4.1 能書き

これまでに実装したインタプリタは，与えられたプログラムを，セマンティクスにしたがって評価することにより実行結果を得ていた．例えば，実装したインタプリタを使って式 3+5 を評価すると 8 が評価結果として返ってくる．式 let x = 2 in x + 3 を評価すると 5 が評価結果として返ってくる．

では，プログラムの実行結果についての情報を得る方法は評価だけなのだろうか．**プログラムを実行することなく**プログラムの実行結果についてなんらかの情報を予測することは可能だろうか．この章のテーマは，プログラムを実行せずに解析して，その実行についての情報を得る方法である**静的解析** (*static analysis*) である.[1]

プログラムを実行すれば実行結果が得られるのに，なぜわざわざ静的解析をやろうとするのだろうか．静的解析の用途としては，例えば以下のものがある．

- 静的解析によって，プログラムの実行に一切影響を与えないプログラム中の部分を発見したり，常に一定の値を取る変数を発見するなどして，プログラムの実行効率を上げることができる．これらは言語処理系，特にコンパイラの文脈では**最適化** (*optimization*) として知られる処理である．

- プログラム中には「絶対に成り立っているはずというプログラマの意図」を**アサーション** (*assertion*) として記述することがある．例えば，以下の C 言語の関数 sum は 2 つの整数引数 x と y をとり，x+y を返す関数であるというプログラマの意図が `assert(ret == x + y);` という文で表現されている.[2]

```
/* Returns x+y */
unsigned int sum(unsigned int x, unsigned int y)
  unsigned int i = 0;
  unsigned int ret = x;
  while (i < y)
    ++i; ++ret;

  assert(ret == x + y);
  return ret;
```

  アサーションはプログラマが「絶対に成り立つ」と表明した条件なので，これが実際に成り立っていることを保証するのは，プログラムの信頼性を向上させる上で重要で

[1] ここでいう「静的」とは「プログラムを実行する前に」という意味である．反対にプログラムを実行させて行う解析を**動的解析** (*dynamic analysis*) と呼ぶ．例えば，プログラムを実行して実行時間の大部分を占める（すなわち効率化をすることで効果が上がりやすい）関数を探す方法（**プロファイリング** (*profiling*)）や，プログラムを様々な入力で実行してバグを見つける方法（**ソフトウェアテスト** (*software test*)）はそのような動的解析の一種である．

[2] `assert(e)` は実行時には e を評価し，その結果が偽（C 言語では 0）であればエラーを報告してプログラムを終了する関数やマクロとして実装されていることが多い．

ある．これを保証する手段として静的解析が使われることがある．つまり，`assert` 文が実行されるときには引数が `0` ではないことを，プログラムを解析することによっていわば「証明」することで，アサーションが必ず成り立つことを保証するわけである．

ここでは例として `assert` が成り立っていることを保証するための解析について取り上げたがより一般に「プログラムが意図（＝仕様）通りに動作することを証明するための静的解析」を**形式検証** (*formal verification*) と呼ぶ.[3]

形式検証は，テストを補完する方法として最近結構使われ始めている.[4]本章は，簡単な形式検証手法を実装してみることにより，静的解析に馴染んでもらうことを目的としている．静的解析のうち，上で取り上げた最適化については，後日講義で取り上げる予定である．

本章で実装する形式検証手法は**静的な型推論** (*static type inference*) である．OCaml でプログラムを書くと自動で型を推論してくれて，型エラーがあれば知らせてくれるアレである．型推論はプログラム中の式が（評価が停止するならば）どのような値を返すかを**プログラムを実行することなく**解析するので静的解析の一種である．また，プログラムが**実行時に型エラーを起こさない**ことを証明するための手法であるため，形式検証と言える.[5]

本章では，これまでに実装した言語のための型推論を解説する．まず ML^2 のための型推論からはじめて，徐々に言語を拡張しつつ，拡張された言語機能のための型推論を行う方法を解説する．

