

解 説

関数型言語における型理論†



林

晋竹

1. 関数型言語における型の使われ方

ほとんどの高級プログラミング言語は、型の概念をもっている。たとえば、Pascal、C らでは、変数を宣言するときには、その型を指定しなければならない。これらの言語では変数は、概念的にはメモリの一部分の個定された場所を表す名前であり、コンパイルを行う際、それぞれの変数に、どれだけの広さの場所を割りふるべきかを知らねばならないので変数の型により、必要な広さを計算しなければならないからである。これに反して、LISP や関数型言語では、変数はデータ(値)を表す名前であり、データは、S式や関数のようにサイズに制限がないのでヒープ上におかれ、変数には、そのデータへのポインタのみがおかれる。このため上記のような変数の型宣言を行う必要はない。このため、関数型言語での型の使われ方は、CやPascalでの型の使われ方とは、かなり異なっている。

関数型言語では、型は主としてプログラムの「整合性」の確保のために使われる。つまり、PascalやCの型が、もっぱら機械のために使われるのに対し、関数型言語では、ユーザのために使われるといっている。このため、関数型言語における型の理論は、モジュールの理論やプログラム検証の理論と深く結びつき抽象的・数学的な色彩をおびる。

関数型言語において最小限保証されるべき整合性は関数適用の整合性である。関数型言語の、もっとも基本的なプログラム構成子は、適用(application)である。(関数の結合を基本とする言語もあるが、型について考えるとき、この差は、本質的なものではない。)関数 f をプログラム e に適用したものを $f(e)$ とかく。 f は、 e の値 v が適当な種類のデータであることを期待して定義されるので、 v が期待した種類のデータでなければ暴走したり、まったく無意味な値を返す

ことになる。これをさけるために、 $f(e)$ の引数 e の値 v が、 f が期待する種類のものであることを保証する必要がある。この値の関数が期待する値(データ)の種類が関数型言語における型である。すなわち、関数適用の型の整合性とは、 f の入力型(f が期待する型)と、引数 e の値の型が一致することをいう。

関数適用における型の整合性を保証する方法には、動的型検査と静的型検査の二つのアプローチがある。LISP のように変数が任意の型のデータを参照することができる言語では、引数 e の値の型を、あらかじめ知ることはできないので、関数適用の型検査は、実行時に行われる。これが動的型検査である。このアプローチは、変数の型を宣言する必要がなく、また、すべての値を同格に扱うことができるという大きな自由性をもつ反面、実行時の型検査という大きなオーバーヘッドによる効率の低下という欠点をもつ。さらに、実行時に型の検査が可能であるためには、データの型が計算できる必要があり、型の概念が限定されてしまう。また、変数が任意の型の値を参照できるという柔軟性は、バグの可能性を増大させるという悪い側面ももつ。

他方、MLのような型つき関数型言語では、変数が参照できる値の型を限定することにより、プログラムが実行される前に、引数の型を知ることができるようになっているので、実行を行うことなく関数適用の型の整合性をチェックできる。これが静的型検査である。この方法は、プログラミングの自由性が犠牲になる反面、動的型検査によるオーバーヘッドを取り除くことができる。また、型は大雑把な仕様と考えることもできるため、型検査により、プログラムが正しい型をもつかどうかをチェックすることによりプログラムを実行する前に多くのバグを検出することもできる。

静的型検査では、値が決定していないプログラムの型を調べる必要があるので、プログラムの型についての推論を行う必要がある。これを型推論という。関数型プログラミングについて、盛んに研究されている型

† Type theory in functional languages by Susumu HAYASHI (Univ. of Kyoto)

竹 京都大学数理解析研究所

理論は、この型推論と、その意味論の研究である。

関数型言語における型が通常の言語における型と異なる、もう一つの点は、「関数の型」が使われる点である。C. Pascal らの伝統的言語では、型は、整数、小数による近似としての実数、文字列、レコードなどの第一階のデータ型であるが、関数型言語では、第一階のデータ型に加えて、高階関数の型が導入されることが多い。関数 f が型 σ の入力を受けとり、型 τ の出力を返すとき、 f の型は $\sigma \rightarrow \tau$ であるという。この関数の型 $\sigma \rightarrow \tau$ は、他の一階の型と同格に扱われるので、 $\sigma \rightarrow \sigma$ という型の関数 f をうけとり同じ $\sigma \rightarrow \sigma$ という型をもつ関数は、 $(\sigma \rightarrow \sigma) \rightarrow (\sigma \rightarrow \sigma)$ という型をもつ。たとえば、 f を受けとり f を返す恒等関数、 f を受けとり $f \circ f$ を返す twice などは型 $(\sigma \rightarrow \sigma) \rightarrow (\sigma \rightarrow \sigma)$ の関数である。

