タイトル	UNIXファイルシステムの形式化における不変性の拡張と一般化
著者	佐藤，晴彦；SATO，Haruhiko
引用	北海学園大学工学部研究報告(53)：1-13
発行日	2026-01-09

UNIXファイルシステムの形式化における不変性の拡張と一般化

佐　藤　晴　彦

Extension and generalization of invariance properties in the formalization of UNIX file−system

Haruhiko SATO

概　要

本論文では，WenzelによるUNIXファイルシステムの形式化において成り立つ性質についての拡張と一般化を行った結果について報告する．新たに示した性質に基づき，形式化における重要な結論の一つである特定の状況化での削除不能性の証明において，その一部をより簡潔に記述できることを示す．

1　はじめに

形式検証はシステムの性質を数学的な手法を用いて厳密に調べる手法であり，OSのカーネルやコンパイラのような高信頼性が求められるシステムの検証において特に重要である[1)2)]．システムを数学的に表現しその性質を証明する形式化の工程全体の正しさを保証するため，定理証明支援系が活用されている．

Wenzelは，定理証明支援系Isabelle/HOLを用いてUNIXファイルシステムの形式化を行った[3)]．この形式化では，応用例の一つとして自身で作成したディレクトリが削除不能となる特殊な状態に到達し得ることを証明している．この状態は次の一連の操作によって生じる．

1．ユーザAが，誰でも書き込み可能なディレクトリD1を作成する
2．ユーザBが，所有者のみ書き込み可能なディレクトリD2をD1の直下に作成する
3．ユーザBが，何らかのファイルFをD2の直下に作成する

ここで，ユーザAは管理者権限を持たないものとする．ディレクトリを削除するにはそれが

北海学園大学准教授　工学部電子情報工学科
Assistant Professor, Department of Electronics and Information Engineering, Faculty of Engineering, Hokkai−Gakuen University

空である必要があるため，AがD1を削除するにはBが所有するD2の削除が必要であり，またD2を削除するにはD2を空にする必要がある．しかしながら，D2の要素であるFをAが削除することはできないため，AはD2を削除することもできない．よってその親であるD1を削除することもできない．

一般的に，形式的な証明は通常の自然言語による証明よりも長くなる傾向にある．前述の削除不能性に関する議論はそれほど複雑なものではないが，形式証明における上記の議論に対応する命題の証明は50ステップを超える．このような通常の証明と形式証明とのギャップが生じる主な原因は，形式証明に要求される厳密さから生じる本質的な記述量の増加である．厳密さを損なわずに可読性を向上させるためには，通常の証明の簡潔さを支える暗黙的なステップを形式証明においても表現できるよう定義や補題を拡充し，その上で証明の構造を整理しギャップを小さくする試みが重要であると考えられる．

本論文では，前述の削除不能性に関する証明を再構成することを目的とした，Wenzelによる形式化（以降，原形式化）の拡張について報告する．[*1]具体的には，木構造の参照・操作に関する基本的な性質やシステムコール操作に対する木構造の不変性を表す性質を証明した上で，それらを用いて削除不能性の証明の一部をより簡潔にするための，直感的に理解しやすい補題を構成する．

2 ファイルシステムの形式化

本節では，原形式化における基本概念および証明された性質について，本研究で行う拡張に関連のある部分について説明する．以降では，定理証明支援系Isabelleの用語や記法[4)5)]を用いる．

2.1 入れ子環境による木構造表現

ファイルシステムを表す木構造の表現には，Isabelle/HOLのNested_Environmentライブラリが提供するenv型を用いる．env型は型変数$'a$，$'b$，$'c$および2種類の構築子Val，Envを持つものとして次のように定義される．

```
datatype ('a, 'b, 'c) env = Val 'a | Env 'b ('c ⇒ ('a, 'b, 'c) env option)
```

型$'a$と$'b$は葉ノードと内部ノードが保持する値の型を，型$'c$は内部ノードに子を対応させる枝の型を表す．Val aは$'a$型の値aを持つ葉ノードを表し，Env b esは$'b$型の値bと，枝を表す$'c$型の値にその先の部分木を対応させる部分関数esからなる内部ノードを表す．Val型の値

[*1] 本文中で省略した証明の細部を含む，本研究で作成した形式化全体は https://github.com/h-sato-hgu/isabelle-unixfsで公開している．

は子を持たないが，Env型の値が子を持たない場合もある．以降，子を持たないものも含めEnv型の値を内部ノードと呼ぶ．またVal型，Env型の両方を指すものとして，env型の値を環境と呼ぶ．

