木上の双方向変換を利用したファイルマネージャの実現

松田一孝[†] 大川徳之[†] 野村芳明[†]
森田直幸[†] 筧一彦[†]
胡振江[†] 武市正人[†]

ファイルの操作において，ショートカットまたはシンボリックリンクといった別名の存在は，しばしばユーザを混乱させる．初心者にとってファイル実体と別名との区別は難しい，たとえば通常のファイルマネージャでは，ファイル実体を削除してもそれを指し示している別名は消えずに残り，逆に別名を削除しただけではファイル実体は削除されない．そこで，我々はファイルへの参照すべてを対称かつ統一的に抽象化して扱うファイルマネージャを，木上の双方向変換の技術を用いて実現する．我々のファイルマネージャの上では，1 つのファイルへの複数の参照は互いに同期されており，1 つを変更すると必ず同期された別の部分に伝播される．たとえば，1 つをディレクトリ木から削除すると残りも削除される．さらに，双方向変換を利用することにより，ディレクトリ木からの情報を抽出し操作して表示するなどのディレクトリ木の “見せ方” を抽象化および拡張でき，またその “見せ方” 自体も変更することができる．このような “見せ方” の操作やファイル木に対する編集操作は，GUI を通して対話的に実行することができる．本論文では，この木上の双方向変換を利用したファイルマネージャの設計とその実装とを示す．

A Synchronizable File Manager Based on Bidirectional Tree Transformations

Kazutaka Matsuda,[†] Noriyuki Ohkawa,[†] Yoshiaki Nomura,[†]
Naoyuki Morita,[†] Kazuhiko Kakehi,[†] Zhenjiang Hu[†]
and Masato Takeichi[†]

File management is sometimes confusing due to the difference between files and their shortcuts (or symbolic links). While deleting shortcuts to a file can leave the file entity as it is, deleting a file entity may lead to dangling shortcuts. To resolve this problem, we apply the technique of bidirectional tree transformations to design and implement a new file manager which can treat file references in a symmetric and uniform way. This file manager is unique in its synchronization mechanism which eliminates the confusing difference of file references. Plural file references are synchronized, and changes applied on one reference will propagate to the others. For example, if one of the synchronized file references is deleted, the rest are also deleted automatically. This synchronization mechanism can be efficiently implemented based on bidirectional tree transformations. In this paper, we show the design and the implementation of our file manager. The interactive graphical user interface of our file manager enables us to manipulate file references easily, in which bidirectional tree transformations can produce complicated dependency.

1. はじめに

ファイルの操作において，ショートカット，シンボリックリンクといったファイル別名の存在は，しばしばユーザを混乱させる．初心者にとってファイル実体とそれに対する別名の区別は難しいため，メールへの添付やメディアへのバックアップの際にファイル実体でなく別名を用いてしまうことがある．また，別名と実体の関係は非対称であり，ファイル別名を消してもファイル実体は削除されず，逆にファイル実体を消すと無効なファイル別名が残る．アプリケーションのアンインストールを行う際に，ファイル別名のみを削除し，ファイル実体を削除しないという失敗は少なくない．実際に Microsoft Windows のファイルマネージャであるエクスプローラでは，このような失敗を防ぐため，デスクトップなどに存在するファイル別名を消去する際に，ファイル実体が削除されない旨の確認ダイ

† 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo

アログが出る．また，一般にユーザにとって，あるファイル実体を参照しているすべてのファイル別名を列挙するのは容易ではないため，ファイル実体を消去した後に，存在しない実体を指し示している無効なファイル別名が残ったままとなりやすい．

そこで本研究では，ファイルへの参照すべてを対称かつ統一的に抽象化して扱うことのできるファイルマネージャ「梅林」を，木上の双方向変換[8]の技術を用いて実現した．木上の双方向変換とは，ソースとなる木からターゲットとなる木への変換を特定の言語[6),8),10)]によって記述することにより，ターゲットとなる木に加えた変更を自動的にソースの木に反映させる手法である．複数の場所から 1 つのファイルを参照したい場合，既存のファイルマネージャはファイル実体と別名，つまりファイル名であるファイル参照とファイル参照への参照を用いる．これに対し，ファイルマネージャ「梅林」では，あるファイル参照を双方向変換により複数のファイル参照に変換する方法を用いる．この複数のファイル参照は互いに同期され，1 つに加えられた変更は別の同期されたファイル参照に反映される．たとえば，あるファイル参照の名前を変更すると同期されたファイル参照の名前も変更され，あるファイル参照の指す実体を削除すると同期されたファイル参照はすべて削除される．ファイル実体を削除せずに同期されたファイル参照の 1 つを削除したい場合には，双方向変換によって同期されたファイル参照を “見えなく” することにより，ファイル別名の削除に対応する操作ができる．このように同期されたファイル参照は互いに対称に扱われ，無効なファイル別名を生じない．

また，既存のファイルマネージャは実際のディレクトリ木の見せ方に関する自由度が低い．ファイルに対する注釈，表示するファイルの順序，特定のファイルの隠蔽などいくつかの基本的な機能は提供されているが，ユーザはそれらを細かにカスタマイズすることはできない.「梅林」では，このような “見せ方” を双方向変換として記述する．そのため，これらの機能を統一的に表現でき，なおかつ見せ方の自由度を高めている．

「梅林」の特長は以下のようなものである．

見せ方の統一的な抽象化 ファイルマネージャは，ファイルへの複数の同期された参照，ソート，お気に入りリストなどの様々な機能を持つ.「梅林」はそのような表示に関する様々な機能を双方向変換を利用して統一的に扱うことができる．

ファイル実体の操作と見せ方の操作 「梅林」では新規作成，削除，名前変更などのディレクトリ木に関する操作とソート順の変更，同期されるファイル参照の追加などの見せ方に対する操作を提供している．これらの操作は GUI を通して対話的に行うことができる．

拡張性 「梅林」では，ファイルへの複数の同期された参照やソートなどの機能を双方向変換を利用して実現している．新たな見せ方に関する機能も，ユーザが双方向変換を用いて記述することにより，ファイル参照の同期関係を保ったまま追加することができる．

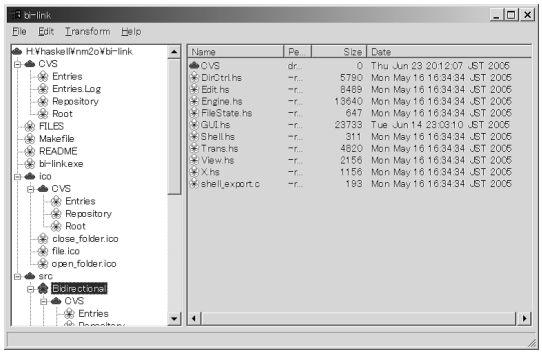


図 1 「梅林」のスクリーンショット
Fig. 1 Screenshot of the bi-link system.

本論文は以下のように構成される．2 章では,「梅林」のファイルマネージャとしての機能の概観を示す．3 章では,「梅林」で用いられている木上の双方向変換技術について述べ，4 章で「梅林」の設計および実装の詳細を述べる．5 章では，関連研究および今後の課題について議論する．6 章で本論文の内容のまとめを行う．

2. 梅林概要

図 1 が本研究において開発された双方向変換ファイルマネージャ「梅林（bi-link）」のスクリーンショットである．本章ではこの「梅林」の特徴的な機能および活用例について説明する．

2.1 機能説明

「梅林」では，ディスク上のディレクトリ木に対し，ある変換によって見せ方を変更したものがビューとして表示され，ユーザはこのビューを通して様々な操作を対話的に実行することができる．ビュー上で実行できる操作には，ディレクトリ木そのものを変更する操作と，ビューのみを変更する操作とがある．ビューのみを変更する操作では，ディレクトリ木の見せ方のみが変更され，ディレクトリ木自体は変更されない．これらの操作によるディレクトリ木の変更や見せ方の変更をもとに，新たなビューが生成される．

表 1 梅林の提供する操作
Table 1 Operations provided by bi-link.

