可逆プログラミング言語 ROOPL++ のインタープリタの実装及び言語機能の拡張

2017SE004 青柳 裕樹　2017SE057 新美 伊織　2017SE086 竹市 翔哉　2017SE098
指導教員：横山 哲郎

1　はじめに

可逆計算は反転可能な 2 方向性計算モデルである．つまり，計算過程においてどの状態からもその直前と直後の状態が高々一意に決まる．そのため，計算過程で情報損失がないので，可逆計算では，ランダウアの原理 [1] に基づいたエネルギーの放出はないと言える．

可逆化の方法としては，計算過程における全ての中間の値を保存しておく方法や可逆プログラミング言語でプログラミングをする方法がある．前者の方法では，実行時間に比例したメモリ使用量が必要である．一方で後者の方法では，可逆言語でプログラミングするだけで可逆性が保証され，逆実行に必要となる逆プログラムを単純な再帰で導くことが可能であるので，計算過程における中間の値を保存する必要がない．

現在の可逆プログラミングの応用例としては，産業用ロボット制御においての逆実行によるエラー処理や離散事象シミュレーションにおいての逆実行による効率的なロールバックの実現などがある．

このような応用例がある中で，近年，可逆プログラミング言語にオブジェクト指向を取り入れた可逆オブジェクト指向プログラミング言語 (以下 ROOP 言語) についての研究が行われている．ROOP 言語は，複雑なデータを抽象化しやすく，可読性も高い．また，連結リストや木などの複雑なデータ構造を表現できるので，可逆アルゴリズムの開発にも役立つと考えられる．

本研究では，関数型プログラミング言語 OCaml を用いて ROOP 言語である ROOPL++ のインタープリタの実装及び言語機能の拡張を行う．我々の知る範囲において現段階では，ROOPL++ のコンパイラは実装されているが，インタープリタは実装されていない．さらに，既存のコンパイラでは構文解析時点での静的なエラー内容しか出力されないので，可逆言語におけるプログラムのアサーションの誤りや変数の値がゼロクリアされているかどうかなどの重要な情報がわからない．したがって，プログラマにとって非常に不便であり，ROOPL++ プログラムの作成が難しいというのが現状である．また，ROOPL++ の意味論の正しさも確認できていない．そのため，本研究では初めに，操作的意味論で記述された ROOPL++ の意味論を表示的意味論で記述し直し，意味論の修正及び正しさの確認を行う．具体的には既存の意味論の誤った部分を修正したり，**uncall** 文や **uncopy** 文などの操作が不明確な部分も追加する．そして，意味論をもとにインタープリタ及びオンラインインタープリタを実装し，プログラムを実行することによりその意味論の正しさを検証する．また，よりプログラミングしやすくなるように簡単な言語機能の拡張や基本的なデータ構造を扱うための標準ライブラリの実装も行う．このような拡張された機能を持ったインタープリタを Web 上で公開し，Web 上での実行が可能になることにより，簡単に ROOPL++ を試すことができる．また，コンパイラよりも拡張が容易である．なぜなら，コンパイラの実装や拡張を簡略化するツールとして COINS[*1]などが挙げられるが現時点では可逆言語には対応していないからである．それに対してインタープリタは構文，意味論，逆変換規則の定義だけで実装や拡張ができ，可逆な低水準言語である PISA の理解を必要としない．したがって，ユーザが思いついた新しい言語機能を追加し，それを試すことも容易になると考えられる．最後には，可逆言語 Janus とプログラムを比較することにより，ROOPL++ の使いやすさと表現力を確認する．この研究は，可逆アルゴリズム開発などに役立つと考えられる．

2　関連研究

2.1　可逆プログラミング言語

可逆プログラミング言語とは，プログラムの原子的な構文要素の実行過程が必ず可逆になるように設計されているプログラミング言語である．可逆性を保つために，プログラムには制約が設けられている．例えば，Janus の可逆更新文において，`x -= x` といった左辺の変数と同じ変数が右辺に出現する文は非可逆となるため許されていない．また，条件分岐文や繰り返し文には，通常の条件式の他にアサーションという条件式も必要となる．

