D7-1

関数型言語における参照透明性を保持する参照型
A Referentially Transparent Reference Type in Functional Languages

石井 裕一郎[†] 武市 正人[†]
Yuichiro Ishii Masato Takeichi

[†]東京大学大学院工学系研究科情報工学専攻
Department of Information Engineering, Graduate School of Engineering
University of Tokyo

概 要

関数型言語には変数への一般的な代入がないので参照透明性があるという利点があるが、一方で代入を利用した方が効率がよく記述性が高くなるような場合も多い。そこで参照透明性を保持しつつ代入演算を関数型言語に導入する試みもおこなわれているが、これらはすべての代入演算に順序を付けることにより実現されているため柔軟性に欠けることが多い。これに対して我々は高々1回だけの代入を許す参照型データを関数型言語に導入して、従来の方法より柔軟で記述しやすいものとした。本稿では、この新しい参照型の性質と応用について述べる。

1 はじめに

関数型言語では変数に対する代入がないので参照透明性があるという利点がある [5]。

手続型言語の代入に類似したものを関数型言語に導入する試みもされている [8] [11] [12]。これらは参照透明性維持のため、すべての計算に順序をつけるという手法を用いているが、これではプログラムの記述に柔軟さが欠けるという問題点がある。

参照透明性を犠牲にしても効率向上のために代入を導入する例もあるが [10]、関数プログラミングにおいては、参照透明性を失うことにはあまりに惜しいものといえる。

本稿では、代入を高々1回だけ許すこととし、代入を実行できる状況を限定することによって参照透明性を保持する手法を提案する。以下ではまず、代入の許される状況とその表現法を述べ、参照型データを導入する。ついで代入と類似の入出力動作に参照型データを用いる記述法を示し、最後に今後の展望と課題を述べる。

2 従来の手法

参照透明性とは、「同一の文脈内で同一の識別子は同一の値を表す」という性質である。一般の代入によって参照透明性が損われるのは、複数回の代入が実行されると変数を参照したときの値が場合によって変わってしまうからである。したがって参照透明性を損わずに代入を導入するには、

1. 代入の実行と代入される変数への参照の間に順序を付ける。
2. 複数回の代入を禁止する。

という2種類の解決法が考えられる。

従来は複数回の代入をできるようにするため、上記1の手法が用いられてきた [12] [11] [8] [3] [4]。これらの手法の本質は以下の例で説明できる。

$$
\begin{array}{lll}
foo :: World \to ((Int, Int), World) & & \\
foo\ w = \ \mathsf{let} & (r, w1) & = new\ w \\
 & (s, w2) & = new\ w1 \\
 & (_, w3) & = assign\ r\ 123\ w2 \\
 & (_, w4) & = assign\ s\ 789\ w3 \\
 & (a, w5) & = deref\ r\ w4 \\
 & (b, w6) & = deref\ s\ w5 \\
 \quad \mathsf{in}\ ((a + a, b), w6) & &
\end{array}
$$

r と s が参照型の変数である。new は未だ代入さ

れていない(「未定」という。これに対し代入後は「既定」という)参照の生成、$assign$ は代入、$deref$ は参照外しを表す。$a = 123, b = 789$ が代入と参照の結果である。

この例では、$World$ 型の変数 $w, w1, \ldots, w6$ を次々と渡して代入と参照の順序を限定することにより参照透明性を維持している。$World$ 型の変数を表記上隠してプログラムを作成しやすく読みやすくし、効率良く翻訳する手法も提案されている [8] [7]。しかしこれらの手法では、本来独立に実行できる代入すら順序を付けなければならないという欠点がある。たとえば上記の r と s の代入と参照は本来独立にできるはずである。具体的には、関数

$$\begin{aligned} boo\ w = \ & \text{let } (r, w1) &&= new\ w \\ & \quad\ (_, w2) &&= assign\ r\ 123\ w1 \\ & \quad\ (a, w3) &&= deref\ r\ w2 \\ & \text{in } (a, w3) \\ woo\ w = \ & \text{let } (s, w1) &&= new\ w \\ & \quad\ (_, w2) &&= assign\ s\ 789\ w1 \\ & \quad\ (b, w3) &&= deref\ s\ w2 \\ & \text{in } (b, w3) \end{aligned}$$

によって foo を表すことができないことが問題となる。つまり遅延評価の「必要になるまで評価をしない / 計算の順序は必要性によって決まる」という特徴が「順序を決めて実行するため」に失われる。

