レベル付き持続型例外処理機構の設計

八杉 昌宏　江本 健斗　平石 拓

安全な計算状態操作機構は，C 言語などの実行スタック中に眠る呼び出し元の変数の値への合法的アクセスを提供するもので，高信頼・高性能な高水準言語のための，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水準サービスの効率的実装を可能とする．入れ子関数の形態をとる場合，クロージャを生成し，それへのポインタを用いて間接的に入れ子関数を呼び出すことで，クロージャ生成時の環境に含まれる変数へのアクセスが可能となる．一方，新しい形態として設計中の持続型例外処理機構は，非局所脱出することなく例外ハンドラを呼び出せるとしたもので，ポインタではなく動的スコープに基づく．また，一級継続生成中のごみ集めといった複数レベルでの計算状態操作機構の利用も想定する．本発表では，その性質の理解や検証を可能とするため，レベル付き持続型例外処理機構を備えたコア言語の仕様を整理する．

1　はじめに

逐次命令型言語（手続き型言語）に，**計算状態操作機構** [8] [4] [3] [9] [5] を備えさせる研究を行っている．計算状態操作機構とは，存命の変数の値へ（手順を踏むことで）いつでも合法的にアクセスするための機構で，（デバッガなどではなく）計算主体が自分自身で利用する．例えば，C 言語などの実行スタック中に眠る呼び出し元の変数の値への合法的アクセスを提供する．これより，高信頼・高性能な高水準言語のための，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水準サービスの効率的実装を可能とする．

これまで計算状態操作機構は入れ子関数の形態で提供してきた [8] [4] [3] [9] [5]．入れ子関数の形態をとる場合，クロージャを生成し，それへのポインタを用いて間接的に入れ子関数を呼び出すことで，クロージャ生成時の環境に含まれる変数へのアクセスが可能となる．例えば，

```
Pair copy_tree(void (*scan0)(), Pair x) {
  if (IS_NIL(x)) return x;
  Pair a = NIL, d = NIL;
  //GC に備え，クロージャ scan1 を生成
  void scan1() {
    scan0(); //scan0 で呼び出し元のルートをスキャン
    x = move(x); a = move(a); d = move(d);
  }
  Pair a = copy_tree(scan1, CAR(x)); //scan1 を渡す
  Pair d = copy_tree(scan1, CDR(x));
  return CONS(scan0, a, d);
}
```

のように，より古くに生成され仮引数（scan0）に渡されたクロージャを，呼び出し先でより最近生成されたクロージャ（scan1）が連鎖的に呼び出すことで，実行スタック中に眠る任意段数前の（この場合，再帰）呼び出し元の変数（x，a，d）へアクセスできる．この例では，コピー型ごみ集めにおけるルートとして，実行スタック全体の存命の変数へ全面的にアクセスできる．

一方，新しい形態として設計中の持続型例外処理機構は，非局所脱出することなく例外ハンドラを呼び出せるとしたもので，ポインタではなく動的スコープに

---

* Designing Mechanisms for Leveled Restartable Exception Handling.
This is an unrefereed paper. Copyrights belong to the Author(s).

Masahiro Yasugi, 九州工業大学, Kyushu Institute of Technology.
Kento Emoto, 九州工業大学, Kyushu Institute of Technology.
Tasuku Hiraishi, 京都橘大学, Kyoto Tachibana University.

基づくため，より安全となる．上の入れ子関数の例では仮引数（scan0）として受け取ったクロージャへのポインタを，クロージャを所有する関数（copy_tree）の終了に伴いクロージャや変数（x，a，d）の寿命が尽きた後に利用しようとしてしまう可能性がある．ポインタは「いつでも」だけでなく,「いつまでも」（不正）利用できてしまう．延命により安全性を高めるには自動的に「ごみ集めされる」ヒープに対象を確保または移動することが考えられるが，計算状態操作機構としては，高水準言語のための「ごみ集め」の効率的実装は，基盤となる低水準言語として提供したいことの 1 つであるため，提供を受ける側になってしまっては目的に逆行する．

入れ子関数の形態のままで，一定の規律に従うことで存命でない対象へのアクセスが発生しないことを保証する型システムも考えられるが，本研究では本質的に安全となる言語仕様に基づくアプローチをとる．動的スコープにより例外ハンドラが呼び出せるのは，例外ハンドラが存命の間に限られ，例外ハンドラによる変数アクセスも変数が存命の間に限られる．実行スタック全体の変数にアクセスするための連鎖は例外ハンドラから例外ハンドラを呼び出すことで可能となる．これは非局所脱出の場合の例外の再 throw に類似している．

一方，入れ子関数の形態では，同じクロージャを（一級継続生成とごみ集めのルートスキャンのように）別の用途に用いられたが，持続型例外処理機構ではハンドラの同じ実体が多重実行されることはない．よって，一級継続生成中のごみ集めといった複数レベルでの計算状態操作機構の利用も想定する．例えば，一級継続生成のためのハンドラの実行中に,（そのハンドラ（だけ）が利用する局所変数を対象とした）ルートスキャンのためのハンドラの別レベル（より基礎的なレベル）での実行が必要といった場合である．これはレベル付けされた（階層化された）限定継続に類似している．

本発表では，その性質の理解や検証を可能とするため，レベル付き持続型例外処理機構を備えたコア言語の仕様を整理する．まず，伝統的な逐次命令型言語（手続き型言語）のコア言語の仕様を整理し，次に持続型例外処理機構を追加したコア言語を整理する．ここでのコア言語（core language）とは低レベルで実装用あるいは中間言語としてできるだけ最小の言語である．つまり，後の拡張の元とすることができる言語である．また，内部と外部を分離したときに，内部ができるだけ安全に保たれているもの †1 とする．

コア言語としては SC 言語処理系 [10] の利用を想定して，S 式ベースの C 言語のサブセットとする．SC 言語処理系は Common Lisp をベースとした処理系であり，SC 言語（S 式ベースの構文を持つ C 言語の総称）間の翻訳を，指定された変形規則セットによる S 式の変形により処理できる．

本研究の貢献は以下の 2 点である．

- 逐次命令型言語であるコア SC 言語の仕様
- コア SC 言語に持続型例外処理機構を追加したコア SC-RE 言語の仕様，ならびに静的リンクを明示化したコア SC-REs 言語の仕様

本稿の構成は以下の通りである．2 章では既存手法を大局的観点から俯瞰する．3 章では計算状態操作機構のこれまでと現在，設計中の持続型例外処理機構について述べる．4 章では，S 式ベースの構文を持つ逐次命令型言語であるコア SC 言語の仕様を示す．5 章では，持続型例外処理機構を備えたコア SC-RE 言語（静的リンク非明示で元の文脈においてハンドラ起動）ならびにコア SC-REs 言語（静的リンクを明示化して隣接する呼び出しは隣接する配置にしたもの）の設計（構文，意味論など）を示す．また，意味論に基づく展望（今後の課題）を述べる．6 章でまとめと今後の課題について述べる．

2 既存手法

2.1 宣言型言語

プログラミング言語は，宣言型言語と命令型言語に大別される．宣言型言語はどれも高水準言語であり，論理型言語と関数型言語に大別される．副作用のない純粋関数型言語では参照透過性などの利点があるが，実用的な評価戦略として non-strict な遅延評価を用

†1 いわゆるマルチコア・プロセッサのコアとは異なるが，内部にレジスタを持つ CPU コアが，外部のメモリなどを扱うという点では共通点もある．

いるなど，高水準言語の実装は複雑となる．

2.2 低水準逐次命令型言語と実行スタック

通常,「コンピュータ」は，機械語プログラムの逐次実行が可能な機械として順序回路をベースとして作られている．機械語プログラムは低水準（低レベル／低級）の逐次命令型言語とみなすことができるが，低水準ではあるもの，ある程度うまく抽象化された（多くの部分では）機械独立な命令型言語が一般には用いられている．機械独立な逐次命令型言語は，コンパイラやインタプリンタといった言語処理系により，様々な命令セットアーキテクチャの「コンピュータ」で実行可能となる．機械語の分岐命令の代わりに機械独立な逐次命令型言語では if 文や while 文が用いられる他，多くの場合，手続き呼び出しが利用可能である．ここで，命令型言語であっても「関数」が手続きの意味で用いられることも多い．関数や手続きは仮引数（あるいはパラメータ）を用いて定義され，呼び出し（call）時には実引数が通常，値として呼び出し元から呼び出し先に渡される．また，呼び出し先で求めた結果が呼び出し元に返（return）される．手続き（procedure）や関数（function）は，コンパイラを用いた場合は機械語命令列に翻訳され，その呼び出し（call）とリターン（return）のためには，命令セットアーキテクチャに適した呼び出し慣例が用いられている．

呼び出し（call）とリターン（return）は，データ構造のスタックにおける push と pop のようにペアになっていて，呼び出された手続き A からさらに手続き B を呼び出すと，A からのリターンは B からのリターンの後になるなど，call と return には自然な順序がつく．呼び出しが完了すれば求められた結果としての状態更新を除き基本的には，状態が元に戻されるため，呼び出し元で想定すべき状態変化は限定的で想定しやすくなる.「実行スタック」をうまく利用して call と return できる．call 時には実行スタックを介して引数を渡したり，戻り番地（リターンアドレス）を渡したり，実行スタック上に局所変数を確保しでき，return 時には実行スタックの形状は復元される．呼び出しが入れ子になる場合もその分だけ「実行スタック」が伸び縮みする．

実行スタックを利用する際，実際には一部のデータを callee save レジスタに置くこと（レジスタ割り当て）で効率化がはかれる.（ただし却って無駄になることもあり，callee save レジスタの lazy save，eager restore が望ましいが，普及した手法とはいいがたい.）SPARC のような命令セットアーキテクチャではレジスタ・ウインドウも利用できる．また，実行スタックへのアクセスは参照局所性が高く，CPU が用いるキャッシュとも相性が良い．

call と return に基づく場合，return 前に呼び出し元に眠る変数の値へアクセスすることは正式にはできない．特に，レジスタ割り当てを考慮すると，実行スタックを単なるメモリとしてメモリアドレスやポインタを用いて正しくアクセスすることは困難で危険といえる．一般には，ソースレベルデバッガは，この困難に頑張って対処するが，デバッグオプションを有効にしてデバッグ対象をコンパイルする必要がある．

2.3 制御構造

遠隔手続き呼び出し（remote procedure call）は，呼び出し元と呼び出し先が離れていても手続き呼び出しを可能としたものである．また，オブジェクト指向言語でのメッセージパッシングに基づくメソッド呼び出しでは，呼び出し側からは起動されるメソッドを直接は定めず，メッセージのみ定める．呼び出し先では，メッセージに適したメソッドが選択され，起動される．

低水準逐次命令型言語で入れ子関数（入れ子手続き）や λ 式が提供される場合，前章で述べたように，低水準であるか安全であるかのどちらかを犠牲にする必要がある．入れ子関数定義からそれが置かれた環境を伴うクロージャを生成するには，安全のため，ごみ集めされるヒープに場所を確保したくなる．この場合，環境（変数から location への写像），ストア（location から値への写像）といったすっきりした理論的道具立てで意味論を定めたりもできる．一方，（機械語に近いものの，適度に機種独立な）低水準言語としての役割を果たし続けるにはこのアプローチをとることはできない．

最初から存在する大域環境で定義される大域関数のみでも，再帰的計算や分割統治，バックトラック探索といった有意義な計算は実現できる．バックトラック探索では命令型言語であることを活かして，単一の作業空間で副作用による状態変化をバックトラック時に元に戻す形で限られた作業空間を逐次的に利用した効率化も可能である．

末尾呼び出しなどで適切な末尾再帰（proper tail recursion）が求められる高水準言語を実装するには，低水準命令型言語で凝った工夫が必要となる，つまり，高水準言語の呼び出しを低水準命令型言語での呼び出しとなるようなコンパイル方式や，インタプリタ実装を開発するのは難しい．実行スタックには典型的には，引数，戻り番地が，この順で push されているため，末尾呼び出し先の関数の引数が元の関数より増えた場合は戻り番地の位置を変えないで末尾呼び出し先へ「ジャンプ」するのは面倒となる．

一級継続，限定継続，コルーチンなどは，高水準言語なら提供できる機能と考えられる．その意味論は継続，環境，ストアなどを用いてすっきりと表現できるものの，そのためにはごみ集めが必須になる．また，CPS 変換によっても実装できるが，変換の結果，末尾呼び出しばかりとなり，properly tail recursive な高水準言語処理系を要することとなる．

例外処理や catch & throw は C++言語にも採用されているように，低水準逐次命令型言語でも利用できる．例外の throw に対して，これを catch する例外ハンドラは動的スコープで決まるといえ，非局所脱出に伴い実行スタックを縮める形で実行スタックとの相性もよい．また，例外を再 throw することで，実行スタック全体にわたり例外ハンドラを動かすこともできる．ただし，Common Lisp のコンディションシステムを除くと，catch 後は例外ハンドラが設定されたところまでの限定継続が非局所脱出により捨てられ，その部分の再実行／再開が不可能となる．

比較的低レベルの視点から例外処理が必要な状況は，割り込みやシグナルハンドリングでも生じるが．catch & throw の catch 先に対して，割り込みハンドラやシグナルハンドラから例外を throw するのは危険と思われる．

2.4 並行性／並列実行主体

並行計算は，計算したいことを十分な並行性を持つ形で表現し，進められる計算は進めるといったものといえる．関数型言語で non-strict で eager な評価戦略を用いると，計算したいことはデータフロー計算の形に表現される．高水準言語として十分な並行性を表現した上で，限られたあるいは多すぎる並列実行主体（CPU 等などのワーカ）との対応関係はどうすべきか，各実行主体（ワーカ）はどのような低水準命令型言語プログラムの逐次実行を行えばよいかといった点が課題となる．

