1L-5

無限ループを検出するインタプリタの実装手法の検討*

桜井 勇貴† 佐々木 晃† 雪田 修一†
法政大学 情報科学部†

1. 序論

関数型言語などの高階関数や再帰呼び出しを中心にプログラミングを行う言語では, プログラムの呼び出しパスが複雑になりやすく, 無限ループを生成しやすい. この時, 呼び出しパスがデバッグ時に非常に重要な意味をもつが, これを得るためには, デバッガなどを用いたステップ実行などによる手法しか存在せず, 内部的にこの事象を扱うことはできない. しかし, 一部の無限ループは, その性質から実行時に検出可能である. これらを内部的に表現することが可能となれば, テストケースの記述や複雑な停止条件の記述を除去することができる.

本研究では関数型言語における検出可能な無限ループを実行時に検出することを試みる. そして, 検出時に, プログラムの実行時エラーとして表現する方法を説明する.

2. 関連研究

関数型言語の型システムに用いられる単純型付ラムダ計算は停止性を持つが, 計算のための十分な表現力を備えていない. また, この型システムに再帰型を導入すると, 停止性が失われる. 関数型言語の停止性証明という点では,Size-Change-Principle[4] という原理が存在する. これは, 任意の関数において, 停止性を持つ関数は関数呼び出しの引数に無限の要素を与えると, 無限にデクリメントする関数の呼び出しパスを形成する性質である. これらは停止性を持つプログラムであることを証明する静的なアプローチである. 実行時の無限ループ検出を扱う研究としては, Bolt[3] が存在し, メモリ状態を用いて外部からターゲットプログラムの無限ループを扱う.

3. 無限ループの分類と定義

本研究では, 関数型言語において発生する無限ループをその性質から次の三種類に分類する.

- 循環する無限ループ
- 循環しない無限ループ
- 循環する関数呼び出しパス

“循環する無限ループ” とは, プログラム実行時の状態の列において, ある瞬間の状態と別の瞬間の状態が一致

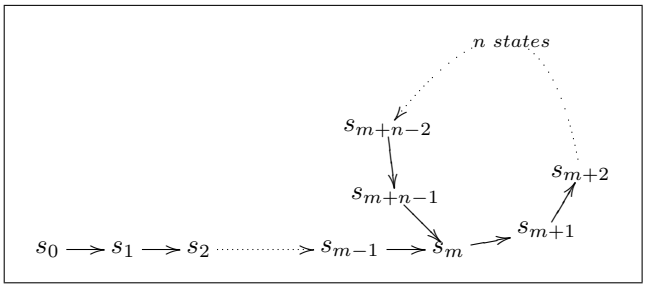

図 1: 循環する無限ループの状態遷移

するものを言う. つまり, ある実行時の状態において n 個後ろの状態と同じ状態が発生するということは, 数ステップ後に必ず自分自身と同じ状態を生成し, 無限ループとなる.“循環する無限ループ” のクラスの補集合として, “循環しない無限ループ” のクラスが考えられる. 本研究においてこの種類のループは扱わない.

“循環しない無限ループ” 場合でも, 関数呼び出しのパスを調べることで, 関数の呼び出しのスタックに循環構造を見つけることができる場合がある. プログラムの参照等価性が保証される時, 状態列の循環と同様に, 無限ループを構成していると言える.

4. 循環検出アルゴリズム

循環する無限ループは, ある有限個の状態列を無限に繰り返す. 図 1 のような状態列を形成する.

循環型の無限ループには, ループに入る前までの状態数 m および, ループに入った後の有限の状態数 d が存在し, 任意の a 回目のループについて, $m+ad \leq k, k \bmod d = 0$ となるような, k が常に存在する. このことから, 図 2 のような循環検出アルゴリズムが考えられる. 状態遷移のある n 番目の状態と $(n+k)$ 番目の状態同士を $k = 1, 2, 3, ...$ と順に比較していく方法である.

detect 関数は, 内部状態 $(n, skips, state_{last})$ を持つ. n と $skips$ の初期値は, 任意の整数を指定できるが, 簡単のため $n = 0, skips = 0$ で考える. $state_{last}$ の初期値は, 任意の $state$ の値と重複しないような値を指定する. 初期化後, 循環検出関数を状態遷移毎に呼び出す. 状態が一致していた時, 循環しているとみなし, **true** を返す.

5. インタプリタ

循環する無限ループを検出するためのインタプリタの実装について述べる. ここでは, プログラムの状態を

*A study of methodologies towards the implementation of interpreters detecting infinite loops
†SAKURAI Yuuki, SASAKI Akira, YUKITA Shuichi, Faculty of Computer and Information Science, Hosei University

detect $state_{current}$ =
  let $cond$ = **check** $(n, skips, state_{last}), state_{current}$ **in**
    if $cond$ **is** $\langle circular \rangle$ **then**
      true
      else $(n, skips, state_{last})$ **:=** $cond$ **; false**

check$(n, skips, state_{last}), state_{current}$ =
  if $n = 0$ **then**
    if $state_{last} = state_{current}$ **then**
      $\langle circular \rangle$
      else $(skips + 1, skips + 1, state_{current})$
  else $(n - 1, skips, state_{last})$

図 2: 循環検出関数 **detect** とその補助関数 **check**

捕縛するために継続を用い, 無限ループ時の脱出に限定継続を用いる. これらを用いるために, 次に述べる Meta Continuation Semantics を導入する.