ディレクトリ木に対する操作	
Delete	ファイルおよびディレクトリの削除を行う
New File	新規ファイルを作成する
New Directory	新規ディレクトリを作成する
Rename	ファイルおよびディレクトリの名前を変更する

ビューに対する操作	
Hide	指定されたノードを表示されないようにする
Duplicate	複製先を指定し，指定されたノードが 2 箇所に表示されるようにする
Sort by Name	指定されたディレクトリの内容を名前でソートして表示する
Sort by Size	指定されたディレクトリの内容を容量でソートして表示する
Sort by Time	指定されたディレクトリの内容を更新日時でソートして表示する

「梅林」の提供する操作を**表 1** に示す．提供される操作のうち，特徴的なのは Duplicate である．Duplicate は同期された 2 つのファイル参照を作る機能である．Duplicate により同期された複数のファイル参照に対し，一方に加えたディレクトリ木に対する操作は必ずもう一方にも反映される．また，ディレクトリ木に対する操作とビューに対する操作とが異なる例として Hide と Delete があげられる．ビューのみに対する操作である Hide では指定されたファイル参照またはディレクトリ参照がビュー上で見えなくなるだけで実体は残り，ディレクトリ木に関する操作である Delete と異なる．

2.2 活 用 例

「梅林」では双方向変換を用いることにより，従来のファイルマネージャの持つ問題を解決できるだけでなく，様々な機能を統一的に扱うことができる．

2.2.1 同期されたファイルの追加

Duplicate により同期された 2 つのファイル参照は，ァイル別名と同様の機能を実現でき，また 2 つは同等に扱われる．そのため，実体と別名の取り違えや無効な参照といった煩わしい問題に悩まされることがなくなる．

この様子を**図 2** に示す．ビュー（A）においてファイル a をディレクトリ dir 以下に，Duplicate を用いて同期されたファイル参照を追加することでビュー（B）が得られる．ビュー上で 2 つの a は同期されており，どちらの a に対する操作も両方に反映される．したがって，たとえば Rename により一方の a の名前を b に変更に変更すると，両方の名前が b に変更され（ビュー（C））, Delete により一方の a を削除すると両方の a が削除される（ビュー（D））．一方のみをビュー上から削除するには，削除するノードに Hide を適用すればよい（ビュー（E））.

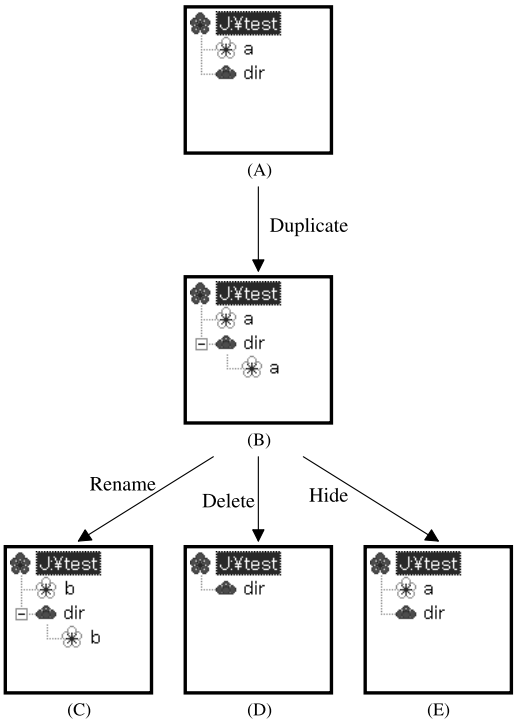


図 2 Duplicate により同期された参照
Fig. 2 Synchronized file references by Duplication.

2.2.2 お気に入りリスト

別名機能は特定のファイルやディレクトリへの簡単なアクセスを可能にする．したがって，お気に入りのファイルや使用頻度の高いディレクトリなど，頻繁にアクセスする箇所へのファイルやディレクトリのファイル別名を集めたお気に入りリストは非常に便利である．

「梅林」で “HotTopics” のような機能を実現するには，ホームディレクトリなどのアクセスしやすい場所にディレクトリを作成し，ここに対象ファイルの複製を集めるだけでよい．分類し整理されたリンク先の

構造を変えることなく好きなファイルを同じ場所に集めることができるだけでなく，複製元のディレクトリ木に対する操作がお気に入りリストの方にも同期される点で従来のファイルマネージャより優れている．

2.2.3 表示のカスタマイズ

「梅林」では従来のファイルマネージャよりも自由度の高い表示のカスタマイズが可能である．たとえば，ビューに対する操作は同期された複数のファイル参照に対してもそれぞれ個別に適用することができるため，あるディレクトリに Duplicate を適用し，一方に Sort by Name を，もう一方には Sort by Time を適用した場合，同じディレクトリの内容が名前でソートされた様子と更新日時でソートされた様子を両方同時に確認することができる．

2.2.4 様々な分類によるファイルの整理

ここまでの活用例の具体的な応用が様々な分類の仕方によるファイルの整理である．たとえば音楽ファイルの整理のように，ジャンル，アーティストなど，様々な要素に基づいてファイルを分類したいことはしばしばある．しかし，木構造を利用した階層構造に基づく整理ではそれらのうち 1 つの要素によってしか分類を行うことができない．ある分類によってディレクトリ分けしたものと別の分類によってディレクトリ分けしたものの 2 つのコピーをつくってしまうと，ファイルの名前変更，削除によって生じる両者の内容のくい違いに悩まされることになる．

Duplicate を用いることにより，このような問題も簡単に解決できる．音楽ファイルをジャンル別やアーティスト別などで仕分けされたディレクトリの下に同期されたファイル参照を用意しておけば，削除により，コピーやファイル別名が残ることがない．

3. 双方向変換

本章では，ファイルマネージャ実現のための双方向変換[6),8)] の定式化について述べる．

3.1 表 記 法

本論文では関数型言語 Haskell[2)] に倣った記法を用いる．

関　　数

関数定義は次のように書く．

$$double\ x = x + x$$

これは，引数の値を 2 倍する関数 $double$ の定義である．引数の括弧は省略され，スペースが用いられる．関数適用も同様にスペースを用いて表し，括弧は省略する．また，関数適用が最も結合順位が高い．よって $double\ 1{+}2$ は $double\ (1{+}2)$ ではなく，$(double\ 1){+}2$ を表す．関数合成は $f \circ g$ と書き，

$$(f \circ g)\ x = f\ (g\ x)$$

である．

関数はカリー化され，関数適用は左結合である．すなわち，$f\ a\ b$ は，$f\ (a\ b)$ でなく $(f\ a)\ b$ を表す．例として，2 つの値をとりその和を返す，カリー化された 2 引数関数 add は以下のように定義される．

$$add\ x\ y = x + y$$

ここで，引数の値を 1 増やした値を返す関数 inc は

$$inc = add\ 1$$

のように定義できる．このように add は引数を 1 つとり，「引数を 1 つとって値を返す関数」を返す関数であると考えられる．

変数 x が型 T であることを $x :: T$ のように書く．値の場合も同様である．また関数 f が A 型の引数をとり B 型の値を返すとき，f の型を

$$f :: A \to B$$

と書く．関数適用が左結合なのに対応し，$\to$ は右結合になる．たとえば，整数どうしの加算を行う add は

$$add :: \mathrm{Int} \to \mathrm{Int} \to \mathrm{Int}$$

という型を持つ．

組

複数の値を (a, b) や (x, y, z) のようにいくつかまとめたものを組と呼ぶ．ある組において，各々の要素の型が同じである必要はない．$a :: A, b :: B$ の組 (a, b) の型を (A, B) と書く．