4.2 ML^2 のための型推論

まず，ML^2 の式に対しての型推論を考える．ML^2 では，トップレベル入力として，式だけでなく let 宣言を導入したが，ここではひとまず式についての型推論のみを考え，let 宣言については最後に扱うことにする．ML^2 の文法は (記法は多少異なるが) 以下のようでった．

$$
\begin{array}{rcl}
e & ::= & x \mid n \mid \mathsf{true} \mid \mathsf{false} \mid e_1\ op\ e_2 \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\
op & ::= & + \mid * \mid <
\end{array}
$$

---

[3] ちなみに，上記のプログラムでは，`while` ループで条件がテストされる際に必ず `ret - i == x && i <= y` が成り立っていることを発見できれば，アサーションが必ず成り立つことが証明できる.（ループを抜けたときには，`while` 文の条件節が偽になるはずなので，`i >= y` が成り立っているはずである．すると，上記の条件と合わせて `ret - i == x && i == y` が成り立っていることになり，ここから `ret == x + y` が導ける.）このようなループ文の先頭に到達したときに必ず成り立っている条件を**ループ不変条件** (*loop invariant*) と呼ぶ．良いループ不変条件を発見するのは，形式検証においてとても重要なテクニックである．また，自分でプログラムを書く際にも，ループ不変条件を意識して書くことで，バグを減らせることが多い．

[4] 例えば Facebook は infer という形式検証ツールをソースコード管理ツールと統合して動作させており，プログラマがコミットした内容を自動で検証し，誤りの可能性を自動的に指摘するということをやっているとのことである [].

[5] 上で説明したちょっと格好いい形式検証に比べて，だいぶ保証する性質がしょぼく見えるかも知れないが，人間はそういうしょぼいエラーを含むプログラムを頻繁に書く（実行時型エラーに遭遇してしょんぼりした経験ない？），軽い解析なのに結構役に立つ，モジュールに基づく情報の隠蔽と相性が良い，関数型言語ととても相性がよい，型推論をベースにしてさらに格好いい形式検証を作ることができる等の利点がある．

ここでは ⟨ 式 ⟩ の代わりに e という記号 (メタ変数)，⟨ 識別子 ⟩ の代わりに x という記号 (メタ変数) を用いている．また，型 (メタ変数 τ) として，（ML^2 は関数を含まず，値は整数値とブール値のみなので）整数を表す `int`, 真偽値を表す `bool` を考える．[6]

$$\tau ::= \texttt{int} \mid \texttt{bool}.$$

4.2.1 型判断と型付け規則

我々がこれから作ろうとする型推論アルゴリズムは，式 e を受け取って，その e の型を（くどいようだが）e を評価することなく推論する．ここでさらっと「e の型」と書いたが，この言葉の意味するところはそんなに明らかではない．素直に考えれば「e を評価して得られる値 v の型」ということになるのだが「じゃあ v の型って何？」「v の型を定義できたとして，型推論アルゴリズムが正しくその型を推論できていることはどう保証するの？」「e が停止しないかもしれないプログラムだったら評価して得られる値はどう定義するの？」などの問題点にクリアに答えられるようにアルゴリズムを作りたい．

そのために，型推論アルゴリズムを作る際には，普通**型とは何か，プログラム e が型 τ を持つのはどのようなときか**等をまず厳密にに定義し，その型を発見するためのアルゴリズムとして型推論アルゴリズムを定義することが多い．このような，型に関する定義やアルゴリズムを含む体系を**型システム** (*type system*) と呼ぶ．[7]具体的には，「式 e が型 τ を持つ」という関係を**型判断** (*type judgment*) と呼び，$e : \tau$ と略記する．[8]何が正しい型判断で，何が間違った型判断なのかをあとで定義するのだが，例えば「式 $1+1$ は `int` を持つ」ように型システムを作りたいので，$1 + 1 : \texttt{int}$ は正しい型判断になるように，式 $\textsf{if 1 then } 2+3 \textsf{ else } 4 : \texttt{int}$ は型がつかないプログラムなので，$\textsf{if 1 then } 2+3 \textsf{ else } 4 : \tau$ はいかなる τ についても正しくない型判断となるようにしたい．