プロセス計算では，プロセスが互いに協調して計算を進める．協調にはチャネルやリンクを介してのつながりが利用され，新しいチャネルやリンクが追加されることもある．アクターや並行オブジェクト計算では，アクターや並行オブジェクトを宛先として非同期メッセージ送信に基づく．これらも実行主体（ワーカ）との対応関係をどうすべきか，どのような低水準命令型言語プログラムの逐次実行を行えばよいかといった点が課題となる．

共有メモリ環境を前提としたマルチスレッド計算ではデータレースフリーが課題となる．計算したいことが分割統治と fork/join（spawn/sync）で表現できるならワーカ間でのワークスティール実行に適しているといえる．

ワーカを主体に考えた場合，逐次実行する低水準命令型言語プログラムの実行効率の観点からいえば，できるたけ自分だけで逐次実行できることが望ましいといえる．そのためには，ワークスティールのためのタスク要求がない限り自分で計算，あるいは当初はすべて自分で逐次実行する計画で始め，他のワーカから結果が得られた部分の計算は省略するような並列実行方式が効率良いといえる．

3 計算状態操作機構

計算状態操作機構は，高性能・高信頼な高水準言語において，実行スタック全体の存命の変数に関する全面的操作が必要な高水準サービスの実装に適している．例えば，ごみ集めや一級継続や proper tail recursion の実装 [7] [5]，ワークスティールにおける

タスクの遅延生成の実装 [2]，計算省略のための遅延された結果出力の実装 [6] などである．

入れ子関数（から生成されるクロージャ）の形態の場合，GCC が提供する入れ子関数 [1] も利用できるが，これを改良するものとして，これまで，GCC 改造による L-closure [8] [4]，GCC 改造による closure [8] [4] [9]，標準 C への変換によるスタック再構築に基づく L-closure [3]，標準 C への変換によるフレーム毎復元に基づく L-closure [9]，標準 C への変換による closure [5] などの実装を行ってきた．

本研究では，計算状態操作機構の新しい形態として持続型例外処理機構を提案する．入れ子関数の形態の場合は，存命でない対象にアクセスしないように注意する必要があるが，持続型例外処理機構の場合，呼び出せるのは利用可能なハンドラだけでそのアクセス対象も存命のものとできる．

持続型例外処理機構を利用した継続キャプチャ中に持続型例外処理機構を利用した GC 開始，持続型例外処理機構を利用したワークスティール処理中に持続型例外処理機構を利用した GC 開始のように，複数のレベルで持続型例外処理機構を利用できるようにする．このような場合のプログラムの意味を明確にするため，本研究では言語仕様を整理することとした．これにより，計算状態操作機構の安全性，有効性を高めるほか，その性質の検証に取り組めるようにする．

次章で，ベースとなるコア言語として S 式ベースの C 言語のサブセット「コア SC 言語」について述べたのち，持続型例外処理機構を追加したコア言語「コア SC-RE 言語」を整理する．「コア SC-RE 言語」としては静的リンク非明示で元の文脈においてハンドラ起動としたものと，「コア SC-REs 言語」（静的リンクを明示化して隣接する呼び出しは隣接する配置にしたもの）の設計（構文，意味論など）を示す．

SC-RE 言語の翻訳/変換に基づく実装としては，標準 C への変換，入れ子関数を利用するコードへ翻訳，例外処理を利用したコードへの変換を予定している．

4 コア SC 言語

コア SC 言語の基本設計としては，スタックベースの逐次命令型言語であって，ごみ集めされるヒープは持たない，シンプルとするため値は整数値のみとし，ポインタはないとする．

また，仕様においてα同値などは用いない．インライン展開の形では呼び出しを実行しないため，変数名のつけかえはない．また，意味論においてフレッシュな location は用いない．

入出力を備える．入出力は，外部からの要求や通知のポーリング，データの送受信，（コア言語側でごみ集めをする）ヒープにもなり得る（そのアドレスとデータはコア言語では整数値となる）外部メモリの利用，といった使い道を想定するが，これら入出力先の仕様はコア言語の仕様に含まれない．これら外部に起因する非決定性を除き，コア言語のプログラムは決定的で逐次的に動作する．

意味論は small step とし，構文は，実行時のプログラムを表現可能とする．適切な型システムを利用することで型安全性が得られるようにする．

4.1 Syntax

コア SC 言語の構文を以下に示す．

$$
\begin{array}{rl}
num & ::= 0 \mid 1 \mid 2 \mid \ldots \\
dn & ::= num \\
v\ (\text{値}) & ::= num \\
e\ (\text{式}) & ::= v \mid id \mid \texttt{(+}\ e\ e\texttt{)} \mid \texttt{(>}\ e\ e\texttt{)} \\
vdef & ::= \texttt{(def}\ id\ \texttt{int}\ v\texttt{)} \\
s\ (\text{文}) & ::= \texttt{(=}\ id\ e\texttt{)} \\
 & \quad \mid \texttt{(=}\ id\ \texttt{(in}\ e\texttt{))} \mid \texttt{(}out\ e\ e\texttt{)} \\
 & \quad \mid \texttt{(if}\ e\ s\ s\texttt{)} \mid \texttt{(while}\ e\ s\texttt{)} \\
 & \quad \mid \texttt{(begin}\ vdef^*\ s^*\texttt{)} \\
 & \quad \mid \texttt{(=}\ id\ \texttt{(call}\ id\ e^*\texttt{))} \\
 & \quad \mid \texttt{(=}\ id\ \texttt{(calling}\ dn\texttt{))} \\
 & \quad \mid \texttt{(return}\ e\texttt{)} \\
gdef & ::= vdef \\
 & \quad \mid \texttt{(def (}id\ id^*\texttt{) (fn int int}^*\texttt{)}\ s\texttt{)} \\
exec & ::= \texttt{(exec}\ dn\ id\ vdef^*\ s\texttt{)} \\
prog & ::= \texttt{(prog}\ gdef^* \\
 & \qquad exec^*\ \texttt{(exec 0 main}\ vdef^*\ s\texttt{))}
\end{array}
$$

dnは動的リンク番号で戻り番地に相当し，(begin) が何もしない文（典型的には skip 等）に相当する．$gdef^*$ で定義される大域変数や大域関数の idは distinct とする．$exec$は実行中の関数を表し，初期プログラムは main を除き，これを持たないとする．

$$
\begin{array}{rl}
init\text{-}prog & ::= \texttt{(prog}\ gdef^* \\
 & \qquad \texttt{(exec 0 main}\ vdef^*\ s\texttt{))}
\end{array}
$$

4.2 Semantics

コア SC 言語の意味論を small step semantics で示す.

4.2.1 Contexts and instructions

e を $ec[einsn]$ と分解するための文脈とインストラクションを示す.

$$
\begin{aligned}
ec ::= {} & [\,] \mid \texttt{(+}\ ec\ e\texttt{)} \mid \texttt{(+}\ v\ ec\texttt{)} \\
& \mid \texttt{(>}\ ec\ e\texttt{)} \mid \texttt{(>}\ v\ ec\texttt{)} \\
einsn ::= {} & id \mid \texttt{(+}\ v\ v\texttt{)} \mid \texttt{(>}\ v\ v\texttt{)}
\end{aligned}
$$

s を $ecs[einsn]$ と分解するための文脈を示す.

$$
\begin{aligned}
ecs ::= {} & \texttt{(=}\ id\ ec\texttt{)} \\
\mid {} & \texttt{(=}\ id\ \texttt{(in}\ ec\texttt{))} \\
\mid {} & \texttt{(out}\ ec\ e\texttt{)} \\
\mid {} & \texttt{(out}\ v\ ec\texttt{)} \\
\mid {} & \texttt{(if}\ ec\ s\ s\texttt{)} \\
\mid {} & \texttt{(=}\ id\ \texttt{(call}\ id\ v^*\ ec\ e^*\texttt{))} \\
\mid {} & \texttt{(return}\ ec\texttt{)}
\end{aligned}
$$

s を $sc[sinsn]$ と分解するための文脈とインストラクションを示す.

$$
\begin{aligned}
sc ::= {} & [\,] \\
\mid {} & \texttt{(begin}\ vdef^*\ sc\ s^*\texttt{)} \\
skip\text{-}or\text{-}return ::= {} & \texttt{(begin)} \mid \texttt{(return}\ e\texttt{)} \\
sinsn ::= {} & \texttt{(=}\ id\ e\texttt{)} \\
\mid {} & \texttt{(=}\ id\ \texttt{(in}\ e\texttt{))} \\
\mid {} & \texttt{(out}\ e\ e\texttt{)} \\
\mid {} & \texttt{(if}\ e\ s\ s\texttt{)} \\
\mid {} & \texttt{(while}\ e\ s\texttt{)} \\
\mid {} & \texttt{(begin}\ vdef^+\texttt{)} \\
\mid {} & \texttt{(begin}\ vdef^*\ skip\text{-}or\text{-}return\ s^*\texttt{)} \\
\mid {} & \texttt{(=}\ id\ \texttt{(call}\ id\ e^*\texttt{))}
\end{aligned}
$$

$prog$を $scprog[sinsn]$ と分解するための文脈を示す.

$$
\begin{aligned}
calling\text{-}s ::= {} & sc[\texttt{(=}\ id\ \texttt{(calling}\ dn\texttt{))}] \\
calling\text{-}exec ::= {} & \texttt{(exec}\ dn\ id\ vdef^*\ calling\text{-}s\texttt{)} \\
scexec ::= {} & \texttt{(exec}\ dn\ id\ vdef^*\ sc\texttt{)} \\
scprog ::= {} & \texttt{(prog}\ gdef^*\ scexec\ calling\text{-}exec^*\texttt{)}
\end{aligned}
$$

4.2.2 Configurations

動的リンク番号 dnまでを用いている $prog$に関する評価関係 $(prog_1, dn_1) \to (prog_2, dn_2)$ を与える.

$$
\frac{\begin{array}{c}\texttt{(def (}id_1\ id^*\texttt{) (fn int int}^*\texttt{) }s\texttt{)} \in gdef^* \\ vdef_2^* = \texttt{(def}\ id^*\ \texttt{int}^*\ v^*\texttt{)} \\ dn_2 = dn_1 + 1\end{array}}{\begin{array}{l}\texttt{((prog}\ gdef^* \\ \quad scexec[\texttt{(=}\ id_2\ \texttt{(call}\ id_1\ v^*\texttt{))}] \\ \quad calling\text{-}exec^*\texttt{)}, dn_1) \to \\ \texttt{((prog}\ gdef^* \\ \quad \texttt{(exec}\ dn_2\ id_1\ vdef_2^*\ s\texttt{)} \\ \quad scexec[\texttt{(=}\ id_2\ \texttt{(calling}\ dn_2\texttt{))}] \\ \quad calling\text{-}exec^*\texttt{)}, dn_2)\end{array}} \text{(CALL)}
$$

$$
\frac{dn_2 = dn_1 + 1}{\begin{array}{l}\texttt{((prog}\ gdef^* \\ \quad \texttt{(exec}\ dn_2\ id_1\ vdef_2^*\ \texttt{(return}\ v_3\texttt{))} \\ \quad scexec[\texttt{(=}\ id_2\ \texttt{(calling}\ dn_2\texttt{))}] \\ \quad calling\text{-}exec^*\texttt{)}, dn_2) \to \\ \texttt{((prog}\ gdef^* \\ \quad scexec[\texttt{(=}\ id_2\ v_3\texttt{)}] \\ \quad calling\text{-}exec^*\texttt{)}, dn_1)\end{array}} \text{(RETURN)}
$$

call された関数の本体は，呼び出し元にインライン展開されるのではなく，大域環境中で実行される．このため呼び出し元の変数名をつけかえたりする必要はない．これら call と return 以外では動的リンクの変更はなく，評価関係 $prog_1 \to prog_2$ に基づく.

$$
\frac{prog_1 \to prog_2}{(prog_1, dn) \to (prog_2, dn)} \quad \text{(PROG)}
$$

4.2.3 Expressions

$prog_1 = scprog[ecs[einsn]]$ のときの評価関係 $prog_1 \to prog_2$ を与える．プログラム全体として対象となる式が一意に定まる（つまり，逐次実行である）．idについては，きれいな環境やストアではなく，実行中のプログラムが文脈として持つ変数の値を用いる．これは文脈から $lookup$する．コア SC 言語の $lookup$は後で示す．id以外の式に関しては，標準的な操作的意味論といえる.

$$\frac{lookup(id, scprog) = v}{scprog[ecs[id]] \to scprog[ecs[v]]} \text{ (REFVALDEF)}$$

$$\frac{v_3 = v_1 + v_2}{scprog[ecs[(\texttt{+}\ v_1\ v_2)]] \to scprog[ecs[v_3]]} \text{ (PLUS)}$$

$$\frac{v_1 > v_2}{scprog[ecs[(\texttt{>}\ v_1\ v_2)]] \to scprog[ecs[1]]} \text{ (GT1)}$$

$$\frac{v_1 \not> v_2}{scprog[ecs[(\texttt{>}\ v_1\ v_2)]] \to scprog[ecs[0]]} \text{ (GT0)}$$

