Mikiβ における相互再帰の実装

大石　美緒 (指導教員：浅井 健一)

1　はじめに

GUI 上で証明木を作成することのできる Mikiβ [2] では、型付けしたい項を入力すると適用可能な規則が表示され、その中から適用したい規則を選択していくと、それに従って証明木がコンピュータ上に書かれていく。これを用いると、証明木を書く時間を短縮したり、ミスを減らしたりすることができ、初学者の理解を助けることができる。

しかし問題点として、新しいデータ型や規則のもとで使用できる Mikiβ を作成する時にユーザが書く必要のあるコードの量が多い点や、相互再帰するようなデータ型は Mikiβ では扱うことができない点が挙げられていた。

これらの問題点を generic programming [1] と GADT を用いることで改良した。

相互再帰する型の例として、本論文では次のようなデータ型を例として用いる。

```
type zig_t = Zig of zag_t
           | End
and  zag_t = Zag of zig_t
```

以下、このデータ型のことを ZigZag データと呼ぶ。

2　Mikiβ の構成

Mikiβ は、ライブラリとユーザが作成するファイルから構成される。ライブラリでは、GUI を作るために必要な関数を提供している。そのため、GUI に関する知識はユーザに要求されない。ユーザが作成するファイルは、ユーザが OCaml 上で書く必要があり、証明木のデータ定義、map 関数、単一化関数、出力関数、推論規則の定義からなる。ユーザが定義するファイルはデータ定義に依存するためユーザ自身が定義していたが、データ定義以外の関数は機械的に生成可能な部分が多いため、自動生成できるようにすることでユーザの負担を減らすように改良していく。

3　generic programming

3.1　generic programming とは

例えば ZigZag データでは、それぞれの型特有の情報として以下のようなものが含まれていた。

- `zig_t`、`zag_t` という型の名前
- `Zig`、`Zag`、`End` というコンストラクタ名
- 引数の型

この型特有の情報を用いてコードを記述していたため、型が変わるたびにユーザが毎回新たにコードを書く必要があった。

generic programming は、型の構造に対して処理を書くようなプログラミング方法のことである。つまり、上記のような特有の情報を用いない形で型の情報を記述する。具体的には、型の構造を表す値を定義し、その値を使って型の構造を表すことで型の情報を一般化する。すると、一つのコードで様々なデータ型に対して処理が行えるようなる。

3.2　`code_t` を用いた実装

型の構造を表す値として、以下のように `code_t` を定義する

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

`U` は Unit を表す。`I` は再帰を表す。`I` の int 型の引数で再帰先を指定する。例えば ZigZag データにおいては、`zig_t` は 0、`zag_t` は 1 として再帰先の型を表すように使う。`Plus` はヴァリアント型を表し、`Times` は組やレコード型に対応する。

これらを用いて ZigZag データを表すと、

```
let zigzag = Plus (Plus (I 1, U), I 0)
```

となる。`Plus (I 1, U)` の部分が `zig_t` を表していて、`I 0` の部分は `zag_t` を表している。これは、型特有の情報を含まない形となっており、一般的化されている。

`code_t` は型でなく値であるため、新たに定義した ZigZag データも値であり型ではない。そのため OCaml 上で新たに定義した、一般化された ZigZag データに対して直接処理を書くことはできない。処理を記述するためには、値である `code_t` に対応する、実際の OCaml 上での型を定義する必要がある。

実際の型である `'a type_f` を以下のように定義する。

```
type 'a type_f =
    Unit of unit
  | Rec of int * 'a
  | Sum of ('a type_f, 'a type_f) sum_t
  | Pair of 'a type_f * 'a type_f
```

`Unit` に対応しているのが `U`、`Rec` に対応しているのが `I`、`Sum` に対応しているのが `Plus`、`Pair` に対応しているのが `sum_t` となっている。なお、`sum_t` は次のように定義される。

```
type ('a, 'b) sum_t = Left of 'a
                    | Right of 'b
```

処理を書くためには `code_t` と `'a type_f` の両方を使う必要がある。

4　GADT(Generalized Algebraic Tata Type)

3 節における `code_t` と `'a type_f` はそれぞれ対応しているが、その対応関係はユーザに任せられている状態であり、ユーザが間違った対応関係でプログラムを書いてしまうと、Mikiβ は正しく動作しない可能性がある。GADT を導入することでこの対応関係を Mikiβ のシステムの方できちんと保証するように改良する。

4.1　GADT とは

GADT とは、構成子によって型パラメータに制約を加えることができるようヴァリアント型を拡張したも

のであり、GADT を用いると、意図しないような型を定義することが不可能となる。つまり、code_t と'a type_f の対応関係は一意に定まり、Mikiβ 上で対応関係が正しいことが保証される。

4.2 GADT を導入する

GADT を実際に導入するため、まず値だった code_t を型レベルに引き上げる。

```
type u_t = U
type i_t = I
type ('a, 'b) plus_t = Plus of 'a * 'b
type ('a, 'b) times_t = Times of 'a * 'b
```