型 $\sigma \rightarrow \tau$ の直観的な解釈は、 σ から τ への関数の集合であるが、計算機のうえの話であるから、 σ から τ への、すべての集合論的関数をモデルにとると、いろいろ矛盾がおきることがある。実際の言語のデザインでは、 $\sigma \rightarrow \tau$ という型をもつ関数を、型 σ の入力に対し、型 τ の出力を返すようなプログラムとして捉え、通常の数学の関数もたない性質も使ってしまふことが多いからである。たとえば、ML では、恒等関数 Id や twice

$$\text{Id}(f) = f, \text{twice}(f)(x) = f(f(x))$$

は、同じ関数が、 $(\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int})$, $(\text{bool} \rightarrow \text{bool}) \rightarrow (\text{bool} \rightarrow \text{bool})$ のような異なった型を同時にもちことができる。と考える。 Id や twice の定義をみれば、型 σ にかかわらず、 f が $\sigma \rightarrow \sigma$ をもち限り $\text{Id}(f)$, $\text{twice}(f)$ が型 $\sigma \rightarrow \sigma$ をもちことは明らかで、 Id , twice を f からその出力を作り出す操作（アルゴリズム）と考えれば、 σ が、どんな型であろうと、その操作が実行可能であるのは明らかだから、任意の $(\sigma \rightarrow \sigma) \rightarrow (\sigma \rightarrow \sigma)$ という形の型が Id , twice の型となり得ることは納得できる。しかし、数学的に考えると、 $\sigma \rightarrow \tau$ の関数は、グラフであり $(F \subseteq \sigma \times \tau$ という集合で、 $\forall x \in \sigma \exists ! y \in \tau, \langle x, y \rangle \in F$ という性質をもつもの)、 $(\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int})$ と $(\text{bool} \rightarrow \text{bool}) \rightarrow (\text{bool} \rightarrow \text{bool})$ は共通の元をもたない。関数型言語が、ソフトウェア工学・科学の立場から注目されている理由は、意味論に数学的明晰さがあり、抽象的なプログラムも、より自然に考えることができるからである。したがって、高階関数型言語の型が伝統的な数学の枠組み（集合論）から、はみ出す以上、それにかかわる意味論の指導原

理が必要となる。型理論のための新しい数学的指導原理は、スコット理論と構成的理論（直観主義的論理）である。特に、構成的論理は、いわゆる Curry-Howard の原理 (Curry-Howard isomorphism) により、関数型言語の型理論を 70 年代に盛んに研究された構成的数学の形式系や、その意味論と結びつけることを可能にする。このため、型理論は計算機科学からのみではなく、論理学、カテゴリー論の研究者の間でも盛んに研究されるようになってきている^{[H], [R]}。本稿では、関数型言語 ML の型について詳述し、型理論への入門の一助としたい。

2. 型推論と型合成

適用の型の整合性をチェックするためには、 $f(e)$ の f と e の型が分かればよい。すなわち、 f が型 $\sigma \rightarrow \tau$ をもち、 e が型 σ をもつならば、 $f(e)$ は整合的で型 τ をもち、これを、論理の推論図風に書くと、

$$\frac{f \text{ が型 } \sigma \rightarrow \tau \text{ をもち } e \text{ が型 } \sigma \text{ をもち}}{f(e) \text{ は整合的で、型 } \tau \text{ をもち}}$$

となる。プログラムが、型をもてば整合的であると考えるので、「 $f(e)$ は整合的で、型 τ をもち」は、単に「 $f(e)$ は型 τ をもち」といってよい。プログラム e が型 τ をもちことを $e : \tau$ とかけば、この推論は、

$$\frac{f : \sigma \rightarrow \tau \quad e : \sigma}{f(e) : \tau}$$

となる。このような、プログラムの型についての推論法則を使って、プログラムの型を導出することを型推論という。また、部分的に型の与えられているプログラムから、その型を自動的に推論することを型合成という。

型推論・型合成は、Pascal や C のような単純な型のシステムをもつ言語でも行われている。たとえば、

```
var x : int
    .. -(x) ..
```

というプログラムをコンパイルするとき、“-” が、 $\text{int} \rightarrow \text{int}$ という型をもつことが、あらかじめ分かっているの

$$\frac{- : \text{int} \rightarrow \text{int} \quad x : \text{int}}{- : \text{int} \rightarrow \text{int} \quad -x : \text{int}} \quad \frac{}{-(-x) : \text{int}}$$

のように、 $-x$, $-(-x)$ の型を推論できる。コンパイラは、これを自動的に行うから型合成を行っていることになる。

通常の手続き型言語では、複雑な型が存在しないので、大げさに型推論・型合成という必要はないが、高

階関数型言語では、複雑な高階の型が存在するため、言語を正確に把握するには、型推論を正確に定義し、そして、その自動化として型合成のアルゴリズムを理解することが必要となる。これを関数型言語 ML の多相型を例にして解説しよう。

ML は、エジンバラ大学で開発された高階関数型言語で、現在は、エジンバラ大、INRIA、Bell 研などで、実現が行われ、ほぼ実用言語としての形を整えている言語である。(エジンバラ大・計算機科学科では、ML が主力言語として使われており、X-window とのインタフェースなども開発されている。) ML の型システムの大きな特長は、多相型 (polymorphic type) という概念を導入することにより型によるプログラミングの不自由性を大幅に解消するとともに、強力な型合成のアルゴリズムにより、型推論を完全に自動化している点である。

実用言語としての ML は、レコード型、実数型、抽象型のような豊富な型を含む言語であるが、多相型の基本的アイデアを解説するには、これらの具体的な型は不必要なので、いっさい考えない。また、ML の型は、型合成が可能な範囲に制限されているが、その意味を理解しようとするときには、ML の型よりさらに一般的な型を考えた方が分かりやすい。そこで、最初に MacQueen らによって導入された型推論の体系 PT を定義し、その意味論を与え、その部分体系を使って ML の型推論と型合成について説明する。