4.2.4 Statements

$prog_1 = scprog[sinsn]$ のときの評価関係 $prog_1 \to prog_2$ を与える．ただし，CALL については 4.2.2 節で $(prog_1, dn) \to (prog_2, dn)$ としてすでに与えた．

idへの値のアサインメントについては，すっきりとした環境やストアではなく，実行中のプログラムの文脈自体を $update$で更新する．

$$\frac{update(id, v, scprog_1) = scprog_2}{scprog_1[(\texttt{=}\ id\ v)] \to scprog_2[(\texttt{begin})]} \text{ (UPDATEVALDEF)}$$

コア SC 言語の $update$は後で示す．以下は入出力命令である．

$$\frac{v_1 \text{ is read from port } v}{scprog[(\texttt{=}\ id\ (\texttt{in}\ v))] \to scprog[(\texttt{=}\ id\ v_1)]} \text{ (IN)}$$

$$\frac{v_1 \text{ is written to port } v}{scprog[(\texttt{out}\ v\ v_1)] \to scprog[(\texttt{begin})]} \text{ (OUT)}$$

以下は if 文，while 文に関する標準的な遷移である．

$$scprog[(\texttt{if 0}\ s_1\ s_2)] \to scprog[s_2] \text{ (IF0)}$$

$$\frac{v \neq 0}{scprog[(\texttt{if}\ v\ s_1\ s_2)] \to scprog[s_1]} \text{ (IF1)}$$

$$scprog[(\texttt{while}\ e\ s)] \to scprog[(\texttt{if}\ e\ (\texttt{begin}\ s\ (\texttt{while}\ e\ s))\ (\texttt{begin}))] \text{ (WHILE)}$$

以下は実行を完了して次に進むためであるが，BEGINEND と BEGINRETURN については変数を削除している．このような削除を行っても安全に計算が進められる必要がある．

$$scprog[(\texttt{begin}\ vdef^+)] \to scprog[(\texttt{begin})] \text{ (BEGINEND)}$$

$$scprog[(\texttt{begin}\ vdef^*\ (\texttt{begin})\ s^*)] \to scprog[(\texttt{begin}\ vdef^*\ s^*)] \text{ (BEGINNEXT)}$$

$$scprog[(\texttt{begin}\ vdef^*\ (\texttt{return}\ v)\ s^*)] \to scprog[(\texttt{return}\ v)] \text{ (BEGINRETURN)}$$

4.2.5 lookup

$lookup(id, scprog) = v$ は変数アクセスの文脈 $scprog$において最も近い $vdef$から外側の大域変数に向かい idが一致する $vdef$を見つけ（てその値 v を得）る形で定義できる．

$$\frac{(\texttt{def}\ id\ \texttt{int}\ v) \in gdef^*}{lookup(id, (\texttt{prog}\ gdef^* (\texttt{exec}\ dn_1\ id_1\ [])\ calling\text{-}exec^*)) = v} \text{ (REFGDEF)}$$

$$lookup(id_1, (\texttt{prog}\ gdef^* (\texttt{exec}\ dn_2\ id_2\ vdef^* (\texttt{def}\ id_1\ \texttt{int}\ v_1)\ [])\ calling\text{-}exec^*)) = v_1 \text{ (REFPARAM)}$$

$$\frac{id_1 \neq id_3 \quad lookup(id_3, (\texttt{prog}\ gdef^* (\texttt{exec}\ dn_2\ id_2\ vdef^*\ [])\ calling\text{-}exec^*)) = v_3}{lookup(id_3, (\texttt{prog}\ gdef^* (\texttt{exec}\ dn_2\ id_2\ vdef^* (\texttt{def}\ id_1\ \texttt{int}\ v_1)\ [])\ calling\text{-}exec^*)) = v_3} \text{ (REFPARAMFIND)}$$

$$lookup(id, scprog[(\texttt{begin}\ vdef^*\ (\texttt{def}\ id\ \texttt{int}\ v)\ []\ s^*)]) = v \text{ (REFLDEF)}$$

$$\frac{id_1 \neq id_2 \quad lookup(id_2, scprog[(\texttt{begin}\ vdef^*\ []\ s^*)]) = v_2}{lookup(id_2, scprog[(\texttt{begin}\ vdef^*\ (\texttt{def}\ id_1\ \texttt{int}\ v_1)\ []\ s^*)]) = v_2} \text{ (REFLDEFFIND)}$$

$$\frac{lookup(id, scprog) = v}{lookup(id, scprog[(\texttt{begin}\ []\ s^*)]) = v} \text{ (REFFINDBEGIN)}$$

4.2.6 update

$update(id, v, scprog_1) = scprog_2$ も変数アクセスの文脈 $scprog_1$ において最も近い $vdef$から外側の大域変数に向かい idが一致する $vdef$を見つけ（てその値を v に更新した $scprog_2$ を得）る形で定義できる．

$$update(id_1, v_1, \texttt{(prog } gdef_1^* \texttt{ (def } id_1 \texttt{ int } v_2\texttt{) } gdef^* \texttt{ (exec } dn_2\ id_2\ []\texttt{) } calling\text{-}exec^*\texttt{))} = \texttt{(prog } gdef_1^* \texttt{ (def } id_1 \texttt{ int } v_1\texttt{) } gdef^* \texttt{ (exec } dn_2\ id_2\ []\texttt{) } calling\text{-}exec^*\texttt{)} \quad \text{(updateGdef)}$$

$$update(id_1, v_1, \texttt{(prog } gdef^* \texttt{ (exec } dn_2\ id_2\ vdef^* \texttt{ (def } id_1 \texttt{ int } v_2\texttt{) []) } calling\text{-}exec^*\texttt{))} = \texttt{(prog } gdef^* \texttt{ (exec } dn_2\ id_2\ vdef^* \texttt{ (def } id_1 \texttt{ int } v_1\texttt{) []) } calling\text{-}exec^*\texttt{)} \quad \text{(updateParam)}$$

$$\frac{id_1 \neq id_3 \qquad update(id_3, v_3, \texttt{(prog } gdef_1^* \texttt{ (exec } dn_2\ id_2\ vdef_1^*\ []\texttt{) } calling\text{-}exec^*\texttt{))} = \texttt{(prog } gdef_2^* \texttt{ (exec } dn_2\ id_2\ vdef_2^*\ []\texttt{) } calling\text{-}exec^*\texttt{)}}{update(id_3, v_3, \texttt{(prog } gdef_1^* \texttt{ (exec } dn_2\ id_2\ vdef_1^* \texttt{ (def } id_1 \texttt{ int } v_1\texttt{) []) } calling\text{-}exec^*\texttt{))} = \texttt{(prog } gdef_2^* \texttt{ (exec } dn_2\ id_2\ vdef_2^* \texttt{ (def } id_1 \texttt{ int } v_1\texttt{) []) } calling\text{-}exec^*\texttt{)} \quad \text{(updateParamFind)}$$

$$update(id_1, v_2, scprog[\texttt{(begin } vdef^* \texttt{ (def } id_1 \texttt{ int } v_1\texttt{) [] } s^*\texttt{)}]) = scprog[\texttt{(begin } vdef^* \texttt{ (def } id_1 \texttt{ int } v_2\texttt{) [] } s^*\texttt{)}] \quad \text{(updateLdef)}$$

$$\frac{id_1 \neq id_2 \qquad update(id_2, v_2, scprog_1[\texttt{(begin } vdef_1^* \texttt{ [] } s^*\texttt{)}]) = scprog_2[\texttt{(begin } vdef_2^* \texttt{ [] } s^*\texttt{)}]}{update(id_2, v_2, scprog_1[\texttt{(begin } vdef_1^* \texttt{ (def } id_1 \texttt{ int } v_1\texttt{) [] } s^*\texttt{)}]) = scprog_2[\texttt{(begin } vdef_2^* \texttt{ (def } id_1 \texttt{ int } v_1\texttt{) [] } s^*\texttt{)}] \quad \text{(updateLdefFind)}$$

$$\frac{update(id, v, scprog_1) = scprog_2}{update(id, v, scprog_1[\texttt{(begin [] } s^*\texttt{)}]) = scprog_2[\texttt{(begin [] } s^*\texttt{)}] \quad \text{(updateFindBegin)}}$$

4.3 Typing

コア SC 言語の仕様で stuck する可能性があるのは，

- 整数演算の対象となる値が式から求められない．特に削除された局所変数を含め，存在しない変数の値を得ようとするアクセス．
- 整数演算の対象となる値が整数でない．
- 文の実行に必要な値が式から求められない．特に削除された局所変数を含め，存在しない変数の値を得ようとするアクセス．
- 削除された局所変数を含め，アサインメントの対象となる変数が存在しない．
- 整数値を必要とする文の実行で値が整数でない．
- 存在しない関数の call
- 呼び出しの引数の個数が異なる
- `main` 以外の関数で，実行すべき文が return 前に無くなる．
- return 先が無い．
- 呼び出し中の文の実行が return 後に再開されない
- 入出力待ち．入力された値が整数でない．

があるが，これらは入出力を除き適切な型システムにより防止できる．特に $gdef$ の関数定義の文が `calling` 文を含んでいる場合は型検査をパスさせない．入出力が常に正常に可能とするなら，「型検査をパスしたプログラムの実行が停止するなら，それは `main` で実行する文が無くなる形の正常終了」となるような型システムとできる．

5 コア SC-RE 言語

コア SC 言語の計算状態を `do-handle` 構文を用いて操作できるようにしたコア SC-RE 言語を示す．静的リンク非明示で元の文脈においてハンドラ起動とするコア SC-RE 言語については有効となる S 式や遷移規則に「`#-s`」を付与して示す．一方，コア SC-REs 言語（静的リンクを明示化して隣接する呼び出しは隣

接する配置にしたもの）については有効となる S 式や遷移規則に「`#+s`」を付与して示す．コア SC-RE 言語とコア SC-REs 言語は本来等価であるが，SC-REs 言語には意味論において静的リンクを明示化するための識別子を構文においても求めることとする．

5.1 Syntax

コア SC-RE 言語／コア SC-REs 言語の構文を以下に示す．

```
    num ::= 0 | 1 | 2 | ...
     dn ::= num
 #+ssid ::= id
      v ::= num
      e ::= v | id | (+ e e) | (> e e)
   vdef ::= (def id int v)
 hcalls ::= (hcalls id+) | ε
  hexec ::= (hexec dn id vdef* s)
#+scexec ::= (seeing (sid dn) hexec)
   hdef ::= (def (id id*) s)
      s ::= (= id e)
          | (= id (in e)) | (out e e)
          | (if e s s) | (while e s)
          | (begin vdef* s*)
          | (do-handle #-shexec* #+ssid hdef* s)
          | (= id (call id e*))
          | (= id (calling dn))
          | (return e)
          | (= id (hcall id e*))
          | (= id (hcalling dn))
          | (hreturn e)
  hdecl ::= (decl id (handler int int*) hcalls v)
   gdef ::= vdef
          | (def (id id*) (fn int int*) hcalls s)
   exec ::= (exec dn id vdef* s)
          | #+scexec
   prog ::= (prog hdecl* gdef*
                  exec* (exec 0 main vdef* s))
```

*hdecl*はハンドラ宣言であり，ハンドラ *id*（レベル），引数の個数，*hcalls*とデフォルト値を宣言する．この宣言はプログラム全体における大域的宣言である．*hcalls*には，そのハンドラの実行中に新たに呼び出せる別（レベル）のハンドラ *id*のリストを指定する．宣言済のハンドラ *id*のみ許すことでレベル間の関係は半順序関係となる．*hdecl*で指定したデフォルト値は実行スタック全体について連鎖的にハンドラを呼び出した場合などで，`main` に達してそれ以上呼び出せるハンドラが存在しないときに返す値とする．

*hcalls*は *gdef* の大域関数でも指定する．これは Java 言語でメソッドが潜在的に throw し得る例外のリストをメソッド定義時に `throws` で指定させるのと同様である．

`do-handle` 構文は実際のハンドラ定義 *hdef* を持つ．1 つの *hdecl*に対応して複数の *hdef* が存在してよく，`do-handle` 文を複数回実行するときはそのたびに *hdef* のインスタンスが存在し，*gdef* 内の *hdef* の再帰呼び出しなどでは同時に複数のインスタンスも存在する．

`hcall` によりハンドラを呼び出し，`hreturn` により呼び出し元に復帰する．動的スコープに基づき，特定の *hdef* のインスタンスが呼び出されたときには `do-handle` インスタンスの実行文脈において呼び出される．SC-RE 言語では *hdef* に対応する *hexec*が（最大 1 回）この場で起動する形となり，SC-REs 言語では同様の *hexec*をこの場（*sid*で識別）への静的リンク（*sid dn*）付きの *cexec*として起動する形となる．SC-REs 言語の *sid*については実際にはプログラムとしては書かず，実行時に *exec*内で一意になるように自動生成してもよいし，コンパイラなどが各 *exec*内（*gdef* の関数定義内）で一意になるように自動生成してもよい．

初期プログラムはハンドラ宣言などの違いを除き，コア SC 言語と同様である．

```
init-prog ::= (prog hdecl* gdef*
                    (exec 0 main vdef* s))
```

