多次元拡張 for 文を実現した新たな関数型プログラミング言語の開発

横山 航基[1,a)]　木村 大輔[1,b)]

概要：新たなプログラミング言語 Qsitory（シトリー）を提案する．この言語は，関数型言語をベースとしながら手続き型言語の特徴ももち，Python に近い記述を提供する．特徴としては，リストやタプルなどの基本的なコレクションの操作が演算子で可能であること，代入式の左辺にパターンが利用でき，タプルの 1 要素のみを取得するなどの処理が容易であること，Java や Python などの拡張 for 文をさらに拡張し，複数のリストから 1 つずつ要素を取ってくり返しを行うようにした多次元拡張 for 文を備えていること，構造体のフィールドを動的に追加可能，構造体のフィールド名を文字列として処理することができる，などがある．これらの特徴は構造体の扱いを柔軟にする効果がある．本発表では，提案内容を実現するための評価アルゴリズム・型推論アルゴリズムを与え，これらに基づくインタープリタの OCaml 上でのプロトタイプ実装を紹介する．

キーワード：関数型プログラミング言語 Qsitory，多次元拡張 for，インタープリタ

1. はじめに

構造体は複数の (型が異なるかもしれない) データをまとめて格納し，扱うことを可能にする基本的なデータ構造であり，C 言語の `struct` など様々なプログラミング言語でサポートされている．

構造体はユーザー定義可能な型であり，定義時には構造体の各メンバにアクセスするためのフィールド名とその型を組にした形で与える．すなわち，通常は型の定義時にフィールド名とその個数は固定され，それ以降これらは途中で変更されることがないのが普通である．このような構造体の設計を前提とすることで，プログラマは構造体の内容を把握しやすくなるだけでなく，C 言語のように構造体をメモリ上の連続した領域に配置することでプログラムの実行効率やコンパイラを効率的にできる利点がある．その一方で，ソフトウェアのバージョンアップなどで既存の構造体に新たなフィールドを追加する必要が発生したときは，その構造体が関わっている全ての箇所を変更する必要があるなど，メンテナンス時に手間がかかることもある．

本発表では新たなプログラミング言語 Qsitory（シ

[1] 東邦大学理学部情報科学科
[a)] kokiprograming@gmail.com
[b)] kmr@is.sci.toho-u.ac.jp

T ::= `int` | `double` | `string` | `bool` | `unit`

| `fun(`T `->` T`)` | T `list` | `(`T $*\cdots*$ T`)`

| `struct` ⟨*name*⟩ | `struct` ⟨*name*⟩`()` : B

図 1 Qsitory の型

トリー）を提案する[*1]．この言語は，関数型言語の強力な関数機構を備えていると同時に手続き型言語の特徴ももち，Python に近い記述を提供する．大きな特徴として，構造体のフィールドが動的に追加可能であったり，構造体のフィールド名を文字列として処理することでき，柔軟な構造体の扱いを可能にする．また，この機能を有効に利用するために，Java や Python などの拡張 for 文をさらに拡張し，複数のリストから 1 つずつ要素を取ってくり返しを行うようにした多次元拡張 for 文を備えている．その他の特徴として，リストやタプルなどのコレクションに追加，削除などの操作が代入演算子 `+=` や `-=` で可能であること，代入文の左辺にパターンを書くことでタプルの 1 要素のみを取得するなどの処理が容易であることも挙げられる．

ここでは，提案内容を実現するための評価アルゴリズム・型推論アルゴリズムを与え，これらに基づくインタープリタの OCaml 上でのプロトタイプ実装を紹介する．

2. Qsitory の構文

本節では現状の Qsitory の構文について述べる．Qsitory の構文は型，式，パターン，値からなる．

Qsitory の型 (T で表す) は図 1 で定義される．ここで，`int`，`double`，`string`，`bool`，`unit`，`fun(`T `->` T`)` はそれぞれ整数型，浮動小数点型，文字列型，ブール型，ユニット型，関数型を意味し，T `list` と `(`T $*\cdots*$ T`)` はそれぞれリスト型および組型を意味する．また，`struct` ⟨*name*⟩`()` : B の ⟨*name*⟩ はユーザーが定義する構造体名であり，B 部分は T : ⟨*fld*⟩ の形の式がいくつか並んだブロックがくることを想定している．ブロックは一般に，次で与えられる式の列であり，改行で区切られる．また，ブロックの開始および終了はインデントを用いた Python 風の判別をする．

