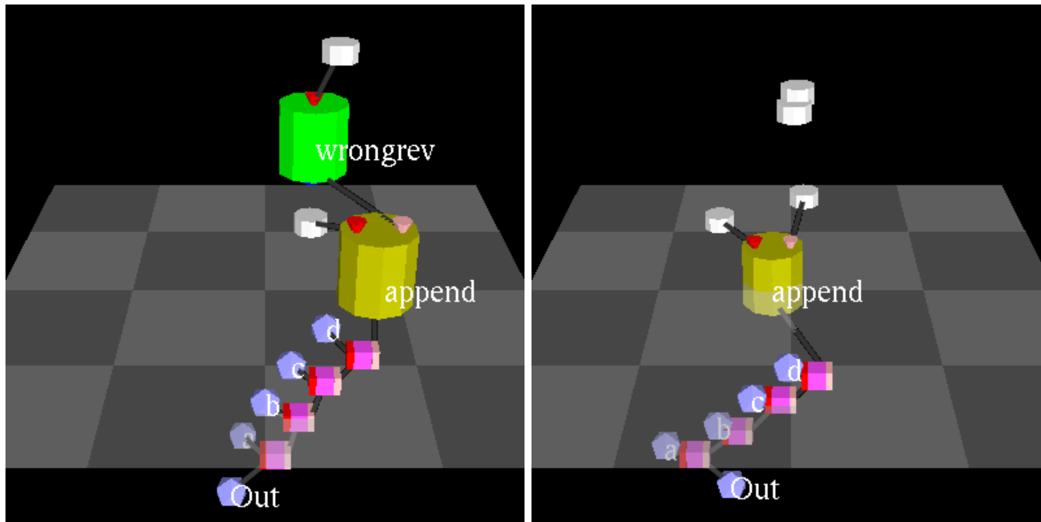
(a) 実行前　　(b) 間違った実行結果

図 4.8: バグのあるプログラムの実行結果



(a) 巻き戻し前　　(b) 巻き戻し後

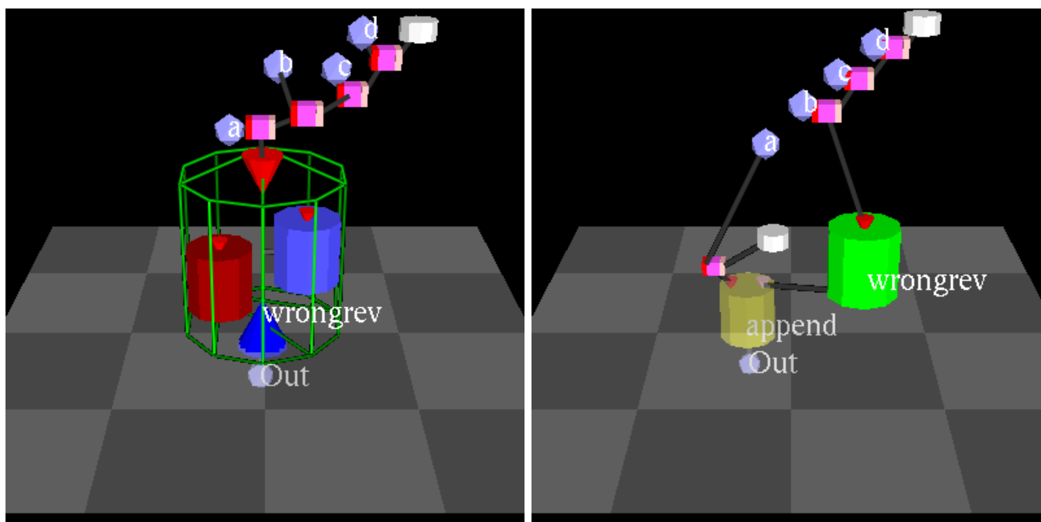
図 4.9: ゴールの動作の調査 – 正しく動作 (append/3)



(a) 巻き戻し調査 (wrongrev/2 の節 1) – リダクション前

(b) 巻き戻し調査 (wrongrev/2 の節 1) – リダクション後

図 4.10: ゴールの動作の調査 – 正しく動作 (wrongrev/2)



