論　文

# 関数型プログラミング言語 Standard ML を用いた項書換え計算の視覚化の実現

非会員　河口　信夫　(名古屋大)
非会員　坂部　俊樹　(名古屋大)
正　員　稲垣　康善　(名古屋大)

Implementing Visualization of Term Rewriting Computation in Standard ML

Nobuo Kawaguchi, Non-member, Toshiki Sakabe, Non-member
*and* Yasuyoshi Inagaki, Member (*School of Engineering, Nagoya University*)

Term Rewriting System (TRS) is one of the simplest computation models for functional programming languages. It can also be used to model term manipulation required in program verification and transformation. In such applications, an environment for term rewriting with user friendly graphical interface is strongly required for analyzing structure of terms and rewriting processes. Most TRS interpreters developed so far are text-based ones, and hence they do not provide sufficient supports for analyzing structure of terms. We have proposed a new idea for a *term visualization* based on Graphical User Interfaces(GUI) [11] [12]. This paper shows how to realize the idea with the functional programming language Standard ML(SML). We illustrate the implementation can be easily modified and extended since the rewriting part and GUI part of the program are clearly separated owing to the module system of SML.

キーワード: 項書換え系，視覚化，視覚的支援環境，関数型言語，ユーザインタフェース

## 1. まえがき

項書換え系 (TRS:Term Rewriting Systems) は項の書換えを基本とする関数型言語の計算モデルであり，直観に優れた代数的意味論を持つ。TRS はその形式の単純さからプログラムの検証や変換の分野で注目されており，被覆集合帰納法による定理証明 [22]や Knuth-Bendix アルゴリズム [13]による完備化を用いた検証など，多くの成果が報告されている。実際に TRS をシミュレートして検証や変換を行なう処理系も多く実現されている [1] [15] [23]。しかし，ほとんどの処理系はテキスト処理を用いているため，項の構造の解析や変換着目点の発見など，人の直観を必要とするような要求に十分に応えられないことが多い。その上，表示の複雑さや操作の繁雑さから各々の処理系の機能を十分に活用できていない。

これらの問題の解決法として，著者らは書換え計算を図形を用いて視覚的に表現し，操作にグラフィカルユーザインタフェース (GUI) を用いて，直観的な計算の理解や操作を可能にすることを提案している。これを実現するためには (1)「書換え計算の解析に必要な視覚化を提案し」(2)「視覚的環境を計算機上に実現する」ことが必要である。(1) についてはすでに [11] [12]で述べている。この中で著者らは項書換え系の計算を解析する上で必要な情報を詳細に検討し，項，計算，情報，操作の 4 種類の視覚化を提案している。また，実際にこの視覚化を用いて項書換え系の解析・変換を行ない，著者らが提案した支援手法の有効性を確かめている。

本稿では特に (2) について，すなわち「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。この環境は項書換え系の解析が主な目的であるため，実行効率はそれほど問題ではなく，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。著者ら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) [18]を並列化した Concurrent ML(CML) [21]および eXene [20]ライブラリを採用した。環境の設計においては Smalltalk-80 [3]の MVC(モデル・ビュー・コントローラ) 構造 [14]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。

この実現手法に基づき，著者らは項書換え支援環境 TERSE 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。

以下，2章でTRS，SML，CML，eXeneについて簡単に解説し，3章で著者らが提案している視覚化手法について述べる。4章では，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。5章では変更容易性，再利用性および応答速度について評価を行なう。また，6章では視覚化システムの実現において，本研究の手法と，他の視覚的システム構築ツールを用いた場合とを開発効率の観点から比較する。

## 2. 準　備

**〈2·1〉 項書換え系**　本節では項書換え系の解説を簡単に行なう。詳細な定義は[5]等を参照されたい。

項は変数記号 $(x,y,z,\ldots)$ と関数記号 $(f,g,add,s,0,\ldots)$ により構成される記号列である。例えば $x, s(0), add(0,f(x))$ などはすべて項である。項を木とみなすと，ルートに対するノードの相対的位置をそのノードの出現という。項書換え系は左辺項と右辺項からなる有限個の書換え規則により定められる。ただし規則の右辺項に現れる変数は左辺項に必ず現れる。書換え対象の項の一部分と規則の左辺が変数への代入により一致した場合，その出現をリデックスと呼ぶ。リデックスの存在しない項を正規形と呼ぶ。任意のリデックスを一致した規則の右辺に同じ代入を施した項に置き換えることを1ステップの書換えと呼び，その繰返しが項書換え系の計算である。例えば加算を行なう項書換え系Rは2つの書換え規則のみにより構成され，以下のように表される。

$$\mathrm{R} = \begin{cases} add(0,x) & \rightarrow \quad x \\ add(s(x),y) & \rightarrow \quad s(add(x,y)) \end{cases}$$