2.1 多相型理論 PT

PT は、表現、型表現、型推論からなる。表現は、ラムダ計算の項である。すなわち、

定義 1 (表現)。

1. 変数は表現である。
2. e_1, e_2 が表現ならば、 $e_1(e_2)$ は表現である。
3. x が変数で、 e が表現ならば、 $\lambda x. e$ は表現である。

ラムダ計算については、既知のものとして、これ以上解説しない。

PT の表現は、型のついていないプログラムであると考え、次に型表現を定義しよう。

定義 2 (型表現)。

1. 型変数は型表現である。
2. σ_1, σ_2 が型表現ならば、 $\sigma_1 \rightarrow \sigma_2$ は型表現である。
3. α が型変数、 σ が型表現ならば、 $\forall \alpha. \sigma$ は型表現である。

型表現とは、文字どおり型を表すものである。型変数とは、任意の型を表す変数で、もちろん無限個あると仮定する。 $\sigma_1 \rightarrow \sigma_2$ は、入力型が σ_1 ならば、出力型が σ_2 となる関数 (表現) の型である。出力型が σ_2 というのは、必ず型 σ_2 の出力があるともとれるし、出力があれば、その型は必ず σ_2 であるともとれる。PT の範囲では、どちらに解釈してもかまわないが、計算機言語の意味論としては、後者の方が自然だが、説明を簡単にするため、本稿では前者のように解釈する。

さて、型 $\forall \alpha. \sigma$ であるが、これが多相型である。あるデータ v が型 $\forall \alpha. \sigma$ をもつ条件は、どのような型 α に対しても、 v が型 σ をもつことである。たとえば、恒等関数 $Id \lambda x. x$ は、すでに注意したように、 $\alpha \rightarrow \alpha$ という型をもち、 α は任意でよい。したがって $\lambda x. x$ は $\forall \alpha. \alpha \rightarrow \alpha$ という型をもつことになる。同様に、 $twice \lambda f. \lambda x. f(f(x))$ は、 $\forall \alpha. (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$ という型をもつ。

表現と型表現が定義できたから、表現 e の値が型表現 σ が表す型をもつことを推論する型推論を定義しよう。表現 e が値 v をもち、型表現 σ が値の集合としての型 S を表し、 $v \in S$ であることを $e : \sigma$ とかく。 $e : \sigma$ を、型 (type statement) と呼び、 e は型 σ をもつと読む。表現の型は、その自由変数の型によって左右されるから、自由変数の型宣言を定義し、その下で表現の型を定義する。

定義 3 (型宣言)

型の有限集合 $\{x_1 : \sigma_1, \dots, x_n : \sigma_n\}$ が、 $i \neq j$ ならば $x_i \neq x_j$ という条件をみたすとき型宣言という。空集合 $\{\}$ とかく) も型宣言である。型宣言は、 $\Gamma, \Gamma_1, \dots$ とかく。また、 $\Gamma, x : \sigma$ で $\Gamma \cup \{x : \sigma\}$ を表す。

定義 4 (型判定)

Γ が型宣言であるとき、 $\Gamma \vdash e : \sigma$ という形式を型判定形式という。次の推論法則によって導出できる型判定形式を型判定 (judgement)、あるいは、正しい型判定形式という。

$$(asp) \quad \{x : \sigma\} \vdash x : \sigma$$

$$(add) \quad \frac{\Gamma_1 \vdash e : \sigma}{\Gamma_2 \vdash e : \sigma} \quad (\Gamma_1 \subseteq \Gamma_2)$$

$$(\rightarrow I) \quad \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x. e : \sigma \rightarrow \tau}$$

$$(\rightarrow E) \quad \frac{\Gamma \vdash e_1 : \sigma \rightarrow \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1(e_2) : \tau}$$

$$(\forall I) \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall \alpha. \sigma} \quad (\alpha \text{ は } \Gamma \text{ の自由変数ではない})$$

$$(\forall E) \frac{\Gamma \vdash e : \alpha. \sigma}{\Gamma \vdash e : \sigma[\tau/\alpha]}$$

($\forall I$) の条件「 α は Γ の自由変数ではない」というのは、 Γ を $\{x_1 : \sigma_1, \dots, x_n : \sigma_n\}$ とするとき、 $\sigma_1, \dots, \sigma_n$ に α が自由変数として表れないことをいう。型表現中に現れる型変数は、すべて自由出現か束縛出現に分類される。 σ の型変数の出現 α が、 $\forall \alpha$ のスコープに現れるとき、すなわち

$$\sigma \equiv \dots \forall \alpha. (\dots \alpha \dots) \dots$$

というように現れるとき、この出現は束縛変数あるいは束縛出現であるという。束縛出現でないものを、自由出現、または、自由変数という。たとえば、

$$\forall \gamma ((\alpha \xrightarrow{1} \beta) \rightarrow (\alpha \xrightarrow{3} \gamma))$$