しかし，型判断を定義するのに e と τ だけでは実は情報が足りない．一般に式には自由変数が現れるからである．，例えば「式 $x + 2$ は `int` を持つ」は正しい判断にしたいだろうか．「それは x の型による」としか言いようがない．(x が `int` 型であれば正しい判断にしたいし，x が `bool` 型であれば正しい型判断と認めたくはないだろう．）このため，自由変数を含む式に対しては，それが持つ型を何か仮定しないと型判断は下せないことになる．この，変数に対して仮定する型に関する情報を**型環境** (*type environment*)(メタ変数 Γ) と呼ぶ．型環境は変数から型への部分関数で表される．これを使えば，変数に対する型判断は，例えば

$\Gamma(x) = \texttt{int}$ の時 $x : \texttt{int}$ である

[6]**メタ変数** (*metavariable*) とは，プログラム中で使われる普通の変数と異なり，「式」「値」「型」などのプログラム中で現れる「もの」を総称的に指すために使われる変数である．例えば，上記の BNF では「式」を表すメタ変数として e が，「自然数」を表すメタ変数として n が用いられている．また，少しややこしいが，「変数」を表すメタ変数として x が用いられている．なお，「式 (expression) を表すメタ変数として e を用いる」ことを表す英語の表現はいくつかあり，"Expressions are ranged over by metavariable e" とか，"e is the metavariable that represents an expression" とか言ったりする．

[7]大体動けばいいんだよ，こまけぇこたぁいいんだよ！という考えもあるだろうが，だいたい動くと思って作ったものが動かないことはよくある．

[8]「判断」という言葉はフレーゲに由来するらしい．裏は取っていない．

と言える．このことを考慮に入れて，型判断は，$\Gamma \vdash e : \tau$ と記述し，

型環境 Γ の下で式 e は型 τ を持つ

と読む．また，空の型環境を $\emptyset$ で表す．$\vdash$ は数理論理学などで使われる記号で「〜という仮定の下で判断〜が導出・証明される」くらいの意味である．インタプリタが (変数を含む) 式の値を計算するために環境を使ったように，型推論器が式の型を計算するために型環境を使っていると考えてもよい．式 $\Gamma \vdash e : \tau$ が成り立つような Γ と τ が存在するときに，e に**型がつく** (*well-typed*)，あるいは e は**型付け可能** (*typable*) という．逆にそのような Γ と τ が存在しないときに，e は**型がつかない** (*ill-typed*)，あるいは e は**型付け不能** (*untypable*) という．

> **型環境の表し方** $\Gamma(x_1) = \tau_1, \ldots, \Gamma(x_n) = \tau_n$ を満たし，それ以外の変数については型が定義されていないような型判断を $x_1 : \tau_1, \ldots, x_n : \tau_n$ と書くことが多い．

型判断を導入したからには「正しい型判断」を定義しなければならない．これには**型付け規則** (*typing rule*) を使うのが定石である．これは，記号論理学の証明規則に似た「正しい型判断」の導出規則で

$$\frac{\langle \text{型判断}_1 \rangle \qquad \cdots \qquad \langle \text{型判断}_n \rangle}{\langle \text{型判断} \rangle} \quad (\langle \text{規則名} \rangle)$$

という形をしている．横線の上の $\langle$ 型判断 $_1\rangle, \ldots, \langle$ 型判断 $_n\rangle$ を規則の**前提** (*premise*)，下にある $\langle$ 型判断 $\rangle$ を規則の**結論** (*conclusion*) と呼ぶ．例えば，以下は加算式の型付け規則である．

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 {+} e_2 : \texttt{int}} \quad (\text{T-P{\scriptsize LUS}})$$

この，型付け規則の直感的な意味（読み方）は，

前提の型判断が全て導出できたならば，結論の型判断を導出してよい

ということである．[9]