2.2　可逆オブジェクト指向プログラミング

近年では，可逆プログラミングにオブジェクト指向概念（継承，カプセル化，ポリモーフィズムなど）を取り入れた ROOP についての研究が進められている [2]．これまでの既存の可逆プログラミング言語 (Janus，R など) は，ROOP 言語とは違い，プログラマが抽象化をしづらく，また，複雑なデータを扱う場合，コードが長くなり，可読性も低くなるという欠点があった．ROOP 言語の場合，プログラマはクラスを使用することにより，抽象データ型を自由に定義できるので，抽象化しやすく可読性の向上も望める．

ROOPL++ は，ROOP 言語 ROOPL[3] を拡張した言

[*1] COINS : `http://coins-compiler.osdn.jp/index.html`（Accessed on 2021/01/26）

語である [2]．ROOPL では，オブジェクトが **construct** と **destruct** の間でしか存在できなく，使いにくいものになっていた．さらに，配列型の不足により，言語の表現力にも問題があった．このような欠点を解決するために ROOPL++ が設計された．ROOPL++ は，新しいオブジェクト定義文 **new**/**delete** 文の追加により，オブジェクトが **construct**/**destruct** ブロック外でも存在できる．また，配列型の追加や **copy**/**uncopy** 文によるオブジェクトの複数参照も追加され，より複雑な可逆プログラムの作成が可能である．

3 設計

インタープリタの実装を行う前に ROOPL++ の操作的意味論を修正し，表示的意味論で記述し直す．そうすることで，意味論の正しさが確認でき，本研究で用いる関数型プログラミング言語でのインタープリタの実装がしやすくなると考えられる．

3.1 表示的意味論

表示的意味論は，プログラミング言語の意味を記述する手法の一つであり，構文領域から意味領域への関数 (意味関数) を定義することにより意味を表現する．

3.2 意味関数

次に，意味関数を定義する．意味関数を図 1 に示す．意

$$\mathcal{E} : \mathit{Expressions} \to (\mathit{Envs} \times \mathit{Stores}) \to \mathit{Values}$$
$$\mathcal{S} : \mathit{Statement} \to (\mathit{Envs} \times \mathit{ClassEnvs}) \to \mathit{Stores} \to \mathit{Stores}$$
$$\mathcal{P} : \mathit{Programs} \to (\mathit{Envs} \times \mathit{Stores})$$

図 1 意味関数 ([2] を基に作成)

味関数 $\mathcal{E}$ は，式に対する部分関数，$\mathcal{S}$ は，文に対する部分関数，$\mathcal{P}$ は，プログラムに対する部分関数である．

本研究で修正した表示的意味論の一部を図 2 に示す．1 つ目は，配列を変数 x に格納する文の意味論である．要素数 e の分だけメモリのロケーションが確保され，全ての配列要素の値を 0 にする．既存の意味論では，意味が理解しづらかったので，修正を行った．2 つ目は，delete 文の意味論である．既存の意味論では，delete 文での具体的な操作が定義されていなかったので，追加をした．補助関数 remove は，指定したストア μ の定義域から指定したロケーション l を取り除いたストア μ' を返す関数である．

本研究では，これらの意味論の他に，**swap** 文，**uncall** 文，**copy** 文，**uncopy** 文及び意味関数 $\mathcal{P}$ の修正も行った．

4 実装

本研究では，関数型言語の OCaml を用いてインタープリタの実装を行った．OCaml はデータ型を簡単に定義でき，パターンマッチング文や再帰によって処理を簡潔に記述することができるので，処理系の実装に適切だと言える．

$$\mathcal{S}[\![\textbf{new}\ a\,[\,e\,]\ x]\!]_{\gamma,\Gamma}(\mu) = \mu[l \mapsto \{l'_1,\ \ldots, l'_n\},\ l'_1 \mapsto 0, \ldots, l'_n \mapsto 0]$$
where
$n = \mathcal{E}[\![e]\!](\gamma,\mu),\ l = \gamma(x),$
$\{\ l'_1,\ \ldots,\ l'_n\ \} \cap \mathrm{dom}(\mu) = \emptyset$

