定理証明支援系Coqによる持続型例外処理機構の形式化

森 公哉[1,a)]　江本 健斗[1]

概要：ごみ集めや一級継続などの高度なプログラミング言語機能は，その実装が安全で正しくなければ意味がない．その様な高度な言語機能を安全に実装するための機構として，持続型例外処理機構が提案された．この機構は，関数呼び出しの先祖をたどり，そのローカル変数への合法な（安全な）アクセスを提供する．しかし，その制御構造が通常のプログラム実行とは異なるため，意図通りの動作を実装できているかの確認が難しく，意図通りの動作を保証する仕組みの提供が望ましい．本研究は，持続型例外処理機構を定理証明支援系 Coq 上で形式化し，持続型例外処理機構を用いたプログラムの形式的検証を行える環境の構築を行う．

キーワード：定理証明支援系，Coq，形式化，Scheme

Formalization of Resumable Exception Mechanism in Coq

Mori Kouya[1,a)]　Emoto Kento[1]

Abstract: High-level programming language features such as garbage collection and first-class continuations have to be implemented safely and correctly. Resumable-exception mechanism has been proposed to provide a safe way to implement such language features. It provides a safe way to access local variables in functions called so far. However, its control flow differs from that of the ordinal program execution, and thus it is not so easy to verify whether language features are implemented correctly. In this research, we formalize the mechanism in Coq, a proof assistant, to provide a formal method to verify properties of programs written with the mechanism.

Keywords: Proof Assistant, Coq, Formalization, Scheme

1. はじめに

ごみ集め・一級継続・動的負荷分散などの高度なプログラミング言語機能は，その実装が安全でかつ正しくなければ意味がない．しかし，これらの機能のアドホックな実装は，間違ったメモリ領域の解放などの危険を容易に生じる．

高度な言語機能を安全に実装するための機構として，持続型例外処理機構が提案された [6]．この機構は，関数呼び出しの先祖をたどり，そのローカル変数への合法な（安全な）アクセスを提供する．本機構を用いることにより，ごみ集め・一級継続・動的負荷分散といった，計算状態の動的再構成を必要とする高度な言語機能を高信頼かつ高性能に実現できる．

持続型例外処理機構は，一方で，その制御構造が通常のプログラム実行とは異なるため，意図通りの動作を実装できているかどうかの確認が簡単であるとは言えない．そのため，持続型例外処理機構を用いたプログラムが意図通りの動作をすることを保証するための仕組みが望まれる．

近年，プログラムの性質に関する形式的な保証を与える手法として，Coq [1] や Agda [3] 等の定理証明支援系の利用が広く認知されてきた．この手法では，定理証明支援系の提供するプログラミング言語で対象プログラムやその性質を記述し，その性質の証明を定理証明支援系の確認のもとで対話的に行っていく．証明が完了した際には，その証明が見落としなく正しいものであると保証される（人手で

[1] 九州工業大学
Kyushu Institute of Technology, Iizuka, Fukuoka 820–8502, Japan
[a)] k_mori@pl.ai.kyutech.ac.jp

$$
\begin{array}{rcl}
E & ::= & \texttt{(quote } c\texttt{)} \mid I \mid L \\
 & \mid & \texttt{(if } E_0\ E_1\ E_2\texttt{)} \mid \texttt{(set! } I\ E_0\texttt{)} \mid \texttt{(}E_0\ E_1\ \cdots\texttt{)} \\
L & ::= & \texttt{(lambda (}I_1\ \cdots\texttt{)}\ E\texttt{)} \\
c & ::= & \mathrm{TRUE} \mid \mathrm{FALSE} \mid \mathrm{NUM} : z \\
z & \in & \textbf{Number} \\
\alpha, \beta & \in & \textbf{Location} \\
I & \in & \textbf{Identifier}
\end{array}
$$

図 1 Scheme の文法

$$
\begin{array}{rcl}
\textbf{Configuration} & ::= & \langle v, \sigma\rangle \mid \langle E, \rho, \kappa, \sigma\rangle \mid \langle v, \rho, \kappa, \sigma\rangle \\
v \in \textbf{Value} & ::= & c \mid \mathrm{UNSPECIFIED} \mid \mathrm{UNDEFINED} \\
 & \mid & \mathrm{PRIMOP} : \psi \mid \mathrm{CLOSURE} : \langle \alpha, L, \rho\rangle \\
\kappa & ::= & \textbf{halt} \\
 & \mid & \textbf{select} : \langle E_1, E_2, \rho, \kappa\rangle \\
 & \mid & \textbf{assign} : \langle I, \rho, \kappa\rangle \\
 & \mid & \textbf{push} : \langle\langle E, \ldots\rangle, \langle v, \ldots\rangle, \rho, \kappa\rangle \\
 & \mid & \textbf{call} : \langle\langle v, \ldots\rangle, \kappa\rangle \\
\rho & \in & \textbf{Identifier} \to \textbf{Location} \\
\sigma & \in & \textbf{Location} \to \textbf{Value}
\end{array}
$$

図 2 Scheme の計算状態

の証明では見落としがないことの保証はない).

