8　多相型の体系

型システムの役割は、7+true などの無意味な式を発見して、排除することにあった。しかし、単純な型システムでは、時として制限が強くなり過ぎてしまい、意味がある計算をあらわしている式まで排除してしまうことがある。そこで、型システムを拡張することが考えられる。

この章は、そのような拡張のうち、非常に強力で応用範囲の広い多相型 (polymorphic type) を扱う。これは、「すべての型」をあらわす型を導入して、型システムを拡張するものである。

まず、なぜ、そのような型が必要かを、例で見てみよう。

2 つの要素からなる対の左右をいれかえる関数 $\lambda x.\,(\mathtt{right}(x), \mathtt{left}(x))$ を型推論すると、$(\alpha \times \beta) \to (\beta \times \alpha)$ という型が得られる。よって、この関数を、$(\mathtt{int} \times \mathtt{int}) \to (\mathtt{int} \times \mathtt{int})$ という型の式として使うこともできるし、$(\mathtt{int} \times \mathtt{bool}) \to (\mathtt{bool} \times \mathtt{int})$ という型の式として使うこともできる。しかしながら、前節までの体系では、1 つのプログラムにおいて、ある場所では $(\mathtt{int} \times \mathtt{int}) \to (\mathtt{int} \times \mathtt{int})$ という型の式として使い、別の場所では、$(\mathtt{int} \times \mathtt{bool}) \to (\mathtt{bool} \times \mathtt{int})$ という型の式として使うことはできなかった。つまり、

$$\vdash \mathtt{let}\ f = \lambda x.\,(\mathtt{right}(x), \mathtt{left}(x))\ \mathtt{in}\ f@(10, 20)\ :\ \mathtt{int} \times \mathtt{int}$$
$$\vdash \mathtt{let}\ f = \lambda x.\,(\mathtt{right}(x), \mathtt{left}(x))\ \mathtt{in}\ f@(10, \mathtt{true})\ :\ \mathtt{int} \times \mathtt{bool}$$

は導出できる (これらの式は型を持つ) が、それらの両方を同時に使おうとして、

$$\mathtt{let}\ f = \lambda x.\,(\mathtt{right}(x), \mathtt{left}(x))\ \mathtt{in}\ (f@(10, 20),\ f@(10, \mathtt{true}))$$

とすると、$(\mathtt{int} \times \mathtt{int}) \times (\mathtt{int} \times \mathtt{bool})$ という型は付かずに、型付け不能になってしまう。

同様に、$\lambda x.x$ という式を、1 つのプログラムの中で、$\mathtt{int} \to \mathtt{int}$ と $\mathtt{bool} \to \mathtt{bool}$ の 2 つの目的で使うことはできなかった。つまり、

$$\mathtt{let}\ f = \lambda x.x\ \mathtt{in}\ \mathtt{if}\ f@\mathtt{true}\ \mathtt{then}\ (f@10) + 3\ \mathtt{else}\ 20$$

という式は型付けに失敗する。このようなプログラムを書きたい場合は、

$$\mathtt{let}\ f = \lambda x.x\ \mathtt{in}\ \mathtt{let}\ g = \lambda x.x\ \mathtt{in}\ \mathtt{if}\ f@\mathtt{true}\ \mathtt{then}\ (g@10) + 3\ \mathtt{else}\ 20$$

というように、型ごとに異なる変数に束縛して使い分ける必要があった。後者のプログラムは、前者よりも長くなり、実行時にプログラムが占めるメモリが無駄となるし、さらに、このようなプログラムの維持のコストが増大する (検証も大変になるし、プログラムを修正する場合に手間が増える) という問題が生じる。

もし、「任意の α と β に対して $(\alpha \times \beta) \to (\beta \times \alpha)$ 型である」や、「任意の α に対して $\alpha \to \alpha$ 型である」という事を意味する型を追加して、型の世界を拡張すれば、上記の式をいろいろな型に対して使うことができる。