[*1] 開発版は https://github.com/koki37-byte/Qsitory で入手できる

E ::= x | i | d | s | `true` | `false` | `()`

| E `<` E | E `<=` E | E `>` E | E `>=` E | E `==` E

| `(`E, . . . , E`)` | `[`E, . . . , E`]` (組とリスト)

| T : x | T : x = E (変数宣言と初期化)

| P = E (パターンマッチ代入)

| E `+` E | E `-` E | E `*` E | E `/` E (算術演算)

| E `and` E | E `or` E | `not` E (ブール演算)

| x `+=` E | x `-=` E | x `*=` E | x `/=` E (算術代入)

| `fun(`x, . . . , x`) ->` B | `(`EE`)` (関数抽象・適用)

| `def` f`(`x, . . . , x`)` : B (関数定義)

| `return` E (リターン)

| `if` E : B `else` : B (条件式)

| `while` E : B (while)

| `match` E : P `->` B | ⋯ | P `->` B (マッチ)

| x `.-` P (パターン減算)

| x `.-=` P (パターン減算代入)

| `for` x, . . . , x `in` E, . . . , E : B (多次元拡張 for)

| `fordict (`x, . . . , x`) in` E : B (辞書型 for)

| x.⟨*fld*⟩ | E..E (構造体メンバ参照)

| x.⟨*fld*⟩ = E | E..E = E (構造体メンバ代入)

P ::= x | i | d | b | s | _ | (P, . . . , P) | [P, . . . , P]

図 2 Qsitory の式とパターン

Qsitory の式およびパターン (それぞれ E と P で表す) は図 2 で定義される．

式における x は変数, i と d と s と () はそれぞれ整数と浮動小数点と文字列とユニット型の定数である．(E,...,E) と [E,...,E] はそれぞれ組とリストである．式 T : x = E と T : x は型 T の変数 x の宣言で E は初期値，パターンマッチ代入 P = E はパターンマッチと代入を連続して行う．例えば $(\_, x) = (1, 2)$ は x に 2 を代入する．`fun`($x, \ldots, x$) -> B は関数抽象，(EE) は関数適用，`def` f($x, \ldots, x$) : B は関数定義であり，f = `fun`($x, \ldots, x$) -> B と等価である．その他の Qsitory に特徴的な式については後述する．パターンにおける x は変数パターンであり，任意の値 (定数，関数，値の組やリスト) にマッチする．ワイルドカード _ も任意の値にマッチする．パターン i と d と s は定数パターンであり，その定数にのみマッチする．組パターン (P,...,P) は同じ要素数の値の組かつ，各要素の値が対応するパターンにマッチする場合にマッチする．リストパターン [P,...,P] も同様である．

以下，Qsitory 特有の構文について述べる．パターン減算 x .- P は，x が組かリストを想定しており，x の値に対してパターン P でマッチした要素を削除する演算である．パターン減算代入 x .-= P は $x = x$.- P と等価である．多次元拡張 for は，複数のリストから 1 つずつ要素を取ってきてループ処理を行う．例えば，`for` x, y `in` [100, 200, 300], [80, 30, 20] : a += x - y は 1 度目のループでは (x, y) を $(100, 80)$，2 度目のループで (x, y) を $(200, 30)$，3 度目のループで (x, y) を $(300, 20)$ としながら a += x - y をくり返し実行する．辞書型 for `fordict` ($x, \ldots, x$) `in` E : B は，E が組を要素にした 1 つのリストであることを想定しており，その組から 1 つずつ要素を取り出してループ処理を行う．例えば，`fordict` (s, v) `in` [("r", 3), ("g", 1), ("b", 5)] : c += s は 1 度目のループでは (s, v) を ("r", 3)，2 度目のループでは (s, v) を ("g", 3)，3 度目のループでは (s, v) を ("b", 5) としながら c += s を繰り返し実行する．構造体メンバ参照 $x.\langle fld \rangle$ は，x が構造体型の変数であることを想定しており，そのフィールド名 $\langle fld \rangle$ のメンバを意味する．C 言語と同じく `a = x.f` により構造体 `x` のフィールド名 `f` のメンバの値を `a` に代入する．式 $x.\langle fld \rangle$ = E は構造体メンバ $x.\langle fld \rangle$ を E の値で更新する．また，$E_1..E_2$ において，E_1 と E_2 はどちらも文字列型を想定しており，E_1 を評価した値が構造体の変数 (`a` とする) の文字列 ‘‘`a"`，E_2 を評価した値がその構造体のフィールド (`f` とする) の文字列 ‘‘`f"` であるとき，$E_1..E_2$ は `a.f` と等価である．式 $E_1..E_2$ = E は構造体メンバ $E_1..E_2$ を E の値で更新する．

