記 号 処 理 39－4
（1986. 11. 10）

型推論によるLispプリコンパイラ

岸田克己　　神尾稔教
NTT電気通信研究所

Lispの変数は任意のデータをその値とすることが可能であり、複雑な構造のデータを扱うプログラムの作成が容易である。しかしこの性質のため、実行時の型検査を必要とし、しばしば処理速度の低下をもたらす。CommonLispでは、コンパイラによる最適化を考慮して、式や変数のデータ型を宣言する方法が与えられているが、実際に宣言の行われているプログラムは少ない。型が宣言されていないLispプログラムの処理効率の向上と静的な型の整合性検査を目的として、式と変数の型の推論を行い、型宣言の付加を支援するシステムの試作を進めている。本報告では、制御構造を制限したプログラムでの型の推論方法と、試作システムの動作の概要を述べる。

A Lisp Pre-Compiler Using Type Inferrence

Katsumi KISHIDA and Minoru KAMIO

NTT Electoronical Communication Laboratories
1-2356 Take Yokosuka-city Kanagawa Japan

Variables of Lisp language have various data types, which enables to develop complex data structure programs easily. As a result, runtime type check is necessary, which often decrease the computational performance. CommonLisp gives type declaration of forms and variables, while, many Lisp programs seems to have no type declarations because of its tiresome. A system which enables to insert type declarations to programs by means of inferring types of forms and variables, has been developed. This paper describes the type inference method concerning to certain Lisp programs with restricted control structure, and shows the outline of the prototype system.

１．はじめに

Lispにおいて、変数は任意の型のデータをその値とすることが可能である。しかしこの性質のため、①プログラム実行時の動的な型検査が必要となり、処理速度が低下する　②プログラム中に型の不整合（誤り）があっても、その箇所を実際に実行してみるまで、その誤りは検出されない、等の欠点が存在する。

CommonLisp〔1〕では、変数の型（後述）の宣言が可能である。処理系は型の宣言を利用して最適化処理を行うことができる。しかし、種々の事情により、実際に型宣言が行われているプログラムは少ない。

このようなプログラムの実行速度向上の一手段として、Lispプログラム中の式（特に変数）の型を推論して型宣言を付加するシステムが考えられる。また、そのようなシステムでの推論過程で型の衝突(Type Conflict)が検出されたならば、それはプログラムの誤りであると考えられ、簡単な動作試験では発見されない誤りの検出にも利用できる。

なおこのような立場に立っての、型推論システムの研究が Smalltalk〔2〕,〔3〕, Prolog〔4〕, APL〔5〕等について行われている。

本報告では、CommonLisp におけるデータ型の扱いを踏まえた上で、試作中のデータ型推論システム（Lisp プリコンパイラ）における、型推論の方法と、型詳細化機構を述べる。ここで示す型推論の方法の基本的な考え方は、Milner の方法〔6〕の拡張による Suzukiの方法〔2〕と、おおむね同じであるが、型判定による条件分岐を推論に用いる等の点に違いがある。また、現時点では代入文の処理を省略している。

２．Ｌｉｓｐにおけるデータ型とその宣言

ここでは、Lisp における型の特徴とその問題点を確認し、CommonLisp〔1〕での型宣言の方法とそのための型指定子について概説する。

Lispにおいては、データ型はデータ（Lispオブジェクト）の集合を表すとものと考えられる。データにはタグ等の補助情報が付随しており、これを用いて、データの所属する型を求めたり、あるいは、そのデータがある特定の型に含まれるかどうかを判定することができる。また、型には部分型／スーパー型の関係による階層構造があり、その頂上は Lispオブジェクトの全体集合を表すｔ型である。変数には型の区別が無く、変数は単に特定のデータを指し示しているに過ぎない。そのため変数は任意の型のデータをその値とすることが可能である。強いて変数の型を考えるならば、全ての変数の型はｔ型であると言えよう。