ある木構造において，根から特定のノードに至るパス（経路）はそれを構成する枝のリストとして表せる．よって（$'a$, $'b$, $'c$）env型の木構造におけるパスは$'c$ list型で表される．以降，パスpが別のパスp'の真の接頭辞であるとき，pはp'よりも上であるもしくはp'はpより下であるという．あるパスpが別のパスp'より上であることは，リスト上の真の接頭辞関係を表す組み込み関数strict_prefixを用いてstrict_prefix p p'と表せる．異なる2つのパスについて一方がもう一方の上という関係にないとき，これらは並列であるという．

lookup関数は環境eとパスxsに対し，eにおけるパスxsに存在する要素を返す．指定されたパスに要素が存在しない場合があるため，返り値の型をenv option型とし存在する場合はSome fを，そうでない場合はNoneを返す．ある環境においてパスxsに要素が存在するとき，xsを定義済みのパスと呼び，そうでない場合は未定義パスと呼ぶ．

update関数は環境eとパスxsおよびenv option型の値optに対し，optがSome e'の場合はeにおいてパスxsに位置する要素をe'で置き換えた環境を返す．またoptがNoneの場合はパスxsの要素を削除した環境を返す．ただしパスが空の場合すなわち根に対する更新においては，optがNoneの場合は元の環境をそのまま返す．

2.2 ファイルシステムに関する概念

ファイルシステムの構成要素であるプレーンファイルやディレクトリの名前を表す型をnameとし，簡単のため自然数を表すnat型の別名として定義する．また同様にユーザIDを表す型をuidとし，これもnat型の別名として定義する．uidの値が0であるユーザは管理者を表す．ファイルやディレクトリへのアクセス権限を表す型permは，3種類の値を持つものとして次のように定義される．

```
datatype perm = Readable | Writable | Executable
```

これらは順に読み込み可能，書き込み可能，実行可能を表すが，実行可能権限はこの形式化では取り扱わない．

型attはファイルやディレクトリの属性を表し，所有者のユーザID ownerとその他のユーザに与えられるアクセス権限の集合othersの2つのフィールドからなるレコード型として次のように定義される．

```
record att = owner :: uid
             others :: perm set
```

ファイルシステム中の要素を表す型fileは，env型の型変数を次のように具体化したものとして定義される．

```
type-synonym file = (att × string, att, name) env
```

以降，file型の値を単に要素と呼ぶ．要素のうちVal型の値はプレーンファイルを表し，そのファイル属性を表すatt型の値とファイル内容を表すstring型の値を持つ．もう一方のEnv型の値はディレクトリを表し，そのディレクトリ属性を表す値を持つ．file型の値はそれを根とする木構造を表すため，file型を持つ変数の名前として以降rootを用いる．またこの木構造上のパスはname list型で表されるため，この型name listの別名として型pathを定義し，またこの型を持つ変数の名前としても以降pathを用いる．

関数accessはアクセス権限を考慮した参照を行う関数であり，次のように定義される．

```
access root path uid perms =
  (case lookup root path of
    None ⇒ None
  | Some file ⇒
      if uid = 0 ∨ uid = owner (attributes file)
      ∨ perms ⊆ others (attributes file) then Some file else None)
```

すなわち，指定した要素が存在しかつpermsに含まれるすべての種類のアクセス権限がユーザuidに認められる場合にのみその要素を返す．そのような場合とは，ユーザが管理者か所有者のいずれかであるか，またはpermsの権限すべてがその他のユーザに認められるものである場合である．アクセス権限の要求のない呼び出しaccess root path uid {}は単なる参照lookup root pathと等価である．

2.3 システムコールに関する概念

ファイルシステムを操作するためのシステムコールを表すoperation型は８種類の値を持つものとして次のように定義される．

```
datatype operation =
    Read uid string path | Write uid string path | Chmod uid perms path
  | Creat uid perms path | Unlink uid path
  | Mkdir uid perms path | Rmdir uid path | Readdir uid "name set" path
```

