4.3 ML^3 の型推論

4.3.1 関数に関する型付け規則

次に，fun 式，関数適用式

$$e ::= \cdots \mid \mathsf{fun}\ x \to e \mid e_1\ e_2$$

で型推論アルゴリズムを拡張しよう.「τ_1 の値を受け取って（計算が停止すれば）τ_2 の値を返す関数」の型を $\tau_1 \to \tau_2$ とすると，型の定義は以下のように変更される.

$$\tau ::= \mathtt{int} \mid \mathtt{bool} \mid \tau_1 \to \tau_2$$

これらの式に関して型付け規則は以下のように与えられる.

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathsf{fun}\ x \to e : \tau_1 \to \tau_2} \qquad \text{(T-Fun)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \qquad \text{(T-App)}$$

それぞれの規則について簡単に説明を加える.

T-Fun: 引数 x が τ_1 を持つという仮定の下で関数本体 e が τ_2 型を持つならば，$\mathsf{fun}\ x \to e$ が $\tau_1 \to \tau_2$ 型を持つことを導いて良い．これは関数のセマンティクスから理解できるであろう.

T-App: e_1 の型が関数型 $\tau_1 \to \tau_2$ であり，かつ，その引数の型 τ_1 と e_2 の型が一致している場合に，適用式全体に e_1 の返り値型 τ_2 がつくことを導いて良い．これも関数型の直観と関数適用のセマンティクスから分かるであろう.

次は型推論アルゴリズムの設計である．これらの規則を含めても型付け規則は構文主導なので，前節の「規則を下から上に読む」という戦略を使ってみよう．入力として型環境 Γ と式 e が与えられ，式 e が $\mathsf{fun}\ x \to e_1$ という形をしていたとしよう．そうすると，T-Fun を下から上に使うことに読んで，以下のように型推論ができそうである.

1. 型環境 $\Gamma, x : \tau_1$ と式 e_1 を入力として型推論アルゴリズムを再帰的に呼び出し型 τ_2 を得る.
2. 型 $\tau_1 \to \tau_2$ を e の型として返す.

ところが，これでは型推論が実装できない．問題は，最初のステップで e_1 の型を調べる際に作る型環境 $\Gamma, x : \tau_1$ である．ここで x の型として τ_1 を取っているが，この型をどのように

取るべきかは，一般には e_1 の中での x の使われ方と，この関数 fun $x \to e_1$ がどのように使われうるかに依存するので，このタイミングで τ_1 を容易に決めることはできない．

簡単な例として，fun x → x + 1 という式を考えてみよう．これは，int→int 型の関数であることは「一目で」わかるので，一見，x の型を int として推論を続ければよさそうだが，問題は，本体式である x + 1 を見るまえには，x の型が int であることがわからない．

4.3.2 型変数，型代入と型推論アルゴリズムの仕様

この「τ_1 の適切な取り方が後にならないとわからない」という問題を解決するために「今のところ正体がわからない未知の型」を表す**型変数** (*type variable*) を導入しよう．

$$\tau ::= \alpha \mid \texttt{int} \mid \texttt{bool} \mid \tau_1 \to \tau_2$$

そして，型推論アルゴリズムの出力として，入力（正確には型環境中）に現れる型変数の「正体が何か」を返すことにする．上の例だと，とりあえず x の型は α などと置いて，型推論を続ける．推論の結果，x + 1 の型は int である，という情報に加え $\alpha = \texttt{int}$ という「型推論の結果 α は int であることが判明しました」という情報が返ってくることになる．最終的に T-Fun より，全体の型は $\alpha \to \texttt{int}$，つまり，int→int であることがわかる．また，fun x → fun y → x y のような式を考えると，以下のような手順で型推論がすすむ．

1. 新しい（つまり，他の型変数とカブらない）型変数 α を生成し，x の型を α と置いて，本体，つまり fun y → x y の型推論を行う．
2. 新しい型変数 β を生成し，y の型を β と置いて，本体，つまり x y の型推論を行う．
3. x y の型推論の結果，この式の型が別の新しい型変数 γ を使って $\beta \to \gamma$ と書け，$\alpha = \beta \to \gamma$ であることが判明する．

さらに詳しい型推論アルゴリズムの中身については後述するが，ここで大事なことは，とりあえず未知の型として用意した型変数の正体が，推論の過程で徐々に明らかになっていくことである．[1]