リ　ス　ト

リストは，空リストを表す [] と，要素とリストをとり，要素を先頭に付け加えたリストを作る : という 2 つのデータ構成子からなる均質なデータ構造である．データ構成子 : は右結合である．簡便のため $1 : 2 : 3 : []$ を [1,2,3] と書く．$[1, (2, 3)]$ などは，1 と $(2, 3)$ の型が異なり均質でないので許されない．要素が A 型であるリストの型を $[A]$ と書く．2 つのリストは，演算子 $+\!\!+$ を用いて連結することができる．すなわち，

$$[1, 2, 3] +\!\!+ [4, 5] = [1, 2, 3, 4, 5]$$

である．

3.2 双方向変換

双方向変換[6),8)] は，2 つのデータの間での同期をとることを目的に考案された技術である．これらの双方向変換は，始点と終点の 2 つのデータ間について，順方向の変換を記述することが同時に逆方向への情報の更新方法も実現するように設計されている．また，一般の逆変換と違い，変換元と変換先の 2 つの情報を使って逆方向の反映を行うため，データの削除や挿入

など幅広い変換を記述することができるのが特徴である．

「梅林」はファイルマネージャであり，ファイルマネージャはディレクトリ木を編集するエディタであると考えられる．本節では，ファイルマネージャを実現するため，双方向変換を利用したエディタ[8]に対し，編集の伝播に関して整理を行う．

3.2.1 エディタのための双方向変換

ある構造を持つデータを直接編集する通常のエディタと違い，元のデータをビューと呼ばれるユーザにとって扱いやすい構造のデータに変換し，それに対する編集を行うエディタを考える．このようなエディタは，元となるデータ，ビュー，元となるデータとビューとの間の変換の 3 つを状態に持つ．本論文では，この元となるデータ構造のことをソースと呼ぶ．また，以下ではソースの集合に対応する型を $\mathcal{S}$，ビューの集合に対応する型を $\mathcal{V}$ で表し，$\mathcal{S}$ と $\mathcal{V}$ との間の変換の集合を $\mathcal{X}_{(\mathcal{S},\mathcal{V})}$ で表す．ファイルマネージャにおいて，$s :: \mathcal{S}$ はディレクトリ木に対応し，$v :: \mathcal{V}$ はアプリケーション上でユーザが実際に操作を行う木構造に対応する．

定義 1 （エディタの状態） ソース $s :: \mathcal{S}$，変換 $x :: \mathcal{X}_{(\mathcal{S},\mathcal{V})}$，ビュー $v :: \mathcal{V}$ の 3 つ組 (s, x, v) を，エディタの状態という． □

変換 $x :: \mathcal{X}_{(\mathcal{S},\mathcal{V})}$ について，$\mathcal{S}$ から $\mathcal{V}$ への順方向の解釈および，$\mathcal{V}$ から $\mathcal{S}$ への反映させる逆方向の解釈が存在する．

定義 2 （get と put） 任意の $x :: \mathcal{X}_{(\mathcal{S},\mathcal{V})}$ について，次の順方向の解釈 get および逆方向の解釈 put が定義される．

$$\mathsf{get}\ x :: \mathcal{S} \to \mathcal{V}$$
$$\mathsf{put}\ x :: (\mathcal{S}, \mathcal{V}) \to \mathcal{S}$$

□

エディタでは，ソースのデータをメモリに置いておくことができるため，ビューの状態をソースへ反映させる際に，変更前のソースを入手可能であるという仮定を自然に導入することができる．そのため，put x の第 1 引数は，ソースとビューの組になっている．

しかし，すべての変換がエディタの設計に有効であるわけではない．このような枠組みにおいては，たとえば以下のような変換を考えることも可能である．

$$\mathcal{S} = \mathcal{V} = \mathrm{Int}$$
$$\mathsf{get}\ x\ s = s + 1$$
$$\mathsf{put}\ x\ (s, v) = v + 1$$

この変換の動作は以下のようになる．

$$\mathsf{get}\ x\ 1 = 2$$
$$\mathsf{put}\ x\ (1, \mathsf{get}\ x\ 1) = 3$$
$$\mathsf{get}\ x\ (\mathsf{put}\ x\ (1, \mathsf{get}\ x\ 1)) = 4$$
$$\vdots$$

このような変換は，ソースとビューがともに変更されてない場合にも，get，put により状態が変化してしまい，なおかつそのような状態変化の伝播は停止することがない．このような挙動は，エディタとしては望ましくない．

エディタとして望ましい性質の議論を行うため，まず伝播が停止した状態を定義する．

定義 3 （定常なエディタの状態） あるエディタの状態 $(s, x, v) :: (\mathcal{S}, \mathcal{X}_{(\mathcal{S},\mathcal{V})}, \mathcal{V})$ が

$$\mathsf{get}\ x\ s = v \wedge \mathsf{put}\ x\ (s, v) = s$$

を満たすとき，(s, x, v) は定常であるという． □

これはすなわち，get，put いずれの操作によっても状態が変化しないということである．次に，状態が変更された場合に get または put により 1 回変更を反映するだけで，定常な状態に達成できる性質を定義する．

定義 4 （安定性） 変換 $x :: \mathcal{X}_{(\mathcal{S},\mathcal{V})}$ が以下の 2 つの性質を満たすとき，変換 x は安定であるという．

(1) ソースに変更が行われた場合

$$\forall s :: \mathcal{S},\ v = \mathsf{get}\ x\ s \Rightarrow \mathsf{get}\ x\ (\mathsf{put}\ x\ (s, v)) = v$$

(2) ビューに変更が行われた場合

$$\forall s :: \mathcal{S},\ \forall v :: \mathcal{V}, s' = \mathsf{put}\ x\ (s, v) \Rightarrow \mathsf{put}\ x\ (s', \mathsf{get}\ x\ s') = s'$$

□

また，ビューがソースの見せ方であるためには，ビュー上で何も変更が行われない場合にソースが変化してはならない．この性質は以下のように書くことができる．

定義 5 （GET-PUT 特性）

$\forall s :: \mathcal{S}, \mathsf{put}\ x\ (s, \mathsf{get}\ x\ s) = s$

このとき次の命題が成り立つ．

命題 1 GET-PUT 特性を満たす変換 $x :: \mathcal{X}_{(\mathcal{S},\mathcal{V})}$ は，安定である．

証明 まず，ソースが変更された場合を考える．つまり，

$$\forall s :: \mathcal{S},\ v = \mathsf{get}\ x\ s \Rightarrow \mathsf{get}\ x\ (\mathsf{put}\ x\ (s, v)) = v$$

を示す．

$$\begin{aligned}\mathsf{get}\ x\ (\mathsf{put}\ x\ (s,v)) &= \{v = \mathsf{get}\ x\ s\}\\ &\quad\ \mathsf{get}\ x\ (\mathsf{put}\ x\ (s, \mathsf{get}\ x\ s))\\ &= \{\text{GET-PUT 特性 }\}\\ &\quad\ \mathsf{get}\ x\ s\\ &= \{v = \mathsf{get}\ x\ s\}\\ &\quad\ v\end{aligned}$$

となるため，ソースが変更された場合に定常な状態を達成できる．

次にビューが変更された場合を考える．つまり，

$$\forall s :: \mathcal{S},\ \forall v :: \mathcal{V}, s' = \mathsf{put}\ x\ (s,v)\\ \Rightarrow \mathsf{put}\ x\ (s', \mathsf{get}\ x\ s') = s'$$

を示す．

GET-PUT 特性により，

$$\mathsf{put}\ x\ (s', \mathsf{get}\ x\ s') = s'$$

が成り立つ．よってビューが変更された場合にも定常な状態を達成できる． □

本論文では，GET-PUT 特性を満たす変換を双方向変換と呼ぶ．

3.2.2 原子編集操作とその反映

「梅林」で取り扱うディレクトリ木では，ソースの変更は OS のシステムコールやライブラリ関数によって行わなければならない．こういった枠組みでは，直接ソース全体を更新することはできず，特定のノードの追加，削除，内容の変更といった単位でソースの変更を行う必要がある．以下では，そのような操作主導の枠組みでの双方向変換について議論していく．