の中で、番号 1, 4 のついた α と γ は束縛出現であり、2, 3 の β と α は自由出現である。 $\forall \alpha$ のように、 $\forall$ のすぐ後ろに、現れる変数は、すべて束縛出

$$\frac{\frac{\{f : \alpha \rightarrow \alpha\} \vdash f : \alpha \rightarrow \alpha}{\{f : \alpha \rightarrow \alpha, x : \alpha\} \vdash f : \alpha \rightarrow \alpha} \quad \frac{\frac{\{f : \alpha \rightarrow \alpha\} \vdash f : \alpha \rightarrow \alpha \quad \frac{\{x : \alpha\} \vdash x : \alpha}{\{f : \alpha \rightarrow \alpha, x : \alpha\} \vdash x : \alpha}}{\{f : \alpha \rightarrow \alpha, x : \alpha\} \vdash f(x) : \alpha}}{\frac{\{f : \alpha \rightarrow \alpha, x : \alpha\} \vdash f(f(x)) : \alpha}{\{f : \alpha \rightarrow \alpha\} \vdash \lambda x. f(f(x)) : \alpha \rightarrow \alpha}} \quad \frac{\{ \} \vdash \lambda f. \lambda x. f(f(x)) : (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)}{\{ \} \vdash \lambda f. \lambda x. f(f(x)) : \forall \alpha. (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)}$$

最後の推論は ($\forall I$) である。型宣言は、空集合になっているので、自由変数についての条件は、満たされている。

PT で、 $\Gamma \vdash e : \tau$ が導出できるとき、型宣言 Γ のもとで、 e は型 τ をもつ、あるいは、 e の型は τ であるという。表現の型は、一意的に決まるわけではない、型をもたない表現もある。たとえば、

$$\frac{\{x : \sigma\} \vdash x : \sigma}{\{ \} \vdash \lambda x. x : \sigma \rightarrow \sigma}$$

だから、 $\text{Id} \lambda x. x$ は、任意の型 σ を使ってできる型 $\sigma \rightarrow \sigma$ をもつ。また、($\forall I$) を使えば、 $\forall \alpha. (\sigma \rightarrow \sigma)$ という型をもつ。また、 $(\lambda x. x(x))(\lambda x. x(x))$ には、型をつけることができない。これについては、後に説明する。

2.2 多相型の意味論

多相型の意味は、すでに直観的に説明したが、これを、定義しよう。まず、表現の意味であるが、これは、ラムダ計算の意味論でよい。ラムダ計算の意味論

を現とする。その定義は、型変数を命題変数、 $\sigma \rightarrow \tau$ を「 σ ならば τ 」、 $\forall \alpha. \sigma$ を「すべての α に対し、 σ 」と解釈して、型表現を論理式（第二階命題論理の論理式）と考えたときの命題変数の自由出現、束縛出現の定義と同じである。

また、($\forall E$) で使った記号 $\sigma[\tau/\alpha]$ は、 σ の中の α の自由出現を、すべて型表現 τ で置きかえたもの、すなわち、 τ を σ の α へ代入したものを表す。もちろん、いわゆる「変数名の衝突」がおきるときには、束縛変数を書き換える。したがって、 α -convertible な型表現は、同じものとみなす。これらは、ラムダ計算や、論理学の場合と同じだから、詳しい説明は省略する。

この推論法則を使って、実際に、 $\text{twice } \lambda f. \lambda x. f(f(x))$ が、型、 $\forall \alpha. ((\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha))$ をもつことを示そう。つまり、

$$\{ \} \vdash \lambda f. \lambda x. f(f(x)) : \forall \alpha. ((\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha))$$

を導出する。これは、次のように導出される。

の定式化は、いろいろあるが、本稿では、environment semantics と呼ばれている方法を使って定義する ([M]) 参照。

定義5 (ラムダ計算のモデル)

D を空でない集合、 $[D \rightarrow D]$ を、 D から D への関数の集合の部分集合とする。 Λ を、PT の表現の集合、 V を PT の変数の集合とする。変数に D の値を対応させる関数 $\rho : V \rightarrow D$ を環境という。環境の集合を Env とかこう。表現 e に、環境 ρ のもとで、値 $\text{value}(e, \rho)$ を与える。

$$(A) \quad \text{value} : \Lambda \times \text{Env} \rightarrow D$$

という関数と、 $[D \rightarrow D]$ の関数 f が、 D の元 $\text{code}(f)$ で「コード」できることを示す。

$$(B) \quad D \xrightarrow[\text{code}]{\text{fun}} [D \rightarrow D]$$

$$\text{fun}(\text{code}(f)) = f \quad (f \in [D \rightarrow D])$$

という関数 $\text{fun} : D \rightarrow [D \rightarrow D]$ 、 $\text{code} : [D \rightarrow D] \rightarrow D$ が与えられていて、次の条件を満たすとき、 D 、 $[D \rightarrow$

$D]$, fun, code, value を, ラムダ計算のモデルという.

1. value $(xp) = p(x)$
2. value $(e_1(e_2), p)$
 $= (\text{fun } (\text{value } (e_1, p))) (\text{value } (e_2, p))$
3. value $(\lambda x. e, p)$
 $= \text{code } (\lambda a: D. \text{value } (e, p[a/x]))$

