汎用的に証明木のGUIを作成する『Miki β』の開発

櫻井 加奈子, 浅井 健一

お茶の水女子大学

{sakurai.kanako, asai}@is.ocha.ac.jp

概 要　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．しかし，手で証明木を描く際には推論規則の本質とは関係のないところで苦労をしなければならない．本研究では，証明木を作成するための GUI の構築を助けるためのシステム『Miki β』を OCaml と Labltk を用いて開発している．Miki β では，推論規則の定義に二つのもの，すなわち画面上のオブジェクトの識別子とメタ変数を加えることで，ユーザが GUI について，ほとんど考えることなく GUI を構築することができる．ここで，メタ変数とは，別の項に変化しうる項のことであり，これを用いることで推論規則を一般的な定義と同じ形で書くことが出来るようになった．現時点では，Miki β はまだ開発途中にあるため，GUI を生成するためにユーザが定義しなければならないことが多いが，将来的には推論規則からの GUI の自動生成も考えている．現在，単純型付きラムダ計算, shift/reset を導入したラムダ計算, System F が Miki β によって実装されており，さらに簡約規則や自然言語の文法規則にも使われ始めている．

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て，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ため複雑な証明木になるほど証明木を手で描くことは難しくなる．

- 描く証明木に必要なスペースが予め分からない
  例：一段の推論規則を適用するだけで横幅に約２倍のスペースが必要となる．

$$a : int \vdash (\lambda x : T.x)a : T \quad \Rightarrow \quad \frac{a : int \vdash \lambda x : T.x : T \to T \quad a : int \vdash a : T}{a : int \vdash (\lambda x : T.x)a : T}$$

- 規則が適用される度に下段の多くを書き写さなければならない
  上記の例においても，環境 $(a : int)$ や項 $(\lambda x : T.x)$ などをそのまま書き写している．
- 単一化 (unification) によって，多くの項を書き換える必要がある
  例：$a : int$ と $a : T$ が単一化されると，全ての T を int に書き換えなければならない
  (下図では，T を書き換えた後の int を太字 (**int**) で表している)

$$\frac{\dfrac{\overline{x : T, a : int \vdash x : T}}{a : int \vdash \lambda x : T.x : T \to T} \quad a : int \vdash a : T}{a : int \vdash (\lambda x : T.x)a : T} \quad \Rightarrow \quad \frac{\dfrac{\overline{x : \mathbf{int}, a : int \vdash x : \mathbf{int}}}{a : int \vdash \lambda x : \mathbf{int}.x : \mathbf{int} \to \mathbf{int}} \quad \overline{a : int \vdash a : \mathbf{int}}}{a : int \vdash (\lambda x : \mathbf{int}.x)a : \mathbf{int}}$$

このような問題を解決するためには証明木を描くプログラムを作成すれば良い．必要なスペースを確保するためには画面を広くすれば良く，画面のスペースが足りなくなっても，スクロールを利用することで無限に近いスペースを利用できる．式のコピーもプログラムが行うため，きち

んとしたプログラムを書くことができれば，手で証明木を描くよりも正確な証明木を描くことができる．

しかし，このようなプログラムを作るのはそれほど簡単ではない．推論規則には関係のない GUI を表示するための機能をユーザ自らが考え，構築しなければならない．ユーザは必ずしも GUI 周辺のことを理解しているとは限らず，むしろ証明木を描くような論理関係の研究に携わっている人は GUI プログラムをあまり書かないであろうと思われる．さらに，一度そのようなプログラムを作成したとしても，それはひとつの型システムに対してしか使うことができない．型システムを変更したいと思っても，型システムがプログラムの中に組み込まれてしまっていれば，プログラムの内部を詳細に理解した上で変更をする必要がある．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．

本論文で紹介する『Miki β』は，ユーザが推論規則の定義を作成することで GUI の構築を行うシステムを目指して開発を行っている．Miki β は OCaml と Labltk を用いて開発しているため，利用するためには OCaml の基本的な知識を求めているが，それ以外の GUI に関する知識を要求することはない．種々の型システムに対する証明木の GUI を生成出来るようにするためには，プログラム中で GUI の部分と推論規則の定義部分がそれぞれ独立している必要がある．GUI の部分は推論規則に依存する部分と依存していない部分に分けることができる．推論規則に依存してしまう部分は，ユーザが簡単に定義できるように GUI Library に関数を用意し，推論規則に依存しない部分は，フレームワークとしてメイン関数に用意した．

推論規則の定義部分は，推論規則によって用いられるデータ定義や関数が異なるため，ユーザ自身が定義するものとしたが，データ定義以外の必要な関数は機械的に生成できる部分が多いことから，将来的にはデータ定義から必要な定義を自動生成できるようにしていきたいと考えている．現時点では Miki β はまだ開発途中にあるため，ユーザが定義しなければならないところが多くなってしまっている．しかし，これまでに Miki β を利用して単純型付きラムダ計算, shift/reset を導入したラムダ計算, System F の GUI を実装しているだけではなく，簡約規則 [3] や自然言語の文法規則 [4] にも利用され始めており，プログラミング言語の型付け規則以外にも，様々な分野で Miki β を用いることができると思われる．

本論文では，2 節で紹介する単純型付きラムダ計算を例にとり，3 節で Miki β を用いた GUI の作成方法と将来の自動化に向けた考察を述べ，4 節では GUI を操作するためのマウスポインタの動作定義についても述べる．GUI の利用例は 5 節で図を用いながら示す．その際，shift/reset を導入した単純型付きラムダ計算と System F の GUI も示すことにする．まとめとして，6 節で現状を述べ，7 節で関連研究との比較を行い，8 節で今後の課題を議論する．