ソースの上では，ファイルの削除や新規ファイルの作成などの，いくつかの原子的な編集操作が定義されている．これを以下のように定式化する．

定義 6 （原子編集操作） ソースの上で定義されている $\mathcal{S} \to \mathcal{S}$ である関数の集合を $A(\mathcal{S})$ と書き，その元を原子編集操作という．また，この原子編集操作の列で行う編集の集合 $A^*(\mathcal{S})$ を

$$A^*(\mathcal{S}) = \\ \{a_k \circ \cdots \circ a_1 \,|\, k \geq 1, \forall j{:}1 \leq j \leq k.\, a_j \in A(\mathcal{S})\}$$

と定義する． □

ソースは原子編集操作を通してしか変更できないため，ビュー上での操作は必ずソース上で定義された原子編集操作の列に翻訳されなければならない．

定義 7 （ビュー上の許容される操作） すべての定常な $(s,x,v) :: (\mathcal{S}, \mathcal{X}_{(\mathcal{S},\mathcal{V})}, \mathcal{V})$ に対し，ビュー上の許容される編集 $A(\mathcal{V})$ を以下のように定める．

$$A(\mathcal{V}) \\ = \{a_v \,|\, \exists e_s \in A^*(\mathcal{S}),\ \mathsf{put}\ x\ (s, a_v\ v) = e_s\ s\}$$

□

ビュー上の操作がソース上の操作に翻訳されるためには，ビュー上での操作は $A(\mathcal{V})$ の元の列として表されなければならない．$A(\mathcal{V})$ はビュー上の操作を構成する最小単位ととらえることができるため，$A(\mathcal{S})$ と同様の表記法を用いる．

我々の目的は，$A(\mathcal{V})$ の元に対し，$\mathcal{X}_{(\mathcal{S},\mathcal{V})}$ の各要素について対応する $A^*(\mathcal{S})$ の元を求めることである．しかし，このような編集の翻訳は難しい．$A(\mathcal{V})$ や $A^*(\mathcal{S})$ の元が関数であり，取扱いが難しいためである．ここで，文献 10) において，リスト間の対応関係などの状態の整合性を保ちつつ編集の反映を行うために利用されたマーク付けの技術を利用することができる．ビュー上で編集操作が行われたデータに対してマークを付け，それを逆方向の変換によりソースに反映させることにより，編集操作をソースに伝播することを考える．

定義 8 （マーク付けによる編集操作の反映） ビュー上での編集操作の集合 $E \subset A(\mathcal{V})$ が与えられているとする．任意の $a_s \in A(\mathcal{S})$ および $a_v \in E$ に対し，

$$mark_s\ a_s :: \mathcal{S} \to \mathcal{S}'$$
$$mark_v\ a_v :: \mathcal{V} \to \mathcal{V}'$$

という関数およびこれらに対し，

$$reflect_v\ (mark_v\ a_v\ v) = a_v\ v$$
$$reflect_s\ (mark_s\ a_s\ s) = a_s\ s$$

である関数 $reflect_s$ と $reflect_v$ を考える．このとき，

$$reflect_s\ (\mathsf{put}_m\ x\ (s, mark_v\ a_v\ v)) = \mathsf{put}\ x\ (s, a_v\ v)$$

となる put_m および $mark_s$，$mark_v$，$reflect_s$，$reflect_v$ が定義できることをマーク付けによる編集操作が正しく反映できるという． □

この定義において，$\mathcal{S}'$ や $\mathcal{V}'$ は $\mathcal{S}$ や $\mathcal{V}$ に対応するマーク付けを含んだデータ構造を表している．通常 $A(\mathcal{V})$ を求めることは容易ではないため，ビュー上で行う編集の集合 E は $A(\mathcal{V})$ の部分集合となっている．ここで，$reflect_s$ はマーク付けされたソースを元にソース上での編集操作を実行する関数ととらえることができる．put_m および $mark_s$，$mark_v$，$reflect_s$，$reflect_v$ を定義することにより，put が定義でき，このマーク付けによる編集操作の反映性を考えなくてよくなるため，今後は put_m および $mark_s$，$mark_v$，$reflect_s$，$reflect_v$ に対して議論を行う．

3.3 双方向変換言語の設計

3.3.1 対象とするデータ構造

「梅林」の操作対象であるディレクトリ木を表現するためのデータ構造として，木（*Tree*）を以下のように定義する．

$$\mathbf{data}\ \mathit{Tree} = \mathit{Node}\ D\ [\mathit{Tree}]$$

「梅林」では子がリストである木を用いている．それは，子がリストである木のほうが get，put などの変換の際に扱いやすく，また木を表示する際の兄弟間に順序を自然に定めることができるためである．それに対し，ソースが表現すべきディレクトリ木では子が集合である．そこで，ラベルをファイル名やディレクトリ名に対応して生成し，ラベル上に全順序関係を定義することにより，ディレクトリ木を子がリストである木に変換している，しかし，ディレクトリ木は子が集合であるため，ソースの上での兄弟間の順序の違いはディレクトリ木の状態の違いを表さない．そのため，GET-PUT 特性などを考えるうえで，ソースの木の等価性を子の順序を無視した場合の等価性で定義する．

データ型 D は，ファイル名などのファイル情報や兄弟ノード間で一意に識別可能であるラベル（$Label$）などを含んでいる．ラベルを導入することにより，

- 変換の適用先を兄弟間の順序と独立させる，
- ノードと複製されたノードを区別する，

ことを実現する．リストのインデックスでは前者の，ファイル名では後者の実現が容易ではない．D に関する詳細は,「梅林」における D の具体的な定義として 4 章で議論することとする．

木上の任意のノードを一意に差し示すために，以下のようにパス（$Path$）をラベルのリストとして定義する．

type $Path = [Label]$

空リスト [] は木の根を表し，$[l_1, l_2, \ldots, l_n]$ は根のラベル l_1 を持つ子ノードの，ラベル l_2 を持つ子ノードの，... ラベル l_n を持つ子ノードを表す．

3.3.2 X 言 語

ここでは，これまで議論された安定性を満たし，なおかつ許容される具体的な変換と編集の組を定義する．我々は，既存の双方向変換言語[6),8)] に倣った変換記述言語を定める．この言語の上では，ある操作に関していくつかの安定で許容される基本変換が定義されており，基本変換を様々な結合子により組み合わせることにより，安定で編集反映可能な枠組みを構成することができる．既存の研究[8)] と同様にこの変換記述言語を X 言語と呼ぶ．X 言語の文法は**図 3** のようになっている．X 言語により，ソースである木構造とビューの木構造を同期することができる．ただしソースとビューは同じ型である必要がある．

この言語のいくつかの基本変換および結合子の直観的な動作の概要を**図 4** に示す．Id は恒等変換である．Sort は表示のためにソートを行う．CMap は適用されるノードのすべての子に x を適用し，If は条件分岐を行う．Insert は与えられた木を子として挿入する．Hide はノードの隠蔽を行う．HSwap は兄弟間の順序を入れ替える．DupTo は複製変換である．複製変換はある部分木を指定されたノード以下に複製し，編集に関し同期された 2 つの部分木を作る．複製変換により同期された部分木に対し，一方に加えた編集は必ずもう一方に反映されるため，編集に関して対称に振る舞う．複製操作は 2 章の Duplicate に対応する．Apply は指定されたノードに適応される具体的な変換を指定する．これらを結合子 ; により連接させていくことにより様々な表示の操作を行うことができるようになる．

$$
\begin{aligned}
X &:= \mathsf{Apply}\ Path\ X' \\
&\mid X; X \\
\\
X' &:= \mathsf{Id} \\
&\mid \mathsf{Sort}\ (D \to D \to Int) \\
&\mid \mathsf{CMap}\ X' \\
&\mid \mathsf{If}\ (Tree \to Bool)\ X'\ X' \\
&\mid \mathsf{Insert}\ (Tree) \\
&\mid \mathsf{Hide}\ Label \\
&\mid \mathsf{HSwap}\ Label\ Label \\
&\mid \mathsf{DupTo}\ Path
\end{aligned}
$$

図 3 X 言語
Fig. 3 Formal definition of X Language.

