神奈川大学理学部　情報科学科　馬谷研究室

Redex 上の Scheme インタプリタにおける ヒープの有限化による抽象解釈の実現

森田 翔大 201803690

1　はじめに

本研究室では，Scheme 言語の抽象解釈器の構築を目指し，Scheme サブセットを Redex に実装すること，実装されたものの最終処理をヒープ化すること，そのヒープ化されたものを有限化することの 3 点に取り組んでいる．

本研究では，ヒープの有限化について取り組む．

Scheme サブセットを Redex 上で表した SCM のプログラム [4] は，スタックを無制限に増やすこと，無制限にクロージャを作ること，置換によってソースプログラムには現れない無制限の数の項を作ることができる．一方，定義した SCM 言語に継続のヒープ化をした SCMt*のプログラム [5] では，すべてのスタックフレームとクロージャがヒープに計上される．数字を除いて，プログラムのソースに存在しない表現がプログラムの実行中に発生することはない．

そのためヒープを有限化することで，抽象解釈機の完成に繋がる．

2　背景

ヒープとは，アドレスと値を関連付けたデータ構造のことである．参考文献 [5] より Scheme インタプリタの内部構造の内，無制限にサイズが大きくなる可能性があるデータをヒープを用いて表した．

そのヒープが有限でないとアドレスと値の組み合わせが無限にできてしまうため抽象化をすることができない．そのため，ヒープを有限化する必要性がある．文献 [4, 5] の研究で Scheme サブセットを Redex に実装し，それをヒープ化した．ヒープを含む状態を Redex で定義すると以下のようになる．

```
(define-extended-language SCMt* SCMt
    (C () | (F A))
    (Σ ((A U) ...))
    (U v | C))
```

SCMt を SCMt*に上書きすることを-->rSt*とする．SCMt*では，SCMt 言語の継続の文法を上書きしている．継続は空か，継続の残りの部分へのポインタを持つ単一フレームになる．ヒープが拡張され，値または継続にアドレスを関連付けられるようになる．

また，ヒープを拡張することでデータを入れる準備をするため，SCMt*言語を拡張する必要がある．

```
(define-extended-language SCMt' SCMt*
    (Σ ::= any))
(define-metafunction/extension alloc* SCMt'
    alloc' : s -> (A ...))
```

SCMt*を SCMt' に上書きすることを-->rSt'とする．

-->rs' は-->rs*と全く同じ計算をするが，表現が異なるということである．-->rs' は，ヒープを Racket のハッシュテーブルとしてモデル化している．

3　問題点

プログラム解析とは，プログラムを実行したときに何が起こるかを実際に実行することなく発見することである．そのためには，無限ループに陥るようなソースコードを解析するような場合であっても，

解析自体は必ず停止する必要がある．

プログラムを解釈する-->rSt*（あるいは-->rSt'）が永遠に実行されるとき，無限大のメモリを確保する方法と，任意の大きさの数を計算する方法が用いられている．つまり，プログラムの実行を有限化するには，そのメモリと数値を有限とすればよい．

4 ヒープの有限化

ヒープを有限化するために数の抽象化をする．ここでは抽象化するものの一例として数を用いているが，健全で有限な抽象度であれば何でもよい．

```
(define-extended-language SCMs^ SCMs'
    (N ::= .... num))
```

数の集合を拡張して，任意の数を表すと解釈すべきnum という新しい数を含める．

次にメモリに移り，割り当て関数を有限のアドレスセットしか生成しないものに置き換える．アロケーションを細かくする簡単な方法として，alloc'の結果を 1 つのシンボルに切り詰める方法がある．

```
(define-metafunction/extension alloc' SCMs^
    alloc'^ : ((C K) Σ) -> (A ...))
(define-metafunction SCMs^
    alloc^ : ((C K) Σ) -> (A ...)
    [(alloc^ s)
     (A ...)
     (where ((A _) ...) (alloc'^ σ))])
```

if0 と抽象化 num の組み合わせで，num で抽象化されたすべての動作をカバーするように意味論を設計する．数値は 0 でも 0 以外でもよいので，条件式が num が存在したときに両方の分岐をとるように場合分けをする．

```
(define -->vs^
    (extend-reduction-relation
     (-->vs'/Σ alloc^ ext-Σ lookup-Σ)
     SCMs^
     (--> (((O N ...) K) Σ)
          ((N_1 K) Σ)
          (judgment-holds (δ
^ (O N ...) N_1))
          δ)
     (--> (((if0 num C_1 C_2) K) Σ)
          ((C_1 K) Σ)
          if0-num-t)
     (--> (((if0 num C_1 C_2) K) Σ)
          ((C_2 K) Σ)
          if0-num-f)))
```

5 まとめ

ここでは，Scheme サブセットを Redex 上でヒープを利用し表したものの有限化をした．有限化をしたために，プログラムが永遠に実行されるという懸念は解消された．しかし，有限化をした後の評価をしていないため，健全性が保証されていないという課題が残っている．

参考文献

[1] Robert Bruce Findler，Casey Klein，Burke Fetscher，Matthias Felleisen，Redex: Practical Semantics Engineering. https://docs.racket-lang.org/redex/index.html

[2] David Van Horn, An Introduction to Redex with Abstracting Abstract Machines (v0.6). https://dvanhorn.github.io/redex-aam-tutorial/

[3] Jacob Matthews．Robert Bruce Findler，An Operational Semantics for Scheme，2007.

[4] 武田雄一郎，Scheme サブセットの操作的意味論の Redex による実装，2021 年度神奈川大学理学部情報科学科卒研要旨集，2022.

[5] 山本怜，Redex 上の Scheme インタプリタにおける抽象解釈のための継続のヒープ化，2021 年度神奈川大学理学部情報科学科卒研要旨集，2022.