このようなことから、次のような疑問が出てくる。果たして常に複数回の代入を考慮する必要があるのだろうか? 場合によっては単一代入のみですむ問題も多いのではないだろうか? たとえば [8]では入出力の実装に、[12]では関数 map の実装に、代入を使用しているが、これは単一代入で済む問題である。また、Committed-Choice 型言語のユニフィケーションは単一代入と極めて類似している。

このような観点から、ここでは複数回の代入を禁止する上記 2のアプローチを考察することとした。

3 単一代入と参照型

代入演算の対象となるデータとして参照型を導入する。ここでは、この参照型を Pascal のように ↑ を使用して表記することとした。これは [12]の表記と同様である。たとえば、r が整数のリスト $[Int]$ への参照型データを表すとき、その型は $r :: \uparrow[Int]$ であり、$r\uparrow :: [Int]$ が参照外しに相当する。上記 foo に対応するプログラムは

$$\begin{aligned} foo\ &:: \uparrow(Int, Int) \to (Int, Int) \\ foo\ w = \ & \text{case } w\uparrow \text{ of} \\ & \quad _@(r\uparrow, s\uparrow) \to \\ & \quad \text{let } a = \text{case } r\uparrow \text{ of } _@(123) \to r\uparrow \\ & \quad\quad\ \ b = \text{case } s\uparrow \text{ of } _@(789) \to s\uparrow \\ & \quad \text{in } (a + a, b) \end{aligned}$$

のようになる。代入は case 式の中の $_@(\ldots)$ のようなパターンによって表す。たとえば、$_@(123)$ は整数 123 を代入することを表現している。foo の case 式の $_@(r\uparrow, s\uparrow)$ は、

1. $w\uparrow$ が既定の場合は、通常の case 式と同様のパターン照合がおこなわれる。
2. $w\uparrow$ が未定の場合は、以下の過程が不可分におこなわれる。
   (a) 未定の参照型データ r と s を生成する。
   (b) 対 $(r\uparrow, s\uparrow)$ を生成する。
   (c) この対を w に代入する。$w\uparrow$ を評価して弱頭部正規形 (WHNF) を得る。

のように動作する。参照 r、s の生成が既存の参照 w への埋め込みとして実現されている点に特徴がある。したがって、前述の new に相当する部分はない。

この手法では、2 つの代入が独立に表現されている。このため、

$$\begin{aligned} boo, woo\ &:: \uparrow Int \to Int \\ boo\ r = \ & \text{case } r\uparrow \text{ of } _@(123) \to r\uparrow \\ woo\ s = \ & \text{case } s\uparrow \text{ of } _@(789) \to s\uparrow \\ foo\ w = \ & \text{case } w\uparrow \text{ of} \\ & \quad _@(r\uparrow, s\uparrow) \to (boo\ r + boo\ r, woo\ s) \end{aligned}$$

のような書き換えも形式的に行える。これは従来の手法に欠けていたモジュール性を改善する。

式 $boo\ r + boo\ r$ においては参照 r に対する代入は 1 度のみ実行される。参照 r を共有するため + の右辺左辺とも同じ値に評価される。

case 式以外の文脈で未定の参照を評価すると、ほかの文脈でその参照への代入がされるまでサスペンドする。

これらの性質は関数型言語に非決定性を導入するための Burton の手法 [2]を一般化したものである。

4 参照型と入出力

入出力などの非決定性動作を代入の一般化として

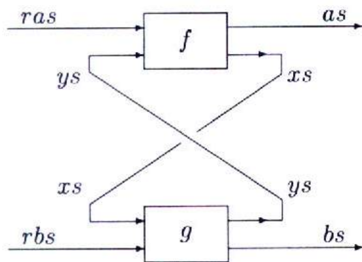

図1　相互に通信するプロセス

5　並行システムの記述

たとえば、2つのプロセス f と g が相互に通信を行いつつ独立して入出力を行う場合を関数プログラムで記述することを考える。f と g の間のメッセージを xs と ys で、f および g と OS との応答を $as = ras\uparrow$ と $bs = rbs\uparrow$ で表す。

$$
\begin{array}{lll}
f & :: & ([y], \uparrow[a]) \to ([x], [a]) \\
g & :: & ([x], \uparrow[b]) \to ([y], [b]) \\
f \bowtie g & :: & \uparrow([a], [b]) \to ([a], [b]) \\
(f \bowtie g)\ rzs = & & \mathsf{case}\ rzs\uparrow\ \mathsf{of} \\
& & \quad _@(ras\uparrow, rbs\uparrow) \to \\
& & \quad\ \mathsf{let}\ \ (xs, as) = f\ (ys, ras) \\
& & \quad\quad\quad\ (ys, bs) = g\ (xs, rbs) \\
& & \quad\ \mathsf{in}\ (as, bs)
\end{array}
$$