たとえば，

```
Apply p1 (
  CMap (
    If pred? (DupTo p2) Id));
Apply p2 (Sort cmp)
```

という変換はパス p_1 の表すノードの $pred?$ を満たすすべての子の同期された複製をパス p_2 の表すノード以下に追加し，比較関数 cmp により並べ替える．

図 3 で使用される $Path$ は絶対パスを表す．すなわち，入力された木の根からのパスであり，Apply されたノードからパスではない．これは，上の例のように CMap の引数として DupTo を使用できるようにするためである．パスに親を表す表現を付け加えることにより，相対的なパスを用いた言語も定義できるが，扱いが複雑になるためここでは絶対パスを用いて言語の定義を行った．また，そのため Apply $p\ x$ 中の x に Apply を含むことを避ける必要があり，文献 8) と異なり，X と X' と 2 階層に分けた．

次にソースおよびビュー上の原子編集操作を定義する．

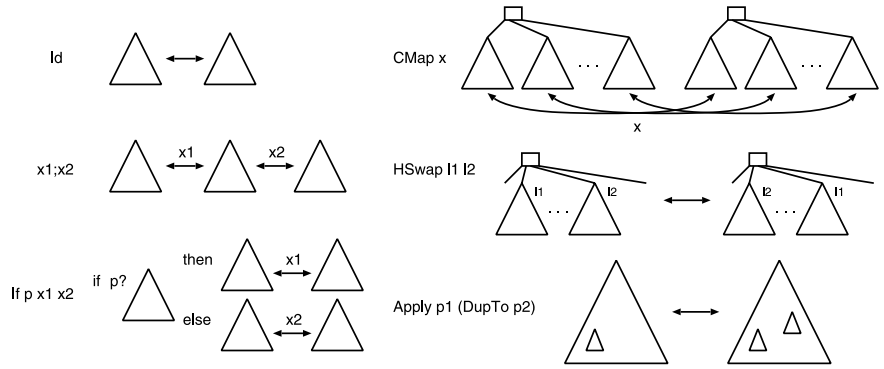


図 4 X 言語の基本変換および結合子の直観的な動作
Fig. 4 Intuitive behavior of basic transformations and combinators in our X Language.

```
get (Apply p x) s = getLocal x p s
get (x1; x2) s = get x2 (get x1 s)

getLocal Id p s = s
getLocal (Sort cmp) p s =
    let Node n cs = getTree p s
    in repTree p s (Node n (sortBy cmp cs))
getLocal (CMap x) p (Node n cs) =
    let Node n cs = getTree p s
    in getChildren x p cs s
getLocal (If pred x1 x2) p s = if pred (getTree p t) then getLocal x1 p s else getLocal x2 p s
getLocal (Insert t) p s = insTree (newLabel p s) p s t
getLocal (Hide l) p s = repTree p s (remChild l (getTree p s))
getLocal (HSwap l1 l2) p s = repTree p s (hswap l1 l2 (getTree p s))
getLocal (DupTo p2) p1 s = insTree (newLabel p2 s) p2 s (getTree p1 s)

getChildren x p (c : cs) t = getChildren cs (getLocal x (p ++[getLabel c]) t)
getChildren x p [] t = t
```

図 5 X 言語の get 時の意味
Fig. 5 Semantics of our X Language in get phase.

- *add p t*：木 *t* をパス *p* のノードに子として追加
- *del p*：パス *p* のノードを削除
- *mod p n*：パス *p* のノードの内容を *n* :: *D* に変更

このとき，X 言語の get および put_m に解釈は**図 5**，**図 6** のように与えられる．ここで $mark_s$，$mark_v$ は，

- *add*：追加されるノード
- *mod*，*del*：*p* で指定されたノード

にマーク付けを行い，$reflect_s$，$reflect_v$ はそれに対応した形で自然に定義される．

図 5，図 6 で使用される補助関数について説明を行う．*getTree p t*，*repTree p t t′*，*insTree l p t t′* はそれぞれ，木 *t* のパス *p* で示される部分木を返す関数，木 *t* のパス *p* 示される部分木を木 *t′* で置換する関数，木 *t* のパス *p* で示される場所に子として木 *t′* のラベルを *l* に変更したものを追加する関数である．*newLabel p t* は，その位置の部分木の子のラベルを調べユニークなラベルを生成する関数であり，同じ部分木に対しては，同じラベルが生成される．このような *newLabel* の性質を利用し，*getRecent* や *remRecent* は，*newLabel* によってラベル付けされたノードの取得や削除を行う．*sortBy cmp* は比較関数 *cmp* に応じてソートを行う．比較関数 *cmpLabelWise* はラベルに定義される全順序関係に従い比較を行う．また，*isAdded* は編集 *add* によってマーク付けされたかどうかを検査する．

get は順方向の変換であるので，上で述べたとおりの変換を木に施す．*getLocal* は，図 3 中の X' を処理する．絶対パスを処理するため，どのノードに変換が適用されるかの情報を第 2 引数にとっている．

put_m はビューの編集を反映を行い，GET-PUT を満たすように定められる関数である．編集 *add* によっ

```
put_m (Apply p x) (s,v) = putLocal x p (s,v)
put_m (x1;x2) (s,v) = put_m x1 (s, put_m x2 (get x1 s,v))

putLocal Id p (s,v) = v
putLocal (Sort cmp) p (s,v) = getLocal (Sort cmpLabelwise) p v
putLocal (CMap x) p (s,v) =
    let Node m cs = getTree p s
    in putChildren x p cs (s,v)
putLocal (If pred x1 x2) p (s,v) =
    if pred (getTree p s) then putLocal x1 p (s,v) else putLocal x2 p (s,v)
putLocal (Insert t) p (s,v) = repTree p v (remRecent (getTree p v))
putLocal (Hide l) p (s,v) = repTree p v (addChild (getTree p v) (getTree (p ++[l]) s))
putLocal (HSwap l1 l2) p (s,v) = getLocal (HSwap l1 l2) p v
putLocal (DupTo p2) p1 (s,v) =
    let d = getRecent (getTree p2 v)
        v' = repTree p2 v (remRecent (getTree p2 v))
        d' = getTree p1 v'
    in repTree p1 v' (merge d' d)

putChildren x p (c : cs) (s,v) =
    let p' = p ++[getLabel c]
        v' = putChildren x p cs (getLocal x p' s,v)
    in putLocal x p' (s,v')
putChildren x p [] (s,v) = v

merge (Node m cs) (Node n ds) =
    Node (takeChanged m n) (mergeChildren (sortBy cmpLabelwise cs, sortBy cmpLabelwise ds))

mergeChildren [] [] = []
mergeChildren (x : xs) [] = if isAdded x then x : mergeChildren xs [] else ⊥
mergeChildren [] (y : ys) = if isAdded y then y : mergeChildren [] ys else ⊥
mergeChildren (x : xs) (y : ys) =
    if isAdded x ∧ isAdded y then
        if x == y then x : mergeChildren xs ys
        else x : y : mergeChildren xs ys
    else if isAdded x then x : mergeChildren xs (y : ys)
    else if isAdded y then y : mergeChildren (x : xs) ys
    else merge x y : mergeChildren xs ys
```

図 6 X 言語の put 時の意味
Fig. 6 Semantics of our X Language in put phase.

