4K-05

Racket 上の多段階計算に基づく抽象解釈器の実装

帖佐 龍弥 †　　馬谷 誠二 ‡
神奈川大学 †　　神奈川大学 ‡

1　はじめに

抽象解釈 [1] は，主に静的解析に用いられるプログラム解析の手法の一つである．プログラムを抽象的かつ健全なモデルで表現し，それを用いてプログラムの挙動を近似的に評価する．抽象解釈を高速化する方法として，多段階計算 [2] に着目する．多段階計算は動的にコード生成や実行を行う技法であり，これをインタプリタに応用する事でコンパイラのようにより高速に動作する同等のプログラムを生成することができる．本研究では，Racket 上で実装した簡易なインタプリタに対して，多段階計算による高速化を行った後に抽象解釈器を構成した場合と，抽象解釈器を構成した後に多段階計算を用いて高速化した場合を比較する．

2　抽象解釈

抽象解釈 [1] は，プログラム解析の手法の一つである．プログラムがある性質を満たすか分析する際に，解析に必要な要素だけを取り出して元のプログラムに対して健全なモデルを構築し，それを用いて近似的に評価する．本研究では，加算，減算，乗算と，if 文，while 文からなる単純な While 言語に対して抽象化を行う．

最初に，While 言語に対する抽象解釈器を構成する．今回は，While 言語上の数値に対して，その符号に着目し，正，負，0 のいずれとなりえるかを，$\bot$, $\{+\}$, $\{-\}$, $\{0\}$, $\{+,0\}$, $\{+,-\}$, $\{0,-\}$, $\top$ の 8 つの抽象値で表す．Racket 上では，数値と真偽値の抽象表現と，それらへの演算を実装する．

ここでは，数値演算の解釈について詳しく述べる．加算，減算，乗算をそれぞれ抽象値同士の 2 項演算 $+_a$，$-_a$，$*_a$ で近似する．$+_a$ を例にすると，正の数同士の和は必ず正であるため，$\{+\}\ +_a\ \{+\}$ は $\{+\}$ とする．また，正の数と負の数の和は正，負，0 のいずれもあり得るため，$\{+\}\ +_a\ \{-\}$ は $\top$ とする．

次に，while 文の解釈について詳しく述べる．while 文の解釈は，ループ本体の文に従って変数環境を繰り返し拡大し，それが変化しなくなった時点で終了する．この手法は，0 回以上のあらゆる繰り返し実行の結果を収集する．例として $\mathtt{x} = 2$, $\mathtt{y} = 1$, $\mathtt{n} = 3$ の条件のもとで次の冪乗を計算するプログラムを解釈してみる．

```
(while (>= n 1)
  (set! y (* y x))
  (set! n (- n 1)))
```

抽象解釈器ではコードの解釈に際し，変数名と抽象値が対応するような変数環境を用いる．(set! x e) を解釈すると，e の数値を抽象化した値で既存の x の抽象値を拡大する．開始時の変数環境を抽象化すると，x，y，n はいずれも $\{+\}$ となる．これをプログラムに従って一回拡大すると，n は $\top$，x と y は $\{+\}$ となる．この状態の元でもう一度拡大すると，n は $\top$，x と y は $\{+\}$ となる．2 回目の拡大前後で変数環境は等しく，これ以上拡大を繰り返しても変化は起こらないため，while の解釈は終了となる．

3　抽象解釈の多段階化

多段階計算 [2] は動的にコード生成や実行を行う技法である．これをインタプリタに応用することで，コンパイラのようにソースプログラムをホスト言語のコードへと変換することができる．また，コード生成時に可能な計算を済ませておくことで，より高速に実行可能なコードを生成することができる．Racket では動的にコード生成する方法としてマクロが利用可能である．

本研究では，抽象解釈の実行を高速化するために，多段階計算の技法を抽象解釈器に応用する [3]．多段階化した抽象解釈器では，1 段階目にコード生成を行い，2 段階目に生成したコードの実行を行う．通常のインタプ