分かりづらいだろうから, 少し説明しよう. ラムダ計算は, 関数と, 値が, 混在している体系である. つまり, 値を関数とみなすことも, 関数を値とみなすこともできなくてはいけませんが, code と fun が, これを可能にする. もちろん, すべての関数を値とみなせるわけではないので, $[D \rightarrow D]$ という, 関数の部分集合を考えるわけである. value のもつべき性質 1 が自然であろう. 性質 2 は, $e_1(e_2)$ の値は, e_1 の値 value (e_1, p) を, fun により関数とみなしたとき, e_2 の値に, その関数を適用して得られることを示している. 性質 3 は, $\lambda x. e$ という関数を値とみなしたもので, すなわち, value $(\lambda x. e, p)$ は, D の値 a に, x の値を a としたときの e の値を対応させる関数

$$(*) \quad \lambda x: D. \text{value } (e, p[a/x])$$

を, code によって値にしたものであることを示す. ただし,

$$p[a/x](y) = \begin{cases} a & x=y \\ p(y) & x \neq y \end{cases}$$

である. 条件 3 は, $(*)$ が $[D \rightarrow D]$ に属するという条件を暗黙の内に含んでいる. つまり, 条件 3 の左辺が定義されているのだから, 右辺も定義されていなければならない. したがって $(*)$ に, code を適用したものが定義されていることから, $(*)$ は, code の定義域に入っていることになる.

条件 1, 2, 3 より容易に分かるように, value (e, p) の値は, e の自由変数上で p と一致する p' による値 value (e, p') と一致するし, α -convertible な表現は, 同じ値をもつ. また, β -conversion

$$\text{value } (\lambda x. e_1(e_2), p) = \text{value } (e_1[e_2/x], p)$$

もなり立つ.

さて, ラムダ計算の意味論が定義できたので, 多相型の意味論を定義しよう. まず, 型の解釈であるが, これは, D の任意の部分集合と定義しよう. つまり, 型とは, 単に値 (データ) の集合である. σ, τ が集合ならば, $\sigma \rightarrow \tau$ の解釈は, σ から τ への関数の集合だろう. しかし, 関数は, D の元でないから型の定義に矛盾する. しかし, $[D \rightarrow D]$ の関数ならば,

code により D の元とみなせる. そこで, $\sigma \rightarrow \tau$ は, σ の元を入力すると τ の元を出力する $[D \rightarrow D]$ の関数を code によって D の元に変換したものとす. すなわち, $A, B \subseteq D$ であるとき,

$$\text{Fun } (A, B) = \{\text{code } (f) \mid f \in [D \rightarrow D] \\ \text{かつ } \forall x \in A. f(x) \in B\}$$

と定義し, $\sigma \rightarrow \tau$ は, σ, τ の解釈が, D の部分集合 A, B であるとき, $\text{Fun } (A, B)$ であるとする. $\text{Fun } (A, B)$ の元 code (f) を関数とみなすと $\text{fun } (\text{code } (f)) = f$ だから, f そのものであるが, f は A の上で定義された関数ではなく, D の上で定義された関数であることに注意して欲しい. これが多相型のアイデアを解く鍵となる. つまり, 関数 $\lambda x. e$ が, $\sigma_1 \neq \sigma_2$ という異なる入力型の型をもつ. 二つの型 $\sigma_1 \rightarrow \tau_1, \sigma_2 \rightarrow \tau_2$ をもったとしても, 実際には $\lambda x. e$ は, σ_1, σ_2 を部分集合とする D という universal な領域の上で定義された関数で, その定義域を σ_1, σ_2 に制限したとき, σ_1 から τ_1 への関数, σ_2 から τ_2 への関数とみなせるということにすぎないので矛盾は生じない.

多相型 $\forall \alpha. \sigma$ そのものの定義はごく簡単で

$$\forall \alpha. \sigma = \bigcap_{A \subseteq D} \sigma[A/\alpha]$$

である.

以上を, まとめると型表現の意味論は, 次のようになる.

定義 6 (型表現の意味論)

1. Type を D の部分集合の全体 $\text{Pow } (D)$ とする.
2. 型表現の集合を Type Exp とし, 型変数の集合から Type への関数の集合を Type Env とする. (Type Env の元を型環境とよぶ.) 型表現 σ の型環境 η における意味 $\text{type } (\sigma, \eta)$ は, 次のように定義される.

- 2.1. $\text{type } (\alpha, \eta) = \eta(\alpha)$
- 2.2. $\text{type } (\sigma_1 \rightarrow \sigma_2, \eta)$
 $= \text{Fun } (\text{type } (\sigma_1, \eta) \text{ type } (\sigma_2, \eta))$
- 2.3. $\text{type } (\forall \alpha. \sigma, \eta) = \bigcap_{A \in \text{Type}} \text{type } (\sigma, \eta[A/\alpha])$

明らかに, type は, Type Exp と Type Env の直積から, Type への関数である.

表現と型表現の意味が定義できたので, 型判定形式の意味を定義しよう.

定義 7 (型判定形式の意味論)

$\{x_1: \sigma_1, \dots, x_n: \sigma\} \vdash e: \sigma$ が恒真であるとは, $p \in \text{Env}$, $\eta \in \text{Type Env}$ で,

$$p(x_1) \in \text{type}(\sigma_1, \eta), \dots, p(x_n) \in \text{type}(\sigma_n, \eta)$$

となるものに対し,

$$\text{value}(e, p) \in \text{type}(\sigma, \eta)$$

となることである.

PT は, この意味論を満たす. すなわち, 次の定理がなりたつ.

定理 1. PT で導出可能な型判定形式は, 恒真である.