多相型 (polymorphic type) は、このような要求に答えるものである。具体的には、$\forall\alpha.\forall\beta.((\alpha \times \beta) \to (\beta \times \alpha))$ や $\forall\alpha.\,(\alpha \to \alpha)$ などという表現を、型の世界に取りいれるものである。

8.1　多相型を持つ体系 CoreML$^+$

体系 CoreML に、このような多相型を取りいれた体系として、CoreML$^+$ を定義する。

まず、型の世界を以下のように拡張する。

$$A, B ::= \alpha \mid \mathtt{int} \mid \mathtt{bool} \mid A \times B \mid A \to B$$
$$P ::= A \mid \forall\alpha.P$$

A, B は CoreML の型と同じであるが、いま導入しようとしている多相型と区別して、単相型 (monomorphic type) と呼ぶ。P が多相型であり、その形は、単相型 A の前に 0 個以上の $\forall\alpha.$ をつけたものとなっている。この定義のものとで、たとえば、$\forall\alpha.\forall\beta.((\alpha \times \beta) \to (\beta \times \alpha))$ が型となることがわかるであろう。このほかにも、$\forall\alpha.((\alpha \times \texttt{bool}) \to (\texttt{int} \to \alpha))$ も (多相) 型となる。

2 種類の多相型: 上記の定義は、$\forall\alpha.$ が型の構文の一番外に現れることのみを許していて、内側に来ることは許していない。つまり、$(\forall\alpha.(\alpha \times \alpha)) \to \texttt{int}$ や、$\texttt{int} \times (\forall\alpha.\alpha)$ というものは、CoreML$^+$ における型ではない (単相型でも多相型でもない)。
この制限をせずに、$\forall\alpha.$ が型の内側に現れてもよいという定義にした体系は、System F と呼ばれ、Girard と Reynolds による多相型の体系として知られている。System F は、プログラミング言語の型システムの理論的基盤の 1 つとして有用であり広く使われているものであるが、現実のプログラミング言語で直接 System F の型をすべて許しているものはほとんどなく、ML でも許していない。その理由は、このような型を許すと、型推論ができなくなるという問題があるからである。ML の言語の趣旨は、人間が型を一切書かなくてもシステムが自動的に推論してくれる、ということなので、System F の型を全面的にいれるわけにはいかない。本節では、ML にならって、上記の制限をつけた多相型 (let 多相、あるいは ML 多相と呼ばれる) を導入した。
一方、プログラミング言語 Haskell では、System F の型 (および、それをさらに発展させた Fω と呼ばれる体系の型) を使うことができる。これらの型を使った式については、自動的に型推論できないので、プログラマが型を明示的に書く必要がある。

次に、CoreML$^+$ の型付け規則を述べる。まず判断は、$\Gamma \vdash M : A$ の形であるが、このうち Γ の形が以下のものに変更されている。

$$x_1 : P_1, \cdots, x_n : P_n$$

つまり、変数の型が CoreML では単相型に限定されていたが、CoreML$^+$ では多相型に拡張されている。なお、判断 $\Gamma \vdash M : A$ において M の型は単相型 A であることに注意されたい。

さて、CoreML$^+$ の型付け規則である。これは、驚くべきことに、ほとんどの規則が CoreML と同じであり (ただし、CoreML の型付け規則で Γ と書いてある部分は、上記のように「多相型を許す」という風に読みかえる必要がある)、変更が必要なのは、var 規則と let 規則の 2 つだけである。

まず、多相型を導入するのは、ML では let 式の役割としている。

$$\frac{\Gamma \vdash M : A \quad \Gamma, x : \texttt{Gen}(\Gamma; A) \vdash N : B}{\Gamma \vdash \texttt{let}\ x = M\ \texttt{in}\ N : B}\ \text{let}$$