2　単純型付きラムダ計算

本論文では，単純型付きラムダ計算を例にとり，Miki β を利用した GUI の作成方法を述べる．単純型付きラムダ計算の構文 [5] とその型付け規則は図 1 のようになる．なお，いずれも標準的な定義だが，型付け規則には（TWEAK）を含めている．これは，図 1 ではコンテキストが（集合ではなく）binding の順序付きリストで表現されているためである．従って（TVAR）では x の binding はコンテキストの先頭に現れていなくてはならない．x がコンテキストの後ろの方にある場合は（TWEAK）を使って x を先頭に持ってくる必要がある．

ここで，図 1 で示した型付け規則の中に使われている変数は具体的なものを表しているわけではないことに注意する．例えば，（TAPP）の規則には，Γ, t_1, t_2, T_{11}, T_{12} という 5 つの変数が出

Syntax

t ::=		terms:
	x	variables
	λx:T.t	abstraction
	t t	application
T ::=		types
	B	base type
	T → T	type of functions
Γ ::=		contexts:
	∅	empty context
	x:T, Γ	term variable binding
j ::=	Γ ⊢ t : T	judgement

Typing

$$\frac{}{x : T, \Gamma \vdash x : T} \quad \texttt{(TVAR)}$$

$$\frac{x : T_1, \Gamma \vdash t : T_2}{\Gamma \vdash \lambda x : T_1 . t : T_1 \to T_2} \quad \texttt{(TABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} \quad \texttt{(TAPP)}$$

$$\frac{\Gamma \vdash t : T}{x : T_1, \Gamma \vdash t : T} \quad \texttt{(TWEAK)}$$

図 1. 単純型付きラムダ計算の構文定義と型付け規則

てくる．しかし，実際の証明木を書く際には，これらの変数がそのまま使われることはなく，具体的なものに置き換わった上で使用される．具体的な例で示すと，

$$\frac{f : int \to int \vdash f : int \to int \quad f : int \to int \vdash 3 : int}{f : int \to int \vdash f\ 3 : int}\ (\texttt{TAPP})$$

では (TAPP) の規則の Γ を $f : int \to int$ に，t_1 を f に，t_2 を 3 に，T_{11} と T_{12} を int にして利用している．推論規則を定義する際に用いられている変数は，他の項に変化することのできるメタ変数として使われていて，これらを具体的な項に置き換えて推論規則の適用をすることで一段の証明木ができることがわかる．また，同じメタ変数は同じ項を表していて，メタ変数の一つが他の項に置き換わると同じメタ変数も全て置き換わる必要がある．

3 Miki β を利用するためのユーザ定義

Miki β を利用して GUI を作成するためにユーザが定義しなければならない部分は，推論規則に依存してしまう『証明木のデータ定義・推論規則の定義・単一化関数・出力関数』の４つである．これらの定義を行う際にはメタ変数と特別な識別子 id_t を用いる．id_t は，画面上のオブジェクトと内部データの対応付けをするものであり，これを導入することで，ユーザは GUI についてあまり意識することなく GUI を作成できる．定義方法の概要は次のようになる．

1. 証明木のデータ定義に id_t とメタ変数を加える．(3.1 節)
2. メタ変数を用いて推論規則を作る．(3.2 節)
3. 推論規則を実際の項に置き換える際に必要となる単一化関数を作成する．(3.3 節)
4. 画面上へ文字を配置する出力関数を定義する．(3.4 節)

この中で，id_t について考えなくてはならないのは出力関数のみである．出力関数を定義する際には，画面上のオブジェクトと内部データを対応付ける必要があるため，ユーザが定義する時に id_t を意識する必要がある．しかし，それ以外の定義では id_t を意識する必要はなく，id_t に何か入れる必要がある場合には，id_t のダミーの値として予め定義されている NEW を入れておけば良い．以下，Miki β におけるこれら４つの定義方法を順に説明する．

3.1 データ定義

Miki β では，前の節に示した構文の定義をほぼそのまま使うことで，分かりやすいプログラムを実現する．これは，初学者が論理を学ぶ際に Miki β を利用することを想定し，定義の意味さえ理解していれば GUI を作成できるようにするためである．

Miki βでは証明木の構造を次のように与えている．

```
type tree_t =
   Infer of judge_t * tree_t list * id_t  (* 結論，前提のリスト，id_t *)
 | Axiom of judge_t * id_t
```

Infer が証明木の内部ノード，Axiom が葉を示す．Miki βでは前提を持たない Infer を推論途中の木として扱っているため，前提を持たない Infer を Axiom として扱うことは出来ない．これは，Miki βでは推論を完了したことをユーザに伝える際に，推論が終了しているのかを判定する為に用いられる．

ユーザが定義する必要があるのは，証明木の各段で使われる式 (judge_t) の定義である．その際，次の二つのことをしなければならない．

id_t の付加 画面上のオブジェクトとの対応を取るための識別子として各項の定義に id_t を付ける．

メタ変数の導入 前節で述べたように，推論規則で用いられている変数はメタ変数として用いられている．そのため，データ定義でもメタ変数を扱えるようにする．具体的には，(judge_t を除く[1]) 各データ定義に（string * 自身への参照 * id_t）という構造を持ったメタ変数を加える．

