書換え型計算モデルの解析・検証・変換のための視覚的支援手法に関する研究

河口信夫

目次

図一覧

表一覧

第 1 章

序論

1.1 はじめに

計算機技術の発達や計算機ネットワークの巨大化にともない，社会のあらゆる場面で計算機システムが利用される高度情報化社会が到来しつつある．計算機システムが社会において果たす役割が重要になればなるほど，システムが誤りを犯すと社会に重大な損害を与えかねない．例えば，ネットワーク管理システムや銀行等のオンラインシステム，工場管理システムにおける誤りは，それを利用する企業や個人に多大な損害を与えかねない．また，自動車や飛行機の操縦や運用を管理する移動体管理システムや，医療機器の制御システムにおける誤りは，人命に危害を及ぼす可能性を持つ．このようなミッション・クリティカルな計算機システムでは，それを構成するハードウェア，ソフトウェア両面の信頼性が厳しく要求される．ハードウェアに対しては，多重化や誤り訂正などに基づく信頼性向上のための実用的手法が数多く存在し，高い成果が得られている．ソフトウェアに対しても近年，マルチバージョンソフトウエア [85] やリカバリーブロック法 [96] など，ハードウェアと同様の手法を用いて信頼性を向上させる研究 [66] が行なわれているが，その開発・保守には，多くの場合，人手と時間を大量に用いた人海戦術が用いられている．しかし，このような手法をもってしても信頼性の高いソフトウェアを確実に開発・保守することはできない．ソフトウェアの誤りは本質

的にハードウェアに起因する物理的故障によるものではなく，論理的な誤りによるものである．これは，ソフトウェア開発が純粋に人の知的作業であり，欠陥や誤りが潜入することが避けられないことに起因する．そのため，信頼性の高いソフトウェアを構築するための計算機による支援が切実に望まれている．これに応えるためには，ソフトウェアの開発手法や開発支援環境等の整備が急務であり，その理論的根拠や指針を支えるための基礎的な研究が必要である．

近年，計算モデルに基づいたソフトウェアの検証技法が研究されている [28, 81, 59]．このような形式的アプローチに基づいてソフトウェアを検証することにより，高い信頼性を持つソフトウェアの構築が可能になると期待できる．関数型計算モデルである項書換え系 (Term Rewriting Systems)[22, 44] は，マッチングと書換えを基本とした単純な計算モデルであり，直観に優れた代数的意味論 [61] を特長とし，豊かな理論的成果を持つ．例えば，等式によって記述される代数的仕様記述は，等式を左辺から右辺への書換え規則とすることで項書換え系とみなすことができ，その結果，項書換え系の計算は代数的仕様の直接実行とみなせる．酒井らによる Cdimple[80] はこのことを利用した抽象データ型の高速な直接実現システムである．

項書換え系においては様々な性質が解明されており，形式的検証に適用することが可能である．項書換え系の停止性 [10] や合流性 [22] の十分条件に関する研究成果に基づく Knuth-Bendix アルゴリズム [47] や，被覆集合帰納法 [81] による検証技法は，与えられた項書換え系が所期の性質を満たすかどうかの検証に使われる．また [65] では，モジュール化された項書換え系を合成して得られる項書換え系においても，モジュールの持つ性質が保存されることを保証する十分条件が述べられており，プログラム合成に関する基本的性質を明らかにしている．

しかし，このような形式的検証を利用して項書換え系に誤りが発見された場合，これまでの形式的検証手法では，誤りの有無を示すことはできても，具体的にどの部分が誤っているかを指摘することは困難であった．また，形式的検証による

発見が困難な誤りも存在する．例えば，設計者が要求仕様を勘違いした場合などによる項書換え系設計時の誤りは，形式的手法を用いて検出できるものばかりではない．そこで，具体的な誤りの位置の確認や，検証により検出されないような誤りを発見するには，実際に項書換え系を実行してその振舞いを観察することが一つの解決策となる．また，正しく動作する項書換え系であってもその実行効率に問題がある場合には，プログラム変換によって効率改善を行なうことが望ましいが，具体的に項書換え系のどの部分を変換するべきかを判断する手法についてはこれまでに十分な研究成果は得られていない．そのため，項書換え系の動作を観察し，項の構造の繰り返しパターン等を発見することによって，効率改善の鍵となる部分を発見することが必要となる．さらに，検証手法や変換手法そのものを構築する場合も，項の構造の解析がヒントになる場合が多い．

このように検証・変換技法を実際に適用する場合や，検証・変換手法の研究においては，実際に項書換え系の動作の解析を行なう事例研究が重要な役割を果たす．そのため，これまでに様々な項書換え系の処理系が開発されている．ReDuX[4]はテキストインタフェースによって実現された項書換え系のワークベンチであり，再帰経路順序 [10] などによる停止性の判定や Knuth-Bendix アルゴリズムなどが実現されている．MERILL[52] は等式推論を扱うシステムであり，順序づけされた項の上での推論など，様々な解析が可能である．しかし，ReDuX や MERILL など，これまでに開発されてきた項書換え系の解析・検証のための支援環境は，そのほとんどがテキスト処理[†1]を基本としている．項書換え系では計算過程において様々な種類の情報が大量に存在するため，ある程度の規模の計算をテキスト処理によって解析を行なうと，膨大な量のテキストが表示されることとなり，適切な情報を選択して解析を行なうことが困難である．特に項の構造に対する直観的な判断を必要とするような「設計時の誤りの発見」，「変換着目点の検出」を行なうには，テキスト処理に基づく支援は困難であるといえる．

[†1] 文字による処理

一方，近年のハードウェア環境の充実とともに，「視覚化 (Visualization)」の技術が急速に発展している．一般のワークステーションによっても高度なグラフィック表示が十分な速度で可能になり，次節に示すように計算機上のデータやプログラムに対し，様々な視覚化手法が提案されている．視覚化技術を用いて，数値情報や構造を持つ情報をグラフや図形として表現することにより，図形に対する人の直観的理解能力やパターン認識能力を利用できる．複雑な事象や構造を解析するために，計算機による支援に加えて，人の視覚的理解能力を解析の支援に利用することによって，情報に対する直観的な理解が可能になる．

ところが，項書換え系に代表される基礎的な計算モデルや形式仕様の研究においては，その解析・検証のための実用的な視覚化手法がほとんど提案されていないのが現状である．その一つの原因としては，これまでの形式的アプローチの研究がトイ・プログラムと呼ばれる小規模なプログラムを主な対象としてきたことが挙げられる．小規模なプログラムでは，注意深く解析を行なうことによってその性質をある程度把握できるため，計算機による視覚化などの特別な支援を必要としない．これまでの計算機による検証技法の実現は，項書換え系のプログラミングに対する実用的な支援を目的とするより，むしろ形式的検証手法の実現可能性を確認するために行なわれている場合が多い．

本論文では，項書換え系の解析や検証・変換などの形式的手法の支援に視覚化技術を用いることを提案する．視覚化技術の導入により，データの構造の直観的理解や検証対象や変換対象の絞り込みなど，形式的手法に不足している部分を人の直観により補うことが可能な実用的な解析支援システムの実現が期待される．さらには以下に述べる事項が期待できる．

- 視覚的表現による情報の直観的理解

  情報を図形や色を用いて表現することによって，人の直観的理解力やパターン認識の能力を利用することが可能になり，項そのものの構造や，項の間の書換え関係などの直観的理解が可能になる．

- 計算系列中における誤りの出現位置の発見

  項書換え系の計算系列を視覚化することにより，特別なパターンを持つ構造の出現位置の発見が容易になる．形式的検証によって検出された誤りや設計時の勘違い等に基づく誤りは，誤りを含む項の構造が特徴的なパターンを持つ場合，その構造の具体的な出現位置の発見が視覚化によって容易になる．

- 実行効率改善の鍵の発見

  項書換え系の実行の効率化等を行なう場合，効率化変換などの理論的成果を単純に応用するのは困難な場合が多く，そのような場合は効率改善の鍵となる部分を発見する必要がある．項書換え系の計算をその系列や状態遷移関係を用いて視覚化することにより，プログラム変換が適用可能な項の構造の繰り返しパターンや，冗長な書換え系列の直観的な発見が可能になる．

- 高度な解析手法や様々な性質の直観的利用

  複雑な条件を持つ項の上の順序や，モジュール分割に基づく性質など，項書換え系に関する様々な研究成果を，計算機の支援なしに利用するのは困難である．それぞれの条件や性質に対し視覚化手法を定め，計算機による視覚的支援を行なうことにより，項や書換え系列の具体的にどの部分がそれらの条件を満たし，性質を持つかの直観的な判断が可能になる．

- プログラミングやデバッグの視覚的実現

  計算を実行するためのインタフェースを視覚的に実現することにより，視覚的なデバッグが可能となり，プログラムを直観的に操作できる．また，ユーザによる図形の配置や操作は，パターンや例示に基づく視覚的プログラミングとみなせる．

- 新たな解析手法や変換手法の獲得

  視覚的表現を用い，直観的に解析や変換を行なうことにより，新たな解析手法や変換手法を獲得できる可能性がある．

このように，形式的手法に対し視覚化技術を適用により，さまざまな事柄が可能になると期待されるが，この場合，どのような視覚化技術をどのように項書換え系に適用させるかが問題となる．次節以降では，まず，視覚化技術の現在の研究動向について述べ，視覚化技術の分類ならびにその有用性について述べる．次に，項書換え系に関する近年の研究動向について述べ，視覚化技術の適応可能性について議論する．最後に視覚化技術の項書換え系への適用手法を述べ，本論文の目的と方法を示す．

1.2 視覚化技術とその有用性

視覚的に計算機上の情報を扱う手法については様々な研究 [78] がなされているが，本論文では特にソフトウェアに関連する視覚化に焦点を絞った議論を行なう．市川ら [25] はこの分野の研究がプログラムの視覚化，視覚的プログラミング，アルゴリズムの視覚化，の 3 種類に分類できるとしているが，本論文では，プログラムが用いるデータに対する視覚化を新たに加え，視覚化技術を以下の 4 種類に分類する．

(1) データの視覚化

計算機上の構造データや数値データを直観的理解が可能な形で表示する．

(2) プログラムの視覚化

プログラムを視覚的に表示し，その構造や関係を明らかにする．

(3) 視覚的プログラミング

視覚的なプログラム構成要素を用い，テキストを用いずに直接プログラミングを行なう．

(4) プログラムアニメーション

プログラムの動作をアニメーションを用いて表現し，プログラムの動的な変化の様子を明らかにする．

これ以外にも，計算機の支援による視覚化として，流体現象や温度変化などに代表される直接目に見えない物理現象を，理論や数値データに基づいて可視化し，直観的な現象の理解を可能にする手法としてサイエンティフィック・ビジュアライゼーションがあるが，ソフトウェアの視覚化とは直接結び付かないので，本論文では検討の対象としない．

以下では，それぞれの分野について現在の研究動向を述べ，その有用性を示す．

1.2.1 データの視覚化

様々な情報をどのように加工して人間にわかりやすく表示するか，という視覚化技術のもっとも基本となるのがデータの視覚化（情報の可視化）である．一般に，構造を持つ情報を図式化するには木構造やグラフ構造が用いられる．[101, 3] ではいわゆる木構造やグラフ表現をどのように描画すれば良いのかについて議論しており，美しい図式を描画する問題はNP完全であることを明らかにしている．また，[100] は並列プログラムに対して様々な情報の可視化を解説している．一般に2次元的な図形で問題となるのは表現の対象となるディスプレイ (表示領域) の大きさの問題である．つまり，大規模データを扱おうとすると画面の大きさが足らず，縮小表示をすると個々の情報が利用できないなどの問題が起こる．そこで，可視化に新しい視点を採り入れる手法や3次元空間を用いる解決法が提案されている．

[58] はFishEye View や Perspective View などに代表される非均一表示方法の新しい手法の提案である．複数の注目点に対しXY軸方向のみを拡大することで，歪みの少ない2次元表示を得ることができ，必要な情報のみを正しく取り出すことが可能になる．[75] では構造を持つデータを Information Cube と呼ばれ

る 3 次元空間上の半透明な箱の包含関係を用いて表現している．[46] では xy 軸でプロセスの通信関係，z 軸で時間軸に沿った通信を表すことで，複数の情報を同時に表現することを可能にしている．様々な情報を 3 次元空間を用いて可視化しようという試みについては [45] が良い解説になっており，近年のハードウェアの発達が計算機上での 3 次元情報可視化を容易にしていることがわかる．

1.2.2 プログラムの視覚化

プログラムの視覚化 (*program visualization*) とは，プログラムそのもの，もしくはプログラムの動作の状態 (スナップショット) を図形を用いて表現することにより，プログラムに対する直観的理解を可能にするものである [24]．すでにテキストで記述されているプログラムは，そのままでは内容や動作を理解することが困難であるので，実際の動作や状態変化に対して，何らかの視覚化手法を用いて表現を行なう．例えば，プログラムをフローチャート，PAD，NS チャートなどのプログラム図式で表すこともプログラムの視覚化の一種である．二村 [12] はこれまでに提案されてきた 17 種類の図式が，実は表現形態が異なるのみで，本質的には同一であることを述べており，同時にプログラム図式を用いることの利点を主張している．その他の図式として HiChart[62, 17, 101]，λ 式のための図式 LAD[13] などが提案されている．

最近では，いわゆる CASE ツール [89] と呼ばれる開発環境が進められており，C 言語では，データフロー図 [103] や関数の呼び出し関係 [29] を作図するシステム [51] が開発されている．また，ユーザインタフェース構築のための環境としては [1, 26, 94, 43, 20, 30, 48, 57, 92, 102] などがある．

FE'92(Future Environment)[93] は GHC プログラムの視覚的入力システムである．ゴールに対して入力変数と出力変数を明示的にすることで，入力されたプログラムを図式表現へ変換することと同時に図式からプログラムを生成することができる．つまりプログラムの視覚化と同時に，視覚的プログラミングを可能に

しているともいえる．この支援環境はその視覚化の対象言語である GHC 自身によって記述されている．Polak ら [72] は Prolog のプログラムを単純な図式を用いて表現する手法を提案している．Prolog の節を 4 つのアークを持つノードとして記述し，その結合によってプログラムを表現している．

1.2.3 視覚的プログラミング

視覚的プログラミング (*visual programming*)[25, 18] とは，図形を基本にしてプログラミングを行なう手法であり，アイコンプログラミングや例示プログラミング (*programming by example*) などが研究されている [84]．プログラムの視覚化は，すでに存在するプログラムテキストや，プログラムの動作をどのように視覚化するかが問題となっている．それに対し，視覚的プログラミングでは，最初に視覚的要素としてのプログラム部品があり，それを組み合わせてプログラミングを行なう．つまり，視覚的な部品をどのように構成するかが問題となる．

布川らは [64] で図による関数型プログラミング言語を提唱している．これはどちらかというとプログラム図式に近い考え方であり，元にしている関数型言語に比べ制約の多いものになっている．

PSDL-GR[82] は E-R モデル (*entity relationship model*) を用いた視覚的言語であり，制約を用いた論理言語で解釈して実行する．すでに視覚的プログラミングの処理系のプロトタイプが実装されている．

VPE[16] は IntelligentPad の組み合わせでプログラミングを行なう視覚的統合論理型プログラミング環境である．VPE ではプログラミング，デバッグ，実行をすべて同じ IntelligentPad の枠組で行なうことが可能であり，パッドを組み合わせることで合成プログラミングを行なう．

Prograph CPX[7] は Macintosh 上で実現されている視覚的プログラミング言語であり，十分に実用アプリケーションを開発する能力を持つ．

1.2.4 アルゴリズムアニメーション

アルゴリズムなどの動作を視覚化するために，アニメーションを使う技術が近年注目を浴びている [68, 50]．高橋ら [56, 91] はアルゴリズムを構成する各オブジェクトに対し宣言的記述を行なうことで，容易にアルゴリズムアニメーションを実現する言語を提案している．また 3 次元可視化手法 [45, 75] では，3 次元空間での移動をユーザに認識させるために，認知をさまたげない速度でのアニメーションを利用することを提案している．Arya[2] は関数型言語と遅延評価を用いてアニメーションを作成する手法を提案している．このアニメーションシステムは Haskell によって実現されている．

1.3 項書換え系に関する研究動向

本節では，本論文が視覚化の対象とする項書換え系について，近年の研究動向を述べ，視覚化技術の適用可能性についての議論を行なう．

1.3.1 等価性，等価変換

複数の項書換え系が同じ意味を持つことを示す手法や，項書換え系をその意味を保存したまま等価な項書換え系へ変換する手法が研究されている．この場合，項書換え系をどのような意味で捉え，どのような等価性を用いるかが問題となる．また，等価変換においては，変換後の項書換え系が変換前のものより実行効率が良い場合，この変換は効率化変換と呼ばれる．

Partsch らによる [71] はプログラム変換手法・システムに関する総括的な解説である．プログラム変換の分野では非常に多くのシステムが実現されてきたが，大きな成功を収めたものは少ない．

Burstall ら [5] は fold/unfold 法と呼ばれる古典的な効率化のためのプログラム効率化変換手法を提案した．この手法は，冗長な計算のステップをまとめて一度

に行なうことにより実現されるため，計算の系列の解析によって適用可能性が判断できる．すなわち，視覚化によって適用場所を発見できる可能性を持つ．

また，外山 [99] は，限られた関数記号に基づく項に対して，2 つの項書換え系によって計算される正規形が等しくなるための十分条件を，到達可能性を用いて明らかにした．また，その性質を用いてプログラム等価変換の基本方法を示した．この手法に従えば，等価なプログラムが得られるため，変換の適用によっては効率化が可能である．この場合も変換の適用場所を発見するための視覚的支援は有効であると考えられる．しかし，効率化が可能なプログラムのクラスは非常に限られたものであり，多くのさらなる研究成果を必要としている．

1.3.2 停止性

項書換え系が無限の書換え系列を持たないことを，項書換え系が停止性を持つという．Dershowitz[9, 10] は項書換え系の停止性判定手法に関して，単純化順序と呼ばれる整礎な順序を用いる手法を定め，そのクラスに属する再帰経路順序などの順序について総括的な解説を行なっている．Steinbach[87, 88] は，これまでに提案されてきた多くの種類の停止性判定手法の間にどのような関係があるかを，項書換え系に制限を加えて図式化した．それぞれの判定手法には得意，不得意があるため，どの項書換え系にどの手法を適用させるかを判断するために，この図式は有用である．

Zantema[105] は停止性に関して total termination というクラスを定義しており，その非決定性を明らかにしている．また，大崎ら [69] は意味ラベリングと分配消去法に基づく停止性判定手法を提案している．このように，停止性判定手法が次々と考案される中，それらをどのように利用するかに関する知識が必要となる．計算機による停止性判定のための視覚的支援の構築により，停止性判定手法の適用手法の知識の獲得が期待される．

1.3.3 モジュール性

項書換え系 R_1, R_2 の合成に関して性質 P がモジュール性を持つとは，R_1, R_2 がともに性質 P を持ち，かつそのときに限り項書換え系 $R_1 \cup R_2$ が性質 P を持つことである．

この分野の研究では，まず，外山 [97, 98] が重なりのない項書換え系の直和に関して，合流性および，左線形かつ完備性がモジュール性を持つことを明らかにしている．項書換え系の階層的な組み合わせについては Krishna[74] が完備性がモジュール性を持つ十分条件を明らかにした．近年では Ohlebusch[65] がモジュール性についての体系的な整理を行なっている．

モジュール性は単純に組み合わせが可能な場合と，複雑な条件が必要な場合が存在し，それぞれが項書換え系に必要とする条件が異なり，すべてを考慮に入れるのは困難である．そのため，計算機による支援が望ましい．これらの性質を計算機による視覚化を用いて図式化することにより，モジュール性の利用が容易になることが期待される．

1.3.4 実行・検証支援環境

項書換え系の実行環境，検証のための支援環境は多くの種類が実際に計算機上に開発されている．

ReDuX[4] はテキストインタフェースによって実現された項書換え系のワークベンチである．ReDuX は様々なサブコンポーネントを持ち，Knuth-Bendix 完備化アルゴリズムや AC マッチング，AC 単一化，AC 完備化などを実現している．

MERILL[52] は等式を扱うシステムであり，関数型言語 StandardML により実現されている．順序づけされた項上での等式推論や，AC 演算子の利用，完備化アルゴリズムなど様々な解析をテキストレベルのコマンドを入力して行なうことができる．MERILL は ReDuX より若干洗練されたインタフェースを持つが，

やはりテキストベースのシステムであり，視覚的な表示を行なう機能は持たない．MERILL や ReDuX などのシステムでは代数的な解析を行なうことは可能であるが，特定の構造を直観的に発見することは困難である．

1.4 本論文の目的と方法

本論文では，項書換え系に対する解析・検証・変換のための視覚的支援手法を確立することを目的とし，前節で述べた 4 種類の視覚化技術を項書換え系，項書換え系の計算，および項書換え系のプログラミングツール [33] に適用する．項書換え系の基本となる項や，計算の過程で抽出される数値情報に対してはデータの視覚化の技術を用いる．項書換え系の規則や，動作の系列，状態間の関係などは，プログラムの視覚化として実現する．また，項書換え系そのものをプログラムの視覚化を利用して，視覚的プログラミング言語として用いる．項書換え系の動作を表すためにデータの視覚化やプログラムの視覚化に基づくプログラムアニメーションを用いる．

本論文では，これらの視覚化により，項書換え系に対する直観的理解を深めるとともに，その解析・検証・変換を容易にする視覚的支援手法を提案する．また，その手法に基づく支援環境の実現を行なう．本論文の具体的な目的として以下の 2 点を挙げる．

1. **項書換え系の解析・検証・変換のための視覚的支援手法の確立**

   これまでのテキスト環境では困難な，図を用いた高度な解析や検証に対する支援，項の構造の直観的理解による変換着目点の発見などを可能にする視覚的支援手法が必要である．この視覚化手法は，大規模な項に対しても有効であることが要求される．

2. **視覚的支援手法に基づく支援環境の実現**

   本論文で提案する視覚的支援手法に基づいて，項書換え系の視覚的支援環

境を実現する．この支援環境は，項書換え系に関する理論的研究の成果に基づいて，項書換え系の解析や検証のために用いるものであるため，常に最新の成果を用いられるよう，研究者自身による変更が容易であることが必要となる．

実用的なツールとして利用するためには，計算時間や，ユーザからの入力に対する反応時間が短いこと，すなわち，実行効率が良い実現であることが望ましい．

本論文では，これらの目的を達成するために具体的に以下に挙げる方法に基づいて研究を行なった．

(1) 項書換え系の視覚化の提案

項書換え系の解析・検証・変換などで必要な操作を直観的に支援するための手法として，項書換え系の視覚化を提案する．そのために，項書換え系の計算を詳細に調査・検討し，視覚化技術を項書換え系に特化して，視覚化を項，計算，インタフェース，数値情報の４種類に分類し，さらに個々の視覚化についての検討を行なった．

項の視覚化は項の構造の直観的理解を支援するため，木構造図式で項を表す．この木構造図式は部分項を単一のノードとして表現する機能や正規形を簡単な構造として表現する機能を持つため，大規模な項であっても表現が可能である．また，各ノードの関数記号に対して，色や図形を用いて関数のソートなどの情報をわかりやすく表現する．

計算の視覚化は計算の振舞いを理解するために，項の視覚化に基づいて，項書換え系の計算過程を表す．計算の視覚化として，２種類の手法を考案した．項書換え系の個々の計算状態を計算対象としている項の視覚化を用いて表し，その系列として表示する手法を書換え系列の視覚化と呼び，計算の状態をグラフの一つのノードとみなし，書換え回数に基づく階層グラフで，ある項を根とするすべて

の書換え関係を表す手法を書換え関係の視覚化と呼ぶ.

インタフェースの視覚化は単なるコマンドに対するグラフィカルユーザインタフェースではなく，視覚化された項や書換え系列，書換え関係などを操作するためのインタフェースを視覚的に実現することである．この実現により，項の木構造図式に対し，書換え位置などの計算に対する指示や，省略の場所などの表現手法に関する指示などの直接的な操作を可能にする.

数値情報の視覚化は計算の過程から抽出される項や計算に関する様々な数値情報を折れ線グラフなどを用いて視覚的情報として表示する．数値情報としては，項のサイズ，深さ，幅，リデックスの数など，様々な情報が挙げられる.

この視覚的支援手法に基づいて実現された支援環境上で，ハノイの塔の例題を用いて具体的な視覚的支援の実際を示し，本論文で提案する視覚的手法が項の構造の理解・解析や変換着目点の検出に対する支援手法として有効であることを確かめた.

(2) 視覚的環境の実現

先に提案した視覚的支援手法に基づいて具体的な視覚的支援環境をグラフィカルユーザインタフェースを持つ視覚的項書換え支援環境 TERSE(Term Rewriting Support Environment) として実現した．この支援環境ではモデルや視覚化の手法の変更に柔軟に対応できるような実現が必要となる．そこで，本論文では実現に用いる言語として近年注目されている関数型言語 Standard ML(SML)[67] を並列化した Concurrent ML(CML)[77] および eXene[76] ライブラリを採用した．環境の設計においては Smalltalk-80[15] の MVC(モデル・ビュー・コントローラ) 構造 [49] に基づくモジュール分割を行なった．また，項書換え系を直接にモデルとして扱わず，視覚化の対象として抽象木を導入した．項と抽象木の間ではモデルの変換を行なう.

(3) 項書換え系の視覚化に基づく可換則変換の評価

本論文で提案した項書換え系の解析・検証・変換のための視覚的支援手法，およ

びその実現である TERSE が，項書換え系のプログラム変換に関する研究に対し，実際に適用できることを例証するための研究を行なった．

具体的には，可換な演算の引数を入れ換えるという単純な変換を用いた項書換え系効率化変換手法を提案する．まず，可換則変換だけによっても項書換え系の効率を大きく改善できることを実例で示し，その上で可換則変換の適用戦略のアルゴリズムを提案する．このアルゴリズムは項の間の単純化順序がパラメータとなっており，この順序は必要に応じて入れ換えることができる．また，必須呼びの観点から変換の効果を評価し，実際に効果があることを例証する．

可換則に基づく効率化変換は，計算の視覚化を用いることにより，変換前と変換後の書換え系列の違いを明確に理解することができる．また，数値情報の視覚化により，変換の前後で書換え回数（時間）においても，項の大きさ（空間）においても効率が良くなることが確認できる．これにより，本論文で提案する項書換え系の視覚化手法，およびその実現が，効率化変換のためのアルゴリズムの発見のために有用であることが確認できた．

1.5 本論文の構成

本論文は 2 章 (基本的な概念)，3 章 (項書換え系の解析・検証・変換のための視覚的支援手法)，4 章 (視覚的支援手法に基づく項書換え支援環境 TERSE の実現)，5 章 (可換則に基づく項書換え系の変換と視覚化に基づく実行効率の評価) から構成される．

まず，2 章において，本論文で扱う項書換え系の基礎となる多ソート代数と項書換え系の基礎的な性質について述べる．

3 章では，項書換え系の解析・検証・変換のための視覚的支援手法について，項書換え系の計算の詳細な検討から，具体的な視覚化手法について述べる．

4 章では，項書換え系の視覚的環境の計算機上への実現手法について述べる．関数型言語 Standard ML を用いた視覚的支援環境がどのように構築されている

のかについて述べる.

5章では，本論文で実現したシステムを実際の項書換え系の研究に応用し，その有用性の評価を行なうための実例として，可換則変換に基づく項書換え系の効率化変換について述べる.

最後に6章において，本論文のまとめと今後の展望について述べる.