> **型変数と多相型** OCaml の**多相型** (*polymorphic type*) とここで導入する型変数を含む型とを混同してはならない．OCaml における多相型は，例えば OCaml では `fun x -> x` が型 `'a -> 'a` を持ち，ここで表示される `'a` を「型変数」ということがあるが，この `'a` は上記の型変数とは異なる．この `'a` は「何の型にでも置き換えてよい」変数であるが，上記の型変数は「特定の型を表す」記号である．「任意の型で置き換え可能」な型変数は次節で再登場する．

[1] 余談ではあるが，ここで用いられている方法は，未知の情報を含む問題を解くために (1) 未知の情報をとりあえず変数において (2) その変数が満たすべき制約を生成し (3) その制約を解くという，より一般的な問題解決の手法の適用例と見ることができる．方程式を立ててそれを解くとか，散々やってきたよね？

ここまで述べたことを実装したのが図 4.1 である．型 ty を型変数を表すコンストラクタ TyVar と関数型を表すコンストラクタ TyFun とで拡張する．TyVar は tyvar 型の値を一つとるコンストラクタで TyVar(tv) という形をしており，これが型変数を表す．tyvar 型は型変数の名前を表す型で，実体は整数型である．TyFun は ty 型の引数を 2 つ持つ TyFun(t1,t2) という形をしており，これが型 $\tau_1 \to \tau_2$ を表す．

型推論アルゴリズムの実行中には，他のどの型変数ともかぶらない新しい型変数を生成する必要がある．(このような型変数を *fresh* な**型変数**と呼ぶ．) これを行うのが関数 fresh_tyvar である．この関数は引数として () を渡すと（すなわち fresh_tyvar () のように呼び出すと）新しい未使用の型変数を生成する．この関数は次に生成すべき型変数の名前を表す整数への参照 counter を保持しており，保持している値を新しい型変数として返し，同時にその参照をインクリメントする．上で説明したように，T-FUN のケースでは新しい型変数を生成するのだが，その際にこの関数を使用する．[2]

上述の型変数とその正体の対応関係を，**型代入** (*type substitution*) と呼ぶ．型代入（メタ変数として $\mathcal{S}$ を使用する．）は，型変数から型への（定義域が有限集合な）写像である．以下では，$\mathcal{S}\tau$ で τ 中の型変数を $\mathcal{S}$ を使って置き換えたような型，$\mathcal{S}\Gamma$ で，型環境中の全ての型に $\mathcal{S}$ を適用したような型環境を表す．例えば $\mathcal{S}$ が $\{\alpha \mapsto \texttt{int}, \beta \mapsto \texttt{bool}\}$ であるとき，$\mathcal{S}\alpha = \texttt{int}$ であり，$\mathcal{S}(\alpha \to \beta) = \texttt{int} \to \texttt{bool}$ であり，$\mathcal{S}(x:\alpha, y:\beta) = (x:\texttt{int}, y:\texttt{bool})$ である．$\mathcal{S}\tau$，$\mathcal{S}\Gamma$ はより厳密には以下のように定義される．

$$
\begin{aligned}
\mathcal{S}\alpha &= \begin{cases} \mathcal{S}(\alpha) & \text{if } \alpha \in dom(\mathcal{S}) \\ \alpha & \text{otherwise} \end{cases} \\
\mathcal{S}\texttt{int} &= \texttt{int} \\
\mathcal{S}\texttt{bool} &= \texttt{bool} \\
\mathcal{S}(\tau_1 \to \tau_2) &= \mathcal{S}\tau_1 \to \mathcal{S}\tau_2 \\
\\
dom(\mathcal{S}\Gamma) &= dom(\Gamma) \\
(\mathcal{S}\Gamma)(x) &= \mathcal{S}(\Gamma(x))
\end{aligned}
$$

$\mathcal{S}\alpha$ のケースが実質的な代入を行っているケースである．$\mathcal{S}$ の定義域 $dom(\mathcal{S})$ に α が入っている場合は，$\mathcal{S}$ によって定められた型（すなわち $\mathcal{S}\alpha$）に写像する．int と bool は型変数を含まないので，$\mathcal{S}$ を適用しても型に変化はない．τ が $\tau_1 \to \tau_2$ であった場合は再帰的に $\mathcal{S}$ を適用する．