ここで I の引数の int がなくなっている。その変わり、以下で定義する ('a, _)type_f 内の Rec に int を追加し、再帰先を指定するようにする。

type_f に GADT を導入して記述すると以下のようになる。

```
type ('a, _) type_f =
   Unit : unit -> ('a, u_t) type_f
 | Rec  : int * 'a -> ('a, i_t) type_f
 | Sum  :
  (('a, 'c1) type_f,
   ('a, 'c2) type_f) sum_t
   -> ('a, ('c1, 'c2) plus_t) type_f
 | Pair :
  ('a, 'c1) type_f * ('a, 'c2) type_f
   -> ('a, ('c1, 'c2) times_t) type_f
```

変更前では処理を書くためには code_t と type_f の両方が必要であったが、code_t を型レベルに引き上げ、GADT を導入したことにより type_f のみで処理を書けるようになった。

5 コード

ユーザが OCaml 上で記述していた部分に対し、実際に Mikiβ に行った変更を以下に示す。

5.1 証明木のデータ定義

generic programming を用いて一般化し、以下のように記述するように変更した。

```
type t = ((i_t, u_t) plus_t, i_t) plus_t
```

5.2 map 関数、単一化関数

それぞれのデータ型について、毎回ユーザが map 関数と単一化関数を書いていたが、generic programming を導入し型の構造について処理を書くように変更したことによりユーザが書く必要がなくなったため、ライブラリに移すことができた。これにより、ユーザが書くコードの量を大きく減らすことができる。

また、単一化関数の変更により、相互再帰する型も扱えるよう改良された。Mikiβ にはユーザが定義する型である S.t を持つ Fix とメタ変数 Meta なる型 fix_t があり、Mikiβ の各関数は fix_t を用いて呼び出される。これを使うことにより、再帰で繋がっているデータを分離することができる。fix_t は以下の通りである。

```
type fix_t =
   Fix of (fix_t, S.t) type_f
 | Meta of string * fix_t option ref
```

Fix、Meta それぞれに再帰の番号を追加し、その番号を比較し、異なっていたらエラーを起こすように変更した。GADT を使用することで、型が合わない場合、コンパイル時にエラーが起こり検出できるようになっているが、相互再帰の再帰先が合わないというエラーに関しては、単一化関数を使用する際の実行時にチェックするようになっている。

5.3 出力関数

証明木によって最終的に表示されるジャッジメントの形が異なるため、generic programming を用いても、毎回ユーザが書く必要がある。そのため、ライブラリに移すことはできない。

5.4 推論規則の定義

メタ変数を生成し、それを用いて公理や推論規則を定義する。メタ変数を生成する際、そのメタ変数自体の再帰が何番目かを int で保持するようにする。それぞれのメタ変数が再帰の番号を持っている状態で公理や推論規則を書くことで、再帰先を指定できるようになった。ZigZag データの中の Zig についての推論規則を Mikiβ 上で記述すると以下のようになる。

```
let zig n = Term.make_fix 0
 (Sum (Left (Sum (Left (Term.make_rec 1 n)))))
let t_zig () =
 let tm1 = Term.make_meta 1 "t" in
 Infer(zig tm1, [Infer(tm1, [])])
```

まず Zig のデータを作成する。Term.make_fix で fix_t 型のデータを生成し、引数の 0 で zig の再帰の番号を指定する。Term.make_rec 1 n では再帰先の番号が 1 であることを表す。次に、t_zig で Zig の推論規則を生成する。make_meta で再帰の番号が 1 のメタ変数を生成し、Infer で推論規則を生成する。

推論規則も証明木によって異なるため、ユーザが記述する必要があり、ライブラリに移すことはできない。また、再帰する型の数が多くなると推論規則を記述するために書く必要のあるコード量が多くなってしまい使いづらく、改良が必要である。

6 まとめと今後の展望

generic programming と GADT を用いて、ユーザ定義の部分を減らし、相互再帰の番号をデータに追加することで、相互再帰を実現した。

しかし、ユーザが書くところに改良後でも煩雑な部分が多くなっている。これを、OCaml 上でユーザが定義する必要があるデータ定義や、データ定義に依存してしまう出力関数、推論規則の入力も全て GUI 上で入力できるように改良したいと考えている。GUI 上で全てが完結するようにし、ユーザが OCaml の知識が全くなくても Mikiβ を利用したり、新たな証明木を作成できるようなものを目指していきたい。

参考文献

[1] J. P. Magalhaes and A. Loh. A formal comparison of approaches to datatype-generic programming. *Proceedings of the Fourth Workshop on Mathematically Structured Functional Programming (MSFP 2012)*, pp. 50–67, March 2012.

[2] Kanako Sakurai and Kenichi Asai. Mikibeta: A general gui library for visualizing proof trees. *20th International Symposium on Logic-Based Program Synthesis and Transformation*, pp. 1–15, July 2010.