第 2 章

基本的な概念

本章では，本論文で用いる基本的な概念について説明する．より詳細な項書換え系についての定義は [22, 44] 等を参照されたい．

2.1 シグニチャ，項，出現

【定義 2.1.1 】 シグニチャ (*signature*)

シグニチャ $\Sigma = \langle S, F \rangle$ はソートの集合 S，関数記号の集合 F によって定められる．F 上には関数記号のソートを示す関数 $sort : F \to S$ と，アリティ (引数のソート, 数) を示す関数 $arity : F \to S^*$ が定義されている．ここで S^* はソート S の元からなる有限系列の集合である．また空系列は ε と表す．関数記号 $f \in F$ において，$sort(f) = s$，$arity(f) = s_1 s_2 \ldots s_n$ であるとき，f は第 i 引数にソート s_i の値をとり，ソート s の値を返す．また，$arity(f) = \varepsilon$ のとき，f は定数記号と呼ぶ．

【定義 2.1.2 】 項 (*term*)

$\Sigma = \langle S, F \rangle$ をシグニチャ，V を S ソート付き変数記号の可算無限集合とする．V の変数 x のソートを $sort(x)$ で表すとする．Σ 項の集合 $T_\Sigma(F, V)$ は以下の 2 条件を満たす最小の集合である．また，同時に Σ 項にソートを割り当てる写像 $\overline{sort}$ を定める．

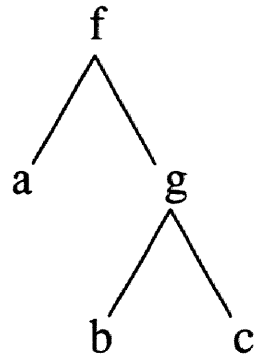

図 2.1: 項 $f(a, g(b, c))$ の木構造図式による表現

1. $x \in V$ ならば,

$$x \in T_\Sigma(F, V)$$
$$\overline{sort}(x) \stackrel{def}{=} sort(x)$$

2. $f \in F$, $arity(f) = s_1 s_2 \ldots s_n$,
$t_1, \ldots, t_n \in T_\Sigma(F, V)$ かつ, $\overline{sort}(t_i) = s_i$ $(1 \leq i \leq n)$ ならば,

$$f(t_1, \ldots, t_n) \in T_\Sigma(F, V)$$
$$\overline{sort}(f(t_1, \ldots, t_n)) \stackrel{def}{=} sort(f)$$

以下では，混乱のない限り $\overline{sort}$ を $sort$ と書くことにする．また，文脈から Σ や F, V が明らかな場合には Σ 項集合 $T_\Sigma(F, V)$ を項集合 $T(F, V)$ や，単に T と書く．

項 $t \in T$ は木構造を用いて図式化すると分かりやすい．例えば，項 $f(a, g(b, c))$ は図 2.1のように表現することができる．

変数を含まない項を基礎項 (*ground term*) と呼び，基礎項の集合は $T_\Sigma(F, \emptyset)$ で表す．また，項 t に含まれる変数記号の集合を $V(t)$ で表す．

次に，Σ 項集合 $T_\Sigma(F, V)$ をソートに分割する．ソート $s \in S$ の項集合 T_s は

$$T_s = \{t \mid t \in T_\Sigma(F, V), sort(t) = s\}$$

と定義する．

【定義 2.1.3 】出現 (*occurence*)

項 t の出現の集合 $O(t) \subseteq N^*$, および出現 $u \in O(t)$ に対する項 t の部分項 t/u を以下のように帰納的に定義する.

1. $\varepsilon \in O(t)$
   $t/\varepsilon = t$

2. $u \in O(t_i), 1 \leq i \leq n$のとき, $iu \in O(f(t_1, \ldots, t_n))$
   $f(t_1, \ldots, t_n)/iu = t_i/u$

出現 u, v に対して w が存在して $v = uw$ となるとき $u \preceq v$ とする. もし $w \neq \varepsilon$ ならば $u \prec v$ と書く. また $v = uw$ のとき v/u は w を表す. $u \not\preceq v, v \not\preceq u$ の時は $u \mid v$ と書く. 出現の深さは u の長さであるとし, $|u|$ で表す. 項 t, t' に対し $t[u : t']$ は項 t の出現 u における部分項を t' で置き換えた項を表す.

2.2 項書換え系

以下では, $\Sigma = \langle S, F \rangle$ をシグニチャとし, V を S ソート付き変数集合, $T = T_\Sigma(F, V)$ とする.

【定義 2.2.1 】項書換え系 (*term rewriting system*)

書換え規則 (*rewrite rule*) はそれぞれ左辺, 右辺と呼ばれる 2 項 $\alpha, \beta \in T$ から成り, $\alpha \to \beta$ と書く. ただし, β に含まれる変数はすべて α に出現していなければならない. また, 両辺のソートは同一である必要がある. すなわち, $V(\beta) \subseteq V(\alpha), sort(\alpha) = sort(\beta)$. 項書換え系 R は以下に示すように書換え規則の有限集合により定義される.

$$R = \{\alpha_i \to \beta_i \mid 1 \leq i \leq n\}$$

また, i 番目の規則を $r_i = \alpha_i \to \beta_i \in R$ と表す.

まず，項 $t \in T$ の先頭の関数記号（もしくは変数記号）を取り出す関数 Top を定義する．

$$Top(t) = \begin{cases} f & t = f(t_1, \ldots, t_n) \text{ のとき} \\ x & t = x \in V \text{ のとき} \end{cases}$$

項書換え系 R に対し，被定義関数記号の集合 D，構成子関数記号の集合 C を以下のように定める．

$$D = \{f \mid \forall \alpha_i \to \beta_i \in R, f = Top(\alpha_i)\}$$
$$C = F - D$$

すなわち，$F = C \cup D$ かつ $C \cap D = \emptyset$ である．また，項書換え系の規則の左辺にそれぞれの被定義関数記号が高々一回しか出現しない場合，その項書換え系は構成子系と呼ばれる．以下，本論文では主に構成子系である項書換え系を扱う．ソート s の構成子関数記号の集合は C_s と表す．

記述を簡単にするため以下の記法を用いる．

1. $ConOcc(t) = \{u \mid u \in O(t), Top(t/u) \in C\}$

2. $FunOcc(t, f) = \{u \mid u \in O(t), Top(t/u) = f\}$

3. $VarOcc(t, x) = \{u \mid u \in O(t), Top(t/u) = x\}$

4. $Def(t) = \{f \in D \mid u \in O(t), Top(t/u) = f\}$

$ConOcc$, $FunOcc$, $VarOcc$ は，それぞれ項の構成子関数記号，関数記号，変数記号の出現の集合である．また，Def は項に出現する被定義関数記号の集合である．

2.2.1 項書換え系の計算

項書換え系の計算を定義するため，まず代入の定義を行なう．

【定義 2.2.2 】代入 (*substitution*)

$\Sigma = \langle S, F \rangle$ をシグニチャ，V を S ソート付き変数集合とする．変数と項の対の有限集合 $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$ は，$x_i \neq x_j (i \neq j)$ かつ $sort(x_i) = sort(t_i)$ であるとき，代入であるという．項 $t \in T$ に対して，$\sigma[t]$ を次のように定義する．

$$\sigma[t] = \begin{cases} x & t = x \in V \text{ かつ } x \neq x_i (i = 1, \ldots, n) \text{ のとき} \\ t_i & t = x_i \text{ かつ } x_i/t_i \in \sigma \text{ のとき} \\ f(\sigma[t_1], \ldots, \sigma[t_n]) & t = f(t_1, \ldots, t_n) \text{のとき} \end{cases}$$

【定義 2.2.3 】マッチング (*matching*)

項 u に対して項 t がマッチングするとは，項 t に対して適当な代入 σ が存在して，$\sigma[t] = u$ となることである．また，適当な代入 σ により，$\sigma[t] = \sigma[u]$ となることを単一化 (*unification*) といい，項 t と項 u は単一化可能であるという．

例えば，$t = f(x), t' = f(g(b))$ とすると，代入 $\sigma = \{x/g(b)\}$ が存在して，$\sigma[t] = t'$ となる．このマッチングと次に定義する文脈により，項書換え系における項の書換えが定義される．

【定義 2.2.4 】文脈 (*context*)

ソート $s \in S$ に対して $\Box_s$ は F, V に含まれない特別な記号とする．項 $C \in T_\Sigma(F \cup \{\Box_s \mid s \in S\}, V)$ は，$\Box_s$ がただ 1 つ出現するときを文脈と呼び，$C[\Box_s]$ と表す．$\Box_s$ をソート s の項 t $(sort(t) = s)$ で置き換えたものを $C[t]$ と表す．例えば，項 t に部分項 t' が存在する場合，ある文脈 C が存在して $t = C[t']$ となる．t' の t における出現が $u \in O(t)$ であった場合，文脈 $C[\Box_s]$ の $\Box_s$ を変数 x に置き換えた項 $C[x]$ は $t[u : x]$ と一致する．

【定義 2.2.5 】リデックス (*redex*)

項書換え系 R の規則 $\alpha_i \rightarrow \beta_i \in R$ に対し，左辺項 α_i に適当な代入 σ を施した項が項 t に一致するとき，すなわち，$\sigma[\alpha_i] = t$ であるとき，t をリデックスと呼ぶ．

項書換え系 R に対し，項 t が部分項にリデックスを持つ場合，その出現と書換え規則の対の集合を $Red_R(t)$ で表す．

$$Red_R(t) = \{(u, r_i) \mid \exists \sigma, t/u = \sigma[\alpha_i], r_i = \alpha_i \to \beta_i \in R\}$$

R が明らかな場合，$Red(t)$ と書く．部分項にリデックスを含まないリデックスを最内リデックス，他のリデックスの部分項にならないリデックスを最外リデックスと呼ぶ．また，$Red(t)$ が空集合の項 t を正規形 (*normal form*) と呼ぶ．

リデックス s_1, s_2 が $s_1 = t/u, s_2 = t/v, v = uw$, $(u, r_i), (v, r_j) \in Red_R(t)$ であるとき，$w \in O(\alpha_i) - VarOcc(\alpha_i)$ ならば，リデックス s_1 と s_2 は重なっているという．

【定義 2.2.6 】書換え関係 (*rewriting relation*)
書換え規則 $r_i = \alpha_i \to \beta_i \in R$ による項 t の書換え関係を定義する．t の出現 u における部分項 $s = t/u$ と α_i が代入 σ によりマッチングしているとする．すなわち $t = C[s] = C[\sigma[\alpha_i]], (u, r_i) \in Red_R(t)$ である．このとき，項 s を $\sigma[\beta_i]$ で置き換えることにより t から $t' = C[\sigma[\beta_i]]$ を得ることを，項 t のリデックスの出現 u の規則 r_i による書換えと呼び，$t \xrightarrow{R}_{(u,r_i)} t'$，もしくは，単に $t \to_{(u,r_i)} t'$ と書く．適当な $(u, r_i) \in Red_R(t)$ に対して $t \to_{(u,r_i)} t'$ となる時 $t \to t'$ と書き，t と t' の間には書換え関係 $\to$ があるという．書換え関係

$$t_0 \to t_1 \to t_2 \to \ldots$$

による項 t_i の系列を書換え系列 (*rewriting sequence*) と呼ぶ．書換え関係の反射的推移的閉包を $\xrightarrow{*}$ と書く．重なりのないリデックスの集合 $\{(u_1, r_{k_1}), \ldots, (u_m, r_{k_m})\}$ を同時に書換える場合を，$t \xrightarrow{R}_{\{(u_1,r_{k_1}),\ldots,(u_m,r_{k_m})\}} t'$ と書く．ここで，$u_i \not> u_j$ $(i < j)$ を満たす時，t' は $t \xrightarrow{R}_{(u_1,r_{k_1})} t_1 \to \cdots t_{m-1} \xrightarrow{R}_{(u_m,r_{k_m})} t'$ で得られる．

一般に項は複数のリデックスの出現を持つ. そのため, 計算を行なうためには, 書換えるリデックスを選ぶ必要があり, その選択のアルゴリズムは戦略と呼ばれる.

【定義 2.2.7 】 戦略 (*strategy*)

項 t に対するリデックスの出現と規則の対の集合 $Red_R(t)$ から, 書換え対象のリデックスの出現の集合を一意に選ぶアルゴリズムを戦略と呼び, Str で表す. ただし $Str(Red_R(t))$ で表されるリデックスは互いに重ならないとする.

最内リデックス, 最外リデックスの出現を選ぶ戦略は, それぞれ最内戦略, 最外戦略と呼ばれる. 最も左の出現を選ぶアルゴリズムは最左戦略と呼ばれる. また, リデックスの出現を一つだけ選択する戦略を逐次戦略と呼び, 複数の重ならないリデックスを同時に選択するものを並列戦略と呼ぶ.

【定義 2.2.8 】 項書換え系の計算

項書換え系 R の項 t_0 に対する計算は戦略に基づく項の書換え系列で表される. これは戦略を Str としたとき,

$$t_i \xrightarrow{R}_{Str(Red_R(t_i))} t_{i+1} \quad (i \geq 0)$$

となる t_i の系列である.

必須リデックス (*needed redex*) とは, そのリデックスを書換えないと正規形に到達できないようなリデックスである. また, 必須リデックスを書換えることを必須呼び (*needed reduction*)[23, 60] という. 任意の時点でリデックスが必須リデックスであるかどうかを示すことはできないが, 停止する計算である場合, 必ず必須リデックスを示すことができる.

【定義 2.2.9 】 出現の残余 (*residual*)

規則 $\alpha \to \beta \in R$ によって項 t が出現 $u \in O(t)$ で書換えられて項 t' が得られてい

るとき，出現 $z \in O(t)$ の残余 $Res(z)$ を以下のように定義する．

$$Res(z) = \begin{cases} \{z\} & z \prec u \text{ もしくは } z \mid u \text{ であるとき} \\ \{uv'w \mid \beta/v' = x\} & \\ \qquad z = uvw \text{ かつ } \alpha/v = x \in V \text{ であるとき} & \\ \{u\} & \text{それ以外} \end{cases}$$

出現 z の残余 $Res(z)$ は書換えによる出現 z の移動先の出現の集合を表す．出現 u の書換えが出現 z に関係無い場合，残余は z になる．規則の左辺の変数 x にマッチングしている部分項を出現 z が指しており，右辺に複数の x の出現があれば，残余はコピーされた複数の出現の集合となる．逆に規則の右辺に x の出現がない場合，残余は空集合になる．それ以外は出現 z がリデックスの部分項を指している場合で，残余は書換えた部分項の出現 u になる．

2.2.2 項書換え形の諸性質

以下に項書換え系に関する諸定義とそれらの性質を述べる．これらの性質の成立に関する視覚化により，項書換え系の直観的理解が可能になる．

- 停止性 (*termination*)

  項書換え系 R が停止性を満たすとは， R において無限の書換え系列が存在しない事である．

- 合流性 (*confluence*)

  項書換え系 R が合流性を満たすとは，任意の項 p， q, r において $p \xrightarrow{*} q$， $p \xrightarrow{*} r$ ならば，適当な項 s が存在して， $q \xrightarrow{*} s$， $r \xrightarrow{*} s$ となる事である．

- 完備性 (*completeness*)

  項書換え系が停止性と合流性を満たす時，その項書換え系は完備 (*complete*) であるという．

- 無曖昧性 (*non-overpping, non-ambiguous*)

  項書換え系 R が無曖昧であるとは，任意の書換え規則 $r_i, r_j \in R$ の左辺 α_i, α_j がお互いの部分項と単一化可能でない事をいう．

- 危険対 (*critical pair*)

  ある項が重なりのあるリデックスを持ち，各々のリデックスを書換えた結果得られた二つの項が異なる場合，この二項を危険対と呼ぶ．項書換え系が曖昧であると危険対が存在する．

- 左線形性 (*left linear*)

  項書換え系が左線形性を満たすとは，すべての規則の左辺に，同じ変数が２度以上出現しない事である．

- 正則性 (*regular, othogonal*)

  項書換え系が無曖昧で左線形の場合，その項書換え系は正則であるという．

これらの定義に基づき，以下のような様々な重要な性質が明らかとなっている．

【性質 2.2.1 】 [22]

左線形かつ無曖昧な項書換え系（正則）は，合流性をみたす． □

また，

【性質 2.2.2 】

項書換え系が合流性を満たせば，項に正規形が存在するなら必ず唯一である． □

正規形が存在すれば必ずそれを求めることができる書換え戦略のことを正規化戦略と呼ぶ．すると，次の性質がいえる．

【性質 2.2.3 】

左線形かつ無曖昧な項書換え系では，必須呼びや，最外並列戦略が正規化戦略となる．□

また，停止性を満たす項書換え系においては，以下の定理がある．

【定理 2.2.1 】 *Knuth, Bendix*(1970)
停止性を満たす項書換え系において，全ての危険対が同じ正規形を持てば，その項書換え系は合流性を持つ．□

項書換え系が完備であれば，次の重要な二つの性質が言える．

【性質 2.2.4 】
項書換え系が完備であれば，全ての項について正規形が唯一に定まる．□

本論文の議論では特に断わらない限り，正則な項書換え系を対象とする．

2.2.3 再帰経路順序

項書換え系の停止性を検証するための手法として，単純化順序を用いる手法がある．

【定義 2.2.10 】単純化順序 (*simplification ordering*)
以下の性質を満たす半順序 $\succ$ を項の上の単純化順序と呼ぶ．

1. $t \succ s$ ならば $f(\ldots,t,\ldots) \succ f(\ldots,s,\ldots)$ (単調性)
2. $f(\ldots,t,\ldots) \succ t$ (部分項性)

単純化順序は整礎 (*well-founded*)[†1] な順序であることが示されており，この単純化順序を用いて，項書換え系のすべての規則に (左辺)$\succ$(右辺) という順序をつけることが可能ならば，その項書換え系が停止性を持つことを示すことができる．

比較的適応範囲が広い単純化順序として，再帰経路順序 (*recursive path ordering*)[10] が挙げられる．その定義の前に，多重集合上の半順序である多重集合順序を定義する．

[†1] 無限減少列が存在しない．

【定義 2.2.11 】多重集合順序 (*multi-set ordering*)

半順序集合 (S,$\succ$) が与えられた時，多重集合 $M(S)$ 上の順序 $\succ\!\!\succ$ は次のように定義される．

多重集合 M, M' に対して $M \succ\!\!\succ M'$ であるのは以下の条件を満たす時である．

$$\begin{aligned}&\text{多重集合} Y, \text{空でない多重集合} X \subseteq M \text{が存在し，}\\ &M' = (M - X) \cup Y \quad \text{かつ}\\ &\forall y \in Y, \exists x \in X, x \succ y\end{aligned}$$

【定義 2.2.12 】再帰経路順序 (*recursive path ordering*)

関数記号の集合 F に半順序 $\succ$ が与えられた時，再帰経路順序 $\succ_{rpo}$ は次のように定義される．ここで $\succ\!\!\succ_{rpo}$ は $\succ_{rpo}$ による多重集合順序である．ただし，$t \succeq t'$ は $t = t'$ または $t \succ t'$ であることを意味する．

$$s = f(s_1, \ldots, s_m) \succ_{rpo} g(t_1, \ldots, t_n) = t$$

が成り立つのは以下のいづれかが成り立つときである．

(1) $s_i \succeq t$ となる $i(= 1, \ldots, m)$ が存在する．

(2) $f \succ g$ かつ $s \succ_{rpo} t_j$ がすべての $j(= 1, \ldots, n)$ で成り立つ．

(3) $f = g$ かつ $\{s_1, \ldots, s_m\} \succ\!\!\succ_{rpo} \{t_1, \ldots, t_n\}$ が成り立つ．

先行順序と呼ばれる関数記号上の半順序を与えることにより，再帰経路順序は定義される．そのため，先行順序を定めることが，項書換え系の停止性判定問題を解くために必要となる．

2.2.4 項書換え系の実際例

項書換え系の書換え対象である項は，関数記号と変数記号により定められる．例えば正の整数の階乗を求める項書換え系の規則は以下のようになる．

$R = \{$

1) $add(0, x) \quad \rightarrow x,$

2) $add(s(x), y) \rightarrow s(add(0, y)),$

3) $mult(0, x) \quad \rightarrow 0,$

4) $mult(s(x), y) \rightarrow add(mult(x, y), y),$

5) $fact(0) \quad \rightarrow s(0),$

6) $fact(s(x)) \quad \rightarrow mult(s(x), fact(x)),$

$\}$

この項書換え系を用いて 1 の階乗を求める場合を以下に示す.

1. 初期項として $fact(s(0))$ を用いる.

2. 規則 6 にマッチングし, $mult(s(0), fact(0))$ に書換えられる.

3. $mult$ の第 2 引数の $fact(0)$ が規則 5 にマッチングして,
   $mult(s(0), s(0))$ に書換えられる.

4. 規則 4 にマッチングして $add(mult(0, s(0)), s(0))$ に書換えられる.

5. 部分項 $mult(0, s(0))$ が規則 3 にマッチングして,
   $add(0, s(0))$ に書換えられる.

6. 規則 1 にマッチングして $s(0)$ に書換えられる.

7. これ以上マッチングする規則がなくなり, この計算は停止する.

第 3 章

項書換え系の解析・検証・変換のための視覚的支援手法

3.1 はじめに

項書換え系は，マッチングと書換えによって計算が進む単純な計算モデルであり，代数的意味論に基づいた豊かな理論的成果を持つ．しかし，実際に項書換え系に対し，形式的アプローチに基づく検証や解析・変換を行なう場合，具体的にどの部分に形式的手法を適用させるべきか明らかでない場合が多い．本章では1章で述べた4種類の視覚化の概念に基づいて項書換え系を総合的に視覚化することによって，項書換え系の直観的理解・洞察を深めるとともに，その解析・検証・変換を容易にする視覚的支援手法を提案する．この手法により，これまでのテキスト環境では困難な，図を用いた高度な解析や検証に対する支援，直観的洞察による変換着目点の発見などが可能になる．また，新たな検証・変換手法や効率的な書換え戦略などの開発が，この項書換え系の総合的な視覚化により触発されることも期待される．

本章ではまず，項書換え系の解析・検証・変換の支援のために，計算・操作の両面から項書換え系の視覚化を次の4種類に分類する．

- 項の視覚化

- 計算の視覚化
- インタフェースの視覚化
- 数値情報の視覚化

項の視覚化は，項が木構造で表されることを利用して，木構造図式上に様々な情報を付加する．項に含まれるそれぞれの関数記号が持つ情報を色や形状の違いで表すことに加え，部分木の省略や正規形の抽象化を用いて変換した図により項の構造の直観的理解を可能にする．計算の視覚化は，項書換え系の計算を計算状態と関係によって捉え，「書換え系列の視覚化」と「書換え関係の視覚化」に分類される．書換え系列の視覚化は項の視覚化を用いて，項書換え系の書換え系列を木構造図式の系列として表し，計算状態間の関係の直観的理解や繰り返し構造の発見を可能にする．書換え関係の視覚化は，一つの項を根とする書換え関係の部分グラフである．インタフェースの視覚化は視覚化された項や計算に対する操作のためのインタフェースを視覚化することであり，マウスなどのポインティングデバイスを用いて実現され，視覚化された情報の直観的な操作を可能にする．数値情報の視覚化は項書換え系の計算過程から得られる項に出現するリデックスの数や，項のサイズ，幅，深さなど，様々な数値情報を折れ線グラフ等を用いて視覚的情報として表し，数値情報の変化を直観的に理解することを可能にする．これら 4 種類の視覚化手法により，項書換え系の総合的な視覚的支援が実現される．

次に，具体例としてハノイの塔の問題をとりあげ，この視覚的支援手法を用いて項書換え系の解析・検証・変換が的確に行なえることを示す．

また，これらの視覚的支援手法を実現した「視覚的項書換え支援環境 (TERSE: TErm Rewriting Support Environment)」 [38, 39, 40, 41] を計算機上に作成した．TERSE の実現に関しては次章で述べる．なお，本章の例はすべて TERSE を用いて実行している．

以下では，各々の視覚化についての詳細を述べ，次に，視覚的支援手法を用い

た実際の解析，変換例を示す．また，項の視覚化を拡張した，再帰経路順序の視覚化についても述べる．さらに，他のプログラミング言語における視覚的支援手法との比較により，本手法の位置付けを行なう．

3.2 項書換え系の視覚化

本論文では，1.2節で述べたようにソフトウェアの視覚化を (1) データの視覚化，(2) プログラムの視覚化，(3) 視覚的プログラミング，(4) アルゴリズムアニメーション，に分類する．(1) は，計算過程や計算結果のデータに関する視覚化手法であり，2 次元上における木構造図式やグラフ図形を用いた表現や，3 次元表現 [75, 46, 45] を用いるものがある．(2) は本来テキストなどで表されているプログラムを視覚的な図式で表すことであり，FE'92[93] のように視覚的な操作環境を与えたものなどがある．(3) は視覚的プログラミング言語 [64] や PAD[12] など，言語そのものが視覚的要素からなるものに代表される．(3) はアルゴリズムアニメーションとも呼ばれ [78]，一般に，アルゴリズムの動作をアニメーションを用いて表現し，アルゴリズムの動的な特徴の理解を支援することを目的としている．近年では宣言的記述を用いたアニメーションの定義 [91] などが研究されている．

本章では (1) ～ (4) の視覚化手法を用いて項書換え系の解析・検証・変換を支援するための総合的な視覚化を行なう．項書換え系（プログラム），計算，計算の対象（データ），操作（インタフェース）を直観的理解が容易な視覚的な形式で表現する．まず，項書換え系の視覚化を計算と操作の 2 面に分類する．項書換え系の計算は規則，計算状態，状態遷移からなり，規則および計算状態は項により表されるので項の視覚化により視覚化する．また，状態遷移の視覚化は計算の視覚化と呼ぶ．これらの視覚化された計算を視覚的に操作するためにインタフェースの視覚化を行なう．項にはリデックスの数などの数値情報も付随するので，この視覚化を数値情報の視覚化と呼ぶ．

本節では，以上のように項書換え系の視覚化を項の視覚化，計算の視覚化，イ

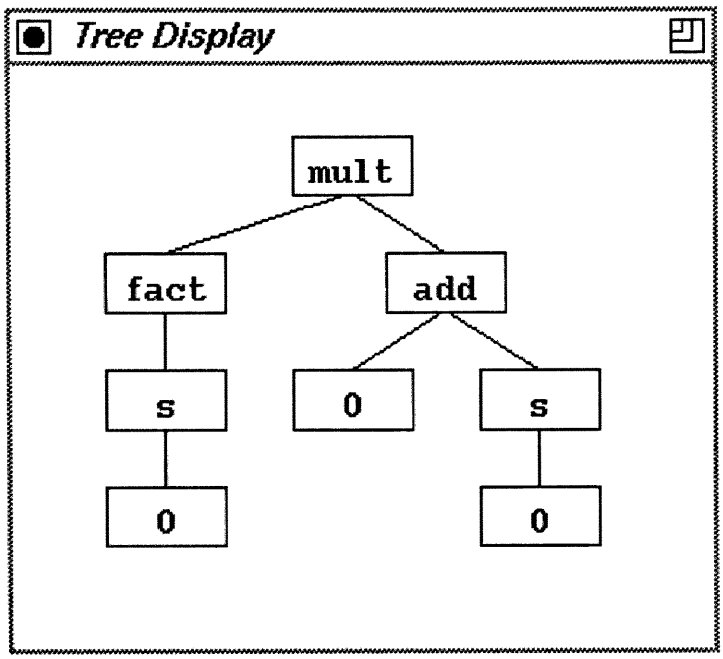

図 3.1: 単純な木表現

ンタフェースの視覚化，数値情報の視覚化の４種類に分類して説明する．

3.2.1 項の視覚化

項書換え系において，書換え対象の項はその時点における計算状態を表す．また，項書換え系は書換え規則の集合であり，規則は項の対であるため，項の視覚化はデータの視覚化であると同時にプログラムの視覚化でもある．

項の視覚化では，巨大な項の任意の部分の解析や，項の構造の直観的理解の実現を目的としている．項の構造解析を視覚的に行なう場合，項の情報をできるだけ多く視覚的に表現することが必要である．以下に，計算機上において項を解析する場合に必要な，あるいは有用な事項を示す．