以降，operation型の値を操作と呼ぶ．どの操作も，その操作の実行ユーザを表すuidと，その操作対象の要素のパスを表すpathが付随する．操作xに付随するこれらのuid，pathをそれぞれuid_of x，path_of xで表し，特にpath_of xを操作パスと呼ぶ．プレーンファイルの内容およびディレクトリのエントリ集合の読み込みを表すRead，Readdirは，その結果を表すstringまたはname set型の値を持つ．またプレーンファイルへの書き込みを表すWriteは，その書き込み内容を表すstring型の値を持つ．新規プレーンファイル・ディレクトリの作成を表すCreat，Mkdirおよび要素の属性を更新するChmodは，その作成・更新される要素に設定されるアクセス権限を表すperms型の値を持つ．

これらの操作の意味は，述語transitionによって表される．transition $r\ x\ r'$は要素rを根として操作xを行った結果が要素r'である関係を表し，略記としてr −−x−−> r'を用いる．ある要素rootに対し，操作xの種類ごとのroot −−x−−> $root'$が成り立つ条件$condition$および操作の結果$root'$の組み合わせを**表 1** に示す．条件$access$（$perms$, $file$）は操作パスに要素$file$が存在し，それに対し操作ユーザが$perms$に含まれる全ての権限を持つことを表す．条件uidは操作ユーザが管理者もしくは操作パスに存在するファイルの所有者であることを表す．条件$parent$は操作パスの親ディレクトリに対し書き込み権限を持つことを表す．条件$none$は操作パスが未定義であることを表す．upd（$file$）は操作パスに位置する要素を$file$で置き換えた要素，upd_{none}は操作パスに位置する要素を削除した要素を表す．f_{chmod}は操作パスに位置する要素fileに含まれるatt型のレコード値について，そのフィールドothersの値をpermsで置き換えて得られる新しい要素を表す．f_{creat}およびf_{mkdir}はどちらもファイルの属性としてuidとpermsを持つ，空のプレーンファイルおよび空のディレクトリを表す．

任意の要素rと操作xに対し，r −−x−−> r'を満たすr'が一意に定まることが示せる．操作の列xsについて，要素rを根としてxsの操作を先頭から順に行った結果がr'である関係transitions $r\ xs\ r'$が定義され，その略記としてr ==xs==> r'を用いる．

2.4 削除不能性を表す不変条件

原形式化では，UNIXファイルシステムで生じる特殊な状況の例として，あるユーザが所有するディレクトリの要素がそのユーザ自身の操作のみでは削除不能となる状況を示している．以降このユーザをuser1で表し，このユーザは管理者ではないものとする．すなわちuser1 ≠

x	*condition*	*root′*
Read uid text path	*access* ({Readable}, Val (att, text))	root
Write uid text path	*access* ({Writable}, Val (att, text′))	*upd* (Val (att, text))
Chmod uid perms path	*access* ({}, file) and *uid*	*upd* (f_{chmod})
Creat uid perms path	*parent* and *none*	*upd* (f_{creat})
Unlink uid path	*parent* and *access* ({}, Val plain)	upd_{none}
Mkdir uid perms path	*parent* and *none*	*upd* (f_{mkdir})
Rmdir uid path	*parent* and *access* ({}, Env att′ Map.empty)	upd_{none}
Readdir uid names path	*access* ({Readable}, Env att dir) and names = dom dir	root

上記の表においては，次の表記を用いている．

access (*perms*, *file*)	access root path uid *perms* = Some (*file*)
uid	uid = 0 ∨ uid = owner (attributes file)
parent	path = parent_path @ [name] and access root parent_path uid {Writable} = Some (Env att parent)
none	access root path uid {} = None
upd (*file*)	update path (Some *file*) root
upd_{none}	update path None root
f_{chmod}	map_attributes (others_update (λ_.perms)) file
f_{creat}	Val ((|owner = uid, others = perms|), [])
f_{mkdir}	Env (|owner = uid, others = perms|) Map.empty

表 1：操作の種類ごとの成立条件

0とする．このuser1にとっての削除不能性を表す命題は，特定の要素rootおよびそこからのパスpathについての述語invariantとして，次のように表される．

定義（削除不能性）．user1をあるユーザとする．このユーザにとっての要素rootにおけるパスpathの削除不能性invariant root pathとは，rootにおいてパスpathに空でないディレクトリが存在し，user1はそのディレクトリの所有者ではなく，そのディレクトリへの書き込み権限も持たないことである．

```
invariant root path ⟷
 (∃att dir. access root path user1{} = Some (Env att dir) ∧ dir ≠ Map.empty ∧
    user1 ≠ owner att ∧ access root path user1 Writable = None)
```