証明は, 導出についての帰納法による. (asp) が恒真であるのは明らかである. また, 推論 (add) が恒真性を保存するものも明らかだろう. ($\rightarrow I$) の仮定

$$\Gamma, x: \sigma \vdash e: \tau$$

が恒真としよう. 環境 p と型環境 η のもとで, Γ が正しく, $p(x) \in \text{type}(\sigma, \eta)$ のとき, $\text{value}(e, p) \in \text{type}(\tau, \eta)$ である. したがって,

$$f = \lambda a: D. \text{value}(e, p[a/x])$$

とすると, $f \in [D \rightarrow D]$ であり, $p[a/x](x) = a$ だから $a \in \text{type}(\sigma, \eta)$ のとき, $f(a) \in \text{type}(\tau, \eta)$ である. よって,

$$\text{value}(\lambda x. e, p) \in \text{type}(\sigma \rightarrow \tau, \eta)$$

となる. ($\rightarrow E$), ($\forall I$), ($\forall E$) の証明も, 同様なので省略する. ($\forall E$) の証明では,

$$\text{type}(\sigma[\tau/\alpha], \eta) = \text{type}(\sigma, \eta[\text{type}(\tau, \eta)/\alpha])$$

という事実を使う. これは, 代入補題と呼ばれる性質で, σ についての帰納法を使って, $\text{type}(\sigma, \eta)$ の定義から証明できる.

2.3 ML の型

PT の型は, 第二階ラムダ計算の型として知られているものであるが, この型は非常に大きな表現力をもっている. たとえば, 自然数とカリストのような帰納的型も PT の型で表現することができる. 自然数の型は, $\forall \alpha((\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha))$ と表現される. これを N とすると $N \rightarrow N$, $N \rightarrow N \rightarrow N$ など型だから, 自然数上の関数の型も PT の型となる. PT の型推論は, 大変強力で, アッカーマン関数のような複雑な計算量をもつ関数を表す表現にも型をつけられることが知られている. しかし, ML のような実用言語では, 自然数等の型は, ビルト・インな型として捉えればよく, PT の型で定義する必要はない. また, PT の型は強力すぎるため, その性質が, はっきり分かっていないところもある. たとえば, 型判定形式が導出可能か否かを

判定するアルゴリズムが存在するかどうかは, まだ分かっていない. これは, 実用言語として PT を使う上で大変な障害となる. つまり, 変数の型宣言を行って, プログラム (表現) e を, その下で書いたとしても, e が期待する型をもつことを, いちいち推論法則を使って導出しなくてはいけない. そこで, PT の型を制限して, 型判定形式の導出可能性のチェックを自動化することが考えられる. ML の型と型合成アルゴリズムは, そのようなサブセットと考えられる.

ML の型とは, PT の型の内では

$$\forall \alpha_1 \forall \alpha_2 \dots \forall \alpha_n. \tau \quad (\tau \text{ は } \forall \text{ を含まない})$$

という形をしているものである. τ のように, $\forall$ を含まない型を単純型ということにしよう. 以下では, ML の型を, $\sigma, \sigma_1, \sigma_2, \dots$ と書き, 単純型を $\tau, \tau_1, \tau_2, \dots$ とかくことにする.

たとえば, 上で定義した型 $N = \forall \alpha. ((\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha))$ は, ML の型になるが, $N \rightarrow N$, $N \rightarrow N \rightarrow N$ などは, ML の型ではない.

ML が多相型を使う理由は, 関数定義を行ったときに, その関数が多相型をもつようにし, いろいろな型の引数に適用できるようにすることにある. たとえば,

$$\text{define } Id(x) = x$$

$$\text{in } \dots Id(1) \dots Id("ABC") \dots$$

のようなプログラムが書けるようにしたい. ML は高階の型をもつので, これを let を使って

$$\text{let } Id = \lambda x. x$$

$$\text{in } \dots Id(1) \dots Id("ABC") \dots$$

のように書くことができる. $\text{let } x = e_1 \text{ in } e_2$ は, $(\lambda x. e_2)(e_1)$ と定義できるが, 上の例のように, 多相型 $\forall \alpha. (\alpha \rightarrow \alpha)$ をもつ関数 $\lambda x. x$ を let で Id に assign する場合を考えると,

$$(1) (\lambda Id. e_2)(\lambda x. x)$$

となって, $\lambda Id. e_2$ は, e_2 の型を σ とすると,

$$(\forall \alpha. (\alpha \rightarrow \alpha)) \rightarrow \sigma$$

という型をもつが, これは, ML の型ではない. しかし, let 文(1)自身は, 型 σ をもつので ML の型となる. そこで, let は, primitive なプログラム構成子であるとする. すなわち, PT の表現を拡張して, e_1, e_2 が表現で x が変数であるならば, $\text{let } x = e_1 \text{ in } e_2$ も表現であるとする. let についての型推論は, 次のようになる.

$$\text{LET } \frac{\Gamma \vdash e_1: \sigma_1 \quad \Gamma, x: \sigma_1 \vdash e_2: \sigma_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2: \sigma_2}$$

ML の型推論とは、この LET と、PT の型推論を、すべて、ML の型に制限したものである。 σ_1 か σ_2 が、 $\forall$ を含むと $\sigma_1 \rightarrow \sigma_2$ は単純型ではないので、 $(\rightarrow I)$ 、 $(\rightarrow E)$ に現れる型は、すべて単純型でなくてはならない。たとえば、ML の $(\rightarrow I)$ は

$$\frac{\Gamma, x: \tau_1 \vdash e: \tau_2}{\Gamma \vdash \lambda x. e: \tau_1 \rightarrow \tau_2} \quad (\tau_1, \tau_2 \text{ は単純型})$$