本研究は，持続型例外処理機構を定理証明支援系 Coq 上に形式化し，同機構を用いるプログラムの形式的検証を行える環境を提案する．具体的には，既存の Scheme 言語の定式化 [2] に対し，持続型例外処理機構の構文と意味を追加し，Coq 上に形式化する．また，提案する形式化の評価として，簡単なプログラムの性質の証明を示す．

以降の本論文の構成は以下のとおりである．まず，第 2 節にて，既存の事柄についての導入を行う．次に，第 3 節にて，持続型例外処理機構の形式化を示す．そして，第 4 節にて，簡単なプログラムについての証明例を通して提案手法を評価する．最後に，第 5 節にて，本論文をまとめる．

2. 準備

本節では，本論文で用いる既存結果について導入する．

2.1 Scheme 言語とその定式化

本論文は，ベースのプログラミング言語として Scheme 言語を用いる．そのために，以下，William らによる定式化 [2] を導入する．

図 1 に Scheme の文法を示す．Scheme のプログラムは式 E であり，定数 c, 変数 I, ラムダ式 L, 条件分岐, 代入, 関数呼び出しからなる．ラムダ式 L は，仮引数の列 $I_1\ \cdots$ と本体の式 E から構成される．定数 c は，Bool 値 TRUE と FALSE のほか，整数値がある．なお，元の定式化に存在する SYM と VEC は，本論文で扱う形式化等に現れないため略している．**Number**, **Location**, **Identifier** は，それぞれ，数値，メモリアドレス，識別子の集合である．

Scheme の計算状態（**Configuration**）を図 2 に示す．Scheme の計算状態は，二つ組の最終状態 $\langle v, \sigma\rangle$ か，四つ組の中間状態 $\langle E, \rho, \kappa, \sigma\rangle$ ないし $\langle v, \rho, \kappa, \sigma\rangle$ である．ここで，ρ は変数からアドレスへの，σ はアドレスから値へのマッピングである．値 $v \in$ **Value** は，定数 c，不定の値 UNSPECIFIED（存在は保証されるが正体は不明な値），未定義の値 UNDEFINED（存在の保証もされない），四則演算等の基礎的な演算子 PRIMOP，クロージャ CLOSURE のいずれかである．クロージャは，主としてラムダ式 L と自身が作成された時点の環境（変数からアドレスへのマッピング）のペアであり，歴史的事情によりタグとなるアドレス α も保持する．元論文に存在する ESCAPE は，本論文で扱う形式化等に現れないため略している．さらに，κ は継続であり，構文の処理を進める際，その一部の処理を保留しておく際に用いられる．継続の詳細は後述する．なお，元の定式化では **push** に順列 π が含まれるが，本論文では簡単のためこれを恒等射に固定して省略する．

最後に，Scheme の意味を図 3 に示す．Scheme の意味は大きく二つの遷移規則で与えられ，計算状態の第一要素が式 E の場合に用いられる簡約規則と，値 v の場合に用いられる継続規則である．

簡約規則は，計算状態の第一要素の式 E を値 v にする規則である．定数式はその定数の値に，変数式はその変数の中身の値に，ラムダ式はクロージャに置き換わる．残りの規則では，継続 κ に，注目中の式 E_0 を値にし終えた後にやるべき処理を保存する．例えば，`if` の際には，E_0 を値にし終えた後に E_1 か E_2 の処理を実行できるよう，継続にそれらを積む．同様に，(`set!` $I\ E_0$) では，E_0 の結果を I に代入する操作を継続に積む．関数適用では，引数を後ろから順に評価する継続を積む．

継続規則は，計算状態から最新の継続を一つ取り出し，実行の保留されていた計算（式）を第一要素に設定した計算状態に遷移する．例外として，継続が **halt** の場合には終了状態に遷移する．継続 **select** は，`if` の条件部分の結果が FALSE 以外であれば E_1 を，FALSE であったら E_2 を実行する状態に遷移する．継続 **assign** は，得られた結果 v を変数 I の領域にストアし，不定の値を返す．継続 **push** は関数適用時の引数の評価を順に行い，最後に継続 **call** によって関数本体を実行する状態へ遷移する．

なお，元の形式化には Scheme 言語のごみ集めの規則も存在するが，本論文の形式化では扱わないため省略する．

2.2 持続型例外処理機構

持続型例外処理機構 [6] は，関数呼び出し先祖のローカル変数への合法な（安全な）アクセスを提供する機構である．その動作は，通常の例外処理機構と似ており，例外が発生した際に，その例外に対応するよう予め登録しておいたハンドラに実行制御が移る．しかし，通常の例外処理機

簡約規則：

$$
\begin{aligned}
\langle (\texttt{quote}\ c), \rho, \kappa, \sigma\rangle &\rightarrow \langle c, \rho, \kappa, \sigma\rangle \\
\langle I, \rho, \kappa, \sigma\rangle &\rightarrow \langle \sigma(\rho(I)), \rho, \kappa, \sigma\rangle \text{ where } I \in \mathrm{Dom}\ \rho\ \wedge\ \rho(I) \in \mathrm{Dom}\ \sigma\ \wedge\ \sigma(\rho(I)) \neq \mathrm{UNDEFINED} \\
\langle L, \rho, \kappa, \sigma\rangle &\rightarrow \langle \mathrm{CLOSURE} : \langle \alpha, L, \rho\rangle, \rho, \kappa, \sigma[\alpha \mapsto \mathrm{UNSPECIFIED}]\rangle \text{ where } \alpha \notin L, \rho, \kappa, \sigma \\
\langle (\texttt{if}\ E_0\ E_1\ E_2), \rho, \kappa, \sigma\rangle &\rightarrow \langle E_0, \rho, \mathbf{select} : \langle E_1, E_2, \rho, \kappa\rangle, \sigma\rangle \\
\langle (\texttt{set!}\ I\ E_0), \rho, \kappa, \sigma\rangle &\rightarrow \langle E_0, \rho, \mathbf{assign} : \langle I, \rho, \kappa\rangle, \sigma\rangle \\
\langle (E_0\ E_1\ \cdots), \rho, \kappa, \sigma\rangle &\rightarrow \langle E_0', \rho, \mathbf{push} : \langle\langle E_1', \ldots\rangle, \langle\rangle, \rho, \kappa\rangle, \sigma\rangle \text{ where } \langle E_0', E_1', \ldots\rangle = \mathrm{reverse}(\langle E_0, E_1, \ldots\rangle)
\end{aligned}
$$