てノードが追加された場合，木の構造がソースとビューで変化してしまうので，対応付けの際に考慮する必要がある．*merge* t t' は，子供のリストがマークを含まない場合には，定常性の仮定から $t = t'$ となるため t を返す．マーク含む場合には t と t' はマークと *add* なノードを除いて同じであるので，これらのマーク情報を再帰的にマージしていった木を返す．*mergeChildren* が *add* されたノードを，*merge* が *takeChanged* によりそれ以外のマークをマージする．

以下，X 言語により定義される変換が GET-PUT 特性を満たし，*add*，*mod*，*del* が許容されることを簡単に示す．まず，GET-PUT 特性を考える．Hide は get 時に削除した子を put_m 時にソースから取得し，Insert は get 時に挿入したノードを *newLabel* の性質によって put_m 時にビューから削除することで，GET-PUT 特性を満たしていることが直観的に確認できる．ソース木の等価性は子の順序を無視して定義されるため，Sort や HSwap が GET-PUT 特性を満たすのは明らかである．CMap は，get 時にソースの子をラベル順に変換を適用していったのを，put_m 時にソースの子のラベル順とは逆順に put_m によって逆方向に変換するため，子供に使用される変換が GET-PUT を満たすなら，CMap も GET-PUT 特性を満たすことがいえる．DupTo は，get 後の状態では d' と d が等しいことを考えると，put_m 時に get 時に挿入された木を取り除くため GET-PUT 特性を満たす．変換 $x_1; x_2$ については，$\mathsf{put}_m(x_1; x_2)$ の途中の木で子の順序が変わりうるが，これは put_m の定義からソースでの子の

順序に影響するだけなので，$x_1; x_2$ も GET-PUT 特性を満たす．次に，許容性を考える．ソース上の *add*，*mod*，*del* の操作の列は，与えられた木を任意の木に変換することができる．そのため，ビュー上での *add*，*mod*，*del* が，許容されるのは明らかである．

4. 実装詳細

本章では「梅林」の実装の詳細を示す．なお,「梅林」の実装は，関数型言語 Haskell によって行った．

4.1 概観

「梅林」では扱うべきデータがディレクトリ木であり，その状態を直接操作することができない．そのため，システムコールを通して，ディレクトリ木と同じ構成の木をソースとして作成する．また変換の記述には X 言語を用いた．X 言語は自然な形で Haskell プログラムとして記述することができる．X 言語を用いるため，ビューとソースは同じ型の木である．

「梅林」の構成を**図 7** に示す．システムコールによりソースが構築され，そこから get により得られたビューが GUI コンポーネントに表示される．ユーザがビュー上で編集を加えると，put によりソースに対する編集に翻訳され，システムコールを用いてディレクトリ木に反映される．その後，新たにソースが再構築され get により変換し表示する．“見せ方” が変更された場合は，その新たな変換を用いて，get によりビューを再構築する．

例として，同期関係にある名前 b のノードに対し，c に改名する編集を行った場合の挙動を**図 8** に示す．ここで，図中の b の下に添字されている`{Mod c}`はマークを表している．

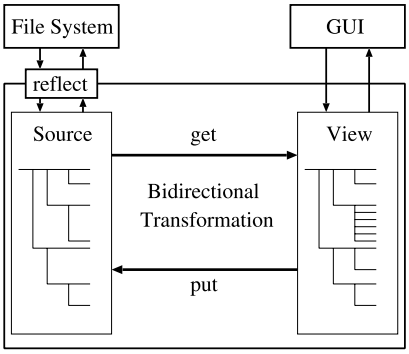


図 7 「梅林」の構成
Fig. 7 Framework of bi-link.

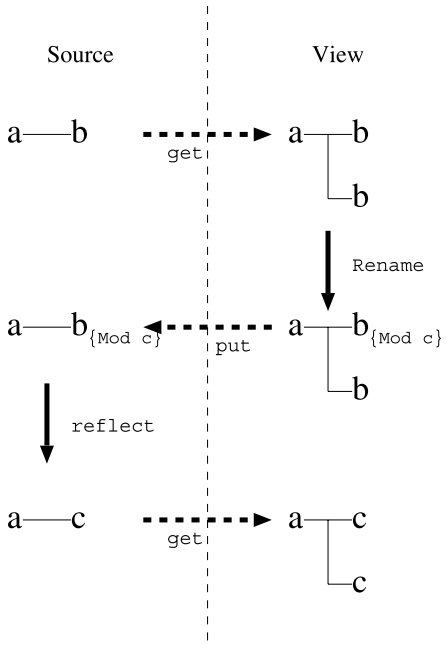


図 8 改名編集の例
Fig. 8 Rename a node on the view.

4.2 データ構造の詳細

木

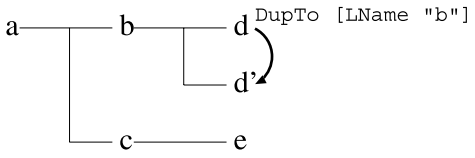
「梅林」で用いたソースおよびビューの木は，以下のように定義される．

```
type Source = Tree
type View   = Tree
data Tree = Tree D [Tree]
type D = ( ViewState, Label, Mark )
data ViewState = Concrete FileState
               | Abstract String
data FileState =
     {
         parentpath   :: String,
         filename     :: String,
         filetype     :: FileType,
         size         :: Integer,
         permissions  :: Permissions,
         modifiedtime :: ClockTime
     }
data FileType = File | Dir
```

Mark はマーク情報を表す．簡便のため，マーク情報がノードに含まれている．これにより，マーク付きの木とそうでない木を同一のデータ構造で扱うことができる．FileState はファイルまたはディレクトリの様々な状態を表す．parentpath は親へのパス情報を表し，filename は名前である．size，permissions，modifiedtime はサイズ，パーミッション，更新日時を表す．

表 2 各マークに対応した処理
Table 2 Operations corresponding to marks.

Add	そのファイル/ディレクトリを新規作成する
Del	そのファイル/ディレクトリを削除する
Mod a	そのファイル/ディレクトリの名前を a に変更する
Open	そのディレクトリ以下のソースをディレクトリ木から再構築する



path d : [LName "b", LName "d"]

path d' : [LName "b", LAbstract 1]

図 9 パスの例
Fig. 9 Examples of path.

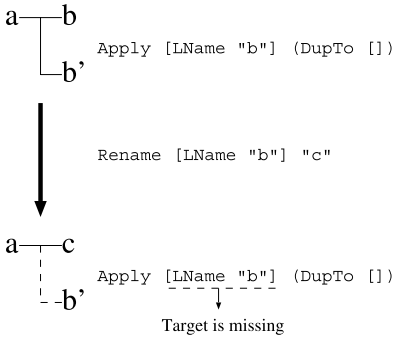


図 10 改名により変換が適用されなくなる例
Fig. 10 Invalid transformation caused by a rename operation.

ラ ベ ル

「梅林」では各ノードに対して以下のようにラベルを定義している．

```
data Label = LName String
           | LAbstract Int
           | LOther
```

LName は実際にディレクトリ木上に存在するノードのラベルであり，ファイル名と同一の文字列が格納される．LAbstract は抽象ノードを表す．抽象ノードは Insert または DupTo により挿入されるノードである．その際に，挿入される先の兄弟間で重複しない番号が与えらえる．LOther は，編集により挿入される予定のノードに付けられる．挿入されるノードが他のノードと同じラベルを持つこと防ぐためである．

ディレクトリ木において，あるノードの子の名前はすべて異なるため，LName は兄弟間ですべて異なる．また，LOther は決して参照されることがないため，その存在は，あるパスがノードを一意に指し示すことに関して影響を与えない．よって，ここで定義されたラベルを用いて一意識別可能なパスを定義することができる．

「梅林」における具体的なパスの例を**図 9** に示す．図 9 中において d' は d を複製してできたノードである．

4.3 *reflect* の実装

「梅林」では編集操作を行う際，ビュー木の各ノードに編集操作反映のためのマークを付け，それを put によりソース木まで伝播させる．マークの実装を以下に示す．

```
type Mark = Maybe NotableMark

data NotableMark = Add
                 | Del
                 | Mod String
                 | Open
```

表 2 はマークされたノードに対し，ディレクトリ木上で行われる処理である．Open は，ディレクトリ木からソースを構築する際に使用される．

4.4 双方向変換の実装上の問題とその解決法

対象ノードを指定し双方向変換を適用する場合，ノードの指定に起因する問題がある．その問題と，解決法を示す．

4.4.1 ファイルの改名時に生じる問題

あるファイルの改名を行うと，そのラベルが改名後の名前に変換され，今までのパスが宙吊りになる．結果，改名されたノードとその子孫ノードに対する変換が無効になる．

改名により複製変換が適用されなくなる例を**図 10** に示す．図 10 では，ノード b に対する変換 DupTo が存在し，その同期されたノードの一方を b' とするとき，ノード b を c に改名する編集を適用すると，b のラベルが LName "c"に変更されるため，複製元のパスが改名されたノードを指すことができず，変換が適用されなくなる．