具体的に，Miki β を利用するための単純型付きラムダ計算の構文定義を図 2 に示す．イタリック部分は，Miki β を利用するために付け加えた部分である．id_t は全てのデータ定義に付加することにしているが，id_t は GUI オブジェクトと内部データの対応を取るためのものであるから，GUI 上で選択されることがないものに関しては，id_t を付加する必要がない．しかし，推論規則によって選択項目（式を選択するのか，項を選択するのか，など）が変化してくる場合に備え，すべての定義に予め id_t を付加しておくことにしている．

図 2 をみると，普通の定義とは違って変数の定義 (var_t) が独立した型になっている．これは，（TVAR）や（TABS）に出てくる x のように，変数を表すメタ変数を使用したいからである．

この先，上記のように定義されたデータに関して二つの関数が必要となる．現状では，これらの関数はユーザが書かなくてはならない．ひとつは各データの id_t を取り出す関数である．例えば term_t 型のデータの id_t を取り出す関数は以下のようになる．

```
let get_term_id term = match term with
  Var(_, id) | Abs(_, _, id) | App(_, _, id) | MetaTerm(_, _, id) -> id
```

単純型付きラムダ計算の場合は，これ以外に get_var_id，get_type_id，get_env_id，get_judge_id も同様に作成する．

もうひとつは，新しいメタ変数を生成する関数である．例えば type_t 型のメタ変数を生成するには以下のような関数を作る．

---

[1] judge_t は，式の形を定義しているものであり，それ自体がメタ変数になって証明木の推論に関わるということがないので，メタ変数を付加する必要がない．

```
type var_t =
    V of string * id_t
  | MetaVar of
      string * var_t option ref * id_t

type term_t =
    Var of var_t * id_t
  | Abs of var_t * term_t * id_t
  | App of term_t * term_t * id_t
  | MetaTerm of
      string * term_t option ref * id_t

type type_t =
    TVar of string * id_t
  | Fun of type_t * type_t * id_t
  | MetaType of
      string * type_t option ref * id_t
```

```
type env_t =
    Empty * id_t
  | Cons of
      var_t * type_t * env_t * id_t
  | MetaEnv of
      string * env_t option ref * id_t

type judge_t = {
    env :  env_t;
    term :  term_t;
    typ :  type_t;
    id : id_t }
```

図 2. Miki β を利用するための単純型付きラムダ計算の構文定義

```
let new_type () =
  let s = gensym () in
  MetaType (s, ref None, NEW)
```

ここで，gensym は新しい文字列（変数名）を作る関数，NEW は id_t 型のダミーの値である．これで，新しい名前の（type_t 型の）メタ変数を作成している．単純型付きラムダ計算の場合は，これ以外に new_var，new_term，new_env も同様に作成する．

メタ変数を付加したり，先に述べた 2 つの関数 (id_t を取り出す関数 (get_term_id), 新しいメタ変数を生成する関数 (new_type)) を作ることは，単純なデータ定義 (図 2 からイタリックの部分を除いたもの) から機械的に行うことができるため，Miki β 用のデータ定義を単純なデータ定義から自動生成することはさほど難しくはないと思われる．

また，Miki β の GUI では，エントリに式を入力することによって，証明木の結論部を読み込んでいる．そのために，現在は構文解析 (parser) と字句解析 (lexer) をユーザが定義しなければならない．

3.2 推論規則の定義

データ定義にメタ変数が入ると，推論規則の定義をそのままの形で定義することができるようになる．これは，前節で示したように，推論規則が抽象的に定義されているためである．推論規則を定義する際には，推論規則の中に現れるメタ変数をまず定義し，それらを使って公理，または推論規則を作成する．例えば，(TVAR) は以下のようになる．

```
let tvar() =
 (* 新しいメタ変数の生成 *)
  let e = new_env() in                (* 環境 *)
```

```
  let x = new_var() in                (* 項 *)
  let tp = new_type() in              (* 型 *)
 (* メタ変数を利用して推論規則を作成する *)
  Axiom({ env = Cons(x, tp, e, NEW);
          term = Var(x, NEW); typ = tp; id = NEW}, NEW)}
```

(TVAR) には，メタ変数として Γ と x と T が現れているので，それに対応して e，x，tp というメタ変数をまず生成する．それらを使って，公理 $x : T, \Gamma \vdash x : T$ をそのまま Axiom として定義する．このように Miki β における推論規則の記述は型付け規則の定義そのものになっていることがわかる．同様にして，(TABS) は以下のようになる．

```
let tabs() =
    (* 新しいメタ変数の生成 *)
  let e = new_env() in      (* 環境 *)
  let t = new_term() in     (* 項 *)
  let x = new_var() in      (* 変数 *)
  let tp1 = new_type() in   (* 型 *)
  let tp2 = new_type() in
    (* メタ変数を利用して推論規則を作成する *)
  Infer({ env = e; term = Abs(x, tp1, t, NEW);
          typ = Fun(tp1, tp2, NEW); id = NEW},
        [Infer({ env = Cons(x, tp1, e, NEW); term = t;
                 typ = tp2; id = NEW}, [], NEW)], NEW)
```