ここで、右側の前提における x の型が、CoreML では A だったものが、$\texttt{Gen}(\Gamma; A)$ になった点が変更点である。

これを定義するため、まず、「型に含まれる自由な型変数の集合」という概念を定義する。単相または多相

型 A, P に対して、その中に含まれる自由な型変数の集合 $\mathtt{FTV}(A)$ や $\mathtt{FTV}(P)$ は以下のように定義される。

$$\begin{aligned}
\mathtt{FTV}(\alpha) &\stackrel{\text{def}}{=} \{\alpha\} \\
\mathtt{FTV}(\mathtt{int}) &\stackrel{\text{def}}{=} \{\} \\
\mathtt{FTV}(\mathtt{bool}) &\stackrel{\text{def}}{=} \{\} \\
\mathtt{FTV}(A \to B) &\stackrel{\text{def}}{=} \mathtt{FTV}(A) \cup \mathtt{FTV}(B) \\
\mathtt{FTV}(A \times B) &\stackrel{\text{def}}{=} \mathtt{FTV}(A) \cup \mathtt{FTV}(B) \\
\mathtt{FTV}(\forall\alpha.P) &\stackrel{\text{def}}{=} \mathtt{FTV}(P) - \{\alpha\}
\end{aligned}$$

ここで、$-$ というのは差集合を取る演算であり、たとえば、$\{a, b\} - \{a, c\} = \{b\}$ である。自由な型変数の集合の例としては、$\mathtt{FTV}(\forall\alpha.(\alpha \to (\beta \times \mathtt{int}))) = \{\alpha, \beta\} - \{\alpha\} = \{\beta\}$ となる。

次に、Gen を定義する。

$$\begin{aligned}
\mathtt{Gen}(\Gamma; A) &\stackrel{\text{def}}{=} \forall\alpha_1.\cdots\forall\alpha_n.A \\
&\quad \text{where} \;\; \mathtt{FTV}(A) - \mathtt{FTV}(\Gamma) = \{\alpha_1, \cdots, \alpha_n\}
\end{aligned}$$

$\mathtt{Gen}(\Gamma; A)$ は、直感的には、A に含まれる型変数たちのうち、Γ に (自由に) 出現しないものを多相型にする ($\forall\alpha.$ をつける) ということである。

やや複雑な定義なので、例を見てみよう。

例 26　Gen の計算例をあげる。

$\mathtt{Gen}(x : \alpha \to \alpha,\ y : \mathtt{int} \times \beta;\ \alpha \times \beta \to \gamma) = \forall\gamma.(\alpha \times \beta \to \gamma)$

$\mathtt{Gen}(x : \forall\alpha.(\alpha \to \alpha),\ y : \mathtt{int} \times \beta;\ \alpha \times \beta \to \gamma) = \forall\alpha.\forall\gamma.(\alpha \times \beta \to \gamma)$

これらの定義を踏まえて let 規則を見てみよう。たとえば、$M = \lambda y.y$ のとき、$\vdash M : \alpha \to \alpha$ が推論できる。そこで、$x : \forall\alpha.(\alpha \to \alpha)$ という仮定のもとで N の型を推論してよい、ということであり、この x を多相型で使えることになる。

同様に、$M = \lambda y.(\mathtt{right}(y), \mathtt{left}(y))$ のとき、$\vdash M : (\alpha \times \beta) \to (\beta \times \alpha)$ が推論できる。そこで、$x : \forall\alpha.\forall\beta.((\alpha \times \beta) \to (\beta \times \alpha))$ という仮定のもとで N の型を推論してよい、ということであり、この x を多相型で使えることになる。

次に、変数規則において、多相型を使う。これは、以下の形にかわる。

$$\frac{((x : P) \in \Gamma) \quad (A \preceq P)}{\Gamma \vdash x : A}\ \mathtt{var}$$

ここで $A \preceq P$ は以下のように定義される。$P = \forall\alpha_1.\cdots\forall\alpha_n.B$ の形とするとき、ある型代入 Θ で、$\Theta(B) = A$ となるとき、および、その時にかぎり、$A \preceq P$ と定義する。