3. Qsitory のプログラム例

例えば，Qsiotry では以下のようなプログラムを書くことができる．なお，`getfield` は動的構造体のその時のフィールド名リストを取得するビルトイン関数である．

```
1   struct A():
2     int: f
3     int: g
4
5   struct A aa
6   aa.f = 10
7   aa.g = 20
8   aa.."h" = 30
9   struct A bb
10
11  for fld in getfield aa:
12    bb.fld = aa.fld
```

上記のコードは，まず `struct A` 型の構造体 `aa` を宣言し，その初期フィールド名 `f` と `g` のメンバを初期化する．次に新規フィールド名 `h` を `aa` に追加する (この時点で `struct A` 型は 3 つのフィールドをもつ構造体であるとみなされる)．次に `aa` と同

じ内容を別の構造体 bb にコピーする．このコードは，aa のメンバ数が多数であっても特に変更をする必要はなく利用できる．また，もし仮に aa 以前に struct A 型の構造体が宣言されていた場合，そのフィールド h の値は未定義 (*undef*) とされる．

4. Qsitory インタープリタの実装

本節では現在実装しているプロトタイプインタープリタの内容を述べる．作成しているインタープリタは以下の 3 段階からなる．

- 字句解析・構文解析：インデントを考慮した，Python 風のレイアウトを前提とした字句解析および構文解析を行う．
- 型推論：単一化のアルゴリズムに基づいた型推論を行う．不整合な型が発見されたときは型エラーにより終了する．
- 評価関数：関数型言語の理論 [1] を参考にして実装した，式の評価関数を用いる．

式の評価関数の定義においては型環境と変数環境を用いる．これらは式を評価するたびに更新される．型環境 ($\mathcal{T}$ で表す) は以下で与えられる．

$$\mathcal{T} ::= [(\langle \textit{type-name} \rangle, \mathrm{T}), \ldots, (\langle \textit{type-name} \rangle, \mathrm{T})]$$

ここで，$\langle \textit{type-name} \rangle$ は型の名前を意味する文字列である．整数型は “INT”，文字型は “STRING”，構造体 struct A は “A” のように与えられる．また，T は型の名前に対応する型である．型環境は特に構造体に関する情報を保持する．例えば struct A がフィールド名 f の整数型メンバのみをもつときは (“A”, struct A() : int : f) を型環境は保持する．

値 (V で表す) および変数環境 ($\mathcal{E}$ で表す) は図 3 で定義される．値は式を評価した結果である．値 $\mathrm{Clos}(\mathcal{E}, \texttt{fun}(x, \ldots, x)\ \texttt{->}\ \mathrm{B})$ は関数閉包である．値 $\texttt{struct}\ (\langle \textit{name} \rangle, [(\langle \textit{fld} \rangle_1, \mathrm{T}_1, \widetilde{\mathrm{V}}_1), \ldots, (\langle \textit{fld} \rangle_n, \mathrm{T}_n, \widetilde{\mathrm{V}}_n)])$ は構造体 struct $\langle \textit{name} \rangle$ の値で，フィールド $\langle \textit{fld} \rangle_i$

$$\begin{aligned}
\mathrm{V} ::= {} & i \mid d \mid s \mid \texttt{true} \mid \texttt{false} \mid \texttt{()} \mid (\mathrm{V}, \ldots, \mathrm{V}) \mid [\mathrm{V}, \ldots, \mathrm{V}] \\
& \mid \mathrm{Clos}(\mathcal{E}, \texttt{fun}(x, \ldots, x)\ \texttt{->}\ \mathrm{B}) \\
& \mid \texttt{struct}\ (\langle \textit{name} \rangle, [(\langle \textit{fld} \rangle, \mathrm{T}, \widetilde{\mathrm{V}}), \ldots, (\langle \textit{fld} \rangle, \mathrm{T}, \widetilde{\mathrm{V}})]) \\
\widetilde{\mathrm{V}} ::= {} & \mathrm{V} \mid \textit{undef} \\
\mathcal{E} ::= {} & [(x, \mathrm{T}, \widetilde{\mathrm{V}}), \ldots, (x, \mathrm{T}, \widetilde{\mathrm{V}})]
\end{aligned}$$