型代入を使うと，新しい型推論アルゴリズムの仕様は以下のように与えられる．

入力: 型環境 Γ と式 e

出力: $\mathcal{S}\Gamma \vdash e : \tau$ を結論とする判断が存在するような型 τ と代入 $\mathcal{S}$

[2]関数 fresh_tyvar は呼び出すたびに異なる値を返すことに注意せよ．これは fresh_tryvar が純粋な意味での計算ではない（参照の値の更新や参照からの値の呼び出しといった）**副作用** (*side effect*) を持つためである．

型推論アルゴリズムを実装する前に，以降で使う補助関数を定義しておこう．

Exercise 4.3.1 図 4.1 中の pp_ty，freevar_ty を完成させよ．freevar_ty は，与えられた型中の型変数の集合を返す関数で，型は

```
val freevar_ty : ty -> tyvar MySet.t
```

とする．型 'a MySet.t は mySet.mli で定義されている 'a を要素とする集合を表す型である．

さて，型推論アルゴリズムを実装するためには，型代入を表すデータ構造を決める必要がある．様々な表現方法がありうるが，ここでは素直に型変数と型ののペアのリストで表現することにしよう．すなわち，型代入を表す OCaml の型は以下のように宣言された subst である．

```
type subst = (tyvar * ty) list
```

subst 型は [(id1,ty1); ...; (idn,tyn)] の形をしたリストである．このリストは [idn $\mapsto$ tyn] $\circ \cdots \circ$ [id1 $\mapsto$ ty1] という型代入を表すものと約束する．つまり，この型代入は「受け取った型中の型変数 id1 をすべて型 ty1 に置き換え，得られた型中の型変数 id2 をすべて型 ty2 に置き換え... 得られた型中の型変数 idn をすべて型 tyn に置き換える」ような代入である．リスト中の型変数と型のペアの順序と，代入としての作用の順序が逆になっていることに注意してほしい．また，リスト中の型は後続のリストが表す型代入の影響を受けることに注意してほしい．例えば，型代入 [(alpha, TyInt)] が型 TyFun(TyVar alpha, TyBool) に作用すると，TyFun(TyVar TyInt, TyBool) となり，型代入

```
[(beta, (TyFun (TyVar alpha, TyInt))); (alpha, TyBool)]
```

が型 (TyVar beta) に作用すると，まずリストの先頭の (beta, (TyFun (TyVar alpha, TyInt))) が作用して TyFun (TyVar alpha, TyInt) が得られ，次にこの型にリストの二番目の要素の (alpha, TyBool) が作用して TyFun(TyBool, TyInt) が得られる．

以下の演習問題で，型代入を作用させる補助関数を実装しよう．

Exercise 4.3.2 型代入に関する以下の型，関数を typing.ml 中に実装せよ．

```
type subst = (tyvar * ty) list

val subst_type : subst -> ty -> ty
```

例えば，

```
let alpha = fresh_tyvar () in
subst_type [(alpha, TyInt)] (TyFun (TyVar alpha, TyBool))
```

の値は TyFun (TyInt, TyBool) になり，

```
let alpha = fresh_tyvar () in
let beta = fresh_tyvar () in
subst_type [(beta, (TyFun (TyVar alpha, TyInt))); (alpha, TyBool)] (TyVar beta)
```

の値は TyFun (TyBool, TyInt) になる．

syntax.ml:

```
  ...
type tyvar = int

 type ty =
     TyInt
   | TyBool
   | TyVar of tyvar
   | TyFun of ty * ty

(* pretty printing *)
let pp_ty = ...

let fresh_tyvar =
  let counter = ref 0 in
  let body () =
    let v = !counter in
      counter := v + 1; v
  in body

let rec freevar_ty ty = ... (*  ty -> tyvar MySet.t *)
```

図 4.1: ML^3 型推論の実装 (1)

4.3.3 単一化

型変数と型代入を導入したところで型付け規則をもう一度見てみよう．T-If や T-Plus などの規則は「条件式の型は bool でなくてはならない」「then 節と else 節の式の型は一致していなければならない」「引数の型は int でなくてはならない」という制約を課していることがわかる．

これらの制約を ML^2 に対する型推論では，型 (すなわち TyInt などの定義される言語の型を表現した値) の比較を行うことでチェックしていた．例えば与えられた式 e が $e_1 + e_2$ の形をしていたときには，e_1 の型 τ_1 と e_2 の型 τ_2 を再帰的にアルゴリズムを呼び出すことにより推論し，**それらが int であることをチェックしてから**全体の型として int を返していた．

