A Theory of Type Polymorphism in Programming
〜〜 概要 〜〜

児玉靖司
東京理科大学理工学部情報科学科

1 はじめに

- この研究は，実際的なものである．
- 型を強制しないようなプログラミング言語は，柔軟性があるが，幅広い意味に対して，機能する手続きを定義するよう注意しなければならない．
- このような柔軟性は，プログラミング言語に対して，本質的であるが，なぞのバグを発見するのが難しい．
- Lisp において，CDR をミスするかもしれない．
- ALGOL68 では，ALGOL60 に比べ，上のような柔軟性を排除した．ALGOL60 の方がより柔軟性があった．
- Strachey により，初期の「柔軟性」に対する議論があった．彼は，おそらく最初に多相性 (polymorphism) を唱えた人であろう．
- 彼は，パラメトリック多相性を定義した (他をアドホック多相として)．
- 例えば + に対して，整数の加算と，実数の加算を与える (複素数の加算や，ベクタの加算も考えることができる)．
- 今では，異なった型に対する識別子のこのような扱いを「オーバーローディング」(overloading) という．
- この論文では，オーバーローディングに対しては議論しない．
- この論文では，型の柔軟性を増すための一つの方法を提案する．
- 理論的な演習だけでなく，実際的な方法を証明する．
- 我々が議論する型の規則 (discipline) は，LCF のメタ言語 ML において導入されたものである．
- この論文の主なものは意味的な側面 (semantic)，文法的な側面 (syntactic) における技術的な説明を中心にしている．
- 第一に，型に対する処理は，コンパイル時に行われる．型チェッカがプログラムを受理したら，実行時に型を扱わないようにコードを生成する．
- 第二に，プログラム全てに対して，型を指定することはしない．型は，プログラムコンテキストから推論される (ML では，しかし，時々いくつかの型に対して作用する演算に対して，時々型を指定しなければならない)．
- 最近の言語 (CLU，ALPHARD，Euclid) などは，ユーザが型を定義することができる．対して，我々の提案する型チェッカは大変シンプルである．
- 第三に，例えば，一般に，多相型を割り当てられる手続きは，そのコンテキストから引数と返り値の値が一意に決まる場合に限られる．
- Gries と Gehani(その他) は，多相プログラミングを制御する場合に対して，説得力のある説明を与えている．
- Tennent は，型変数または，識別子に対して，多相型は指定する必要があることを主張した．
- 我々は「プログラムに出現する多相性は，プリミティブな演算子 (全てのプログラミング言語に存在する) の自然な産物である.」と主張する．そのような演算子は，代入，関数適用，組 (pairing, tupling) と，リスト処理である．
- 型強制 (coercion) または，オーバーローディングに対しては議論しない．我々の考えでは，これらの概念は，実行時の型の扱いであると考える．
- 第 2 節では，ML を用いたプログラム断片により，例を用いて，型の規則を概観する．この断片は，self-explanatory である．
- 残りの部分では，簡単な言語 Exp を用いてこの規則を justify する．
- 第 3 節では，*well typing* (正しい型割り当て) の概念を定義し，意味論的健全性 (Semantic Soundness Theorem)(well-typed プログラムは，意味的に，型エラーを起こさない) を証明する．
- 第 4 節では，well-type アルゴリズム $\mathcal{W}$ を提案し，文法的健全性 (Syntactic Soundness Theorem)(もし $\mathcal{W}$ が成功すれば，プログラムの well-typing を生成する) を証明する．
- さらに $\mathcal{W}$ をシミュレートするアルゴリズム $\mathcal{J}$ を提案する．

- Exp における型は，基本型の集合上での純粋な関数型の組み合わせとなる．
- Exp における多相性は，単一のプリミティブ多相演算子，関数適用など，の自然な産物である．
- 代入演算子 (副作用のある) を，文法的健全性の証明に導入することは容易であるが，意味論的健全性の理論に拡張することは難しい (第 5 節で議論する)．
- Morris の博士論文「引数として異なった型を持つかもしれない関数を，関数内での別々の位置で呼ぶ関数をプログラマが定義する，言語と型システムを設計するためには」で表現されている問題を解こうとしている．
- 問題についてのわかりやすい導入として，この論文の第 4 章を読むことを薦める．
- Morris は，型の意味論についてフォーマルには議論していないが，多相型システムを与えている．
- この研究をした後で，我々は コンビナトリロジックの項のための「主要な型スキーマ」(我々が型スキーマといっているもの) を導出する Hindley の方法に気がついた．
- Hindley は 「Robinson [4] の Unification Algorithm は，この問題を解くのに適している」ことに気がついた最初の人であろう．
- 我々の研究は，Hindley の方法を局所的な定義を持ったプログラミング言語に拡張し，この方法の意味論的な justification をすることであると認識している．