以降，この性質におけるパスpathを削除不能パス，このパスに存在する空でないディレクトリを削除不能ディレクトリと呼ぶ．要素*root*の直下にuser1が書き込み可能なディレクトリh1が存在するものと仮定すると，*root*に対し実行することでこの性質が成り立つような状態に至る操作列の例として，次のものが考えられる．

```
[Mkdir user1 {Writable} [h1, d1], Mkdir user2 {} [h1, d1, d2],
 Creat user2 {} [h1, d1, d2, f]]
```

ここでuser2はuser1とは異なるユーザとする．この命令列を*xs*とすると，*root* ==*xs*==> *root'*のとき，invariant *root'* [h1, d1, d2] が成り立つ．すなわち，user2が作成したd2が削除不能ディレクトリである．

ある要素rootとそこからのパスpathについてこの削除不能性が成り立つとき，その要素に対しuser1がいかなる操作を実行した結果に対しても，同じパスにおいてこの性質が成り立つ．このことは次の命題preserve_invariantとして表される．

```
lemma preserve_invariant：
  assumes "root --x--> root' " and "invariant root path" and "uid_of x = user1"
  shows "invariant root' path"
```

この命題が成り立つこと，すなわちuser1にとっての削除不能性invariantがuser1による操作に対する不変条件であることは，user1はpathに存在するディレクトリを自身の操作のみでは削除できないことを意味する．

原形式化におけるこの不変条件の証明においては，削除不能パスに対する操作パスの相対位置について並行，一致，上，下の4通りの場合分けを行っている．並行の場合は操作が影響を与えないことから不変性が容易に示せる．また削除不能ディレクトリが非空かつ所有者がuser1でないことより，操作パスが削除不能パスと一致する場合は起こり得ないことが示せる．また操作パスが上の場合はそれより下のパスの不変性により示される．操作パスが下の場合が最も記述量が多く，約30ステップからなる記述の大半は，削除不能ディレクトリが操作後に空でないことの導出である．

3 形式化の拡張

本節では，原形式化をもとに本研究で新たに証明する性質について述べる．またそれらの性質を用いて，削除不能性における削除不能ディレクトリが空でないことの証明に有用な補題が得られることを示す．以降の説明においては，各性質についての実際の形式化のうち命題部分のみを示し，証明部分は省略する．

3.1 木の更新に関する性質

update関数による更新の正しさを表す自然な性質として，更新が確かになされていること，すなわちある値*x*で更新したパスを直ちにlookup関数で参照するとその*x*が得られること

がある．原形式化においてはファイルを表す値Some fによる更新の場合についての性質のみを示していたが，これは削除を表す値Noneによる更新についても同様の性質が成り立つ．ただし，空のパスに対するNoneでの更新は値の維持を意味するため，パスが空ではないという条件が必要となる．そのような性質は次のように表される．

```
lemma assumes "lookup env path = Some file" and "path ≠ []"
  shows "lookup (update path None env) path = None"
```

これにより，定義済みかつ空でないパスの更新と参照について，更新に用いる値の種類によらない次の一般的な性質が得られる．

```
lemma assumes "lookup env path = Some file" and "path ≠ []"
  shows "lookup (update path opt env) path = opt"
```

また類似の性質として，未定義パスに対して新たに追加したノードが参照により得られることがある．この場合，その未定義パスの親がディレクトリである必要がある．

```
lemma assumes "path = parent @ [name]" and "lookup env parent = Some (Env b es)"
  shows "lookup (update path (Some env') env) path = Some env'"
```

入れ子構造を表すenv型は再帰的・非破壊的に定義される代数的データ型であるため，あるパスに位置するノードの更新はそれより上に存在するすべての内部ノードの再構築を伴う．しかしながら，それらの各内部ノードEnv b esのうち更新されるのは部分関数esのみであり，ノードに付随する値bは不変である．このことは次の命題で表される．

```
lemma assumes "lookup env xs = Some (Env b es)"and "strict_prefix xs ys"
  obtains es' where "lookup (update ys opt env) xs = Some (Env b es')"
```

ファイルシステムの操作という観点では，内部ノードの付随値はディレクトリのアクセス権限を表すため，この命題はプレーンファイルやディレクトリを更新した場合も，それより上に位置するディレクトリのアクセス権限は不変であるという，自然な性質に対応する．

3.2 終端パスとその性質

プレーンファイルと空のディレクトリは，それが存在するパスは定義済みであるが，そのパスよりも下のパスは未定義であるという点で特徴的である．よって，そのような要素を終端要素と呼び，それらが存在するパスを終端パスと呼ぶものとする．ある要素が終端要素であるか否かを判定する述語terminalを次のように定義する．

```
fun terminal where
  "terminal (Val _) = True" | "terminal (Env _ es) = (es = Map.empty)"
```