ところで、処理系で用意された関数の多くは、その引数としてある特定の型に属するデータを要求する。もし、この要求を満たさないデータが実引数として与えられたならば（型の不整合）、それはプログラムの誤りである。この場合そのまま処理が進行すると、無意味なあるいは間違った結果が出力されるか、処理系に混乱が生じ、最悪の場合には処理系が破壊される。そのような事態を防止するため、通常、実行時にデータ型の判定が行われる。型の不整合が検出された場合には、処理は中止され、プログラムの誤りが指摘される。また、与えられたデータの型に応じた処理の振り分けが必要な場合にも、実行時にデータ型の判定を行う必要がある。列型を扱う関数が、その一例である。列型は内部構造の全く異なるリスト型とベクトル型の和集合であるため、これを扱う上で動的な型判定は欠かせない。しかし、このような動的な型判定は、オーバヘッドとなって処理速度の低下を引き起こす。

CommonLispでは、変数の型（変数が実際に指し示しうるデータの集合）や、関数の型（実引数の所属すべき型と、戻り値の型）、およびプログラム中の式の型等、様々な付加情報を宣言する手段が定められている。（例１）　このとき、データ型は型指定子(type specifier)を用いて表現する。型指定子自身もシンボル型あるいはリスト型のデータである。シンボル型の型指定子は、処理系によってあらかじめ用意された標準型指定子シンボルであるか、ユーザが定義した型指定子シンボル、あるいは構造型の名前である。特に注意を要する標準型指定子シンボルとして、t,

nil がある。t は、全ての データを含む型を表し、nil は、データを一つも含まない型を表す。また、リストである型指定子は、型指定子シンボルで表される型について、数値の範囲の限定、配列の次元の指定、要素型の指定等の特殊化を行う。あるいは、型の積集合、和集合、補集合等、より複雑な型を構成するのに用いられる。

```
(declare (integer x))  ≡  (declare (type integer x))
    変数 x の値は常に整数型のデータである
(declare (function char (string integer) string-char))  ≡
(declare (ftype (function (string integer) string-char) char))
    関数 char は、第１の引数として文字列型、第２の引数として
    整数型のデータを受け取り、文字列文字型のデータを返す。
(the list Form)
    式 Form の値は、常にリスト型である。
```

例１　型の宣言

プログラムに宣言を付加して、プログラマが自分の意図を明確に表現することは、プログラムの文書としての価値を高めるだけでなく、コンパイラによる最適化処理の可能性を高める。（例２）　しかし、Pascal や Ada のような、宣言が必須である言語と異なり、Lispにおいては宣言を付加しなくとも、プログラムの実行が可能である。修正、実行の繰り返しである対話的プログラミングにおいて、宣言の必要が無いことは、この繰り返しのサイクルタイムを縮めプログラム開発の省力化に役立つ。

しかし、完成段階にあるプログラムについては、その処理速度が重要視される。同時に、動作条件の記述や宣言の付加によって、プログラムの文書としての価値を高めることも重要である。

そこで、プログラムを解析して型を推論することによって、型宣言の付加を支援するシステム（プリコンパイラ）の有用性が考えられる。また、このシステムによって、静的な型の整合性検査が可能となる。

以下に示す関数 tarai の、型宣言による最適化効果、

```
(defun tarai (x y z)
   (declare (fixnum x y z))
   (if (> x y)
       (tarai (tarai (the fixnum (1- x)) y z)
              (tarai (the fixnum (1- y)) z x)
              (tarai (the fixnum (1- z)) x y) ))
       y ))
```

VAX-LISP(VAX-8600) において、宣言がない tarai 関数に対して、
declare による型宣言を付加した場合に、実行速度は　約１．７倍
さらに、the による宣言を付加した場合、実行速度は　約２．５倍

例２　型の宣言による最適化

３ ：型の推論

先に述べてきた目的を達成するための、型推論についてその方法を以下に示す。

３．１　推論の手がかり

プログラム中に現れる式の型の推論を行うための手がかりとして、(1)プログラマによる型宣言、(2)関数の呼出、(3)データ型の述語による分岐の３項目を用いる。以下に、その例を示す。

(1)プログラマによる型宣言

(... (declare (type character c)) P)

⇨ 変数 c の型は、その有効範囲Pの中で常に文字型

(the string P)

⇨ 式Pは、必ず文字列型の値を返す

(2)関数の呼出

・結果の型

(list y z)

⇨ この式は、必ずリスト型の値を返す

・実引数の型の制限

(gcd i j)