継続規則：

$$
\begin{aligned}
\langle v, \{\}, \mathbf{halt}, \sigma\rangle &\rightarrow \langle v, \sigma\rangle \\
\langle v, \rho', \mathbf{halt}, \sigma\rangle &\rightarrow \langle v, \{\}, \mathbf{halt}, \sigma\rangle \\
\langle v, \rho', \mathbf{select} : \langle E_1, E_2, \rho, \kappa\rangle, \sigma\rangle &\rightarrow \langle E_1, \rho, \kappa, \sigma\rangle \text{ where } v \neq \mathrm{FALSE} \\
\langle \mathrm{FALSE}, \rho', \mathbf{select} : \langle E_1, E_2, \rho, \kappa\rangle, \sigma\rangle &\rightarrow \langle E_2, \rho, \kappa, \sigma\rangle \\
\langle v, \rho', \mathbf{assign} : \langle I, \rho, \kappa\rangle, \sigma\rangle &\rightarrow \langle \mathrm{UNSPECIFIED}, \rho, \kappa, \sigma[\rho(I) \mapsto v]\rangle \\
\langle v_0', \rho', \mathbf{push} : \langle\langle E_1', E_2', \ldots\rangle, \langle v_1', \ldots\rangle, \rho, \kappa\rangle, \sigma\rangle &\rightarrow \langle E_1', \rho, \mathbf{push} : \langle\langle E_2', \ldots\rangle, \langle v_0', v_1', \ldots\rangle, \rho, \kappa\rangle, \sigma\rangle \\
\langle v_0, \rho', \mathbf{push} : \langle\langle\rangle, \langle v_1, \ldots\rangle, \rho, \kappa\rangle, \sigma\rangle &\rightarrow \langle v_0, \rho, \mathbf{call} : \langle\langle v_1, \ldots\rangle, \kappa\rangle, \sigma\rangle \\
\langle \mathrm{CLOSURE} : \langle \alpha, L, \rho\rangle, \rho', \mathbf{call} : \langle\langle v_1, \ldots, v_n\rangle, \kappa\rangle, \sigma\rangle &\rightarrow \langle E, \rho'', \kappa, \sigma'\rangle \text{ where } L = (\texttt{lambda}\ (I_1\ \cdots)\ E) \\
&\qquad \beta_1, \ldots, \beta_n \notin E, \rho, \kappa, \sigma \\
&\qquad \rho'' = \rho[I_1, \ldots, I_n \mapsto \beta_1, \ldots, \beta_n] \\
&\qquad \sigma' = \sigma[\beta_1, \ldots, \beta_n \mapsto v_1, \ldots, v_n]
\end{aligned}
$$

図 3 Scheme の意味（記号 $\notin$ は，左のものが右のものの中に現れないことを意味する）

構とは異なり，ハンドラの実行終了時に例外が投げられた時点へと復帰する．

この機構は二つの構文 `do-handle` と `hcall` を用いる：

```
(do-handle L E)
(hcall E0 E1 ...)
```

構文 `do-handle` は，持続型例外の処理を行うハンドラ L を登録し，式 E を実行する．この式 E の実行中に，他のハンドラによって捕捉されない持続型例外が発生すると，このハンドラ L が実行される．そして，L の実行後は，例外の発生地点から実行が再開される．構文 `hcall` は，式 $E_0, E_1, \ldots$ を引数に持続型例外を発生させる．

簡単な例として，次の小さな関数を考える．この関数は，「受け取った引数 `x` の値で変数 `n` を上書きする」というハンドラを登録した後，「変数 `n` を出力，引数として 3 を与えた持続型例外を発生，再び変数 `n` を出力」という式を実行する．ここで，`do` は，引数を順に実行する関数である．

```
1 (lambda (n)
2  (do-handle
3   (lambda (x) (set! n x))
4    (do (print n) (hcall 3) (print n))))
```

この関数を実引数 `5` に対して呼び出すと，その出力は `5 3` となる．最初の `n` の出力時には `n` の値は実引数で渡された `5` であるが，その直後の `hcall` による持続型例外の発生によりハンドラが実行され，`n` が持続型例外の引数の値 `3` で上書きされる．そして，ハンドラの実行終了後に `hcall` 直後に復帰し，二回目の `n` の出力を行うものの，ハンドラによって `n` の値が書き換えられているためその出力は `3` となる．ここでは簡単のため同一関数内でハンドラが動作する例を示したが，一般には，通常の例外処理と同様に深い関数呼び出しから最寄りのハンドラまで一気に制御がジャンプする．

2.3 定理証明支援系 Coq

定理証明支援系 Coq [1] は，依存型とカリー・ハワード同型対応に基づき，形式的証明の正しさを保証するシステムである．ユーザは，Coq の提供する関数型言語 Gallina を用いて，プログラムを含む数学的オブジェクトやその性質に関する定理（命題）を記述する．そして，Coq の型システムによる確認のもとで，ユーザはその定理の証明をタクティックというコマンドを用いて対話的に進めていく．証明が完了すれば，その証明が見落としなく正しいことが保証される．

Coq ではユーザが独自にデータ型（`Type`）を定義することができる．例として，自然数を考える．自然数は，「0 は自然数である」と「n が自然数なら，$n+1$ も自然数である」により，帰納的に定義される．Coq はこの様な帰納的データ型を定義するコマンド **Inductive** を提供しており，これを用いると自然数の型 `nat` は次のように定義される：

```
1 Inductive nat : Type :=
2   | O
3   | S (n : nat).
```