ここで $x$ は変数記号であり，$s(x)$ は $x+1$ という値を表す後者関数である。例えば $s(0)$ は1，$s(s(0))$ は2を表す。また $add(x,y)$ は $x+y$ を表す関数である。$1+2$ の計算は以下の書換え系列で表される。

$$\begin{aligned} add(s(0),s(s(0))) &\rightarrow s(add(0,s(s(0)))) \\ &\rightarrow s(s(s(0))) \end{aligned}$$

得られた結果は $s(s(s(0)))$ すなわち3である。このように項書換え系は記号の書換えのみで計算が進む単純で記述性，読解性に優れた計算モデルであり，多くの研究に基づく豊かな研究成果を持つ。

**〈2·2〉 Standard ML**　Standard ML(SML)[18]はEdinburgh LCF[16]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，シグネチャ(`signature`)とストラクチャ(`structure`)による抽象化の機構と，ファンクタ(`functor`)を用いたパラメータ化の機構が挙げられる。モジュール間のインタフェースはシグネチャによって記述され，実現はストラクチャで記述される。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現実のプログラミングに耐えられ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。

**〈2·3〉 Concurrent ML**　Concurrent ML(CML)[21]はStandard ML(SML/NJ)を拡張した並列関数型言語である。CML上ではlight weight processである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 `'a chan`,`'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。同期をサポートするeventの導入により，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。

**〈2·4〉 eXeneライブラリ**　eXene[20]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。

## 3. 項書換え系の視覚化

著者らは項書換え系の計算を解析する上で必要な情報について詳細に分析した結果，項，計算，情報，操作の4種類の視覚化を提案した。ここでは各々の視覚化について簡単に述べる。視覚化や支援手法の詳細については[11][12]を参照されたい。

図1は**項の視覚化**を実現したTerm Viewerである。項の視覚化は，項を型やリデックスなどの情報を持つ木構造と

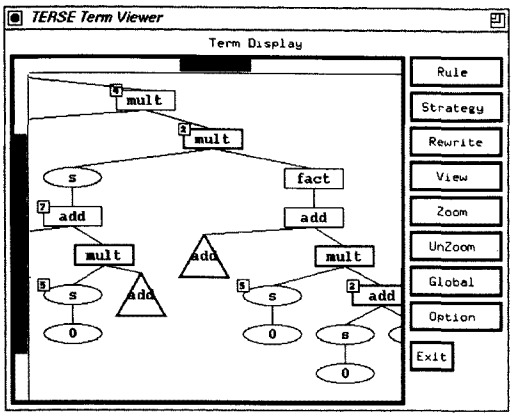



図1 項の視覚化
**Fig. 1. Term Visualization.**

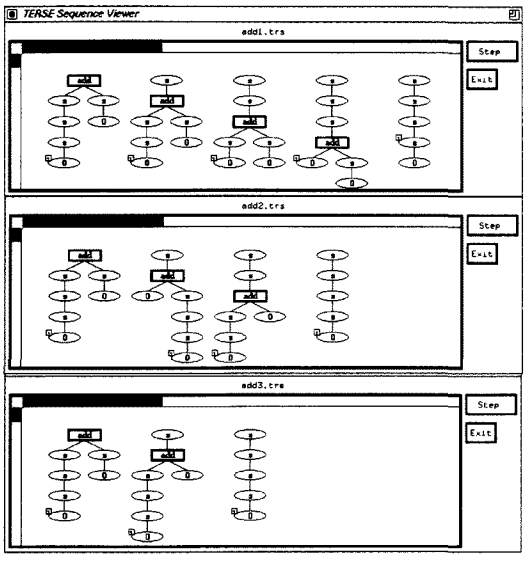



図2 計算の視覚化
**Fig. 2. Computation Visualization.**

して描画する。各々のノードは長方形や楕円などの形状と色で関数の型やソートが表され，三角形は部分木を表している。リデックスは太い枠線で表され，出現のマークが数字でつけられている。Term Viewerは図形の任意の位置を適当な倍率で見ることができるようにスクロールバーや拡大，縮小の機能を持つ。項の視覚化により，項の構造を理解できるだけでなく，ノードの形状や色から直観的に関数の型やソートも把握することができる。

図2は**計算の視覚化**を実現したSequence Viewerである。計算の視覚化は，計算系列の各々の項に対して項の視覚化を行ない，系列として同時に描画する。図2は同じ項に対して異なる規則を用いて加算の計算を行なった場合を表している。計算の視覚化により項書換え系の計算の進行状況を直観的に理解できる。実際，計算系列を視覚的に解析するこ