5.2 Semantics

コア SC-RE 言語，コア SC-REs 言語の意味論を small step semantics で示す．

5.2.1 Contexts and instructions

e を *ec*[*einsn*] と分解するための文脈とインストラクションは 4.2.1 節と同じである．*s* を *ecs*[*einsn*] と分解するための文脈はハンドラ呼び出しの分 4.2.1 節より拡大している．

$$
\begin{array}{rcl}
ecs & ::= & \texttt{(= } id\ ec\texttt{)} \\
& | & \texttt{(= } id\ \texttt{(in } ec\texttt{))} \\
& | & \texttt{(out } ec\ e\texttt{)} \\
& | & \texttt{(out } v\ ec\texttt{)} \\
& | & \texttt{(if } ec\ s\ s\texttt{)} \\
& | & \texttt{(= } id\ \texttt{(call } id\ v^*\ ec\ e^*\texttt{))} \\
& | & \texttt{(return } ec\texttt{)} \\
& | & \texttt{(= } id\ \texttt{(hcall } id\ v^*\ ec\ e^*\texttt{))} \\
& | & \texttt{(hreturn } ec\texttt{)}
\end{array}
$$

s を $sc[sinsn]$ と分解するための文脈とインストラクションも拡大している．SC-RE 言語の場合，scは `do-handle` の持つ $hexec^*$ の任意の 1 つに（高々1 つに）$sinsn$のための [] を持てる．このとき，$hexec^*$ の残りは呼び出し中の *calling-hexec*で，`do-handle` の本体も呼び出し中の *calling-s*である．`do-handle` の本体が $sinsn$のための [] を持つなら，$hexec^*$ はすべて呼び出し中の *calling-hexec*である．

$$
\begin{array}{rcl}
\textit{calling-s} & ::= & sc[\texttt{(= } id\ \texttt{(calling } dn\texttt{))}] \\
& | & sc[\texttt{(= } id\ \texttt{(hcalling } dn\texttt{))}] \\
\textit{calling-hexec} & ::= & \texttt{(hexec } dn\ id\ vdef^*\ \textit{calling-s}\texttt{)} \\
\textit{schexec} & ::= & \texttt{(hexec } dn\ id\ vdef^*\ sc\texttt{)} \\
sc & ::= & [] \\
& | & \texttt{(begin } vdef^*\ sc\ s^*\texttt{)} \\
& | & \texttt{(do-handle \#-s}\textit{calling-hexec}^*\ \texttt{\#+s}sid\ hdef^*\ sc\texttt{)} \\
& | & \texttt{\#-s(do-handle } \textit{calling-hexec}^*\ \textit{schexec}\ \textit{calling-hexec}^*\ hdef^*\ \textit{calling-s}\texttt{)} \\
\textit{skip-or-return} & ::= & \texttt{(begin)} \mid \texttt{(return } e\texttt{)} \\
& | & \texttt{(hreturn } e\texttt{)} \\
\textit{sinsn} & ::= & \texttt{(= } id\ e\texttt{)} \\
& | & \texttt{(= } id\ \texttt{(in } e\texttt{))} \\
& | & \texttt{(out } e\ e\texttt{)} \\
& | & \texttt{(if } e\ s\ s\texttt{)} \\
& | & \texttt{(while } e\ s\texttt{)} \\
& | & \texttt{(begin } vdef^+\texttt{)} \\
& | & \texttt{(begin } vdef^*\ \textit{skip-or-return}\ s^*\texttt{)} \\
& | & \texttt{(do-handle \#+s}sid\ hdef^*\ \textit{skip-or-return}\texttt{)} \\
& | & \texttt{\#-s(do-handle (hexec } dn\ id\ vdef^*\ \texttt{(hreturn } e\texttt{))}\ \textit{calling-hexec}^*\ hdef^*\ \textit{calling-s}\texttt{)} \\
& | & \texttt{(= } id\ \texttt{(call } id\ e^*\texttt{))} \\
& | & \texttt{(= } id\ \texttt{(hcall } id\ e^*\texttt{))}
\end{array}
$$

*prog*を *scprog*$[sinsn]$ と分解するための文脈も 4.2.1 節より拡大している．*scprog*は `(prog` $hdecl^*\ gdef^*\ exec^*$`)` と `main` を含めた $exec^*$ のうち，任意の 1 つに（高々1 つに）$sinsn$のための [] を持てる．*scprog*は hcall 時に対象となる *hdef*を持つ `do-handle` 文の文脈としても用いる．これにより手順を踏んでハンドラを連鎖的に hcall することで実行スタックに存在する任意の変数へアクセスできる．

$$
\begin{array}{rcl}
\textit{calling-exec} & ::= & \texttt{(exec } dn\ id\ vdef^*\ \textit{calling-s}\texttt{)} \\
& | & \texttt{\#+s(seeing (}sid\ dn\texttt{)}\ \textit{calling-hexec}\texttt{)} \\
\textit{scexec} & ::= & \texttt{(exec } dn\ id\ vdef^*\ sc\texttt{)} \\
& | & \texttt{\#+s(seeing (}sid\ dn\texttt{)}\ \textit{schexec}\texttt{)} \\
\textit{scprog} & ::= & \texttt{(prog } hdecl^*\ gdef^*\ \textit{calling-exec}^*\ \textit{scexec}\ \textit{calling-exec}^*\texttt{)}
\end{array}
$$

5.2.2 Configurations

評価関係 $(prog_1, dn_1) \to (prog_2, dn_2)$ を与える．

$$
\frac{\begin{array}{c} vdef_2^* = \texttt{(def } id^*\ \texttt{int}^*\ v^*\texttt{)} \\ dn_2 = dn_1 + 1 \\ \texttt{(def (}id_1\ id^*\texttt{) (fn int int}^*\texttt{)}\ hcalls\ s\texttt{)} \in gdef^* \end{array}}{\begin{array}{l} (\texttt{(prog } hdecl^*\ gdef^* \\ \quad \texttt{\#-s}\textit{calling-exec}_1^* \\ \quad scexec[\texttt{(= } id_2\ \texttt{(call } id_1\ v^*\texttt{))}] \\ \quad \textit{calling-exec}^*\texttt{)}, dn_1) \to \\ (\texttt{(prog } hdecl^*\ gdef^* \\ \quad \texttt{(exec } dn_2\ id_1\ vdef_2^*\ s\texttt{)} \\ \quad \texttt{\#-s}\textit{calling-exec}_1^* \\ \quad scexec[\texttt{(= } id_2\ \texttt{(calling } dn_2\texttt{))}] \\ \quad \textit{calling-exec}^*\texttt{)}, dn_2) \end{array}} \ (\textsc{call})
$$

$$
\frac{dn_2 = dn_1 + 1}{\begin{array}{l} (\texttt{(prog } hdecl^*\ gdef^* \\ \quad \texttt{(exec } dn_2\ id_1\ vdef^*\ \texttt{(return } v_3\texttt{))} \\ \quad \texttt{\#-s}\textit{calling-exec}_1^* \\ \quad scexec[\texttt{(= } id_2\ \texttt{(calling } dn_2\texttt{))}] \\ \quad \textit{calling-exec}_2^*\texttt{)}, dn_2) \to \\ (\texttt{(prog } hdecl^*\ gdef^* \\ \quad \texttt{\#-s}\textit{calling-exec}_1^* \\ \quad scexec[\texttt{(= } id_2\ v_3\texttt{)}] \\ \quad \textit{calling-exec}_2^*\texttt{)}, dn_1) \end{array}} \ (\textsc{return})
$$

ここで，SC-RE 言語／SC-REs 言語では，実行中のハンドラが大域関数を call することがある．

例外ハンドラの「hcall」では，*hlookup*により，対応する例外ハンドラ定義を見つける．これは動的スコープに基づいており，`main` に達しても例外ハンドラ定義が見つからないときは hcall の結果としてデ

フォルト値を返す.

$$\frac{\begin{array}{c} dn_2 = dn_1 + \texttt{1} \\ hlookup(id, scprog, \texttt{(= } id_1 \texttt{ (hcalling } dn_2\texttt{)))} \\ = (\texttt{(prog } hdecl^* \ gdef^* \ calling\text{-}exec^* \\ \quad \texttt{(exec 0 main } vdef^* \ \texttt{[])}), \\ \quad calling\text{-}s_1) \\ \texttt{(decl } id \texttt{ (handler int int}^*\texttt{) } hcalls \ v\texttt{)} \in hdecl^* \end{array}}{\begin{array}{l} (scprog[\texttt{(= } id_1 \texttt{ (hcall } id \ v^*\texttt{))}], dn_1) \\ \to (scprog[\texttt{(= } id_1 \ v\texttt{)}], dn_1) \end{array}} \text{(HCALLFAIL)}$$

また, *hlookup*で例外ハンドラ定義 `(def (`id id^*`) `s`)` が見つかるとき，これを持つ `do-handle` において（SC-RE 言語），あるいは，これを持つ `do-handle` への静的リンク下において（SC-REs 言語; このとき `do-handle` を持つのが大域関数のことと例外ハンドラのことがある），ハンドラを *hexec*として起動する.

$hlookup(id, scprog_1, calling\text{-}s_1) = (scprog_2, calling\text{-}s_2)$ は，文脈 $scprog_1$ で $calling\text{-}s_1$ がハンドラ idを hcall するとき，実行中のプログラム全体 $scprog_1[calling\text{-}s_1]$ を全体としては等しい $scprog_2[calling\text{-}s_2]$ に分け直し，$scprog_2$ が対応する *hdef*を持つ `do-handle` を直接含む文脈で，$calling\text{-}s_2$ が捨てずに残す文となるようにしたものである．ただし，$calling\text{-}s_2$ 以外に $scprog_2$ にも捨てないでいる文脈（動的リンクをたどる前の文脈）が残っている．また，対応する *hdef*が `main` までさかのぼっても見つからなければ，その文脈を $scprog_2$ として返す．さらに同一レベルの hcall の場合は，ハンドラの呼び出し元へと動的リンクをたどるのではなく，例外ハンドラを持つ（静的リンク先の）`do-handle` として hcall することで連鎖できるようにする．これは catch & throw にて catch 先の例外ハンドラで再 throw したとき，同じ例外ハンドラに再度 catch されないのと同様である．*hlookup*の具体的な定義は後で述べる.

$$\frac{\begin{array}{c} dn_2 = dn_1 + \texttt{1} \\ hlookup(id, scprog, \texttt{(= } id_1 \texttt{ (hcalling } dn_2\texttt{)))} \\ = (\texttt{(prog } hdecl^* \ gdef^* \\ \quad calling\text{-}exec_1^* \\ \quad scexec[\texttt{(do-handle } calling\text{-}hexec^* \ hdef^* \ \texttt{[])}] \\ \quad calling\text{-}exec^*\texttt{)}, calling\text{-}s_1) \\ \texttt{(decl } id \texttt{ (handler int int}^*\texttt{) } hcalls \ v\texttt{)} \in hdecl^* \\ \texttt{(def (}id \ id^*\texttt{) } s\texttt{)} \in hdef^* \\ vdef^* = \texttt{(def } id^* \texttt{ int}^* \ v^*\texttt{)} \end{array}}{\begin{array}{l} (scprog[\texttt{(= } id_1 \texttt{ (hcall } id \ v^*\texttt{))}], dn_1) \\ \to (\texttt{(prog } hdecl^* \ gdef^* \\ \quad calling\text{-}exec_1^* \\ \quad scexec[\texttt{(do-handle} \\ \qquad \texttt{(hexec } dn_2 \ id \ vdef^* \ s\texttt{)} \\ \qquad calling\text{-}hexec^* \\ \qquad hdef^* \ calling\text{-}s_1\texttt{)}] \\ \quad calling\text{-}exec^*\texttt{)}, dn_2) \end{array}} \text{(\#-S:HCALL)}$$

$$\frac{\begin{array}{c} dn_2 = dn_1 + \texttt{1} \\ hlookup(id, scprog, \texttt{(= } id_1 \texttt{ (hcalling } dn_2\texttt{)))} \\ = (\texttt{(prog } hdecl^* \ gdef^* \\ \quad calling\text{-}exec_1^* \\ \quad \texttt{(exec } dn_3 \ id_3 \ vdef^* \\ \quad\quad sc[\texttt{(do-handle } sid \ hdef^* \ \texttt{[])}]\texttt{)} \\ \quad calling\text{-}exec^*\texttt{)}, calling\text{-}s_1) \\ \texttt{(decl } id \texttt{ (handler int int}^*\texttt{) } hcalls \ v\texttt{)} \in hdecl^* \\ \texttt{(def (}id \ id^*\texttt{) } s\texttt{)} \in hdef^* \\ vdef^* = \texttt{(def } id^* \texttt{ int}^* \ v^*\texttt{)} \end{array}}{\begin{array}{l} (scprog[\texttt{(= } id_1 \texttt{ (hcall } id \ v^*\texttt{))}], dn_1) \\ \to (\texttt{(prog } hdecl^* \ gdef^* \\ \quad \texttt{(seeing (}sid \ dn_3\texttt{) (hexec } dn_2 \ id \ vdef^* \ s\texttt{))} \\ \quad calling\text{-}exec_1^* \\ \quad \texttt{(exec } dn_3 \ id_3 \ vdef^* \\ \quad\quad sc[\texttt{(do-handle } sid \ hdef^* \ \texttt{[])}]\texttt{)} \\ \quad calling\text{-}exec^*\texttt{)}, dn_2) \end{array}} \text{(\#+S:HCALLEXEC)}$$

$$dn_2 = dn_1 + \texttt{1}$$
$$\begin{array}{l} \mathit{hlookup}(\mathit{id}, \mathit{scprog}, \texttt{(= } \mathit{id}_1 \texttt{ (hcalling } \mathit{dn}_2\texttt{)))} \\ = (\texttt{(prog } \mathit{hdecl}^* \ \mathit{gdef}^* \\ \quad \mathit{calling\text{-}exec}_1^* \\ \quad \texttt{(seeing (}\mathit{sid}_1 \ \mathit{dn}_4\texttt{)} \\ \quad\ \texttt{(hexec } \mathit{dn}_3 \ \mathit{id}_3 \ \mathit{vdef}^* \\ \quad\quad \mathit{sc}[\texttt{(do-handle } \mathit{sid} \ \mathit{hdef}^* \ [\,]\texttt{)}]\texttt{)} \\ \quad \mathit{calling\text{-}exec}^*\texttt{)}, \mathit{calling\text{-}s}_1) \\ \texttt{(decl } \mathit{id} \texttt{ (handler int int}^*\texttt{) } \mathit{hcalls} \ v\texttt{)} \in \mathit{hdecl}^* \\ \quad\ \texttt{(def (}\mathit{id} \ \mathit{id}^*\texttt{) } s\texttt{)} \in \mathit{hdef}^* \\ \quad\ \mathit{vdef}^* = \texttt{(def } \mathit{id}^* \texttt{ int}^* \ v^*\texttt{)} \end{array}$$
$$\begin{array}{l} (\mathit{scprog}[\texttt{(= } \mathit{id}_1 \texttt{ (hcall } \mathit{id} \ v^*\texttt{))}], \mathit{dn}_1) \\ \to (\texttt{(prog } \mathit{hdecl}^* \ \mathit{gdef}^* \\ \quad \texttt{(seeing (}\mathit{sid} \ \mathit{dn}_3\texttt{) (hexec } \mathit{dn}_2 \ \mathit{id} \ \mathit{vdef}^* \ s\texttt{))} \\ \quad \mathit{calling\text{-}exec}_1^* \\ \quad \texttt{(seeing (}\mathit{sid}_1 \ \mathit{dn}_4\texttt{)} \\ \quad\ \texttt{(hexec } \mathit{dn}_3 \ \mathit{id}_3 \ \mathit{vdef}^* \\ \quad\quad \mathit{sc}[\texttt{(do-handle } \mathit{sid} \ \mathit{hdef}^* \ [\,]\texttt{)}]\texttt{)} \\ \quad \mathit{calling\text{-}exec}^*\texttt{)}, \mathit{dn}_2) \end{array}$$
(#+S:HCALLHEXEC)