(1) 項のソートの判別

(2) 構成子関数記号, 被定義関数記号, 変数記号の区別

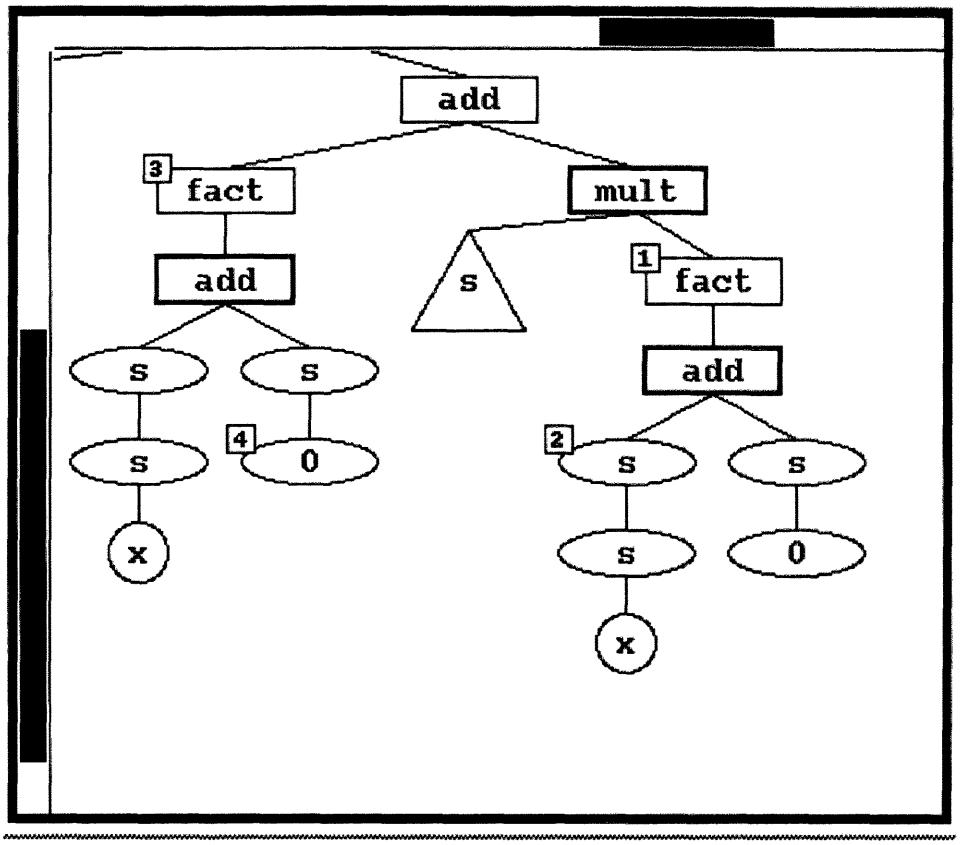

図 3.2: 視覚化された項

(3) リデックスの出現位置の判別

(4) ユーザが指示する出現位置の記録

(5) 画面上に表示できない巨大な項の解析

(6) 部分木の省略

(7) ユーザが指示する項の構造や正規形の簡単化

(1),(2) は基本的な項の性質であり，項がどのように構成されているかの解析に必要である．(3) は，書換え規則によって定まるリデックスの出現の確認に役立つ．(4) は，ユーザが注目している出現位置を記録し，部分項の書換えがどの部分に

影響を与えるか等の解析に役立つ．(5) は，計算機の制限された画面上で視覚化を行なうために必要とされる．(6),(7) は，複雑な項の構造を単純にすることを目的とし，ユーザの指示により，部分項やユーザが指示する構造，正規形などを簡単な構造に変換して表示する．ここで，項の構造や正規形の簡単化手法はユーザが指示する．例えば，自然数の 2 は項書換え形では一般に “$s(s(0))$” と表されるが，これを単に “2” と表示するのが正規形の簡単化である．また，構造パターンの例としては，リスト構造の項 “cons(a,cons(b,nil))” に対し，注目したい情報が，リストの内容ではなく長さのみである場合，リストの長さのみの情報をもつ項 “list(list(nil))” に簡単化して表現する．

図 3.1に示すような項の構造の単純な木表現は上記の要求事項を満たしているとはいい難く，より高度な表現が必要である．本論文では上記の要求を満たす表現として，以下のように項の視覚化を提案する．(1),(2),(3) の要求に対しては，各々のノードに対するソート，リデックス，構成子関数記号，被定義関数記号，変数記号などの情報をノードの形状や色の違いによって付加する．(4) については，ユーザの指示により出現位置のノードにマークをつけて判別する．項の書換えを行なった後も出現の残余の概念に基づいて出現位置のマークを移動させる．(5) に対しては，項の視覚化を仮想画面上で行ない，実際の画面上にはその一部を表示し，拡大, 縮小や表示部分の移動を可能にする．(6),(7) については，項に対して部分木の省略や，正規形や特定の構造を簡単化するために，項から項への変換を行なう．部分木の省略は，部分木を一つのノードとして表す変換を行なう．正規形や構造の簡単化のためには，各ソートの正規形を変換するための知識を持つデータベースを用いたり，構造の簡単化のための項書換え系を用いる．この簡単化のための項書換え系をユーザによって与えることにより，ユーザの意図を反映した簡単化が可能になる．特定の構造パターンを簡単化するためには，このパターンを視覚的に直接指示し，その結果を新たな簡単化のための項書換え系として用いる．これは，一種の視覚的プログラミングとみなすことができる．

項の視覚化の例を図3.2に示す．この図では各ノードの図形は構成子記号は楕円，被定義関数記号は長方形，変数記号は円で表され，リデックスは太い線で描かれている．また，部分木は三角形で表されている．ノードの左上に表示されている数字を中に含む小さな四角は出現位置のマークを意味している．任意の出現に対し番号によるマークづけを行なうことにより，巨大な項であっても注目している部分項を即座に発見できる．図形を用いた視覚化に加えて色を用いることによって，より多くの情報を視覚的に読みとることができる．

単純な木表現によって表された項 (図 3.1) と項の視覚化によって視覚化された項 (図 3.2) を比較すると，明らかに後者のほうからは視覚的に様々な情報を読みとることができる．単なる木表現だけでなく様々な視覚化技術を用いた項の視覚化により，項から得られる情報は増大し，従来のテキスト中心の処理系では困難であった直観的な解析が可能になる．

3.2.2 計算の視覚化

計算の視覚化は，項書換え系の計算の実行の様子を視覚化する．項書換え系の計算は次の２つからなるとみなせる．

- 各々の時点での実行状態
- 状態間の遷移

つまり項書換え系の計算は，状態から状態への遷移の繰り返しで表すことができる．項書換え系における実行状態とは，書換え対象の項に他ならない．つまり，前節で述べた項の視覚化を実行状態の視覚化とみなすこともできる．また，状態間の遷移は項の書換え関係によって定められる．一般に項書換え系においては，書換えが可能なリデックスは単一の項であっても同時に複数存在する．単一の書換え系列を考えた場合，書き換えるリデックスは書換え戦略によって選択される．

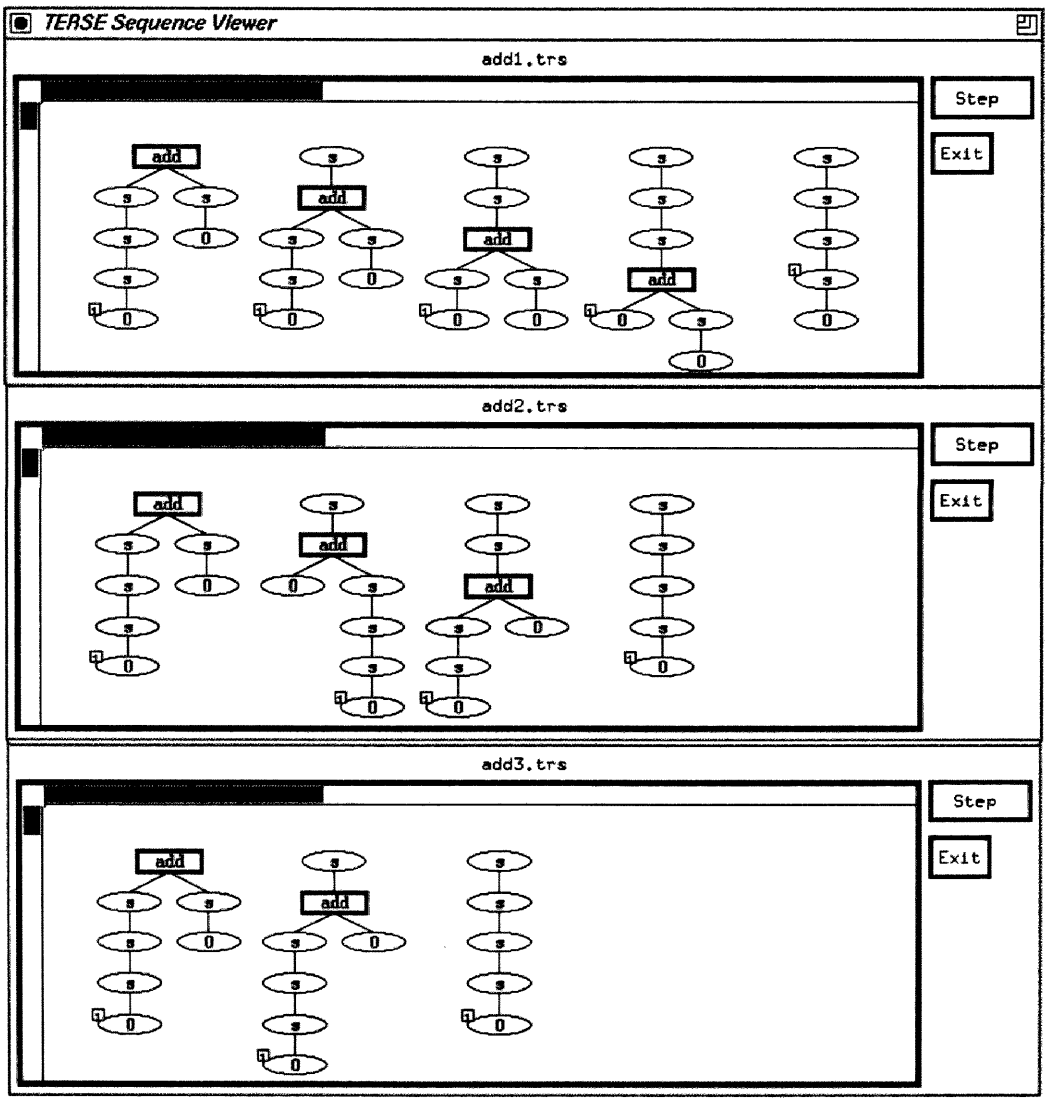

図 3.3: 書換え系列の視覚化

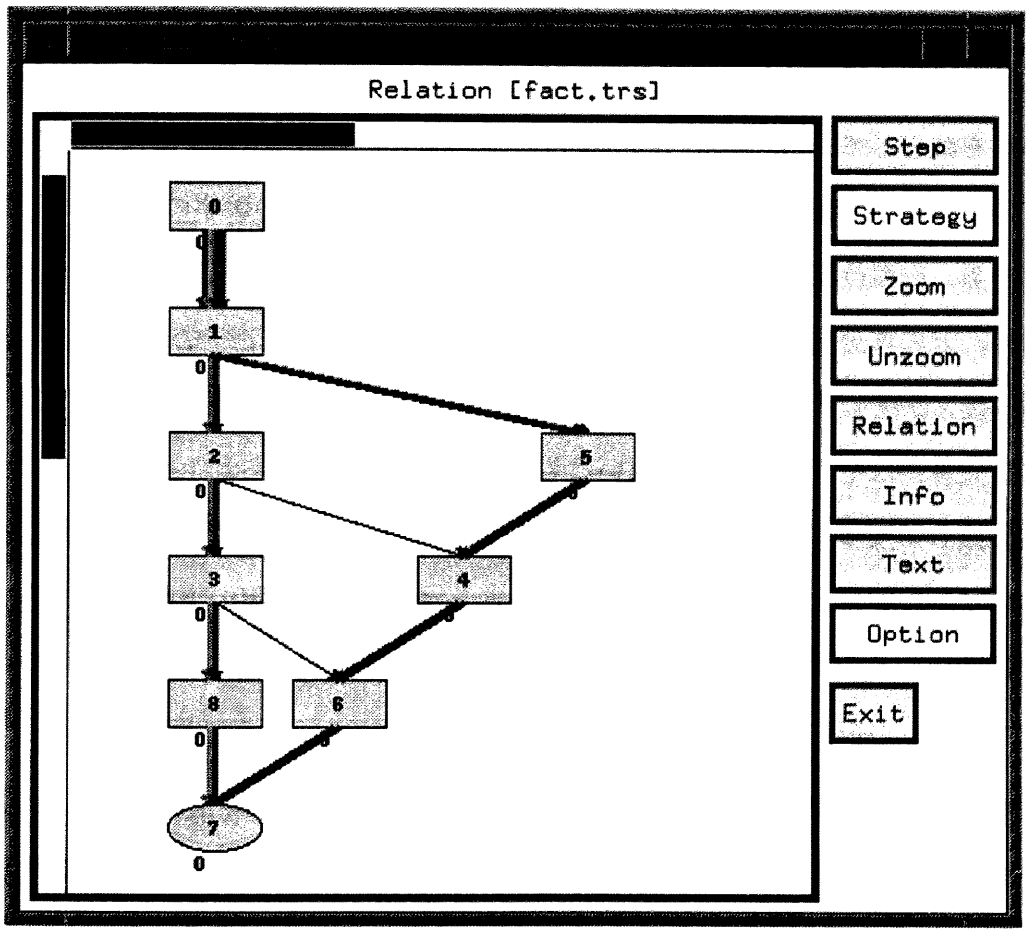

図 3.4: 書換え関係の視覚化

戦略を変更することにより，同じ初期状態からでも異なる複数の書換え系列を生成することができる．

計算の視覚化の一つの手法は，単一の戦略について各々の状態を順に並べて同時に表示することである．図 3.3は同じ項に対して異なる項書換え系を用いて計算を実行した場合の計算の視覚化を示している．これを書換え系列の視覚化と呼ぶ．この図を用いることにより各々の項書換え系の特徴が理解できる．例えば，上段の書換え系列は左の引数を順に減らしていくのに対し，最下段の系列は右の引数を減らしている．中央の系列では，左右の引数が書換えごとに入れ換えられていることが，出現位置のマーク (0 を表すノードの左上の数字の 1) により確認できる．出現位置のマークは出現の残余の概念によって書換えの前後を伝搬するため，どの部分項がどこへ書換えられたかを解析することに役立つ．また，上段の書換え系列の最後の書換えでは，出現位置のマークがついた項が書換えられて，書換え後にはその関数記号がないにもかかわらず，書換えられた結果得られた項に伝搬して残っていることが読みとれる．このように，書換え系列の視覚化を持ちいて，項書換え系の計算を視覚化することによって，計算の実行状態がどのように変化するかを直観的に把握できる．書換え系列の視覚化は，広い意味でのアルゴリズムの視覚化とみなせる．

一方，単一の項から同じ規則に基づき異なる書換え戦略を用いた複数の計算について，それらがどのような関係にあるかを表現するものを書換え関係の視覚化と呼ぶ．各々の実行状態である項に対し，その状態間がどのような戦略で遷移するかを有向グラフで表すことにより，計算状態間の関係を把握するとともに，どのような戦略が効率的な計算を可能にするかを直観的に理解できる．単一の項を根として，ノードは項を唯一に表し，エッジは書換え関係を表す．また，書換え回数に基づく階層グラフで表現することにより，項へ到達するための書換え回数や冗長な書換えを容易に理解することができる．図 3.4は書換え関係の視覚化の例である．ここでは，代表的な戦略（最外最左，最外最右，最内最左，最内最

右）については，エッジの色を変えて太く表示している．

3.2.3 インタフェースの視覚化

視覚化された項や計算を解析する場合，それらに対する様々な操作も視覚的に行なうことが望ましい．つまり，操作のためのインタフェースの視覚化を行なう．本節では，項書換え系に対する各々の操作に対してどのようにインタフェースを視覚化するかを提案する．

まず，項に対しては次のような操作が考えられる．

(1) 書換え規則の選択

(2) 書換え戦略の選択

(3) 書換え対象の項の選択

(4) 出現位置のマークづけ

(5) 部分木の表示に対する指示

(6) 項，規則の編集

(1),(2) は項の計算を制御する操作であり，すでに与えられている規則や戦略を選ぶ操作である．これらは，選択肢のリストの中からポインタによって選択するインタフェースによって実現する．(3) は戦略で指定できないような書換え対象の項を指示する操作である．対象とする項書換え系は無曖昧性を満たすので，リデックスの出現に対し一意に書換え規則が定まる．(4),(5) は項の視覚化に対しての指示であり，(4) は項の任意の出現について，指示により出現位置のマークをつける操作，(5) は任意の部分項を部分木として表示することを指示する操作である．(3),(4),(5) の操作は視覚化された項の図形に対しポインタを用いて直接指示するインタフェースを用いる．(6) は項や規則の入力や変更に対する支援を行な

う操作であり，視覚的構造エディタを用いて実現する．関数記号の入力では，その引数に応じて自動的に引数のノードを出現させることや，部分項のコピーなどを視覚的に行なう．

次に項書換え系やその計算に対する操作を挙げる．

(1) 書換え規則の編集

(2) 書換えの実行

(3) 書換え系列の選択

(4) ブレークポイントの設定

(1) の操作は例えばある項書換え系を左辺の先頭に現れる関数記号ごとに分割するような操作や，逆に分割された項書換え系を組み合わせる操作などを意味する．これらは規則のリストをポインタで選択することで行なう．また，編集結果の項書換え系のファイルへの保存やファイルからの読み込みのために，ファイル入出力ダイアログが必要である．(2) は 1 ステップもしくは数ステップの書換えの指示を行なう操作で，ボタンをポインタで押すような操作で実現する．書換えの前後では出現位置のマークや部分木を出現の残余の概念を用いて移動させる．(3) は書換え系列上の任意の状態へ現在の状態を遷移させる操作である．これにより，任意の時点での計算の再実行が可能になる．視覚化された書換え系列上でのポインタ指示で実現する．(4) は項書換え系のデバッグを行なうための操作であり，任意の規則にブレークポイントを設定し，その規則が用いられるまで書換えを行なう．これは規則の選択によって実現する．

3.2.4 数値情報の視覚化

項書換え系の計算の解析において，項の状態や計算の状況を数値で表すことがしばしば行なわれる．例えば項のサイズ，項の深さ，項の幅，リデックスの数やそ

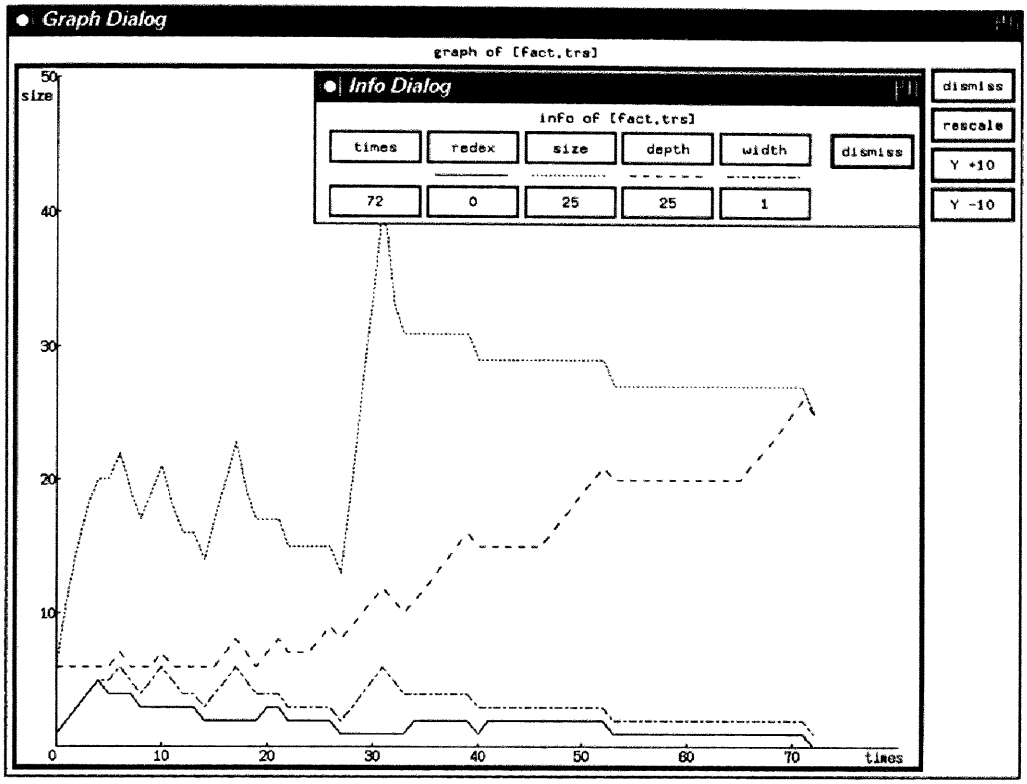

図 3.5: 数値情報の視覚化

れらの書換え全体に対する平均値，最大値，最小値などであり，正規形を得るための書換え回数などは最も重要な値の一つである．これら数値情報のグラフ化を行ない，直観的理解を助けることを数値情報の視覚化と呼ぶ．計算を実行した後にデータを収集するだけでなく，計算途中に動的にグラフ化することで，現在の状態の情報のみでなく，過去の状態との比較を直観的に行なうことができる．また，すでに定められている情報のみではなく，解析を行なうユーザが必要な値を随時計測できることが，より柔軟な解析を行なうためにも必要である．これを実現するために，計測値を得るためのプログラムを記述でき，随時支援環境に組み込めるような仕組みが必要となる．次章で述べる視覚的支援環境 TERSE は実行中にユーザが任意の計測関数を新しく追加する機能を持つ．

図 3.5はある階乗の計算における情報の視覚化である．横軸は書換え回数を表

し，縦軸は様々な数値情報を表す．折れ線グラフは上から項のサイズ (ノードの数)，項の深さ，項の幅，リデックスの数を表している．この図では計算が 72 回の書換えで終了し，この時点での項のサイズおよび深さが 25 であり，幅が 1 であることを示している．この視覚化された情報により，1 ステップずつの書換えにともない動的に項が変化していく様子を確認することができる．

3.3 視覚化による解析・検証・変換の支援

本節では，本論文で提案する視覚化手法を用いてどのように項書換え系の解析・検証・変換が行なえるかを具体例で示す．

例としてハノイの塔の解法を求める項書換え系を用いる．このゲームは，「n 本の塔の 1 本に上から小さい順に積まれた m 個の円盤がある．1 度に 1 個の円盤を別の塔に動かすという操作を繰り返してゴールの塔に全ての円盤を移動することが目的である．ただし，大きい円盤を小さい円盤の上に置くことは許されない．」というものである．以下の例では 3 本の塔 A,B,C に対し小さい順に $0, s(0), s(s(0))$ の 3 個の円盤が A にあるとして，これらの円盤を C に動かす場合の解を求める．
以下に書換え規則を示す．

```
var x,y,z,from,other,to;
1: hanoi(s(x),from,to,other)
      => do(hanoi(x,from,other,to)
            ,do(move(s(x),from,to)
                ,hanoi(x,other,to,from)));
2: hanoi(0,from,to,other)
                      => move(0,from,to);
end;
```

ここで，関数 $hanoi(x, from, to, other)$ は 「大きさが 0 から x までの円盤を塔

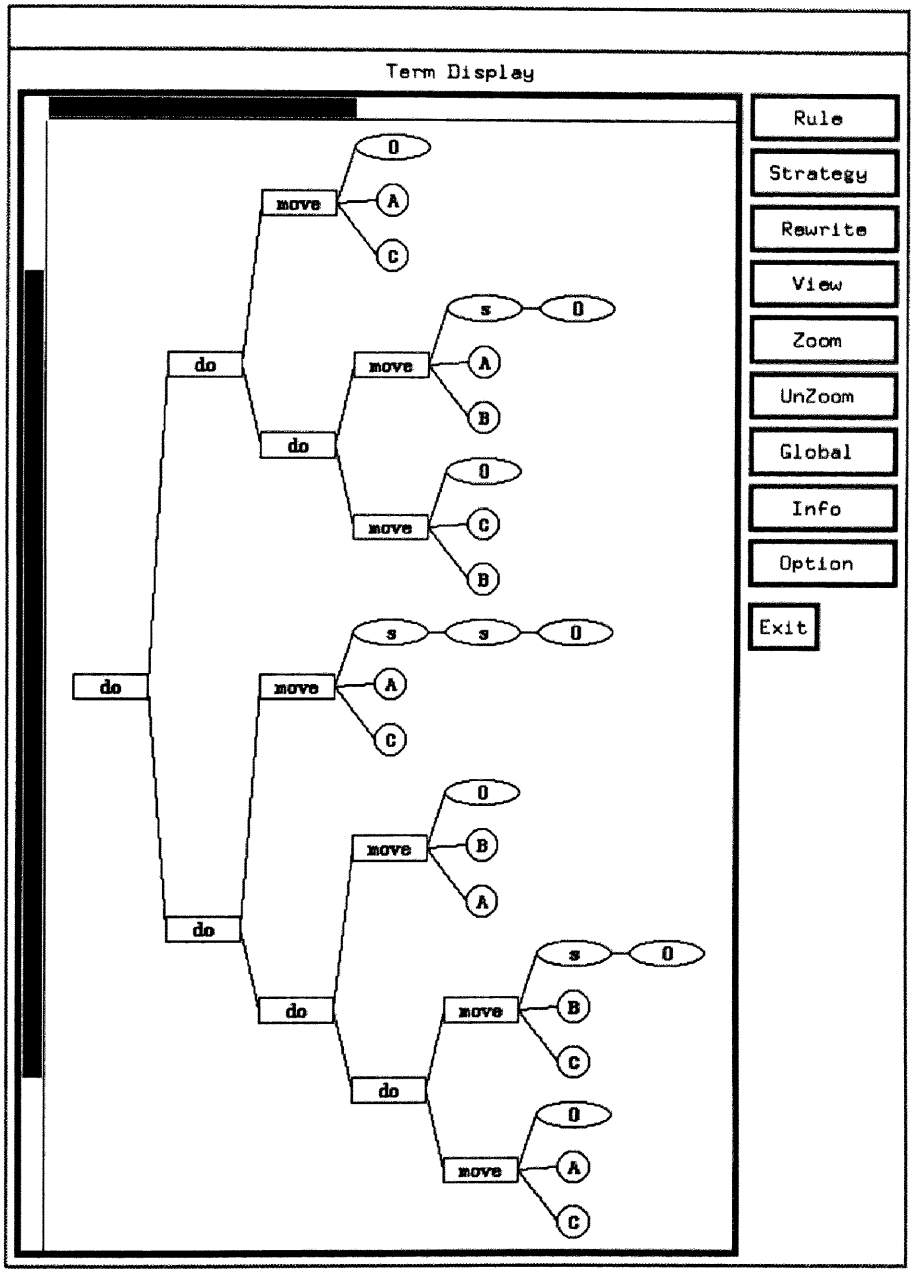

図 3.6: 2 文木構造を持つハノイの塔の解

$from$ から塔 to へ塔 $other$ を使って移動させる」という意味の関数である．また関数 $move(x, from, to)$ は「円盤 x を塔 $from$ から塔 to へ移動させる」，構成子 $do(x, y)$ は「x を実行してから y を実行する」という意味である．

項 $hanoi(s(s(0)), A, C, B)$ をこの項書換え系の入力として与える．すると 7 回の書換えの後，ハノイの塔の解として以下の正規形が得られる．

```
do(do(move(0,A,C)
      ,do(move(s(0),A,B),move(0,C,B)))
   ,do(move(s(s(0)),A,C)
      ,do(move(0,B,A)
         ,do(move(s(0),B,C)
            ,move(0,A,C)))))
```

この項を視覚化すると図 3.6になる[†1]．この解は図からもわかるように構成子関数 do による 2 分木構造を持つ．円盤の移動は根から do をだどり，第 1 引数 (上)，第 2 引数 (下) の順に関数 $move$ を実行することになる．この図の上から順に $move$ の実行することにより問題は解けるが，2 分木構造を持つ解では $move$ の実行順序が直観的ではなく理解が難しい．これは特にテキストを用いてこの項を表示する時に顕著となる．

