OCommand: OCaml上の型安全なシェルプログラミングのための領域特化言語

朝倉 泉[1,a)]　増原 英彦[1,b)]　青谷 知幸[1,c)]

概要：OCaml プログラム中からシェルコマンドを型安全に実行するための領域特化言語 OCommand を提案する. OCommand はコマンドの仕様である, 出力行の各欄の型とコマンドオプションによって起きる変化の記述を受け取り, コマンド起動関数とコマンドオプションを表す値定義を持つ OCaml モジュールを生成する. 生成された起動関数は, 複数のオプション値を受け取りコマンドを実行し, オプションに応じてコマンド出力行をパースして型付きのフィールドからなるレコードを返す. 出力行レコードにおけるフィールドの型や存在の変化を静的に検査するため, 一般化代数的データ型を用いた. Camlp4 を用いて実現した処理系を用いて, これまでに ls や ps などのコマンドを扱えることを確めている.

Izumi Asakura[1,a)]　Hidehiko Masuhara[1,b)]　Tomoyuki Aotani[1,c)]

Abstract: We propose a domain-specific language (DSL), called OCommand, for executing shell commands from within OCaml programs in a type safe way. OCommand takes a command specification consisting of types of output columns from the command and effects of command options on the types of the columns, and generates an OCaml module that contains a command-executing function and a set of values representing the command options. The command-executing function takes options as arguments, runs the command, parses the output lines, and returns them as a list of records. In order to statically-check existence and types of fields of the output record that can be changed by command options, we used generalized algebraic data type. With our implementation constructed by using Camlp4, we successfully handle typical Unix commands like ls and ps by using OCommand.

1. はじめに

システム管理のための簡単なプログラムを書く際に, シェルスクリプトや perl のようなスクリプト言語は非常に有用であることが知られている. それらの言語は sed などの正規表現を利用したコマンドや正規表現ライブラリなどでコマンドの実行結果の文字列 (コマンド出力) を扱う. しかし, シェルスクリプトや perl のような動的型付けのプログラミング言語では, 型が静的に決まらないために, 安全性に欠けるという問題点がある.

豊富なデータ構造や強力なモジュール化機構を備えた一般的なプログラミング言語でコマンドを扱うには, コマンドをコマンド出力の各欄が整数や日付・時刻などのデータ型に変換されたレコード (出力行レコード) を返す関数として扱うのが望ましい. 一方で, OCaml や Haskell などの強く型付けされた関数型言語で安全にコマンドを扱うには, コマンドをオプションを引数にとり, 静的型が付いた出力行レコードのリストを返すような関数 (コマンド関数) として扱うのが望ましい. しかしコマンドの中にはコマンドの出力形式を変化させるようなオプションを複数個組み合わせて指定することができるものがあり, そのようなコマンドの実行結果の型は引数のオプションに対して変化することがあるため, それらの言語でコマンドを安全かつ簡単に

[1] 東京工業大学 数理計算科学専攻
Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology
[a)] asakura.i.aa@m.titech.ac.jp
[b)] masuhara@acm.org
[c)] aotani@is.titech.ac.jp

扱うことは難しい.

OCaml 上でシェルプログラミングを行うためのライブラリとしては Shcaml[3], Haskell 上では Shelly[2] などのライブラリがあるが, コマンド出力に型を付けるための機能は提供していない.

本稿では OCaml 上でシェルコマンドのユーザがコマンド出力を安全に扱うための DSL である OCommand を提案する. OCommand では DSL で記述されたコマンドの仕様からコマンドの出力を構文解析して適切なデータに変換するためのコードを生成する. OCommand が生成するコードには, 以下のコードが含まれる.

- 指定されたオプションの集合を表現する値 (オプション集合)
- オプション集合を引数にとってそのオプションでコマンドを実行して出力レコードのリストを返す関数 (コマンド関数)

OCommand のコマンド関数には適切に静的型が付いているため, 引数のオプションに対してユーザが出力レコードを正しく扱っていることを型検査することができる.

本稿では 2 節で我々が目標とするようなコマンドの型付けについて説明する. 3 節では OCommand の DSL を提案する. 4 節では 2 節で挙げた目標を実現するための型付けやコード生成法を提案する. 5 節ではいくつかのコマンドについて OCommand を用いた例を挙げる. 6 節では関連研究を述べる. 7 節では結論と課題について述べる.

2. 本研究の目標

本研究では以下の目標を実現するための OCaml 上のシェルプログラミングのための処理系 OCommand を提案する.

(1) 外部コマンドを OCaml の関数として呼び出せること
(2) 出力行レコードに対して, ユーザがアクセスしたフィールドが存在することが検査できること
(3) 出力行レコードへの型変換が自動で行われること
(4) 外部コマンドに与えるオプションの変化に応じて出力行レコードの型が適切に変化すること
(5) 外部コマンドに与えるオプションが合成可能なこと

本節では OCommand と Shcaml のプログラム例を比較して, 上の目標を説明する.

OCommand では, `command` という関数 (コマンド関数) を OCaml 上でオプションを表す値 (オプション集合) に適用することでコマンド呼び出しを行う. コマンド関数の戻り値は出力行レコードのリストである. 出力行レコード `v` のフィールド `l` にアクセスするには `v..l` という記法をもちいる. 図 1 は, OCommand のプログラム例である. 合計ファイルサイズ `size` を受け取って, カレントディレクトリにあるファイルの部分集合で, 合計ファイルサイズが `size` を超えないようなものを求めるプログラムである. 図 1 のプログラムは, `ls` コマンドを呼び出して, その戻り値の出力行レコードに対するフィールドアクセスで, ファイルの名前やサイズの情報を取得している. 2 行目は `ls` コマンドを利用するために `Ls` モジュールを `open` している. 3 行目の式 `command (add_l empty)` という項はコマンド `ls` の呼び出しであり, 引数の `add_l empty` がオプション`-l` を表すオプション集合である. 4 行目から 11 行目は補助関数 `iter` の定義である. 補助関数 `iter` は

- これまでの合計サイズを表す `int` 型の引数 `cur_size`
- これまでの結果を表す `string list` 型の引数 `acc`
- 残りのリストを表す `files`

を引数に受け取る. `iter` 関数は `files` が空ならこれまでの結果を返し (6 行目), そうでないなら, `files` の先頭要素 `file` のサイズを調べてここで処理を終えるかどうかを判定する (8 行目から 10 行目). 8 行目の `file..size` と 10 行目の `file..name` はそれぞれ出力行レコード `file` の `size` フィールドと `name` フィールドへの参照を行う式である.