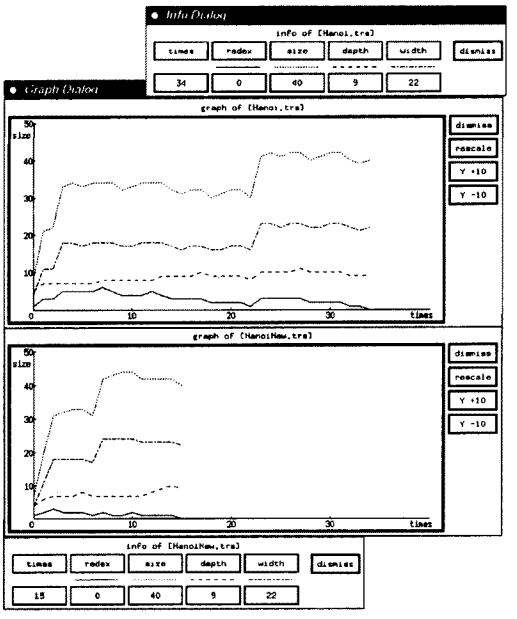



図3 情報の視覚化
**Fig. 3 Information Visualization.**

とで効率の評価を行なったり，fold/unfold変換などの変換着目点を直観的に発見する試みが行なわれている[10][11]。

図3は**情報の視覚化**を実現したGraph DialogおよびInfo Dialogである。情報の視覚化は項書換え系の計算の解析に必要な様々な数値，例えばリデックスの数や項の大きさなどの数値情報を書換えと同時にグラフ化する。この図ではプログラム変換の前後での書換え系列の情報が視覚化されている。上のTRSが下のTRSより2倍程度の書換え回数を必要とすることが直観的に理解できる。視覚化の対象になる情報は環境の実行中にユーザが自由に変更できるため，必要な情報を得ることができる。

**操作の視覚化**は項，計算，情報のそれぞれの視覚化に対してのユーザからのマウスなどによる直観的な操作を実現する。図1のTerm Viewerでは木構造の図形をマウスの直接操作により書換え対象の項の直接指示や，表示形式の変更，出現に対するマーク付けなどを行なう機能が実現されている。

## 4. 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[18]を並列化したConcurrent ML[21]およびeXene[20]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[3]で提唱されたMVC(Model, View, Controller)構造[14]を採用した(図4)。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[6][17]，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，本稿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5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。

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[18]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。

**⟨4·1⟩ 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。

**⟨4·1·1⟩ 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   *)
```

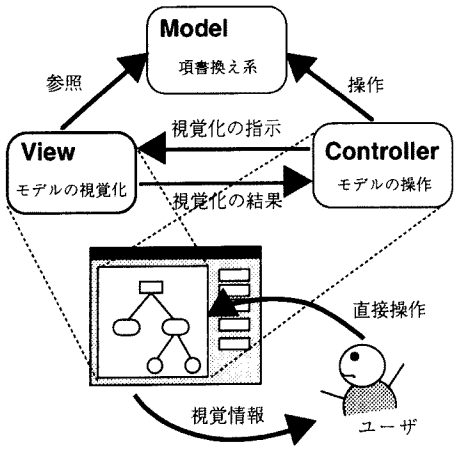



図4 MVC構造
Fig. 4. MVC structure.

```
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。例えば項と項のマッチングを行なう関数matchの定義を図6に示す。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。また，'let xxx in yyy end'は宣言xxxがyyy内で有効であることを表し，値としてyyyの評価結果を持つ。'::'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。

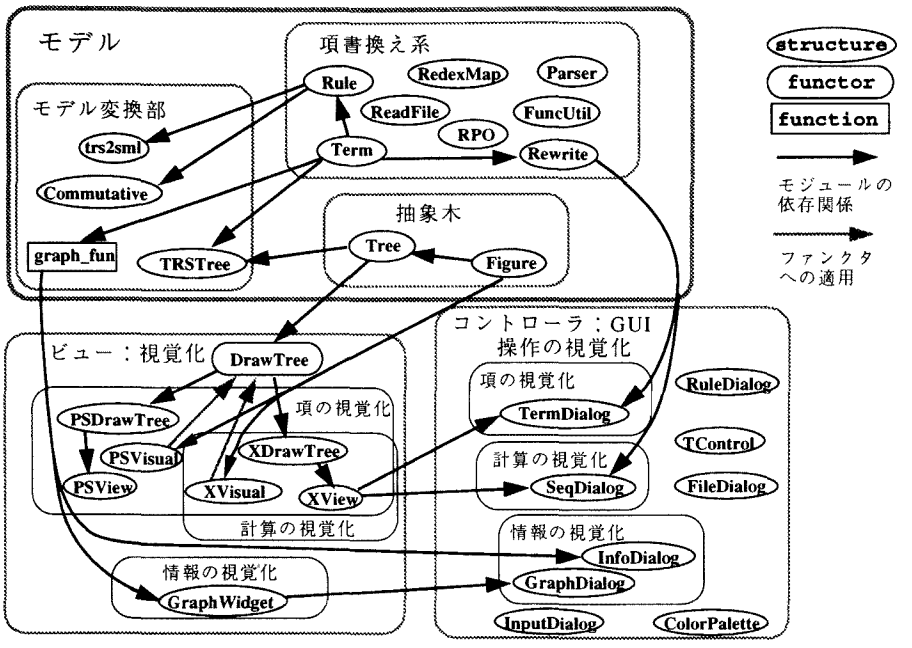



図5 モジュールの依存関係
**Fig. 5. Module dependency.**

```
(* 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);
```