そこで「移動の順序が理解しやすいように 2 分木構造をリスト構造へ変更したい」という要求がある．そのために構成子関数 do による構造をリスト構造にする関数 $list$ および，リストの接続を行なう関数 $append$ を導入する．以下に 2 分木構造をリストに変換する規則を示す．

```
3: append(nil,y)  =>  y;
4: append(cons(x,y),z)
       =>  cons(x,append(y,z));
```

[†1] この図は項の木構造を左から右へ表示している．関数記号の引数は上下の順で表示される．

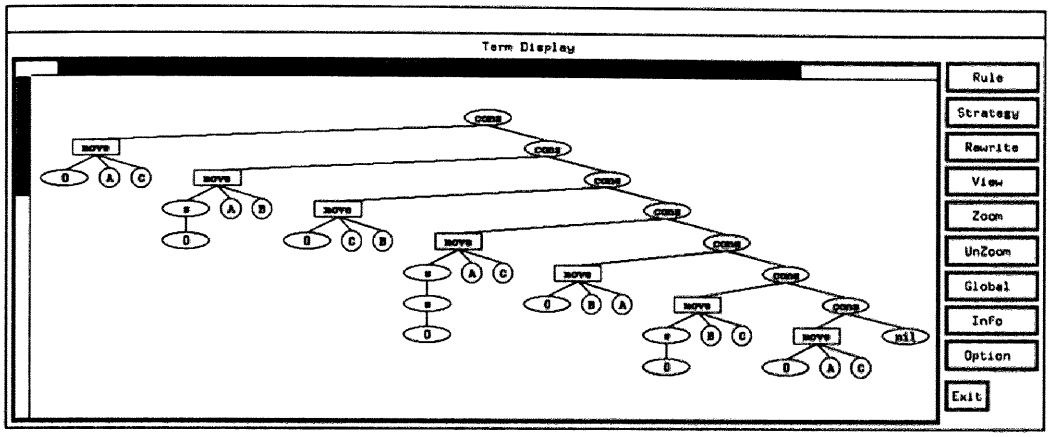

図 3.7: リスト構造を持つハノイの塔の解

```
5: list(move(x,y,z))
        => cons(move(x,y,z),nil);
6: list(do(x,y))
        => append(list(x),list(y));
```

これらの規則を追加した項書換え系の入力として $list(hanoi(s(s(0)), A, C, B))$ を与える．書換えは 34 回で停止する．計算の結果，得られた正規形がリスト構造になっていることを確かめるため，図 3.7に得られた項を視覚化する．この図から項がリスト構造を持つことが視覚的に確認できる．このように項の視覚化を行なうことにより，項の構造が目的とした形をしているかを図形の形状から視覚的に確かめることができる．

ところがこの計算は非常に効率が悪く，規則を追加する前は 7 回であった書換え回数が，追加後には 34 回となり，4 倍以上も多くなっている．そこで，この効率を改善するために計算の解析を行なう．最初に書換え系列の一部を視覚化 (図 3.8) する．この図では書換え対象のリデックスの出現が太い枠線で描画されている．図を見ると，$list(hanoi(x, f, t, o))$ という形，もしくは $list(do(hanoi(p), do(q)))$ の形の繰り返しがあることが，リデックスの出現を順に追うことにより直観的に

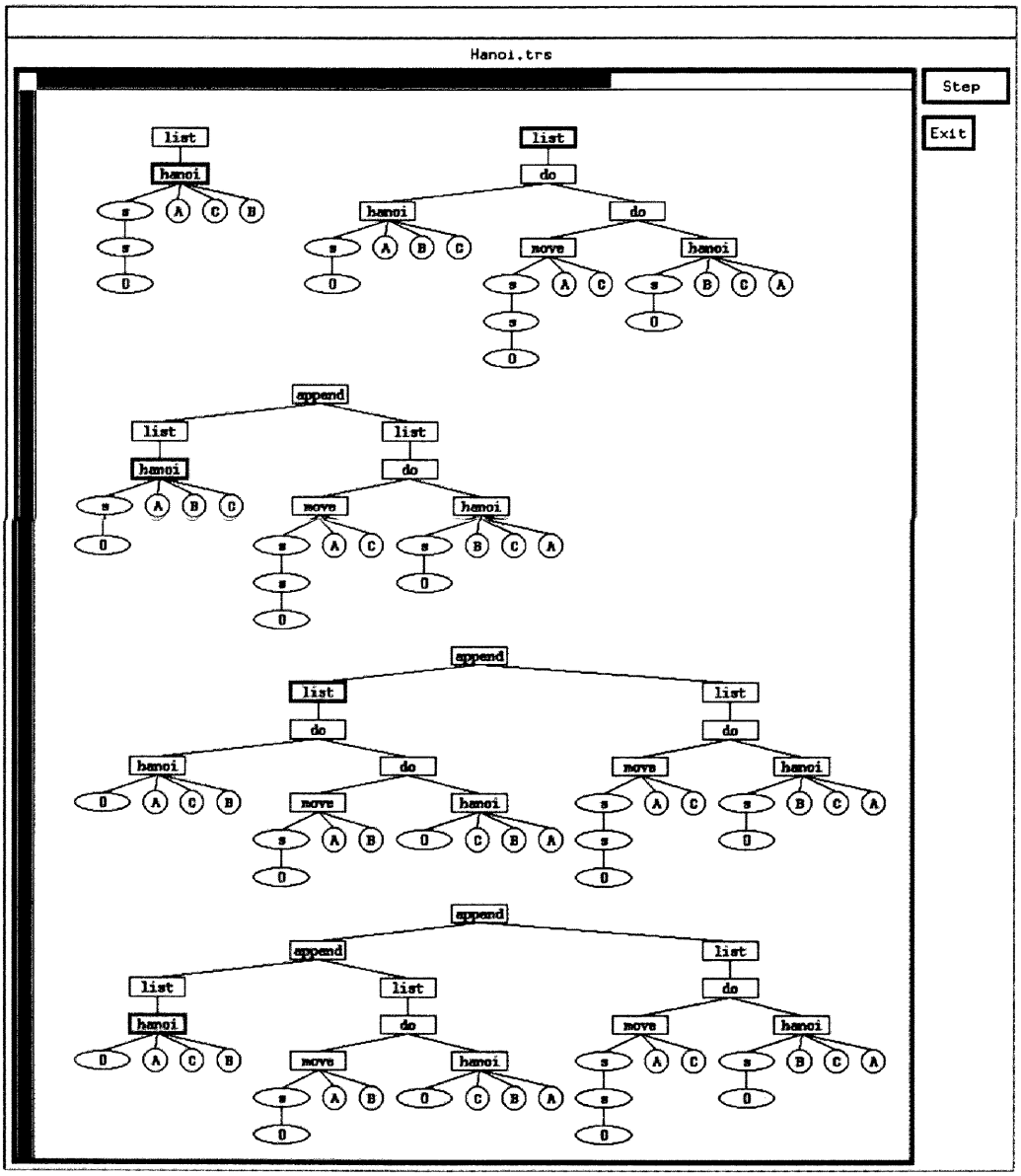

図 3.8: ハノイの塔の計算の視覚化

発見できる．視覚化された書換え系列に対し解析を行なう場合，このような繰り返し構造を持つパターンに着目することが重要である．このように多く出現する関数の組み合わせパターンは，何らかの意味を持つものと考えられる．そこで $list($ $hanoi(x,f,t,o))$ と等価な関数として新しい関数 $lhanoi(x,f,t,o)$ を導入する．

```
7: lhanoi(x,from,to,other)
     =>  list(hanoi(x,from,to,other))
```

7 の規則と 1,2 の規則より次の規則が導ける[†2]．

```
8: lhanoi(s(x),from,to,other)
    -> list(hanoi(s(x),from,to,other))
    -> list(do(hanoi(x,from,other,to)
              ,do(move(s(x),from,to)
                 ,hanoi(x,other,to,from))))
    -> append(list(hanoi(x,from,other,to))
             ,list(do(move(s(x),from,to)
                 ,hanoi(x,other,to,from))))
    -> append(lhanoi(x,from,other,to)
        ,append(list(move(s(x),from,to))
            ,list(hanoi(x,other,to,from))))
    -> append(lhanoi(x,from,other,to)
        ,append(cons(move(s(x),from,to),nil)
            ,lhanoi(x,other,to,from)))
    -> append(lhanoi(x,from,other,to)
        ,cons(move(s(x),from,to)
            ,append(nil
```

[†2]矢印-> は書換えおよび unfold 操作を表し，最後の項を規則の右辺とする．

```
                ,lhanoi(x,other,to,from)))
    -> append(lhanoi(x,from,other,to)
        ,cons(move(s(x),from,to)
                ,lhanoi(x,other,to,from)));
9: lhanoi(0,from,to,other)
    ->  list(hanoi(0,from,to,other))
    ->  list(move(0,from,to))
    ->  cons(move(0,from,to),nil);
```

8,9 の規則を追加した項書換え系を用いた書換えでは効率が向上している．入力 $lhanoi(s(s(0)), A, C, B)$ に対し，正規形を得るまでの書換え回数は 15 回であり，図 3.7と同じ解を得ることができる． これは，変換以前の 34 回の半分以下である．

この規則の変換の作業そのものは [5] で用いられている fold/unfold 変換と同一であるが，どの関数に着目して変換を行なうかを見つけることが従来の手法では困難であった． 本論文が提案する項の視覚化，および計算の視覚化を用いた解析・検証手法により，直観的にどの関数に変換を適用すればよいのかを発見することが可能で，実際の例でその効果を確かめることができた．また，図 3.9に変換前と変換後での数値情報の視覚化を示す．上のウィンドウが変換前を示し，下のウィンドウが変換後を示している．このグラフには，横軸に書換え回数，縦軸は項のサイズ，深さ，幅，リデックスの数等の数値情報が表示されている．上のグラフは下のグラフより右にグラフが長いことから，下のグラフより書換え回数が多いことがわかる．また，上のグラフにおける項のサイズ，深さ，幅の変化の様子と下のグラフにおけるそれらを比較すると，上のグラフを左右に約半分に縮小したように見え，面積で考えるとほぼ半分になっていることが視覚的に確認できる．つまり，この数値情報の視覚化により変換後の項書換え系の実行が変換前と比べて書換え回数，サイズが共に少なくなっていることが視覚的に確かめられ

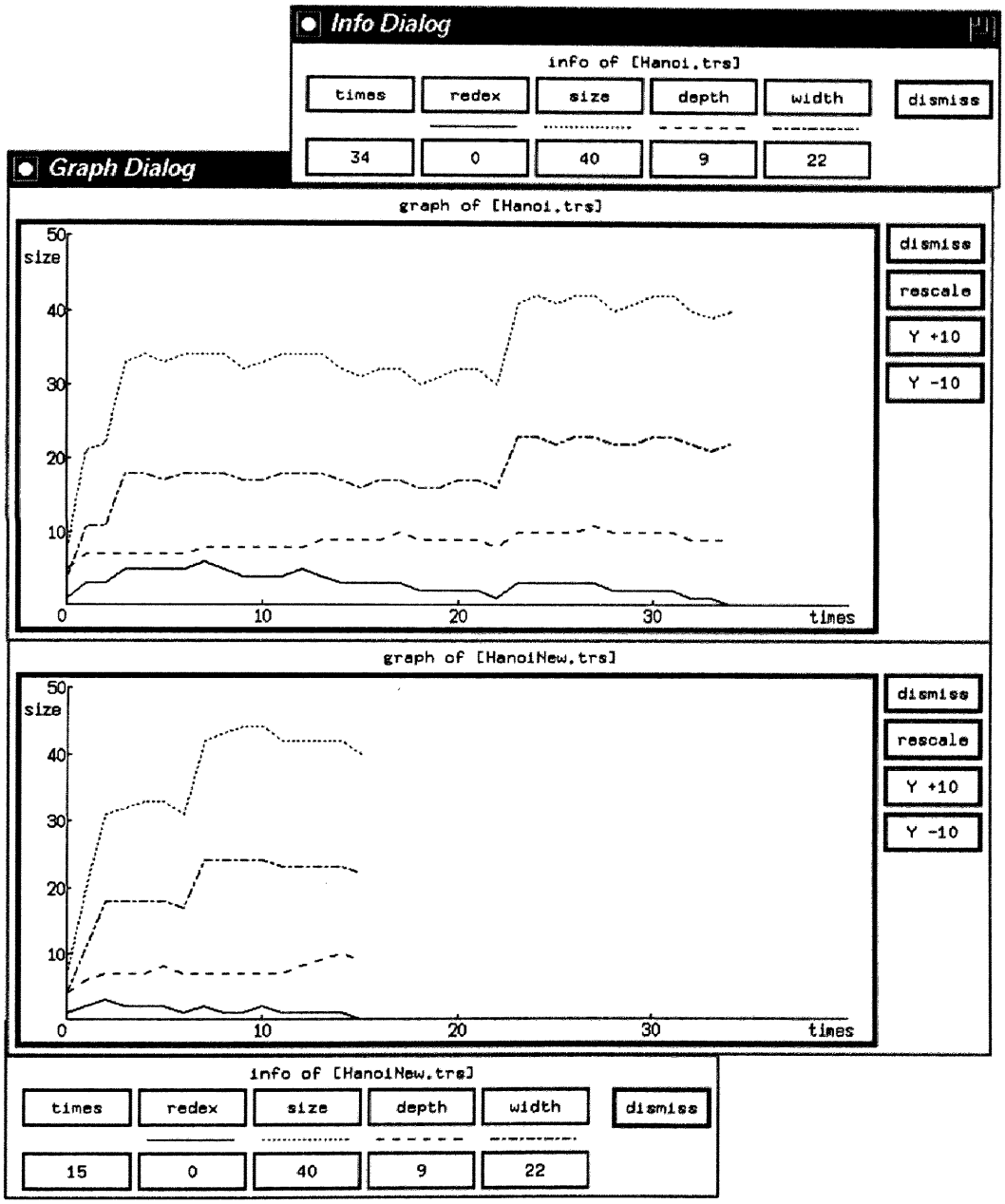

図 3.9: 変換の前後における計算過程の数値情報の視覚化

た．また，これらの数値情報は書換え中の任意の時点で参照することができるため，解析の手助けとなる．

3.4 視覚化手法の応用

これまでに述べたのは，項書換え系の基本的な構成要素に対しての視覚化手法であった．本節では，より複雑な解析手法に対する視覚化手法の高度化について述べる．

3.4.1 再帰経路順序の視覚化

項書換え系の停止性判定は完備化手続きなどに必須な手続きであり，一般に規則の左辺と右辺の間に整礎な順序を定めて証明する．再帰経路順序 (RPO) は適用範囲の広い順序であることが知られているが，関数記号間に先行 (Precedence) 順序と呼ばれる半順序を必要とし，この順序は計算により求めることができないため，適切な先行順序を与えることが必要である．本節では項の視覚化を拡張し，RPO の直観的理解を可能にする視覚化手法を提案する [42]．また，それに基づき先行順序を定めることを支援するユーザインタフェースを実現した．

3.4.2 順序の視覚化手法

RPO は先行順序ならびに多重集合順序を用いて定められ，その上，順序が成立する場合が 3 通りあるため，直観的に理解することが非常に困難である．

そこで，本節で提案する RPO の視覚化は以下の手順で行なった．

(1) 順序づけを行なう 2 項を同時に視覚化する

(2) 個々の部分項の比較を項を囲む枠で表す

(3) 枠の対応を数字で表す

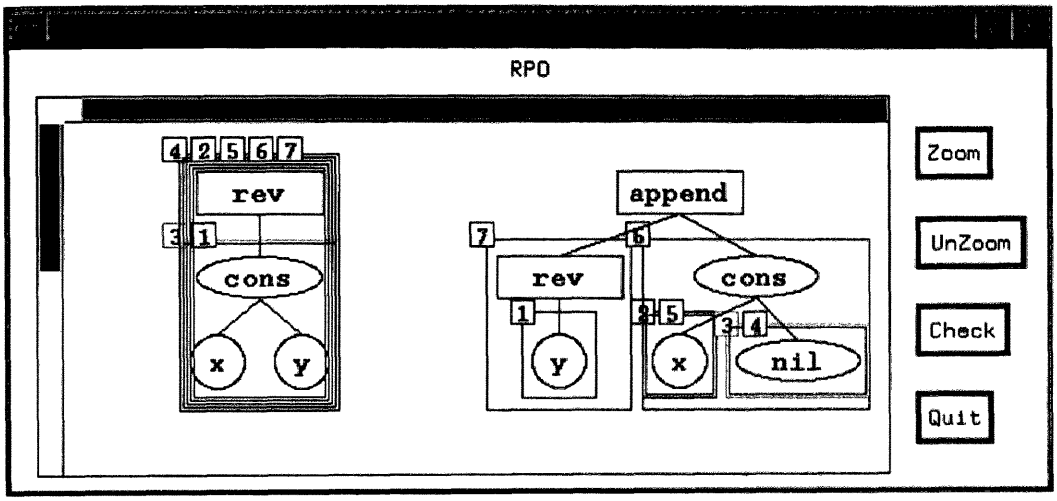

図 3.10: RPO の視覚化

(4) 順序が成立した場合 (3 通りのいずれか) を枠の色で区別する

(1) は，再帰経路順序によって比較する 2 項を同時に視覚化し，比較する点を理解しやすくすることである．再帰経路順序は部分項を再帰的に探索して 2 項での比較を行なうが，実際に比較が行なわれた場合，どの部分項が比較の対象であったかを理解するのは困難である．そこで (2),(3) では，比較する部分項を互いに枠で囲み，同じ番号づけを行なうことにより，実際に比較が行なわれた部分項を把握することが可能になる．さらに，(4) によって再帰経路順序の定義のどの条件によって順序が成立しているかを確かめることができる．

これらのアイディアにより，本手法では順序の直観的理解を深めることに成功している．図 3.10は実際に実現した RPO の視覚化例である．1 から 7 までの番号が枠の対応を表しており，番号が小さいものから順に順序が決定されている．複数の枠や番号が同一の部分項に対応している場合，枠や番号が重ならないような配置を行なっている．

再帰経路順序では比較的単純な例でも多くの部分項の比較が行なわれているこ

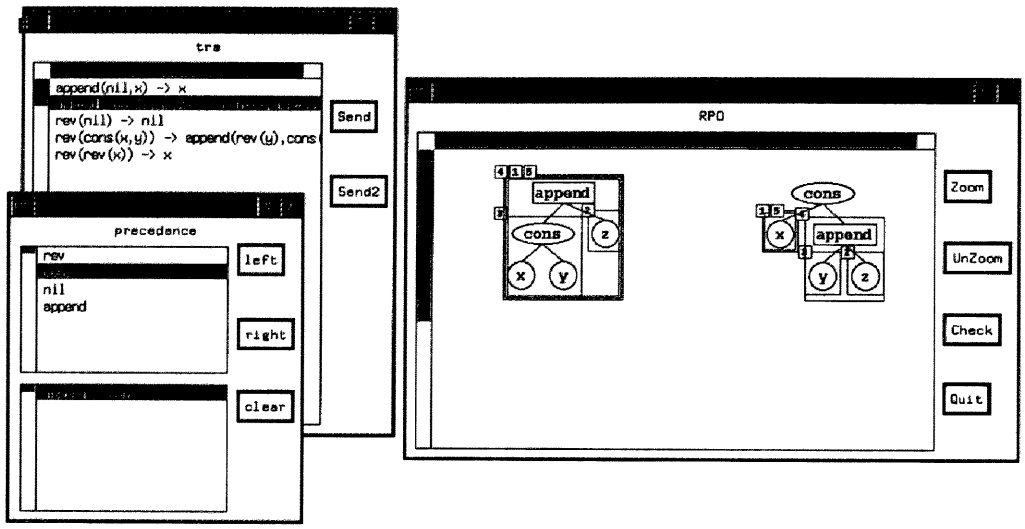

図 3.11: 先行順序決定支援システム

とが理解できる.

3.4.3 先行順序決定支援システム

図 3.11は我々が作成した先行順序を定めるシステムであり，以下に挙げる 3 つのウィンドウを持つ.

(1) 規則の選択

(2) 先行順序の決定

(3) RPO の視覚化

このシステムにより，関数記号間の先行順序を定め，規則を個々に視覚化して RPO で順序づけがなされることを確認し，項書換え系の停止性の証明を行なうことができる.

全ての規則上に (左辺)>(右辺) の順序関係が成り立てば停止性が証明できる. そこで，(1) のウィンドウで，比較対象とする規則を選ぶ. 次に (2) のウィンド

ウで規則上に現れる関数記号間の先行順序を決定する．先行順序は半順序であるので，すべての記号の間に順序を定める必要はなく，時には関数記号上の順序が全く必要無い場合もある．

先行順序を決定したら (3) において，RPO が成立するかを確かめる．失敗した場合，どの部分の比較がうまくいかなかったかを確かめ，先行順序の訂正を行なう．

本システムでは，グラフィカルユーザインタフェースを採用したため，RPO の視覚化により，順序づけの様子を確認しつつ試行錯誤が容易に行なえる．

3.5 他の視覚的支援手法との比較

視覚的なプログラミング支援手法は様々な言語モデルで実際のシステムとして実現されている．本節では関数型言語，手続き型言語，論理型言語，オブジェクト指向言語，プロトコル記述言語についてそれぞれ具体的な視覚的支援環境を挙げ，それらの視覚的支援手法についてまとめるとともに，本章で提案する視覚的支援手法の位置づけを行なう．

関数型言語の支援環境としては，グラフィカルユーザインタフェースを用いた統合環境である Interlisp-D[11] が有名である．Interlisp-D は構造エディタや MASTERSCOPE と呼ばれる解析ツールなど，lisp プログラミング支援のための様々な機能が実現されている．視覚的な支援としては関数の呼び出し関係の図式化 (Call-Graph) や spy 関数で調査した実行効率を図で表す機能を持つ．

Linos らによる CARE (Computer Aided RE - engineering) [51] は C 言語のプログラムを読み込み，その関数の呼び出し関係の図形表示を行なうことができるプログラム解析ツールである．Linos らは 40 人の学生を CARE を用いるグループと CARE を用いないグループに分け，2 種類のプログラムの変更に関する実験を行ない，CARE を使うことによって 37 パーセントの時間を節約できることを確かめ，100 パーセントの学生が CARE によるプログラムの視覚化がプログ

ラム理解に役に立つと述べていることを報告している.

FE'92[93] は並列論理型言語 GHC のための視覚的環境であり，図式により述語やゴールの入出力関係を明確に表すことができる．FE'92 ではプログラムのテキスト表現と図式表現の因果結合を実現しており，各々の変更がただちに他方へ反映される．AMLOG[55] は Prolog と等式論理を融合した言語である．三宅らは，AMLOG の実行モデルの視覚化手法を定め，視覚化を行ないながら動作する実行モニタを提案している．HyperDEBU[95] は並列論理型言語 Fleng のためのデバッグ環境であり，様々なビューを用いてプログラムのプロセスモデルを参照することができる．実現されているビューは TREE ウィンドウ，I/O tree ウィンドウ，GOAL ウィンドウであり，それぞれインデントがついたテキストベースのウィンドウとして表示することが可能である．また，デバッグのための機能として，様々なレベルでのブレークポイントの設定が可能である.

SmalltalkAgents[73] はオブジェクト指向言語 Smalltalk の統合開発環境であり，クラスブラウザやインタフェースビルダを持つ．クラスブラウザは様々な手法でプログラムの検索が可能で，クラスの継承関係のグラフを図式で表すことができる．また，インタフェースビルダにより，ユーザインタフェースを視覚的に実現できる．他のオブジェクト指向の視覚化システムとしては [83] がある.

Chao らは [6] でペトリネットによるプロトコル設計・検証・合成のためのグラフィカルツールについて報告している．このツールではペトリネット視覚的に画面に表示し，直接編集することが可能であり，その動作のシミュレーションや，検証，合成などが可能である．その他のプロトコル作成支援システムとしては [32, 14, 53] があり，それぞれ対象としているプロトコル記述言語に対して，直接的な視覚的操作が可能である.

このように様々な言語に対して視覚的な支援が試みられているが，これらは以下のように分類できる.

(1) プログラムモジュール間の関係の視覚化によるプログラム理解支援 (Interlisp-

D,CAREにおける CallGraph およびSmalltalkAgents における継承グラフ)

(2) プログラムの視覚化によるプログラム理解支援 (FE'92)

(3) プログラム実行結果の視覚化による実行の解析支援 (Interlisp-D の spy 関数の図)

(4) インタフェースの視覚化 (すべての支援環境)

(5) 視覚的プログラミング (FE'92,SmalltalkAgents のインタフェースビルダ)

ここで (1) はプログラムの一部の関係を表しているので, (2) を特定の関係に関して詳細化したものともいえる. 一方, 本章で提案する視覚的支援手法は, 項, 数値情報, インタフェースの視覚化がそれぞれ (2),(3),(4) に対応する. また, (5) は構造図形エディタおよび, 簡単化のための項書換え系の例示による作成などに対応する. しかし計算の視覚化に対応する支援はこれまでの支援環境にはない. 計算の視覚化は, 項が計算状態を表す項書換え系の特長を生かした独自の支援手法であり, 計算の過程を完全に表している点が新しい. 計算のすべての実行が視覚的に表現されていることにより, 計算状態 (項) の構造がどのように変化していくかを視覚的に確認でき, 高度なプログラムの解析が可能である. 本章が提案する視覚的支援手法では (1) について述べられていないが, これは, 項書換え系の関数間の依存関係や到達可能性に関するものに対応すると考えられる.

3.6 おわりに

本章ではこれまで項書換え系に対して考えられていなかった項書換え系の解析・検証・変換のための視覚的支援手法を項, 計算, 操作, 情報のそれぞれの視覚化を用いて提案した. これらは項書換え系の特長を生かしたものであり, 項書換え系で表現される計算の視覚的理解を行なうことができる. 項や計算の視覚化によ

り，従来のテキスト環境では発見することが困難であった変換の着目点や，効率の問題点の直観的な検出が可能である．特に計算の視覚化はこれまでの視覚的支援手法では考えられておらず，項が計算状態を表す項書換え系の特長を生かした手法である．また，数値情報の視覚化により計算状態（項の構造）とその具体的な数値や数値変化の情報を同時に得ることができる．インタフェースの視覚化により，思考を妨げない直観的な操作が可能となる．我々は実例を用いて視覚化を用いた解析・検証・変換を示し，本論文で提案した視覚的支援手法が項の構造の理解・解析や変換着目点の検出に有効であることを確かめた．

さらに，項の視覚化を拡張して再帰経路順序の視覚化手法を提案した．この視覚化により，順序が成立する場合の直観的理解が可能になり，先行順序の決定の支援システムが作成できた．

今後の課題には，関数間の依存関係の視覚化や項書換え系のモジュール化への対応，アニメーションを用いた計算系列等の表現方法の考察が挙げられる．また，計算系列上の繰り返しパターンの発見を支援するため，構造パターンをポインタにより直接指示し，計算系列上に同じ構造が現れる場所を示す手法などを導入する必要があろう．より高度で使いやすい視覚的環境を構築するためには，他の支援環境ではすでに実現されている完備化の機能などを理論的成果に基づいて視覚化することや，現在の木表現による視覚化を，部分項の共有の様子をより直観的に表すグラフ表現に拡張すること，項の構造エディタ [79] の作成，書換え関係の視覚化の実現，書換え例による規則の追加，デバッガの作成などが必要である．また，順序づけのシステムに関しては，現在は順序づけに成功した部分をすべて同時に表示しているが，問題のある場所のみを表示したり，階層的な表示を行なうなどの拡張が考えられる．また，より強力な順序である Path Ordering[31] も RPO と同様に先行順序を必要とするため同様の視覚化による支援が可能であろう．

第 4 章

視覚的支援手法に基づく項書換え支援環境 TERSE の実現

4.1 はじめに

本章では，前章で提案された視覚的支援手法に基いた項書換え系の視覚的支援環境の実現について述べる．この環境は項書換え系の解析や検証を支援することが目的であるため，モデルや視覚化の手法の変更に柔軟に対応できるような実現が必要となる．本章では実現に用いる言語として近年注目されている関数型言語 Standard ML(SML)[19, 67] を並列化した Concurrent ML(CML)[77] および eXene[76] ライブラリを採用した．

環境の設計においては Smalltalk-80[15] の MVC(モデル・ビュー・コントローラ) 構造 [49] に基づくモジュール分割を行なった．また，項書換え系を直接視覚化の対象として扱わず，視覚化の対象として抽象木を導入し，項と抽象木の間ではモデルの変換を行なう．

この実現手法に基づき，項書換え支援環境 TERSE(TErm Rewriting Support Environment) をワークステーション上に実現した．その結果，以下の事項が明らかになった．

1. 項書換え系の処理系の実現が SML の機能により簡潔に行なえること．

2. 抽象木の導入により，モジュールの再利用性が向上すること．

3. SML の参照型を用いて，環境の実行中に視覚化手法を変更できる枠組が提供できること．

4. 高い抽象度とモジュール性により保守，変更が容易なシステムが実現できること．

5. SML により実現された視覚的環境が実用的に十分な応答速度を持つこと．

特に SML のパラメータ化の機構の利用により，抽象度の高いモジュール化が可能になり，また，抽象木の導入により汎用的な木構造の描画アルゴリズムを記述することができた．

著者らは，変更容易性を確かめるために，試験的にTERSEを変更して命題論理のタブロー法の視覚化を行なったが，そのための変更は全記述量の 1 割程度であった．このことからも，本実現手法の有効性が確かめられた．

以下，4.2 節で SML，CML，eXene について簡単に解説し，4.3 節では，MVC 構造に基づくTERSEの設計方針を述べ，モデル，ビュー，コントローラについてそれぞれの実現手法を解説する．4.4 節では変更容易性，再利用性および応答速度について評価を行なう．

4.2 準備

項書換え系の視覚化システムの実現に用いる言語に望まれる性質としては，以下の事項が挙げられる．

- 高度な記述力を持つ

  計算モデルの持つデータ構造等が直接表現できるような記述力が必要．

- 視覚的表現の機能を持つ

  データの視覚化を行なうために必要．

- ある程度の実行速度を持つ

  視覚的操作がインタラクティブに実行できる反応速度が必要.