以下は SC-RE 言語，SC-REs 言語それぞれでの hreturn による遷移である．SC-RE 言語の場合，制御を戻す動的リンク先は実行中プログラム内の様々なところに存在し得る．SC-REs 言語の場合は，*prog*の *exec** の先頭の *exec*から隣接する *exec*へ制御を渡し，*exec** は 1 つ縮める（先頭を除く）形となる．

$$\begin{array}{l} \mathit{scprog}_1[\texttt{(do-handle } \mathit{calling\text{-}hexec}_3^* \ \mathit{hdef}^* \ \mathit{calling\text{-}s}\texttt{)}] \\ = \mathit{scprog}_2[\texttt{(= } \mathit{id}_2 \texttt{ (hcalling } \mathit{dn}_2\texttt{))}] \\ \qquad\qquad dn_2 = dn_1 + \texttt{1} \end{array}$$
$$\begin{array}{l} (\mathit{scprog}_1[\texttt{(do-handle} \\ \qquad\quad \texttt{(hexec } \mathit{dn}_2 \ \mathit{id}_1 \ \mathit{vdef}^* \texttt{ (hreturn } v_3\texttt{))} \\ \qquad\quad \mathit{calling\text{-}hexec}_3^* \ \mathit{hdef}^* \ \mathit{calling\text{-}s}\texttt{)}], \mathit{dn}_2) \\ \to (\mathit{scprog}_2[\texttt{(= } \mathit{id}_2 \ v_3\texttt{)}], \mathit{dn}_1) \end{array}$$
(#-S:HRETURN)

$$dn_2 = dn_1 + \texttt{1}$$
$$\begin{array}{l} (\texttt{(prog } \mathit{hdecl}^* \ \mathit{gdef}^* \\ \quad \texttt{(seeing (}\mathit{sid} \ \mathit{dn}_3\texttt{)} \\ \quad\ \texttt{(hexec } \mathit{dn}_2 \ \mathit{id}_1 \ \mathit{vdef}^* \texttt{ (hreturn } v_3\texttt{)))} \\ \quad \mathit{scexec}[\texttt{(= } \mathit{id}_2 \texttt{ (hcalling } \mathit{dn}_2\texttt{))}] \\ \quad \mathit{calling\text{-}exec}_2^*\texttt{)}, \mathit{dn}_2) \to \\ (\texttt{(prog } \mathit{hdecl}^* \ \mathit{gdef}^* \\ \quad \mathit{scexec}[\texttt{(= } \mathit{id}_2 \ v_3\texttt{)}] \\ \quad \mathit{calling\text{-}exec}_2^*\texttt{)}, \mathit{dn}_1) \end{array}$$
(#+S:HRETURN)

これら call と return と hcall と hreturn 以外では動的リンクの変更はなく，評価関係 $prog_1 \to prog_2$ に基づく．

$$\frac{prog_1 \to prog_2}{(prog_1, dn) \to (prog_2, dn)} \quad \text{(PROG)}$$

5.2.3 Expressions

$prog_1 = scprog[ecs[einsn]]$ のときの評価関係 $prog_1 \to prog_2$ は，4.2.3 節と同じとなる．

5.2.4 Statements

$prog_1 = scprog[sinsn]$ のときの評価関係 $prog_1 \to prog_2$ には，4.2.4 節のものに以下を加える．

$$\begin{array}{l} \mathit{scprog}[\texttt{(begin } \mathit{vdef}^* \texttt{ (hreturn } v\texttt{) } s^*\texttt{)}] \\ \to \mathit{scprog}[\texttt{(hreturn } v\texttt{)}] \end{array}$$
(BEGINHRETURN)

$$\begin{array}{l} \mathit{scprog}[\texttt{(do-handle \#+s}\mathit{sid} \ \mathit{hdef}^* \texttt{ (begin))}] \\ \to \mathit{scprog}[\texttt{(begin)}] \end{array}$$
(DOHANDLEBEGIN)

$$\begin{array}{l} \mathit{scprog}[\texttt{(do-handle \#+s}\mathit{sid} \ \mathit{hdef}^* \texttt{ (return } v\texttt{))}] \\ \to \mathit{scprog}[\texttt{(return } v\texttt{)}] \end{array}$$
(DOHANDLERETURN)

$$\begin{array}{l} \mathit{scprog}[\texttt{(do-handle \#+s}\mathit{sid} \ \mathit{hdef}^* \texttt{ (hreturn } v\texttt{))}] \\ \to \mathit{scprog}[\texttt{(hreturn } v\texttt{)}] \end{array}$$
(DOHANDLEHRETURN)

5.2.5 hlookup

$\mathit{hlookup}(\mathit{id}, \mathit{scprog}_1, \mathit{calling\text{-}s}_1) = (\mathit{scprog}_2, \mathit{calling\text{-}s}_2)$ は，文脈 scprog_1 で $\mathit{calling\text{-}s}_1$ がハンドラ *id*を hcall するとき，実行中のプログラム全体 $\mathit{scprog}_1[\mathit{calling\text{-}s}_1]$ を全体としては等しい $\mathit{scprog}_2[\mathit{calling\text{-}s}_2]$ に分け直し，scprog_2 が対応する *hdef*を持つ `do-handle` を直接含む文脈で，$\mathit{calling\text{-}s}_2$ が捨てずに残す文となるようにしたものである．

`do-handle` が対応する *hdef*を持たない場合は，その外側の探索結果を用いる．

$$\frac{\begin{array}{l}\forall\texttt{(def (}id_1\ id^*\texttt{) }s\texttt{)} \in hdef^*.id_1 \neq id \\ hlookup(id, scprog, \\ \quad \texttt{(do-handle \#-s}calling\text{-}hexec^*\ \texttt{\#+s}sid \\ \quad\quad hdef^*\ calling\text{-}s\texttt{))} \\ = (scprog_1, calling\text{-}s_1)\end{array}}{\begin{array}{l}hlookup(id, \\ \quad scprog[\texttt{(do-handle \#-s}calling\text{-}hexec^*\ \texttt{\#+s}sid \\ \quad hdef^*\ []\texttt{)}], \\ \quad calling\text{-}s) \\ = (scprog_1, calling\text{-}s_1)\end{array}}\ (\textsc{hlookupDoHandle})$$

以下のように対応する $hdef$ を持つ `do-handle` を直接含む文脈 $scprog$ が見つかるときはこれを返す．また $calling\text{-}s$ が捨てずに残す文となるようする．ただし，$calling\text{-}s$ 以外に $scprog$ にも捨てないでいる文脈（動的リンクをたどる前の文脈）が残っている．

$$\frac{\texttt{(def (}id\ id^*\texttt{) }s\texttt{)} \in hdef^*}{\begin{array}{l}hlookup(id, \\ \quad scprog[\texttt{(do-handle \#-s}calling\text{-}hexec^* \\ \quad\quad \texttt{\#+s}sid\ hdef^*\ []\texttt{)}], \\ \quad calling\text{-}s) \\ = (scprog[\texttt{(do-handle \#-s}calling\text{-}hexec^* \\ \quad\quad \texttt{\#+s}sid\ hdef^*\ []\texttt{)}], \\ \quad calling\text{-}s)\end{array}}\ (\textsc{hlookupHdef})$$

以下のように `main` の時点の文脈までさかのぼっても対応する $hdef$ が見つからなければ，その文脈を返す．

$$\begin{array}{l}hlookup(id, \\ \quad \texttt{(prog }hdecl^*\ gdef^*\ calling\text{-}exec^* \\ \quad\ \texttt{(exec 0 main }vdef^*\ []\texttt{))}, \\ \quad calling\text{-}s) \\ = (\texttt{(prog }hdecl^*\ gdef^*\ calling\text{-}exec^* \\ \quad\ \texttt{(exec 0 main }vdef^*\ []\texttt{))}, \\ \quad calling\text{-}s)\end{array}\ (\textsc{hlookupFail})$$

id_2 の例外ハンドラの処理内部に別レベルの例外 id_1 に対応するハンドラが見つからなければ，id_2 の呼び出し元への動的リンク dn_1 をたどる．

$$\frac{\begin{array}{l}\texttt{(prog }hdecl_1^* \\ \quad \texttt{(decl }id_1\ \texttt{(handler int int}_1^*\texttt{)} \\ \quad\quad hcalls_1\ v_1\texttt{)} \\ \quad hdecl_2^* \\ \quad \texttt{(decl }id_2\ \texttt{(handler int int}_2^*\texttt{)} \\ \quad\quad \texttt{(hcalls }id_2^*\ id_1\ id_3^*\texttt{)}\ v_2\texttt{)} \\ \quad hdecl_3^*\ gdef^*\ calling\text{-}exec^*\texttt{)} \\ = scprog_1[\texttt{(do-handle} \\ \quad\quad\quad calling\text{-}hexec_1^* \\ \quad\quad\quad \texttt{(hexec }dn_1\ id_2\ vdef^*\ calling\text{-}s_1\texttt{)} \\ \quad\quad\quad calling\text{-}hexec_2^* \\ \quad\quad\quad hdef^*\ calling\text{-}s\texttt{)}] \\ = scprog_2[\texttt{(= }id_3\ \texttt{(hcalling }dn_1\texttt{))}] \\ hlookup(id_1, scprog_2, \texttt{(= }id_3\ \texttt{(hcalling }dn_1\texttt{)))} \\ = (scprog_3, calling\text{-}s_3)\end{array}}{\begin{array}{l}hlookup(id_1, \\ \quad scprog_1[\texttt{(do-handle} \\ \quad\quad\quad calling\text{-}hexec_1^* \\ \quad\quad\quad \texttt{(hexec }dn_1\ id_2\ vdef^*\ []\texttt{)} \\ \quad\quad\quad calling\text{-}hexec_2^* \\ \quad\quad\quad hdef^*\ calling\text{-}s\texttt{)}], calling\text{-}s_1) \\ = (scprog_3, calling\text{-}s_3)\end{array}}\ (\texttt{\#-s}\textsc{:hlookupHexec})$$

$$\frac{\begin{array}{l}\texttt{(prog }hdecl_1^* \\ \quad \texttt{(decl }id_1\ \texttt{(handler int int}_1^*\texttt{)} \\ \quad\quad hcalls_1\ v_1\texttt{)} \\ \quad hdecl_2^* \\ \quad \texttt{(decl }id_2\ \texttt{(handler int int}_2^*\texttt{)} \\ \quad\quad \texttt{(hcalls }id_2^*\ id_1\ id_3^*\texttt{)}\ v_2\texttt{)} \\ \quad hdecl_3^*\ gdef^*\ calling\text{-}exec^*\texttt{)} \\ = \texttt{(prog }hdecl^*\ gdef^* \\ \quad calling\text{-}exec_1^* \\ \quad \texttt{(seeing (}sid\ dn_2\texttt{)} \\ \quad\quad\quad \texttt{(hexec }dn_1\ id_2\ vdef^*\ calling\text{-}s_1\texttt{))} \\ \quad calling\text{-}exec_2^*\texttt{)} \\ = scprog_1[\texttt{(= }id_3\ \texttt{(hcalling }dn_1\texttt{))}] \\ hlookup(id_1, scprog_1, \texttt{(= }id_3\ \texttt{(hcalling }dn_1\texttt{)))} \\ = (scprog_2, calling\text{-}s_2)\end{array}}{\begin{array}{l}hlookup(id_1, \texttt{(prog }hdecl^*\ gdef^* \\ \quad\quad\quad calling\text{-}exec_1^* \\ \quad\quad\quad \texttt{(seeing (}sid\ dn_2\texttt{)} \\ \quad\quad\quad\quad \texttt{(hexec }dn_1\ id_2\ vdef^*\ []\texttt{))} \\ \quad\quad\quad calling\text{-}exec_2^*\texttt{)}, calling\text{-}s_1) \\ = (scprog_2, calling\text{-}s_2)\end{array}}\ (\texttt{\#+s}\textsc{:hlookupHexec})$$