$$\mathcal{S}[\![\textbf{delete}\ c\ y]\!]_{\gamma,\Gamma}(\mu) = \mathrm{remove}(\{\gamma(x),\ l_0,\ \gamma(t_1), \ldots,\ \gamma(t_m)\},\ \mu)$$
where
$$\Gamma(c) = \Big(\ \overbrace{\{\langle t_1,\ f_1\rangle,\ \ldots,\ \langle t_m,\ f_m\rangle\}}^{fields},\ methods\ \Big)$$
$$l = \mathrm{search\_l}(y,\gamma,\mu) \quad \mu(l_0) = \langle c,\ \gamma'\rangle$$
$$\mu(\gamma(t_1)) = 0\ \cdots\ \mu(\gamma(t_m)) = 0$$

$$\mathrm{search\_l}(y,\gamma,\mu) = \begin{cases} l & \text{if } y = x\,[\,e\,] \\ & \quad \text{where } l = \mu(\gamma(x))[v],\ v = \mathcal{E}[\![e]\!](\gamma,\mu) \\ \gamma(x) & \text{if } y = x \end{cases}$$

$$\mathrm{remove}(l,\mu) = \mu'$$
where $\mathrm{dom}(\mu') \subset \mathrm{dom}(\mu),\ \mathrm{dom}(\mu) \setminus \mathrm{dom}(\mu') = \{l\}$

図 2 意味関数 $\mathcal{S}$ の定義の一部 ([2] を基に作成)

4.1 データ型の定義

初めに，抽象構文木でプログラムを処理するために，データ型を定義しなければならない．OCaml では，type 宣言を使うことでデータ型を定義できる．データ型を定義することで，プログラムを抽象構文木で表すことができる．

4.2 字句解析器と構文解析器の実装

プログラムを抽象構文木に変換するためには字句解析器と構文解析器が必要である．字句解析器は，文字列を字句列に変換，構文解析器は，字句列を抽象構文木に変換するプログラムである．本研究では，字句解析器の実装には ocamllex を，構文解析器の実装には ocamlyacc を用いた．これらは定義ファイルを入力するだけで解析プログラムを自動生成するツールである．

4.3 逆変換器の実装

逆変換器は，プログラムを逆変換するためのプログラムである．例えば，x **+=** e は，x **−=** e に変換される．逆変換器は，メソッドの逆呼び出し，つまり，逆実行を実装するために必要である．逆変換規則にしたがってパターンマッチング文を用いることで関数を定義することで実装する．

4.4 評価器の実装

評価器は，表示的意味論を基に，意味関数 $\mathcal{E}$，$\mathcal{S}$ 及び $\mathcal{P}$ を OCaml 上で実装することで実現する．

例えば，式を評価する関数における整数，変数及び nil の処理のプログラムは図 3 のようになる．このように意味関数の定義をそのまま実装するだけでよいので簡単に実装が可能である．構文や式を拡張した場合は，評価器においてはパターンを追加するだけでよいので，拡張も容易に可能である．また，アサーション誤りなどの可逆特有のエラー

```
let rec eval_exp exp env st = match exp with
  | Const(n) -> IntVal(n)
  | Var(x)   -> lookup_st (lookup_envs x env) st
  | Nil      -> IntVal(0)
```

図 3　式の意味関数の実装

も出力可能である．

4.5　オンラインインタープリタの実装

オンラインインタープリタは，環境を構築しなくても Web 上でプログラムを実行できる環境である．本研究では，プログラムの実行以外に逆プログラムの表示や，本研究で実装した標準ライブラリの読み込みをできるように実装した．

4.6　実行例

実装したインタープリタでのエラー表示の例を示す：

```
1 class Program
2     int x
3     method main()
4         x ^= 1
5         if x = 1 then
6             x += 1
7         else
8             skip
9         fi x = 1 // アサーションの評価で異常終了
```

実行結果

```
if x = 1 then
    x += 1
else
    skip
fi x = 1
ERROR:Assertion should be true in this statement
```

実行するとアサーションの評価の結果によって異常終了する．このように実装したインタープリタでは，異常終了の原因になったアサーションを出力すること可能である．

5　言語機能の拡張

本研究では，より ROOPL++ を使いやすくするために言語機能を拡張する．拡張するためには構文，逆変換規則及び意味論を定義する．また，その拡張した機能を試したい場合，これらの定義をインタープリタに追加するだけで良い．そのため，インタープリタは，機械語への変換が必要なコンパイラよりも拡張が容易であり，新しい機能を試したい場合に便利である．

