第3章 Provably Correct Implementation

November 12, 2008

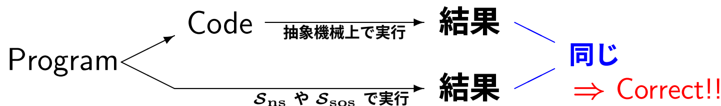
プログラミング言語の semantics

- **プログラミング言語の** semantics **を形式的に定めることが役立つのは，その言語を実装をするとき**
- **さらに，実装の正当性が示せる**！
- While **言語を，抽象機械のアセンブラのコードに変換して，その変換の正当性を示す**

abstruct machine ··· **抽象機械，抽象計算機**
instruction ··· **命令**
correctness ··· **正しさ，正確さ，正当性**

Correctness ?

- どうなっていたら，その実装は正しいと言えるか ?
- プログラムをコードに変換し，
- そのコードを抽象機械上で実行した結果，
- 2 章の $\mathcal{S}_{\rm ns}$ や $\mathcal{S}_{\rm sos}$ で得られる結果と同じになるなら OK



抽象機械

- **抽象機械** AM **の** configuration **の形は，**$\langle c,\ e,\ s \rangle$
- $c \cdots$ **実行される命令またはコードの列**
- $e \cdots$ **計算のスタック**
- $s \cdots$ storage

storage $\cdots$ **外部記憶装置** (?)

Evaluatioin stack

- 計算のスタック (?)
- 算術式や論理式の計算をするときに使われる
- 形式的には値のリストで，

$$\mathrm{Stack} = (\mathrm{Z} \cup \mathrm{T})^*$$

$$e \in \mathrm{Stack}$$

Storage

- シンプルさのために，ここでは state と同じようなものということにする
- つまり，$s \in \mathrm{State}$
- 変数の値を保持するものとして使われる

命令，configuration

- AM の命令 → p.64 参照
- syntactic category は Code で，c は，Code のメタ変数
- 結局，$\langle c,\ e,\ s\rangle \in \mathrm{Code} \times \mathrm{Stack} \times \mathrm{State}$
- 最終的な configuration とは？
  $\Rightarrow$ コードの要素が空のもの $\cdots\langle \epsilon,\ e,\ s\rangle$ の形

component ⋯ 成分，要素

遷移関係 $\triangleright$

- 抽象機械の命令の意味は，operational semantics で与えられる
- 遷移関係 $\triangleright$ は，命令をどのように実行するかを定める

$$\langle c,\ e,\ s\rangle \triangleright \langle c',\ e',\ s'\rangle$$

- 1 ステップの実行は，$\langle c,\ e,\ s\rangle$ を，$\langle c',\ e',\ s'\rangle$ という configuration に変換する
- 各関係は，p.65 の表 3.1 の公理参照

命令

- 計算のスタックを変える命令
  ⇒ 普通の算術演算 + 論理演算の他に，6 つ
- コントロールの流れを変える命令
  ⇒ $\mathtt{BRANCH}(c_1, c_2)$ と $\mathtt{LOOP}(c_1, c_2)$ の 2 つ

halt ⋯ 停止する，止まる

Computation sequence

- 2 章の導出列に対応するもの：計算列
- 命令列 c，storage s を与えられての計算列
  - 有限の列 $\gamma_0, \gamma_1, \gamma_2, \cdots, \gamma_k$ ⇔ 終了する
  - 無限の列 $\gamma_0, \gamma_1, \gamma_2, \cdots$ ⇔ ループする
  - どちらも， $\gamma_0 = \langle c,\ \epsilon,\ s \rangle$
- 最初の configuration の計算のスタックは空

例 & 練習問題

- **例** 3.1 : **終了する計算列の例**
- **例** 3.2 : **ループする計算列の例**
- **練習** 3.3
  どういう計算をする関数のコードか ?

例 & 練習問題

- 例 3.1：終了する計算列の例
- 例 3.2：ループする計算列の例
- 練習 3.3
  どういう計算をする関数のコードか？

  $\Rightarrow$ x を y で割った商と剰余を計算するコード

AM の性質

- 抽象機械に対して定義した semantics は，個々の命令の実行に着目している
  $\Rightarrow$ 2 章でやった structural operational semantics に近い
- コード生成の正当性の証明をしようとすると，sos で成り立ったことに似たような結果が必要になる
- 証明のやり方も似ている

AM での証明法

- 計算列の長さによる帰納法
  1. その性質が長さ 0 の全ての計算列に対して成り立つことを示す
  2. その性質が長さ k 以下の全ての計算列に対して成り立つことを仮定して，長さ k+1 の計算列に対しても成り立つことを示す
- 帰納のステップは，大抵コードの先頭の命令での場合分けで OK

怒濤の練習問題 (その 1)