図 1 外部コマンドの起動と出力行レコードのアクセスを行う OCommand を用いたプログラム例

```
1  let collect_files size =
2    let open Ls in
3    let files = command (add_l empty) in
4    let rec iter cur_size acc files =
5      match files with
6      | [] -> acc
7      | file :: files ->
8        let cur_size = cur_size + file..size in
9        if cur_size > size then acc
10       else iter cur_size (file..name :: acc) files
11   in
12   iter 0 [] files
```

次に Shcaml で図 1 と同様の処理を行うプログラムを図 2 に示す. 1 行目から 7 行目では `ls -l` の出力を構文解析してラベルが付いた要素に分解するためのモジュール (Shcaml ではアダプタと呼ばれる) を生成している. 11 行目では `ls -l` コマンドを起動し, 生成したアダプタの `fitting` 関数で構文解析を行っている. 12 行目から 14 行目では `sed` 関数を用いてコマンド出力の各行に対して, `size` フィールドと `name` フィールドを含む出力行レコードを作成している. 13 行目では `Line.Delim.get` 関数を用いて `size` ラベルにアクセスし, それを `int` 型に変換している.

以下では図 1 と図 2 を比較して各目標を説明する.

2.1 コマンドの関数化

OCommand ではコマンドは OCaml の関数になっている. Shcaml ではコマンドは関数 `command` に実行するコマンドを文字列として渡すことで起動する. これはコマンド名やオプション名を書き間違いを静的に検出できないという欠点がある. OCommand ではそれらの間違いは多くの場合コンパイラが存在しない変数へのアクセスとして検出

図 2 Shcaml のコード例

```
module Ls_l = Adaptor.Delim.Make_names(struct
  let options = { Delimited.default_options with
                  Delimited.field_sep = ' '}
  let names = [ "perm"; "hard"; "owner";
                "group"; "size"; "mon";
                "day"; "time"; "name" ]
end)

let collect_files size =
  let res = Fitting.run_source ^$
    command "ls␣-l" -| Ls_l.fitting () -|
    sed (fun line ->
      (Line.Delim.get "size" line |> int_of_string,
       Line.Delim.get "name" line)) in
  let rec iter cur_size acc files =
    match files with
    | [] -> acc
    | (size,  name) :: files ->
      let cur_size = cur_size + size in
      if cur_size > size then acc
      else iter cur_size (name :: acc) files in
  iter 0 [] (Shtream.of_list res)
```

することが出来る.

2.2 フィールドが存在することの検査

OCommand ではコマンド出力を静的型が付いた出力行レコードとして扱う. そのため, 図 1 の 8 行目の `file..size` を `file..inode` のように書き換えると, 出力行レコードに `inode` というラベルのフィールドが無いため型エラーになる. 一方 Shcaml では図 2 のように出力を文字列をキーとするマップのような動的な構造で扱っている. そのため, 図 2 の 13 行目の`"size"`を`"inode"` と書き換えても, それを静的に検出することはできない.

2.3 出力行レコードへの変換

OCommand では出力行レコードへの変換はコマンド関数の内部で自動的に行われるため, スクリプトを書くユーザは型変換を行う必要がない. 例えば `ls -l` のコマンド出力は以下のようになる.

```
$ ls -l
total 8
-rw-rw-r--. 1 asakura asakura  0 Dec 23 14:17 aaa
-rw-rw-r--. 1 asakura asakura  0 Dec 23 14:17 bbb
-rwxrwxr-x. 1 asakura asakura  0 Dec 23 14:17 ccc
```

OCommand ではこのようなコマンド出力を図 9 出力行レコードで扱うことが出来る (型 `ls_l`)[*1]. Shcaml では `fitting` 関数によってコマンド出力の分割は自動で行うことができるが, コマンド出力の分割された要素は文字列型になっているため適切な型の変換を行うのはユーザの責任である. 例えば図 2 の 13 行目ではユーザが `size` ラベルの

[*1] 実際には 4 節で述べるように OCommand ではこのような OCaml のレコード型ではなく, OCommand 独自のレコード型の表現を用いる

図 3 出力行レコードの型

```
type perm = ... (*パーミッションを表す型*)
type date = ... (*日付を表す型*)
type time = ... (*時刻を表す型*)
type ls_l = {perm:perm; hard:int; owner:string;
             group:string; size:int; date:date;
             time:time; name:string}
```

要素を明示的に `int` 型に変換している. このようにフィールドの値に正しい変換が行われていることはスクリプトを書くユーザが保証しなくてはならない.