- 移植性が高い

  特定の計算機を対象とせず，様々な計算機で実行が可能であること.

本論文では，これらの性質を満たす言語として，Standard ML を並列化した Concurrent ML(CML)[77] および eXene[76] ライブラリを採用した．以下，Standard ML, Concurrent ML, eXene ライブラリについて説明する.

4.2.1 Standard ML

Standard ML(SML)[67] は Edinburgh LCF[54] をルーツに持つ，静的スコープ及び多相型を特徴とする関数型プログラミング言語であり，強力な型推論とモジュール化の機構を持つ．関数の定義にパターンマッチングが使えるため，ユーザが定義したデータ型の処理を簡潔に記述することができる．モジュール化の機構としては，データ型や関数の型の宣言をし，モジュールのインタフェースを定めるシグネチャ (`signature`) と，モジュールの実現であるストラクチャ (`structure`) による抽象化の機構と，パラメータ化の機構をもつモジュールであるファンクタ (`functor`) が挙げられる．また，副作用を持つ代入が可能な参照型を持つため SML は純粋な関数型言語ではないが，そのため実用的な言語として実用的なプログラムを実現できる能力を持っている.

SML はこのように高度な記述力を持つ実用的な言語であり，優秀なコンパイラも多様な計算機上で実装されているため，計算モデルの研究者の間では計算モデルを実際に計算機上で実装しシミュレートする言語として期待を集めている.

4.2.2 Concurrent ML

Concurrent ML(CML)[77] は Standard ML(SML/NJ) を拡張した並列関数型言語である．CML 上ではプロセスよりも小さく，切替えが高速なスレッド (*thread*) を用い，複数のスレッドが共有データを使い同時に動作することが可能で，各スレッド間ではチャネル及びイベントを用いた同期通信を行なうことができる．CML ではスレッドの制御および通信に関する関数や型が追加されているが，SML の機能をそのまま利用することができる．CML で新しく導入された型には通信のチャネルを表す型である `'a chan` と，イベントに対する型の `'a event` などがある．また，スレッドの発行に`spawn`，イベントの送受信のために`send`, `accept` などの関数が定義されている．具体的に複数のスレッド間で通信を行なう時は，チャネルをスレッド間で共有する．一方はチャネルに対する送信 (`send`) を行ない，他方は受信 (`accept`) を行なう．送信と受信が一つのチャネルに対し発行されたときに同期が行なわれ，それぞれのスレッドでイベントが発生し，受信側では送信されたデータを受けとることができる．同期が行なわれるまでは，双方のスレッドは待機状態となる．このように，CML ではスレッド間の通信を容易に扱うことが可能である．付録 A に CML を用いた簡単な通信プログラムの例を挙げておく．

4.2.3 eXene ライブラリ

eXene[76] は CML 上に構築された X Window を利用するためのライブラリである．eXene は Xlib の図形描画関数とほぼ同等の機能を持ち，Xtoolkit のボタンやリスト，メニューなどの基本的なウィンドウオブジェクト (`Widget`) と同等の機能も提供している．また，X サーバとの通信には C 言語のライブラリを全く使用せず，プロトコルのレベルからすべて CML で記述されている．個々のウィンドウにはボタンやリストなど複数のスレッドを同時に走らせることが可能で，互いに通信をしながら並行動作を行なうので，ユーザからの指示を処理するプロ

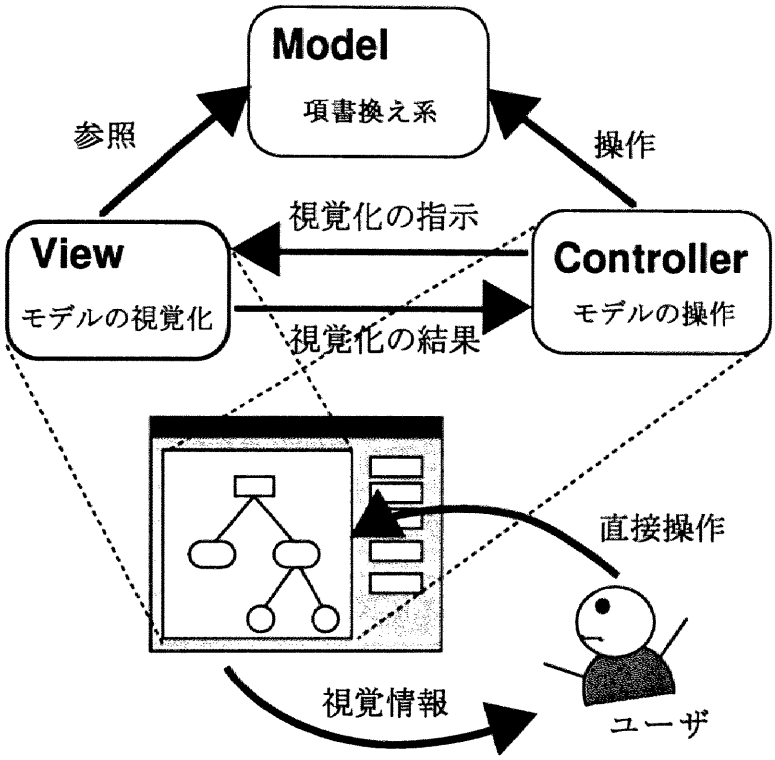

図 4.1: MVC 構造

グラムをイベント駆動を用いて自然に記述できる.

また，各々のWidget は CML のモジュールとして定義されているため，変更や新しいWidget の定義を容易に行なうことができる.

4.3 視覚化の実現

視覚的項書換え支援環境TERSEは関数型言語 Standard ML を並列化した Concurrent ML および eXene ライブラリを用いて実現されている. TERSE の設計方針としては Smalltalk-80 で提唱された MVC 構造を採用した (図 4.1). MVC 構造はこれまでにも様々な視覚化システムに用いられており [27], グラフィカルユーザインタフェースに基づくウィンドウシステムの基本となる設計手法である. MVC 構造はオブジェクト指向言語におけるオブジェクト間の関係を表現しているが,

本論文では各々のモジュール群の分割の方針として捉えている.

TERSE の実現においてはビューのモジュール性を高めるため, 抽象木を中間的なモデルとして導入した. つまり, MVC 構造のモデルの部分を項書換え系と抽象木および項と木構造との変換を行なう部分に分割する. TERSE を構成するモジュール群は次のように分類できる.

- モデル
  - 項書換え系のモデル : 項書換え系の計算を行なう.
  - 抽象木 : 視覚化の対象モデル.
  - モデル変換部 : 項を抽象木に変換する.
- ビュー : 抽象木等を視覚的に表現する.
- コントローラ : 項書換え系に対するユーザからの操作をモデル, ビューに反映させる.

TERSE におけるモジュールの依存関係を図 4.2に示す. この図では主な依存関係を矢印で接続し, 矢印の終点にあるモジュールが始点にあるモジュールを利用していることを表している.

図 4.3に TERSE のソフトウェアの構成を示す. OS 上に SML が動作しており, その上にモデルの実現が載っている. ビューの実現はモデルと eXene に依存している. また, コントローラの実現は CML と eXene , ビューの実現に依存している.

以下では TERSE のプログラムの一部を例として挙げて, モデル, ビュー, コントローラの順に各々の SML による実現を説明する. SML の詳細については [67] に詳しいのでこちらを参照されたい. なお, TERSE は約 7000 行のプログラムで実現されている.

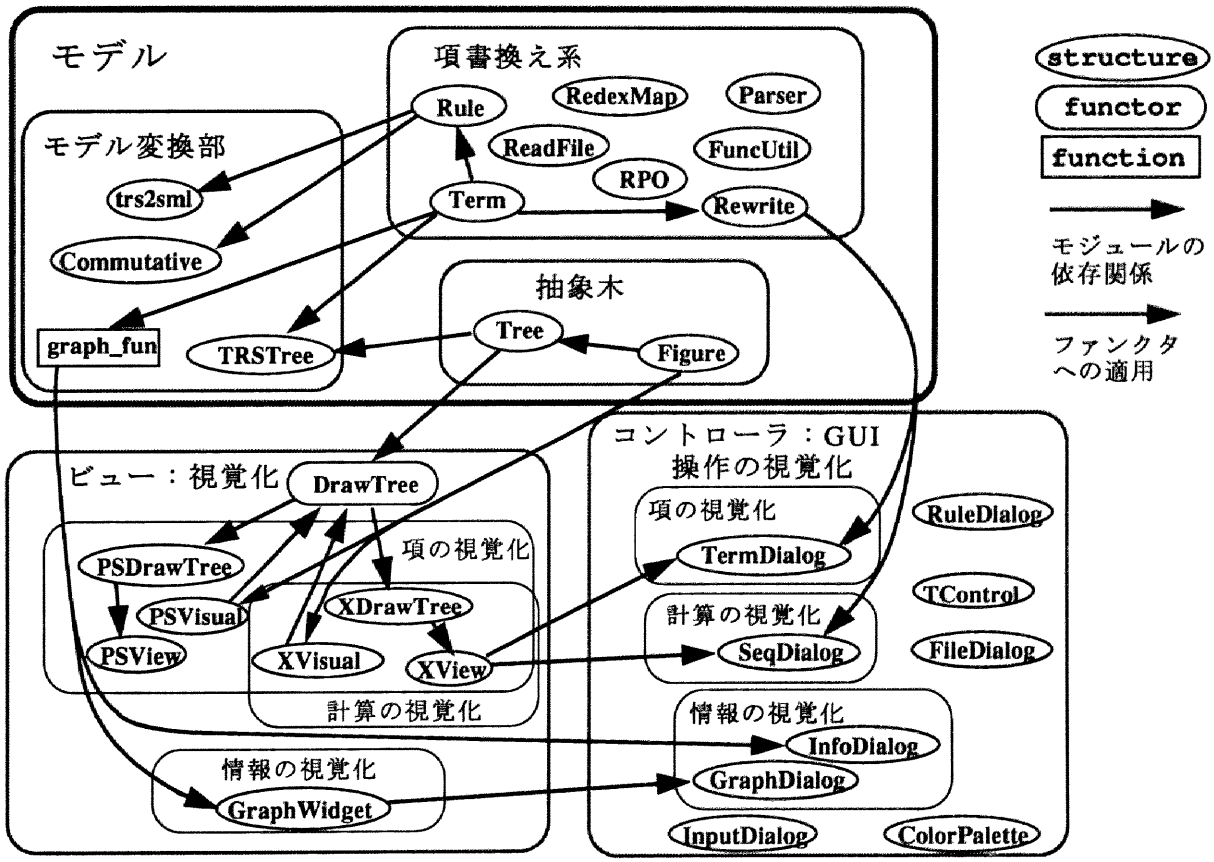

図 4.2: モジュールの依存関係

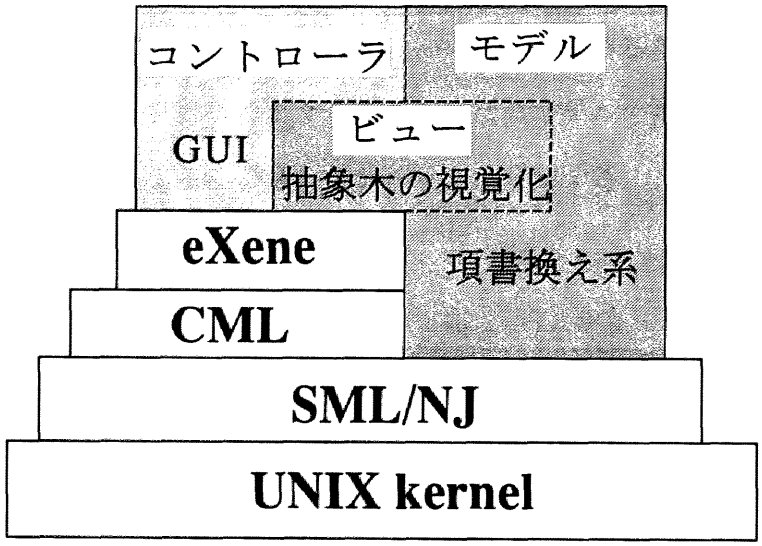

図 4.3: TERSE の構造

4.3.1 モデルの実現

モデルのモジュール群は項書換え系，抽象木，モデル変換部に分割されている．

項書換え系のモデル

項書換え系のモデルはSMLのデータ型の機能を用いて自然に記述することができる．項を表す型はデータタイプ宣言 (datatype) を用いた新たな合成型termで定義されている．TERSEにおける項書換え系の基本的な型定義を以下に示す．

```
(* 項を表すデータ型 term *)
datatype term = Var of string
              | Fun of string * term list
type rule = term * term          (*   規則   *)
type trs = rule list             (*   TRS    *)
type occurrence = int list       (*   出現   *)
type redex = occurrence * rule   (* リデックス *)
```

```
type subst = string * term      (*   代入   *)
type substs = subst list        (* 代入のリスト *)
(* 並列戦略: 複数のリデックスを選ぶ *)
type m_strategy = redex list -> redex list
type strategy = redex list -> redex (* 戦略 *)
```

なお, '(*' で始まり '*)' で終る部分はコメントであり, 'type xxx = yyy' は型yyy の略記名としてxxx を用いることの宣言である. 'datatype xxx = yyy of zzz' は新たなxxx というデータ型がデータ構成子yyy とその引数zzz からなることを宣言し, '|' はデータ構成子が複数ある場合に用いる. 'xxx * yyy' は型xxx とyyy の対の型を表し, xxx list は型xxx のリスト型であることを表している. ここではterm はデータ構成子 'Var' に文字列 (string), もしくはデータ構成子 'Fun' に文字列とterm のリストの対を引数にとる型になる. 例えば, $fact(s(x))$ という項 (term) は SML 上では

```
Fun("fact",[Fun("s",[Var "x"])])
```

と表すことができる.

項を操作するために, 項と規則からリデックスを発見する関数findRedex や, 一般に複数存在するリデックスから書換え対象を選択する戦略を表す型strategy の関数などが定義されている. 'xxx -> yyy' は引数に型xxx の値をとり, 型yyy の値を返す関数の型を表す. 例えば型m_strategy の関数は, リデックスのリストを引数にとり, リデックスのリストを返す一種の並列戦略関数である. 項の操作を行なう関数の型定義の一部を以下に示す.

```
(* 並列戦略関数 *)
val outmost : m_strategy
(* TRS と書換え対象の項からリデックスを探す *)
val findRedex   :trs * term -> redex list
```

```
(* リデックスをその規則で書換える関数 *)
val reduceRedex :term * redex -> term
val reduceRedexs:term * redex list
                                -> term
(* TRS, 戦略を用いて項を書換える関数  *)
val rewrite:    trs * strategy
                      -> term -> term
```

ここで, 'val xxx : yyy' はxxx が型yyy の値を持つことを宣言する. 例えばoutmost は最外並列戦略を実現する関数であり, その型はm_strategy である.

これらの関数の実現はいくつかの補助関数を用いて定義される.

例として, 項と項のマッチングを行なう関数match の定義の一部を示す. 関数全体の定義は付録 B に示す.

```
fun match (Var(s),t1,sub: substs)
          = (true,(s,t1)::sub)
  | match (Fun(s,t0),Var(x),sub)
          = (false,nil)
 ....
```

ここで'fun' は関数定義を表し'=' 以後が関数の本体である. SML では関数の引数に直接データ構成子を記述することができるため, '|' を用いてデータ構造のパターンマッチングによる関数定義が可能である. '::' はリストのデータ構成子であり, 空リストは'nil' で表される. 例えば 1::2::nil は [1,2] を表す. match は 2 つの値 (マッチングの成否と変数への代入) を返す関数であるため, 返り値は値の対になっている. SML のパターンマッチングの機能を用いてmatch が簡潔に記述できることに注意されたい.

抽象木

抽象木はビューのモジュール性を高めるために導入された．項を直接視覚化せずに一旦抽象木に変換してから視覚化を行なうため，モデルの構造の変更がビューの視覚化モジュールに影響を与えない．以下に抽象木の型定義をシグネチャにより示す．

```
signature TREE =
  sig
    structure Figure : FIGURE
    datatype tree =
            Leaf of Figure.object
          | Node of (Figure.object  * tree list);
  end
```

ここでFigureは木のノードの形状を表すストラクチャでありobjectは抽象的な図形の形状を表す．Figure.objectはモジュール（ストラクチャ）内の型を参照する．抽象木では個々のノードの形状をFigureを用いてさらに抽象化している．

モデル変換部

TERSEのモデル変換部では，以下の４種類の変換が実装されている．

- trs2sml：TRSをSMLのプログラムに変換
- Commutative：可換則に基づく効率化変換 [34, 35, 36, 37]
- TRSTree：項から抽象木へのモデルの変換
- graph_fun：情報の視覚化で用いる項の評価関数

trs2sml は TRS を高速に実行するための一種のコンパイラで，SML のパターンマッチングの機能を用いて実現されている．Commutative は TRS から TRS への可換則に基づく効率化変換 [35, 37] を行なうモジュールである．著者らはこのモジュールを用いて実際に様々な解析・評価を行なっている．

ストラクチャTRSTree は項から抽象木へのモデルの変換を行なう．部分木の省略や出現のマークづけもこのモジュールで行なう．以下にストラクチャTRSTree の一部を示す．

```
structure TRSTree : TRSTREE =
 struct
   structure T:TREE = Tree
   datatype ftype = CON of sort| DEF of sort
                    |VAR of sort |TREE
   type trsdef = (string * sort )list
                     * (string * ftype ) list
   type mlist = (occurence * tredex) list
                  * (occurence * int) list
      ...
(* 項から抽象木への変換関数 term2tree            *)
(* val term2tree : trsdef -> mlist             *)
(*                       -> term  -> T.tree    *)
   fun term2tree (vl,_) (red,m) (Var x) = ...
     | term2tree (_,fl) (red,m) (Fun(x,nil))=
           ...
  end (* struct  *)
```

関数記号，変数記号のリスト，リデックスの出現位置などの情報を関数term2treeに与えると個々のノードがその関数記号やリデックスなどに対応したobject である抽象木が得られる．抽象木の導入により項の構造と抽象木の構造とが異なるような任意の構造変換が可能になり，TERSEでは部分木の省略表示が実現されている．この手法を用いれば特定の構造の省略や，正規形の省略表現なども可能である．

情報の視覚化は項から数値情報を求め，その視覚化（グラフ化）を行なう．モデル変換部では項から数値情報を得る関数graph_fun が以下のように定義されている．

```
val graph_name = ref ["times","redex"
                ,"size","depth","width"];
val graph_fun =  ref (fn t => [redex_size t
                ,term_size t,depth t,width t]);
```

graph_fun は参照型 (ref) で定義されており，任意の等しい型の関数が代入できる．参照型は純粋な関数型言語の範囲を越える副作用を持つ型であるが，この機構により，視覚化する数値情報を環境の実行中に変更することが可能になる．

4.3.2 ビューの実現

ビューでは項の視覚化，計算の視覚化，情報の視覚化が実現されている．

項，計算の視覚化

項，計算の視覚化では，抽象木を木構造として描画するアルゴリズムを実現しているファンクタDrawTree が中心的な役割を果たしている．図 4.4に簡略化した木構造描画関数drawTree のアルゴリズムを示す．drawTree は図形描画関数drawObj, drawCObj と線分描画関数drawLine を用いている．このアルゴリズムが 10 数行

```
[アルゴリズム drawTree]
入力   : 描画する抽象木 tree , 描画位置 pos
副作用 : 図形を描画する
出力   : 図形の幅, ノードを描画した矩形 rect

補助関数: drawObj (pos) 位置 pos を左端としてノードを描画する
                        図形の幅, 図形の矩形 rect を返す
          drawCObj(pos) 位置 pos を中心としてノードを描画する
                        rect を返す.
          drawLine(rect,rect) rect 間に線分を描画する

1. tree が 葉の場合, drawObj(obj,pos) を呼び出して終了
2. tree が子を持つノードの場合,
   a. 子供を左から順に drawTree によって描画する.
   b. 子供の幅の合計の中心に drawCObj によって自分のノードを描画する
   c. 各子供に対し, 自分のノードまでの線分を drawLine によって描画する.
   d. 幅 = 子供の幅の合計 , rect = 自分の rect を出力して終了.
```

図 4.4: 木構造の描画アルゴリズム

で記述できることから SML が高度な記述力を持つことが理解できる (関数の定義は付録 C 参照).

また, 実際のアルゴリズムでは描画方向や倍率の変更を可能にしたり, コントローラで使用するための個々のノードの描画位置をその出現とともに返すためにここで挙げたdrawTree よりは複雑なものになっている. TERSE では図形や線分の描画を行なう関数drawObj, drawLine はシグネチャVISUAL を満たすストラクチャで定義されており, ファンクタDrawTree のパラメータになっている. 実際の木構造描画モジュール (XDrawTree 等) は図形, 線分の描画モジュールのDrawTree への適用で得られる. TERSEではX Window への描画モジュール XVisual と PostScript 言語を用いた描画モジュール PSVisual が実現されている. この仕組みにより, 具体的な描画関数と木構造の描画のアルゴリズムが明確に分割されている.

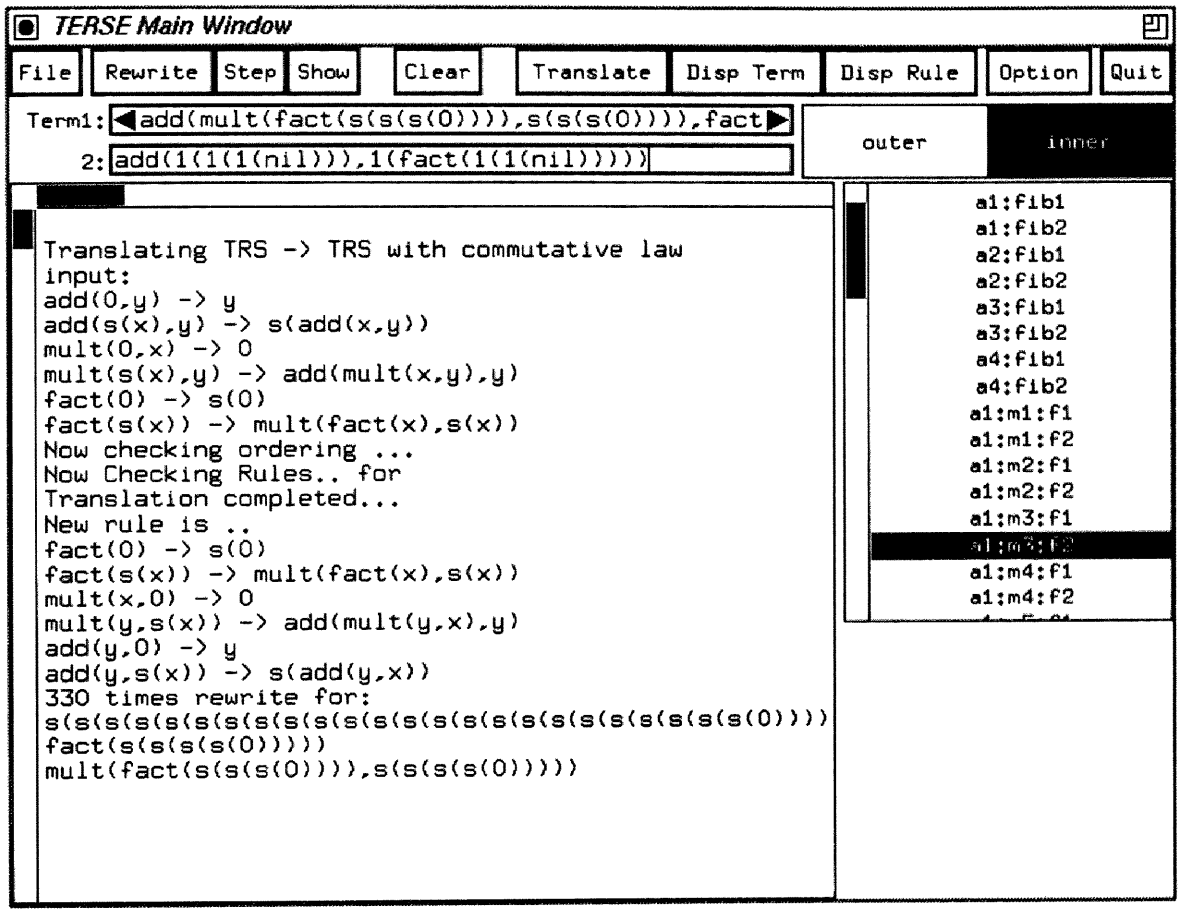

図 4.5: Main Window

数値情報の視覚化

数値情報の視覚化はモデル変換部のgraph_fun により得られた数値情報を折れ線グラフにするストラクチャ Graph Widget により実現されている．このモジュールではチャネルを用いて環境の実行時の値の更新やスケールの変更を可能にしている．チャネルにはユーザからの指示や値のリストなどがコントローラを通じて入力される．

4.3.3 コントローラの実現

コントローラでは操作の視覚化およびその他のグラフィカルユーザインタフェースが実現されている.

ボタンやメニューに対するユーザからの操作は eXene が持つ機能を用いて処理される. eXene では, ボタンやメニューなどの基本的な Widgets は押下されるとイベントをチャネルに送るように設計されている. このチャネルを監視するスレッドを作成し, 各々のボタンやメニューに割り当てられた機能を実行することで, イベント駆動のコントローラを容易に実現できる. 図 4.5 に TERSE の Main Window を示す. 様々なメニューやボタンにより, 種々の機能を実現している.

また, Term Viewer(図 4.6) の右側には書換えや戦略の指示を行なうボタン, ビューの視覚化をコントロールする拡大, 縮小などのボタンが並んでいる. これらのボタンの機能に対応する関数を各々の監視スレッドに記述することでコントローラは実現されている.

木構造に対する操作は以下の手順で実現している.

1. 抽象木を木構造として描画するときに個々のノードの描画位置と出現を保存する.
2. マウスがクリックされた位置, ボタンの種類を eXene のイベントから得る.
3. ノードの情報とクリック位置との比較により, クリック位置の抽象木上での出現を得る.
4. 抽象木上の出現から項上の出現への変換を行なう.
5. ボタンの種類に対応する関数を項と項上の出現を引数として呼び出す.