同一レベルの hcall の場合はハンドラの呼び出し元へと動的リンクをたどるのではなく，例外ハンドラを持つ（静的リンク先の）`do-handle` として hcall することで連鎖できるようにする．

$$\frac{hlookup(id_1, scprog, \texttt{(do-handle } calling\text{-}hexec_1^* \texttt{ (hexec } dn_1\ id_1\ vdef^*\ calling\text{-}s_1\texttt{) } calling\text{-}hexec_2^*\ hdef^*\ calling\text{-}s\texttt{)}) = (scprog_2, calling\text{-}s_2)}{hlookup(id_1, scprog[\texttt{(do-handle } calling\text{-}hexec_1^*\ \texttt{(hexec } dn_1\ id_1\ vdef^*\ \texttt{[]}\texttt{)}\ calling\text{-}hexec_2^*\ hdef^*\ calling\text{-}s\texttt{)}], calling\text{-}s_1) = (scprog_2, calling\text{-}s_2)}\ (\texttt{\#-s:}\textsc{hlookupDoHandleHexec})$$

$$\frac{hlookup(id_1, \texttt{(prog } hdecl^*\ gdef^*\ \texttt{(seeing (}sid\ dn\texttt{) (hexec } dn_1\ id_1\ calling\text{-}s_1\texttt{))}\ calling\text{-}exec_1^*\ \texttt{(exec } dn\ id_2\ vdef^*\ sc\texttt{[]}\texttt{)}\ calling\text{-}exec_2^*\texttt{)}, \texttt{(do-handle } sid\ hdef^*\ calling\text{-}s\texttt{)}) = (scprog_2, calling\text{-}s_2)}{hlookup(id_1, \texttt{(prog } hdecl^*\ gdef^*\ \texttt{(seeing (}sid\ dn\texttt{) (hexec } dn_1\ id_1\ \texttt{[]))}\ calling\text{-}exec_1^*\ \texttt{(exec } dn\ id_2\ vdef^*\ sc\texttt{[(do-handle } sid\ hdef^*\ calling\text{-}s\texttt{)]}\texttt{)}\ calling\text{-}exec_2^*\texttt{)}, calling\text{-}s_1) = (scprog_2, calling\text{-}s_2)}\ (\texttt{\#+s:}\textsc{hlookupDoHandleHexecExec})$$

$$\frac{hlookup(id_1, \texttt{(prog } hdecl^*\ gdef^*\ \texttt{(seeing (}sid\ dn\texttt{) (hexec } dn_1\ id_1\ calling\text{-}s_1\texttt{))}\ calling\text{-}exec_1^*\ \texttt{(seeing (}sid_2\ dn_2\texttt{)}\ \texttt{(hexec } dn\ id_2\ vdef^*\ sc\texttt{[]))}\ calling\text{-}exec_2^*\texttt{)}, \texttt{(do-handle } sid\ hdef^*\ calling\text{-}s\texttt{)}) = (scprog_2, calling\text{-}s_2)}{hlookup(id_1, \texttt{(prog } hdecl^*\ gdef^*\ \texttt{(seeing (}sid\ dn\texttt{) (hexec } dn_1\ id_1\ \texttt{[]))}\ calling\text{-}exec_1^*\ \texttt{(seeing (}sid_2\ dn_2\texttt{)}\ \texttt{(hexec } dn\ id_2\ vdef^*\ sc\texttt{[(do-handle } sid\ hdef^*\ calling\text{-}s\texttt{)]))}\ calling\text{-}exec_2^*\texttt{)}, calling\text{-}s_1) = (scprog_2, calling\text{-}s_2)}\ (\texttt{\#+s:}\textsc{hlookupDoHandleHexecHexec})$$

また，以下のように，大域関数の呼び出し元や `begin` の外側へと探索を続ける．

$$\frac{\texttt{(def (}id_2\ id^*\texttt{) (fn int int}^*\texttt{)}\ \texttt{(hcalls (}id_1^*\texttt{) }s\texttt{)} \in gdef^* \quad id_1 \in id_1^* \quad \texttt{(prog } hdecl^*\ gdef^*\ calling\text{-}exec_1^*\ \texttt{(exec } dn_1\ id_2\ vdef^*\ calling\text{-}s_1\texttt{)}\ calling\text{-}exec_2^*\texttt{)} = scprog_1\texttt{[(= } id_3\ \texttt{(calling } dn_1\texttt{))]} \quad hlookup(id_1, scprog_1, \texttt{(= } id_3\ \texttt{(calling } dn_1\texttt{))}) = (scprog_2, calling\text{-}s_2)}{hlookup(id_1, \texttt{(prog } hdecl^*\ gdef^*\ calling\text{-}exec_1^*\ \texttt{(exec } dn_1\ id_2\ vdef^*\ \texttt{[]}\texttt{)}\ calling\text{-}exec_2^*\texttt{)}, calling\text{-}s_1) = (scprog_2, calling\text{-}s_2)}\ (\textsc{hlookupExec})$$

$$\frac{hlookup(id_1, scprog, \texttt{(begin } vdef^*\ calling\text{-}s_1\ s^*\texttt{)}) = (scprog_2, calling\text{-}s_2)}{hlookup(id_1, scprog[\texttt{(begin } vdef^*\ \texttt{[]}\ s^*\texttt{)}], calling\text{-}s_1) = (scprog_2, calling\text{-}s_2)}\ (\textsc{hlookupBegin})$$

5.2.6 lookup

$lookup(id, scprog) = v$ は 4.2.5 節同様，変数アクセスの文脈 $scprog$において最も近い $vdef$から外側の大域変数に向かい idが一致する $vdef$を見つけ（てその値 v を得）る形で定義できるが，起動されたハンドラの「外側」はその定義を持つ `do-handle` の文脈となる．これは SC-RE 言語においては構文的な外側であり，SC-REs 言語においては静的リンク先である．

$$\frac{\texttt{(def } id_1\ \texttt{int } v_1\texttt{)} \in gdef^*}{lookup(id_1, \texttt{(prog } hdecl^*\ gdef^*\ \texttt{\#-s}calling\text{-}exec_1^*\ \texttt{(exec } dn_2\ id_2\ \texttt{[])}\ calling\text{-}exec^*\texttt{)}) = v_1}\ (\textsc{refGdef})$$

$$lookup(id_1, \texttt{(prog } hdecl^*\ gdef^*\ \texttt{\#-s}calling\text{-}exec_1^*\ \texttt{(exec } dn_2\ id_2\ vdef^*\ \texttt{(def } id_1\ \texttt{int } v_1\texttt{) []})\ calling\text{-}exec^*\texttt{)}) = v_1\ (\textsc{refParam})$$

$$\frac{id_1 \neq id_3 \quad lookup(id_3, \texttt{(prog } hdecl^* \ gdef^* \ \texttt{\#-s}calling\text{-}exec_1^* \ \texttt{(exec } dn_2 \ id_2 \ vdef^* \ \texttt{[])} \ calling\text{-}exec^*\texttt{))} = v_3}{lookup(id_3, \texttt{(prog } hdecl^* \ gdef^* \ \texttt{\#-s}calling\text{-}exec_1^* \ \texttt{(exec } dn_2 \ id_2 \ vdef^* \ \texttt{(def } id_1 \ \texttt{int } v_1\texttt{) [])} \ calling\text{-}exec^*\texttt{))} = v_3} \ (\textsc{refParamFind})$$

$$lookup(id_1, scprog[\texttt{(do-handle } calling\text{-}hexec_1^* \ \texttt{(hexec } dn_2 \ id_2 \ vdef^* \ \texttt{(def } id_1 \ \texttt{int } v_1\texttt{) [])} \ calling\text{-}hexec^* \ hdef^* \ calling\text{-}s\texttt{)}]) = v_1 \ (\texttt{\#-s}:\textsc{refHparam})$$

$$lookup(id_1, \texttt{(prog } hdecl^* \ gdef^* \ \texttt{(seeing (}sid \ dn\texttt{)} \ \texttt{(hexec } dn_2 \ id_2 \ vdef^* \ \texttt{(def } id_1 \ \texttt{int } v_1\texttt{) []))} \ calling\text{-}exec^*\texttt{))} = v_1 \ (\texttt{\#+s}:\textsc{refHparam})$$

$$\frac{id_1 \neq id_3 \quad lookup(id_3, scprog[\texttt{(do-handle } calling\text{-}hexec_1^* \ \texttt{(hexec } dn_2 \ id_2 \ vdef^* \ \texttt{[])} \ calling\text{-}hexec^* \ hdef^* \ calling\text{-}s\texttt{)}]) = v_3}{lookup(id_3, scprog[\texttt{(do-handle } calling\text{-}hexec_1^* \ \texttt{(hexec } dn_2 \ id_2 \ vdef^* \ \texttt{(def } id_1 \ \texttt{int } v_1\texttt{) [])} \ calling\text{-}hexec^* \ hdef^* \ calling\text{-}s\texttt{)}]) = v_3} \ (\texttt{\#-s}:\textsc{refHparamFind})$$

$$\frac{id_1 \neq id_3 \quad lookup(id_3, \texttt{(prog } hdecl^* \ gdef^* \ \texttt{(seeing (}sid \ dn\texttt{)} \ \texttt{(hexec } dn_2 \ id_2 \ vdef^* \ \texttt{[]))} \ calling\text{-}exec^*\texttt{))} = v_3}{lookup(id_3, \texttt{(prog } hdecl^* \ gdef^* \ \texttt{(seeing (}sid \ dn\texttt{)} \ \texttt{(hexec } dn_2 \ id_2 \ vdef^* \ \texttt{(def } id_1 \ \texttt{int } v_1\texttt{) []))} \ calling\text{-}exec^*\texttt{))} = v_3} \ (\texttt{\#+s}:\textsc{refHparamFind})$$

$$lookup(id_1, scprog[\texttt{(begin } vdef^* \ \texttt{(def } id_1 \ \texttt{int } v_1\texttt{)} \ \texttt{[]} \ s^*\texttt{)}]) = v_1 \ (\textsc{refLdef})$$

$$\frac{id_1 \neq id_2 \quad lookup(id_2, scprog[\texttt{(begin } vdef^* \ \texttt{[]} \ s^*\texttt{)}]) = v_2}{lookup(id_2, scprog[\texttt{(begin } vdef^* \ \texttt{(def } id_1 \ \texttt{int } v_1\texttt{)} \ \texttt{[]} \ s^*\texttt{)}]) = v_2} \ (\textsc{refLdefFind})$$

$$\frac{lookup(id, scprog) = v}{lookup(id, scprog[\texttt{(begin [] } s^*\texttt{)}]) = v} \ (\textsc{refFindBegin})$$

$$\frac{lookup(id, scprog) = v}{lookup(id, scprog[\texttt{(do-handle \#-s}calling\text{-}hexec^* \ \texttt{\#+s}sid \ hdef^* \ \texttt{[])}]) = v} \ (\textsc{refFindDoHandle})$$

$$\frac{lookup(id, scprog) = v}{lookup(id, scprog[\texttt{(do-handle } calling\text{-}hexec_1^* \ \texttt{(hexec } dn_1 \ id_1 \ \texttt{[])} \ calling\text{-}hexec^* \ hdef^* \ calling\text{-}s\texttt{)}]) = v} \ (\texttt{\#-s}:\textsc{refFindHexec})$$

$$\frac{lookup(id, \texttt{(prog } hdecl^* \ gdef^* \ \texttt{(exec } dn \ id_2 \ vdef^* \ sc\texttt{)} \ calling\text{-}exec_2^*\texttt{))} = v}{lookup(id, \texttt{(prog } hdecl^* \ gdef^* \ \texttt{(seeing (}sid \ dn\texttt{)} \ \texttt{(hexec } dn_1 \ id_1 \ \texttt{[]))} \ calling\text{-}exec_1^* \ \texttt{(exec } dn \ id_2 \ vdef^* \ sc[\texttt{(do-handle } sid \ hdef^* \ calling\text{-}s\texttt{)}] \ calling\text{-}exec_2^*\texttt{))} = v} \ (\texttt{\#+s}:\textsc{refFindHexecExec})$$

$$\frac{lookup(id, \texttt{(prog } hdecl^* \ gdef^* \ \texttt{(seeing (}sid_2 \ dn_2\texttt{)} \ \texttt{(hexec } dn \ id_2 \ vdef^* \ sc\texttt{))} \ calling\text{-}exec_2^*\texttt{))} = v}{lookup(id, \texttt{(prog } hdecl^* \ gdef^* \ \texttt{(seeing (}sid \ dn\texttt{)} \ \texttt{(hexec } dn_1 \ id_1 \ \texttt{[]))} \ calling\text{-}exec_1^* \ \texttt{(seeing (}sid_2 \ dn_2\texttt{)} \ \texttt{(hexec } dn \ id_2 \ vdef^* \ sc[\texttt{(do-handle } sid \ hdef^* \ calling\text{-}s\texttt{)}]\texttt{))} \ calling\text{-}exec_2^*\texttt{))} = v} \ (\texttt{\#+s}:\textsc{refFindHexecHexec})$$

5.2.7 update

$update(id, v, scprog_1) = scprog_2$ も 4.2.6 節同様，変数アクセスの文脈 $scprog_1$ において最も近い $vdef$ から外側の大域変数に向かい idが一致する $vdef$を見つけ（てその値を v に更新した $scprog_2$ を得）る形で定義できるが，起動されたハンドラの「外側」はその