2.4 オプションに対する出力型の変化

コマンドの中にはコマンド出力の形式を変化させるようなオプションを複数組み合わせて指定することができるものがある. 例えば `ls` コマンドには `i` というオプションがあり, これを指定すると inode 番号をファイル名の左側に表示するようになる.

```
$ ls -i # inode 番号を表示
2892425 aaa
2892606 bbb
2892607 ccc
```

これをオプション `l` と組み合わせて指定すると, `l` オプションのメタデータに加えて inode 番号も表示するようになる.

```
$ ls -li
total 8
2892425 -rw-rw-r--. 1 ... aaa
2892606 -rw-rw-r--. 1 ... bbb
2892607 drwxrwxr-x. 2 ... ccc
```

また `ls` コマンドには `h` というオプションがあり, これは `l` オプションと同時に指定すると, ファイルサイズを表す部分の形式を `K`, `M`, `G` が付いた形式に変更する.

```
$ ls -lh
-rw-rw-r--. 1 asakura asakura    0 Dec 23 14:17 aaa
-rw-rw-r--. 1 asakura asakura    0 Dec 23 14:17 bbb
drwxrwxr-x. 2 asakura asakura 4.0K Dec 23 14:17 ccc
```

OCommand ではコマンド関数はオプション集合を引数に受け取る. そのため例えば図 1 のコマンド呼び出しを `ls -li` に変更するには, 3 行目を

```
let files = command (add_i (add_l empty)) in
```

のように, コマンド関数に渡す引数を変更するだけでよい. また OCommand では引数のオプションの値が変化すると, 出力行レコードの型も適切に変化するので, オプションの値を変えても型検査に通る限りフィールドアクセスに関するエラーは起こらないことが保証される. 一方で Shcaml では複数のオプションを受け取るコマンドを利用するには, オプションの組み合わせ毎にアダプタを作らなくてはならない. 例えば図 2 のコマンド呼び出しを `ls -li` に変更するには, `ls -li` 用のアダプタ `Ls_li` を作る必要がある. ま

た 11 行目のコマンド呼び出しと fitting 関数の呼び出しの 2 箇所を

```
command "ls␣-li" -| Ls_li.fitting ()
```

と書き換える必要がある.

2.5 オプションの合成

OCommand ではオプション集合は動的に合成することが出来る. 例えば図 1 において, 関数を呼び出す側でファイルをソートする順序を作成日時順やファイルサイズ順に変更するには以下のようにプログラムを書き換えれば良い [*2].

```
let collect_files default_opt size =
  let open Ls in
  let opt = default_opt (add_l empty)) in
  let files = command opt in
  ...
```

collect_files 関数に新しい引数 default_opt を増やして, そのオプションと l オプションを合成して ls コマンドを呼び出す. 例えばファイルを時刻順にソートする時は以下のように呼び出す.

```
collect_files add_t 10000
```

また, ファイルサイズ順にソートする時は以下のように呼び出す.

```
collect_files add_c 10000
```

Shcaml では複数のオプションを複数の別のモジュールとして扱うため, このようなプログラムは書くことができない.

2.6 課題

これらの 5 つの目標すべてを同時に達成するような OCaml の型システムの範囲で検査できるようなエンコードを行うには以下にあげる 2 つの問題がある. 1 つは出力行レコードの型を引数のオプションの値に対して変化させなくてはならないが, OCaml では引数の値に依存して返り値型が変化するような型 (依存積型) は直接はサポートされていないということである. この問題を解決するために我々はオプションの表現として一般化代数的データ型 (GADTs)[9] を用いることで解決した. また, GADTs で表現できるのは排他的な値のみであるため, オプションの合成を自然に行うことができないという問題がある. 我々はオプション集合の表現をオプションが指定されているかされていないかという情報の組で表し, 引数のオプション集合からコマンド関数の戻り値型を決めるためにある種の型レベル関数を用いることで解決した. 4 節ではこれらの解決法について解説する.

[*2] 現在の OCommand では動的なオプション合成は引数の値が静的に決まるときのみ可能であるため, このように引数の値が動的に決まるような合成は書くことができない. 今後のバージョンではこのような合成も書けるようになる見通しである

3. OCommand DSL

本節では OCommand の DSL について説明する.

OCommand を用いるユーザはコマンドの仕様を DSL で記述し, OCommand 処理系は仕様からコマンド関数やオプション集合の定義などを生成する.

コマンドの仕様とはオプション集合とそれをコマンドに指定した時の出力形式の間の関係である. DSL ではコマンドの仕様としてコマンド名, コマンドの出力に出現しうるフィールドの名前のリスト, コマンドのオプション名とそのオプションを指定した時の出力行レコードの変化, 何もオプションを指定しなかった時の形式を記述する.

図 4 は ls コマンドのための仕様の OCommand DSL での記述例である. 2 行目から 3 行目ではコマンドの出力に

図 4 DSL 例

```
COMMAND ls : BEGIN
  ORDER : {inode; perm; hard; owner; group; size;
           date; time; name} ;;

  OPTION i:ADD {inode:int};;
  OPTION l:ADD {perm:perm; hard:int; owner:string;
                group:string; size:int;
                date:date; time:string} ;;
  OPTION h:MODIFY {size:string} ;;
  DEFAULT:{name:string with delim = ""}
END
```

出現しうるフィールドの名前のリストである. 5 行目から 9 行目はオプション仕様の記述である. 5 行目では i オプションを指定すると出力行レコードに inode という int 型のフィールドが加わることを表す. 6 行目から 8 行目では l オプションを指定すると出力行レコードに perm という perm 型のフィールドや hard という int 型のフィールドなどのファイルのメタデータを表すフィールドが加わることを表す. 9 行目では h オプションを指定すると出力行レコードの size という名前のフィールドが string 型に変化することを表す. 10 行目では何もオプションを指定しないと出力行レコードは name という名前の string 型のフィールドのみ存在することを表す.

DSL で記述されたコマンドの仕様を OCommand で変換すると, コマンドを OCaml 上で利用するためのモジュール (コマンドモジュール) が生成される. 生成されたモジュールの中には以下の定義などが含まれる.