2 型規則の概要

- いくつかの簡単な例を用いて多相性の概念を説明する．ML のプログラム断片で書かれている．
- $$let\ x = e\ in\ e'$$
  $$let\ f(x_1, \cdots, x_n) = e\ in\ e'$$
  により，値 e を持つ x と，値 (抽象)$\lambda(x_1, \cdots, x_n) \cdot e$ を持つ f を与える．
- 再帰関数を使う場合は，let の代わりに $letrec$ を使う．$in\ e'$ が省略された場合は，宣言を行う．
- ML の fully determined 型 (monotypes) は，中値二項演算子 $\times$(カルテシアン積)，$+$ (和)，$\rightarrow$ (関数型)，後置単項演算子 ($list$) により基本型 (int, $bool$ など) から構築される．
- 多相型 (polytype) は，型変数 (α，β，γ) を許すことにより導入される．通常の型は ρ，σ，τ で表現する．
- 例 1: リスト上の関数のマップ
  $$letrec\ map(f, m) = if\ null(m)\ then\ nil$$
  $$else\ cons(f(hd(m)), map(f, tl(m))).$$
  を考える．
- 型は $((\alpha \rightarrow \beta) \times \alpha\, list) \rightarrow \beta\, list$ となる．ここで α，β は型変数である．
- どのようにして，map の生の宣言より型を決定することができるのだろう．
- 最初に，宣言中の自由である (として出現する) 識別子の generic 型 (後述) は，以下のようになる．
  $$null:\ \alpha\, list \rightarrow bool,$$
  $$nil:\ \alpha\, list,$$
  $$hd:\ \alpha\, list \rightarrow \alpha,$$
  $$tl:\ \alpha\, list \rightarrow \alpha\, list,$$
  $$cons:\ (\alpha \times \alpha\, list) \rightarrow \alpha\, list,$$
- 各型は，一つ以上の型変数を持つので多相型である．
- そのような識別子の全ての出現 generic 変数に対して，置換したインスタンスである型を代入することができる．
- これらの識別子の各々は，宣言の中でちょうど一回出現する．σ_{id} (識別子 id に対して型を割り当てる) と書こう．
  $$\sigma_{null} = \tau_1\, list \rightarrow bool,$$
  $$\sigma_{nil} = \tau_2\, list,$$
  $$\sigma_{hd} = \tau_3\, list \rightarrow \tau_3,$$
  $$\sigma_{tl} = \tau_4\, list \rightarrow \tau_4\, list,$$
  $$\sigma_{cons} = (\tau_5 \times \tau_5\, list) \rightarrow \tau_5\, list.$$
- 他の識別子 (map，f，m) は，二回以上出現する．
  $$\sigma_{map} = \sigma_f \times \sigma_m \rightarrow \rho_1,$$
  $$\sigma_{null} = \sigma_m \rightarrow bool,$$
  $$\sigma_{hd} = \sigma_m \rightarrow \rho_2,$$
  $$\sigma_{tl} = \sigma_m \rightarrow \rho_3,$$
  $$\sigma_f = \rho_2 \rightarrow \rho_4,$$
  $$\sigma_{map} = \rho_f \times \rho_3 \rightarrow \rho_5,$$
  $$\sigma_{cons} = \rho_4 \times \rho_5 \rightarrow \rho_6,$$
  $$\rho_1 = \sigma_{nil} = \rho_6$$
- これらの等式が，変数 ρ_i，τ_i，σ_{id} に対して解答を得られるかもしれない．Morris は，そのような等式の解答を議論した．

- 実際，この設定は Robinson の Unification Algorithm の使い方に適していた．我々の well-typing algorithm は，このアルゴリズムを参考にしている．
- Robinson の研究より，map の most general type が求める解であることが結論づけられる．
- 上の等式を満たす，他の (map に対する) 型は，必ず置換による型のインスタンスとして与えられる．
- 実際上の等式の解は，以下の通りである．

$$\rho_{map} = (\gamma \to \delta) \times \gamma\ list \to \delta\ list,$$

ここで γ，δ は，独立した型変数である．

- map の generic type である．この宣言のスコープの中でのあらゆる map の出現に対して，この型の置換によるインスタンスが割り当てられるに違いない．
- これらのインスタンスは，同一である必要はない．事実以下のような例がある．
- *tok* は基本型として，

$$tokl:\ tok\ list \quad \text{変数},$$

$$length:\ tok \to int,$$

$$sqroot:\ int \to real,$$

とすると，

map(sqroot,map(length,tokl))

とすることが可能である．

- 上の 2 つの map は別々の型を持つ．

$$((tok \to int) \times tok\ list) \to int\ list,$$

$$((int \to real) \times int\ list) \to real\ list$$

- 例えば `null` (map の定義では 2 回出現する) は，以下の型から異なったインスタンスを生成することができる．

$$\alpha\ list \to bool$$

- 異なった位置に出現するフォーマルパラメータ (引数), 再帰的に定義された識別子 (map) は，同じ型とすべきである．
- 上で定義した map は，2 つの独立して宣言された map 関数としてみなされることができる (単相!).
- Gries と Gehani は,「コンパイラがこのように区別した宣言を正確に生成すべきである」と示唆している．そして，プログラマが，それらが複製であるかを指定する必要がない，または，気づかせないように．
- 本論文では，概念的なフレームワークに注目する．インプリメンタには，別の議論が必要である．
- 例 2: Tagging
  以下の関数 tagpair を仮定する．