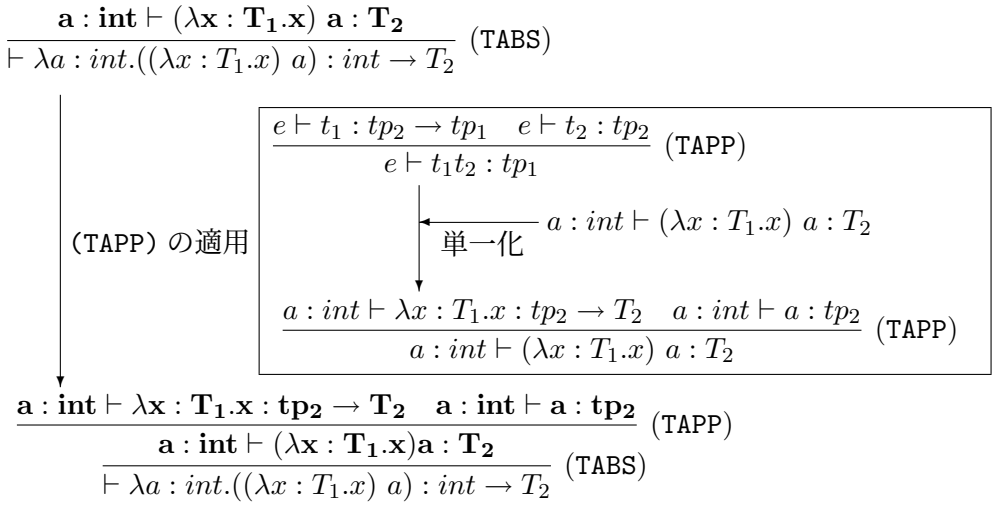
このようにしてメタ項を用いて定義された推論規則は，次節で述べる推論規則の適用に用いられる．

3.3 推論規則の適用と単一化関数

単一化関数とは，メタ変数のポインタが指す項を変更する関数である．3.1 節で示したように，メタ変数は参照する項のポインタ (_ option ref) を保持しており，同じメタ変数は同じポインタを保持している．そのため，あるメタ変数が保持しているポインタの指す内容が変更されると，同じポインタを保持しているメタ変数には同じ変更がされることになる．

推論を一段進めるためには，単一化関数とメタ変数で書かれた推論規則を用いる．推論規則の結論部分と推論規則を適用しようとしている式を単一化すると，メタ変数で書かれた推論規則は，推論規則を適用する式を一段推論した木となり，この木を元の式と入れ替えると，推論を一段進めることができる．

例えば，$a : int \vdash (\lambda x : T_1.x)\ a : T_2$　(下図上端の太字部分) に (TAPP) の規則を適用すると，次のようになる．



この時, 環境・項・型はそれぞれ表 1 のような単一化が起きている.

	推論規則のメタ変数	推論規則を適用する式の項
環境	e	$a : int$
項 1	t_1	$\lambda x : T_1.x$
項 2	t_2	a
型	tp_1	T_2

表 1. 単一化の対応

この単一化を行う関数は，ユーザが定義する必要がある．要素と再帰構造のみの単純な構造を持つ項の単一化であれば，次の規則を用いて単一化関数を定義することができる．しかし，この単一化の規則を適用することが出来ないものがある．その例は，5.2 節で示すことにする．ここでは，メタ変数以外の項を固定項と呼んでいる．

- メタ変数とメタ変数の単一化
  - 同じ変数名であれば既に同じメタ変数なので，単一化が起こらない.
  - 異なる変数名であれば一方が他方を参照するように単一化する.
- メタ変数と固定項の単一化
  occur check を行った上で，メタ変数が参照先を持っていなければ固定項を参照する．また，参照先がある場合は参照先の項と固定項を単一化する.
- 固定項と固定項の単一化
  - 同じ項であれば既に同じ値なので単一化は起こらない.
  - 同じ構造の項であれば再帰的に単一化をし，異なる構造の項であればエラーを返す.

この規則に沿って，型の単一化関数 `unify_type` は以下のように定義できる.

```
(* unify_type : type_t -> type_t -> bool *)
let rec unify_type t1 t2 = match (t1, t2) with
    (* メタ変数とメタ変数の単一化 *)
    (* 同じ変数名の場合 *)
```

```
    (MetaType(str1,_,_),MetaType(str2,_,_)) when str1 = str2 -> ()
    (* 異なる変数名の場合 *)
  | (MetaType(_,op1,_),MetaType(_,op2,_)) ->
      (match (!op1, !op2) with
           (None,None) ->  op1 := Some(t2)
         | (Some(tp),None) -> unify_type tp t2
         | (None,Some(tp)) -> unify_type t1 tp
         | (Some(tp1),Some(tp2)) -> unify_type tp1 tp2)
    (* メタ変数と固定項の単一化 *)
  | (MetaType(_,op,_), Var(_,_)) ->
      (match !op with
           None -> op := Some(t2)
         | Some(tm) -> unify_type tm t2)
  | (MetaType(str1 ,op1,_),Fun(tp1,tp2,_)) ->
      if (occur str1 tp1) || (occur str1 tp2) then raise Unify_Error
      else (match !op1 with
                None  ->  opt1 := Some(t2)
              | Some(t) -> unify_type t t2)
....
    (* 固定項と固定項の単一化 *)
 | (Var(str1,_),Var(str2,_)) ->
     if str1 = str2
     then ()
     else raise Unify_Error
  | (Fun(tp1,tp2,_),Fun(tp3,tp4,_)) -> unify_type tp1 tp3; unify_type tp2 tp4
  | (Var(_,_), Fun(_,_,_)) | (Fun(_,_,_), Var(_,_)) -> raise Unify_Error
```

同様にして，unify_var，unify_term，unify_env も作成することができる．これらの関数を使うと，unify_judge は以下のように定義される．

```
(* unify_judge : judgement の各要素を単一化する *)
let unify_judge rule_judge judge = match (rule_judge, judge) with
    ({env = r_env; term=r_tm; typ=r_tp}, {env = env; term = tm; typ = tp}) ->
      unify_env r_env env;  (* 環境の単一化 *)
      unify_term r_tm tm;   (* 項の単一化 *)
      unify_type r_tp tp    (* 型の単一化 *)
```