- オプションを指定しないことを表す値 (empty) と, オプションを追加する関数 (add_i, add_l, add_h)
- コマンドを表現する関数 (command)

例えば, 図 4 から生成されたモジュールを用いると,

```
command empty
```

は{name:string} list という型を持ち,

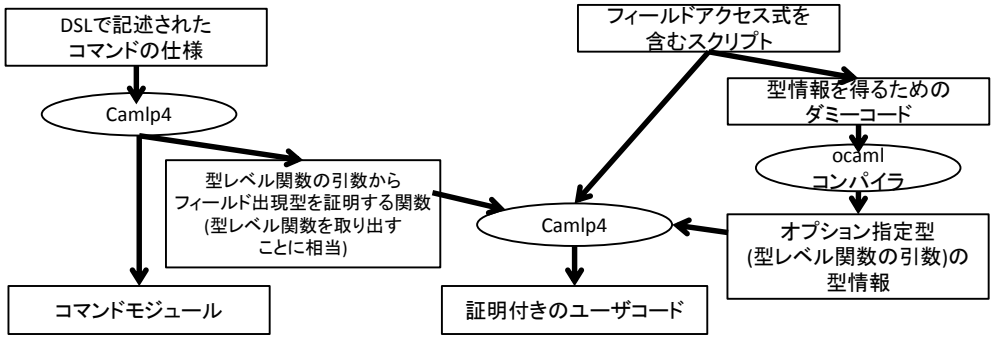

図 5 コード変換手順

```
command (add_l empty)
```

は{perm:perm; hard:int;...; name:string} list という型を持つ. また,

```
command (add_h (add_l empty))
```

は command (add_l empty) の出力行レコードの size フィールドの型が string 型に置き換わったレコード型のリスト型を持つ.

OCommand で生成したコマンドモジュールを利用するスクリプトは, コマンドの出力の各行を表すレコード e のフィールド l の値を取り出すために e..l という構文を用いる. 例えば, command (add_l empty) の出力行レコード line に対して,

```
line..hard
```

というフィールドアクセスを行うと, line のレコード型には hard:int というフィールドが含まれるので,int 型になる. また, 存在しないフィールドへのアクセス

```
line..inode
```

は型エラーになる.

OCommand で扱うことができるコマンドの出力は以下の条件を満たさなくてはならない.

- コマンドの出力は各行がある決まった形式になっていて, その形式が複数行連続するようなものである
- コマンドのフィールドには出現する順序が決まっている

1 つ目の仮定は 2 節で見た ls コマンドのように, 出力はある固定された形式の繰り返しになっているということである.

2 つ目の仮定は, コマンド出力においてフィールドの出現順序が入れ替わることはないということである. 例えば ls コマンドでは, どんなオプションを指定してもフィールドは inode フィールド, perm フィールド, hard フィールド, ..., name フィールドという順序で出現する.

コマンド関数は引数のオプションを指定して適切なコマンドを呼び出し, 結果を DSL で定義された形式で構文解析することでコマンド出力を出力行レコードのリストに変換する.

4. 実現方法

本節で OCommand の実現方法を説明する. 特に次の 2 点について説明する.

- コマンド関数とオプション集合の OCaml プログラムへのエンコード
- 出力行レコードへのフィールドアクセスに対するコード変換

OCommand では図 5 で表される手順でコマンドの仕様からコマンドモジュールを生成し, それを使用するスクリプトから構文拡張を含まない OCaml プログラムを生成する. 本節では図 4 の ls コマンドの仕様と図 1 のフィールドアクセスを例に説明する.

4.1 実現方法のアイデア

コマンド関数は引数の値に依存して出力行レコードの型を変える必要がある. そこで我々はオプション集合を GADTs を用いて指定されているオプションの集合とその型が一対一で対応するように表現した (singleton type). また, オプション集合の型からフィールド出現型を返すような型レベル関数を用いて出力行レコードの型を表した. フィールド出現型とは, 出力行レコードにそのフィールドが存在してその型が τ であるときに τ pre, そのフィールドが存在しない時 abs となるような型である. 出力行レコードの型は各フィールドの出現型のタプル型とみなすことができる.

オプション集合は各オプションのオプション指定値のタプルとして表した. オプション指定値は各オプションについてそのオプションが指定されているかどうかを表す値であり, GADTs を用いて singleton type にした. OCaml は直接型レベル関数をサポートしていないので, 型レベル関数は GADTs を用いて表現した.

出力行レコードのフィールド出現型は引数のオプション

集合の型に依存していて, オプション集合の型は値と一対一で対応するので, オプション集合に応じて出力行レコードの型を変えることができる.

4.2 コマンド関数とオプション集合の OCaml プログラムへのエンコード

OCommand DSL 処理系はコマンドの仕様から以下の定義を生成する.

- オプション集合
- コマンド関数定義
- オプション集合の型からフィールドの出現型を決定する型レベル関数

例えば図 4 のプログラムからは図 6 のコードが生成される. ここでは特に size フィールドに関するコードのみ示した. 実際には他のフィールドに関する定義も生成される.

各オプション o について, オプション指定値は O_On と O_Off の 2 つの値が存在し, それぞれオプションが指定されていることと指定されていないことを表す. それぞれ O_On と O_Off は on と off という型を引数に持つ. 2 行目から 8 行目が opt_i, opt_l, opt_h がそれぞれ i, l, h オプションを表すオプション指定値の定義である. 各オプション集合定義は GADTs を用いている.

オプション集合はこれらのタプルとして表現される. よって指定されているオプションの組み合わせとオプション集合の型は一対一で対応する. また 10 行目から 13 行目が何も指定されていないことを表すオプション集合 (empty) とオプション集合にあるオプションを追加するための関数 (add_i, add_l, add_h) である.

