ジェネリックプログラミングを用いた証明木 GUI ライブラリの実装

松本 晴香　(指導教員：浅井 健一)

1　はじめに

証明木は，ある論理体系の元で成り立つ命題が推論規則や公理によって導かれる過程を図に表し，その正しさを検証するときに用いられる．証明木は単純でわかりやすいが，証明木を手で描くことには，完成させるために必要なスペースを予め正確に知ることができない等，不便な場面が多い．これらの問題を解決するためには，コンピュータ画面上に証明木を描画できる GUI が望まれる．これについては先行研究である櫻井 (2011)[4] により開発されたライブラリ『Miki β』がある.『Miki β』は OCaml や js_of_ocaml により実装されており，ユーザは各項の定義，推論規則を OCaml プログラムにより定義し，GUI への入力のための字句解析 (lexer)，構文解析 (parser) を実装するだけで，GUI 作成の知識がなくても，証明木を構築する GUI を得ることができるという特色がある．しかし，先行研究には次のような欠点がある．

- ユーザが定義するべきプログラム量が多い．
- 相互再帰する項を定義できない．

そこで本研究では，ジェネリックプログラミング [3] を応用し『Miki β』を改良することを試みる．本研究が用いるジェネリックプログラミング手法は Haskell や Agda で実装される言語の拡張方式であり，データ構造に依存しない汎用的な関数の定義を可能にする．

2　先行研究

まず先行研究の手法を説明する．先行研究は two-level types[5] を用いて実装される．この手法は，ある再帰的な型を定義するときに，再帰を閉じた部分と，再帰とは無関係なデータの構造を表す部分の二つに分けることで，汎用的なプログラムの定義を可能にするものである．ユーザは GUI に載せる各項に対して，定義をモジュール単位で行い，以下のインタフェースを定義しなければならない．

`type 'a t`
　プログラム上の項の型の定義を表すヴァリアント型．再帰部分は`'a` 型で表される．

`map : ('a →'b) →'a t →'b t`
　引数に渡した関数を再帰部分に適用する関数．

`unify : ('a →'a → unit) →'a t →'a t → unit`
　二つの項に対し，単一化を行う関数．

`draw : id_t t → id_t`
　項を画面にどのように描画するかを定義する関数．

`int` 型を値に持つ二分木を例にとると，二分木を定義したモジュールは図 1 のようになる．two-level types を用いたことにより，再帰部分に対しての処理をライブラリ内に閉じることが可能になっているが，関数 `map`，`unify` は項の型に依存してしまうため，項の追加の度に定義される必要があり，ユーザの書くプログラム量が多い主な原因である．

また，項の定義は OCaml のモジュール単位で行われるため，相互再帰する項を定義することができない．

```
module Tree_e = struct
  'a t = Empty
       | Node of 'a * int * 'a
  let map f t = match t with
      Empty -> Empty
    | Node (l, i, r) -> Node (f l, i, f r)
  let unify f t1 t2 = match (t1, t2) with
      (Empty, Empty) -> ()
    | (Node (l1, i1, r1), Node (l2, i2, r2))
      when i1 = i2 -> f l1 l2; f r1 r2
    | _ -> raise Unify_Error "Tree"
  let draw t = match t with
      Empty -> create_str "empty"
    | Node (l, i, r) ->
      combineH [create_str "node "; l;
      create_str " ";
      create_str (string_of_int i);
      create_str " "; r]
end
```

図 1: 先行研究にて，構文 $i \in Int, t \in Tree := empty \mid node\ t\ i\ t$ で定義される二分木を用いるときに，定義しなければいけないモジュールの例.

例えば，以下のような構文を持つ項は定義できない．

$$v \in Value \quad := x \mid \lambda x.\ e$$
$$e \in Expr \quad := v \mid e\ e$$

3　ジェネリックプログラミングの応用

PolyP[1] は，5 つのコンストラクタ: ユニット (U)，パラメータ (P)，再帰 (I)，和 ($\oplus$)，積 ($\otimes$) で構成される Code 型と呼ばれるデータ型を用いて，言語の組み込みの型定義を使わずに，データの型付けを擬似的に行う．例えば，二分木と同等な型は $U \oplus I \otimes P \otimes I$ のように表せる．U が引数のないコンストラクタ Empty，$I \otimes P \otimes I$ がコンストラクタ Node が持つ三つ組に対応しており，二つを和 ($\oplus$) でつなぐことで，ヴァリアントを表現している．型の表現を単純な要素のみで構成することにより，データ構造に依存しない汎用的な関数の定義が可能になる．2 節で述べた項の定義をこの Code 型を用いて行えば，関数 `map`，`unify` の汎用的な定義が期待できる．

次に，相互再帰する項のサポートを考える．これについてはそれぞれの項に一意のインデックスを割り振る Indexed Functor[2] がある．再帰を表すコンストラクタ I にインデックスを引数として付加し，Code 型 I n を「インデックス n を割り当てられた項について再帰する型」とする手法である．

PolyP を含むジェネリックプログラミングは Haskell での実装が主だが，OCaml による実装で手法の導入を行う．OCaml で Code 型と同等の働きをするデータ型 `code_t` は図 2 のように定義できる．さらに，`code_t` 型によって表現された任意の型に属するような項`'a type_f` を図 3 のように定義した．これを用いると，二分木の項 `Node (Empty, 5, Empty)` は，`'a type_f` の再帰を閉じた型 `type fix_t = Fix of fix_t type_f` を与えたとき，`Fix (Sum (Right`

```
type code_t = U | I of int | PI | PS
            | Plus of code_t * code_t
            | Times of code_t * code_t
```