5.1. Meta Continuation Semantics [1], [2]

下記の Semantics により継続および限定継続をプログラム上で扱うことができ, インタプリタが末尾形式で実装可能になる. $\rho[[x] \mapsto a]$ は, 環境 ρ の変数 $[\![x]\!]$ を a に書き換える. $E_1 \longrightarrow E_2|E_3$ は, if を表し, E_1 を評価した結果を元に, E_2 か, E_3 を評価する.

$$\begin{aligned}
\mathcal{E}: &\quad Exp \to Env \to Cont \to MCont \to Ans \\
f \in Proc: &\quad Value \to Cont \to MCont \to Ans \\
\kappa \in Cont: &\quad Value \to MCont \to Ans \\
\gamma \in MCont: &\quad Value \to Ans
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}[\![c]\!]\rho\kappa\gamma &= \kappa c\gamma \\
\mathcal{E}[\![x]\!]\rho\kappa\gamma &= \kappa(\rho[\![x]\!])\gamma \\
\mathcal{E}[\![E_1\ E_2]\!]\rho\kappa\gamma &= \mathcal{E}[\![E_1]\!]\rho(\lambda f\gamma'.\mathcal{E}[\![E_2]\!]\rho(\lambda a\gamma''.fa\kappa\gamma'')\gamma \\
\mathcal{E}[\![\lambda x.E]\!]\rho\kappa\gamma &= \kappa(\lambda\nu\kappa'\gamma'.\mathcal{E}[\![E]\!]\rho[[\![x]\!] \mapsto \nu]\kappa'\gamma')\gamma \\
\mathcal{E}[\![E_1 \to E_2|E_3]\!]\rho\kappa\gamma & \\
&= \mathcal{E}[\![E_1]\!]\rho(\lambda b\gamma'.b \to \mathcal{E}[\![E_2]\!]\rho\kappa\gamma'|\mathcal{E}[\![E_3]\!]\rho\kappa\gamma')\gamma
\end{aligned}$$

5.2. 脱出

無限ループからの脱出機構について述べる. 本研究では, 脱出に限定継続を用いる. 限定継続は, shift オペレータ及び, reset オペレータから構成される. これらオペレータは, それぞれ限定された Control Context の変数への束縛とその範囲の捕縛を行う. これら, shift/reset により脱出演算子を記述することが可能であり [1], それぞれ以下のように実装される.

$$\begin{aligned}
&\mathcal{E}[\![\xi k.E]\!]\rho\kappa\gamma \\
&= \mathcal{E}[\![E]\!]\rho[[\![k]\!] \mapsto (\lambda\nu\kappa'\gamma'.\kappa\nu(\lambda w.\kappa' w\gamma'))](\lambda x\gamma''.\gamma'' x)\gamma \\
&\mathcal{E}[\![\langle E \rangle]\!]\rho\kappa\gamma = \mathcal{E}[\![E]\!]\rho(\lambda x\gamma'.\gamma' x)(\lambda\nu.\kappa\nu\gamma)
\end{aligned}$$

5.3. 無限ループ検出アルゴリズムの実装

Meta Continuation Semantics により, インタプリタは常に末尾形式でプログラムを解釈する. 制御効果のある最小単位の状態を比較すればよいため, 関数適用の瞬間の継続, 関数とその引数の情報を関数評価毎に確保する. したがって, 下記のように関数適用である $\mathcal{E}[\![E_1\ E_2]\!]\rho\kappa\gamma$ に生成される継続に, 検出関数を挿入する. ここで $\mathcal{D}$ は循環検出関数であり, この関数は, 図 2 の関数 **detect** に対応する. この関数が循環を検出した時, 実行時エラーとして一番最後に囲われた **reset** までジャンプする. 脱出時のパラメータ $E_{escape\_parameter}$ を用意し, プログラム中に適宜この値を書き換えられるようにする. 脱出時は最後に書き換えられた値とともに, 例外オブジェクトとしての値を返す.

$$\begin{aligned}
\mathcal{E}[\![E_1\ E_2]\!]\rho\kappa\gamma = \mathcal{D}(E_1, E_2, \rho, \kappa, \gamma) \longrightarrow & \\
\gamma\ (\textbf{“exception”}\ E_{escape_parameter}) & \\
|\ \mathcal{E}[\![E_1]\!]\rho(\lambda f\gamma'.\mathcal{E}[\![E_2]\!]\rho(\lambda a\gamma''.\ fa\kappa\gamma'')\gamma &
\end{aligned}$$

6. 結論

本研究により, ある一定の種類の無限ループが検出可能であるということが確認でき, 循環する無限ループに対して, プログラミング言語処理系の側からのアプローチが可能であることがわかった. 実用的なプログラミング言語処理系への実装のためには, 5 節で述べたインタプリタを VM の実装へ拡張し直すなどの修正が必要である.

7. 謝辞

本研究の一部は, 科研費 (課題番号 23700043, 23500034) の助成を受けた.

参考文献

[1] O. Danvy, A. Filinski, *A Functional Abstraction of Typed Contexts*, DIKU Technical Report 89/12, Copenhagen, Denmark, 1989.

[2] O. Danvy, A. Filinski, *Abstracting Control*, Proceedings of the 1990 ACM Conference on Lisp and Functional Programming, pp. 151-160, 1990.

[3] M. Kling, S. Misailovic, M. Carbin and M. Rinard. *Bolt: On-Demand Infinite Loop Escape in Unmodified Binaries*, Proceedings of the ACM international conference on Object oriented programming systems languages and applications, pp. 431-450, 2012.

[4] C. Lee, N. Jones, and A. Ben Amram. *The Size-Change Principle for Program Termination*, Proceedings of the 28th ACM Symposium on Principles of Programming Languages. pp. 81-92, 2001.