オプション集合の型から出力行レコードの型を静的に決めるために, オプション集合と各フィールドの出現型の関係 (関係型) を GADTs で定義した. 図 6 の 17 行目から 26 行目がフィールド size に対する関係型の定義 rel_size である. rel_size の第 1 引数から第 3 引数はそれぞれ i, l, h オプションのオプション指定型を表している. 第 4 引数が size フィールドの出現型を表している. 例えば Rel_size_2 は l オプションだけが指定された時, size フィールドは存在してその型が int であることを表している. また, Rel_size_3 は l オプションと h オプションが指定された時, size フィールドは存在してその型が string であることを表している.

各フィールドを表す値 (フィールド値) の型はコマンド関数で指定されたオプションのオプション指定型を型引数に持つ. オプション集合型からフィールドの存在の証明できるときのみ, ユーザはフィールド値にアクセスできる. よってフィールド型はオプション集合からフィールドの出現型への型レベル関数として見ることができる. 図 6 の 33 行目から 36 行目がフィールド値 size の型定義である. 型 size の第 1 引数から第 3 引数はそれぞれ i, l, h のオプション指定型である. Size ヴァリアントは引数に型'res field と関係型の組をとる. 出現型'res は存在量化されていて, 外からは隠されている. 型 field の定義は 29 行目から 30 行目にあり, 引数の出現型が τ pre であるときのみ, フィー

図 6 ls の仕様から生成されたコマンドモジュール

```
1  (* option 値定義 *)
2  type on = On ;; type off = Off ;;
3  type _ opt_i = I_On  : on  opt_i
4               | I_Off : off opt_i
5  type _ opt_l = L_On  : on  opt_l
6               | L_Off : off opt_l
7  type _ opt_h = H_On  : on  opt_h
8               | H_Off : off opt_h
9
10 let empty_opt = (I_Off, L_Off, H_Off)
11 let add_i (_, l, h) = (I_On, l,    h)
12 let add_l (i, _, h) = (i   , L_On, h)
13 let add_h (i, l, _) = (i   , l   , H_On)
14
15 (* オプション指定型とフィールド出現型の関係 *)
16 type _ pre = Pre ;; type abs = Abs ;;
17 type ('i, 'l, 'h, 'fld) rel_size =
18   | Rel_size_0 :
19       (off, off, off, abs)        rel_size
20   | Rel_size_1 :
21       (off, off, on,  abs)        rel_size
22   | Rel_size_2 :
23       (off, on,  off, int pre)    rel_size
24   | Rel_size_3 :
25       (off, on,  on,  string pre) rel_size
26  ... (* other 4 combinations of options *)
27
28 (* フィールド型 *)
29 type _ field = FPre : 'a -> 'a pre field
30               | FAbs : abs field
31
32 (* フィールド出現型を決める型レベル関数 *)
33 type ('i, 'l, 'h) size =
34   | Size :
35     ('res field) * ('i, 'l, 'h, 'res) rel_size ->
36       ('i, 'l, 'h) size
37
38 (* コマンド関数定義 *)
39 let commnad : type ti tl th.
40   (ti opt_i * tl opt_l * th opt_h) ->
41    ((ti, tl, th) inode *
42     ... * (ti, tl, th) size * ... *
43     (ti, tl, th) name) list = function
44   | I_Off, L_Off, H_Off -> ...
45   | I_Off, L_Off, H_On  -> ...
46     ... (* other 6 combinations of options *)
47
48 (* フィールドアクセスのための型と関数 *)
49 type (_, _) eq = Eq : ('a, 'a) eq
50 let apply_eq : type a b. (a, b) eq -> a -> b =
51   fun eq v -> match eq with Eq -> v
52
53 (* フィールド出現型の証明 *)
54 let eq_rel_size_off_on_off :
55   type res. (off, on, off, res) res_size ->
56    (res field, int pre field) eq = function
57   | Rel_size2 -> Eq
58
59 (* タプルからフィールドを取り出す *)
60 let get_size_field (_, .., _, size, _, .., _) =
61   size
```

ルドの値をもつような GADTs である.

コマンド関数はオプション集合を引数に取り, フィールド値のタプルである出力レコードを返す. コマンド関数の本体ではオプション集合に対してパターンマッチする. パターンマッチの各節ではコマンドを実行して, 指定されたオプションに対応する形式で出力を構文解析し, 指定されたオプションに対応する出力行レコードのリストに変換して返す (39 行目から 46 行目).

4.3 フィールドアクセス式の変換

コマンド関数の出力行レコードに対するフィールドアクセスは構文拡張を含まない OCaml プログラムに変換される. 本節では図 1 の項 `file..size` を用いて説明する.

出力行レコードに対するフィールドアクセスは出力行レコードのフィールド値 `file` から実際の値 `v` を取り出すプログラムに変換される. `v` の型は存在量化されているので, `v` の型がある具体的な型に等しいことを型レベルで証明する.

OCommand 処理系はまず `file` の型を求めて, 型変換を行うために必要な関数名を決定する. その関数名を使って型検査可能なプログラムを生成する.

4.3.1 型情報の取得

OCommand は型情報を得るために項 `file..size` を

```
match get_size_field v with
| Size (_,  x_...) -> assert false
```

という式にコンパイルする (... の部分には変換前の式の位置情報が_ 区切りで含まれている). `assert false` の型は ∀'a.'a なので, 変換後のプログラムは必ず型検査に成功する.

バージョン 4.0 以降の OCaml コンパイラは `annot` というオプションとともにコンパイルすると, `annot` という拡張子のテキストファイルを出力する. `annot` ファイルには識別子や部分式の型が含まれていて, このファイルを利用すると, 識別子 `x_...` の型を調べることが出来る. 例えば図 1 の `file..size` からは

```
x_... : (off, on, off, res#0) rel_size
```

という型情報が得られる. `res#0` は存在量化された型を表す.