5.1　ドット演算子

ROOPL++ ではインスタンス変数にアクセスする際，getter や setter のようなメソッドが必要である．また，毎回変数を用意しなければならないのでプログラマにとって不便であり，コードも冗長になってしまう．そこで，インスタンス変数にアクセスするためのドット演算子を追加する．構文を図 4 に示す．左辺値と右辺値のどちらでも使えるようにするために変数識別子と式の構文に追加した．アクセスには一般的なオブジェクト指向プログラミング言語と同じように，“インスタンス . メンバ変数” と記述する．

$$
\begin{aligned}
y &::= x \mid x[e] \mid y.y \\
e &::= \overline{n} \mid x \mid x[e] \mid \textbf{nil} \mid e \otimes e \mid e.e
\end{aligned}
$$

図 4　変数識別子と式へのドット演算子 (.) の構文の追加

```
local int i = 0
from i = 0 do
    // 処理
    i += 1
until i = 10
delocal int i = 10
```

```
for i in (0..9) do
    // 処理
end
```

図 5　10 回繰り返す文

$$
s ::= \cdots \mid \textbf{for } x \textbf{ in } (e_1 \,..\, e_2) \textbf{ do } s \textbf{ end}
$$

図 6　回数指定 for 文の構文の追加

ドット演算子の追加により，より柔軟なプログラム作成が可能になったと考えられる．

5.2　回数指定の繰り返し文

繰り返し文を記述する際には通常の終了条件式以外に，アサーションが必要となる．そのため，決まった回数の繰り返し文であってもコードが冗長になってしまう．例えば，ある処理を 10 回繰り返す文は，図 5 左 のようになり，**local/delocal** 文も必要となる．そこで，指定した回数だけ繰り返す **for** 文を追加する．

構文を図 6 に示す．ここで x はループカウンタ変数，e_1 と e_2 の値がそれぞれカウンタ変数の初期値と最終値，s が繰り返しを行う文である．カウンタ変数の初期値が最終値より小さい場合，繰り返し毎にカウンタ変数をインクリメントし，逆の場合はデクリメントする．例えば，この **for** 文を用いて特定の処理を 10 回繰り返す文を記述すると，図 5 右のようになる．このように明示的な局所変数宣言が必要なく，アサーションも記述する必要がないのでプログラムが簡潔になり，わかりやすくなると考えられる．

5.3　switch 文

ROOPL++ において多分岐処理を **if** 文で記述する場合，アサーションの対応などが分かりづらく，コードが読みにくくなってしまう．そこで，一般的な構造化プログラミング言語にも実装されているような switch 文を可逆性を保つように実装する．本研究で考案した switch 文は図 7 のようになる．このようにアサーションの対応がわかりやすくなり，また，値の網羅性の自動確認も可能である．実装した switch 文の構文を図 8 に示す．一般的な switch 文とは異なり，可逆な switch 文ではアサーションとなる **esac** と **hctiws** が必要である．アサーションは一つの **case** の実行後，**hctiws** で指定されている変数の値が **esac** で指定されている値と一致しなければならな

```
switch x
    case 1 // 処理1
    esac // アサーション1
    break
    case 2 // 処理2
    esac // アサーション2
    break
    case 3 // 処理3
    esac // アサーション3
    break
    default // 処理4
    break
hctiws x
```

図 7 可逆 switch 文の例

$$
\begin{array}{llll}
c & ::= & \textbf{case} \mid \_ & p ::= \textbf{esac} \mid \_ \\
q & ::= & e(\textbf{:}e)^? \mid \_ & b ::= \textbf{break} \mid \_ \\
s & ::= & \cdots \mid \textbf{switch}\ y\ (c\ q\ s\ p\ q\ b)^*\ \textbf{default}\ s\ \textbf{break}\ \textbf{hctiws}\ y &
\end{array}
$$