ここで，`O` と `S` は `nat` の要素を構成するための演算子であり，構成子と呼ばれる．前者は「0 は自然数である」に対応し，後者は「n が自然数なら，$n+1$ も自然数である」に対応するため引数として自然数 n を受け取る．例えば，`S O` は 1 を[*1]，`S (S O)` は 2 を意味する．

また，定義した `nat` 型を対象とする関数を定義すること

[*1] 関数型言語であるため，引数に括弧をつけないことに注意する．

も可能である．以下は nat 型の引数 n と m とを加算する関数 plus の定義である．

```
1 Fixpoint plus (n : nat) (m : nat) : nat :=
2   match n with
3   | O ⇒ m
4   | S n' ⇒ S (plus n' m)
5   end.
```

Coq では，再帰関数の定義にはコマンド **Fixpoint** を用いる．このコマンドの直後が関数名であり，続いて引数がその型とともにならび，その後の : の後に戻り値の型が書かれて := の右辺に関数本体が記述される．この関数の本体は，**match** 構文を用いて n に関する場合分けを行っている．n が O のとき，すなわち $n = 0$ のときは，$n + m = m$ となるため，⇒ の後ろに書かれる結果が m になっている．他方，S n' であるとき，すなわち，$n = 1 + n'$ のときには，$n + m = 1 + (n' + m)$ であるから，⇒ の後ろは S (plus n' m) として再帰呼び出しを行っている．この再帰がいずれ止まって和が計算できることは容易に分かる．

最後に，簡単な定理の証明例として，$0 + n = n$ を示す：

```
1 Theorem plus_O_n' : ∀n : nat, plus O n = n.
2 Proof.
3   intros. simpl. trivial.
4 Qed.
```

Coq での定理や補題は，**Theorem** や **Lemma** コマンドの後に名前を書き（上の例では plus_O_n'），: の後ろに命題を書く．そして，続く **Proof** コマンドから，Coq との対話を通して命題の証明を進め，証明が完了したら **Qed** コマンドで証明を閉じる．この際，Coq は証明すべき命題をゴールとして提示してくるので，ユーザは，そのゴールをより自明な命題に還元するようにタクティックを入力していく．上記の例では，introsと simpl のタクティックでゴールを自明な等式に還元し，その等式が自明であることを trivial タクティックで伝え証明を完了している．

3. 持続型例外処理機構の形式化

本節では，既存の Scheme の意味を持続型例外へ対応するよう拡張し，Scheme 全体を Coq 上に形式化する．

3.1 持続型例外へ対応する意味の拡張

第 2.2 節で導入した二つの構文に対する意味を Scheme に追加する．基本的なアイデアは，例外処理ハンドラを保持するスタックを計算状態内に構成し，その先頭を指し示す特別な変数 handler を環境 ρ に保持するようにすることである．例外処理ハンドラはクロージャとして保持されるため，スタックの構成にはクロージャのアドレス部 α に次のクロージャへのポインタを保持する形で行う．

まず，式 E に二つの構文を追加する：

$$E \quad ::= \quad \cdots \mid (\texttt{do-handle}\ L\ E) \mid (\texttt{hcall}\ E_0\ E_1\ \cdots)$$

そして，これらの構文の処理のための継続を追加する：

$$\begin{aligned} \kappa \quad ::= \quad & \cdots \mid \mathbf{hpush} : \langle\langle E, \ldots\rangle, \langle v, \ldots\rangle, \rho, \kappa\rangle \\ & \mid \mathbf{hcallA} : \langle\langle v, \ldots\rangle, \kappa\rangle \end{aligned}$$

これらの追加に伴う簡約規則・継続規則の追加と修正を図 4 に示す．構文 do-handle では，変数 handler がハンドラのクロージャを指すように更新される．その際，クロージャのアドレス部には，既存のハンドラがあったならばそのアドレスを，そうでなければ終端を表す値を設定する．構文 hcall の規則は，関数適用のルールと同様に，引数の評価を行う継続を積んで引数の評価を進め，全ての引数が値になったところでハンドラ本体の式の実行を行う．このほか，**call** の継続規則に対して，handler にハンドラが設定されている場合にそれを引き継ぐように修正を加える．

3.2 Coq 上での形式化

前節で拡張した Scheme を Coq 上で形式化する．形式化の基本方針は，文献 [4] に倣う．また，Coq の標準ライブラリの他に，文献 [5] のモジュール Maps を利用した．

3.2.1 構文と計算状態の形式化

まず，Scheme プログラムの構文を全て帰納的データ型として定義する．**Location** の型 loc はアドレスを nat で保持し，**Identifier** の型 idtf は名前を string で保持する．また，定数の型 tmc は，ブール値を構成子で，整数値を nat で保持する：

```
1 Inductive loc : Type := loc_ : nat →loc.
2 Inductive idtf : Type := idtf_ : string→idtf.
3 Inductive tmc : Type :=
4   | tmc_true : tmc
5   | tmc_false : tmc
6   | tmc_num : nat →tmc.
```

ラムダ式の型 lmbd と式の型 tme は互いに依存するため，同時に定義する．型 tme の各構成子は，式 E の構文にそのまま対応する．特に，tme_dh が do-handle 構文に，tme_hcl が hcall 構文に対応する．

```
1 Inductive lmbd : Type :=
2   lmbd_ : list idtf →tme →lmbd
3 with tme : Type :=
4   | tme_con : tmc →tme
5   | tme_var : idtf →tme
6   | tme_lam : lmbd →tme
7   | tme_if : tme →tme →tme →tme
8   | tme_set : idtf →tme →tme
9   | tme_app : list tme →tme
10  | tme_dh : lmbd →tme →tme
11  | tme_hcl : list tme →tme.
```