4.3.2 証明のコード生成

`file` という出力行レコードに対して, `size` フィールドを参照するには, `file` に対してパターンマッチで必要な値を取り出す. 型を無視すると以下の式に変換すれば良い.

```
let Size (Fpre res, _) = get_size_field file in res
```

`get_size_field` は図 6 の 60 行目から 61 行目で定義される関数で, 出力行レコードから対応するフィールド値を取り出す関数である. しかし, このプログラムをコンパイル

図 7 `--time-style=full-iso` の形式

```
$ ls --time-style=full-iso
2892425 ... asakura 0    2013-12-23 14:17 aaa
2892606 ... asakura 0    2013-12-23 14:17 bbb
2892607 ... asakura 4096 2013-12-23 14:17 ccc
```

すると `res` の型が存在量化されているため型エラーが起こる.

フィールド型に安全にアクセスするには図 6 の 49 行目から 57 行目の補助関数を用いる. 型 `eq` は 2 つの型の等価性の証人を表す型である. `apply_eq_field` は 2 つの型'a と'b の等価性の証人から型'a の値を型'b に変換する関数である. また `eq_rel_size_off_on_off` 関数はコマンドの仕様から OCommand DSL 処理系が生成する関数で, `rel_size` の初めの 3 つの型引数が `off`, `on`, `off` ならば, 第 4 引数が `int pre` であることを証明する関数である (フィールド出現型の証明).

これらを使って `file..size` は次のように変換される.

```
let Size (x, rel) = get_size_field file in
let eq = eq_rel_size_off_on_off rel in
let FPre res = apply_eq eq x in
res
```

`eq_rel_size_off_on_off` のような適切なフィールド値の証人を使ってコード生成するためには, 4.1 節で述べた `Size` バリアントの第 2 引数 (`rel`) の型が必要である. 求めた型を `eq_rel_`の後に付け加えることで適切なフィールド出現型の証明を得ることができる.

4.4 実装の規模

実装にはコード生成, 変換のために Camlp4 を用いた. コマンドの仕様のためのコード生成器が 600 行程度, フィールドアクセスのためのコード変換器が 300 行程度, その他のライブラリなどが合わせて 200 行程度である.

5. 事例研究

本節では OCommand でいくつかのコマンドに対して簡単なスクリプトを示すことで, OCommand の有用性を考察する.

5.1 MODIFY オプション

図 4 では `l` オプションが指定されている時 `time` フィールドは単に文字列となるように定義した. `ls` コマンドには `time-style` というオプションがあり, 引数に `full-iso` を指定すると, 図 7 のように年表示を含めた ISO 形式で日付を表示するようになる. 本副節ではこのオプションを OCommand で対応することを考える

図 4 を図 8 のように書き換えることでこのオプションを使うことができる. 現在の OCommand では引数付きの

図 8 ls コマンドのための DSL 定義

```
1  type iso_date = {year:int; month:int; day:int}
2  let iso_date str =
3    (* long iso 形式で構文解析する *)
4
5  COMMAND ls : BEGIN
6    ORDER : {inode; perm; hard; owner; group; size;
7             date; time; name} ;;
8
9    OPTION i:ADD {inode:int};;
10   OPTION l:ADD {perm:perm; hard:int; owner:string;
11                 group:string; size:int;
12                 date:iso_date; time:string} ;;
13   OPTION h:MODIFY {size:string} ;;
14   OPTION x:MODIFY {time:iso_time} ;;
15   DEFAULT:{name:string with delim = ""}
16 END
```

オプションやロングオプションには対応していないため, `--time-style=full-iso` を `-x` というオプションとした. 1 行目は full iso 形式に対応する型定義, 2 行目から 3 行目は full iso 形式の `date` フィールドを構文解析するための関数定義である. 14 行目が`--time-style=full-iso` オプションについての仕様である.

このように `h` オプション以外の形式を変更するオプションについても `MODIFY` 形式で扱うことができる.

5.2 複数のフィールドを変化させるオプション

シェルコマンドの中には複数のフィールドを同時に変化させるオプションがある. 例えば `df` コマンドはファイルシステムの使用状況を表示するためのコマンドで, 以下のような出力形式を持つ.

```
$ df
Filesystem 1K-blocks Used Available Use% Mounted on
/dev/sda1  59732092 82460  42492380  26% /
udev        1011416     4   1011412   1% /dev
tmpfs        204304  1072    203232   1% /run
none           5120     0      5120   0% /run/lock
none        1021520     0   1021520   0% /run/shm
none         102400     8    102392   1% /run/user
.host:/    36905468 52960 156952508  34% /mnt/hgfs
```

`df` コマンドには `ls` コマンドと同様に `h` オプションがあり, これは出力中の各数値を `K`, `M`, `G` を使った形式に変更する.

```
$ df -h
Filesystem Size  Used Avail Use% Mounted on
/dev/sda1   57G   14G   41G  26% /
none       4.0K     0  4.0K   0% /sys/fs/cgroup
udev       988M  4.0K  988M   1% /dev
tmpfs      200M  1.1M  199M   1% /run
none       5.0M     0  5.0M   0% /run/lock
none       998M     0  998M   0% /run/shm
none       100M  8.0K  100M   1% /run/user
.host:/    226G   77G  150G  34% /mnt/hgfs
```

また, `df` コマンドには `T` オプションがあり, これは下の `Type` フィールドのように, ファイルシステムの種類も出力する.

```
$ df -T
Filesystem Type   1K-blocks ... Mounted on
/dev/sda1  ext4    59732092 ... /
none       tmpfs          4 ... /sys/fs/cgroup
udev       devtmpfs 1011416 ... /dev
tmpfs      tmpfs     204304 ... /run
none       tmpfs       5120 ... /run/lock
none       tmpfs    1021520 ... /run/shm
none       tmpfs     102400 ... /run/user
.host:/    vmhgfs 236905468 ... /mnt/hgfs
```

`df` コマンドのオプションを扱えるような DSL 定義は図 9 のようになる.