しかし，型の構文が型変数で拡張されたいま，この方法は不十分である．というのは，部分式の型（上記の τ_1 と τ_2）に型変数が含まれるかもしれないからである．例えば，fun x $\to$ 1 + x という式の型推論過程を考えてみる．まず，$\emptyset \vdash$ fun x $\to$ 1 + x: int$\to$int であることに注意しよう.（実際に導出木を書いてチェックしてみること.）したがって，型推論アルゴリズムは，この式の型として int$\to$int を返すように実装するのが望ましい．

では，空の型環境 $\emptyset$ と上記の式を入力として，型推論アルゴリズムがどのように動くべきかを考えてみよう．この場合，まず T-Fun を下から上に読んで，x の型を型変数 α とおいた型環境 $x:\alpha$ の下で 1 + x の型推論をすることになる．その後，各部分式 1 と x の型を，アルゴリズムを再帰的に呼び出すことで推論し，int と α を得る．ML^2 の型推論では，ここでそれぞれの型が int であるかどうかを単純比較によってチェックし，int でなかったら型エラーを報告していた．しかし今回は後者の型が α であって int ではないため，**単純比較による部分式の型のチェックだけでは型推論が上手くいかない.**

では，どうすれば良いのだろうか. 定石として知られている手法は**制約による型推論** (*constraint-based type inference*) という手法である．この手法では，与えられたプログラムの各部分式から型変数に関する**制約** (*constraint*) が生成されるものと見て，式をスキャンする過程で制約を集め，その制約をあとで解き型代入を得る，という形で型推論アルゴリズムを設計する．例えば，上記の例では,「α は実は int である」という制約が生成される．この制約を解くと型代入 $\{\alpha \mapsto \texttt{int}\}$ が得られる．

上記の場合は制約が単純だったが，T-App で関数 e_1 の受け取る引数の型と e_2 の型が一致すること，また T-If で then 節と else 節の式の型が一致することを検査するためには，より一般的な，

> 与えられた型のペアの集合 $\{(\tau_{11}, \tau_{12}), \ldots, (\tau_{n1}, \tau_{n2})\}$ に対して，$\mathcal{S}\tau_{11} = \mathcal{S}\tau_{12}$, $\ldots$, $\mathcal{S}\tau_{n1} = \mathcal{S}\tau_{n2}$ なる $\mathcal{S}$ を求めよ

という制約解消問題を解かなければいけない．このような問題は**単一化** (*unification*) 問題と呼ばれ，型推論だけではなく，計算機による自動証明などにおける基本的な問題として知られている．例えば，α と int は $\mathcal{S}(\alpha) =$ int なる型代入 $\mathcal{S}$ により単一化できる．また，$\alpha\to$bool と (int$\to\beta)\to\beta$ は $\mathcal{S}(\alpha) =$ int$\to$bool かつ $\mathcal{S}(\beta) =$ bool なる $\mathcal{S}$ により単一化できる．

単一化問題は，対象（ここでは型）の構造や変数の動く範囲によっては，非常に難しくな

るが[3]，ここでは，型が単純な木構造を持ち，型代入も単に型変数に型を割当てるだけのもの（**一階の単一化** (*first-order unification*) と呼ばれる問題）なので，解である型代入を求めるアルゴリズムが存在する.[4]（しかも，求まる型代入がある意味で「最も良い」解であることがわかっている.）

一階の単一化を行うアルゴリズム $\mathcal{U}(X)$ は，型のペアの集合 X を入力とし，X 中のすべての型のペアを同じ型にするような型代入を返す.（そのような型代入が存在しないときにはエラーを返す.）$\mathcal{U}$ は以下のように定義される.

$$
\begin{array}{lcl}
\mathcal{U}(\emptyset) & = & \emptyset \\
\mathcal{U}(\{(\tau,\tau)\} \uplus X') & = & \mathcal{U}(X') \\
\mathcal{U}(\{(\tau_{11}\to\tau_{12}, \tau_{21}\to\tau_{22})\} \uplus X') & = & \mathcal{U}(\{(\tau_{11},\tau_{21}),(\tau_{12},\tau_{22})\} \uplus X') \\
\mathcal{U}(\{(\alpha,\tau)\} \uplus X') \quad (\text{if } \tau \neq \alpha) & = & \begin{cases} \mathcal{U}([\alpha\mapsto\tau]X') \circ [\alpha\mapsto\tau] & (\alpha \notin FTV(\tau)) \\ \textbf{error} & (\text{その他}) \end{cases} \\
\mathcal{U}(\{(\tau,\alpha)\} \uplus X') \quad (\text{if } \tau \neq \alpha) & = & \begin{cases} \mathcal{U}([\alpha\mapsto\tau]X') \circ [\alpha\mapsto\tau] & (\alpha \notin FTV(\tau)) \\ \textbf{error} & (\text{その他}) \end{cases} \\
\mathcal{U}(\{(\tau_1,\tau_2)\} \uplus X') & = & \textbf{error} \quad (\text{その他の場合})
\end{array}
$$