$$(b, c) \mapsto ((a, b), (a, c))$$

もちろん，以下のように定義できる．

$$let\ tagpair(a) = \lambda(b, c) \cdot ((a, b), (a, c)).$$

等式を設定せずに説明しよう．

- 最初に unknown type として (型変数)α，β，γ を各々 a，b，c に割り当てる．
- $((a, b), (a, c))$ は $(\alpha \times \beta) \times (\alpha \times \gamma)$ となる．λ 式は，$\beta \times \gamma \to (\alpha \times \beta) \times (\alpha \times \gamma)$ となる．よって，

$$\alpha \to (\beta \times \gamma \to (\alpha \times \beta) \times (\alpha \times \gamma))$$

となり (*) とする．

- 他の方法で tagpair を定義する．まず $(f\#g)(a, c) = (f(a), g(c))$ により，

$$\#:\ (\alpha \to \beta) \times (\gamma \to \sigma) \to ((\alpha \times \gamma) \to (\beta \times \sigma))$$

を定義する．$pair(a)(b) = (a, b)$ により

$$let\ tagpair = \lambda \cdot (let\ tag = pair(a)\ in\ tag\#tag)$$

を定義する．

- 以下のように well-typing アルゴリズムが進む．第一に，a に α を割り当て，pair の generic type を使って，$pair(a)$ を $\delta \to \alpha \times \delta$ にする．
- 次に，*tag* の局所的な generic type を使って，tag の 2 つの出現 ($tag\#tag$) に対して別々の型を割り当てる．$\beta \to \alpha_1 \times \beta$，$\gamma \to \alpha_2 \times \gamma$.
- よって $tag\#tag$ の型は以下のようになる．

$$\beta \times \gamma \to (\alpha_1 \times \beta) \times (\alpha_2 \times \gamma)$$

また *tagpair* の型は以下のようになる．

$$\alpha \to (\beta \times \gamma \to (\alpha_1 \times \beta) \times (\alpha_2 \times \gamma))$$

これを (**) とする．

- 何かが間違っている;
- 第二の型は too general である．*tag* と その generic type は，λ 束縛された変数 a に依存している．λ 束縛された変数に対して，異なった型を割り当てるべきではない．
- 事実以下のように定義すると，(*) と同じ結果を得ることができる．

$$let\ tagpair = \lambda a \cdot (pair(a)\#pair(a))$$

- *let* と *letrec* により束縛された変数の型のインスタンシエーションにのみ宣言すればよい．λ 束縛された変数や，フォーマルパラメータの束縛ではない型変数にのみインスタンシエートすればよい．
- 第二の tagpair の宣言では，局所的に宣言された tag は，generic type $\delta \to \alpha \times \delta$ は，*delta* は，generic type であるが，α はそうではない (λ 束縛されている)．
- そのため α は型を割り当てる場合 (*tag* に対して) にインタシエートされない．(**) と (*) は同じになった．
- 上の例で，明らかになったのは *let*(または *letrec*) と，λ は，異なった扱いをしなければならないことである．
- $let\ x = e\ in\ e'$ と，$(\lambda x \cdot e')e$ は意味的には同じであるが，型を正しく割り当てる場合は，前者のやり方が可能となる．
- 例として，$let\ I = \lambda x \cdot x\ in\ I(I)$ と $(\lambda I \cdot I(I))(\lambda x \cdot x)$ をあげることができる．
- λ 束縛と *let* 束縛の扱いをどうするかが我々のアプローチの核である．
- 多相型に対して，より決定的な難しさを明らかにするために，簡単な言語 Exp を導入し，項に対しての解析をする．
- 読者の中には，まだ，我々のルールが直感的に導入され，部分的にサポートされ，勝手に選択されたものであると思うかもしれない．我々は，規則はたった一つである，と主張するわけではない．しかし，意味論的に justify されたものであることを示すことができる．
- 実際，最後に紹介する $\mathcal{J}$ は大変シンプルになっている．
- Burge により考察されたデータ構造を処理するための一般的な関数への考慮は行っていない．
- well type にならない有益な式がある．

  $$Y = \lambda f \cdot (\lambda x \cdot f(x(x)))(\lambda x \cdot f(x(x)))$$

  カリーの Y コンビネータは，self-application なので ill typed となる．
- *letrec* を使うことにより Y を使わなくてもよくしている．
- 他の例:

  $$let\ F(f) = \lambda(a,b) \cdot (f(a), f(b))$$

- この定義に対して，

  $$F(reverse)(x,y)$$

  を考える．これにより異なった型 (でもよい) をもつ 2 つのリスト (リバースリスト) が生成される．
- 我々のシステムでは，このような定義は許されない (λ 束縛である a，b は同じ型でなければならない)．
- しかし，以下のようにして回避することができる．

  $$let\ reversepair = \lambda(x,y) \cdot (reverse(x), reverse(y))$$

- この例は，我々のシステムの主な制限を概観している．