⇨ 変数 i,j の型は、整数型

(3)型の述語による分岐

(if (symbolp s) P1 P2)

⇨ 変数 s の型は、P1の中でシンボル型、P2の中ではそれ以外の型

３．２　データ型整合の規則

プログラムの構造を、(1)型の宣言、(2)関数の呼出、(3)ifによる分岐に限定して、データ型が整合しているプログラムについての規則を与える。

なおここでは、データ型を Lispデータオブジェクトの集合として扱い、型について、積(∩)、和(∪)等の集合演算と、同値(≡)、包含(⊂)等の関係の存在を仮定する。

また、小文字は Lispのシンボルを表す。大文字は超変数を表すものとし、その意味を次のように定める。

P	式
V	変数（関数の仮引数を含む）
F	関数
L	リテラル
T	データ型

τ〔P,Th〕 は、式Pの値がTh型に属するという制約のもとで、式Pを評価して得られるデータの型を表す。（注　τ〔P,Th〕⊂Th である）

なお、論理記号として、積(∧)、和(∨)と含意(⊃)を用いる。

(1)　型の宣言

式 (... (declare (type T V)) P) をPgとする。

このとき、

$$\tau[Pg,Th] \equiv \tau[P,Th] \wedge \tau[V,T] \subset T \qquad (*)$$

である。

式 (the T P) をPgとする。

このとき、

τ〔Pg,Th〕≡ τ〔P,Th∩T〕
である。

(2) 関数の呼出
関数Fの型を
(or (function (T11 ... T1n) T1)
...　　...
(function (Tm1 ... Tmn) Tm)) とし、
式 (F P1 ... Pn) をPgとする。
このとき、
∃i((1≦i≦m) ∧
Ti⊂Th ∧
τ〔Pg,Th〕≡Ti ∧
∀j((1≦j≦n) ⊃ (τ〔Pj,Tij〕⊂Tij)) (*)
)
である。
特殊な場合として、関数Fの型が
(function (T1 ... Tn) T) であるとき、
すなわち結果の型が、引数の型に依存しない場合には、
τ〔Pg,Th〕≡T ∧
∀j((1≦j≦n) ⊃
(τ〔Pj,Tj〕⊂Tj) (*)
)
である。

(3) ifによる分岐
式 (if Pc Pt Pe) をPgとする。
このとき、
τ〔Pg,Th〕≡ τ〔Pt,Th〕∪ τ〔Pe,Th〕 ∧
τ〔Pc,t〕⊂t (*)
である。
ただし、Pcの形が (typep 'Tc Pf) であるならば、
このとき、
(τ〔Pg,Th〕≡ τ〔Pt,Th〕 ∧
τ〔Pc,t〕⊂t ∧ (*)
τ〔Pf,Tc〕⊂Tc) ∨
(τ〔Pg,Th〕≡ τ〔Pe,Th〕) ∧
τ〔Pc,t〕⊂t ∧ (*)
τ〔Pf,(not Tc)〕⊂(not Tc))
である。

(4) 自己評価形式、リテラルに対する τ の解釈
式Pg が自己評価形式であれば、
τ〔Pg,Th〕⊂Tr は
Pg∈(Th∩Tr) であるとき、そしてそのときに限って成り立つ。
式が quote によるリテラル、'L であるとき
τ〔'L,Th〕⊂Tr は
L∈(Th∩Tr) であるとき、そしてそのときに限って成り立つ。

3．3　推論の進め方

(defun F (V1 ... Vn) P) によって、関数Fが定義されているとする。
このとき、関数Fの型（引数と結果の型）を次のようにしてもとめる。

τ〔P，t〕に、(1),(2),(3),(4)を繰り返し適用する。

この過程において、(1),(2),(3)によって成立を示された（正確には成立を仮定された）論理式の中で、τ〔P',T〕⊂T の形をした素論理式（(*)がついている）に、(1),(2),(3)が適用可能であるならば、それらを繰り返し適用して、成立すべき論理式を得る。

得られた論理式を、もとの素論理式 τ〔P',T〕⊂T に、論理積によって追加する。ただし、(3)によって特定の部分式 Ps の中に限って τ〔Pf,Tc〕の成立が示された場合には、Ps についての (1),(2),(3) の適用過程で、τ〔Pf,Tc〕から導かれる論理式の追加を行わない。