となる。

let 文 $\text{let } x = e_1 \text{ in } e_2$ を、ラムダ表現 $(\lambda x. e_2)(e_1)$ に「コンパイル」すれば、新しい推論法則 LET は、PT の推論

$$\frac{\Gamma \vdash e_1: \sigma_1 \quad \frac{\Gamma, x: \sigma_1 \vdash e_2: \sigma_2}{\Gamma \vdash \lambda x. e_2: \sigma_1 \rightarrow \sigma_2}}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2: \sigma_2}$$

と解釈できるから、ML は PT のサブセットと考えることができる。

2.4 ML の型合成

さて、ML の型理論が定義できたので、どうやって型判定形式の導出可能性をチェックできるか説明しよう。実は、ML の型については、単に型判定式の導出可能性を決定できるだけでなく、型のついていない表現から最も一般的な型表現を合成できることが知られている。そのような一般的な型表現を主型表現 (principal type scheme) という。まず、これを説明しよう。

型 $\forall \alpha (\alpha \rightarrow \alpha)$ と $\forall \alpha, ((\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha))$ を比較すると、前者の方がより一般的な型を表していることが分かる。後者の α に、単純型表現 σ を代入して得られる $(\sigma \rightarrow \sigma) \rightarrow (\sigma \rightarrow \sigma)$ は、前者の α に、 $\sigma \rightarrow \sigma$ を代入して作ることができるからである。ML の型表現 $\sigma_1 = \forall \alpha_1 \cdots \forall \alpha_m, \tau_1$ 、 $\sigma_2 = \forall \beta_1 \cdots \forall \beta_n, \tau_2$ に対し、

(*) σ_2 の $\beta_1, \dots, \beta_n$ に単純型表現を代入して得られるものは、 σ_1 の $\alpha_1, \dots, \alpha_m$ に単純型表現を代入して得られる。

という条件が成立するとき、 σ_1 は σ_2 より一般的である。あるいは、 σ_2 は σ_1 の generic instance であるといい、 $\sigma_1 \geq \sigma_2$ とかく。この一般性の概念を使うと、主型表現は次のように定義できる。

定義7 (主型表現)。型宣言 Γ のもとでの、表現 e の主型表現 σ とは、 $\Gamma \vdash e: \sigma$ が導出可能であり、任意の導出可能な $\Gamma \vdash e: \sigma'$ に対し、 $\sigma \geq \sigma'$ となるものである。

主型表現が、束縛型変数の名前の違いを無視すれば、一意にきまることは明らかだろう。ML では、型をもつ表現は、すべて主型表現をもつ。

定理2 (Milnor [ML], Dames [D]). $\Gamma \vdash e: \sigma$ が ML の型推論で導出可能ならば、 Γ のもとで、 e は主型表現をもつ。また、主型表現を計算するアルゴリズムが存在する。

この定理の証明はしないが ((D) に詳しい証明がある)、主型表現を計算するアルゴリズム W を定義しておく。

アルゴリズム W

Γ : 型宣言, e : 表現

$W(\Gamma, e) =$

(i) if e is x and $x: \forall \alpha_1 \cdots \forall \alpha_n. \tau' \in \Gamma$ then

$[\] , \tau'[\text{new}(\)/\alpha_1, \dots, \text{new}(\)/\alpha_n]$

(ii) if e is $e_1 e_2$ then

$\text{let}(S_1, \tau_1) = W(\Gamma, e_1)$ in

$\text{let}(S_2, \tau_2) = W(\Gamma S_1, e_2)$ in

$\text{let } \beta = \text{new}(\)$ in

$\text{let } U = \text{Unify}(\tau_1 S_1, \tau_2 \rightarrow \beta)$

in $(S_2 S_1 U, \beta U)$

(iii) if e is $\lambda x. e$, then

$\text{let } \beta = \text{new}(\)$ in

$\text{let}(S_1, \tau_1) = W(\Gamma_x \cup \{x: \beta\}, e_1)$

in $(S_1, \beta S_1 \rightarrow \tau_1)$

(iv) if e is $(\text{let } x = e_1 \text{ in } e_2)$ then

$\text{let}(S_1, \tau_1) = W(A, e_1)$ in

$\text{let}(S_2, \tau_2) = W(\Gamma_x S_1 \cup \{x: \overline{\tau_1 S_1}(\tau_1)\}, e_2)$

in $(S_2 S_1, \tau_2)$

W は、型宣言 Γ と表現 e を入力して、型変数への型表現の代入と単純型の対を返す。(i) の $[\]$ は、なにもしない代入、 $\text{new}(\)$ は、新しい型変数を返す関数である。 $\text{new}(\)$ は、LISP の gensym のようなもので、呼ばれるごとに新しい変数を作るとする。だから (i) では、 new は n 個の新しい型変数を作る。 $\text{Unify}(\tau_1, \tau_2)$ は、単純型表現 τ_1, τ_2 の most general unifier である。 S_1, S_2 が代入のとき、 $S_1 S_2$ は、 $\tau S_1 S_2 = (\tau S_1) S_2$ という代入を表す。 Γ_x は、 Γ から $x: \sigma$ という形の宣言を取り除いたもの、 $\overline{\tau}(\tau)$ は、 τ に現れる自由型変数で Γ の内に自由型変数として表れないものを $\alpha_1, \dots, \alpha_n$ として、 $\forall \alpha_1 \cdots \forall \alpha_n. \tau$ と定義する。