これを用いて，終端パスを表す述語terminal_pathを次のように定義する．

```
definition terminal_path where
  "terminal_path root path = (∃env. lookup root path = Some env ∧ terminal env)"
```

終端パスより下のパスが未定義となることを表す，次の性質が成り立つ．

```
lemma "terminal_path root path ⟹ lookup root (path @ y # ys) = None"
```

未定義パスに対する操作と終端パスとの関係について述べる．まず，未定義パスに対して実行可能な操作はCreat，Mkdirのいずれかのみである．

```
lemma assumes "root --x--> root'" and "lookup root (path_of x) = None"
  shows "(∃uid perms path. x = Creat uid perms path)
       ∨(∃uid perms path. x = Mkdir uid perms path)"
```

この２種類の操作それぞれを行ったとき，操作パスは終端パスとなる．

```
lemma "root--(Creat uid perms path)--> root' ⟹ terminal_path root' path"
lemma "root--(Mkdir uid perms path)--> root' ⟹ terminal_path root' path"
```

以上より，未定義パスに対して操作を行ったとき，操作パスは操作後に終端パスとなる．

```
lemma assumes "root --x--> root'" and "lookup root (path_of x) = None"
      shows "terminal_path root' (path_of x)"
```

3.3 操作に対する不変性

８種類の操作のうち，読み取り操作に相当するRead，Readdirの２つは要素を変化させない．一方，残りの６種類の操作はその対象とするパスに対しノードの追加・更新・削除を行う．操作パスと並列なパスについての不変性は原形式化において示されているが，操作パスより上や下の位置についても，ある種の不変性を示すことができる．これらは削除不能性の証明においても重要な役割を果たす．

3.3.1 操作パスよりも上の位置の不変性

操作パスは何らかのファイル・ディレクトリ要素が既に存在する，もしくは新たに要素が作成されるパスであるから，ある操作が実行可能であるとき，その操作パスより上のパスの要素はディレクトリであることが示せる．また3.1節で示した木の更新における内部ノードの付随値の不変性より，そのディレクトリの属性は不変であることが示せる．

```
lemma assumes "root --x--> root'" and "strict_prefix path (path_of x)"
      obtains att dir dir' where
        "lookup root path = Some (Env att dir)"
    and "lookup root' path = Some (Env att dir')"
```

3.3.2 操作パスよりも下の位置の不変性

原形式化における削除不能性の証明において，操作パスよりも下のパスが定義済みである場合の不変性が示されている．これを補題として独立させると次のように表される．

```
lemma assumes "root --x--> root'" and "strict_prefix (path_of x) path"
  and "lookup root path = Some file"
shows "lookup root' path = Some file"
```

これと同様に，未定義パスについての不変性を考える．木の内容や構造を変化させる6種類の操作のうち，ノードの更新を行うWrite，Chmodは木の構造を維持するため，そのノードの下の未定義性も維持される．また未定義パスに対しノードの追加を行うCreat，Mkdirは操作後にその位置が終端パスとなるためその下は未定義となる．残りの削除操作Unlink，Rmdirは操作後にその位置が未定義となるためその下も未定義となる．よって操作パスよりも下の未定義パスについても，操作に対する不変性すなわち操作後もその位置が未定義であることを示せる．

```
lemma assumes "root --x--> root'" and "strict_prefix (path_of x) path"
  and "lookup root path = None"
shows "lookup root' path = None"
```

証明においては，操作パスが未定義か定義済みか，また定義済みの場合はプレーンファイルかディレクトリかの場合分けを行っている．その中の未定義の場合において，3.2節で示した未定義パスが終端パスとなる性質を利用している．

定義済み・未定義の両方の場合の不変性より，操作パスよりも下の位置の一般的な不変性が

得られる．

```
lemma assumes "root --x--> root'" and "strict_prefix (path_of x) path"
        shows "lookup root path = lookup root' path"
```

3.4 不変性の応用

操作パスより下のパスの不変性より，操作の前後で変化するのは操作パスより上，すなわち根から操作パスまでの間に存在する要素のみである．したがって，ある操作の前後で何らかの要素が変化した場合，その操作パスは変化した要素のパスと同一かそれより下であることが示せる．

```
lemma assumes "root --x--> root'" and
              "lookup root path ≠ lookup root' path"
        shows "prefix path (path_of x)"
```