この様子を図1に示す。入出力の順序を決める必要のある従来の手法では、このようなシステムの記述は不可能である。この演算子 $\bowtie$ によれば、たとえば、`talk` などの複数の入出力動作が非決定的におこなわれる処理、デッドロックを起こさないため `fork()` を用いて入力と出力をそれぞれ独立に処理する UNIX の手法などが表現できる。

6　おわりに

本稿では、関数型言語において参照透明性を維持する参照型と代入および副作用の記述について提案した。

この参照型は、特に遅延評価と相性がよく、入出力や非決定性を扱うプログラムの作成においてもモジュール性を損わないため、柔軟性に富む。

一方単一代入であるという制限もある。したがって、in-place な複数回代入のできる手法 [8]と状況に応じて使いわけるべきであるということができる。

今後この参照型については、並列計算の記述手法および種々のアルゴリズムの検討、簡約規則の作成とプログラムの性質の証明手法の確立、*getchar* などの他言語の呼び出し機構の一般的な導入、関数の引数部パターン照合等読みやすい表記法、処理系の実現などを検討する必要がある。

参考文献

[1] Bird, R. and Wadler, P.: *Introduction to Functional Programming*, Prentice Hall, 1988.

[2] Burton, F. W.: Nondeterminism with Referential Transparency in Functional Programming Languages, *The Computer Journal*, Vol. 31,No. 3(1988), pp. 243–247.

[3] Gordon, A.: *Functional Programming and Input/Output*, PhD Thesis, Computer Laboratory, University of Cambridge, August 1992.

[4] Gordon, A.: An Operational Semantics for I/O in a Lazy Functional Language, *Functional Programming and Computer Architecture*, Copenhagen, ACM Press, June 1993, pp. 136–145.

[5] Hudak, P.: Conception, Evolution, and Application of Functional Programming Languages, *ACM Computing Surveys*, Vol. 21,No. 3(1989), pp. 359–411.

[6] Hudak, P. and Sundaresh, R. S.: On the Expressiveness of Purely Functional Languages, Technical Report YALEU/DCS/RR-665, Department of Computer Science, Yale University, December 1988.

[7] Hudak, P., Wadler, P., et al.: Report on the Programming Language Haskell, a Non-strict Purely Functional Language Version 1.2, *SIGPLAN Notices, Haskell Special issue*, Vol. 27,No. 5(1992), pp. Section R.

[8] Jones, S. L. P. and Wadler, P.: Imperative Functional Programming, *Principles of Programming Languages*, January 1993, pp. 71–84.

[9] Launchbury, J.: A Natural Semantics for Lazy Evaluation, *Principles of Programming Languages*, Charleston, January 1993, pp. 144–154.

[10] Mauny, M.: *Functional Programming Using CAML Light*, May 1991. INRIA, Domaine de Voluceau, BP 105, F-78153 le Chesnay Cedex, e-mail: Michel.Mauny@inria.fr.

[11] Odersky, M., Rabin, D., and Hudak, P.: Call by Name, Assignment, and the Lambda Calculus, *Principles of Programming Languages*, January 1993, pp. 43–56.

[12] Swarup, V., Reddy, U. S., and Ireland, E.: Assignments for Applicative Languages, *Functional Programming and Computer Architecture*(Hughes, R. J. M.(ed.)), Berlin, Springer-Verlag, 1991, pp. 192–214.

[13] Thompson, S.: Interactive Functional Programming, *Research Topics in Functional Programming*(Turner, D. A.(ed.)), Addison-Wesley Publishing Company, 1990.

表現することができる [8] [3] [4] [7] [6]。従来の手法では代入同様すべての入出力に順序を付けて表現する。したがって、独立に入出力をおこなうプログラムの表現は極めて難しい。たとえば「必要になるまでは端末から文字を読み込まない」という動作が記述できない。このため、プログラムのモジュール性が損なわれる結果となる場合がある。一方、端末からの入力を遅延ストリームと見て対話的プログラムを作成することは関数型言語においても頻繁に使用される手法である [1] [13]。

前節で述べた単一代入のみを許す参照型はこのような入出力を純粋に関数的に記述する際にも役立つものである。

以下では、手続き型言語 C の関数を利用する場合を考える。単一代入と同様に考え、`getchar()` や `putchar()` による結果を未定の参照に代入して、その結果を返すようにすれば参照透明性は保たれる。

$$
\begin{aligned}
&getchar &&:: \ \uparrow Int \rightarrow Int \\
&putchar &&:: \ Char \rightarrow \uparrow() \rightarrow ()
\end{aligned}
$$

