証明木を描くための汎用的なGUIライブラリMikiβ

理学専攻情報科学コース 櫻井 加奈子　(指導教員：浅井 健一)

1　はじめに

型推論等で用いられる証明木は，型システムを用いた項の型付けを検証したり，その挙動を見るために描かれる．証明木は単純な構造をしているため，初学者が推論規則を理解する手助けにもなるが，証明木を手で描くことには次のような問題点がある．

- 証明木は縦横に広がるため、予め描く証明木に必要なスペースを知ることができない
- 推論規則が適用される度に多くの項を書き写さなければならず，ミスが生じる
- 単一化 (unification) によって，多くの項を書き換えなければならない

これらの問題を解決するためには，証明木をコンピュータの画面上に描くGUIプログラムを作成すれば良いが，GUIプログラムを作成することは容易ではない．また一度GUIを作ったとしても，他の型システムに利用することはできない．これはプログラミング言語研究者が種々の型システムを作成し，その証明木を描いていることを考えると，あまり好ましい状況ではない．

2　Mikiβの概要

本稿で紹介する『Mikiβ』は，ユーザが推論したい式の定義と推論規則を与えることで汎用的にGUIの構築を行うシステムである．このシステムはOCamlとLablTkを用いて開発しているため，ユーザにOCamlの基本的な知識を求めているが，GUIを作るために必要な関数はGUIライブラリとして提供しているため，GUIに関する知識は要求しない．

Mikiβはシステムの汎用性を実現するために，*two-level types* [2] を用いて実装している．この手法は，一つの型を汎用的に利用出来る部分と実装しようとしているシステムに依存する部分の二つの型に分け，型に従ってプログラムも二つの部分に分ける．これにより，片方のプログラムを他のプログラムと入れ替えたり，他方のプログラムを汎用的に扱ったりすることが可能になる．two-level typesをMikiβに導入する利点は，ユーザがGUIについてほとんど考えることなく，実装したい型システムのコードを書くことができるという点と，Mikiβを汎用的なプログラムとして扱うことができるため，任意のユーザ定義の型システムに対してGUIを提供できるという点である．

本稿では主に，Mikiβのシステム概念とユーザから見たMikiβの具体的な利用法について述べる．Mikiβと各型システムのユーザコードは `http://pllab.is.ocha.ac.jp/~asai/MikiBeta/` で見ることができる．

3　two-level types の汎用プログラミング

two-level typesを用いた汎用プログラミングでは，一つの型を二つの型に分けるために，相互再帰を用いる．単純型付きラムダ計算 (STLC) [1] の型定義を用いて，two-level typesでの型定義をOCamlで示す．ここで，`'type_t` は型変数，`S` はユーザに定義される引数モジュール，`id_t` は画面上に描かれるオブジェクトの識別子を表す．

- ユーザが定義する型

```
type 'type_t t =
  TVar of string
| Fun of 'type_t * 'type_t
```

- Mikiβに定義されている型

```
type fix_t =
(* 固定項 *)
  Fix of fix_t S.t * id_t list ref
(* メタ変数 *)
| Meta of string * fix_t option ref
          * id_t list ref
(* 任意コード *)
| Code of (unit -> id_t) * code_t ref
          * id_t list ref
and code_t =
  C of (unit -> fix_t)
| V of fix_t
```

ユーザ定義の型は型変数を持つこと以外は，構文定義をそのままOCamlコードにしただけのものである．型変数は再帰的に定義されるところに用いられている．

一方，`fix_t` には，三つのコンストラクタがあり，それぞれGUIに必要な要素を持っている．ユーザが定義する型を持つ `Fix`，現れた時点ではどの値になるか分からないメタ変数 `Meta`，任意のコード (`code_t`) と項を保持している `Code`．`Code` は，推論が進む毎にそのコードを項に適用してある値になるかもしれない，コードを持ったメタ変数のようなコンストラクタである．

ユーザ定義の型変数に `fix_t` を適用し，相互再帰を発生させているのが，`Fix` にある `fix_t S.t` である．Mikiβの各関数は `fix_t` を用いて呼び出されるため，ユーザ定義の型は，それぞれ `fix_t` に包含された形になる．例えば，`a` → `a` の関数型を表すには，一つの型を用いると `TFun (TVar "a", TVar "a")` と表されるが，two-level typesを用いると，`Fix (TFun (Fix (TVar "a"), Fix (TVar "a")))` と表される．この包含関係により，Mikiβで定義されている関数はユーザ定義の型を持つ項を扱う前に，必ず `fix_t` 型の項を扱うことになる．`fix_t` 型の項にGUIに必要な関数を与えれば，ユーザ定義の中ではGUIの関数を定義する必要が無くなる．次節では，ユーザ定義について詳しく述べる．

4　ユーザ定義

Mikiβを利用するためには，ユーザ定義のモジュールの中に実装したい型システムの型定義と四つの展開関数を定義した各型に対するモジュールと推論規則を定義する．各型のモジュールを引数としてMikiβのファンクターに適用すると，それぞれの型に対してGUIに必要な関数とtwo-level typesの型を持ったモジュールができる．次に，ユーザモジュールをGUI生成ファンクターに適用すると，GUIが生成される．四つの展開