簡約規則：

$$\langle \texttt{(do-handle } L\ E\texttt{)}, \rho, \kappa, \sigma\rangle \rightarrow \langle E, \rho[\text{handler} \mapsto \alpha], \kappa, \sigma[\alpha \mapsto \text{CLOSURE} : \langle \alpha_0, L, \rho\rangle][\alpha_0 \mapsto \bot]\rangle \quad \text{where handler} \notin \rho$$

$$\langle \texttt{(do-handle } L\ E\texttt{)}, \rho[\text{handler} \mapsto \alpha], \kappa, \sigma\rangle \rightarrow \langle E, \rho[\text{handler} \mapsto \alpha'], \kappa, \sigma[\alpha' \mapsto \text{CLOSURE} : \langle \alpha, L, \rho\rangle]\rangle$$

$$\langle \texttt{(hcall } E_0\ E_1\ \cdots\texttt{)}, \rho[\text{handler} \mapsto \alpha], \kappa, \sigma\rangle \rightarrow \langle E'_0, \rho[\text{handler} \mapsto \alpha], \mathbf{hpush} : \langle \langle E'_1, \ldots\rangle, \langle\rangle, \rho, \kappa\rangle, \sigma\rangle \quad \text{where } \langle E'_0, E'_1, \ldots\rangle = \text{reverse}(\langle E_0, E_1, \ldots\rangle)$$

継続規則：

$$\langle v_0, \rho', \mathbf{hpush} : \langle \langle E'_1, E'_2, \ldots\rangle, \langle v_1, \ldots\rangle, \rho, \kappa\rangle, \sigma\rangle \rightarrow \langle E'_1, \rho, \mathbf{hpush} : \langle \langle E'_2, \ldots\rangle, \langle v_0, v_1, \ldots\rangle, \rho, \kappa\rangle, \sigma\rangle$$

$$\langle v_0, \rho', \mathbf{hpush} : \langle\langle\rangle, \langle v_1, \ldots\rangle, \rho, \kappa\rangle, \sigma\rangle \rightarrow \langle \text{UNSPECIFIED}, \rho, \mathbf{hcallA} : \langle \langle v_0, v_1, \ldots\rangle, \kappa\rangle, \sigma\rangle$$

$$\langle \text{UNSPECIFIED}, \rho'[\text{handler} \mapsto \alpha], \mathbf{hcallA} : \langle\langle v_0, \ldots, v_n\rangle, \kappa\rangle, \sigma[\alpha \mapsto \text{CLOSURE} : \langle \alpha', L, \rho\rangle]\rangle \rightarrow \langle E, \rho''[\text{handler} \mapsto \alpha'], \kappa, \sigma'\rangle$$
$$\text{where } L = \texttt{(lambda (}I_0\ \cdots\ I_n\texttt{) } E\texttt{)},\ \beta_0, \ldots, \beta_n \notin E, \rho, \kappa, \sigma,\ \rho'' = \rho[I_0, \ldots, I_n \mapsto \beta_0, \ldots, \beta_n],\ \sigma' = \sigma[\beta_0, \ldots, \beta_n \mapsto v_0, \ldots, v_n]$$

$$\langle \text{CLOSURE} : \langle \alpha, L, \rho\rangle, \rho'[\text{handler} \mapsto \alpha'], \mathbf{call} : \langle\langle v_1, \ldots, v_n\rangle, \kappa\rangle, \sigma\rangle \rightarrow \langle E, \rho''[\text{handler} \mapsto \alpha'], \kappa, \sigma'\rangle$$
$$\text{where } L = \texttt{(lambda (}I_1\ \cdots\texttt{) } E\texttt{)},\ \beta_1, \ldots, \beta_n \notin E, \rho, \kappa, \sigma,\ \rho'' = \rho[I_1, \ldots, I_n \mapsto \beta_1, \ldots, \beta_n],\ \sigma' = \sigma[\beta_1, \ldots, \beta_n \mapsto v_1, \ldots, v_n]$$

$$\langle \text{CLOSURE} : \langle \alpha, L, \rho\rangle, \rho', \mathbf{call} : \langle\langle v_1, \ldots, v_n\rangle, \kappa\rangle, \sigma\rangle \rightarrow \langle E, \rho'', \kappa, \sigma'\rangle \text{ where } L = \texttt{(lambda (}I_1\ \cdots\texttt{) } E\texttt{)},\ \beta_1, \ldots, \beta_n \notin E, \rho, \kappa, \sigma,\ \text{handler} \notin \rho,\ \rho'' = \rho[I_1, \ldots, I_n \mapsto \beta_1, \ldots, \beta_n],\ \sigma' = \sigma[\beta_1, \ldots, \beta_n \mapsto v_1, \ldots, v_n]$$

図 4 Scheme の意味の拡張（記号 $\notin$ は，左のものが右のものの中に現れないことを意味する）

以上により，Scheme プログラムが Coq 上に形式化された．

次に，Scheme プログラムの計算状態を形式化する．識別子からアドレスへのマッピング ρ の型 itol は，識別子とアドレスのペアのリストとして定義する：

```
1 Definition itol : Type := list (idtf * loc).
```