ここで，$\emptyset$ は空の型代入を表し，$[\alpha\mapsto\tau]$ は α を τ に写す（そしてそれ以外の型変数については何も行わない）型代入である．また $FTV(\tau)$ は τ 中に現れる型変数の集合である．また，$X \uplus Y$ は，$X \cap Y = \emptyset$ のときの $X \cup Y$ を表す記号である.

この $\mathcal{U}$ の定義は以下のように X を入力とする単一化アルゴリズムとして読める:

- X が空集合であれば空の代入を返す.
- そうでなければ，X から型のペア (τ_1,τ_2) を任意に一つ選び，それ以外の部分を X' とし，(τ_1,τ_2) がどのような形をしているかによって，以下の各動作を行う.
  - τ_1 と τ_2 がすでに同じ形であった場合: X' について再帰的に単一化を行い，その結果を返せばよい.（τ_1 と τ_2 はすでに同じ形なので，残りの制約集合 X' の解がそのまま全体の解となる.）
  - 選んだ型のペアがどちらも関数型の形をしていた場合，すなわち τ_1 が $\tau_{11}\to\tau_{12}$ の形をしており，τ_2 が $\tau_{21}\to\tau_{22}$ の形をしていた場合: τ_1 と τ_2 が同じ形となるためには τ_{11} と τ_{21} が同じ形であり，かつ τ_{12} と τ_{22} が同じ形であればよい．これを満たす型代入を求めるために，$\mathcal{U}$ を $\{(\tau_{11},\tau_{21}),(\tau_{12},\tau_{22})\} \cup X'$ を入力として再帰的に呼び出し，帰ってきた結果を全体の結果とする.

---

[3]問題設定によっては**決定不能** (*undecidable*) になることもある．決定不能であるとは，いい加減に言えば，かつすべての入力について有限時間で停止し正しい出力を返すプログラムが存在しないことを言う．従って，決定不能な問題を計算機でなんとかしようとすると，一部の入力については正しくない答えを返すことを許容するか，一部の入力については停止しないことを許容しなければならない.

[4]このアルゴリズムは，Prolog などの論理型言語と呼ばれるプログラミング言語の処理系において多く用いられる.

– 選んだ型のペアが型変数と型のペア，すなわち (α, τ) か (τ, α) の形をしていた場合[5]: この場合，型変数 α は τ でなければならないことがわかる．したがって，残りの制約 X' 中の α に τ を代入した制約 $[\alpha \mapsto \tau]X'$ を再帰的に解き，得られた解に α を τ に代入する写像 $[\alpha \mapsto \tau]$ を合成して得られる写像 $\mathcal{U}([\alpha \mapsto \tau]X') \circ [\alpha \mapsto \tau]$ を解として返せばよい．ところが，ここで注意すべきことが一つある．もし τ 中に α が現れていた場合[6]，ここでエラーを検出しなければならない．(なぜなのかを考察する課題を以下に用意している．)

Exercise 4.3.3 上の単一化アルゴリズムを

```
val unify : (ty * ty) list -> subst
```

として実装せよ．

Exercise 4.3.4 単一化アルゴリズムにおいて，$\alpha \notin FTV(\tau)$ という条件はなぜ必要か考察せよ．

4.3.4 ML^3 型推論アルゴリズム

以上を総合すると，ML^3 のための型推論アルゴリズムが得られる．例えば，e_1+e_2 式に対する型推論は，T-Plus 規則を下から上に読むと，

1. Γ, e_1 を入力として型推論を行い，$\mathcal{S}_1$，τ_1 を得る．
2. Γ, e_2 を入力として型推論を行い，$\mathcal{S}_2$，τ_2 を得る．
3. 型代入 $\mathcal{S}_1, \mathcal{S}_2$ を $\alpha = \tau$ という形の方程式の集まりとみなして，$\mathcal{S}_1 \cup \mathcal{S}_2 \cup \{(\tau_1, \texttt{int}), (\tau_2, \texttt{int})\}$ を単一化し，型代入 $\mathcal{S}_3$ を得る．
4. $\mathcal{S}_3$ と `int` を出力として返す．