単一化が途中で失敗した場合，これまでに単一化をした項を元の項に戻す必要がある．Miki β では，推論している証明木を保持すると同時に，各段において適用された推論規則も保持している．これを用いて，単一化が失敗した場合や Undo をする際には初めから推論規則を適用し直すことで，元の状態に戻している．

推論規則の適用を GUI 上で実行するためには，マウスポインタで式を選択し，推論規則適用ボタン (図 3) を押す．ユーザは，GUI Library で提供されている関数 register_infer_button_list を用いて予めボタンの内容を登録しておく必要がある．この関数は，ボタン上に表示したい文字列と推論規則を生成する関数の組をリストとして受け取り，ボタンの内容を登録する関数である．

```
register_infer_button_list
     [("TVAR", tvar); ("TABS", tabs); ("TAPP", tapp); ("TWEAK", tweak)]
```

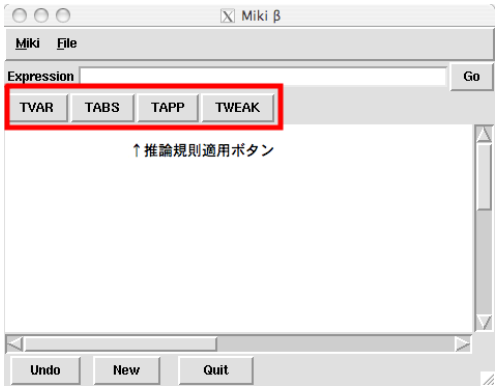


図 3. 推論規則適用ボタン

3.4 出力関数

式のデータ定義 (judge_t) が与えられても，それをどのように画面に表示するかはユーザの式定義に依存するため，出力関数はユーザが定義する必要がある．Miki β には出力関数を簡単に書くための関数が GUI Library に用意されており，ユーザはそれらの関数を利用することによってオブジェクトを適切な位置に配置することができる．GUI Library の代表的な関数を次に示す．

- `create : string → id_t`
  文字列から画面上にオブジェクトを作成し，画面上に配置する関数
- `combine : id_t list → id_t`
  複数のオブジェクトを画面上に横に並べて一つのオブジェクトにする関数
- `put : id_t→ id_t→ id_t`
  二つのオブジェクトを縦に並べ，さらにその間に適当な長さの線を引いて一つのオブジェクトにする関数 (項を横に並べるだけで式ができる場合はあまり使われることはない)

これらの関数は，できたオブジェクトの id_t を返すので，出力関数は新しい id_t を持つデータを返す形で記述すると，内部データと画面上のオブジェクトを対応させることができる．この対応は，マウスをクリックした時，選択されたオブジェクトに対応する内部データを探索する際に用いられる．id_t について考える必要があるのはこの部分のみであり，これ以外の部分には id_t のダミーの値として NEW を用いればよい．これは，推論が一段進むごとに推論木は初めから描かれ，新しい証明木が描かれる度に id_t が書き換えられるためである．

例えば，va_t の出力関数 draw_var は以下のようになる．

```
let rec draw_var v = match v with
    V(str, _) -> V(str, create str)
  | MetaVar(str, op, _) -> match !op with
        None -> MetaVar(str, op, create str)
      | Some(v2) -> draw_var v2
```

同様にして draw_term，draw_type，draw_env，draw_judge も作成する．各描画関数を用いて，式を『環境 ⊢ 項 : 型』という形で画面上に出力するためには，次のように書く．

```
let draw_judge judge = match judge with
    {env=env; term=term; typ=tp} ->
      let env' = draw_env env in
      let term' = draw_term term in
      let tp' = draw_type tp in
      let t1 = get_id_env env' in
      let t2 = get_id_term term' in
      let t3 = get_id_tp tp' in
      let new_id = combine [t1; create " ⊢"; t2; create ":"; t3] in
        {env=env'; term=term'; typ=tp'; id=new_id}}
```

項の括弧付けについては，引数として括弧を付けるか否かのフラグを与え，そのフラグの値によって括弧付けを行っている．括弧付けを簡単にするため，GUI Library に need_paren という関数が用意されている．この関数は，引数に真偽値と id_t を受け取り，真偽値が true であれば受け取った id_t に括弧を付け，新たな id_t を返す関数である．

```
need_paren :  bool → id_t→ id_t
```

これを利用すると，型の描画関数は次のように定義することができる．

```
let draw_type typ =
  let rec loop paren_flag typ = match typ with
    ....
     | Fun(tp1, tp2, _) ->
       let tp1' = loop true tp1 in
       let t1 = get_type_id tp1' in
       let tp2' = loop true tp2 in
       let t2 = get_type_id tp2' in
       Fun(tp1', tp2', need_paren paren_flag (combine[t1; create "→"; t2]))
    ....
  in loop false typ
```

4　マウスの動作

本論文で紹介している単純型付きラムダ計算の GUI では，マウスクリックを用いて式を選択している．(図 4)

描画関数で，画面上のオブジェクトの id_t が内部データに保存されているので，マウスポインタの座標と id_t のアイテム領域とを比較し，アイテム領域内にマウスポインタの座標が存在すれば式の周りに枠を表示している．また，枠を表示すると同時に，選択されたオブジェクトの id_t を保存しておき，推論規則適用の際の内部データ探索に利用している．id_t を利用すれば，式選択だけではなく，式より細かい単位や大きい単位で選択することも可能である．