値の型 value と継続の型 cntn は互いに依存するため，同時に定義する．それぞれ，前節に示した定式化を素直に記述したものである．

```
1  Inductive value : Type :=
2    | v_con : tmc →value
3    | v_unspe : value
4    | v_undef : value
5    | v_primop : primop →value
6    | v_closure : loc →lmbd →itol →value
7  with cntn : Type :=
8    | cntn_halt : cntn
9    | cntn_select : tme →tme →itol
10                   →cntn →cntn
11   | cntn_assign : idtf →itol →cntn →cntn
12   | cntn_push : list tme →list value →itol
13                   →cntn →cntn
14   | cntn_call : list value →cntn →cntn
15   | cntn_hpush : list tme →list value →itol
16                   →cntn →cntn
17   | cntn_hcall : list value →cntn →cntn.
```

アドレスから値へのマッピング σ の型 ltov も，itol と同様に，ペアのリストとして定義する：

```
1 Definition ltov : Type := list (loc * value).
```

以上の準備のもと，計算状態の型 cnfg を次のように定義する．終了状態を cnfg_fin で，第一用要素が式の中間状態を cnfg_procE で，第一要素が値の中間状態を cnfg_procv で表現する：

```
1 Inductive cnfg : Type :=
2 | cnfg_fin : value →ltov →cnfg
3 | cnfg_procE : tme →itol →cntn →ltov →cnfg
4 | cnfg_procv : value →itol →cntn →ltov →
      cnfg.
```

3.2.2 意味の形式化

簡約規則と継続規則は，まとめてひとつの命題 reduc として定義し，各構成子がひとつひとつの規則に対応するようにする．簡便のため，状態 t から状態 t' への遷移を t ⟶t' と記述できるように記法を導入する．まずは定義の全体像を示し，以下で各構成子を導入する：

```
1 Reserved Notation "t␣'⟶'␣t'" (at level 40).
2 Inductive reduc : cnfg →cnfg →Prop :=
3   (* ここに各規則を構成子として記述*)
4   where "t␣'⟶'␣t'" := (reduc t t').
```

定数に関する簡約規則に対応する構成子 reduc_c は次のように定義される．この規則では c や ρ について特に制約がないため，この構成子はこれらを $\forall$ で受け取る：

```
reduc_c : ∀c rho kappa sigma,
  cnfg_procE (tme_con c) rho kappa sigma
   ⟶cnfg_procv (v_con c) rho kappa sigma
```

識別子に関する簡約規則に対応する構成子 reduc_I は次のようになる．この規則は「識別子 I が既に定義されたものである」などの条件をもつため，それらを仮定として受け取るように構成子を定義する．ここで，find_ は，与えられた等価性判定関数でマッピング内の検索を行う関数である（定義は自明なので割愛）．

```
reduc_I : ∀I rho kappa sigma x y
  (Hx : find_ eqb_idtf I rho = Some x)
  (Hy : find_ eqb_loc x sigma = Some y)
  (Hynot : y <> v_undef),
  cnfg_procE (tme_var I) rho kappa sigma
      ⟶cnfg_procv y rho kappa sigma
```

ラムダ式，条件分岐，代入，関数適用についても同様に構成子を用意する．これらは特に特徴もないため，紙面の都合で割愛する．

持続型例外処理機構に関連する構成子として，既存のハンドラが handler に設定されている場合の do-handle に関する構成子を以下に示す：

```
reduc_dh2 : ∀L E rho kappa sigma phi phi'
 (H : In (pair (idtf_ "handler") phi) rho),
  cnfg_procE (tme_dh L E) rho kappa sigma
  ⟶
  cnfg_procE E
   (update_itol rho (idtf_ "handler") phi')
   kappa
   (update_ltov sigma phi' (v_closure phi L
       rho))
```

遷移後の状態では，update_itol および update_ltov を用いて，簡約規則にあった ρ と σ の更新を行っている．これらの関数の定義は自明であるため割愛する．既存のハンドラが設定されていない場合も同様である．

同様に，hcall に関する構成子は次のようにした：

```
reduc_hcall : ∀Es rho kappa sigma x y E0' Es'
  (Hx : find_ eqb_idtf (idtf_ "handler") rho
      = Some x)
  (Hy : find_ eqb_loc x sigma = Some y)
  (HE : E0' :: Es' = rev Es),
  cnfg_procE (tme_hcl Es) rho kappa sigma
      ⟶
  cnfg_procE E0' rho (cntn_hpush Es' nil rho
      kappa) sigma
```

これも，対応する規則を素直に書き下したものである．

次に，継続規則を導入する．基本的に簡約規則と同様であり，最も単純なものは次の **halt** に関するものである：

```
conti_halt1 : ∀v sigma,
   cnfg_procv v nil cntn_halt sigma
     ⟶cnfg_fin v sigma
```

他の継続規則も同様であり，紙面の都合もあるため，大部分を割愛する．持続型例外処理機構に関連する継続規則の構成子として，**hpush** の規則に対応する構成子二つを以下に示す．いずれの規則も，対応する規則を素直に書き下したものである．

```
| conti_hpush1 :
    ∀rho' v0' E1' Es' vs' rho kappa sigma,
    cnfg_procv v0' rho' (cntn_hpush (E1' :: Es
        ') vs' rho kappa) sigma
        ⟶
    cnfg_procE E1' rho (cntn_hpush Es' (v0' ::
         vs') rho kappa) sigma
| conti_hpush2 :
    ∀v0 rho' vs rho kappa sigma,
    cnfg_procv v0 rho' (cntn_hpush nil vs rho
        kappa) sigma
        ⟶
    cnfg_procv v_unspe rho (cntn_hcall (v0 ::
        vs) kappa) sigma
```