この手法により, 任意の抽象木のノードに対する操作を一元的に処理することができ, TERSE では書換え対象の選択, 省略する部分項の指示, 出現のマーク付けの指示が実現されている.

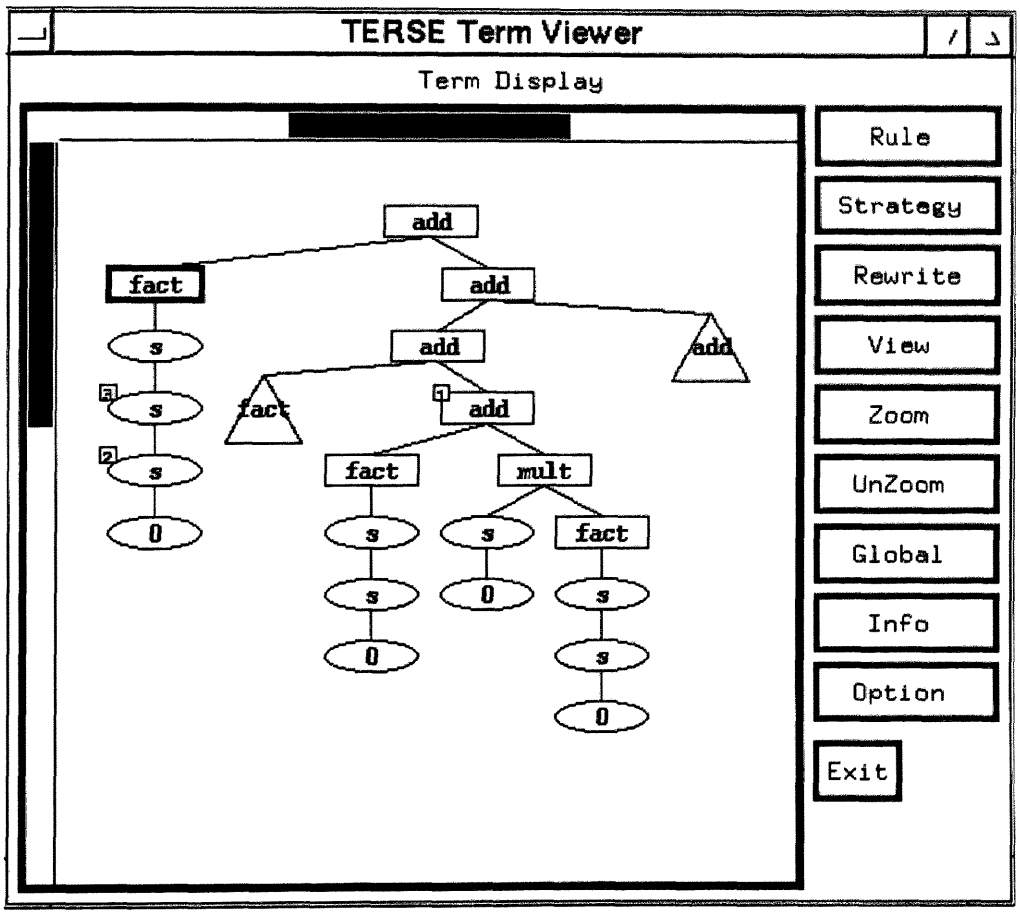

図 4.6: Term Viewer

4.4 評価

4.4.1 変更容易性，再利用性

図 4.2が示すようにTERSEは高度にモジュール化されて実現されている．モジュール化を推し進めると共に，モジュール間インタフェースの設計は複雑になるため，特に木構造の描画に関しては数回にわたり設計の見直しを行なった．その結果，完成したモジュールは関数型言語の利点を生かして，簡潔で抽象度の高い実現になった．

例えば構造を持つ他の計算モデルの視覚化の実現を行なう際には，モデルを対象の計算モデルに置き換え，抽象木へのモデル変換部を作成するだけで，ビュー，コントローラはほとんど変更を必要としない．

著者らはTERSEのモジュールの再利用性を確かめるため，試験的に Term Viewer を変更して命題論理のタブロー法を視覚化するシステム Tableaux Viewer を作成した．すでにタブロー法のモデルは 200 行程度の SML プログラムで記述されているので，このシステムを作成するために必要な作業はタブローから抽象木へのモデル変換部と，コントローラからモデルへ操作の指示を行なう部分のみである．実際に作成したプログラムでは，タブローの構造から抽象木へのモデル変換部が新しく 50 行程度追加された．また， Term Viewer はもともと 700 行で記述されていたが，項書換え系の操作に関する 200 行を削除し， 70 行程度のタブローの操作を追加してシステムが完成した．この作業はほぼ 1 日で終了した．実際にこの Tableaux Viewer を記述しているプログラムは木構造を描画するアルゴリズムなどを含めると 3000 行程度であるため， 1 割程度の修正で新しいモデルの視覚化に対応できることが確認できた．このことからも，本章で提案する実現手法が高度な再利用性，保守性を持つことを確かめることができた．

表 4.1: 書換えの実行時間

計算対象 (最内戦略)	書換え回数	時間 (秒)	平均 ノード数	平均時間 (秒)
$fact(4)$	62	15	24	0.2
$fact(5)$	194	180	98	0.9
$fib(8)$	99	23	23.7	0.2
$fib(9)$	171	73	37.4	0.4

4.4.2 応答速度

関数型言語で作成されたTERSEが実用的な応答速度を持つことを確かめるために簡単な実験を行なった．実験システムは SparcCenter 1000(SuperSPARC 50MHz) を用い，端末として X 端末の XMiNT CSL を用いた．実験結果を表 4.1 に示す．

実験では最内戦略を使用して，木構造の描画を行ないながら各々の書換え計算を連続して行なった．実験の結果，ノードの数が多い場合に平均的に多くの実行時間を必要とすることがわかる．しかし，90 個以上のノードを持つ木を描画しても応答速度は 1 秒以内であり，インタラクティブな視覚的システムとして速度的な問題はないといえる．このことから，関数型言語でも十分に実用的なインタラクティブシステムが実現できることが確かめられた．

4.5 他の視覚的システム構築ツールとの比較

視覚的（視覚化）システムの作成を簡便にすることを目的として，様々なグラフィカルユーザインタフェース (GUI) 構築ツール（ライブラリ）が存在する．本章は，基礎的な計算モデルに対する視覚化システムの構築手法を提案しているが，本手法が他の手法に対し，どれほどの利点を持つか本節では，項書換え系の視覚的環境の実現を行なうことを前提として開発効率の比較を行なう．

4.5.1 Xlib,Xtoolkit

X Window システムでは Xlib, Xtoolkit[8] が最も基本的な GUI 構築ライブラリである．本稿で用いた eXene ライブラリ はこれらの機能を含み，機能的には同等である．ただし実行速度は SML は C 言語よりも一般に 10 倍程度遅い．

一方，開発効率について考察すると，C 言語対 SML では圧倒的に SML のほうが容易にプログラムを構築することが可能である．まず，デバッグの手間が少なくて済むことが一つの理由として挙げられる．SML では静的な型推論により実行時エラーが発生しないため，動作が意図と異なる場合はアルゴリズムが間違っている場合のみであり，原因の追求が容易に行なえる．

4.5.2 Motif, NextStep

Motif ライブラリ [70] や NextStep Development システム [21] は，現在最も広まっている GUI であり，どちらも多くのライブラリと優秀なインタフェースビルダー (IB) を持つ．eXene には IB は存在しないため，ユーザインタフェースの実現のみを行なうことを考えると，Motif や NextStep は開発効率が良い．しかし，IB を用いて得られるソースコードは多くの場合冗長であり，これを人の手で編集することは忍耐を必要とする作業である．

また，特殊な視覚化を行なうためのコードを書く場合は結局低レベルのライブラリを使う必要があり，Xlib を利用する場合と同様に C 言語 (Objective-C) の記述力の低さが問題となる．この点では抽象度を高くすることが可能な関数型言語を用いた手法が開発効率の点で有利である．

4.5.3 Tcl/Tk

Tcl/Tk などの簡易言語（専用言語）を用いた開発は，インタフェースに限れば非常に効率が良く，テキストアプリケーションのグラフィック化などを行なうフロントエンドを開発するには最適な手法である．一方，簡易言語であるために

汎用の型やレコードの概念に乏しく，複雑な構造を自然に表現することは困難である．項の視覚化のような複雑なデータ構造に密接した機能を実現するためには強力なデータ型を持つ必要がある．また，ファンクタのようなパラメータ化の概念を持たないため抽象度を高めることが難しい．最近では [86] のように，Tcl/TK を関数型言語と結びつけて視覚化を行なう試みがなされている．しかし，Tcl/Tk と言語の間で複雑なデータのやりとりを行なう必要があり，プログラムの視覚化等の複雑なデータの表示手法には適当ではない．

4.6 おわりに

項書換え系の視覚的環境TERSEを Smalltalk-80 で提唱された MVC 構造に基づき設計し，近年注目されている関数型言語 Standard ML を並列化した Concurrent ML および eXene ライブラリを用いて実現した．項書換え系のモデルは SML の柔軟な型システムにより簡潔に記述することができた．視覚化の対象をモデル化する抽象木の導入と，詳細なモジュール化により，モジュールの再利用性や変更容易性が高いシステムが実現できた．TERSE が実用的に十分な応答速度で動作することは実験により確かめられた．

これらの結果から，

- 計算モデルの処理系の記述に SML が有用であること
- 並列関数型言語を用いてインタラクティブなシステムが実現できること

が明らかになった．これらは，本研究の実現手法を一般化して，木構造をデータ構造として持つ計算モデルの視覚化ツールキットが構築できることを示唆している．

第 5 章

可換則に基づく項書換え系の変換と項の視覚化による実行効率の評価

5.1 はじめに

本章では，３章で提案する項書換え系の解析・検証・変換のための視覚的支援手法の有効性を確認するために，４章で実現した環境を用い，プログラム変換の分野において，本論文で提案する手法が有効であることを示す．

ソフトウェアの実行効率と開発効率という互いに排反する要請に応えるための一つのアプローチとしてプログラム変換 [71] がある．関数型言語の分野においては，現在までに fold /unfold 変換 [5]，カタログ変換など種々の変換手法が提案されている．しかし，一般にプログラム変換は，変換と効率の関係が複雑であるので，変換の戦略は多くの場合，発見的な手法を用いたり，人間による補助に頼らざるを得ない．

本論文の著者は，３章で提案し，４章で実現した項書換え系の解析・検証・変換のための視覚的支援手法を用いて，様々な項書換え系の解析を行なった．具体例として３章では fold/unfold 変換に基づく変換の解析を行なっている．この経験から，項書換え系が可換則の成り立つ関数を定義している場合，その関数を定義している規則において，関数の引数の順序を入れ換えるという単純な変換が実

行効率を向上させる可能性を持つことに気づいた．この変換は単純ではあるが，複数の関数記号にわたり変換を行なうと，その適用の組み合わせにより効率が大きく変化する．つまり，変換の適用の組み合わせの中から最適な組み合わせを選ぶことにより実行効率を向上させられる．この事実は，4章で実現した項書換え系の視覚的環境を用い，実際に書換え系列の視覚化を用いることによって容易に確認できる．しかし，すべての組み合わせについて実行効率を調べることは実用的では無いため，与えられた項書換え系から最適な組み合わせになる自動変換手法の開発が求められる．

本章ではこの変換方針として，項に例えば再帰経路順序のような単純化順序を与え，帰納の位置という概念を用い，再帰を行なう項を小さくすることにより実行効率の良い項書換え系を得る手法を提案する．

以下，5.2節で項書換え系の効率について，5.3節で項の視覚化を用いて可換則変換の実際と，規則の組み合わせによる実験結果について説明する．5.4節では，最適な組み合わせを得るための可換則変換適用アルゴリズムについて説明する．最後に，5.5節でアルゴリズムを用いた変換の実際と，必須な書換えに基づく効率の評価を行なう．

5.2 項書換え系の実行効率

項書換え系の実行効率を判断する基準は[104]でも議論されており，書換えの回数，リデックスの個数，項の幅，深さ，サイズ，マッチングの回数，リデックスの位置，実計算時間など，様々な基準が考えられるが，どの基準を用いるのが合理的な実行効率の判断かについての理論的な結論はない．

この章では単純化のため最内の必須呼びによる書換えの回数と，項の幅，深さ，サイズなどにより判断することとする．また，必須呼びを行なうために対象とする項書換え系は停止性を持つことを仮定する．

【定理 5.2.1 】最短書換え回数

無曖昧，左線形，かつ停止性を持つ項書換え系において，最内の必須呼びによる書換えはその項の最短の書換え系列の一つである．□

【証明】最内の書換えでは，リデックスのコピーは行なわれない．無曖昧，左線形の項書換え系では，コピーを行なわないかぎり書換える順序を変えても，必須リデックスの数は変わらない．よって，最内の必須呼びによる書換えは最短の書換え系列の一つである．

このことにより，項書換え系の書換え回数による実行効率の評価には，最内の必須呼びを用いればよいことがわかる．

5.3 可換則に基づく変換

可換則は，加算や乗算などの演算で成り立ち，引数の順序を入れ換えても計算結果が変わらないことを表している．この性質は，一般に項の構造帰納法により示すことができる [81]．本節では実際の変換例を示しつつ，可換則に基づく変換の概念を説明し，変換による実行効率の変化を示す．可換則変換の形式的な定義は次節で行なう．

5.3.1 階乗を計算する項書換え系の変換

変換対象として自然数の階乗 ($fact$) を計算する項書換え系を取り挙げる．この規則では，関数 $fact$ は加算，乗算の関数を呼び出している．

まず，自然数の加算 (add) を定義する項書換え系を示す．ここで，自然数は 0 と後者関数 s によって表されている．

$$\text{A1:} \quad \begin{cases} add(0,y) & \rightarrow \quad y \\ add(s(x),y) & \rightarrow \quad s(add(x,y)) \end{cases}$$

この項書換え系の下では全ての基底項 x, y に対して，

$$add(x,y) = add(y,x)$$

が成り立つ，すなわち add が可換則を満たすことが知られている．可換則を満たす関数に対し，その引数を入れ換えることを可換則変換と呼ぶ．項書換え系 A1 に可換則変換を行なうと，次に示すような 3 通りの項書換え系を得ることができる．

$$\text{A2:}\begin{cases} add(0,y) & \to\ y \\ add(s(x),y) & \to\ s(add(y,x)) \end{cases}$$

$$\text{A3:}\begin{cases} add(y,0) & \to\ y \\ add(y,s(x)) & \to\ s(add(y,x)) \end{cases}$$

$$\text{A4:}\begin{cases} add(y,0) & \to\ y \\ add(y,s(x)) & \to\ s(add(x,y)) \end{cases}$$

A2 は A1 の第 2 規則の右辺に可換則変換を行なったものであり，A3 は A2 の左辺に可換則変換を行なって得られた項書換え系である．規則の左辺に可換則変換を行なう時は，関数の全域性を失わないために同じ関数記号の規則全てに同時に変換を行なう．つまり，A2 の左辺には可換則変換が 1 回だけ可能である．また，A4 は A3 の第 2 規則の右辺に変換を行なって得られた項書換え系である．

項書換え系 A1,A2,A3,A4 の書換え回数に基づく実行効率について考察する．項 $add(m,n)$ に対し，A1 は第 1 引数で自然数に対する構造帰納法を行なうため $m+1$ 回の書換えを，同様に A3 は第 2 引数で帰納法を行なうため $n+1$ 回の書換えを必要とする．一方，A2, A4 は書換えを行なう度に第 1 引数と第 2 引数が入れ換わるため，それぞれ $min(m+0.5,n+1)*2, min(m+1,n+0.5)*2$ 回の書換えを必要とする．すなわち add の 2 引数のうち第 1 引数が大きい項に対しては A3 の効率が良く，第 2 引数が大きい項に対しては A1 の効率が良い．また，引数の差が大きい項をいくつか書換える場合は，2 引数の最小値をとる A2, A4 のほうが平均的に考えると効率が良い．つまり，任意の入力項について A[1-4] のうちどの項書換え系が最も効率的かを判断することはできない．

例えば，$add(s(s(s(0))),s(0))$ という項に対し，A1,A2,A3 はそれぞれ異なる計算系列を持つ．図 5.1では，書換え系列の視覚化を用いて，これらの計算系列

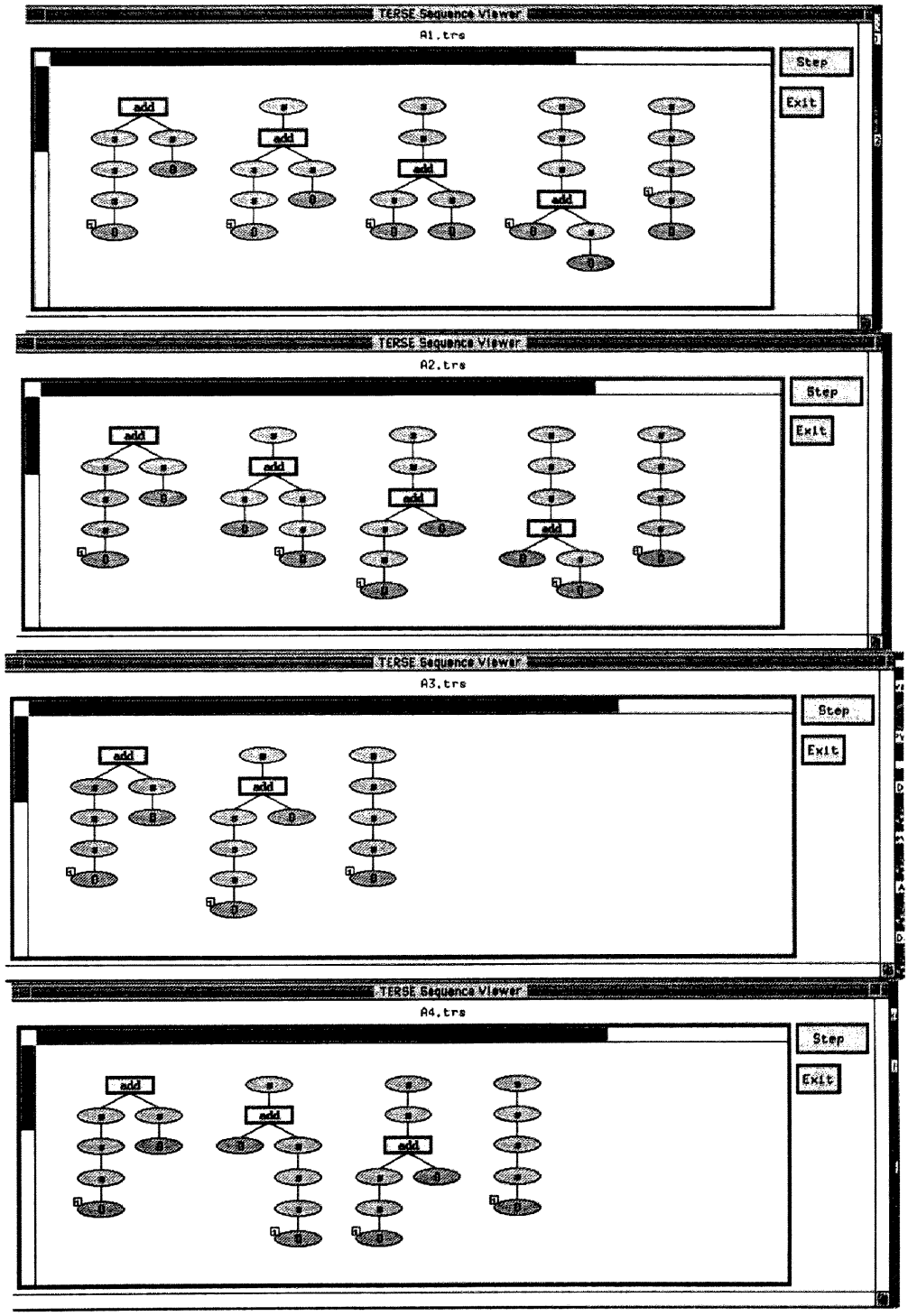

図 5.1: add の計算の例

を同時に表示している．A1 は第 1 引数を順に小さくすることで計算を進めるのに対し，A2 は引数を交換して計算を行なう．A3 は第 2 引数を再帰の対象にしている．A4 は A2 と同様に引数を交換して計算を行なっている．これらの項書換え系は同じ入力に対して，全く同じ計算結果を持つが，その計算の過程が異なる．計算（書換え系列）の視覚化を行なうことにより，これらの違いを直観的に理解することが可能になっている．

次に，乗算 $(mult)$ を定義する項書換え系を示す．

$$\text{M1:} \quad \begin{cases} mult(0,y) & \rightarrow \quad 0 \\ mult(s(x),y) & \rightarrow \quad add(y, mult(x,y)) \end{cases}$$

M1 が定義する $mult$ も可換則を満たすことが知られている．M1 には可換則変換を適用できる場所が左辺に 1 箇所，第 2 規則の右辺に 2 箇所ある．それぞれの場所に可換則変換を行なうことで，全部で 8 通りの項書換え系が得られる．これを M[1-8] と呼ぶ．ここで M2 は右辺の $mult$ に可換則変換を行なったもの．M3 は add に可換則変換を行なったもの，M4 は両方に変換を行なった結果の項書換え系である．また M5-M8 はそれぞれ M1-M4 の左辺に可換則変換を行なった結果の項書換え系である．この項書換え系は add の規則を持たないため，単独では計算を実行することはできないが，add の計算が 1 ステップで行なえると仮定して入力項 $mult(m,n)$ に対して書換え回数を比べると，M1, M3 は $2m+1$ 回，M2, M4 は $min(4m+1, 4n+3)$ 回であり，M6, M8 は $2n+1$ 回，M5, M8 は $min(4m+3, 4n+1)$ 回となる．やはり add の項書換え系の変換結果と同様，任意の入力項に対し，どの項書換え系が最も効率が良いかを判断できない．

最後に階乗 $(fact)$ を定義する項書換え系を示す．

$$\text{F1:} \quad \begin{cases} fact(0) & \rightarrow \quad s(0) \\ fact(s(x)) & \rightarrow \quad mult(s(x), fact(x)) \end{cases}$$

$fact$ では右辺に $mult$ があるため，ここに可換則変換を行ない，2 種類の項書換え系 F1, F2 が得られる．結局，加算，乗算，階乗の項書換え系をそれぞれ組み

表 5.1: 規則の組み合わせによる階乗の書換え回数

戦略	組み合わせ	fact(1)	fact(2)	fact(3)	fact(4)	fact(5)	fact(6)
最内最左	A1:M1:F1	6	14	28	62	194	928
	A1:M1:F2	6	12	24	62	232	1194
	A2:M5:F1	8	15	34	99	326	1567
	A3:M1:F2	5	9	18	92	1522	44604
最外最左	A1:M1:F1	6	20	74	330	1782	11426
	A1:M1:F2	6	12	24	62	232	1194
	A2:M5:F2	9	34	138	594	3126	19838
	A3:M1:F2	5	9	18	92	1522	44604

合わせると，全部で 64 種類の $fact$ を計算する項書換え系が得られる．

5.3.2 変換による効率の変化

前節で得られた規則の組み合わせすべてについて，代表的な逐次戦略である最外最左戦略，最内最左戦略を用いて実際に計算実験を行なった．$fact(1)$ から $fact(6)$ までの書換えを行ない，その結果の一部を表 5.1に示す．

表 5.1からわかるように，各々の変換結果を組み合わせて得られた項書換え系では，戦略に関係なく書換え回数に大きな差が生じていることがわかる．例えば，単純に回数だけで考察すると，最内最左戦略の $fact(6)$ の計算では最少回数と最大回数の差は実に 40 倍以上もある．

A1:M1:F1, A1:M1:F2, A3:M1:F2 に対して，最内最左書換えを行ない，その経過の数値情報の視覚化を図 5.2に示す．これらも全く同一の計算にもかかわらず，計算途中での項のサイズやリデックスの数などが全く異なることがわかる．

次節では, 項書換え系に対して可換則変換を適用する場所を求めるアルゴリズムを示す．

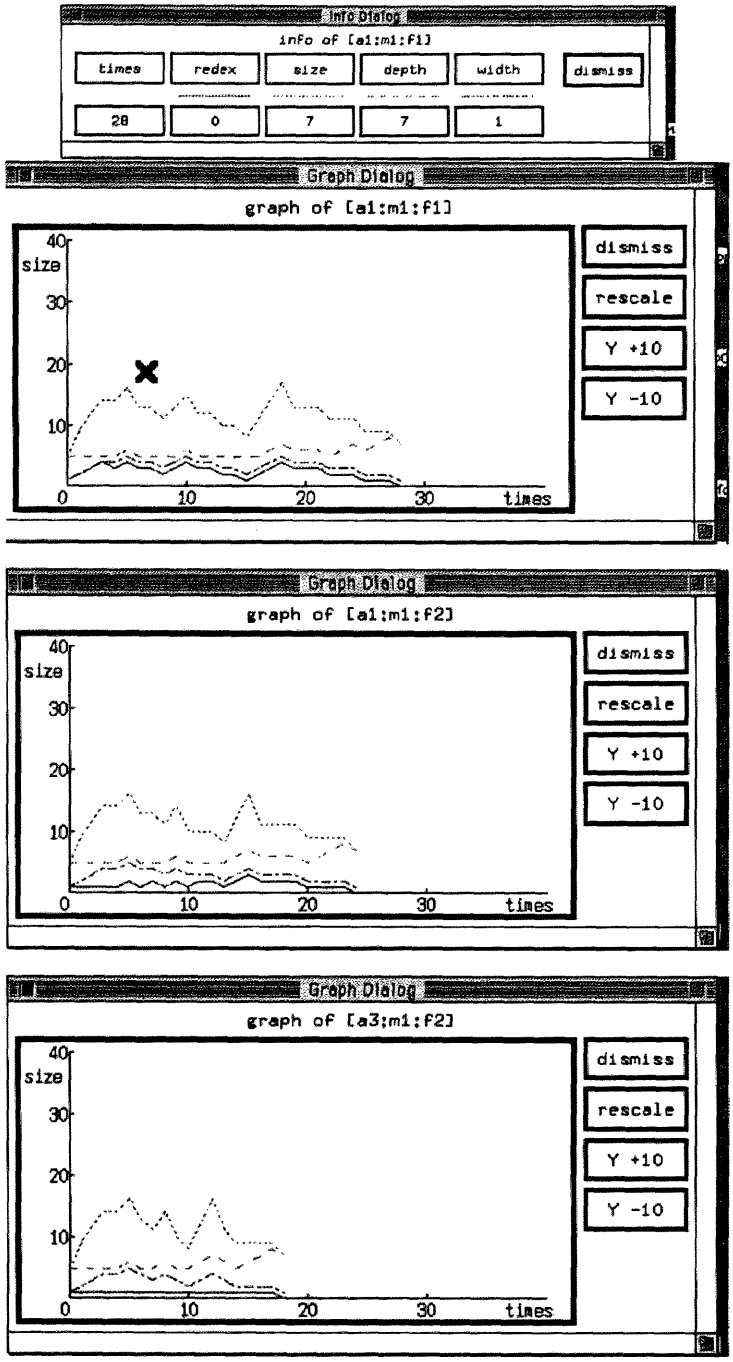

図 5.2: 階乗の計算における数値情報の視覚化

5.4 可換則変換適用アルゴリズム

5.4.1 準備

アルゴリズムを示す準備として諸定義を行なう.

【定義 5.4.1 】被定義関数記号の定義 TRS

TRS R における被定義関数記号 $f \in D$ の定義 TRS は次のように定義される R/f である.

$$R/f = \{(\alpha \to \beta) \in R \mid Top(\alpha) = f\}$$

【定義 5.4.2 】関数の全域性

TRS R が被定義関数記号 $f \in D$ に関して全域であるとは $arity(f) = s_1 \ldots s_n$ のとき，任意の $t_i = T(C_{s_i})(i = 1, \ldots, n)$ について $f(t_1, \ldots, t_n)$ がリデックスになることである．また R がすべての被定義関数記号に関して全域であるとき，単に TRS R は全域であるという.