Implementation of an Staged Abstract Interpreter in Racket

†Tatsuya Chousa, Kanagawa University
‡Seiji Umatani, Kanagawa University

リタと同様に，コード生成の時点で計算可能な部分は計算を完了させることで，実行にかかる時間を短縮できる．

今回の Racket による実装では，1 段階目の変数環境中で束縛されていない変数の値は，2 段階目にだけ用いることができるものとする．1 段階目の変数環境では，各変数は抽象値または抽象値へと評価されるコードに束縛される．たとえば，x が未定義であるとして，While のプログラム (+ (+ 2 3) x) を解釈する場合，(+ 2 3) については 1 段階目で計算が可能である．(+ 2 3) を抽象表現に変換すると $\{+\} +_a \{+\} = \{+\}$ であるから，生成されるコードは (a+ {+} (a x)) となる．a は数値を抽象値に変換する関数である．

抽象解釈における while 文の処理は，変数環境を繰り返し拡大し，それが変化しなくなった時点で終了する．しかし，多段階化を行なった場合においては，1 段階目の変数環境で未束縛の変数を含む式がある場合では，その式の抽象値が変化しないことを判断することは一般的には不可能である．そのため，1 段階目は while 文を実行せず，すべて 2 段階目に行うものとする．この方式では，1 段階目に行える計算がないためコード実行は高速化していない．

4 多段階計算の抽象解釈化

多段階計算を組み込んだ抽象解釈器を構成する別の方法として，まず先に通常のインタプリタを多段階計算を用いて高速化し，それを基に抽象解釈器を構成することを考える．

最初に，抽象化されていない通常のインタプリタを多段階計算を用いて高速化したものを構成する．ここでもコード生成時に可能な計算は行い，計算できない部分として，値が束縛されていない変数を想定する．また，1 段階目で繰り返しの回数を計算できる while 文については，展開した形にする．例えば，2 節で挙げた冪乗計算のコードで，x と y は未束縛として解釈すると，最終的に変数環境中で y はコード (* (* (* y x) x) x) に束縛される．

次に，多段階計算を用いて高速化したこのインタプリタを基に抽象解釈器を構成する．これは全体で 3 つの段階を持つ．1 段階目に数値とその演算からなるコードを生成する．ここで計算可能な部分の処理やループの展開を済ませる．2 段階目には，1 段階目で生成されたコードを抽象値とその演算へと変換する．3 段階目には，そのコードの実行を行う．新たに構成した抽象解釈器は，3 節の抽象解釈器と比較しコード生成にかかるプロセスが増えているが，生成されるコードはより高速に実行でき，抽象化の精度も向上する．

具体的には，加算，減算，乗算については計算した後に抽象値にすることで，抽象値同士の計算に伴う正確さの低下を抑制できる．たとえば (+ 3 -2) を解釈する場合，3 節までの抽象解釈器では $\{+\} +_a \{-\}$ となり，これを計算した値の符号は不明であるため $\top$ としていた．しかし，1 段階目の計算によって (+ 3 -2) が 1 となっていれば抽象値が $\{+\}$ と直ちに判定できる．

while 文の処理を比較すると，3 節の抽象解釈器は，コード生成の段階での変数環境に数値ではなく抽象値を記憶させていたために，繰り返しの回数を計算することが不可能になっていた．しかし，新たに構成した抽象解釈器では，1 段階目に数値のまま計算を行うためにループを展開することが可能であり，実際のループ回数に基づいた解釈が可能になる．その結果，必要以上に抽象値を拡大する必要がなくなり，変数環境の比較や結合といった特有の操作が不要となる．ループ回数が計算不可能な場合は，3 節と同様に実行時に計算する．

5 まとめ

本研究ではで多段階計算の技法を取り入れた抽象解釈器を，まず先に抽象表現に変換しコードを生成する方法と，数値表現のままコードにした後に抽象表現に変換する方法の 2 通りで構成した．後者は前者よりコード生成に必要な工程が多いが，生成されたコードの実行速度や抽象化の精度では前者よりも優れているといえる．

参考文献

[1] Xavier River and Kwangkeun Yi, *Introduction to static analisys : an abstract interpretation perspective,* Cambridge, MA : The MIT Press, 2020.

[2] Walid Taha, *A Gentle Introduction to Multi-stage Programming,* Domain-Specific Program Generation, 2004

[3] Guannan Wei, Yuxuan Chen, and Tiark Rompf, *Staged abstract interpreters: fast and modular whole-program analysis via meta-programming,* Proc. ACM Program. Lang. 3, OOPSLA, Article 126 (October 2019)