となる．部分式の型推論で得られた型代入を方程式とみなして，再び単一化を行うのは，ひとつの部分式から $[\alpha \mapsto \tau_1]$，もうひとつからは $[\alpha \mapsto \tau_2]$ という代入が得られた時に τ_1 と τ_2 の整合性が取れているか（単一化できるか）を検査するためである．

Exercise 4.3.5 他の型付け規則に関しても同様に型推論の手続きを与えよ (レポートの一部としてまとめよ)．そして，図 4.2 を参考にして，型推論アルゴリズムの実装を完成させよ．

---

[5] (α, α) の形だった場合はこのケースではなく，一つ前のケースに当てはまる．

[6] 繰り返しになるが，τ が α 自体であった場合はこのケースには当てはまらない．ここでエラーを報告しなければならないのは，例えば τ が $\alpha \to \alpha$ の場合である．

Exercise 4.3.6 [⋆⋆] 再帰的定義のための let rec 式の型付け規則は以下のように与えられる．

$$\frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash e_1 : \tau_2 \qquad \Gamma, f : \tau_1 \to \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{let\ rec}\ f = \mathsf{fun}\ x\ \to e_1\ \mathsf{in}\ e_2 : \tau} \quad \text{(T-LetRec)}$$

型推論アルゴリズムが let rec 式を扱えるように拡張せよ．

Exercise 4.3.7 [⋆⋆] 以下は，リスト操作に関する式の型付け規則である．リストには要素の型を τ として τ `list` という型を与える．

$$\frac{}{\Gamma \vdash [] : \tau\ \texttt{list}} \quad \text{(T-Nil)}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau\ \texttt{list}}{\Gamma \vdash e_1 :: e_2 : \tau\ \texttt{list}} \quad \text{(T-Cons)}$$

$$\frac{\Gamma \vdash e_1 : \tau\ \texttt{list} \qquad \Gamma \vdash e_2 : \tau' \qquad \Gamma, x : \tau, y : \tau\ \texttt{list} \vdash e_3 : \tau'}{\Gamma \vdash \mathsf{match}\ e_1\ \mathsf{with}\ [] \to e_2 \mid x :: y \to e_3 : \tau'} \quad \text{(T-Match)}$$

型推論アルゴリズムがこれらの式を扱えるように拡張せよ．

typing.ml:

```
type subst = (tyvar * ty) list

let rec subst_type subst t = ...

(* eqs_of_subst : subst -> (ty * ty) list
   型代入を型の等式集合に変換                    *)
let eqs_of_subst s = ...

(* subst_eqs: subst -> (ty * ty) list -> (ty * ty) list
   型の等式集合に型代入を適用                                *)
let subst_eqs s eqs = ...

let rec unify l = ...

let ty_prim op ty1 ty2 = match op with
    Plus -> ([(ty1, TyInt); (ty2, TyInt)], TyInt)
  | ...

let rec ty_exp tyenv = function
    Var x ->
     (try ([], Environment.lookup x tyenv) with
         Environment.Not_bound -> err ("variable not bound: " ^ x))
  | ILit _ -> ([], TyInt)
  | BLit _ -> ([], TyBool)
  | BinOp (op, exp1, exp2) ->
      let (s1, ty1) = ty_exp tyenv exp1 in
      let (s2, ty2) = ty_exp tyenv exp2 in
      let (eqs3, ty) = ty_prim op ty1 ty2 in
      let eqs = (eqs_of_subst s1) @ (eqs_of_subst s2) @ eqs3 in
      let s3 = unify eqs in (s3, subst_type s3 ty)
  | IfExp (exp1, exp2, exp3) -> ...
  | LetExp (id, exp1, exp2) -> ...
  | FunExp (id, exp) ->
      let domty = TyVar (fresh_tyvar ()) in
      let s, ranty =
        ty_exp (Environment.extend id domty tyenv) exp in
        (s, TyFun (subst_type s domty, ranty))
  | AppExp (exp1, exp2) -> ...
  | _ -> Error.typing ("Not Implemented!")
```

図 4.2: ML^3 型推論の実装 (2)