このとき、次の事実がなり立つ。

定理3 (i) $W(\Gamma, e)$ が停止して (S, τ) を返すならば、 $\Gamma \vdash e: \tau$ が導出できる。

(ii) σ に対し $\Gamma \vdash e: \sigma$ が導出できるならば、

$W(\Gamma, e)$ は値 (S, τ) をもち、ある代入 R があって

$$\Gamma = \Gamma SR \text{ かつ } (\Gamma S(\tau))R \geq \sigma$$

$\Gamma = \Gamma SR$ だから、 S と R は、型変数の名前をつけかえているにすぎない。したがって R も Γ と S から計算できる。よって $(\Gamma S(\tau))S$ も計算できて、これが主型表現となるわけである。詳細は [D] を参照して欲しい。

2.5 帰納的定義がある場合の意味論

実際の ML では当然のことながら関数の帰納的定義がゆるされる。帰納的定義は、 $(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$ という型をもつ e に対し、 $\alpha \rightarrow \alpha$ という型をもつ $\text{fix}(e)$ という関数で

$$(**) \quad e(\text{fix}(e)) = \text{fix}(e)$$

を満たすものがあると考えれば十分である。たとえば、

$$\text{def } f(x) = g(f(h(x)))$$

という定義は、

$$\text{let } f = \text{fix}(\lambda f, g(f(h(x))))$$

という定義だと考えることができる。 $(**)$ は、型を考える上では必要ないので、 fix の型のみを考えればよい。すなわち、

$$\sigma_e = \forall \alpha. ((\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)) \rightarrow (\alpha \rightarrow \alpha)$$

という型をもつ fix という定数が存在していると考え $\text{fix} : \sigma_e$ を含む型宣言のもとで、 fix を含む表現の型推論や型合成を行えば十分である。

しかし、意味論を考えるときには、定数 fix を σ_e の値として解釈する必要がある。 $(**)$ の性質を満たすような項は、ラムダ表現により $\lambda f. (\lambda x. fxx)(\lambda x. fxx)$ と定義できることが知られているが、これが σ_e の型をもつことは 2.2 の意味論では確かめられない。そこで、2.2 の意味論を次のように変更する。

1. D は、 cpo で、 $[D \rightarrow D]$ は cpo の連続関数の全体。

2. code , fun は、 cpo の連続関数。

3. Type は、 D の部分 cpo の全体。

cpo とは、complete partial order の略で、最小元をもち、任意の有向集合が、上限をもつ半順序集合のことである。

A, B が D の部分 cpo ならば、 $\text{Fun}(A, B)$ も、そうであるし、任意の部分 cpo の集合の共通部分は、部分 cpo となるので、型表現の解釈は定義できる。また、 cpo では、 $\text{fix}(f)$ は $\bigcup_{n \in \mathbb{N}} f^n(\perp)$ と定義できる

が、 $f \in \text{type}((\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha), \eta)$ ならば、

$f^n(\perp) \in \text{type}(\alpha \rightarrow \alpha, \eta)$ で、 $\text{type}(\alpha \rightarrow \alpha, \eta)$ は cpo だから、 $\bigcup_{n \in \mathbb{N}} f^n(\perp) \in \text{type}(\alpha \rightarrow \alpha, \eta)$ である。よって、 fix は型 σ_e をもつ。このようなラムダ計算のモデルとしては、スコットの D_∞ 、スコット・プロトキンの P_∞ などが知られている。

3. お わ り に

最初に原稿を引き受けたときには、モジュールの型とか、Martin-Löf の型理論についても述べるつもりでいたのだが、分かりやすく詳しい説明をしていたら、ML の型だけで終わってしまった。ML の型の意味論を詳しく述べたのは、数学的な意味論を理解することなく、ML の型を本朝に理解することは不可能だと考えたのと、友人の羅朝暉氏に、初心者にも ML の正確な意味論を教えられると大ミエを切ってしまったので、それを実証するためでもあった。しかし、ページ数の制限と著者の力不足で、思うように説明できなかったところもある。特に、帰納的定義の部分は、大変不満に思っている。[MPS]などを参照していただければ幸いである。型理論については、多相型のほかにも、数多くの面白い話題がある。文献案内 [H] を参照にいただければ幸いである。

参 考 文 献

- [R] Reynolds, J.: Three approaches to type structure, Lecture Notes in Computer Science, Vol. 185, pp. 97-138.
- [H] 林 晋: 文献案内「型理論」, コンピュータ・ソフトウェア.
- [M] Meyer, A.: What is a model of lambda calculus, Information and Control 52 (1982).
- [N] 中島玲二, 数理情報学入門, 朝倉書店.
- [M₂] Milnor, R.: A Theory of Type Polymorphism in Programming, J. of Comp. and Syst. Science 17, pp. 348-375 (1978).
- [D] Damas, L. M. M.: Type Assignment in Programming Languages, Ph. D. thesis, University of Edinburgh.
- [MPS] MacQueen, D. Plotkin, G. and Sethi, R.: An ideal model for recursive polymorphic types, Information and Control 71, pp. 95-130 (1986).

(昭和 63 年 4 月 25 日受付)