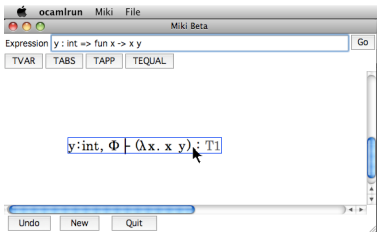

図 1: 式の入力と選択

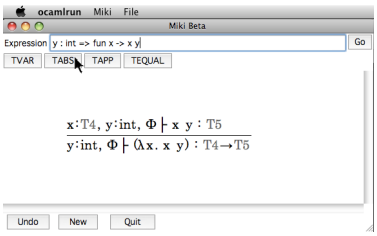

図 2: 推論規則の適用

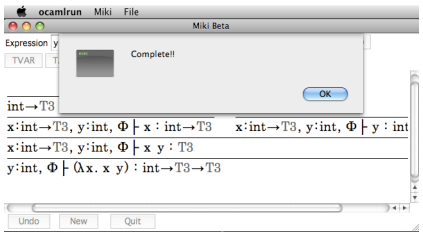

図 3: 推論終了

関数は，ユーザ定義の型に依存している関数であるため，ユーザ自身で定義を与える．これらの関数は型の構造に従って機械的に定義することができるため，容易に定義することができる．また，GUI への入力のために，字句解析 (lexer)，構文解析 (parser) を定義する必要があるが，これらは OCaml の標準ジェネレータによって生成できるため，本稿では述べない．型定義は前節で述べた通りである．この節では四つの展開関数と推論規則についての説明をする．

まず，四つの展開関数を説明する．

`map : ('a→'b)→'a t→'b t`
: コンストラクタの各要素に関数を適用し，別の型の値を得る関数．Mikiβ では，この関数に引数として再帰的な関数を与えることで，二つの型を行き来する再帰関数を定義している．

`search : bool t→bool`
: `bool` 型を持つ各要素の中に `true` があるかを調べる関数．

`unify : ('a→'a→unit)→'a t→'a t→unit`
: 単一化を行う関数．

`draw : id_t→id_t`
: オブジェクトの並べ方を定義する関数．GUI ライブラリに用意されているオブジェクトを並べる関数を用いて定義する．主にユーザが用いるのは `create_str`(文字列オブジェクトを生成する関数) と `combineH`(複数のオブジェクトを横に並べて一つのオブジェクトにする関数) である．

次に，推論規則の定義方法について説明する．次の証明木は，STLC の関数抽象の推論規則である．

$$\frac{E,\ x : T_1 \vdash e : T_2}{E \vdash \lambda x : T_1.\ e : T_1 \to T_2}\ \texttt{(TABS)}$$

この証明木の中で用いられている変数記号 (E, x, e, T_1, T_2) は，具体的な項ではなく，他の項に置き換えて用いられるメタ変数を表している．推論規則を定義する際には，定義と同様にメタ変数を利用して定義する．Mikiβ に用意されている証明木を表す `Infer` を用いて，上記の (`TABS`) は，次のように定義される．

```
let t_abs () =
  (* 各変数に対するメタ変数 *)
  let env = Env.make_meta "E" in   (* 環境 *)
  let x = Var.make_meta "X" in     (* 変数 *)
  let tm = Term.make_meta "E" in     (* 項 *)
  let tp1 = Type.make_meta "T" in    (* 型 *)
  let tp2 = Type.make_meta "T" in
  (* 推論規則: Infer(帰結式, 前提式リスト) *)
  Infer (
    judge e (term_abs x tm) (tfun tp1 tp2),
    [Infer(judge (cons x tp1 e) tm tp2, [])])
```

定義された推論規則は，推論規則適用ボタンに登録するため，ボタンに表示する文字列と推論規則を組にして `infer_rule_list` に登録する．

```
let infer_rule_list =
  [("TVAR", t_var); ("TABS", t_abs);
   ("TAPP", t_app); ("TEQUAL", t_equal)]
```

このようにして定義された推論規則の帰結式とマウスポインタで選択された式を単一化することで，選択された式の前提式リストが作られ，推論が一段進む．単純な推論規則ならば，メタ変数を用いてそのままコードに出来るが，複雑な操作がある推論規則を記述する際には，メタ変数だけでは表すことができない．例えば，STLC の変数の推論規則は次のようである．

$$\frac{x : T \in E}{E \vdash x : T}\ \texttt{(TVAR)}$$

これを表すためには x が T に入っているかどうかを判定するような関数が必要になるため，任意の式が書ける `Code` を利用する．`Code` を利用すること以外は (`TABS`) の定義と同様のため，本稿では (`TVAR`) のコード (`t_var`) を省略する．

5 使い方

Mikiβ によって生成された GUI を用いて式の推論を行うには，式を入力して推論したい式をクリック (図 1) した後，推論規則適用ボタンを押す (図 2)．式の選択と推論規則適用ボタンのクリックを繰り返すことで，推論が完了する (図 3)．Undo 機能や，右クリックによるメタ変数の具体化も実装されている．

6 まとめと今後の課題

two-level types を導入することにより，型と関数を GUI に共通した部分とユーザ定義に依存する部分に分けることができた．Mikiβ を用いて，STLC，SK コンビネータ，System F，let 多相の型システムについての GUI が実装されている．コードを公開したことにより，より多くの人に Mikiβ を利用してもらい，多くの型システムを実装してもらいたい．ここから出た要望や発見されたバグへの対応が今後の課題である．

参考文献

[1] Benjamin C. Pierce, *Types and Programming Languages*, Cambridge: MIT Press (2002).

[2] Tim Sheard and Emir Pasalic “Two-Level Types and Parameterized Modules,” Journal of Functional Programming, 14(5):547-587, (September 2004).