神奈川大学理学部　情報科学科　馬谷研究室

Racket 言語のサブセットのコンパイラにおける クロージャ変換の効率化

横田 政憲 202103346

1　はじめに

書籍 [1] では，Racket 言語のサブセットから x86 アセンブリ言語へのコンパイラの実装を行っている．コンパイラは Racket 言語で実装されており，コンパイルをいくつかのステップに分けて行うためのパスを実装し，x86 アセンブリ言語への変換を行う．

本研究では，[1] で実装されているパス（コンパイルを段階的に処理していくための関数）のクロージャ変換を行うパスについて改善を行い，同じインタプリタで実行した場合における改善後の出力プログラムの処理速度の高速化を達成した．

既存の手法ではすべての関数に対してクロージャ変換を行うようになっているが，改善後はクロージャ変換が必要な関数に対してのみ行うようにし，効率化を行った．

2　対象言語

今回扱う Racket 言語のサブセットの一部を以下に示す．

```
type ::= Integer | (Vector type*)
 exp ::= int | var | (let ([var exp]) exp)
       | (if exp exp exp) | (vector exp*)
       | (vector-ref exp int) | (exp exp...)
       | (lambda:([var:type]...):type exp)
       | (fun-ref label int)
       | (closure exp*)
 def ::= (define (var [var:type]...):type exp)
```

type は型の種類であり，exp は式，def は関数定義である．プログラムは任意の数の関数定義と評価する式で構成されている．

3　既存の実装手法

クロージャ変換を行う convert_to_closures パスでは，すべての関数に対してクロージャ変換を行う．クロージャ変換を行う必要がない関数を含むプログラムの例を以下に示す．

```
(define (fun [n:Integer] [s:Integer]):Integer
  (if (eq? n 0)
      s
      (fun (- n 1) (+ n s))))
(fun 3 0)
```

このプログラムに対し convert_to_closures パスでクロージャ変換を行った結果は以下のようになる．

```
(define (fun [fvs:_] [n:Integer] [s:Integer]):Integer
  (if (eq? n 0)
      s
      (let ([clos1 (closure (list (fun-ref fun 2)))])
        ((vector-ref clos1 0) clos1 (- n 1) (+ n s)))))
(let ([clos2 (closure (list (fun-ref fun 2)))])
  ((vector-ref clos2 0) clos2 3 0))
```

このパスの問題点は，クロージャ変換を行う必要がない関数に対してもクロージャ変換を行ってしまうことである．

クロージャを生成する場合とクロージャを生成しない場合や，クロージャの関数ポインタから関数を呼び出す場合と直接関数を呼び出す場合では処理速度に差がある．処理速度を向上させるために，出力プログラムが必要以上にクロージャを生成せず，関数呼び出しについても，できるだけ直接関数を呼び出すように改善をする．

4　提案手法

本研究では，2 つの最適化を行った．

1. クロージャ変換が必要な関数に対してのみ行うようにし，クロージャの生成を減らすクロージャ変換の最適化．
2. クロージャの関数ポインタから関数を呼び出す場合で，どのクロージャを用いているかを特定できる場合，その呼び出しを直接関数を呼び出す形にするクロージャを用いた関数呼び出しの最適化．

クロージャ変換の最適化について，既存の convert_to_closures パスを改善し，エスケープする関数に対してのみクロージャ変換を行う convert_to_closures+ パスの実装を行った．エスケープする関数としない関数

を含むプログラムの例を以下に示す．

```
(define (fun [n:Interger] [s:Integer]):Integer
  (if (eq? n 0)
      s
      (fun (- n 1) (+ n s))))
(define (add1 [x:Integer]):Integer
  (+ x 1))
(let ([h add1])
   (fun (h 3) 0)))
```

convert_to_closures+ パスで，エスケープする関数に対してのみクロージャ変換を行った結果は以下のようになる．

```
(define (fun [n:Interger] [s:Integer]):Integer
  (if (eq? n 0)
      s
      (fun (- n 1) (+ n s))))
(define (add1 [fvs:_] [x:Integer]):Integer
  (+ x 1))
(let ([h (closure (list (fun-ref add1 1)))])
   (fun ((vector-ref h 0) h 3) 0)))
```