図 9 df コマンドのための DSL 定義

```
1  COMMAND df : BEGIN
2    ORDER : {filesystem; type; size; used;
3             available; use; mounted_on} ;;
4
5    OPTION T:ADD {type:string};;
6    OPTION h:MODIFY {size:string; use:string;
7                     avail:string};;
8    DEFAULT:{filesystem:string; size:int;
9             used:int; available:int; use:int;
10            mouted_on:string}
11 END
```

また, 上の DSL をつかって, 物理デバイスの使用サイズを `K`, `M`, `G` 付きの形式で出力する関数を書くと以下のようになる.

```
1  let is_physical_device name =
2    (* 与えられたファイルシステムが
3       物理デバイスであるか *)
4
5  let disk_use () =
6    let open Df in
7    let fss = command (add_h (add_t empty)) in
8    List.iter fss ~f:(fun fs ->
9      if is_physical fs..type then
10       print_endline fs..used)
```

上のプログラムでは `df` コマンドを起動して, コマンドの実行結果の各行に対して物理ファイルシステムであるものだけその使用サイズを足しあわせている. 物理デバイスであるかどうかの判定は `type` フィールドを `is_physical_device` 関数に渡すことで判定している (9 行目).

このように `df` コマンドのように複数のフィールドを変化させるようなオプションを持つコマンドについても適切に扱うことができる.

5.3 形式を変更しないオプション

`ps` コマンドは現在実行中のプロセスの情報を出力するコマンドである.

```
$ ps
 PID TTY          TIME CMD
2834 pts/3    00:00:00 bash
2983 pts/3    00:00:00 ps
```

また f オプションや l オプションをつけると, より詳細な情報を出力するようになる.

```
$ ps -f
UID      PID PPID C STIME TTY       TIME CMD
asakura 2834 2558 0 03:10 pts/3 00:00:00 /bin/bash
asakura 2985 2834 0 03:11 pts/3 00:00:00 ps -f
$ ps -l
F S  UID  PID PPID C PRI NI ADDR SZ WCHAN TTY   ...
0 S 1000 2834 2558 0  80  0 -  6866 wait  pts/3 ...
0 R 1000 2986 2834 0  80  0 -  3791 -     pts/3 ...
```

f, l オプションを合成すると, 以下の様にそれぞれの出力に出現するフィールドの共通部分をとった出力になる.

```
$ ps -lf
F S UID      PID PPID C PRI NI ADDR SZ WCHAN ...
0 S asakura 2834 2558 0  80  0 -  6866 wait  ...
0 R asakura 2987 2834 0  80  0 -  5899 -     ...
```

また, 現在のユーザだけでなく, すべてのユーザのプロセスに関する情報を見るための e オプションがある.

5.3.1 実装

ps コマンドは OCommand で以下のように扱うことができる.

```
1  COMMAND ps : BEGIN
2    ORDER : {f; s; uid; pid; ppid; c; pri; ni;
3             addr; sz; wchan; stime; tty;
4             time; cmd;} ;;
5
6    OPTION e:ADD {} ;;
7    OPTION f:ADD {uid:string; ppid:int; c:int;
8                  stime:time} ;;
9    OPTION l:ADD {f:int; s:string; uid:int; ppid:int;
10                 c:int; pri:int; ni:int;
11                 addr:string;sz:int;
12                 wchan:string} ;;
13   DEFAULT:{uid:int; tty:string; time:string;
14            cmd:string with delim = ""}
15 END
```

e オプションは何もフィールドを追加しない ADD 形式のオプションとして扱っている.

例えば上の定義を使ってある名前のプロセスとその子孫プロセスの名前を集める関数 collect_descendants は図 10 のようになる. 図 10 の 3 行目で ps コマンドを起動している. 4 行目から 5 行目では name フィールドが引数 name と等しい行を取り出している. iter 関数は ps コマンドの各行から ppid フィールドが受け取ったプロセスのリストの pid に等しいものを抽出して, それが空リストを返すまで繰り返しリストに追加している.

図 10 collect_descendants

```
1  let collect_descendants name =
2    let open Ps in
3    let ps = command (add_e (add_f empty)) in
4    let ps1 = List.filter ps ~f:(fun p ->
5      p..name = name) in
6    let rec iter ps1 =
7      let ps2 = List.filter ps ~f:(fun p ->
8        List.exists ps1 ~f:(fun pp ->
9          not (List.mem ps1 p) &&
10         pp..pid=p..ppid)) in
11     if ps2 = [] then ps1
12     else List.append ps1 (iter ps2) in
13   iter ps1
```

図 10 中では ps コマンドの出力の各行を name フィールドが string 型, pid, ppid フィールドが int 型であるレコードとして扱うことができている.

何も形式を変更しないオプションに対しては, 何も追加しない ADD 形式を使うことで OCommand で扱うことが出来る.

6. 関連研究

Scheme 上でシェルプログラミングを行うためのライブラリに Scsh[7] がある. Scsh ではシェルのパイプに相当する機能を用いて外部コマンドの結果を Scheme の関数に渡すことや, Scheme 関数の結果を外部コマンドに渡すことができる. Scheme は動的型付き言語なので OCommand のように出力行レコードへのアクセスに対する型安全性は保証していない.

OCommand と同じように OCaml 上でシェルプログラミングを行うためのライブラリとして Shcaml[3] がある. Shcaml では OCaml でコマンドを扱うために典型的な形式に対しては, コマンドの形式を記述したモジュールを受け取って構文解析器やフィールドの値を取り出すための関数が含まれるモジュールを作成するファンクターを提供している. Shcaml では例えば Linux の /etc/passwd 形式や, 各行が key=value という形式や, CSV などに対応している. OCommand との違いとして, OCommand では形式の各要素に対する型安全性を目標にしている点が挙げられる.