- 練習 3.4〜3.6 を証明すると，言えること
- 練習 3.4
  コードの要素とスタックの要素は，マシンの挙動を変えることなく，のばすことが出来る
- 練習 3.5
  合成の列の計算は，2 つの部分に分けられる
- 練習 3.6
  $\langle c, e, s \rangle$ から始まる計算列は一意に定まる

実行関数 $\mathcal{M}$

- **命令列の意味**
  ⇒ State から State への部分関数として定義する

$$\mathcal{M} : \mathrm{Code} \rightarrow (\mathrm{State} \hookrightarrow \mathrm{State})$$

$$\mathcal{M}[\![c]\!]s = \begin{cases} s' & \langle c,\ \epsilon,\ s\rangle \rhd^{*} \langle \epsilon,\ e,\ s'\rangle \\ \underline{\mathrm{undef}} & \mathrm{otherwise} \end{cases}$$

- **練習** 3.6 **により，この関数は** well-defined

怒濤の練習問題 (その 2)

- AM は伝統的なマシンアーキテクチャとはかけ離れているように見える

怒濤の練習問題 (その 2)

- AM は伝統的なマシンアーキテクチャとはかけ離れているように見える
  ⇒ 練習 3.7〜3.9 で，そのギャップを徐々に埋める

怒濤の練習問題 (その 2)

- AM は伝統的なマシンアーキテクチャとはかけ離れているように見える
  ⇒ 練習 3.7〜3.9 で，そのギャップを徐々に埋める
- 練習 3.7： 変数の値を番地で参照
- 練習 3.8： プログラムカウンタとラベルを導入
- 練習 3.9： 練習 3.8 の JUMP 命令を絶対番地にする

get rid of · · · 取り除く	absolute address · · · 絶対番地

コード生成 (式)

- 抽象機械のコードをどのように生成するかを考える
- まずは，算術式 & 論理式のコード生成
  ・・・ 計算のスタックを利用して実行する
- p.70 の表 3.2 参照
- ちゃんと compositional な定義

例 & 練習問題

- **例** 3.10
  x+1 **を計算するコード生成の例**
- **練習** 3.11
  - $(a_1 + a_2) + a_3$ **と** $a_1 + (a_2 + a_3)$ **を計算した結果が等価ということは明らか**
  - **なのに，これをコードに変換すると異なるものになってしまう！という話**
  - **ただし，挙動は同様**

コード生成 (statement)

- statement のコード生成

  $\cdots a$ には $\mathcal{CA}$, b には $\mathcal{CB}$ を使う
- p.71 の表 3.3 参照
- ちゃんと compositional な定義

例 & 練習問題

- **例** 3.12 : **階乗の** statement **のコード生成**
- **練習** 3.13 : **コード生成と，**AM **のコードの実行の練習**
- **練習** 3.14

  `repeat` S `until` b **で拡張した** While **のコード生成**
- **練習** 3.15

  `for` $x := a_1$ `to` a_2 `do` S **で拡張した** While **のコード生成**

意味関数 $\mathcal{S}_{\mathrm{am}}$

- statement S の意味は？

- つまり，$\mathcal{S}_{\mathrm{am}} : \mathrm{Stm} \longrightarrow (\mathrm{State} \hookrightarrow \mathrm{State})$ という，次のように定義される関数で表される

$$\mathcal{S}_{\mathrm{am}}[\![S]\!] = (\mathcal{M} \circ \mathcal{CS})[\![S]\!]$$

練習問題

- 練習 3.16
  AM_1 のコードを生成するように変更
  & 練習 1.1 のコードを生成して実行

- 練習 3.17
  AM_2 のコードを生成するように変更
  & 練習 1.1 のコードを生成して実行
  (ラベルが重複しないように気を付けること)

正当性 (式)

- **まずは，式のコード生成の正当性**
- **補題** 3.18 : **算術式のコード生成の正当性の証明**
  $\Rightarrow a$ **の構造に関する帰納法**
- **練習** 3.19 : **論理式のコード生成の正当性の証明**
  $\Rightarrow b$ **の構造に関する帰納法**

正当性 (statement)

- statement のコード生成の正当性
- **定理** 3.20 : $\mathcal{S}_{\mathrm{ns}}[\![S]\!] = \mathcal{S}_{\mathrm{am}}[\![S]\!]$
  $\Rightarrow$ 下の補題 3.21 と 3.22 から証明される
- **補題** 3.21 : ns で S の実行が終了
  $\Rightarrow$ S のコードの実行も終了して，結果が一致
- **補題** 3.22 : S のコードの実行が終了
  $\Rightarrow$ ns における S の実行も終了して，結果が一致

証明手法

- 実装の正当性の証明に用いた証明のまとめ
  - ns $\longrightarrow$ am で用いたのは，導出木の構造による帰納法
  - am $\longrightarrow$ ns で用いたのは，計算列の長さによる帰納法
- $\mathcal{S}_{\mathrm{ns}}$ と $\mathcal{S}_{\mathrm{sos}}$ の等価性の証明 (2.3 節参照) に似ている
- 何か加えた言語や，違うマシン言語にこの手法を適用するときは，気をつけること！

怒濤の練習問題 (その 3)

- **練習** 3.23
  $\mathcal{CS}'[\![\mathtt{skip}]\!] = \epsilon$ **とすると，定理** 3.20 **の証明は面倒になるか** ?
- **練習** 3.24
  $\mathtt{repeat}\ S\ \mathtt{until}\ b$ **で拡張された言語への，定理** 3.20 **の証明の拡張**
- **練習** 3.25
  AM_1 **の正当性の証明** (env **を作る上で必要な仮定は** ?)