以下に，ML^2 の型付け規則を示す．

$$\frac{(\Gamma(x) = \tau)}{\Gamma \vdash x : \tau} \quad (\text{T-V{\scriptsize AR}})$$

---

[9]（この脚注は意味がわからなければ飛ばして良い．）厳密には T-PLUS はメタ変数 e_1, e_2, Γ を具体的な式や型環境に置き換えて得られる（無限個の）導出規則の集合を表したものである．例えば，$\emptyset \vdash 1 : \texttt{int}$ という型判断が既に導出されていたとしよう．T-PLUS の Γ を $\emptyset$ に，e_1, e_2 をともに，1 に具体化することによって，規則の**インスタンス**，**具体例** (*instance*)

$$\frac{\emptyset \vdash 1 : \texttt{int} \qquad \emptyset \vdash 1 : \texttt{int}}{\emptyset \vdash 1 + 1 : \texttt{int}}$$

が得られる．この具体化された規則を使うと，型判断 $\emptyset \vdash 1 + 1 : \texttt{int}$ が導出できる．

$$\frac{}{\Gamma \vdash n : \texttt{int}} \quad \text{(T-INT)}$$

$$\frac{(b = \texttt{true} \text{ または } b = \texttt{false})}{\Gamma \vdash b : \texttt{bool}} \quad \text{(T-BOOL)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \quad \text{(T-PLUS)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 * e_2 : \texttt{int}} \quad \text{(T-MULT)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 < e_2 : \texttt{bool}} \quad \text{(T-LT)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textsf{if } e_1 \textsf{ then } e_2 \textsf{ else } e_3 : \tau} \quad \text{(T-IF)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \textsf{let } x = e_1 \textsf{ in } e_2 : \tau_2} \quad \text{(T-LET)}$$

規則 T-LET に現れる $\Gamma, x : \tau$ は Γ に x は τ であるという情報を加えた拡張された型環境で，より厳密な定義としては，

$$dom(\Gamma, x : \tau) = dom(\Gamma) \cup \{x\}$$
$$(\Gamma, x : \tau)(y) = \begin{cases} \tau & (\text{if } x = y) \\ \Gamma(y) & (\text{otherwise}) \end{cases}$$

と書くことができる.（$dom(\Gamma)$ は Γ の定義域を表す.）規則の前提として括弧内に書かれているのは**付帯条件** (*side condition*) と呼ばれるもので，規則を使う際に成立していなければならない条件を示している.

各々の型付け規則がなぜそのように定義されているか，少しずつ説明を加える.[10]

T-VAR: $\Gamma(x) = \tau$ であれば，Γ のもとで式 x が型 τ を持つという判断を導出してよい．Γ が式の中の自由変数の型を決めているという上述の説明から理解できるはずである.

T-INT,T-BOOL: 整数定数 n は，いかなる型環境の下でも型 `int` を持つ．また，式 `true` と式 `false` は，いかなる型環境の下でも型 `bool` を持つ．これらは直観的に理解できると思う.

[10] 一応書いておくと，ここで説明するのはあくまで理解の助けにするための，型付け規則の背後にある直観であって，型付け規則自体ではない.

T-Plus,T-Mult: 型環境 Γ の下で式 e_1 と式 e_2 が型 `int` を持つことが導出できたならば，Γ の下で式 e_1+e_2 が `int` を持つことを導出してよい．式 e_1*e_2 も同様である．これらは式 e_1+e_2 と e_1*e_2 が，それぞれ整数の上の演算であることから設けられた規則である．

T-Lt: 型環境 Γ の下で式 e_1 と式 e_2 が型 `int` を持つことが導出できたならば，Γ の下で式 $e_1<e_2$ が `bool` を持つことを導出してよい．これらは式 $e_1<e_2$ が整数の比較演算で，返り値がブール値であることから設けられた規則である．