最後に，**hcallA** に関する継続規則の構成子を示す．この規則は条件が多く，最も複雑な規則のひとつである．多くの仮定を構成子が受け取る形となるが，ここまでの構成子と同じ方針で定義されたものである：

```
conti_hcall :
  ∀vs rho rho' kappa sigma
  E Is L alpha alpha' betas
  (Hx : find_ eqb_idtf (idtf_ "handler") rho'
       = Some alpha)
  (Hy : find_ eqb_loc alpha sigma = Some (
      v_closure alpha' L rho))
  (HL : L = (lmbd_ Is E))
  (HNotInRho : multi_In_snd eqb_loc betas rho
       = false)
  (HNotInSigma : multi_In_ eqb_loc betas
      sigma = false)
  (HNotInKappa : multi_InKappa betas kappa =
      false),
  cnfg_procv v_unspe rho' (cntn_hcall vs
      kappa) sigma
      ⟶
  cnfg_procE E
    (update_itol (multi_update_itol rho Is
       betas) (idtf_ "handler") alpha')
    kappa
    (multi_update_ltov sigma betas vs)
```

ここで，multi_In で始まる関数は，与えられた識別子のリストとマッピングについて，何れかの識別子がマッピングに

現れるか否かを判定する関数である．また，multi_update で始まる関数は，マッピングの更新を複数のエントリについて行う関数である．これらの定義は自明なため割愛する．

4. 評価

前節で導入した形式化を用いて，簡単なプログラムについての性質が証明できることを確認した．

4.1 持続型例外処理の動作

まず，持続型例外処理機構が想定通りに動くことを，以下の簡単なプログラムで確認した．このプログラムは，「変数 a に引数の値を代入する」というハンドラを登録し，引数に 2 を指定した持続型例外を発生させる．よって，持続型例外処理機構が正しく動作しているならば，このプログラムの実行後に，変数 a の値が 2 になるはずである．実行前に a の値を 2 以外にしておけば，機構の動作を確認できる．

```
(do-handle (lambda x (set! a x)) (hcall 2))
```

上記の動作の確認は，上記のプログラムを E として，以下を形式的に証明できればよい．ここで，$\rightarrow^*$ は，複数回の遷移を意味する．

$$\langle E, [\mathtt{a} \mapsto 0], \mathbf{halt}, [0 \mapsto 1]\rangle \rightarrow^* \langle \text{UNSPECIFIED}, [\mathtt{a} \mapsto 0, \ldots], \mathbf{halt}, [0 \mapsto 2, \ldots]\rangle$$

これを，本研究の形式化を用いて Coq 上に書き下したものが以下のコードである．ここで，t ⟶* t' は，t から t' へと複数の規則の適用で遷移できることの記法である．

```
Example dohandle_hcall_ex1 :
 cnfg_procE
  (tme_dh
   (lmbd_ ((idtf_ "x") :: nil)
    (tme_set (idtf_ "a")(tme_var(idtf_ "x"))))
   (tme_hcl (tme_con (tmc_num 2) :: nil)))
   ((idtf_ "a", loc_ 0) :: nil)
    cntn_halt
    ((loc_ 0, v_con (tmc_num 1)) :: nil)
 ⟶*
 cnfg_procv
  v_unspe
  ((idtf_ "a",loc_ 0)::(idtf_ "x",loc_ 3)::
    (idtf_ "handler",loc_ 2)::nil)
    cntn_halt
    ((loc_ 0, v_con (tmc_num 2)) ::
     (loc_ 1, v_closure (loc_ 2)
        (lmbd_ (idtf_ "x" :: nil) (tme_set (
            idtf_ "a") (tme_var (idtf_ "x"))))
        ((idtf_ "a", loc_ 0) :: nil)) ::
     (loc_ 2, v_unspe) ::
     (loc_ 3, v_con (tmc_num 2)) :: nil).
Proof.
  eapply multi_step. apply (reduc_dh1 _ _ _ _
       _ (loc_ 1) (loc_ 2)). trivial. simpl.
  eapply multi_step. eapply reduc_hcall.
      exists. exists. exists.
  eapply multi_step. apply reduc_c.
  eapply multi_step. apply conti_hpush2.
  eapply multi_step. eapply (conti_hcall _ _
      _ _ _ _ _ _ _ _ (loc_ 3 :: nil)).
    exists. exists. exists. trivial. trivial.
       trivial.
  eapply multi_step. apply reduc_set. simpl.
  eapply multi_step. eapply reduc_I. exists.
      simpl. eexists. discriminate.
  eapply multi_step. apply conti_assign.
      exists. simpl.
  apply multi_refl.
Qed.
```

実際，これは証明が完了する．すなわち，想定通りに a の値が変更されることが証明された．

次に，$E' =$ (do-handle (lambda x (set! a x)) 2) についての動作を証明した．これは，プログラム E の (hcall 2) の部分を 2 として例外の発生を行わないようにしたものである．プログラム E とは異なり，プログラム E' の実行では変数 a の値は書き換わらないはずである．すなわち，以下が成り立つはずである．

$$\langle E', [\mathtt{a} \mapsto 0], \mathbf{halt}, [0 \mapsto 1]\rangle \rightarrow^* \langle 2, [\mathtt{a} \mapsto 0, \ldots], \mathbf{halt}, [0 \mapsto 1, \ldots]\rangle$$