現在はマウスの動作としてマウスクリックのみをサポートしているが，同じ原理を用いてマウスクリック以外の動作を簡単に登録する工夫をしていきたいと考えている．

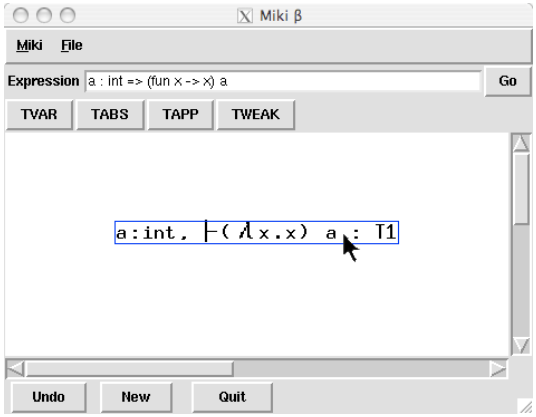


図 4. クリックによる式選択

5 利用例

作成した GUI を用いた推論は，次の手順で行われる．

1. エントリに推論したい式を書き，Go ボタンを押す (図 5)
2. 推論規則を適用する式を選択し (図 6)，推論規則適用ボタンを押す (図 7)．この時，選択された式に対して推論規則が適用出来ない場合は，何も起こらない．
3. 2 を繰り返すことで証明木ができていく (図 8)．

5.1 shift / reset の導入

Miki β では，推論規則の定義部分が GUI の部分から独立した形で設計されているため，一つの推論規則の GUI を構築すれば，それに少し変更を加え，新しい推論規則を定義することは容易である．ここに，単純型付きラムダ計算の型付け規則に，限定継続の命令である shift / reset を加えた型付け規則に対する GUI を示す．限定継続を扱う型付け規則は answer type を明示的に指定するため，多くの型変数を必要とし [1]，手で証明木を描くことはとても大変である．Miki β を利用して GUI を作成すると簡単に証明木を描くことができるようになる．実際にこの GUI を作る際には，単純型付きラムダ計算の GUI を拡張し，構文の定義と型付け規則の定義に少し変更を加えた．この作業に要した時間は 1 時間程度であった．例として，$\lambda f.(shift\ k.\ k)$ の証明木を Miki β で描いた証明木を示す．(図 9)

5.2 System F

同様にして，現在，System F の型付け規則も実装している．System F は，単純型付きラムダ計算の型付け規則に，次の二つの型付け規則を加えたものである．

$$\frac{\Gamma, X \vdash t : T}{\Gamma \vdash \lambda X.t : \forall X.T}\ (\texttt{T-TABS})$$

$$\frac{\Gamma \vdash t : \forall X.T_2}{\Gamma \vdash t\ [T_1] : T_2[X \mapsto T_1]}\ (\texttt{T-TAPP})$$