図 8 switch 文の構文

い．また，それぞれの **esac** の値は異なる値でなければならない．さらに，**default** の文の実行後の **hctiws** の変数の値は全ての **esac** の値と等しくなってはいけない．これらの制約は可逆性を保つためである．この可逆 switch 文はフォールスルーの記述も可能である．フォールスルーとは，**break** がない場合に switch 文を抜けずに次の **case** を実行することである．フォールスルーの場合は，各 **case** にアサーションを記述せずに **break** がある **case** に全てのアサーションを記述する．これらの制約はマッチした場所を一意に定めるために必要である．

5.4 逆変換器の拡張

本節で拡張したドット演算子，for 文，swich 文の逆変換器の拡張も行った．for 文の場合を次に示す：

$$
\begin{aligned}
&\mathcal{I}[\![\textbf{for}\ x\ \textbf{in}\ (e_1\,..\,e_2)\ \textbf{do}\ s\ \textbf{end}]\!] = \\
&\quad \textbf{for}\ x\ \textbf{in}\ (e_2\,..\,e_1)\ \textbf{do}\ \mathcal{I}[\![s]\!]\ \textbf{end}
\end{aligned}
$$

このように拡張した逆変換器を用いて，逆呼出し時における拡張された文の逆変換を実現した．

5.5 拡張された言語の可逆性

本節での文の拡張が行われても ROOPL++ が可逆であるためには以下が成り立つ必要がある:

補題 1 [文の可逆性] 任意のストア μ, μ_1, μ_2 に対して

$$
\mathcal{S}[\![s]\!]_{\gamma,\Gamma}(\mu) = \mu_1 \wedge \mathcal{S}[\![s]\!]_{\gamma,\Gamma}(\mu) = \mu_2 \Longrightarrow \mu_1 = \mu_2 \quad (1)
$$

この補題は文 s の構造帰納法により示せる．

6 標準ライブラリの実装

汎用プログラミング言語のように，基本的なデータ構造を標準で扱えるようにすることは，プログラマに対してのさらなる助けとなる．本研究では連結リスト，スタック，キュー，二分木を操作するクラスを実装した．また，一般的なオブジェクト指向プログラミング言語と同じようにコレクションを階層化することができ，プログラムの再利用性や保守性を向上させることが可能であることがわかった．また，ユーザがこれらのクラスを継承し標準ライブラリを拡張することが可能である．

7 Janus プログラムとの比較

本研究では連結リストを操作する Janus プログラムを作成し，ROOPL++ とのプログラムと比較を行った．Janus では構造体や代数データ型などが扱えないので配列で実装を行った．配列での実装は，必要な情報を全てプロシージャに渡さなければならないのでプロシージャの引数が増加してしまう．また，他のデータ構造を実装する際に共通部分が重複してしまうので，保守性も低いと考えられる．したがって，複雑なデータを扱う場合は ROOPL++ の方がプログラマにとって使いやすいことがわかった．

8 おわりに

本研究では，ROOP 言語 ROOPL++ の意味論を表示的意味論で記述し直し，意味論の修正と正しさの確認を行った．そして，その意味論を基にインタープリタ及びオンラインインタープリタを実装し，実際にプログラムを実行することにより，正しさの検証も行った．また，使いやすさを向上するためにドット演算子，回数指定の繰り返し文及び switch 文の構文，逆変換規則及び意味論を作成した．そして，実際に，実装したインタープリタを拡張することでインタープリタの拡張容易性を示した．また，データ構造を扱うプログラム作成の支援のために，インタープリタで読み込むことができる標準ライブラリとして，連結リスト，スタック，キュー，二分木を操作するクラスを実装した．さらに，ROOPL++ の連結リストプログラムと Janus で記述した連結リストのプログラムを比較し，ROOPL++ の有用性を確認した．本研究で，ROOP 言語のインタープリタの実装と言語拡張について一例を示したことにより，コンパイラしかなかったときに比べて他のユーザが ROOPL++ の拡張をすることが容易になったと考えられる．

参考文献

[1] Landauer, R.: Irreversibility and Heat Generation in the Computing Process, *IBM Journal of Research and Development*, Vol.5, No.3, pp.183–191 (1961).

[2] Cservenka, M.H.: Design and Implementation of Dynamic Memory Management in a Reversible Object-Oriented Programming Language, Master's thesis, University of Copenhagen (2018).

[3] Haulund, T.: Design and Implementation of a Reversible Object-Oriented Programming Language, Master's thesis, University of Copenhagen (2017).