また, OCommand で扱った型レベルの計算を扱う機構として Haskell の関数従属性 [5], 型族 [6], また C++の Template metaprogramming[1] などがある.

7. 結論・課題

本稿では, オプションがあるコマンドを OCaml のような型安全な関数型言語で扱うための DSL を提案した. 型安全性を保証するために GADTs を用いて型レベルの関数を OCaml 上で実現する方法について述べた. また OCommand では型レベルの関数を GADTs を利用して表現し, 事例研究ではいくつかのコマンドに対して正確に型チェックが行えるようになったことを確かめた.

以下では今後の課題について説明する.

7.1 より柔軟な形式への対応

ls コマンドは l オプションを付けて実行すると, 図 11 のように先頭行に total 8 のような出力を表示する このように出力が, ある形式の繰り返しにはなっていないコマンドに対して, OCommand では構文解析に失敗した行は

図 11 ls -l の出力

```
$ ls -l
total 8
-rw-rw-r--. 1 asakura asakura ... aaa
...
```

結果のリストに含めないという対応をしている. しかしそのようにすると ls -l の 1 行目もユーザが利用したいときに対応できない. このような出力を持つようなコマンドに対してもうまく扱えるような型付けと DSL を定義をすることは今後の課題である.

7.2 引数付きのオプションへの対応

多くのコマンドには引数を取るオプションが存在する. 例えば, ps コマンドには-tty オプションが存在し, これは端末名を引数にとって, その端末で実行されているプロセスの情報のみを出力する.

```
$ ps -tty ttys001
   PID TTY           TIME CMD
15819 ttys001    0:00.07 -zsh
```

このように引数をとるようなオプションについては OCommand では未対応であるが, このオプションに関しては引数に対しては出力の形式が変化しないので, OCommand を単純に拡張することで対応できる.

しかし, オプションがとった引数に対して出力の形式が変化するような場合は OCommand では単純には対応できない.

例えば, ls コマンドの time-style オプションは, time フィールドを表示する形式の名前を引数にとる.

```
$ ls -l --time-style=iso
total 8
2892425 ... asakura 0    12-23 14:17 aaa
2892606 ... asakura 0    12-23 14:17 bbb
2892607 ... asakura 4096 12-23 14:17 ccc
$ ls -l --time-style=iso-full
total 8
2892425 ... asakura 0    2013-12-23 14:17 aaa
2892606 ... asakura 0    2013-12-23 14:17 bbb
2892607 ... asakura 4096 2013-12-23 14:17 ccc
```

iso を引数に渡した場合と full-iso を引数に渡した場合で time フィールドの形式が異なっている. time フィールドの型は--time-style オプションに渡した引数が iso の場合は

```
1 type date = {month:int; date:int}
```

full-iso の場合は

```
1 type date_with_year {year:int; month:int; date:int}
```

のような型を与えるべきである.

このようなオプションを扱えるような型付けを考えるには, オプション指定値のバリアントが引数 v を取れるようにして, オプション指定値の型引数に引数 v の値の型を依存させることが必要となる. GADTs を利用することでこのようなオプションも対応できる見込みである.

7.3 ジョブコントロール

シェルスクリプトには非同期にコマンドを実行するための機能や, コマンドの結果を別のコマンドに依存させる機能が用意されている.

例えばバックグラウンドでコマンドを実行するための&や, 出力を別のコマンドに渡すためのパイプ|, 前のコマンドが成功したとき, 次のコマンドを実行するという意味を持つ&&などがある.

OCommand ではこれらの機能を扱うことはできないが, これらの機能も OCaml 上で扱えるようになると非常に有用であると考えられる.

OCaml 上で非同期処理を行うためのライブラリとしては Lwt[8] や Async[4] が有名である. OCommand とこれらのライブラリを利用して, OCaml 上でさらに自然にシェルプログラミングが行えるようにすることは今後の課題である.

7.4 動的なオプションの合成

本稿で説明した型付けでは合成されるオプションが動的に決まるようなオプションの合成を行うことができない. 例えば 2.4 節で挙げたような

```
1 let collect_old_files default_opt size =
2   let opt = default_opt (add_i (add_l empty)) in
3   let files = ls opt in
4   ...
```

という関数に正確に型を付けるには, “引数 default_opt は i オプション, l オプションと合成されたとき, size という int 型のフィールドと, name という string 型のフィールドを持つ” という制約を型で表現しなくてはならない. しかし, 現在の OCaml ではそのような制約を表現することは難しい. Haskell の型クラス, 関数従属性などの機能を用いると自然にそのような制約を表すことが出来ることを確かめたが, OCaml の型システムでそのような制約を表現することが今後の課題である.

参考文献

[1] Alexandrescu, A.: *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001).

[2] Greg Weber, P. R.: shelly, http://hackage.haskell.org/package/shelly.

[3] Heller, A. and Tov, J. A.: Caml-Shcaml: An OCaml Library for Unix Shell Programming, *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, pp. 79–90 (2008).

[4] Jane Street Capital LLC: async, https://github.com/janestreet/async.

[5] Jones, M. P.: Type Classes with Functional Dependencies, *Programming Languages and Systems Lecture Notes in Computer Science*, pp. 230–244 (2000).

[6] Schrijvers, T., Peyton Jones, S., Chakravarty, M. and Sulzmann, M.: Type Checking with Open Type Functions, *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, New York, NY, USA, ACM, pp. 51–62 (online), DOI: 10.1145/1411204.1411215 (2008).

[7] Shivers, O.: A Universal Scripting Framework or Lambda: the ultimate "little language", *Proceedings of ASIAN'96*, Springer-Verlag, pp. 254–265 (1996).

[8] Vouillon, J.: Lwt: A Cooperative Thread Library, *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, pp. 3–12 (2008).

[9] Xi, H., Chen, C. and Chen, G.: Guarded Recursive Datatype Constructors, *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 224–235 (2003).