System F は多相型の導入により，複雑な単一化が起こる．次の式の型付けの挙動をみてもらいたい．

$$a : int \vdash (\Lambda X.\lambda x : X.x)[int]a$$

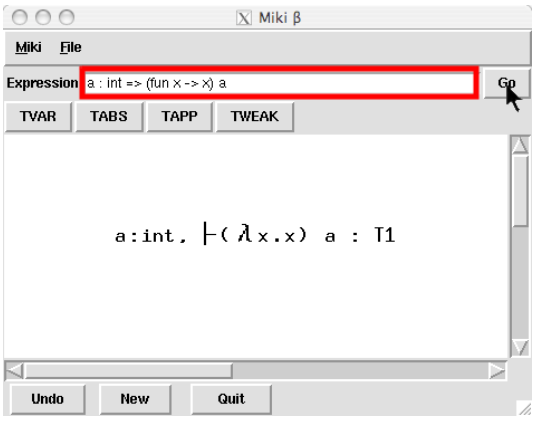


図 5. 式入力と画面出力

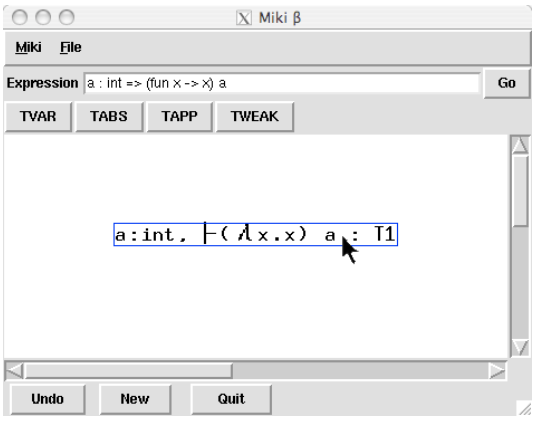


図 6. 式選択

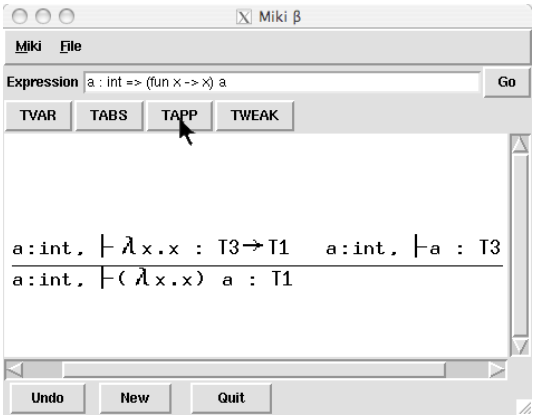


図 7. 推論規則適用

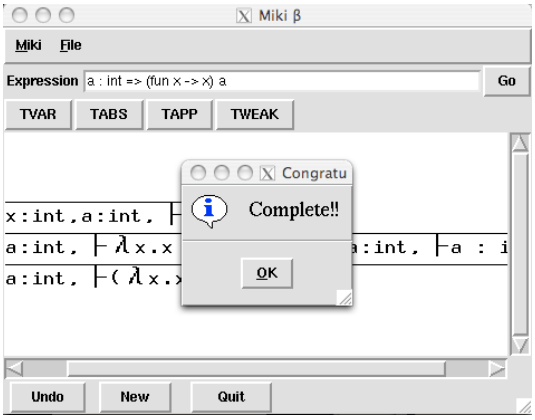


図 8. 推論終了

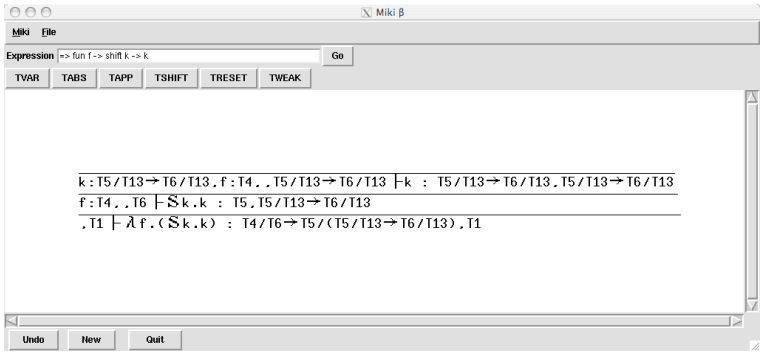


図 9. shift/reset の証明木

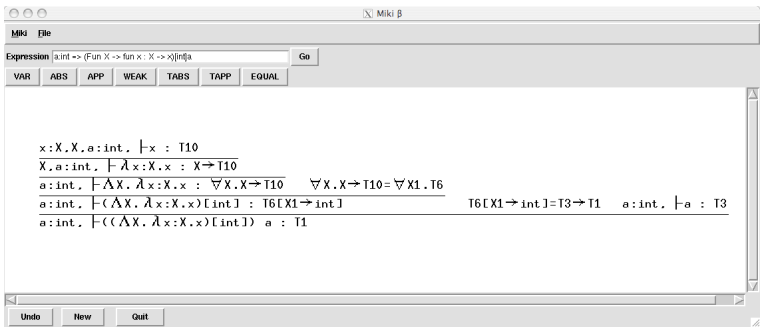


図 10. System F の証明木

これに（TAPP）を適用すると，

$$\frac{a : int \vdash (\Lambda X.\lambda x : X.x)[int] : M_1 \to M_2 \quad a : int \vdash a : M_1}{a : int \vdash (\Lambda X.\lambda x : X.x)[int]a : M_2}\ (\texttt{TAPP})$$

となる（M_1, M_2 はメタ変数）．さらに $a : int \vdash (\Lambda X.\lambda x : X.x)[int] : M_1 \to M_2$ に単純に（T-TAPP）を適用しようとするが，$T_2[X \mapsto T_1]$ と $M_1 \to M_2$ を単一化することができない．なぜならば，$T_2[X \mapsto T_1] = M_1 \to M_2$ としたいが，$T_2[X \mapsto T_1] \neq T_2$ であるため，T_2 をただのメタ項として捉えて $M_1 \to M_2$ と単一化することができないからである．つまり，この時点では単一化ができないので，この等式を持ったまま（T-TAPP）を適用したい．そのために，（TAPP）と（T-TAPP）を次のように変更した．

$$\frac{\Gamma \vdash t_1 : T \quad T = T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}}\ (\texttt{TAPP})$$

$$\frac{\Gamma \vdash t : T \quad T = \forall X.T_2}{\Gamma \vdash t\ [T_1] : T_2[X \mapsto T_1]}\ (\texttt{T-TAPP})$$

同値関係を導入することにより，多相型と $T_{11} \to T_{12}$ の単一化を避けることができる．これを利用すると，

$$\frac{\dfrac{a : int \vdash \Lambda X.\lambda x : X.x : M_4 \quad M_4 = \forall Y.M_3}{a : int \vdash (\Lambda X.\lambda x : X.x)[int] : M_3[Y \mapsto int]}\ (\texttt{T-TAPP}) \quad M_3[Y \mapsto int] = M_1 \to M_2 \quad a : int \vdash a : M_1}{a : int \vdash (\Lambda X.\lambda x : X.x)[int]a : M_2}\ (\texttt{TAPP})$$

のように単一化の問題に悩まされることなく（T-TAPP）を適用することができる．この後，推論が進み，メタ変数 M_3 の値が定まったら $M_3[Y \to int]$ の代入を行うことができるようになり，$M_1 \to M_2$ との単一化も可能となる．

以上の考察から，同値関係を持った証明木を実装するために，judge_t に Equal という式を加えて，定義をした．

```
type judgement_t = {
      env : env_t;
      term : term_t;
      typ : type_t;
      id : id_t}

type judge_t =
    Judge of judgement_t
  | Equal of type_t * type_t * id_t
```