この問題は，put した後 get することにより，改名の起こるノードを列挙することができるため，その後，

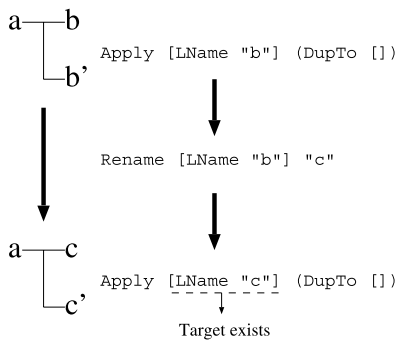


図 11 改名を正しく反映する例
Fig. 11 Successful transformation after a rename operation.

put 時にそれらのノードを通過するパスをすべて書き換えることにより解決できる．これにより get 時には改名されたノードのラベルと，ラベル変更済みの変換の対応がとれ，問題なく変換を適用することが可能となる．この方法により**図 11** のように正しく変換を適用できるようになる．

4.4.2 再生成時に生じる問題

ある名前のノードが削除された後に，削除されたものと同じ名前のノードが同じ位置に作成されると，パスが同一になってしまう．このため，削除前にそのノードに対して変換が適用されていた場合，その変換が再度適用されてしまう．しかし，これらのノードは別のものであるため，この挙動は望ましくない．

この問題は，get 時に適用先ノードの存在しない変換を Id に置換することにより解決できる．ノードが削除されたときにそのノードに関する変換は存在しなくなるため，再生成されたノードに対して変換が適用されることはなくなる．

5. 議　　論

前章まで，双方向変換を利用したファイルマネージャ「梅林」の実現方法に関して説明してきた．本章では，関連研究との比較を通じて，本研究の貢献および残された課題について議論を行う．

関連システム

ここでは，ファイルマネージャ「梅林」と関連の深いシステムについて述べる．

Proxima[13] は汎用エディタであり，「梅林」と同様にビューを持つ．Proxima では両方向の変換をユーザが定義しなければならなかったが，「梅林」は，変換を双方向変換言語で記述することにより，この問題を生じない．

また，BTRON[3] には実身/仮身というファイルシステムがある．仮身はファイル参照に対応する概念であり，実身は必ず仮身を通して参照され，その中に任意の仮身を含むことができる．つまり，ファイルシステムそのものがグラフ構造になっている．「梅林」では，循環を含むようなグラフ構造を現在においては扱えないが，非循環有向グラフ構造を，双方向変換を通して扱うことにより，構造化して扱うことができる．

シンボリックリンクを利用した場合，自分の親へリンクを張ることにより，循環構造を作成できる．DupTo を利用した場合は循環構造は不動点として表現されることになるが，双方向変換において，このような不動点の扱いはまだよく分かっていない．しかし，我々は循環構造がファイル管理において有用であると考えないため，これは問題とはならない．

シンボリックリンクの際に，逆ポインタを持たせることでも，別名の非対称性を解決することができる．そこで，双方向変換を利用したシステムと逆ポインタを持つシステムとの比較を行う．主な相違点を，**表 3** に示す．

逆ポインタを持つシステムでは，個々のファイルやディレクトリといったオブジェクト単位で同期関係の管理を行う．それに対し，双方向変換を利用したシステムは，同期関係を変換がすべて保持し管理を行う．これにより，双方向変換を利用したシステムでは同一のディレクトリ木に対し，異なった変換を適用することにより複数のビューを作成することが可能になる．ユーザが個々の目的に応じたビューを作成できることは，ファイルマネージャとして有用であると考えられる．たとえば，双方向変換を利用したファイルマネージャでは，好みの演奏家を集めたものや好みの年代を集めたものなど，目的別のビューを作成することができ，音楽ファイルを複数人で共有している場合にも，共有している人それぞれに対し固有のビューを作成することができる．

また，逆ポインタを用いたシステムでは，新たな機能や変換を任意の言語で記述できるものの，機能や変換の追加の際に同期関係を適切に管理する必要がある．それに対し，双方向変換を利用したシステムでは，新たな機能や変換を，双方向変換言語により記述することにより，同期関係を保ったまま追加することができる．

変 換 言 語

今回は，ファイルマネージャの作成にあたって，Hu らの双方向変換言語[8] を利用した．これは，

- 木構造を扱うことができる，
- 複製変換により同期機能を実現できる，

表 3 双方向変換を利用したシステムと逆ポインタを利用したシステムとの比較
Table 3 Difference between bidirectional transformation based systems and reverse pointer based systems.

	双方向変換を利用したシステム	逆ポインタを利用したシステム
循環構造	現在では扱えない	扱える
同期関係の保持	変換	ファイルやディレクトリなど
拡張	双方向変換言語で書けば，同期関係を保てる	記述言語は問わないが，同期関係は拡張する側で管理

- 変換の安定性，
- 3 章で示したようにマーク付け[10] により編集操作そのものを伝播できる，

という性質が，ディレクトリ木の見せ方を統一的に抽象化できるファイルマネージャの作成に適していたためである．

しかし，ほかにも多くの同期機能を持つ変換言語が存在する．Meertens は，値の間に制約を付加できるユーザ対話環境の実現を目的として，編集後の 2 つの値間の制約の維持を定式化した[9]．論理型言語を用いて，XML のような型付きデータを変換する研究もある[4]．この言語では，異なる DTD 間の変換を自然に記述することができる．Ohori らもビューを扱う同期機能を持つ言語を作成している[11]．彼らの言語では，オブジェクト指向データベースのビューを定義するうえで，静的にプログラムを多相型付けすることができる．

編集に関する整理

ファイルマネージャにおいて，ファイルの新規作成や名前変更など編集操作は重要な役割を果たす．DupTo のような複製変換での矛盾を防ぐため，文献 8) では限られた編集操作のみを考えそのうえで編集の伝播を考えた．

我々も限られた編集操作に対する伝播を考えてきたが，それは，ビューに対する編集操作をソースに対する編集操作へ翻訳するためである．このような考えかたは決して新しいものではなく，データベースの分野などで古くから議論されている[1),5),7)]．3 章において，双方向変換において翻訳に関して整理を行った．双方向変換を用いたエディタの実現に対しこのようなアイデアを定式化したのは我々の貢献である．

我々は，GUI で対話的にビューを操作することを対象としたため，許される編集操作を自然に限定することができた．それにより，矛盾するような編集操作が同時に起こることを防ぎ，編集操作の伝播の議論を単純化することができた．しかし，複数のファイルマネージャを同時に使用する場合にはこのような状況が仮定できず，互いに矛盾する編集操作が同時に発生しうる．このような場合には，様々な分散ファイルシステムやバージョン管理システム[12),14)] のように，編集のビューへ伝播の際に衝突を解消するポリシを定める必要がある．

表 現 力

双方向変換による編集伝播について，ビューに依存性がある場合の編集伝播に関しては議論されているが，ソースに依存性がある場合の編集伝播についての議論はあまり行われていない．たとえば，

$$\mathsf{get\ Eq}\ (s_1, s_2) = s_1$$
$$\mathsf{put\ Eq}\ (s, v) = (v, v)$$

のような変換は，ある編集操作をソースの 2 つの場所に伝播させることができる．もしこのような変換を矛盾なく体系に組み入れることができれば，ディレクトリのミラーリングといったファイル管理に有用な機能を実現することができる．この場合は，s_2 が s_1 のミラーになっている．この変換を許すと GET-PUT を満たさなくなるため，定式化が多少複雑になる．現在，我々はこの変換を導入することを検討中である．