規則の左辺で，台に対しての構造帰納法がどの引数で行なわれているかを意味する概念として帰納の位置を定義する．例えば TRS A1 では，左辺の第 1 引数のみに構成子関数記号である $\{0, s\}$ が出現し，構造帰納法を用いて add の定義が行なわれているので，帰納の位置は第 1 引数である.

【定義 5.4.3 】帰納の位置

全域な TRS R および被定義関数記号 $f \in D$ に対し，$IndOcc(f, R)$ を次のように定義する.

$$IndOcc(R, f) = \left\{ u \;\middle|\; \begin{array}{l} \exists(\alpha \to \beta) \in R/f \ \text{ s.t.} \\ u \in ConOcc(\alpha), |u| = 1 \end{array} \right\}$$

特に $IndOcc(R, f)$ の要素が 1 つのとき，その要素を f の帰納の位置と呼ぶ.

この帰納の位置に小さな項が入る規則は結果的に再帰の回数が減り効率が良く，逆に大きな項が入る規則は再帰の回数が多く効率が悪いと考えられる．そこで，可換則の成り立つ関数についてそれぞれの引数の大きさを比べ，小さい項で再帰を行なうように変換すればよい．

項の大小を定める半順序 $\succ$ としては，次のような性質を持つ半順序（単純化順序といわれる）を用いる．

$$\begin{aligned}&\text{(1) 単調性} \quad t \succ u \implies \\ &\qquad\qquad f(\cdots t \cdots) \succ f(\cdots u \cdots) \\ &\text{(2) 部分項性} \quad f(\cdots t \cdots) \succ t\end{aligned}$$

可換則を満たす関数については，両辺でその関数の部分項に出現する変数の位置は同一とする．これにより，書換えごとに引数が入れ替わることがなくなり，帰納の位置の引数は単調に小さくなる．これは形式的には次のことを意味する．

【定義 5.4.4 】 変数位置の保存

R を TRS，f を被定義関数記号とする．R が f に関する変数位置を保存するとは，次の条件が成り立つことである．

任意の $\alpha \to \beta \in R/f$，出現 u, v, w および $x \in V$ について

$$\left.\begin{array}{l} u \in FunOcc(\beta, f) \\ v \in VarOcc(\beta/u, x) \\ w \in VarOcc(\alpha, x) \end{array}\right\} \Longrightarrow v \preceq w$$

被定義関数記号の定義 TRS に可換則変換を適用するときの順序を定めるために，次に示す関数の依存関係を用いる．

【定義 5.4.5 】 関数の依存関係

TRSR において，被定義関数記号 f が依存する関数記号の集合 $Dep(R, f)$ は

$$Dep(R, f) = \left\{ g \;\middle|\; \begin{array}{l} \exists \alpha \to \beta \in R/f \text{ s.t.} \\ g \in Def(\beta), g \neq f \end{array} \right\}$$

で定義される．また，関数の依存関係 $Dep(R)$ は，

$$Dep(R) = \{(f, g) \mid g \in Dep(R, f), f \in D\}$$

で表される．

$Dep(R)$ により，関数の依存関係は非巡回有向グラフ (DAG) に書くことができる．このグラフを $Dag(R)$ と書き，DAG G の根に現れる関数記号を $Root(G)$ で表す．また G から関数記号 f を取り除いた DAG を $G - Node(f)$ で表す．

変換の手順は，最初に可換則変換を DAG の根に現れる関数記号の定義 TRS に対して行なう．次に変換の終了した関数記号は DAG から取り除く．この操作を DAG のノードがなくなるまで繰り返す．

5.4.2 可換則変換適用アルゴリズム

入力: TRS R，項の間の単純化順序 $\succ$

ただし，R は全域であり，無曖昧，左線形で停止性を持つ．可換則の成り立つ被定義関数記号 f について $|IndOcc(R, f)| = 1$ である．関数の依存関係は巡回していない．

出力: 項書換え系 R と等価な項書換え系 R'

変数と関数の説明:

Law : 可換則が成り立つ被定義関数記号の集合

Res : 関数記号の帰納の位置に関する制約を表す関数記号と自然数との対の集合を表す変数．空集合で初期化されている．

$G = Dag(R)$

$$Exch(f(t_1,t_2)) = f(t_2,t_1)$$

$$Small(f(t_1,t_2)) = \begin{cases} 1 & \text{if } t_1 \prec t_2 \\ 2 & \text{if } t_1 \succ t_2 \\ \bot & \text{otherwize} \end{cases}$$

手続き:

begin

while $G \neq \varepsilon$ **do**

$f := Root(G)$

(G の根から順に変換手順を行なう)

$R_f := R/f$

(R_f : 変換の対象である f の定義 TRS)

if $f \in Law$ **then**

if $\begin{pmatrix} (f,n) \in Res \\ \textbf{and} \\ IndOcc(R,f) \neq \{n\} \end{pmatrix}$ **then**

$R_f := \{Exch(\alpha) \to \beta | (\alpha \to \beta) \in R/f\}$

end if

if $\begin{pmatrix} R_f \text{ が } f \text{ に関する変数位} \\ \text{置を保存していない} \end{pmatrix}$ **then**

R_f が f に関する変数位置を保存するように規則の右辺に可換則変換を行なう

end if

end if

for each $\begin{cases} (\alpha \to \beta) \in R_f, \\ g \in Law, \\ v \in FunOcc(\beta, g) \end{cases}$ **do**

$m := Small(\beta/v)$

if $(g,n) \notin Res$ **then** (制約がない場合)

if $m \neq \bot$ **then**

$Res := Res \cup (g,m)$ (制約の追加)

end if

else ($(g,n) \in Res$ 制約がある場合)

if $m \neq n$ **and** $m \neq \perp$ **then**

$R_f := R_f - (\alpha \to \beta)$

$R_f := R_f \cup (\alpha \to \beta[v : Exch(\beta/v)])$

end if

end if

end for

$G := G - Node(f)$

end while

$R' = \bigcup_{f \in D} R_f$

end

アルゴリズムの適用の結果得られた項書換え系は R' で表される.

5.5 可換則変換の正当性

項書換え系 R と 2 引数関数記号 f に対し,

$$\forall t, u \in T(F), f(t,u) \stackrel{R}{=} f(u,t)$$

であるとき f は R の下で可換であるという. ただし $\stackrel{R}{=}$ は項書換え系 R による書換えの関係から定まる等価関係である. ある項書換え系 R の下で可換な関数の規則における引数を入れ換えることを引数交換と呼ぶ.

ここで項書換え系 R から R' への変換が正当であるとは以下の条件が成り立つことである.

$$\forall x, y \in T(F), x \stackrel{R}{=} y \Longleftrightarrow x \stackrel{R'}{=} y$$

【定理 5.5.1 】引数交換の正当性

関数記号 f は項書換え系 R の下で可換であり, R に対して f についての引数交

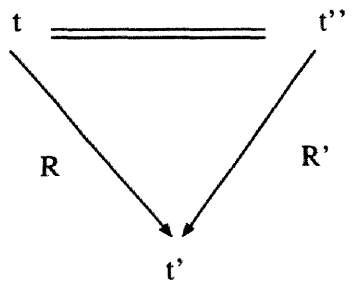

図 5.3: 引数交換の正当性

換を行ない項書換え系 R' が得られているとする．このとき f が R' の下でも可換ならばこの引数交換は正当である． □

【証明】左辺に対して引数交換を適用した場合を考える．$t \xrightarrow{R} t''$ とすると，t' が存在し $t' \xrightarrow{R'} t''$ かつ $t' \stackrel{R'}{=} t''$ となる．従って $t \stackrel{R'}{=} t''$ である．このことから $t \stackrel{R}{=} u \Longleftrightarrow t \stackrel{R'}{=} u$ である．すなわち，引数交換は正当であることが導かれる.(図 5.3) 右辺の引数交換についても同様に証明される．

【系 5.5.1 】 可換則変換の正当性

正当な引数交換の繰返しによる可換則変換は正当である． □

5.6 変換の実際と効率の評価

この節では本章が提案するアルゴリズムを用いて効率の良い項書換え系を求める変換例を示し，その結果の評価を必須呼びと数値情報の視覚化に基づいて行なう．

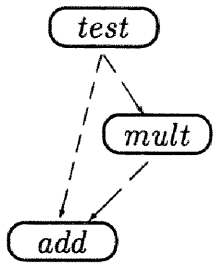

図 5.4: 関数の依存関係

5.6.1 アルゴリズムの実行例

アルゴリズムに入力する項書換え系 R として，次に示す項書換え系 T1 とすでに示した A1, M1 を組み合わせた項書換え系 A1:M1:T1 を用いる．

$$\mathrm{T1}: \begin{cases} test(0) & \to \quad 0 \\ test(s(x)) & \to \quad add(test(x), mult(s(x), x)) \end{cases}$$

また，項の間の順序には，単純化順序である再帰経路順序を用いる．$\succ_{rpo}$を定めるため，R の関数記号間には次のような半順序が定められている．

$$\{test \succ mult, test \succ add, test \succ s, mult \succ add\}$$

A1:M1:T1 は全域であり，無曖昧性，左線形で停止性を持つことが示せる．また可換則が成立する関数は $add, mult$ であり，

$$|IndOcc(R, add)| = |IndOcc(R, mult)| = 1$$

である．

R における関数の依存関係の DAG $G = Dag(R)$ は図 5.4のように表される．変換手順は G の根に現れる被定義関数記号 $test$ の定義 TRS である T1 から始める． T1 は可換則が成り立つ関数 $add, mult$ を第 2 規則の右辺に持つ．このそれぞれについて，引数の順序を確かめる必要がある．まず add についての帰納の位置を定める．引数に $test(x)$, $mult(s(x), x)$ があるので，$\succ_{rpo}$を用いて順序づけを

行なうと，$test(x) \succ_{rpo} mult(s(x), x)$ となる．add に関する制約はないので，add の帰納の位置を第 2 引数にするべきである．そこで，制約集合を $Res := Res \cup (add, 2)$ とする．次に $mult$ であるが，引数 $s(x), x$ の順序は $s(x) \succ_{rpo} x$ なので，$mult$ についての制約もないので，これも第 2 引数を帰納の位置にするべきである．そこで，制約集合を $Res := Res \cup (mult, 2)$ とする．この結果，項書換え系 T1 を変換する必要はない．

次に $mult$ を定義する項書換え系 M1 に対して変換を行なう．$mult$ に対しては可換則が成り立ち，すでに制約 $(mult, 2) \in Res$ が存在するので，この制約に合わせ帰納の位置を第 2 引数にするように左辺の規則に対し同時に可換則変換を行なう．M1 の左辺に可換則変換を行なった結果は項書換え系 M5 となる．

$$\text{M5:} \quad \begin{cases} mult(y, 0) & \rightarrow \ 0 \\ mult(y, s(x)) & \rightarrow \ add(y, mult(x, y)) \end{cases}$$

M5 の第 2 規則では $mult$ に関する変数位置が保存されていない．そこで右辺の $mult$ について可換則変換を行ない変数位置を保存させる．M5 の第 2 規則の右辺の $mult$ に可換則変換を行なった結果は項書換え系 M6 となる．

$$\text{M6:} \quad \begin{cases} mult(y, 0) & \rightarrow \ 0 \\ mult(y, s(x)) & \rightarrow \ add(y, mult(y, x)) \end{cases}$$

M6 の右辺には add があるので，この引数を rpo で順序づけを行なうと $mult(x, y) \succ_{rpo} x$ である．add に対して制約 $(add, 2) \in Res$ が存在するので，第 2 規則の右辺の add に可換則引数を行なう．M6 に第 2 規則の右辺の add に可換則変換を行なった結果は項書換え系 M8 となる．

$$\text{M8:} \quad \begin{cases} mult(y, 0) & \rightarrow \ 0 \\ mult(y, s(x)) & \rightarrow \ add(mult(y, x), y) \end{cases}$$

M8 は制約集合 Res を満たしている．

最後に add の項書換え系 A1 に対して変換を行なう．add には可換則が成り立ち，すでに制約 $(add, < 2) \in Res$ が存在するので，この制約に合わせ帰納の位置

表 5.2: 必須呼びによる変換前後の効率の評価

入力項	組み合わせ	書換え回数	サイズ		幅		深さ		リデックス	
			最大	平均	最大	平均	最大	平均	最大	平均
test(3)	A1:M1:T1(変換前)	32	8	4.6	10	6.8	23	16.2	5	3.1
	A3:M8:T1(変換後)	32	7	3.8	10	7.0	23	14.5	4	2.0
	A1:M5:T1	32	7	4.5	10	6.8	23	16.0	6	3.1
	A3:M5:T1	35	7	4.1	10	6.9	23	15.3	4	1.7
test(5)	A1:M1:T1(変換前)	116	12	5.6	42	17.9	59	42.3	7	3.8
	A3:M8:T1(変換後)	116	11	5.1	42	17.9	58	41.2	6	2.8
	A1:M5:T1	126	11	6.0	42	17.9	62	43.3	10	4.2
	A3:M5:T1	166	11	5.5	42	18.0	62	43.2	6	2.2
test(7)	A1:M1:T1(変換前)	330	16	6.4	114	50.1	137	95.6	9	4.3
	A3:M8:T1(変換後)	288	15	6.5	114	48.1	136	93.0	8	3.6
	A1:M5:T1	358	15	7.0	114	49.7	142	97.0	14	5.0
	A3:M5:T1	533	15	7.0	114	50.4	142	99.4	8	2.7
test(9)	A1:M1:T1(変換前)	778	20	7.0	242	111.9	271	186.2	11	4.7
	A3:M8:T1(変換後)	580	19	7.9	242	105.0	270	178.2	10	4.5
	A1:M5:T1	832	19	7.9	242	110.7	278	187.2	18	5.5
	A3:M5:T1	1336	19	8.4	242	112.4	278	193.1	10	3.2

を第２引数にするように左辺の規則に対し同時に可換則変換を行なう．A1 の左辺に可換則変換を行なった結果は項書換え系 A4 となる．A4 の第２規則では *add* に関する変数位置が保存されていない．変数位置を保存するよう右辺の *add* に可換則変換を行なった結果は項書換え系 A3 となる．項書換え系 A3 は *Res* の制約を満たしている．

得られた各被定義関数記号の定義 TRS を組み合わせ，変換結果の項書換え系 R' は A3:M8:T1 となる．

5.6.2 実験結果の評価

変換の結果得られた項書換え系 A3:M8:T1 と変換前の項書換え系 A1:M1:T1 の効率を必須な書換えの観点から評価する．また，他の組み合わせの項書換え系である A1:M5:T1，A3:M5:F1 についても参考のために評価を行なった．評価に用いる基準は，書換えの回数，項の幅，深さ，サイズ (関数記号の数)，存在するリデックスの数のそれぞれの最大値と平均値である．実験結果を表 5.2に示す．

表 5.2からわかるように，書換え回数においては引数が大きくなるに従い，変換の前後で効率に明らかな差が現れている．書換え回数から帰納的に計算量を求めてみると，$test(n)$ に対して A1:M1:T1 の規則では $(n^4 + 2n^3 + 11n^2 + 46x + 12)/12$ 回の書換えを必要とするのに対し，A3:M8:T1 の規則では $(2n^3 + 3n^2 + 4x + 3)/3$ 回の書換え回数であり，次数が一つ異なることが確かめられた．また，項の幅，深さ，サイズ，についてはほとんど差がないので，可換則による変換により確かに実行効率が良くなったことがいえる．また，A1-A4, M1-M8 の項書換え系について，それぞれ T1 と組み合わせる計算実験を行なったが，得られた 32 種類の項書換え系のうち，A3:M8:T1 の組み合わせが最も効率の良い組み合わせであり，変換アルゴリズムが正しく働いたことが確かめられた．

5.7 可換則変換に基づく自動プログラム変換システム

これまでで可換則変換は単純な変換ながらも，大きな効率化の能力を持つ事がわかった．そこで，単純な可換則変換の自動システムを構築する．そのためには，以下の問題点を解決する必要がある．

- 可換則が成り立つ関数の同定
- 項の上の単純化順序の決定

本節では，可換則な関数の同定については，被覆集合帰納法 [81] を用い，単純化順序の決定については再帰経路順序を用いた停止性判定システムを利用して順序

を定める手法を提案する．また，4章において実現した項書換え系の変換支援環境TERSE 上に，以下に述べる手法による可換則に基づく項書換え系の自動変換システムを実装した．

5.7.1 可換な関数の同定

可換則は帰納的定理であるため，その証明には被覆集合帰納法を用いる．一般に被覆集合は各ソートで無限個存在し，自動的に生成することは難しい．そこで，ソートごとにいくつかの被覆集合を前もって与えることとし，関数記号ごとに被覆集合帰納法を用いて可換則が成立することを確かめる．例えば関数記号 f が可換であるかどうかチェックする場合，まず f のアリティを調べる．$arity(f) = s \times s$ である時，ソート s に対する被覆集合 M により，新たな被覆集合 $M' = \{M \times M\}$ を作成し，これにより $f(x,y) = f(y,x)$ に対する被覆集合帰納法を行なう．すなわち M' の各々の要素 $< m_i, m_j >$ について，$R \cup \{f(p,q) \rightarrow f(q,p) | p \in Sup(m_i), q \in Sup(m_j))\}$ を用いて $f(m_i,m_j) = f(m_j,m_i)$ が成立するかどうかをメタ変数を用いた書換えにより確かめる．仮定としての等式を書換え規則にするときに，書換えの方向を R と重ならないように定めることで，書換えが停止しなくなることを避けることができる．$Sup(m)$ とは直観的には m 中のメタ変数を用いてそのメタ変数へのパス上でメタ変数と同じソートの関数記号とを入れ換えたものの集合を指す．

5.7.2 項の上の単純化順序

単純化順序としては停止性判定のための順序として良く用いられる再帰経路順序$\succ_{rpo}$を採用した．これは関数記号の間に半順序を必要とするので，次のような停止性を判定するための手続きを用いて順序づけを行ない，項の間の大小関係として用いる．

順序づけの手続き: 入力として項書換え系 R, 空集合で初期化された関数記号上の半順序 O_f を用いる. t に現れる関数記号から 1 つを選んだものを $Fun(t)$ とする.

1. 規則 $\alpha \to \beta \in R$ に対して $(Fun(\alpha) > Fun(\beta))$ により O_f を拡張する.
2. 規則 $\alpha \to \beta$ が O_f により $\alpha \succ_{rpo} \beta$ ならば $R = R - \{\alpha \to \beta\}$.
3. $R = \emptyset$ ならば O_f で成功.
4. これ以上 O_f の拡張ができないならば失敗, できるならば 1 へ.

順序づけの制御は関数 Fun により行なわれ, 出現による重みづけなどを用いて探索の効率化を図る必要がある. また, O_f の拡張を失敗した際にバックトラックを行なう必要もある.

5.7.3 自動変換システム

以上の手法を StandardML を用いて計算機上に実現した. 本章で用いた階乗を計算する項書換え系の例は, 引数交換により 64 通りの項書換え系に変換できる. これらの項書換え系すべてについて本自動変換システムを用いて変換を行ない, その結果について評価を行なった. その結果, 64 通り全ての項書換え系が 4 通りの最適な効率を持つ項書換え系に変換された. ここで, 最適な効率を持つ項書換え系とは, 64 通りの項書換え系において, 最内の必須呼びを用いた場合の書換え回数を比較し, 入力の大きさ N に対する次数によって判断した. また, 変換の過程は TERSE のメインウィンドウ上で確認できる.

5.8 おわりに

単純で自動化に適した変換方法である, 可換則に基づく変換を提案した. 関数の帰納の位置が効率を決めているという考えのもとに, 単純化順序に基づき項の

順序を定め，小さい項により再帰が行なわれるような可換則変換適用アルゴリズムを定めた．このアルゴリズムにより，実行効率の良い項書換え系を得られることを例で示した．この際，項書換え系の効率は必須呼びに基づく書換え回数と，項のサイズによって判断した．さらに可換則変換に基づく自動変換システムを実現した．これまで人手で行なってきた変換がすべて自動的に行なうことが可能となり，その結果を目で確認することができるため変換を行なうことが非常に容易である．

本章の成果は，左線形，無曖昧でかつ停止性を持ち，可換則が成り立つ関数という厳しい制約に基づく変換であり，変換によって実行効率が向上することは例によってのみ示されている．しかしながら，個々の変換による効率の向上がなくとも，関数呼び出しの際の引数の関係を変換することにより，効率を向上させる本手法の考え方は，他の変換にも応用が効くと考えられる．[63] では各種の構造帰納法により，様々なアルゴリズムを導出している．今回の変換手法は，関数呼び出しの構造を変換しているとも考えられ，この応用として，構造帰納法そのものの呼び出し構造を変換するという手法への発展が期待される．

また，本章では効率の議論を必須な書換えに基づく書換え回数，項のサイズなどで行なったが，効率を判断する基準についても理論的成果を必要としている．

また自動変換システムについても，現在はまだ被覆集合を前もって与えるなどの処理を行なう必要があることや，結合則を用いた規則などの再帰経路順序では順序がつかない項書換え系が存在することもあり，まだ多くの改良の余地がある．

項書換え系の変換に関して，今後の課題を以下に挙げる．

1. 本変換手法によって実行効率が向上する項書換え系のクラスを求める
2. 可換則が成り立つ関数の判断手順を定める
3. 可換則以外の単純な変換規則を用いた変換について調査を行なう（結合則など）

第 6 章

結論

6.1 本論文のまとめ

本論文では，項書換え系に基づく形式的手法の限界を乗り越えるために，視覚化技術に注目して，項書換え系の解析・検証・変換のための視覚的支援手法を提案した．また，並列関数型言語を用いて本手法に基づく項書換え系の視覚的環境の実現を行なった．

最初に，項書換え系の解析や検証・変換などの形式的手法に対し，視覚化技術の導入により期待される以下の事項を示した．

- 視覚的表現による情報の直観的理解
- 計算系列中における誤りの出現位置の発見
- 実行効率改善の鍵の発見
- 高度な解析手法や様々な性質の直観的利用
- プログラミングやデバッグの視覚的実現
- 新たな解析手法や変換手法の獲得

本論文は，これらの事項を具体化するための手法の提案と環境の実現を行なった研究として位置付けられる．本論文の成果を以下にまとめる．

3 章では，まず，項書換え系に対して詳細な分析を行ない，その特徴に基づいて項書換え系の視覚化を項，計算，インタフェース，数値情報の視覚化として分類した．それぞれの視覚化について具体的な視覚化手法を定め，項書換え系の解析・検証・変換のための視覚的支援手法を提案した．この支援手法により，項書換え系で表現される計算に対し直観的な理解が可能になる．項や計算の視覚化により，従来のテキスト環境では発見することが困難であった誤りの出現位置の発見や変換の着目点等の直観的な発見が可能である．特に本論文で提案する計算の視覚化の枠組はこれまでのプログラミング支援のための視覚化手法では考えられておらず，項が計算状態を表す項書換え系の特長を生かした手法である．また，項の視覚化と数値情報の視覚化の併用により計算状態 (項の構造) と，その項のサイズやリデックスの数などの様々な数値の変化の様子を同時に理解することが可能になる．インタフェースの視覚化により，項書換え系のプログラミングやデバックにおいて，思考を妨げない直観的で効率的な操作が可能となる．また，具体例を用いて視覚的支援に基づく解析・検証・変換の様子を示し，本論文で提案した視覚的支援手法が項の構造の理解・解析や変換着目点の検出に有効であることを確かめた．さらに，項の視覚化をより高度化し，再帰経路順序の視覚化手法を定め，その視覚化に基づく停止性判定システムを提案する．

4 章では，本論文で提案する視覚化に基づく視覚的項書換え支援環境 TERSE の並列関数型言語 CML による実現を示した．強力なモジュール化の機能を持ち，マルチスレッド間でチャネルによる通信が可能な CML を用いることで，イベント駆動のユーザインタフェースが自然に実現できることを明らかにした．また，この実現により以下の事項を明らかにした．

1. 項書換え系の処理系が SML の機能により簡潔に実現できること．

2. 抽象木の導入により，モジュールの再利用性が向上すること．

3. SML の参照型を用いて，環境の実行中に視覚化手法を変更できる枠組が提

供できること.

4. 高い抽象度とモジュール性により保守，変更が容易なシステムが実現できること.

5. CML により実現された視覚的環境が実用的に十分な応答速度を持つこと.

特に SML のパラメータ化の機構の利用により，MVC 構造に基づく抽象度の高いモジュール化が可能になった．また，項を直接木構造図式として視覚化せず，抽象木を視覚化のためのモデルとして導入することにより，木構造の描画アルゴリズムを汎用的に記述することができた．さらに，本実現手法が計算モデルの視覚化システムの構築に有効であることを示すため，実際に多エージェント系自己認識論理のタブロー法による決定手続きを視覚化システムとして実現した．その結果，プログラムの多くの部分を TERSE と共有することが可能であり，少ない修正と追加によりシステムを完成させることができた．この事例研究により，本論文で用いた視覚的支援環境の実現手法が，他の計算モデルの視覚化システムの構築に有効であることを示せた.

5 章では，視覚化手法の有用性を確かめるため，項書換え系のプログラム変換の研究を取り上げ，可換則に基づく効率化変換の解析を行なった．可換則変換は項書換え系の規則において，関数の引数を入れ換えるだけの単純な変換手法であるが，書換え系列の視覚化を用いて，その有効性を容易に理解することができる．また，数値情報の視覚化により，変換の前後において項書換え系の計算が項のサイズに関してそれほど違いがなく，時間的（書換え回数）な効率の向上に対し，空間的な効率が悪化していないことが理解できる．この事例研究によって，項書換え系の具体的な形式的手法に対し，本論文で提案する項書換え系の解析・検証・変換のための視覚的支援手法が有効であることが確認できた.

なお，4 章で実現した項書換え系の解析・検証・変換のための視覚的支援環境は WWW および ftp によって公開している．項書換え系の視覚化環境 TERSE

のホームページの URL は,

```
http://www.inagaki.nuie.nagoya-u.ac.jp/terse/
```

TERSE の配布ファイルは,

```
ftp://jupiter.inagaki.nuie.nagoya-u.ac.jp
                              /nagoya-u/TERSE-2.0.1.tar.gz
```

である. また, TERSE の利用に関する簡単な説明を付録 D に示す.

6.2 今後の課題と展望

本論文が残した課題と将来の展望について述べる.

本論文で提案する項書換え系の解析・検証・変換のための視覚的支援手法は, 主にデータの視覚化, プログラムの視覚化に基づく手法で実現されている. 残る視覚化の分野として視覚的プログラミング, プログラムアニメーションがあるが, 本論文の提案する手法では十分に支援に用いられているとはいえない.

視覚的プログラミングとしては, 項の視覚的エディタの実現が第一に望まれる. 直観的かつ効率的な項や規則の入力が可能になることにより, 項書換え系のプログラミング支援ができるであろう. また, 図形を直接指示することにより例示によるプログラミングが可能である. 例えば, 計算系列上の繰り返しパターンの発見するため, 注目する構造パターンをポインタ等により指示し, 計算系列上に同じ構造が出現する場所を検出するプログラムの作成を行なうことや, 構造パターンの簡単化の方法を簡単化の例を用いて指示することなどが考えられる.