クロージャ変換が必要ないエスケープしない関数には，クロージャ変換を行わないようにする．let 式の第 2 引数と関数呼び出しの引数をチェックし，そこに関数が存在しているなら，その関数は変数に格納される可能性があるため，エスケープする関数である．

クロージャを用いた関数呼び出しの最適化について，クロージャの関数ポインタからの呼び出しを直接呼び出す形にできるかを判断し，可能な限り直接関数を呼び出す形に変換を行う optimize_know_calls パスの実装を行った．convert_to_closures+ パスでクロージャ変換を行った結果のプログラムの末尾 2 行を optimize_know_calls パスで最適化した場合を以下に示す．

```
(let ([h (closure (list (fun-ref add1 1)))])
   (fun (add1 h 3) 0)))
```

let 式の第 2 引数がクロージャであれば，その let 式の第 1 引数を用いて関数呼び出しが行われている部分を，直接関数を呼び出す形に変換する．

5 実装

convert_to_closures+ パスでは，最初にエスケープする関数のリストを生成し，既存の convert_to_closures パスのクロージャ変換を行う部分を，生成したエスケープする関数のリストに含まれていればクロージャ変換を行い，そうでなければクロージャ変換を行わないようにする．

optimize_know_calls パスでは，let 式の第 2 引数がクロージャである場合，let 式の第 1 引数を用いて関数呼び出しを行っている部分が，第 2 引数のクロージャに対応付いた関数を呼び出していることを特定できるため，関数ポインタからの呼び出しから直接呼び出す形にする．

6 評価

すべての関数に対してクロージャ変換を行う convert_to_closures パスと，必要な関数に対してのみクロージャ変換を行う convert_to_closures+ パスの，2 つの出力プログラムを同じインタプリタで実行したときの実行時間の比較と，optimize_know_calls パスを使用しなかったとき（convert_to_closures+ パスの出力プログラム）と使用したときの 2 つの出力プログラムを，同じインタプリタで実行したときの実行時間を比較する．

インタプリタがカウントした直接の関数呼び出し（direct-call），クロージャの関数ポインタからの関数呼び出し（indirect-call），クロージャの生成（closure）と time-apply 関数で測定した 10 回の結果の平均実行時間を以下の表に示す．

表 1 convert_to_closures パスの改善前と改善後の比較

入力プログラム	パス	direct-call	indirect-call	closure	平均実行時間（ms）
A	convert_to_closures	0	100001	100001	8478.3
	convert_to_closures+	100001	0	0	5076.4
B	convert_to_closures	0	242785	242785	10428.7
	convert_to_closures+	242785	0	0	6291

表 2 optimize_know_calls パスの使用前と使用後の比較

入力プログラム	パス	direct-call	indirect-call	closure	平均実行時間（ms）
C	convert_to_closures+	242785	242785	242785	16522.8
	optimize_know_calls	485570	0	242785	14991.6

既存の convert_to_closures パスと改善後の convert_to_closures+ パスの平均実行時間を比較すると，入力プログラム A と B どちらとも実行時間が約 0.6 倍に短縮された．また，optimize_know_calls パスを使用しなかった場合と使用した場合を比較すると，実行時間が約 0.9 倍に短縮された．

7 まとめ

本研究では，すべての関数に対してクロージャ変換を行うパスを改善し，クロージャ変換が必要な関数に対してのみ行うパスを実装した．改善後のパスの出力プログラムは，改善前のより処理速度が向上しており，改善したパスの効率化が実現していたことを示すことができた．

参考文献

[1] Jeremy G. Siek, *Essentials of Compilation: An Incremental Approach in Racket*, The MIT Press, 2023.