これも例を見てみよう。

$$\begin{aligned}
\mathtt{int} \times \beta \to \mathtt{int} &\preceq \forall\alpha.(\alpha \times \beta \to \alpha) \\
\mathtt{bool} \times \beta \to \mathtt{bool} &\preceq \forall\alpha.(\alpha \times \beta \to \alpha)
\end{aligned}$$

また、$\mathtt{int} \times \mathtt{int} \to \mathtt{int} \preceq \forall\alpha.(\alpha \times \beta \to \alpha)$ ではない。

var 規則は、変数 x が多相型をもつとき、x を使う場面では、その型変数を好きな型に具体化して使ってよいということである。

上記の 2 つの規則を使って、多相型を利用した式の型付けを見てみよう。

まず、$\lambda x.x$ を 2 つの異なる型 $\alpha \to \alpha$ と $(\alpha \to \alpha) \to (\alpha \to \alpha)$ で使う例である。

$$\dfrac{\dfrac{\overline{x : \beta \vdash x : \beta}}{\vdash \lambda x.x : \beta \to \beta} \quad \dfrac{\overline{f : \forall\beta.(\beta \to \beta) \vdash f : (\alpha \to \alpha) \to (\alpha \to \alpha)} \quad \overline{f : \forall\beta.(\beta \to \beta) \vdash f : \alpha \to \alpha}}{f : \forall\beta.(\beta \to \beta) \vdash f@f : \alpha \to \alpha}}{\vdash \texttt{let}\ f = \lambda x.x\ \texttt{in}\ f@f : \alpha \to \alpha}$$

この型付け図の左の方で、$\vdash \lambda x.x : \beta \to \beta$ となっていることから、$\texttt{Gen}(\ ; \beta \to \beta) = \forall\beta.(\beta \to \beta)$ を得る。右の方で、この f に対する var 規則が 2 回現れるが、それぞれで β を異なる型に具体化している。

次に $M = (\texttt{let}\ f = \lambda x.\ (\texttt{right}(x), \texttt{left}(x))\ \texttt{in}\ (f@(10, 20),\ f@(10, \texttt{true})))$ の型付けは以下の通りである。(ここで、$N = \lambda x.\ (\texttt{right}(x), \texttt{left}(x))$ および $P = \forall\alpha.\forall\beta.(\alpha \times \beta) \to (\beta \times \alpha)$ と置いた。)

$$\dfrac{\begin{array}{c}\vdots \\ \vdash N : (\alpha \times \beta) \to (\beta \times \alpha)\end{array} \quad \dfrac{\dfrac{\overline{f : P \vdash f : (\texttt{int} \times \texttt{int}) \to (\texttt{int} \times \texttt{int})}\ \cdots}{f : P \vdash f@(10, 20) : \texttt{int} \times \texttt{int}} \quad \dfrac{\overline{f : P \vdash f : (\texttt{int} \times \texttt{bool}) \to (\texttt{bool} \times \texttt{int})}\ \cdots}{f : P \vdash f@(10, \texttt{true}) : \texttt{bool} \times \texttt{int}}}{f : P \vdash (f@(10, 20),\ f@(10, \texttt{true})) : (\texttt{int} \times \texttt{int}) \times (\texttt{int} \times \texttt{bool})}}{\vdash M : (\texttt{int} \times \texttt{int}) \times (\texttt{int} \times \texttt{bool})}$$

この例でも、左の N の型付けにおいては、Γ の部分が空なので、$\texttt{Gen}(;\ (\alpha \times \beta) \to (\beta \times \alpha)) = \forall\alpha.\forall\beta.(\alpha \times \beta) \to (\beta \times \alpha) = P$ となる。右側の f に対する var 規則 (2 か所) で、この α, β を異なる型に具体化していることがわかる。