さらに，（Equal）という推論規則を付け加えた．

$$\frac{}{T = T}\ (\texttt{Equal})$$

これにより，$a : int \vdash (\Lambda X.\lambda x : X.x)[int]a$ は図 10 のように型付けされる．このように型代入のような操作が入ってくると，それに従って型付け規則の方もある程度，変更せざるを得ない．さらに，型代入は，型代入される項がメタ変数ではなくなったときに実行する必要があるため，推論を一段進める度に，全ての型をチェックし，型代入を行わなければならない．現在，そのプロトタイプを実装しているが，型代入処理などでシステム内部を変更しなくてはなず，さらなる改良が求められる．将来的には，GUI システムに深入りすることなく，そのような複雑な処理を外部関数としてユーザが直接，定義して組み込めるような仕組みを考えていく予定である．

6　現状

現在，Miki βで生成されるGUIはGUI Library・システム定義・GUI補助関数・メイン関数から構成されていて (図11)，ユーザが定義する必要があるのはシステム定義である．ユーザはシステム定義をする際にデータ定義・推論規則・単一化関数・出力関数定義の４つのものを書かなくてはならない．データ定義する際には`id_t`を取り出す関数とメタ変数を生成する関数が必要となっているが，これは定義から機械的に定義できるものであり，今後，自動生成できるようにすることも考えている．システム定義ができれば GUI が生成されるが，マウス動作等，GUIの細かい動きを制御するためのGUI補助関数は必要に応じて変更を加えることが出来る．Miki βでは，GUIの部分とシステム定義の部分が独立して構成されているため，汎用的に GUI を作成できるようになっていると考える．

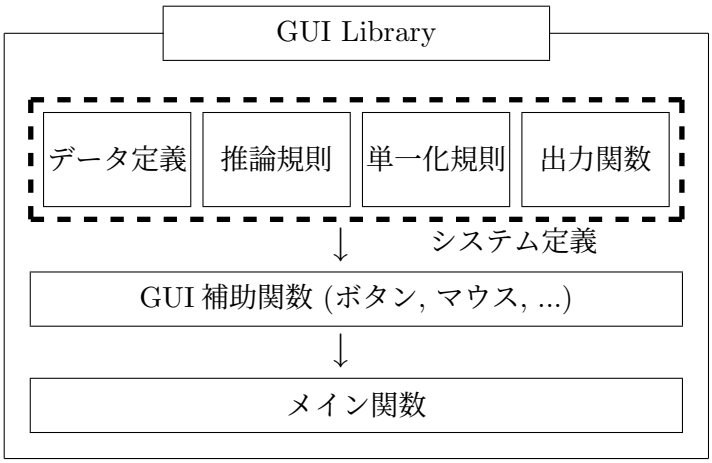


図 11. Miki βシステム概要図

7　関連研究

本研究の目的は，証明木を提示する GUI を作ることだが，そこで使われている技術は定理証明系と共通する部分が多い．実際，証明木を作ることは，定理証明系においてゴールを分解してサブゴールを作り，それらを順に証明していく過程とみなすことができる．定理証明系には，例えば Coq [6] の tactics のように推論規則を拡張する規則があるが，そのようなものを本システムに導入するのは面白い方向性である．実際，(`TWEAK`) の適用などは自動化できるのが望ましい．

メタ変数を用いて不完全な証明木を論理的に扱う研究を Geuvers と Jojgov [2] が行っている．彼らは GUI を作ることはしていないが，定理証明系で行われていることの定式化を目指して研究しており，本研究の理論的な基礎を与えるものと思われる．

8　今後の課題

Miki βは開発されたばかりのため，改良すべき点が多くある．現在検討している課題について述べる．

- 推論規則の適用：現在，メタ変数で書かれた推論規則と式を単一化することで推論規則を適用しているが，システムが複雑になると単一化だけでは推論規則が適用できない場合がある．推論規則に条件文などの式を埋め込められる形に改良できれば，様々な推論規則を導入することができると考えている．
- 証明木の部分的な簡易表示：shift/reset を導入した単純型付きラムダ計算のように，横に大きく広がってしまう推論規則の場合，証明木の全ての部分を表示していると，ユーザが見たいと考えている式までスクロールしなければならない．ユーザが必要としていない部分を簡易表示することで証明木の全体像をつかみやすくしようと考えている．
- 置換の実装：証明木を描いている途中で，式の一部分を変更したい場合がある．Miki β では，メタ変数を用いて証明木を描いているため，メタ変数の参照先を変更することで置換をすることができると考えている．

将来的には機械的に生成出来る部分の自動生成を行えるようにしていきたいが，現段階では使い方を整理したマニュアルを作成し，多くの人に Miki β を利用してもらいたい．その上で問題抽出を行い，より汎用性の高いものに改良していこうと考えている．

参考文献

[1] Danvy, O., and A. Filinski “A Functional Abstraction of Typed Contexts,” Technical Report 89/12, DIKU, University of Copenhagen (July 1989).

[2] Herman Geuvers and Gueorgui I. Jojgov. “Open Proofs and Open Terms: A Basis for Interactive Logic,” In Proc. of CSL 2002. LNCS 2471, pp.537–552.

[3] 石川ちひろ, 浅井健一「簡約過程の一般的可視化システムの実装」第 12 回 プログラミング及びプログラミング言語ワークショップ, ポスター発表 (March 2010).

[4] 尾崎有梨, 櫻井加奈子, 浅井健一, 戸次大介「証明木作成プログラムを用いた CCG 統語導出の実装」, 言語処理学会第 16 回年次大会発表論文集（掲載予定）, 東京大学 (2010).

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

[6] Yves Bertot and Pierre Castéran, *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions*, EATCS Series, Berlin: Springer (2004).