定義を持つ `do-handle` の文脈となる．これは SC-RE 言語においては構文的な外側であり，SC-REs 言語においては静的リンク先である．

$$update(id_1, v_1,\ \texttt{(prog}\ hdecl^*\ gdef_1^*\ \texttt{(def}\ id_1\ \texttt{int}\ v_2\texttt{)}\ gdef^*\ \texttt{\#-s}calling\text{-}exec_1^*\ \texttt{(exec}\ dn_2\ id_2\ \texttt{[])}\ calling\text{-}exec^*\texttt{))} = \texttt{(prog}\ hdecl^*\ gdef_1^*\ \texttt{(def}\ id_1\ \texttt{int}\ v_1\texttt{)}\ gdef^*\ \texttt{\#-s}calling\text{-}exec_1^*\ \texttt{(exec}\ dn_2\ id_2\ \texttt{[])}\ calling\text{-}exec^*\texttt{)}\ (\textsc{updateGdef})$$

$$update(id_1, v_1,\ \texttt{(prog}\ hdecl^*\ gdef^*\ \texttt{\#-s}calling\text{-}exec_1^*\ \texttt{(exec}\ dn_2\ id_2\ vdef^*\ \texttt{(def}\ id_1\ \texttt{int}\ v_2\texttt{)}\ \texttt{[])}\ calling\text{-}exec^*\texttt{))} = \texttt{(prog}\ hdecl^*\ gdef^*\ \texttt{\#-s}calling\text{-}exec_1^*\ \texttt{(exec}\ dn_2\ id_2\ vdef^*\ \texttt{(def}\ id_1\ \texttt{int}\ v_1\texttt{)}\ \texttt{[])}\ calling\text{-}exec^*\texttt{)}\ (\textsc{updateParam})$$

$$\frac{id_1 \neq id_3 \qquad update(id_3, v_3,\ \texttt{(prog}\ hdecl^*\ gdef_1^*\ \texttt{\#-s}calling\text{-}exec_1^*\ \texttt{(exec}\ dn_2\ id_2\ vdef_1^*\ \texttt{[])}\ calling\text{-}exec^*\texttt{))} = \texttt{(prog}\ hdecl^*\ gdef_2^*\ \texttt{\#-s}calling\text{-}exec_1^*\ \texttt{(exec}\ dn_2\ id_2\ vdef_2^*\ \texttt{[])}\ calling\text{-}exec^*\texttt{)}}{update(id_3, v_3,\ \texttt{(prog}\ hdecl^*\ gdef_1^*\ \texttt{\#-s}calling\text{-}exec_1^*\ \texttt{(exec}\ dn_2\ id_2\ vdef_1^*\ \texttt{(def}\ id_1\ \texttt{int}\ v_1\texttt{)}\ \texttt{[])}\ calling\text{-}exec^*\texttt{))} = \texttt{(prog}\ hdecl^*\ gdef_2^*\ \texttt{\#-s}calling\text{-}exec_1^*\ \texttt{(exec}\ dn_2\ id_2\ vdef_2^*\ \texttt{(def}\ id_1\ \texttt{int}\ v_1\texttt{)}\ \texttt{[])}\ calling\text{-}exec^*\texttt{)}\ (\textsc{updateParamFind})$$

$$update(id_1, v_3,\ scprog[\texttt{(do-handle}\ calling\text{-}hexec_1^*\ \texttt{(hexec}\ dn_2\ id_2\ vdef^*\ \texttt{(def}\ id_1\ \texttt{int}\ v_1\texttt{)}\ \texttt{[])}\ calling\text{-}hexec^*\ hdef^*\ calling\text{-}s\texttt{)}]) = scprog[\texttt{(do-handle}\ calling\text{-}hexec_1^*\ \texttt{(hexec}\ dn_2\ id_2\ vdef^*\ \texttt{(def}\ id_1\ \texttt{int}\ v_3\texttt{)}\ \texttt{[])}\ calling\text{-}hexec^*\ hdef^*\ calling\text{-}s\texttt{)}]\ (\texttt{\#-s}:\textsc{updateHparam})$$

$$update(id_1, v_3,\ \texttt{(prog}\ hdecl^*\ gdef^*\ \texttt{(seeing (}sid\ dn\texttt{)}\ \texttt{(hexec}\ dn_2\ id_2\ vdef^*\ \texttt{(def}\ id_1\ \texttt{int}\ v_1\texttt{)}\ \texttt{[]))}\ calling\text{-}exec^*\texttt{))} = \texttt{(prog}\ hdecl^*\ gdef^*\ \texttt{(seeing (}sid\ dn\texttt{)}\ \texttt{(hexec}\ dn_2\ id_2\ vdef^*\ \texttt{(def}\ id_1\ \texttt{int}\ v_3\texttt{)}\ \texttt{[]))}\ calling\text{-}exec^*\texttt{)}\ (\texttt{\#+s}:\textsc{updateHparam})$$

$$\frac{id_1 \neq id_3 \qquad update(id_3, v_3,\ scprog_1[\texttt{(do-handle}\ calling\text{-}hexec_1^*\ \texttt{(hexec}\ dn_2\ id_2\ vdef_1^*\ \texttt{[])}\ calling\text{-}hexec^*\ hdef^*\ calling\text{-}s\texttt{)}]) = scprog_2[\texttt{(do-handle}\ calling\text{-}hexec_1^*\ \texttt{(hexec}\ dn_2\ id_2\ vdef_2^*\ \texttt{[])}\ calling\text{-}hexec^*\ hdef^*\ calling\text{-}s\texttt{)}]}{update(id_3, v_3,\ scprog_1[\texttt{(do-handle}\ calling\text{-}hexec_1^*\ \texttt{(hexec}\ dn_2\ id_2\ vdef_1^*\ \texttt{(def}\ id_1\ \texttt{int}\ v_1\texttt{)}\ \texttt{[])}\ calling\text{-}hexec^*\ hdef^*\ calling\text{-}s\texttt{)}]) = scprog_2[\texttt{(do-handle}\ calling\text{-}hexec_1^*\ \texttt{(hexec}\ dn_2\ id_2\ vdef_2^*\ \texttt{(def}\ id_1\ \texttt{int}\ v_1\texttt{)}\ \texttt{[])}\ calling\text{-}hexec^*\ hdef^*\ calling\text{-}s\texttt{)}]\ (\texttt{\#-s}:\textsc{updateHparamFind})$$

$$id_1 \neq id_3$$

$$\begin{array}{l} update(id_3, v_3, \\ \quad \texttt{(prog } hdecl^* \ gdef_1^* \\ \quad\quad \texttt{(seeing (}sid \ dn\texttt{) (hexec } dn_2 \ id_2 \ vdef_1^* \ \texttt{[]))} \\ \quad\quad calling\text{-}exec_1^*\texttt{))} \\ \quad = \texttt{(prog } hdecl^* \ gdef_2^* \\ \quad\quad \texttt{(seeing (}sid \ dn\texttt{) (hexec } dn_2 \ id_2 \ vdef_2^* \ \texttt{[]))} \\ \quad\quad calling\text{-}exec_2^*\texttt{)} \end{array}$$

$$\begin{array}{l} update(id_3, v_3, \\ \quad \texttt{(prog } hdecl^* \ gdef_1^* \\ \quad\quad \texttt{(seeing (}sid \ dn\texttt{)} \\ \quad\quad\quad \texttt{(hexec } dn_2 \ id_2 \ vdef_1^* \ \texttt{(def } id_1 \ \texttt{int } v_1\texttt{) []))} \\ \quad\quad calling\text{-}exec_1^*\texttt{))} \\ \quad = \texttt{(prog } hdecl^* \ gdef_2^* \\ \quad\quad \texttt{(seeing (}sid \ dn\texttt{)} \\ \quad\quad\quad \texttt{(hexec } dn_2 \ id_2 \ vdef_2^* \ \texttt{(def } id_1 \ \texttt{int } v_1\texttt{) []))} \\ \quad\quad calling\text{-}exec_2^*\texttt{)} \end{array}$$

(#+s:updateHparamFind)

$$\begin{array}{l} update(id, v_2, \\ \quad scprog[\texttt{(begin } vdef^* \ \texttt{(def } id \ \texttt{int } v_1\texttt{) [] } s^*\texttt{)}]) \\ \quad = scprog[\texttt{(begin } vdef^* \ \texttt{(def } id \ \texttt{int } v_2\texttt{) [] } s^*\texttt{)}] \end{array}$$

(updateLdef)

$$id_1 \neq id_2$$

$$\frac{update(id_2, v_2, scprog_1[\texttt{(begin } vdef_1^* \ \texttt{[] } s^*\texttt{)}]) = scprog_2[\texttt{(begin } vdef_2^* \ \texttt{[] } s^*\texttt{)}]}{\begin{array}{l} update(id_2, v_2, \\ \quad scprog_1[\texttt{(begin } vdef_1^* \ \texttt{(def } id_1 \ \texttt{int } v_1\texttt{)} \\ \quad\quad \texttt{[] } s^*\texttt{)}]) \\ \quad = scprog_2[\texttt{(begin } vdef_2^* \ \texttt{(def } id_1 \ \texttt{int } v_1\texttt{)} \\ \quad\quad \texttt{[] } s^*\texttt{)}] \end{array}}$$

(updateLdefFind)

$$\frac{update(id, v, scprog_1) = scprog_2}{\begin{array}{l} update(id, v, scprog_1[\texttt{(begin [] } s^*\texttt{)}]) \\ \quad = scprog_2[\texttt{(begin [] } s^*\texttt{)}] \end{array}}$$

(updateFindBegin)

$$\frac{update(id, v, scprog_1) = scprog_2}{\begin{array}{l} update(id, v, \\ \quad scprog_1[\texttt{(do-handle \#-s}calling\text{-}hexec^* \\ \quad\quad \texttt{\#+s}sid \ hdef^* \ \texttt{[])]}) \\ \quad = scprog_2[\texttt{(do-handle \#-s}calling\text{-}hexec^* \\ \quad\quad \texttt{\#+s}sid \ hdef^* \ \texttt{[])]} \end{array}}$$

(updateFindDoHandle)

$$\frac{update(id, v, scprog_1) = scprog_2}{\begin{array}{l} update(id, v, scprog_1[\texttt{(do-handle} \\ \quad\quad\quad\quad\quad\quad calling\text{-}hexec_1^* \\ \quad\quad\quad\quad\quad\quad \texttt{(hexec } dn_1 \ id_1 \ \texttt{[])} \\ \quad\quad\quad\quad\quad\quad calling\text{-}hexec^* \\ \quad\quad\quad\quad\quad\quad hdef^* \ calling\text{-}s\texttt{)]}) \\ \quad = scprog_2[\texttt{(do-handle} \\ \quad\quad\quad\quad\quad\quad calling\text{-}hexec_1^* \\ \quad\quad\quad\quad\quad\quad \texttt{(hexec } dn_1 \ id_1 \ \texttt{[])} \\ \quad\quad\quad\quad\quad\quad calling\text{-}hexec^* \\ \quad\quad\quad\quad\quad\quad hdef^* \ calling\text{-}s\texttt{)]} \\ \quad\quad\quad\quad\quad\quad \texttt{(\#-s:updateFindHexec)} \end{array}}$$

$$\begin{array}{l} update(id, v, \\ \quad \texttt{(prog } hdecl^* \ gdef_1^* \\ \quad\quad \texttt{(exec } dn \ id_2 \ vdef_1^* \ sc_1\texttt{)} \\ \quad\quad calling\text{-}exec_2^*\texttt{))} \\ \quad = \texttt{(prog } hdecl^* \ gdef_2^* \\ \quad\quad \texttt{(exec } dn \ id_2 \ vdef_2^* \ sc_2\texttt{)} \\ \quad\quad calling\text{-}exec_3^*\texttt{)} \end{array}$$

$$\begin{array}{l} update(id, v, \\ \quad \texttt{(prog } hdecl^* \ gdef_1^* \\ \quad\quad \texttt{(seeing (}sid \ dn\texttt{) (hexec } dn_1 \ id_1 \ \texttt{[]))} \\ \quad\quad calling\text{-}exec_1^* \\ \quad\quad \texttt{(exec } dn \ id_2 \ vdef_1^* \\ \quad\quad\quad sc_1[\texttt{(do-handle } sid \ hdef^* \ calling\text{-}s\texttt{)}]) \\ \quad\quad calling\text{-}exec_2^*\texttt{))} \\ \quad = \texttt{(prog } hdecl^* \ gdef_2^* \\ \quad\quad \texttt{(seeing (}sid \ dn\texttt{) (hexec } dn_1 \ id_1 \ \texttt{[]))} \\ \quad\quad calling\text{-}exec_1^* \\ \quad\quad \texttt{(exec } dn \ id_2 \ vdef_2^* \\ \quad\quad\quad sc_2[\texttt{(do-handle } sid \ hdef^* \ calling\text{-}s\texttt{)}]) \\ \quad\quad calling\text{-}exec_3^*\texttt{)} \end{array}$$

(#+s:updateFindHexecExec)

$update(id, v,$
$\quad$(prog $hdecl^*$ $gdef_1^*$
$\quad\quad$(seeing (sid_2 dn_2)
$\quad\quad\quad$(hexec dn id_2 $vdef_1^*$ sc_1))
$\quad\quad$$calling\text{-}exec_2^*$))
$\quad$= (prog $hdecl^*$ $gdef_2^*$
$\quad\quad$(seeing (sid_2 dn_2)
$\quad\quad\quad$(hexec dn id_2 $vdef_2^*$ sc_2))
$\quad\quad$$calling\text{-}exec_3^*$)

$update(id, v,$
$\quad$(prog $hdecl^*$ $gdef_1^*$
$\quad\quad$(seeing (sid dn) (hexec dn_1 id_1 []))
$\quad\quad$$calling\text{-}exec_1^*$
$\quad\quad$(seeing (sid_2 dn_2)
$\quad\quad\quad$(hexec dn id_2 $vdef_1^*$
$\quad\quad\quad\quad$$sc_1$[(do-handle sid $hdef^*$ $calling\text{-}s$)]))
$\quad\quad$$calling\text{-}exec_2^*$))
$\quad$= (prog $hdecl^*$ $gdef_2^*$
$\quad\quad$(seeing (sid dn) (hexec dn_1 id_1 []))
$\quad\quad$$calling\text{-}exec_1^*$
$\quad\quad$(seeing (sid_2 dn_2)
$\quad\quad\quad$(hexec dn id_2 $vdef_2^*$
$\quad\quad\quad\quad$$sc_2$[(do-handle sid $hdef^*$ $calling\text{-}s$)]))
$\quad\quad$$calling\text{-}exec_3^*$)
$\quad\quad\quad\quad$(#+s:updateFindHexecHexec)

6 おわりに

本発表では，S 式ベースの構文を持つ逐次命令型言語であるコア SC 言語の仕様を示すとともに，持続型例外処理機構を備えたコア SC-RE 言語（静的リンク非明示で元の文脈においてハンドラ起動）ならびにコア SC-REs 言語（静的リンクを明示化して隣接する呼び出しは隣接する配置にしたもの）の設計（構文，意味論など）を示した．また，意味論に基づく展望（今後の課題）を述べた．

謝辞 本研究の一部は JSPS 科研費 JP19H04087，JP21K19774，ならびに JP17K00099 の助成を受けたものである．

参 考 文 献

[1] Breuel, T. M.: Lexical Closure for C++, *In Proceedings of the USENIX C++ Conference*, 1988, pp. 293–304.