(a) 巻き戻し調査 (wrongrev/2 の節 2) – リダクション前

(b) 巻き戻し調査 (wrongrev/2 の節 2) – リダクション後

図 4.11: ゴールの動作の調査 – 間違いを発見 (wrongrev/2)

第5章　関連研究

5.1　実行の視覚化に関する研究

Pictorial Janus[1, 24],SAM[6] はプログラムの実行の視覚化という点で本研究と関連している．また，これらのシステムは図形を編集することによりプログラムの編集を行っているシステムである．

5.1.1　Pictorial Janus

Pictorial Janus は論理型言語 Janus を 2 次元図形を用いて視覚化するビジュアルプログラミングシステムである．また，視覚化されたプログラムの実行をアニメーションで表す．Pictorial Janus では視覚化されたプログラムの実行をアニメーションで表しているが，その実行はビデオにとられたものをただ再生しているように一方向のみであり，実行の操作は行えない．本研究では，アニメーションを操作することで実行をユーザが操作することが出来る．アニメーションは任意の時点で一時停止でき，一時停止中にも視点変更やオブジェクトの移動といったさまざまな操作をすることが出来る．また，実行の速度を任意に調整でき，実行の巻き戻しによって一度実行されたプログラムの状態を任意の時点に巻き戻すことが出来る．これらによって，ユーザが実行を観察する際に，ユーザがユーザ自身のペースで見やすいように調節して観察することが出来る．

5.1.2　SAM

SAM は並列システムを詳細に規定するためのビジュアルプログラミング言語である．SAM のプログラムは message，port 付き agent(port with agent)，rule，action の列といった要素から構成されている．また，それらは 3 次元の図形によって表されている．SAM の実行では，条件によって選択された 1 つの rule が拡大するアニメーションを見せ，そして，rule の持つ action が実行される．SAM の action は message を生成し出力するアニメーションによって表される．これは実行アニメーションにおけるゴールの拡大アニメーションとゴールの展開の後に節の定義のグラフが現れてくるアニメーションと類似している．しかし，実行アニメーションではゴールのオブジェクトの中に節そのもの，節のオブジェクトの中に定義を表すグラフそのものが入っており，定義のグラフとゴールのオブジェクトが置き換えられることによって実行が進められる．また，SAM の実行では，ユーザは実行中に何らかの操作することは出来な

い．一方，実行アニメーションでは，実行を一時停止して観察することができる．また，一時停止中にオブジェクトの移動，視点変更，実行アニメーションの速度の変更といったユーザがより見やすく出来るような操作が行える．さらに，実行の巻き戻しのような実行状態を制御するための操作を行うことが出来る．

5.2 デバッグに関する研究

竹内の [23] と Kish Shen，Steve Gregory らの [25] は並列論理型言語のデバッグの手法について述べている．Animation-3DPP がベースとしている言語も並列論理型言語であり，本研究ではデバッグ手法を扱っているため，これらを関連研究として挙げる．

5.2.1 Algorithmic Debugging of GHC

[23] では GHC プログラムの実行をトレースし，ゴールがリダクションされるたびに動作が正しいかどうかをユーザに問うことによってバグのある箇所を特定する手法を提案している．また，その手法では GHC プログラムの実行をトレースするためのプログラムも GHC で実装されている．

我々はデバッグでユーザが実行を巻き戻し，再度実行することによってゴールの動作をチェックするという手法をとっている．これは [23] における動作が正しいかどうかをユーザに問う，ということに類似する．相違点としては，[23] は動作が正しいかどうかをユーザに問うことで計算によってバグのある節を特定し出力する．Animation-3DPP はユーザは節の色付きオブジェクトで正しい動作をした節に色を付けていくことによって，正しい動作をした節とそうでない節を分けていく．という点が挙げられる．すなわち，計算ではなく視覚的にバグのある可能性のある範囲を狭めていく．これにより，チェックしたゴールとチェックしていないゴールとを見て判断することが出来る．

5.2.2 Instant Replay of Debugging

[25] では，並列論理型言語のゴールのリダクション (引数の値，節の選択) などの記録をとり，そのときの実行の動作と全く同じ動作を行う実行のリプレイを見ることが出来る．