体系 CoreML$^+$ における計算規則は CoreML と同じであるので省略する。

以上のように、体系 CoreML$^+$ では、多相型を導入 (let 規則) し、多相型を使う (var 規則) ということを実現している。なお、CoreML$^+$ においては、λ で束縛される変数は多相型でないことを要求しているので (lambda 規則や apply 規則は、CoreML と同じであるため、単相型しか許されない)、λ を使って let をあらわすことはできない。すなわち、CoreML においては、$\texttt{let}\ x = M\ \texttt{in}\ N$ と $(\lambda x.N)\ @\ M$ は全く同じ (型付けるかどうか、と、計算した場合の意味の両方が同じ) であったが、CoreML$^+$ では型付けは同じではない。たとえば、$\texttt{let}\ f = \lambda x.x\ \texttt{in}\ f@f$ は上述した通り型付け可能であるが、$(\lambda f.\ f@f)\ @\ (\lambda x.x)$ は型付けできない。(後者は、f が単相型であるという制限のため型付けに失敗する。)

8.2 CoreML$^+$ に対する型推論

体系 CoreML$^+$ における型推論についても簡単に触れておこう。

ML 系の言語においては、System F のような多相型ではなく、制限された多相 (let 多相) を導入しているが、これは、このように制限すると、型推論ができる (型を持つかどうかが決定可能であり、型推論アルゴリズムが存在する) からである。

CoreML$^+$ に対する型推論アルゴリズムは、CoreML における型推論アルゴリズムの一点のみを修正すれば得られる。すなわち、「型推論のステップ 1: 制約生成」において、$\texttt{let}\ x = M\ \texttt{in}\ N$ の型推論図の作成を修正する。

これも、一般論ではなく例のみで見ることにする。型環境 Γ および項 $\texttt{let}\ f = \lambda x.x\ \texttt{in}\ f@f$ と型 A に対する制約生成を行なっているとする。($\Gamma \vdash \texttt{let}\ f = \lambda x.x\ \texttt{in}\ f@f : A$ という判断に対する制約生成を行っているとする。)

このとき、CoreML$^+$ の let 規則を適用するかわりに、まず項を、$\texttt{let}\ f_1 = \lambda x.x\ \texttt{in let}\ f_2 = \lambda x.x\ \texttt{in}\ f_1@f_2$ に変形して、CoreML の let 規則を何回か適用すればよい。すなわち、以下のようになる。

- 新しい型変数 α_1 を用意して、$\Gamma \vdash \lambda x.x : \alpha_1$ に対する制約生成を行なう。
- 次に、$\Gamma, f_1 : \alpha_1 \vdash \texttt{let}\ f_2 = \lambda x.x\ \texttt{in}\ f_1@f_2$ に対する制約生成を行なう。このために、新しい型変数 α_2 を用意して、$\Gamma \vdash \lambda x.x : \alpha_2$ に対する制約生成を行なう。
- 次に、$\Gamma, f_1 : \alpha_1, f_2 : \alpha_2 \vdash f_1@f_2$ に対する制約生成を行なう。

このようにすることにより、f が出現するたびに違う型になってよい、という多相性に対応した型推論アルゴリズムとすることができる。

一般に、f が let の本体で N 回でてきた場合は N 回の (単相型の)let の繰返しに置きかえて型推論を行えばよい。

このように変形した型推論アルゴリズムが、CoreML$^+$ の型推論問題に対して正しいことは、CoreML の場合と同様に示すことができる。

演習問題: 以下の式の型付け図を、体系 CoreML$^+$ で書きなさい。

1. 適当な A に対して、$\vdash \texttt{let}\ f = \lambda x.\lambda y.x\ \texttt{in}\ (f@\texttt{true})@(f@10@\texttt{false}) : A$
2. 適当な B に対して、$\vdash \texttt{let}\ f = \lambda x.(x, x)\ \texttt{in}\ (f@f, f@10) : B$