関数 $getchar$ と $putchar$ を使用して、キーボードから入力した文字をそのまま画面に表示するプログラムを作成すると

$$
\begin{aligned}
&echo &&:: \ \uparrow[(Int, ())] \rightarrow () \\
&echo\ rs = &&\textsf{case}\ rs\uparrow\ \textsf{of} \\
& &&\ _@((rc\uparrow, ru\uparrow) : (rs'\uparrow)) \rightarrow \\
& &&\quad \textsf{let}\ \ c = getchar\ rc \\
& &&\quad \textsf{in if}\ c = eof\ \textsf{then}\ () \\
& &&\qquad \textsf{else} \\
& &&\qquad\ \textsf{case}\ (putchar\ (chr\ c)\ ru)\ \textsf{of} \\
& &&\qquad\quad () \rightarrow echo\ rs'
\end{aligned}
$$

のようになる。関数は通常の case 式、let 式によって記述され、入出力の順序は遅延評価で計算が進む順によって決まる。

この $echo$ を文字列の入力と出力にモジュール化するには以下のようにする。文字列出力関数は

$$
\begin{aligned}
&puts &&:: \ [Char] \rightarrow \uparrow[()] \rightarrow () \\
&puts\ [\,]\qquad rus = &&() \\
&puts\ (c : cs)\ rus = &&\textsf{case}\ rus\uparrow\ \textsf{of} \\
& &&\ _@(ru\uparrow : rus'\uparrow) \rightarrow \\
& &&\quad \textsf{case}\ (putchar\ c\ ru)\ \textsf{of} \\
& &&\qquad () \rightarrow puts\ cs\ rus'
\end{aligned}
$$

となる。

従来の手法では記述できない遅延入力ストリームも、自然に記述することができる。

$$
\begin{aligned}
&gets &&:: \ \uparrow[Int] \rightarrow [Char] \\
&gets\ rs = &&\textsf{case}\ rs\uparrow\ \textsf{of} \\
& &&\ _@(rc\uparrow : rs'\uparrow) \rightarrow \\
& &&\quad \textsf{let}\ c = getchar\ rc \\
& &&\quad \textsf{in if}\ c = eof\ \textsf{then}\ [\,] \\
& &&\qquad \textsf{else}\ (chr\ c) : gets\ rs'
\end{aligned}
$$

関数 $gets$ によれば、[1] [13]で挙げられた対話関数を駆動することができる。

繰り返し動作を表す高階関数 $seqs$ とリスト処理などの高階関数を利用して $gets$ を表すこともできる。

$$
\begin{aligned}
&seqs &&:: \ (\uparrow a \rightarrow a) \rightarrow \uparrow[a] \rightarrow [a] \\
&seqs\ f\ ris = &&\textsf{case}\ ris\uparrow\ \textsf{of} \\
& &&\ _@(ri\uparrow : ris'\uparrow) \rightarrow \\
& &&\qquad f\ ri : seqs\ f\ ris' \\
& &&\ [\,] \rightarrow [\,]
\end{aligned}
$$

$$
\begin{aligned}
gets = map\ chr\ \cdot\ takeWhile\ (eof \neq) \\
\cdot\ seqs\ getchar
\end{aligned}
$$

このように高階関数を柔軟に組み合わせてプログラムを構成するという関数プログラムの手法がそのまま応用できる点は注目に値する。

$puts$ と $gets$ を使用すれば、上記 $echo$ と同じ動作をする関数は

$$
\begin{aligned}
&echo' &&:: \ \uparrow([Int], [()]) \rightarrow () \\
&echo'\ rs = &&\textsf{case}\ rs\uparrow\ \textsf{of} \\
& &&\ _@(ris\uparrow, rus\uparrow) \rightarrow \\
& &&\ puts\ (gets\ ris)\ rus
\end{aligned}
$$

のように記述できる。$echo$ と $echo'$ は型が異なるが、動作は「同一」である。この動作の性質を証明するために、Launchbury の Natural Semantics [9]を拡張した簡約規則を定めることができる。

参照型による入出力の記述は [8]などの $World$ による手法とは対照的である。$World$ では「世界は1つ」であるのに対して、参照型を用いる手法では「世界は $\uparrow[(Int, ())]$」と見たり、「世界を $\uparrow[Int]$ と $\uparrow[()]$ に分割」したりできるからである。

また、本稿に述べる手法では、本来独立した入出力の単位に強制的に順序を付けたりせず、また高階関数を利用して、自然に記述することができる。通常の簡約規則にしたがって簡約をおこない、入出力が必要になった時点で初めてこれを行うという遅延評価の簡約系に適した手法といえる。