$\mathcal{S}_{\mathrm{sos}}[\![S]\!] = \mathcal{S}_{\mathrm{am}}[\![S]\!]$?

- **定理** 3.20 **で示したのは，**$\mathcal{S}_{\mathrm{ns}}[\![S]\!] = \mathcal{S}_{\mathrm{am}}[\![S]\!]$
- **すでに** $\mathcal{S}_{\mathrm{ns}}[\![S]\!] = \mathcal{S}_{\mathrm{sos}}[\![S]\!]$ **は証明済み**

$\mathcal{S}_{\mathrm{sos}}[\![S]\!] = \mathcal{S}_{\mathrm{am}}[\![S]\!]$?

- **定理** 3.20 **で示したのは，**$\mathcal{S}_{\mathrm{ns}}[\![S]\!] = \mathcal{S}_{\mathrm{am}}[\![S]\!]$
- **すでに** $\mathcal{S}_{\mathrm{ns}}[\![S]\!] = \mathcal{S}_{\mathrm{sos}}[\![S]\!]$ **は証明済み**

$$\Downarrow$$

- $\mathcal{S}_{\mathrm{sos}}[\![S]\!] = \mathcal{S}_{\mathrm{am}}[\![S]\!]$ **ということは明らか**

けど・・・・・・

- 「直接証明したほうがいいんじゃないの？」

けど……

- 「直接証明したほうがいいんじゃないの？」
- 「どっちも実行の個々のステップに着目しているんだし」

けど……

- 「直接証明したほうがいいんじゃないの ?」
- 「どっちも実行の個々のステップに着目しているんだし」
- ということは，思うかも

ということで

- 別の証明方法

ということで

- **別の証明方法**
- $\mathcal{S}_{\mathrm{sos}}[\![S]\!] = \mathcal{S}_{\mathrm{am}}[\![S]\!]$ ということを**直接**証明してみよう

Bisimuration relation

- 関係 R において，$a\ R\ b$ のとき，
  - a から a' に遷移可能
    $\Rightarrow b$ から同じ方法で遷移可能な b' が存在し，$a'\ R\ b'$
  - b から b' に遷移可能
    $\Rightarrow a$ から同じ方法で遷移可能な a' が存在し，$a'\ R\ b'$

が成り立つならば，bisimuration relation と呼ばれる

関係≈

- sos の configuration と，AM の operational semantics の configuration の間に，bisimuration relation ≈ を以下のように定義する

$$
\begin{aligned}
\langle S,\ s\rangle &\approx \langle \mathcal{CS}[\![S]\!],\ \epsilon,\ s\rangle \\
s &\approx \langle \epsilon,\ \epsilon,\ s\rangle
\end{aligned}
$$

証明の概要 (練習問題)

- sos の 1 ステップの実行に対応する AM の計算列が存在する

  ⇒ 練習 3.26

- AM での計算のスタックが空な configuration から，計算のスタックが空な別の configuration への計算列に対応する sos の 1 ステップの実行が存在する

  ⇒ 練習 3.27

- 練習 3.28 : $\mathcal{S}_{\mathrm{sos}}[\![S]\!] = \mathcal{S}_{\mathrm{am}}[\![S]\!]$ の証明

Bisimulation を用いた証明

- Bisimulation を用いた証明のまとめ
  - sos の 1 ステップ $\longrightarrow$ am での計算列　の対応
  - am での計算列 $\longrightarrow$ sos の 1 ステップ　の対応
- 何か加えた言語や，違う抽象機械にこの手法を適用するときは，気をつけること！

練習問題

- **練習** 3.29

  表 2.2 **の** [while$_{\mathrm{sos}}$] **を，次の一見無害な公理で置き換える**

$$
\begin{array}{lcll}
\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle & \Rightarrow & \langle S; \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle & \text{if } \mathcal{B}[\![b]\!]s = \mathbf{tt} \\
\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle & \Rightarrow & s & \text{if } \mathcal{B}[\![b]\!]s = \mathbf{ff}
\end{array}
$$

- $\mathcal{S}_{\mathrm{sos}}[\![S]\!] = \mathcal{S}'_{\mathrm{sos}}[\![S]\!]$ **の証明**
- **練習** 3.26 **や** 3.27 **の証明は面倒になるか**？