[2] Hiraishi, T., Yasugi, M., Umatani, S., and Yuasa, T.: Backtracking-based Load Balancing, *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2009)*, February 2009, pp. 55–64.

[3] Hiraishi, T., Yasugi, M., and Yuasa, T.: A Transformation-Based Implementation of Lightweight Nested Functions, *IPSJ Digital Courier*, Vol. 2(2006), pp. 262–279. (IPSJ Transactions on Programming, Vol. 47, No. SIG 6 (PRO 29), pp. 50–67, 2006).

[4] Yasugi, M., Hiraishi, T., and Yuasa, T.: Lightweight Lexical Closures for Legitimate Execution Stack Access, *Proceedings of the 15th International Conference on Compiler Construction (CC2006)*, Lecture Notes in Computer Science, No. 3923, Springer-Verlag, March 2006, pp. 170–184.

[5] Yasugi, M., Ikeuchi, R., Hiraishi, T., and Komiya, T.: Evaluating Portable Mechanisms for Legitimate Execution Stack Access with a Scheme Interpreter in an Extended SC Language, *Journal of Information Processing*, Vol. 27(2019), pp. 177–189.

[6] Yasugi, M., Muraoka, D., Hiraishi, T., Umatani, S., and Emoto, K.: HOPE: A Parallel Execution Model Based on Hierarchical Omission, *Proceedings of the 48th International Conference on Parallel Processing (ICPP 2019)*, August 2019, pp. 77:1–77:11.

[7] 八杉昌宏, 小島啓史, 小宮常康, 平石拓, 馬谷誠二, 湯淺太一: L-Closure を用いた真に末尾再帰的な Scheme インタプリタ, 情報処理学会論文誌　プログラミング, Vol. 3, No. 5(2010), pp. 1–17.

[8] 八杉昌宏, 平石拓, 篠原丈成, 湯淺太一: L-Closure：高性能・高信頼プログラミング言語の実装向け言語機構, 情報処理学会論文誌：プログラミング, Vol. 49, No. SIG 1 (PRO 35)(2008), pp. 63–83.

[9] 田附正充, 八杉昌宏, 平石拓, 馬谷誠二: L-Closure の呼び出しコストの削減, 情報処理学会論文誌　プログラミング, Vol. 6, No. 2(2013), pp. 13–32.

[10] 平石拓, 八杉昌宏, 湯淺太一: SC 言語処理系における変形規則の再利用機構, コンピュータソフトウェア, Vol. 28, No. 1(2011), pp. 258–271.

A 付録: プログラム例

A.1 SC 言語

A.1.1 初期プログラム例

```
(prog
 (def r int 0)
 (def (fib n) (fn int int)
  (if (> 3 n) (return 1)
   (begin (def r0 int 0) (def r1 int 0)
    (= r0 (call fib (+ n -1)))
    (= r1 (call fib (+ n -2)))
    (return (+ r0 r1)))))
 (exec 0 main (= r (call fib 3))))
```

A.1.2 実行中プログラム例

```
(prog
 (def r int 0)
 (def (fib n) (fn int int)
  (if (> 3 n) (return 1)
   (begin (def r0 int 0) (def r1 int 0)
    (= r0 (call fib (+ n -1)))
    (= r1 (call fib (+ n -2)))
    (return (+ r0 r1)))))
 (exec 2 fib (def n int 1)
  (if (> 3 1) (return 1)
   (begin (def r0 int 0) (def r1 int 0)
    (= r0 (call fib (+ n -1)))
    (= r1 (call fib (+ n -2)))
    (return (+ r0 r1)))))
 (exec 1 fib (def n int 3)
  (begin (def r0 int 1) (def r1 int 0)
   (= r1 (calling 2))
   (return (+ r0 r1))))
 (exec 0 main (= r (calling 1))))
```

A.1.3 終了時プログラム例

```
(prog
 (def r int 2)
 (def (fib n) (fn int int)
  (if (> 3 n) (return 1)
   (begin (def r0 int 0) (def r1 int 0)
    (= r0 (call fib (+ n -1)))
    (= r1 (call fib (+ n -2)))
    (return (+ r0 r1)))))
 (exec 0 main (begin)))
```

A.2 SC-RE 言語

A.2.1 初期プログラム例

```
(prog
 (decl bt (handler int) 0)
 (decl capt (handler int) (hcalls bt) 0)
 (def _ int 0) (def r int 0)
 (def (fib n) (fn int int) (hcalls bt capt)
  (if (> 3 n) (begin (= _ (hcall capt)) (return 1))
   (begin (def r0 int 0) (def r1 int 0)
    (do-handle
     (def (bt)
      (begin (def z100 int n)
       (= _ (hcall bt))
       (hreturn z100)))
     (def (capt)
      (do-handle
       (def (bt)
        (begin (def z500 int n)
         (= _ (hcall bt))
         (hreturn z500)))
       (begin
        (= _ (hcall capt))
        (out 10 n) (out 10 r0) (out 10 r1)
        (hreturn 1))))
     (begin
      (= r0 (call fib (+ n -1)))
      (= r1 (call fib (+ n -2)))
      (return (+ r0 r1)))))))
 (exec 0 main
  (do-handle
   (def (bt) (hreturn -2))
   (def (capt) (begin (= _ (hcall bt)) (hreturn -1)))
   (= r (call fib 3)))))
```

A.2.2 実行中プログラム例

```
(prog
 (decl bt (handler int) 0)
 (decl capt (handler int) (hcalls bt) 0)
 (def _ int 0) (def r int 0)
 (def (fib n) (fn int int) (hcalls bt capt)
  (if (> 3 n) (begin (= _ (hcall capt)) (return 1))
   (begin (def r0 int 0) (def r1 int 0)
    (do-handle
     (def (bt)
      (begin (def z100 int n)
       (= _ (hcall bt))
       (hreturn z100)))
     (def (capt)
      (do-handle
       (def (bt)
        (begin (def z500 int n)
         (= _ (hcall bt))
         (hreturn z500)))
       (begin
        (= _ (hcall capt))
        (out 10 n) (out 10 r0) (out 10 r1)
        (hreturn 1))))
     (begin
      (= r0 (call fib (+ n -1)))
      (= r1 (call fib (+ n -2)))
      (return (+ r0 r1)))))))
 (exec 2 fib (def n int 1)
  (begin (= _ (hcalling 3)) (return 1))) ;; (capt)
```

```
(exec 1 fib (def n int 3)
 (begin (def r0 int 1) (def r1 int 0)
  (do-handle
   (hexec 6 bt
    (begin (def z100 int 3)
     (= _ (hcalling 7)) ; (bt)
     (hreturn z100)))
   (hexec 3 capt
    (do-handle
     (hexec 5 bt
      (begin (def z500 int 3)
       (= _ (hcalling 6)) ; (bt)
       (hreturn z500)))
     (def (bt)
      (begin (def z500 int n)
       (= _ (hcall bt))
       (hreturn z500)))
     (begin
      (= _ (hcalling 4)) ;; (capt)
      (out 10 n) (out 10 r0) (out 10 r1)
      (hreturn 1))))
   (def (bt)
    (begin (def z100 int n)
     (= _ (hcall bt))
     (hreturn z100)))
   (def (capt)
    (do-handle
     (def (bt)
      (begin (def z500 int n)
       (= _ (hcall bt))
       (hreturn z500)))
     (begin
      (= _ (hcall capt))
      (out 10 n) (out 10 r0) (out 10 r1)
      (hreturn 1))))
   (begin
    (= r1 (calling 2)) ;; (fib 1)
    (return (+ r0 r1))))))
(exec 0 main
 (do-handle
  (hexec 7 bt (hreturn -2))
  (hexec 4 capt
   (begin (= _ (hcalling 5)) (hreturn -1))) ;; (bt)
  (def (bt) (hreturn -2))
  (def (capt) (begin (= _ (hcall bt)) (hreturn -1)))
  (= r (calling 1))))) ;; (fib 3)
```

A.3 SC-REs 言語

A.3.1 初期プログラム例

```
(prog
 (decl bt (handler int) 0)
 (decl capt (handler int) (hcalls bt) 0)
 (def _ int 0) (def r int 0)
 (def (fib n) (fn int int) (hcalls bt capt)
  (if (> 3 n) (begin (= _ (hcall capt)) (return 1))
   (begin (def r0 int 0) (def r1 int 0)
    (do-handle f
     (def (bt)
      (begin (def z100 int n)
       (= _ (hcall bt))
       (hreturn z100)))
     (def (capt)
      (do-handle c
       (def (bt)
        (begin (def z500 int n)
         (= _ (hcall bt))
         (hreturn z500)))
       (begin
        (= _ (hcall capt))
        (out 10 n) (out 10 r0) (out 10 r1)
        (hreturn 1))))
     (begin
      (= r0 (call fib (+ n -1)))
      (= r1 (call fib (+ n -2)))
      (return (+ r0 r1)))))))
 (exec 0 main
  (do-handle m
   (def (bt) (hreturn -2))
   (def (capt) (begin (= _ (hcall bt)) (hreturn -1)))
   (= r (call fib 3)))))
```

A.3.2 実行中プログラム例

```
(prog
 (decl bt (handler int) 0)
 (decl capt (handler int) (hcalls bt) 0)
 (def _ int 0) (def r int 0)
 (def (fib n) (fn int int) (hcalls bt capt)
  (if (> 3 n) (begin (= _ (hcall capt)) (return 1))
   (begin (def r0 int 0) (def r1 int 0)
    (do-handle f
     (def (bt)
      (begin (def z100 int n)
       (= _ (hcall bt))
       (hreturn z100)))
     (def (capt)
      (do-handle c
       (def (bt)
        (begin (def z500 int n)
         (= _ (hcall bt))
         (hreturn z500)))
       (begin
        (= _ (hcall capt))
        (out 10 n) (out 10 r0) (out 10 r1)
        (hreturn 1))))
     (begin
      (= r0 (call fib (+ n -1)))
      (= r1 (call fib (+ n -2)))
      (return (+ r0 r1)))))))
 (seeing (m 0)
  (hexec 7 bt (hreturn -2)))
```

```
(seeing (f 1)
 (hexec 6 bt
  (begin (def z100 int 3)
   (= _ (hcalling 7)) ; (bt)
   (hreturn z100))))
(seeing (c 3)
 (hexec 5 bt
  (begin (def z500 int 3)
   (= _ (hcalling 6)) ; (bt)
   (hreturn z500))))
(seeing (m 0)
 (hexec 4 capt
  (begin (= _ (hcalling 5)) (hreturn -1)))) ;; (bt)
(seeing (f 1)
 (hexec 3 capt
  (do-handle c
   (def (bt)
    (begin (def z500 int n)
     (= _ (hcall bt))
     (hreturn z500)))
   (begin
    (= _ (hcalling 4)) ;; (capt)
    (out 10 n) (out 10 r0) (out 10 r1)
    (hreturn 1)))))
(exec 2 fib (def n int 1)
 (begin (= _ (hcalling 3)) (return 1))) ;; (capt)
```

```
(exec 1 fib (def n int 3)
 (begin (def r0 int 1) (def r1 int 0)
  (do-handle f
   (def (bt)
    (begin (def z100 int n)
     (= _ (hcall bt))
     (hreturn z100)))
   (def (capt)
    (do-handle c
     (def (bt)
      (begin (def z500 int n)
       (= _ (hcall bt))
       (hreturn z500)))
     (begin
      (= _ (hcall capt))
      (out 10 n) (out 10 r0) (out 10 r1)
      (hreturn 1))))
   (begin
    (= r1 (calling 2)) ;; (fib 1)
    (return (+ r0 r1))))))
(exec 0 main
 (do-handle m
  (def (bt) (hreturn -2))
  (def (capt) (begin (= _ (hcall bt)) (hreturn -1)))
  (= r (calling 1))))) ;; (fib 3)
```