図6 マッチング関数の定義
**Fig. 6. Definition of matching function.**

**⟨4·1·2⟩ 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。

**⟨4·1·3⟩ 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 [7] [10]
- **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 [7] [10]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。

**⟨4·2⟩ 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。

**⟨4·2·1⟩ 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7に簡略化した木構造描画関数drawTreeの定義を示す。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。

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。

**⟨4·2·2⟩ 情報の視覚化**　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。

```
(*   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
```

図7 木構造の描画関数

**Fig. 7. Draw function of tree structure.**

**⟨4·3⟩ 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。

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

表1 書換えの実行時間
Table 1. Execution time of rewriting.

| 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 |

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。

## 5. 評　価

**〈5·1〉 変更容易性，再利用性**　図5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。このことにより高度な再利用性，保守性を実現することができた。

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，本稿の実現手法の有効性が確かめられた。

**〈5·2〉 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。

## 6. 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。

**〈6·1〉 Xlib,Xtoolkit**　X Windowシステムでは Xlib，Xtoolkit [2]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言語よりも平均して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。

**〈6·2〉 Motif, NextStep**　Motifライブラリ [19]やNextStep Developmentシステム [4]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。

**〈6·3〉 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。

## 7. むすび

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。また，他の視覚的システム構築ツールとの比較を行ない，速度的な問題は若干あるが，開発効率の点や抽象度の点では非常に有利であることを示した。

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。その具体的な実現は今後の課題である。

御討論頂いた中京大学外山勝彦助教授，名古屋大学山本晋一郎助手，並びに研究室の皆様に感謝致します。

(平成7年5月1日受付，同7年8月14日再受付)

### 文　献

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

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

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

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

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

[6] 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).

[7] 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，信学技報，COMP93-64(1993).

[8] 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).

[9] 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).

[10] 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).

[11] 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).

[12] 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掲載予定(1995).

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

[14] Lalonde,R.Wilf and Pugh,R.John：*Inside SmallTalk*, Volume II, Prentice Hall,(1991).

[15] 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)

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

[17] 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).

[18] 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).

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

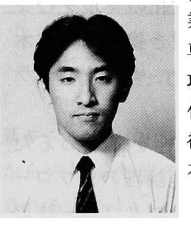
[20] Reppy,J.H. and Gansner,E.R.：The eXene Library Manual(Version 0.4), AT&T Bell Laboratories (1993).

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

[22] 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，信学技報，COMP90-5 (1990).

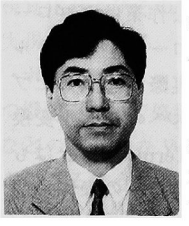
[23] 山本晋一郎，直井徹，坂部俊樹，稲垣康善：項書換えシステムにおける必須な書換えと戦略の効率，信学技報，COMP87-37 (1987).

**河口 信夫**



(非会員)1990年名古屋大学工学部電気学科卒業。1995年同大学大学院工学研究科情報工学専攻博士課程満了。同年同大学工学部助手。項書換え系，関数型プログラミング，代数的仕様記述の研究に従事。現在はプログラムの視覚化に興味を持つ。電子情報通信学会，日本ソフトウェア科学会各会員。

**坂部 俊樹**



(非会員)1972年名古屋大学工学部電気学科卒業。1977年同大大学院博士課程了。三重大学助教授，名古屋大学助教授を経て，平成5年より名古屋大学教授。抽象データ型の理論，プログラミング言語の形式的意味論，書換え型計算モデル，並行プロセスの理論などの研究に従事。工学博士。情報処理学会，電子情報通信学会，人工知能学会，日本ソフトウェア科学会，EATCS各会員。

**稲垣 康善**

(正員)1962年名古屋大学工学部電子工学科卒業。1967年同大大学院博士課程修了。同大助教授，三重大学教授を経て，1981年より名古屋大学工学部教授。工学博士。オートマトン・言語理論，ソフトウェア基礎論，代数的仕様記述法などの研究に従事。電子情報通信学会，情報処理学会，人工知能学会，日本ソフトウェア科学会，IEEE，ACM，EATCS各会員。