図 2: OCaml で定義した Code 型.

```
type ('a, 'b) sum_t = Left of 'a
                    | Right of 'b
type par_t = Int of int | String of string
type 'a type_f =
  Unit of unit | Rec of 'a | Par of par_t
| Sum of ('a type_f, 'a type_f) sum_t
| Pair of 'a type_f * 'a type_f
```

図 3: code_t 型で表現された型に属する項を表す'a type_f 型

(Times (Rec (Fix (Sum (Left (Unit ()))))), Times (Par (Int (5)), Rec (Fix (Sum (Left (Unit ()))))))) と表現できる．2 節での項の型定義を code_t 型，その項を'a type_f 型で表し，その際，相互再帰サポートのため，各項の定義はモジュールに閉じずに行う．これにより，関数 map，unify を汎用的に定義できるようになり (図 4)，二分木は図 5 のように定義すればよい．

4 その他の改善

GUI への入力のために字句解析と構文解析の実装をユーザに必要としていたが，本研究では，入力のための汎用的なインタフェースを作成した．(図 6) これが可能になったのは，ジェネリックプログラミング手法を導入したことにより，再帰部分を含めた期待される入力の型の情報をヴァリアント型として得られるようになったためである．

一方，ジェネリックプログラミング手法の導入により新たに生じた問題点がある．'a type_f による項の表現は，OCaml の組み込みのデータ型を用いるよりも直感的でなくわかりにくい．例えば，図 5 の関数 draw での match 文のパターンは複雑で，間違えやすい．これを回避するため，図 7 のような定義方式に変更し，改良を行った．

5 コード量の変化

相互再帰をする構文を持たない各論理体系で，ユーザ定義のコードの総量の比較を行った．本研究では，コード量が全体を通じて半分以下になっている．

	先行研究		本研究	
論理体系	行数	語数	行数	語数
ペアノ演算	141	596	55	277
自然演繹	387	1968	137	806
環境を持つ四則，比較演算	726	3842	254	1620

6 まとめ

『Miki β』にジェネリックプログラミングを応用し，関数 map，unify を汎用的に定義し，相互再帰する項の定義を可能にした．汎用的な入力インタフェースを構築し，ユーザによる字句解析と構文解析の実装を不要にした．その結果，ユーザが定義しなければならないプログラム量を半分以下に減らすことができた．

```
(* unify : code_t ->
           (code_t -> fix_t -> fix_t -> unit) ->
           fix_t type_f -> fix_t type_f -> unit *)
let rec unify code f fix1 fix2 =
match (code, fix1, fix2) with
  (U, Unit (), Unit ()) -> ()
| (I n, Rec (d1), Rec (d2)) -> f (get_tcode n) d1 d2
| (PI, Par (Int (i1)), Par (Int (i2))) when i1 = i2 -> ()
| (PS, Par (String (s1)), Par (String (s2))) when s1 = s2 -> ()
| (Plus (c1, c2), Sum (Left (d1)), Sum (Left (d2))) ->
    unify c1 f d1 d2
| (Plus (c1, c2), Sum (Right (d1)), Sum (Right (d2))) ->
    unify c2 f d1 d2
| (Times (c1, c2), Pair (d1, d2), Pair (d3, d4)) ->
    unify c1 f d1 d3; unify2 c2 f d2 d4
| _ -> raise (Unify_Error "unify")
```

図 4: code_t, 'a type_f を用いて定義された汎用的な関数 unify. get_tcode は，n 番目の項の型を表した code_t 型の値を返す関数である．

```
(* tree_code : code_t *)
let tree_code =
Plus (U, Times (I 0, Times (PI, I 0)))
(* tree_draw : id_t type_f -> id_t *)
let tree_draw d = match d with
  Sum (Left (Unit ())) -> create_str "zero"
| Sum (Right (
  Pair (Rec (l), Pair (Par (Int (i)), Rec (r))))) ->
  combineH [create_str "node "; l;
  create_str " "; create_str (string_of_int i);
  create_str " "; r]
```

図 5: code_t, 'a type_f で定義した二分木．図 1 と比較して，関数 map，unify の定義が不要になった．

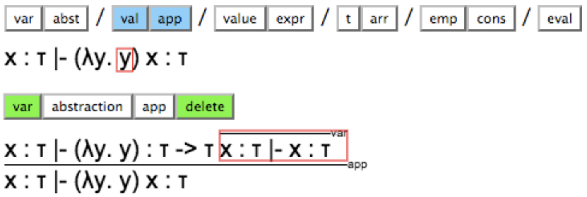

図 6: 汎用的な入力インタフェースの様子 (上)．確定ボタンを押せば，画面下に入力した項がコピーされ，証明木が作れる．(下)

```
let tree =
[("empty", U, default);
 ("node", I 0 ** PI ** I 0,
  fun t -> combineH [create_str "node "; t 0;
  create_str " "; t 1; create_str " "; t 2])]
```

図 7: 図 5 をユーザに使いやすいように改めたもの．コンストラクタ名，code_t 型による型宣言，その描画関数の三つ組のリストで定義する．描画関数は default で省略できる．node の描画関数を default としても同じように描画される．

参考文献

[1] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL '97*, pages 470–482, 1997.

[2] Andres Löh and José Pedro Magalhães. Generic Programming with Indexed Functors. In *WGP '11*, pages 1–12, 2011.

[3] José Pedro Magalhães and Andres Löh. A Formal Comparison of Approaches to Datatype-Generic Programming. volume 76 of *EPTCS*, pages 50–67, 2012.

[4] Kanako Sakurai and Kenichi Asai. Mikibeta : A General GUI Library for Visualizing Proof Trees. In *LNCS 6564*, pages 84–98. 2011.

[5] Tim Sheard and Emir Pasalic. Two-level types and parameterized modules. In *JFP 14:5*, pages 547–587, 2004.