この性質より，削除可能性の証明において本質的であり，かつ直感的に理解しやすい次の補題が示される．ある操作によって特定のディレクトリが非空から空に変化した場合，操作の実行者はそのディレクトリへの書き込み権限を持ち，その操作パスはディレクトリ直下である．

```
lemma assumes "root --x--> root'" and
              "lookup root path = Some (Env att dir)" and
              "lookup root' path = Some (Env att' dir')" and
              "dir ≠ Map.empty" and "dir' = Map.empty"
        shows "access root path (uid_of x) Writable ≠ None"
          and "∃name. path_of x = path @ [name]"
```

形式証明の概略を述べる．ディレクトリが操作前に空でないことから，その直下のパスのうち削除操作の対象を表すパス，すなわち操作前は定義済みで操作後は未定義となるものが存在する．これより，操作パスがディレクトリ直下であることは，

- 前述の補題より，操作パスがディレクトリ直下かそれより下であること
- 操作パスがディレクトリ直下より下であるとすると，操作パスより上の不変性よりディレクトリ直下が操作前・操作後共にディレクトリとなり，操作後に未定義であることと矛盾

から直ちに従う．そのパスに対する操作が成立する条件より，その親ディレクトリへの書き込み権限を持つことも直ちに導かれる．

この補題を利用すると，削除不能性の不変性を表す命題`preserve invariant`の証明における4通りの場合分けのうち操作パスが削除不能パスより下の場合において，削除不能ディレクトリが空でないことが容易に導かれる．具体的には，原形式化において`strict_prefix path（path_of x）`を仮定してから削除不能ディレクトリが空でないことを導くまでの20ステップ程度の推論を，補題を用いた2ステップで置き換えることが可能となる．

4 おわりに

本論文では，WenzelによるUNIXファイルシステムの形式化において成り立つ性質の拡張として，木の更新結果が参照により得られる性質，ファイルやディレクトリのパスの新規作成操作の対象パスが終端パスとなる性質，また操作パスに対して並列でない位置が不変である性質を示した．またこれらの性質を用いて，形式化の重要な結論である削除不能性の証明において有用となる，ディレクトリを空にする操作と書き込みアクセス権限との関係を表す補題を示した．

今回の証明においてはいくつかの箇所で，8種類のどの操作に対しても成り立つ性質を示す際に`cases`メソッドによる場合分けを用いている．場合分けによるサブゴール生成に引き続きそれら全ての証明を完了させる十分強力なメソッドを適用することで，簡潔かつ操作の種類の拡張に対し頑強な証明が構成できる．一方，各操作においてなぜその性質が成り立つのかを説明する記述としては，そのような簡潔な証明は不十分である．より一般的には，`by（cases, use` *lemmas* `in` *method*`）`のような記述からは，利用を明示した補題*lemmas*や全体に適用する強力な手法*method*の特徴が，`cases`が生成するどのサブゴールにおいて重要であるかが読み取れないという問題がある．簡潔かつ可読性の高い記述を行うためには，証明方針の共通性に従いサブゴールを分類し，グループごとに証明を記述するための機能が必要となると考えられる．

定理証明支援システムを用いた形式検証の意義は，期待される性質が成り立つことを厳密に保証できる点に留まらず，性質の証明を通してその根拠である仕様の妥当性について確信を深められる点も大きい．仕様に膨大な場合分けを本質的に含む大規模なソフトウェアの形式検証において，場合分けにより生じる関連の深い小問題群を証明上で整理し理解を支援する機能の更なる整備は今後の課題であると考えられる．

参考文献

1）Gerwin Klein, et al. : seL4 : Formal verification of an OS kernel, Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, 2009.
2）Xavier Leroy : Formal verification of a realistic compiler, Communications of the ACM, Volume 52, Issue 7, 2009.
3）MakariusWenzel : Some aspects of Unix file−system security, Isabelle/Isar proof document : `https : //isabelle.in.tum.de/dist/library/HOL/HOL-Unix/outline.pdf`, 2001.

4）Tobias Nipkow, Programming and proving in Isabelle/HOL, Part of the Isabelle documentation : https : //isabelle.in.tum.de/doc/prog-prove.pdf, 2025.
5）Makarius Wenzel et al : The Isabelle/Isar manual, Part of the Isabelle documentation : http : //isabelle.in.tum.de/doc/isar-ref.pdf, 2025.