本研究の実行アニメーションおよび実行の巻き戻しでは，実行を何度でも巻き戻し，また再度実行することが出来る．このとき，一時停止中の書き換えによってプログラムを変更しなければ，必ず同じ実行が現れる．すなわち，本研究においても全く同じ動作を行う実行のリプレイを見ることが出来る．

相違点としては，本研究では，実行一時停止中の書き換えによってプログラムを一時的に書き換えることが出来る機能がある点が挙げられる．実行一時停止中の書き換えを用いることによって，ゴールに引数に別の値が与えられていたらどのような処理が行われ，どのような結果が出たのかということをテストすることが可能となる．これは，バグのある部分が特

定できた場合，その定義をどのように修正すればよいかということに対しての手助けとなる．

第6章　まとめと今後の課題

本研究では，プログラムの実行における変化をビジュアルシステム上で視覚化するための手法に関して述べた．本研究でのプログラム実行の視覚化は変化の前後が掴みやすい様に表す，プログラムの実行の進み具合などをユーザ自身が手軽に調節できるようにするといったことを重視している．そこで，実行時の変化をオブジェクトが次々と書き換わって行くアニメーションを用いて表す実行アニメーションの手法を提案し，実装を行った．

実行アニメーションの手法を用いることにより，ビジュアルプログラミングシステムのユーザは実行の過程をアニメーションで観察することができる．また，その実行の進み具合をアニメーションを操作することによってユーザの好きなように調節することが可能となった．

また，本研究では実行アニメーションをデバッグに利用することに着目した．そのためにAnimation-3DPP にアニメーションをより細かく操作するための機能と，アニメーション中のオブジェトを色分けする機能を導入した．アニメーションをより細かく操作するための機能は実行の巻き戻し，プログラムの部分実行，プログラムのステップ実行である．アニメーション中のオブジェトを色分けする機能は色付きオブジェクトである．また，ゴールに別の引数が与えられていたときの処理をテストするための機能として，実行の一時停止中の書き換えの各機能を導入した．これによって，プログラム実行をアニメーションで観察し，プログラム中の正しい動作する部分とそうでない部分を色分けしていくことでバグのある部分を特定することが可能となった．

今後の課題としては，いろいろな種類のバグに対してデバッグが行えるように本研究のデバッグ手法の拡張を行うことを考えている．現在適用を考えているバグとしては，無限ループのバグがある．無限ループはいつまでたっても実行が終了しない現象が起こるバグである．視覚化されたプログラムの実行では，無限ループのバグは延々と同じ定義が出現するという形で現れる．同じ定義が出現するため，同名のゴールが何度もリダクションされると考えられる．現在のところ，ゴールのうち，異常な回数リダクションされているものを見つけることがバグのある節の絞り込みに繋がるのではないかと考えている．ゴールにリダクションの回数によって何らかの変化，例えば，色の変化や形の変化などを与えることによって異常な回数のリダクションを発見することを考えている．

また，本論文で例として挙げた GHC プログラムはどれも小規模のものであった．今後は，今回例として用いたプログラムよりも大きなプログラムへの対応を考えて行きたいと思っている．そのために，視覚化されたプログラムの見せ方の工夫や見やすいようにプログラム表現をを変更出来るような機能を考えている．

謝辞

本研究を進めるにあたって指導教官である田中二郎教授，および，志築文太郎先生，三浦元喜助手からは終始丁寧なご指導を頂きました．心より感謝致します．また，IPLAB の皆さんからも貴重なご意見，ご指導を頂きました．特に，同じ VS グループのメンバーである飯塚和久さん，亀山裕亮さん，劉学軍さん，根本浩史さんからは研究の進め方に関して大変有益なご意見を頂きました．ここに感謝の意を表します．

参考文献

[1] Ken Kahn. Concurrent Constraint Programs to Parse and Animate Pictures of Concurrent Constraint Programs. Technical Report SSL91-16/P91-00143, XEROX PARC, 1991.

[2] 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. 共立出版, 1996.

[3] 田中二郎, 後藤和貴, 馬場昭宏. ビジュアルプログラミングシステムにおける入力法の効率化. 日本ソフトウェア科学会第 12 回大会論文集, pp. 165–168. 日本ソフトウェア科学会, September 1995.