プログラムアニメーションとしては, 項の視覚化を拡張したものが考えられる. 本論文で実現した項書換え系の視覚化環境 TERSE では, 項の書換えを連続して行なう機能を持つため, 項の視覚化によりアニメーションに近い効果を得ることができる. しかし, この手法では, 書換えの前後における構造の変化を表示しないため, どの部分がどのように書換えられたを判断することは困難である. 書換えの前後における項の構造に注目し, その関係をアニメーションによって表すこ

とにより，項の構造が書換えられる様子の直観的理解が可能になる．

より高度で使いやすい視覚的環境を構築するためには，多くの理論的成果を利用しやすい形式で環境内に用意することが挙げられる．例えば，停止性判定手法においては，本論文で述べた再帰経路順序以外にも多くの手法 [31, 105, 69, 87, 88] が存在し，それぞれの研究では特徴的な順序が用いられている．これらの順序に関する視覚化を行なうことにより，項書換え系の停止性を判定するために利用する順序の選択が容易になることが期待される．また，完備化アルゴリズムや被覆集合帰納法などの検証手法についても視覚化手法の開発が望まれる．これらの視覚化の実現により，手法の直観的な理解が可能になり，より詳細な議論を行なうことができる．

将来の展望としては，項書換え系の視覚化システムに知的インタフェース技術を導入することが考えられる．これまで考えてきた形式的手法による支援は，論理的な根拠から正しい情報を人に与えることを目的としてきたが，必ずしも論理的根拠を必要としない支援手法も存在する．例えば，項をそのソートや関数の型によって色分けする場合，これまでは人の指示が必要であったが，ある程度までは慣習に基づく自動分類が可能であると考えられる．そこで，ユーザの色分けや図形の割り当ての好みを，学習理論等に基づいて学習を行ない，自動的に分類を行なうような支援手法が考えられる．また，様々な関数を実現する項書換え系やその性質に関する知識をデータベースとして保持すれば，項書換え系のモジュールの検索が可能になる．このデータベースを利用して，システムが積極的にユーザの支援を行なうことも可能になるであろう．例えば，ユーザがプログラミングを行なっている最中であっても，データベースから類似な構造を持つプログラムを検索し，ユーザに提示することによって，そのプログラミングの支援が可能となる．

謝辞

本研究を進め，まとめるにあたり，懇切丁寧な御指導と御鞭撻を頂いた名古屋大学教授の稲垣康善ならびに坂部俊樹両先生に心から感謝致します．

本論文をまとめるにあたり，貴重な御示唆と御指導を頂いた名古屋大学教授阿草清滋先生に深く感謝致します．

本研究の初期の段階から貴重な御意見を頂いた，当時名古屋大学助手で，現在，岐阜大学助教授の直井徹先生，北陸先端科学技術大学院大学助教授の酒井正彦先生に感謝致します．

日頃熱心に御指導下さった名古屋大学平田富夫教授，有益な御討論を頂いた中京大学外山勝彦助教授，名古屋大学杉野花津江助手，結縁祥治助手，馮速助手，山本晋一郎助手，濱口毅助手に感謝します．また稲垣・坂部研究室の多くの皆様から種々の教示と示唆を頂きました．記して感謝します．特に停止性判定のための再帰経路順序の視覚化を実現してくれた学部卒研生の大橋達哉氏，TERSE に対し書換え関係の視覚化を実現してくれた大学院生の大野健治氏に感謝します．

最後に，研究活動を行なうに当たりいろいろとお世話になった山岸慶子事務官に感謝します．

発表論文リスト

発表論文	関連する章
河口信夫, 坂部俊樹, 稲垣康善: “代数的仕様の解析・検証・変換のための視覚的支援環境” , ソフトウェア工学の基礎 I, レクチャーノート／ソフトウェア学 14, 近代科学社,pp 11-20(1995).	3 章
河口信夫, 坂部俊樹, 稲垣康善: “項書換え系の解析・検証・変換のための視覚的支援手法”, コンピュータソフトウェア, Vol.13, No.1, pp.23-36(1996).	3 章
河口信夫, 坂部俊樹, 稲垣康善: “関数型プログラミング言語 StandardML を用いた項書換え計算の視覚化の実現”, 電気学会論文誌 C, Vol.116-C, No. 1, pp. 103-110(1996).	4 章
Kawaguchi,N., Sakabe,T. and Inagaki,Y.: “TERSE: TErm Rewriting Support Environment” ,*Proceedings of the 1994 ACM SIGPLAN Workshop on Standard ML and its Applications*, pp. 91-100(1994).	4 章
Kawaguchi,N., Sakabe,T. and Inagaki,Y.:“TERSE : A Visual Environment for Supporting Analysis, Verification and Transformation of Term Rewriting Systems”,*in Proceedings of AMAST'96* , Lecture Notes in Computer Science, No. 1101, pp. 571–574(1996).	4 章
河口信夫, 坂部俊樹, 稲垣康善 : “可換則に基づく項書換え系の変換と必須呼びによる効率の評価” , 信学技法, COMP93-64(1993).	5 章

参考文献

[1] 秋口忠三: ユーザインタフェース管理システムと CASE 環境への適用, 情報処理, Vol. 33, No. 11, pp. 1314–1323(1992).

[2] Arya, K.: A functional animation starter-kit , *J. Functional Programming*, Vol.4, No.1, pp.1-18(1994).

[3] Brandenburg,F.J.: Nice Drawings of Graphs are Computationally Hard , *Visualization in Human-Computer Interaction* , Lecture Notes in Computer Science, No. 439, pp. 1–15(1990).

[4] Bundgen,R. : Reduce the Redex → ReDuX, *Rewriting Techniques and Applications*, Lecture Notes in Computer Science, No. 690, pp. 446–450(1993).

[5] Burstall,R.M. and Darlington, J. : A Transformation System for Developing Recursive Programs, *J.ACM*, Vol. 24, No. 1, pp. 44–67(1977).

[6] Chao,D.Y. and Wang,D.T.: An Interactive Tool for Design, Simulation, Verification, and Synthesis of Protocols , *Software Practice and Experience*, Vol. 24, No. 8, pp. 747–783(1994).

[7] Cox, P.: Prograph CPX: Visual Programming Applied to Industrial Software Development , *in Procceedngs of 11th IEEE Symposium on Visual Languages* , p. 2(1995).

[8] Cutler,E., Gilly,D., O'Reilly Tim : *The X Window System in a Nutshell* , O'Reilly & Associates Inc.(1992).

[9] Dershowitz,N.: Orderings for Term Rewriting System ,*Theoretical Computer Science*,Vol. 17, pp. 279–301(1982).

[10] Dershowitz,N. : Termination of Rewriting, *J. Symbolic Computation*, Vol. 3, pp. 69–116(1987).

[11] FUJI XEROX : *Interlisp-D* リファレンスマニュアル (1988).

[12] 二村良彦：構造化プログラム図式，コンピュータソフトウェア， Vol. 1, No. 1, pp. 64–77(1984).

[13] 二村良彦，野木兼六，高野明彦：ラムダ式の図形表現， *bit*, Vol. 22, No. 12, pp. 1281–1289(1990).

[14] 藤本洋, 豊島康文, 西山好雄, 深尾至: 通信ソフトウェア向き設計支援環境, 情報処理学会論文誌, Vol.31, No.7, pp.1113-1122(1990).

[15] Goldberg,A. and Robson,D.: *Smalltalk-80:the language and its implementation* , Addison-Wesley(1983).

[16] 後藤文太郎, 田中譲: VPE: 論理プログラミングにおける視覚的統合プログラミング環境, 情報処理学会論文誌, Vol.35, No.7, pp.1390-1401 (1994).

[17] 郷信義, 岸本美紀, 宮寺庸造, 岡田直行, 土田賢省, 夜久竹夫: Hichart プログラム図式の生成手法, 情報処理学会論文誌, Vol. 31, No. 10, pp. 1463–1473(1990).

[18] 萩原昌己: 視覚的プログラミングと自動プログラミング, コンピュータソフトウェア, Vol. 8, No. 2, pp. 27-39(1991).

[19] Harper,R. : Introduction to Standard ML, *Technical Report* ESC-LFCS-86-14, Department of Computer Science, Univ. of Edinburgh, (1986).

[20] 橋本治, 宮井均: ユーザインタフェースシミュレータ INTERA , 情報処理学会論文誌, Vol.31, No.10, pp.1497-1504(1990).

[21] 橋本正, 川端洋一: NextStep のプログラミング, bit, Vol.22,(1990).

[22] Huet,G. : Confluent Reductions:Abstract Properties and Applications to Term Rewriting Systems, *J.ACM*, Vol. 27, No. 4, pp. 797–821(1980).

[23] Huet,G. and Levy,J.J.: Call by need computation in non-ambiguous linear term rewriting systems ,*Rapport Laboria* 359, IRIA(1979).

[24] 市川至, 小野越夫, 毛利友治: プログラム可視化システム, 情報処理学会論文誌, Vol.31, No.12, pp.1801-1811(1990).

[25] 市川忠男, 平川正人 : ビジュアル・プログラミング, *bit*, Vol. 20, No. 4, pp. 404–412(1988).

[26] 今宮淳美: ユーザインタフェース管理システムと応用プログラムとの通信, 情報処理, Vol.33, No.11, pp.1304-1313(1992).

[27] 今宮淳美, 関村勉 : 言語モデルおよびＭＶＣ構造に基づくユーザインタフェース管理システム -GUIDMAS , 情報処理学会論文誌, Vol.31, No.4, pp.599-608(1990).

[28] 稲垣康善, 坂部俊樹：抽象データタイプの代数的仕様記述法の基礎 (1)-(4), 情報処理, Vol. 25, No. 1, No. 5, No. 7, No. 9, (1982).

[29] 岩永将幸, 大森健児：動的振舞いを用いたソフトウェア性能分析支援システムの構築, 情報処理学会研究会資料, SE-93-6 (1993).

[30] 神場知成, 橘本治: マルチビューモデルに基づくユーザインタフェース設計ツール U-face , 情報処理学会論文誌, Vol.34, No.1, pp.167-176 (1993).

[31] Kapur, D., Narendran, P. and Sivakumar, G.: A path ordering for proving termination of term rewriting systems, Lecture Notes in Computer Science, No. 185, pp. 173-187(1985).

[32] 勝山光太郎, 佐藤文明, 中川路哲男, 水野忠則: 国際標準形式記述技法に基づく体系的試験支援環境 FOREST の提案と実現, 情報処理学会論文誌, Vol.31, No.7, pp.1123-1133(1990).

[33] 河口信夫, 酒井正彦, 坂部俊樹, 稲垣康善 : 代数的プログラミング環境, 電気関係学会東海支部連合大会, 542 (1990).

[34] 河口信夫, 坂部俊樹, 稲垣康善 : 可換則に基づいた TRS の変換について, 電気関係学会東海支部連合大会, 613 (1993).

[35] 河口信夫, 坂部俊樹, 稲垣康善 : 可換則に基づく項書換え系の変換と必須呼びによる効率の評価, 信学技法, COMP93-64(1993).

[36] 河口信夫, 坂部俊樹, 稲垣康善 : 可換則に基づく項書換え系の自動変換システムとその評価, 情報処理学会第 48 回全国大会講演論文集, 4-271(1994).

[37] 河口信夫, 坂部俊樹, 稲垣康善：項書換え系の計算量解析の自動化, 電気関係学会東海支部連合大会, 611 (1994).

[38] 河口信夫, 坂部俊樹, 稲垣康善: グラフィカルユーザーインタフェースを持つ項書換え系の解析・変換支援環境, 信学技報, SS93-44(1994).

[39] Kawaguchi,N., Sakabe,T. and Inagaki,Y. : TERSE: TErm Rewriting Support Environment ,*Proceedings of the 1994 ACM SIGPLAN Workshop on Standard ML and its Applications*, pp. 91-100(1994).

[40] 河口信夫, 坂部俊樹, 稲垣康善: 視覚的項書換え環境の SML による実現, ソフトウェア科学会第 11 回大会論文集, pp.285-288 (1994).

[41] 河口信夫, 坂部俊樹, 稲垣康善 : 代数的仕様の解析・検証・変換のための視覚的支援環境, 第 1 回ソフトウェア工学の基礎ワークショップ FOSE'94 論文集, pp.9-16(1994).

[42] 河口信夫, 大橋達哉, 坂部俊樹, 稲垣康善: 項書換え系における再帰経路順序の視覚化, 電気関係学会東海支部連合大会, 637 (1995).

[43] 木下哲男, 岩根典之, 菅原研次, 白鳥則郎: 知識型設計方法論に基づくインタフェース設計法の形式化と設計支援システムの構成, 情報処理学会論文誌, Vol.31, No.6, pp.906-915(1990).

[44] Klop,J.W.: Term Rewriting Systems:a tutorial , *Note* CS-N8701, Centre for Mathematics and Computer Science(1987).

[45] 小池英樹: インタラクティブ 3 次元情報視覚化, コンピュータソフトウェア, Vol.11, No.6, pp.20-31(1994).

[46] 小池英樹, 石井威望: 3 次元ソフトウェア視覚化の枠組と実例による有効性の評価, 情報処理学会論文誌, Vol.33, No.6,pp.778-789 (1992).

[47] Knuth,D.E. and Bendix,P.B. : Simple Word Plobrems in Universal Algebras, *Computational Problems In Abstract Algebra*, Leech,L.(ed.), Pergamon Press, NewYork, pp. 263–297(1970).

[48] 桑原修二, 藤村茂, 富田昭司: MVC の拡張, MVCG とそれに基づく UMIS, Talkie の実装, コンピュータソフトウェア, Vol.9, No.1, pp. 27–41(1992).

[49] Lalonde,R.W. and Pugh,R.J. : *Inside SmallTalk*, Volume II, Prentice Hall,(1991).

[50] Lee,M.C.: An Algorithm Animation Programming Environment, Lecture Notes in Computer Science, No. 602, pp. 367–379(1992).

[51] Linos,P.K., Aubet,P., Dumas,L., Helleboid,Y., Lejeune,P. and Tulula,P. : Visualizing Program Dependencies, *Softw. Pract. Exper.*, Vol. 24, No. 4, pp. 387–403(1994).

[52] Matthews,B. : MERILL: An Equational Reasoning System in Standard ML, *Rewriting Techniques and Applications*, Lecture Notes in Computer Science, No. 690, pp. 441–445(1993).

[53] 松永泰明, 飯田元, 荻原剛志, 井上克郎, 鳥居宏次: 図式表現を用いたソフトウェアプロセス構成実行システムの試作, 電子情報通信学会論文誌, Vol.J76-D-I, No.6, pp.324-326(1993).

[54] Milner,R. : LCF: A Way of doing proofs with a machine , Lecture Notes in Computer Science, No. 74 ,pp. 146–159(1979).

[55] 三宅延久, 富樫敦, 野口正一: 3 層実行モデルによる AMLOG 実行モニタ, 人工知能学会誌, Vol.6, No.5, pp.690-700(1991).

[56] 宮下健, 松岡聡, 高橋伸, 米澤明憲: 複数の視覚的例による直接操作インターフェイスの対話的実現, コンピュータソフトウェア, Vol.11, No.6, pp.41-51(1994).

[57] 宮崎一哉: ユーザインタフェース管理システムと対話制御, 情報処理, Vol.33, No.11, pp.1295-1303(1992).

[58] 三末和男, 杉山公造: 図的思考支援を目的とした図の多視点遠近画法について, 情報処理学会論文誌, Vol.32, No.8, pp.997-1005(1991).

[59] 中島玲二：プログラム検証入門 (1) ～ (4) , bit, Vol. 12, No. 11 ～ 14,(1980).

[60] 直井徹, 山下雅史, 茨木俊秀, 本多波雄: 必須よびが正規化戦略となるあいまいな線形項書換えシステムのクラス, 電子情報通信学会論文誌,J69-D, No.9, pp. 1236–1245(1986).

[61] Naoi, T. and Inagaki,Y.: Algebraic Semantics and Complexity of Term Rewriting Systems , Lecture Notes in Computer Science, No. 355, pp. 311–325(1989).

[62] 西野哲朗: 属性グラフ文法とその Hichart 型プログラム図式に対するエディタへの応用, コンピュータソフトウェア, Vol.5, No.2, pp.81-92(1988).

[63] 野木兼六: 構造帰納法に基づくアルゴリズムの発見, コンピュータソフトウェア, Vol.7, No.4, pp. 39–59(1990).

[64] 布川博士, 富樫敦, 野口正一：図式をシンタックスに持つ関数型言語, コンピュータソフトウェア, Vol. 6, No. 2, pp. 11-23(1989).

[65] Ohlebusch, E.: Modular Properties of Composable Term Rewriting Systems , J. Sym. Comp., Vol. 20, pp. 1–41(1995).

[66] 大場 充: ソフトウェア信頼性モデル入門, 情報処理, Vol. 31, No.12, pp. 1623–1630(1990).

[67] 大堀淳： ML プログラミング (I) , コンピュータソフトウェア, Vol. 12, No. 1, pp. 3–15(1995).

[68] 岡田義広, 田中譲：視覚的シミュレータの開発支援システム, 情報処理学会論文誌, Vol. 32, No. 6, pp. 766–776(1991).

[69] 大崎人士, Aart Middeldorp, 井田哲雄: 意味ラベリングによる分配消去法 – 項書換え系の停止性証明法 –, コンピュータソフトウェア, Vol. 13, No. 2, pp. 58–73(1996).

[70] Open Software Foundation： *OSF/Motif User's Guide*,(1993).

[71] Partsch, H., Steinbrüggen, R.: Program Transformation Systems , *Computing Serveys*, Vol. 15, No. 3, pp. 199–236(1983).

[72] Polak,J. and Guest,S.P.: A Graphical Representation of the Prolog Programmer's Knowledge , *Visualization in Human-Computer Interaction* , Lecture Notes in Computer Science, No. 439, pp.82-104(1990).

[73] Quasar Knowledge Systems Corp.： *SmalltalkAgents Reference Guide*, (1993).

[74] Krishna Rao,M.R.K.: Completeness of Hierarchical Combinations of Term Rewriting Systems, Lecture Notes in Computer Science, No. 761, pp. 125–138(1993).

[75] 歴本純一: InformationCube: 半透明表示を用いた 3 次元情報視覚化技法, コンピュータソフトウェア, Vol.11, No.6, pp.63-74(1994).

[76] Reppy,J.H. and Gansner,E.R. : *The eXene Library Manual(Version 0.4)*, AT&T Bell Laboratories (1993).

[77] Reppy,J.H. : CML : A higer-order concurrent language, *Proceedings of SIGPLAN'91*, pp. 293–305(1991).

[78] Roman,G.C. and Cox,K.C.: A Taxonomy of Program Visualization Systems, *Computer*, Vol. 26, No. 12, pp. 11–24(1993).

[79] 佐藤豊, 板野肯三: 構造エディタとインタプリタの統括的記述とその生成系, コンピュータソフトウェア, Vol.4, No.2, pp.39-50(1987).

[80] 酒井正彦, 坂部俊樹, 稲垣康善 : 抽象データ型の代数的仕様の直接実現系 Cdimple, コンピュータソフトウェア, Vol. 4, No. 4, pp. 312–323(1987).

[81] 酒井正彦, 坂部俊樹, 稲垣康善 : 代数的仕様の検証のための被覆集合帰納法, 電子情報通信学会論文誌, Vol. J75-D-I, No. 3, pp. 170–179(1992).

[82] 佐藤正和, 橋本正明, 寺島信義: E-R モデルを用いた視覚的プログラミング言語: PSDL-GR とその一実現法, 情報処理学会論文誌, Vol.35, No.1, pp.115-126(1994).

[83] Schneider-Hufschmidt,M.: Integrating Visual Aids into an Object Oriented Programming Environment , *Visualization in Human-Computer Interaction* , Lecture Notes in Computer Science, No. 439, pp.123-136(1990).

[84] 芝野耕司: ノンテキスチュアル プログラミング, bit, Vol.23, No.1, pp.15-25(1991).

[85] 島和之，松本健一，鳥居宏次：モジュール交換手法によるマルチバージョンソフトウェアの信頼性向上，電子情報通信学会論文誌， Vol. J79-D-I, No. 9， pp. 558–566(1996).

[86] Sinclair,D.C.: Graphical User Interfaces for Haskell ,*Functional Programming*, Glasgow, Springer-Verlag, pp. 252–257(1992).

[87] Steinbach,J.: Extensions and Comparison of Simplification orderings ,Lecture Notes in Computer Science,No. 355, pp. 434–448(1989).

[88] Steinbach,J.: Simplification Orderings:Putting Them to the Test ,*Journal of Automated Reasoning* ,Vol. 10, pp. 389–397(1993).

[89] 杉浦佐江子, 大林正晴: ソフトウェア開発環境 dmCASE , 情報処理学会論文誌, Vol.31, No.7, pp.1091-1103(1990).

[90] 田中義憲, 直井徹 , 稲垣康善: 近似正規形に基づく項書換え系の unfold/fold 変換, 情報処理学会研究会資料, SF-38-6(1991).

[91] 高橋伸，宮下健，松岡聡，米澤明憲：アルゴリズムアニメーション作成システムにおける宣言的記述方法について，コンピュータソフトウェア, Vol. 11， No. 6， pp. 83–94(1994).

[92] 高濱徹行, 小倉久和, 中村正郎: テキスト型アプリケーション群を統合するグラフィカルユーザインタフェースシステム:XTSS , 電子情報通信学会論文誌, Vol.J77-D-I, No.7, pp.493-502(1994).

[93] 田中二郎，太田祐紀子： GHC プログラムの視覚的入力システム： FE'92, 信学技報， COMP90-67 (1990).

[94] 谷正之, 平沢宏太郎: ユーザインタフェース管理システムと制御システムへの適用, 情報処理, Vol.33, No.11, pp.1324-1330(1992).

[95] 舘村純一, 小池汎平, 田中英彦: 並列論理型言語 Fleng のマルチウィンドウデバッガ HyperDEBU , 情報処理学会論文誌, Vol.33, No.3, pp.349-359 (1992).

[96] Tomek,L. A., Muppala,J. K. and Trivedi,K. S.: Modeling Correlation in Software Recovery Blocks ,*IEEE Transactions on Software Engineering*, Vol. 19, No. 11, pp. 1071-1086(1993).

[97] Toyama,Y.:On the Church-Rosser Property for the Direct Sum of Term Rewriting Systems, *J. ACM*, Vol. 34,pp. 128–143(1987).

[98] Toyama,Y., Klop,J.W. and Barendregt,H.P.: Termination for the Direct Sum of Left-linear Term Rewriting Systems ,Lecture Notes in Computer Science, No. 308,pp.128-141(1988).

[99] Toyama, Y. : How to prove Equivalence of Term Rewriting Systems without Induction, *Theor. Comput. Sci.*, Vol. 90, pp. 369–390(1991).

[100] Tomas, G. and Ueberhuber,C.W.: Visualization of Scientific Parallel Programs , Lecture Notes in Computer Science, No. 771,(1994).

[101] 海野浩, 安斎公士, 小倉耕一, 西野哲朗, 中西美智子, 夜久竹夫: 木構造図式の描画問題, 情報処理学会論文誌, Vol.33, No.7, pp.879-886 (1992).

[102] 渡辺喜道, 今宮淳美, 三宅一巧: 仕様記述変換に基づく対話型ユーザインタフェース設計システム, 情報処理学会論文誌, Vol.34, No.7, pp.1589-1600(1993).

[103] 山本修一郎, 太田賢治: データフロー図の自動生成方式の提案, 電子情報通信学会論文誌, Vol.J76-D-I, No.10, pp.504-513(1993).

[104] 山本晋一郎，直井徹，坂部俊樹，稲垣康善：項書換えシステムにおける必須な書換えと戦略の効率，信学技報， COMP87-37 (1987).

[105] Zantema,H.:Total Termination of Term Rewriting is Undecidable, *J. Sym. Comp.*,Vol. 20, pp. 43-60(1995).

付録

A. CML による同期通信のプログラム例

```
fun test() = let
open CML;
 fun test_event() =
    let
        val myChan = channel()
val pr = CIO.print
fun sender() =
    let
val strId = tidToString (getTid())
    in
(pr ("Send"^(strId)^"\n");
 send(myChan,strId);
   pr ("Send"^(strId)^"Done\n"))
    end
fun receiver() =
    let
val myId = tidToString (getTid());
    in
```

```
pr ("Recv"^myId^"\n");
pr ("Recv"^myId^".."^(accept myChan)^"\n")
    end
    in
spawn sender;spawn receiver;
spawn sender;spawn receiver;()
    end
     in
 RunCML.doit(test_event,SOME 20)
end
```

実行例

```
Concurrent ML -- version 0.9.8 -- February 1, 1993
Standard ML of New Jersey, Version 0.93, February 15, 1993
val it = () : unit
- use ``res.sml'';
..
- test();
Send[9]
Recv[10]
Send[11]
Recv[10]..[9]
Recv[12]
```

```
Send[9]Done
Recv[12]..[11]
Send[11]Done
val it = () : unit
```

B. マッチング関数

```
(* 項のマッチングを行なう関数 match *)
(* 2つの項と代入を受けとり,
   マッチングの成否と変数への代入を返す *)
(* val match : term * term * substs
                        -> bool * substs *)
fun match (Var(s),t1,sub: substs)
          = (true,(s,t1)::sub)
  | match (Fun(s,t0),Var(x),sub)
          = (false,nil)
  | match (Fun(s,t0),Fun(t,t2),sub) =
   if t = s then
    let
     fun matchSub(nil,nil,sbs) = (true,sbs)
       | matchSub(nil,h::t,sbs) = (false,nil)
       | matchSub(h::t,nil,sbs) = (false,nil)
       | matchSub(h::t,h1::t1,sbs) =
         let val (b2,sb2) = match(h,h1,sbs)
         in if b2 then matchSub(t,t1,sb2)
           else (false,nil)
         end
      in
         matchSub(t0,t2,sbst)
    end
    else (false,nil);
```

C. 木構造図式描画関数

```
(*    木構造を描画する関数 drawTree  *)
type point = int * int
type rect  = int * int * int * int
(*    ノード間の垂直距離
val Vwid : int
   object を左から描画する関数。図形の幅と領域を返す。
val drawObj  : object * point -> int * rect
   object を中心から描画する関数。図形の領域を返す。
val drawCObj : object * point -> rect
   ノード間の線分描画関数
val drawLine : rect * rect -> unit
   抽象木，描画位置 を引数とし，
   木の幅，木のルートノードの描画領域 を返す。
val drawTree : tree * point -> int * rect       *)

fun drawTree(Leaf(obj),pt) = drawObj(obj,pt)
  | drawTree(Node(obj,tlist),(x,y)) =
   let
    fun drawTrees(nil,sizes,rl) = (sizes,rl)
      | drawTrees(hd::tl,s,rl) =
       let
        val (ss,rect) = drawTree(hd,(x+s,y+Vwid))
       in
        drawTrees(tl,s + ss,rect::rl)
```

```
      end
    val (size,rlist) = drawTrees(tlist,0,nil);
    fun drawLines p = map (fn x => drawLine(p,x));
    val rect = drawCObj(obj,(x+(size div 2),y));
  in
    (drawLines rect rlist;(size,rect))
  end
```

D. 視覚的項書換え支援環境 TERSE 導入の手引

この付録では，TERSE の配布元からの入手法について述べる．

TERSE を動作させるために必要な環境では，以下の言語／ライブラリが動作する必要がある．

```
Standard ML of New Jersey(SML/NJ) 0.93
Concurrent ML 0.9.8
eXene 0.4
```

これらの言語の入手は

```
ftp://research.att.com/dist/ml/
```

ミラーサイト

```
ftp://ftp.nuie.nagoya-u.ac.jp/languages/sml-nj/
```

から可能である．インストールは各 README に従って頂きたい．

また，項書換え系の視覚化環境 TERSE のホームページの URL は，

```
http://www.inagaki.nuie.nagoya-u.ac.jp/terse/
```

TERSE の配布ファイルは，

```
ftp://jupiter.inagaki.nuie.nagoya-u.ac.jp/nagoya-u/
  (ソースファイル)        TERSE-2.0.1.tar.gz
  (Sun Soralis バイナリ) TERSE-2.0.bin.Soralis2.3.gz
```

である．インストールは README を読んで頂ければ，スクリプトによってほぼ自動的に実行される．

現在，TERSE の動作が確認されている機種／OS は

```
Sun /  SunOS 4.X.X
       Soralis 2.X
PC-AT/ FreeBSD 2.X.X
```

である．Sun のSoralis2.3 については実行可能バイナリを配布しているので，こちらを利用することにより容易に導入が行なえる．

本システムに関する，御意見，御質問，御感想，コメント等は

kawaguti@nuie.nagoya-u.ac.jp（河口 信夫）

まで e-mail にてお送り下さい．