T-If: 型環境 Γ の下で式 e_1 が `bool` を持ち，式 e_2 と式 e_3 が**同一の**型 τ を持つならば，if e_1 then e_2 else e_3 がその型 τ を持つことを導出してよい．式 e_1 は if 式の条件部分なので，型 `bool` を持つべきであることは良いであろう．式 e_2 と式 e_3 が同一の型 τ を持つべきとされていること，if 式全体としてその型 τ を持つとされていることについては少し注意が必要である．これは，条件式 e_1 が true と false のどちらに評価されても実行時型エラーが起こらないようにするために設けられている条件である．これにより，**実際は絶対に実行時型エラーが起こらないのに型付け可能ではないプログラムが生じる．**たとえば，(if true then 1 else false) + 3 というプログラムを考えてみよう．このプログラムは，if 式が必ず 1 に評価されるため，実行時型エラーは起こらない．しかし，この if 式の then 節の式 1 には型 `int` がつき，else 節の式 false には型 `bool` がつくので，if 式は型付け不能である．[11]

T-Let: 型環境 Γ の下で式 e_1 が型 τ_1 を持ち，式 e_2 が Γ を $x\!:\!\tau_1$ というエントリで拡張して得られる型環境 $\Gamma, x\!:\!\tau_1$ の下で型 τ_2 を持つならば，式 let $x = e_1$ in e_2 は全体として τ_2 を持つという判断を導いてよい．この規則は let 式がどのように評価されるかと合わせて考えると分かりやすい．式 let $x = e_1$ in e_2 を評価する際には，まず e_1 を現在の環境で評価し，得られた結果に x を束縛した上で e_2 を評価して，その結果を全体の評価結果とする．そのため，型付け規則においても，e_1 の型付けには（「現在の環境」に対応する）型環境 Γ を使い，e_2 の型付けには e_1 の型 τ_1 を x の型とした型環境 $\Gamma, x\!:\!\tau_1$ を用いるのである．

ここで型判断 $\Gamma \vdash e\!:\!\tau$ が**導出できる** derivable とは，根が型判断 $\Gamma \vdash e\!:\!\tau$ で，上記のすべての辺が型付け規則に沿っている木が存在することである．(すべての葉は前提が無い型付け規則が適用された形になっている．）この木を型判断 $\Gamma \vdash e\!:\!\tau$ を導出する**導出木** (*derivation tree*) という．例えば，以下は型判断 $x\!:\!\mathtt{int} \vdash \mathsf{let\ y = 3\ in\ x + y}\!:\!\mathtt{int}$ の導出木である．

$$\dfrac{\dfrac{}{x\!:\!\mathtt{int} \vdash 3\!:\!\mathtt{int}}\ \text{T-INT} \qquad \dfrac{\dfrac{}{x\!:\!\mathtt{int}, y\!:\! \vdash x\!:\!\mathtt{int}}\ \text{T-VAR} \qquad \dfrac{}{x\!:\!\mathtt{int}, y\!:\! \vdash y\!:\!\mathtt{int}}\ \text{T-VAR}}{x\!:\!\mathtt{int}, y\!:\!\mathtt{int} \vdash x + y\!:\!\mathtt{int}}\ \text{T-PLUS}}{x\!:\!\mathtt{int} \vdash \mathsf{let}\ y = 3\ \mathsf{in} x + y\!:\!\mathtt{int}}\ \text{T-LET}$$

この導出木が存在することが，型判断 $x\!:\!\mathtt{int} \vdash \mathsf{let\ y = 3\ in\ x + y}\!:\!\mathtt{int}$ が正しいということの定義である．

---

[11] ある式 e が型付け不能であることを言うには，いかなる Γ と τ をもってきても，$\Gamma \vdash e\!:\!\tau$ を導けないことを言わなければならないので，この説明は厳密には不十分である．

4.2.2 型推論アルゴリズム

以上を踏まえると，型推論アルゴリズムの仕様は，以下のように考えることができる．

入力: 型環境 Γ と式 e.