[4] 南雲淳, 田中二郎. viewPP:グラフ構造とアニメーション表現に基づくプログラム実行の視覚化. 日本ソフトウェア科学会第 14 回大会論文集, pp. 17–20. 日本ソフトウェア科学会, September 1997.

[5] 遠藤浩通, 田中二郎. 単一ビュー/モードに基づくビジュアルプログラミング環境の構築. インタラクション'99 論文集, pp. 81–87. 情報処理学会, March 1999.

[6] Christain Geiger, Wolfgang Mueller and Waldemar Resenbach. SAM - An Animated 3D Programming Language. In *Proceedings 1998 IEEE Symposium on Visual Languages(VL '98)*, pp. 228–235, September 1998.

[7] Marc A. Najork. Programming in Three Dimensions. *Journal of Visual Languages and Computing(1996)*, pp. 219–242, September 1996.

[8] Kakuya Yamamoto. 3D-Visulan: A 3D Programming Language for 3D Applications. In *Proc. of Pacific Workshop on Distributed Multimedia Systems*, pp. 199–206, 1996.

[9] Takashi Oshiba and Jiro Tanaka. "3D-PP": Visual Programming System with Three-Dimensional Representation. In *Proceedings of International Symposium on Future Software Technology*, pp. 61–66, 1999.

[10] 宮城幸司, 大芝崇, 田中二郎. 三次元ビジュアル・プログラミング・システム 3D-PP. 日本ソフトウェア科学会第 15 回大会論文集, pp. 125–128, September 1998.

[11] 古川康一, 溝口文雄. 並列論理型言語 GHC とその応用. 共立出版, 1987.

[12] Kazunori Ueda. Guarded Horn Clause. Technical report, ICOT, 1985.

[13] Jiro Tanaka. Visual Programming System for Parallel Logic Languages. In *Proceedings of the NSF/ICOT Workshop on Parallel Logic Programming and its Program Environments*, pp. 175–186. the University of Oregon, March 1994.

[14] 神谷誠. 3 次元ビジュアルプログラミングシステムにおけるドラッグ＆ドロップ手法の拡張. 筑波大学 第三学群 工学システム学類 卒業論文, 1999.

[15] 大芝崇. 3 次元モデリングツール "Claymore":付加情報によって強化された直接操作. 日本ソフトウェア科学会第 15 回大会論文集, pp. 161–164, 1998.

[16] 宮下貴史, 田中二郎. 3 次元スプリングモデルと拡張直接操作手法の統合. 情報処理学会論文誌, Vol. 42, No. March, pp. 565–576, 2001.

[17] 山田英仁, 田中二郎. 3 次元空間上での自由な配置が可能なメニュー. 日本ソフトウェア科学会第 18 回大会, September 2001.

[18] 神田正和. ３次元ビジュアルプログラミングシステム 3D-PP の編集効率の向上に関する研究. 筑波大学 第三学群 情報学類 卒業論文, 2001.

[19] 岡村寿幸, 田中二郎. 3 次元ビジュアルプログラミング環境における視覚化手法. 日本ソフトウェア科学会第 19 回大会, September 2002.

[20] 岡村寿幸, 田中二郎. 3 次元ビジュアルプログラミングにおける視覚化手法. 日本ソフトウェア科学会 WISS2002, 2002.

[21] Jackie Neider Manson Woo and Tom Davis. OpenGL プログラミングガイド第 2 版. ピアソン・エデュケーション, 1997.

[22] Mark J. Kilgard. *The OpenGL Utility Toolkit (GLUT) Programming Interface: API Version 3*. Silicon Graphics Inc, 1996.

[23] A. Takeuchi. Algorithmic Debugging of GHC Programs and its implementation in GHC. Technical Report TR-185, ICOT, 1986.

[24] Kenneth M. Kahn and Vijay A. Saraswat. Complete Visualizations of Concurrent Programs and their Excutions. *Workshop on Logic Programming Environments*, pp. 30–34, 1990.

[25] Kish Shen and Steve Gregory. Instant Replay Debugging of Concurrent Logic Programs. *New Generation Computing*, Vol. 14, No. 1, pp. 79–107, 1996.