現時点では，Fold[8] などの再帰的な結合子は簡略化のために考慮していない．もしこれを導入することができれば，ディレクトリのファイルサイズの総計の計算や，検索結果を 1 つのディレクトリに集めるといった処理が行えるようになる．文献 6)，8) での議論がすでにあるため，このような演算子を体系に組み入れるのは比較的容易であると考えられる．

現在「梅林」ではコピーや移動を許していない．それは，ビュー木にはファイルに関する注釈を表す抽象ノードが存在しており，注釈を表すノードがコピーされると注釈の文字列をファイル名とする空のファイルがソース木に作成されてしまうためである．この問題は，

- ディレクトリの子はディレクトリかファイルか注釈である，
- 注釈は子を持たない，
- ファイルは子を持たない，
- ソース木にはディレクトリとファイルしか存在しない，

といったデータのスキーマを利用することで無意味な反映を防ぐことができ，コピーや移動なども自然な形

で体系に組み込めることが期待できる．

効 率 性

ファイルマネージャ「梅林」では，同期されたファイル参照やソートなどの機能を双方向変換を用いて実現している．そのため，双方向変換に関する編集を重ねることにより，保持している変換のサイズが肥大化してしまう．「梅林」では，変換はプログラムとして記述されるため，プログラム最適化技術を応用することでこれを解決することが期待できる．

一 貫 性

ファイルマネージャでは，あるノードが削除された場合にそのノードに適用されていた変換が他のノードに適用されてしまうのは望ましくない．Hu ら[8] や Mu ら[10] では，子をリストとして扱い，あるノードの子を指し示すのにインデックスを使用していた．この場合にノードの追加または削除を行うと，変換の適用先が変化してしまう．変換をノードそのものに適用するのを実現するため，本研究では 3 章で示したように，子にラベルを導入し，子にインデックスではない一意な識別子を与えることにより，この問題を解決した．

「梅林」では，ソース木に含まれるような具体的なノードに関するパスの一貫性の問題を解決した．しかし，DupTo や Insert で挿入される抽象ノードに関する問題はまだ残っている．抽象ノードに関して，今回は挿入される側のノードでの抽象ノードの数を利用してラベル付けを行った．この方法では編集などでパスが無効になった場合に，挿入される抽象ノードの数が変化しラベルが以前のものと変化してしまう．しかし編集操作を通して一貫したラベルを抽象ノードに付けるのは難しい．1 つの方法として，エディタがこのような挿入される抽象ノードのラベルを管理し，ラベル情報を変換の中に埋め込むことが考えられるが，この方法では，連接などにおいてラベルが衝突しないことを確認する必要がある．このような一貫したラベルを付ける方法は今後の研究課題である．

6. ま と め

ファイルマネージャの実現のために，双方向変換の枠組みに編集操作の翻訳という概念を持ち込み，それに対し整理を行った．そして，それを利用しファイルマネージャ「梅林」の作成を行った．「梅林」ではすべてのファイル参照を対称的に扱うことができ，従来のファイル別名機能のような複数箇所からのファイル参照は，双方向変換により，同期された複製として表現できる．また，ソート，お気に入りリストなどのディレクトリ木の見せ方に関する機能も双方向変換を通して，統一的に扱うことができる．ユーザはディレクトリ木の変更および見せ方の変更を GUI を通して対話的に実行できる．

参 考 文 献

1) Bancilhon, F. and Spyratos, N.: Update semantics of relational views, *ACM Trans. Database Syst.*, Vol.6, No.4, pp.557–575 (1981).
2) Bird, R.S.: *Introduction to Functional Programming Using Haskell*, Prentice Hall (1998).
3) BTRON. http://www.assoc.tron.org/
4) Coelho, J. and Florido, M.: Type-Based XML Processing in Logic Programming, *PADL '03: Proc. 5th International Symposium on Practical Aspects of Declarative Languages*, pp.273–285, Springer-Verlag (2003).
5) Dayal, U. and Bernstein, P.A.: On the correct translation of update operations on relational views, *ACM Trans. Database Syst.*, Vol.7, No.3, pp.381–416 (1982).
6) Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C. and Schmitt, A.: Combinators for bi-directional tree transformations: A linguistic approach to the view update problem, *POPL '05: Proc. 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Long Beach, California, USA, pp.233–246, ACM Press (2005).
7) Gottlob, G., Paolini, P. and Zicari, R.: Properties and update semantics of consistent views, *ACM Trans. Database Syst.*, Vol.13, No.4, pp.486–524 (1988).
8) Hu, Z., Mu, S.-C. and Takeichi, M.: A programmable editor for developing structured documents based on bidirectional transformations, *PEPM '04: Proc. 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, Verona, Italy, pp.178–189, ACM Press (2004).
9) Meertens, L.: Designing Constraint Maintainers for User Interaction. http://www.kestrel.edu/home/people/meertens/
10) Mu, S.-C., Hu, Z. and Takeichi, M.: An Algebraic Approach to Bidirectional Updating, *APLAS '04: 2nd ASIAN Symposium on Programming Languages and Systems*, pp.2–18, Springer Verlag (2004).
11) Ohori, A. and Tajima, K.: A polymorphic calculus for views and object sharing (extended abstract), *PODS '94: Proc. 13th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, Minneapo-

lis, Minnesota, United States, pp.255–266, ACM Press (1994).
12) Saito, Y. and Shapiro, M.: Optimistic replication, *ACM Computing Surveys*, Vol.37, No.1, pp.42–81 (2005).
13) Schrage, M.M.: Proxima — A presentation-oriented editor for structured documents, Ph.D. Thesis, Utrecht University, The Netherlands (2004).
14) subversion. http://subversion.tigris.org/

(平成 17 年 7 月 5 日受付)
(平成 17 年 11 月 28 日採録)

松田 一孝（学生会員）
1982 年生．2004 年東京大学工学部計数工学科卒業．同年同大学大学院情報理工学系研究科入学．木構造処理，アルゴリズムの導出に興味を持つ．

大川 徳之
1982 年生．2005 年東京大学工学部計数工学科卒業．同年同大学大学院情報理工学系研究科入学．アルゴリズムの導出，並列分散計算に興味を持つ．

野村 芳明
1982 年生．2005 年東京大学工学部計数工学科卒業．同年同大学大学院情報理工学系研究科入学．アルゴリズムの導出，並列分散計算に興味を持つ．

森田 直幸
1980 年生．2005 年東京大学工学部計数工学科卒業．同年同大学大学院情報理工学系研究科入学．アルゴリズムの導出，木構造処理に興味を持つ．

筧 一彦（正会員）
1997 年早稲田大学理工学部情報学科卒業．1999 年同大学大学院理工学研究科情報科学専攻修士課程修了，2002 年同博士課程修了．博士（情報科学）．1999 年から 2002 年まで日本学術振興会特別研究員．2002 年より東京大学大学院情報理工学系研究科助手．2005 年より同研究科講師．関数型言語やプログラム変換，アルゴリズムの導出に興味を持つ．日本ソフトウェア科学会，ACM 各会員．

胡 振江（正会員）
1966 年生．1988 年中国上海交通大学計算機科学系を卒業．1996 年東京大学大学院工学系研究科情報工学専攻博士課程修了．同年日本学術振興会特別研究員を経て，1997 年東京大学大学院工学系研究科情報工学専攻助手，同年 10 月同専攻講師，2000 年同専攻助教授．2001 年より東京大学大学院情報理工学系研究科助教授．博士（工学）．関数プログラミング，プログラム変換，アルゴリズムの導出，並列プログラミング等に興味を持つ．日本ソフトウエア科学会，ACM 各会員．

武市 正人（正会員）
1948 年生．1972 年東京大学工学部助手，講師，電気通信大学講師，助教授，東京大学工学部助教授を経て 1993 年東京大学大学院工学系研究科教授（情報処理工学講座），2001 年より同大学大学院情報理工学系研究科教授，現在に至る．工学博士．プログラミング言語，関数プログラミング，言語処理システムの研究・教育に従事．日本ソフトウェア科学会，日本応用数理学会，ACM 各会員．