図 3 値と変数環境

のメンバは型 T_i をもち，$\widetilde{\mathrm{V}}_i$ (値もしくは未定義) をもつことを意味する．変数環境が保持する三つ組 $(x, \mathrm{T}, \widetilde{\mathrm{V}})$ は変数 x の型が T であり，その評価時点で $\widetilde{\mathrm{V}}$ をもつことを意味する．

式の評価は記法 $\mathcal{T}, \mathcal{E} \vdash \mathrm{E} \Downarrow \mathrm{V}, \mathcal{T}', \mathcal{E}'$ を用いる．これは「型環境 $\mathcal{T}$ と変数環境 $\mathcal{E}$ の下で式 E を評価した結果は値 V であり，評価後の型環境と変数環境はそれぞれ $\mathcal{T}'$ と $\mathcal{E}'$ である」を意味する．評価はいくつかの推論規則により定義される．例えば条件式についての評価は以下の 2 つの規則で与えられる．

$$\frac{\mathcal{T}, \mathcal{E} \vdash \mathrm{E}_0 \Downarrow \texttt{true}, \mathcal{T}_1, \mathcal{E}_1 \quad \mathcal{T}_1, \mathcal{E}_1 \vdash \mathrm{E}_1 \Downarrow \mathrm{V}, \mathcal{T}', \mathcal{E}'}{\mathcal{T}, \mathcal{E} \vdash \texttt{if}\ \mathrm{E}_0 : \mathrm{E}_1\ \texttt{else} : \mathrm{E}_2 \Downarrow \mathrm{V}, \mathcal{T}', \mathcal{E}'}$$

$$\frac{\mathcal{T}, \mathcal{E} \vdash \mathrm{E}_0 \Downarrow \texttt{false}, \mathcal{T}_1, \mathcal{E}_1 \quad \mathcal{T}_1, \mathcal{E}_1 \vdash \mathrm{E}_2 \Downarrow \mathrm{V}, \mathcal{T}', \mathcal{E}'}{\mathcal{T}, \mathcal{E} \vdash \texttt{if}\ \mathrm{E}_0 : \mathrm{E}_1\ \texttt{else} : \mathrm{E}_2 \Downarrow \mathrm{V}, \mathcal{T}', \mathcal{E}'}$$

構造体メンバ参照 $\mathrm{E}_1..\mathrm{E}_2$ の評価は次で与える．

$$\frac{\begin{array}{c}\mathcal{T}, \mathcal{E} \vdash \mathrm{E}_1 \Downarrow \textit{str}, \mathcal{T}_1, \mathcal{E}_1 \\ \mathcal{T}_1, \mathcal{E}_1 \vdash \mathrm{E}_2 \Downarrow \textit{fld}, \mathcal{T}', \mathcal{E}' \\ \mathcal{E}'(\textit{str}) = \texttt{struct}\ (\langle \textit{name} \rangle, [\ldots, (\textit{fld}, \mathrm{V}), \ldots])\end{array}}{\mathcal{T}, \mathcal{E} \vdash \mathrm{E}_1..\mathrm{E}_2 \Downarrow \mathrm{V}, \mathcal{T}', \mathcal{E}'}$$

このような推論規則を用いて式 E を環境 $\mathcal{T}$ と $\mathcal{E}$ の下で評価する関数 *eval* を以下で定義する．

$$\textit{eval}\ \mathcal{T}\ \mathcal{E}\ \mathrm{E} \stackrel{\mathrm{def}}{=} \begin{cases} (\mathrm{V}, \mathcal{T}', \mathcal{E}') & \text{if } \mathcal{T}, \mathcal{E} \vdash \mathrm{E} \Downarrow \mathrm{V}, \mathcal{T}', \mathcal{E}' \\ \textbf{fail} & \text{otherwise} \end{cases}$$

参考文献

[1] 五十嵐淳, “プログラミング言語の基礎概念” (ライブラリ情報学コア・テキスト), サイエンス社, 2011.