出力: $\Gamma \vdash e : \tau$ という型判断が導出できるような型 τ．もしそのような型がなければエラーを報告する．

さて，このような仕様を満たすアルゴリズムを，どのように設計したらよいだろうか．これは，$\Gamma \vdash e : \tau$ を根とする導出木を構築すればよい．では，このような導出木をどのように作ればよいだろうか．

この答えは型付け規則から得られる．上に挙げた型付け規則は**構文主導な規則** (*syntax-directed rules*) になっているというよい性質を持っている．これは，Γ と e が与えられたときに，$\Gamma \vdash e : \tau$ が成り立つような τ が存在するならば，これを導くような規則が e の形から一意に定まるという性質である．例えば，Γ と e が与えられ，e が $e_1 + e_2$ という形をしていたとしよう．このとき，型推論アルゴリズムは $\Gamma \vdash e : \tau$ を根とする導出木を構築しようとする．型付け規則をよく見ると，このような導出木は（存在するならば）最後の導出規則が T-Plus でしかありえない．すなわち，

$$\frac{\vdots}{\Gamma \vdash e : \tau}\ \textrm{T-P{\scriptsize LUS}}$$

という形の導出木だけを探索すればよいことになる．このように適用可能な最後の導出規則が e の形から一意に定まる型付け規則を構文主導であるという．

構文主導な型付け規則を持つ型システムでは，各規則を下から上に読むことによって型推論アルゴリズムを得ることができることが多い．例えば，T-Int は入力式が整数リテラルならば，型環境に関わらず，`int` を出力する，と読むことができる．また，T-Plus は

> 入力式 e が $e_1 + e_2$ の形をしていたならば，Γ と e_1 を再帰的に型推論アルゴリズムに入力して型を求めて（これを τ_1 とする）Γ と e_2 とを再帰的に型推論アルゴリズムに入力して型を求めて（これを τ_2 とする）τ_1 も τ_2 も両方とも `int` であった場合には `int` 型を出力する

と読むことができる．[12]

Exercise 4.2.1 図 4.1, 図 4.2 に示すコードを参考にして，型推論アルゴリズムを完成させよ．

[12] 明示的に導出木を構築していないので，なぜこれで「導出木を構築している」ことになるのかよくわからないかもしれない．この型推論アルゴリズムは再帰呼出しをしているが，この再帰呼出しの構造が導出木に対応している．

syntax.ml:

```
type ty =
    TyInt
  | TyBool

let pp_ty = function
      TyInt -> print_string "int"
    | TyBool -> print_string "bool"
```

main.ml:

```
open Typing

let rec read_eval_print env tyenv =
   print_string "# ";
   flush stdout;
   let decl = Parser.toplevel Lexer.main (Lexing.from_channel stdin) in
   let ty = ty_decl tyenv decl in
   let (id, newenv, v) = eval_decl env decl in
     Printf.printf "val %s : " id;
     pp_ty ty;
     print_string " = ";
     pp_val v;
     print_newline();
     read_eval_print newenv tyenv

let initial_tyenv =
   Environment.extend "i" TyInt
     (Environment.extend "v" TyInt
       (Environment.extend "x" TyInt Environment.empty))

let _ = read_eval_print initial_env initial_tyenv
```

図 4.1: ML^2 型推論の実装 (1)

typing.ml:

```
open Syntax

exception Error of string

let err s = raise (Error s)

(* Type Environment *)
type tyenv = ty Environment.t

let ty_prim op ty1 ty2 = match op with
    Plus -> (match ty1, ty2 with
                 TyInt, TyInt -> TyInt
               | _ -> err ("Argument must be of integer: +"))
    ...
  | Cons -> err "Not Implemented!"

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 tyarg1 = ty_exp tyenv exp1 in
      let tyarg2 = ty_exp tyenv exp2 in
        ty_prim op tyarg1 tyarg2
  | IfExp (exp1, exp2, exp3) ->
      ...
  | LetExp (id, exp1, exp2) ->
      ...
  | _ -> err ("Not Implemented!")

let ty_decl tyenv = function
    Exp e -> ty_exp tyenv e
  | _ -> err ("Not Implemented!")
```

図 4.2: ML^2 型推論の実装 (2)