これを Coq 上に書き下し，証明も完了した：

```
Example dohandle_hcall_ex2 :
 cnfg_procE
  (tme_dh
   (lmbd_ ((idtf_ "x") :: nil)
    (tme_set(idtf_ "a")(tme_var(idtf_ "x"))))
     (tme_con (tmc_num 2)))
    ((idtf_ "a", loc_ 0) :: nil)
    cntn_halt
    ((loc_ 0, v_con (tmc_num 1)) :: nil)
 ⟶*
 cnfg_procv
  (v_con (tmc_num 2))
  ((idtf_ "a", loc_ 0) ::
    (idtf_ "handler", loc_ 1) :: nil)
  cntn_halt
  ((loc_ 0, v_con (tmc_num 1)) ::
   (loc_ 1, v_closure (loc_ 2)
      (lmbd_ (idtf_ "x" :: nil) (tme_set (
          idtf_ "a") (tme_var (idtf_ "x"))))
      ((idtf_ "a", loc_ 0) :: nil)) ::
   (loc_ 2, v_unspe) :: nil).
Proof.
  eapply multi_step. apply (reduc_dh1 _ _ _ _
       _ (loc_ 1) (loc_ 2)). trivial. simpl.
```

```
23   eapply multi_step. apply reduc_c.
24   apply multi_refl.
25 Qed.
```

これにより，最初の例で変数 a の値が変更されたのが，持続型例外処理機構の作用であることが確認された．

以上により，提案する形式化により想定通りの動作をする持続型例外処理機構が表現されていることを確認した．

4.2 再帰関数の動作

もう少し複雑なプログラムに関する検証として，階乗を計算する再帰関数 fact に関する以下を証明した：

$$\langle (\texttt{fact 3}), \rho, \mathbf{halt}, \sigma \rangle \to^* \langle 6, \sigma' \rangle$$

ここで，fact の中身は以下のラムダ式である：

```
(lambda n (if (= n 0) 1 (* n (fact (- n 1)))))
```

上記の命題を Coq で書き下したものが以下である．ここで，fact_lambda は上記のラムダ式を提案する形式化の構成子で記述し，別に定義したものである（定義は割愛）．

```
1  Example factorial_ex1 :
2   cnfg_procE
3    (tme_app (
4     (tme_var (idtf_ "fact")) ::
5     (tme_con (tmc_num 3)) :: nil))
6    ((idtf_ "fact", loc_ 0) :: nil)
7    (cntn_halt)
8    ((loc_ 0,
9      (v_closure (loc_ 0) fact_lambda
10      ((idtf_ "fact", loc_ 0) :: nil))) :: nil)
11   ⟶*
12   cnfg_fin
13    (v_con (tmc_num 6))
14    ((loc_ 0,
15       v_closure (loc_ 0) fact_lambda
16        ((idtf_ "fact", loc_ 0) :: nil)) ::
17     (loc_ 1, v_con (tmc_num 3)) ::
18     (loc_ 2, v_con (tmc_num 2)) ::
19     (loc_ 3, v_con (tmc_num 1)) ::
20     (loc_ 4, v_con (tmc_num 0)) :: nil).
21 Proof. (* 長いため割愛. *) Qed.
```

証明を完了することができ，提案の形式化により再帰関数の動作も正常に表現されていることが確認された．

なお，上記の例の証明は，非常に長いものとなった．これは，現状の形式化においては，ある計算状態から別の計算状態に遷移することの証明が，その遷移を実際に行うための構成子の指定によって行われる形になるからである．すなわち，証明が「プログラムの手動実行」になってしまうからである．

5. おわりに

本論文では，持続型例外処理機構を定理証明支援系 Coq で形式化した．具体的には，Scheme の既存の定式化 [2] を do-handle と hcall の追加により拡張し，その構文と計算状態を帰納的データ型として，その簡約規則と継続規則を帰納的命題の構成子として Coq に形式化した．また，構築した形式化を用いて，簡単なプログラムの動作について証明が行えることを確認した．

今後の課題として以下が考えられる．まず，持続型例外処理機構へのレベルの導入 [6] が挙げられる．本論文の形式化は，同機構の形式化の第一歩として，単一レベルの機構を対象とした．しかし，ごみ集めと一級継続を同時に実装するような場面にはレベルが必要となるため，レベル付きの形式化が望まれる．次に，プログラムに関する証明に使う自動タクティックの開発が挙げられる．第 4 節で述べたように，現状の証明は「プログラムの実行を手動でおこなう」ということを要求する．プログラムの実行は機械的に出来るはずであるから，それを自動化するタクティックの提供が望ましい．最後に，「適用可能な遷移規則の一意性」の形式的証明が挙げられる．遷移規則の設計から，一意性が存在していると考えられるものの，形式的な証明はまだない．一意性がない場合にはプログラム実行が非決定的になってしまうため，一意性の形式的証明が望まれる．

謝辞 本研究は JSPS 科研費 JP19K11903, JP19H04087 の助成を受けたものです．

参考文献

[1] Bertot, Y. and Castéran, P.: *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*, Springer (2004).
[2] Clinger, W. D.: Proper Tail Recursion and Space Efficiency, *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pp. 174–185 (1998).
[3] Kokke, W., Siek, J. G. and Wadler, P.: Programming Language Foundations in Agda, *Science of Computer Programming*, Vol. 194, p. 102440 (2020).
[4] Pierce, B. C., de Amorim, A. A., Casinghino, C., Gaboardi, M., Greenberg, M., Hritcu, C., Sjöberg, V., Tolmach, A. and Yorgey, B.: *Programming Language Foundations*, Software Foundations, Vol. 2, Electronic textbook, August 2021. Version 6.1. https://softwarefoundations.cis.upenn.edu/plf-current/ (2021).
[5] Pierce, B. C., de Amorim, A. A., Casinghino, C., Gaboardi, M., Greenberg, M., Hritcu, C., Sjöberg, V. and Yorgey, B.: *Logical Foundations*, Software Foundations, Vol. 1, Electronic textbook, August 2021. Version 6.1. https://softwarefoundations.cis.upenn.edu/lf-current/ (2021).
[6] 八杉昌宏，江本健斗，平石 拓：レベル付き持続型例外処理機構の設計，日本ソフトウェア科学会第 38 回大会講演論文集 (2021).