以上の操作により、その成立を仮定される論理式が得られる。
この中で、(2)によって得られる存在記号については、Ti⊂Th について i を具体化し、論理和の形に変形する。
また、素論理式 τ(V,Th)⊂Tr を、τ(V,t)⊂(Th∩Tr) で置き換え、加法標準系に変形する。さらに、次の置き換えを繰り返し、整理して、Φ(P) とする。

τ(V,t)⊂T1∧τ(V,t)⊂T2 を次の様に置き換える。
(T1∩T2)≡nil でないとき　　τ(V,t)⊂(T1∩T2)
(T1∩T2)≡nil のとき　　false　（型不整合）

Φ(P)の一般形は次のようになる。

τ(P,t)≡T1∧(τ(V1,t)⊂T11)∧ ... ∧(τ(Vn,t)⊂T1n) ∨
... ...
τ(P,t)≡Tm∧(τ(V1,t)⊂Tm1)∧ ... ∧(τ(Vn,t)⊂Tmn)

このとき、関数Fの型は

```
(or (function (T11 ... T1n) T1)
    ...        ...
    (function (Tm1 ... Tmn) Tm) )
```

となる。
また、Φ(P) を求める過程で、プログラム中の式の型が求まる。
なお、Φ(P) が成立しないときは、Pの中でデータ型の不整合が発生している。

例として、絶対値を求める関数absについて、上の手法を適用する。

```
(defun abs (x)
  (if (> x 0) x (- x)) )
```

ただし単純化のため、

```
関数 > の型を (function (number number) t)
関数 - の型を (or (function (integer) integer)
                 (function (float) float)
                 (function (number) number) )
```

と定める。
Pを (if (> x 0) x (- x)) として、Φ(P)を求める。
Pについて(3)を適用

τ〔P，t〕≡τ〔x，t〕∪τ〔(- x)，t〕　　①
τ〔(> x 0)，t〕⊂t　　②

②と関数 > の型から

τ〔(> x 0)，t〕≡t
τ〔x,number〕⊂number　　③

τ〔0,number〕⊂number ④

①と関数 - の型から

(τ〔(- x), t 〕≡integer ∧ τ〔x,integer〕⊂integer)) ∨

(τ〔(- x), t 〕≡float ∧ τ〔x,float〕⊂float)) ∨

(τ〔(- x), t 〕≡number ∧ τ〔x,number〕⊂number)) ⑤

④は (4) により真である。①③⑤から、Φ(P)は次の様になる。

(τ〔P, t 〕≡integer ∧ τ〔x, t 〕⊂integer)) ∨

(τ〔P, t 〕≡float ∧ τ〔x, t 〕⊂float)) ∨

(τ〔P, t 〕≡number ∧ τ〔x, t 〕⊂number))

こうして、関数 abs の型が次のように求まる。

```
(or (function (integer) integer)
    (function (float) float)
    (function (number) number) )
```

この手法をそのまま実現すると、多量の論理式についてそれを満たす解を求める必要が出てくる。試作中のシステムでは、最初に各変数の型を ι とした上で、変数が所属しなければならない型との積を次々に求めていく。(2),(3)で現れる存在記号と論理和については、場合分けによる処理を行う。また、再帰関数の場合には、関数の結果の型を nil とした上で、実際にその関数の結果の型を求める。ただし、この場合には、型の整合性がうまく検査できない場合が出てくるため、関数の結果型が求まった段階で、もう一度整合性の検査を行う必要がある。

4．対話による型の詳細化

4章で述べた方法で関数の引数の型が得られるが、引数として得られた型の部分型に属するデータだけが、実際にこの関数に実引数として与えられるという場合が考えられる。そのような場合での最適化効果をあげるために、プログラマとの対話により引数の型を絞り込みを行う（型の詳細化）機構を導入する。

例えば、3章の tarai 関数の型が次のように得られたとする。

```
(function (integer integer integer) integer)
```

しかし、通常 tarai 関数に与えられるのは、値が20以下の正の fixnum 型データに限られるであろう。すなわち、実質的には tarai 関数の型は、

```
(function (fixnum fixnum fixnum) fixnum)
```

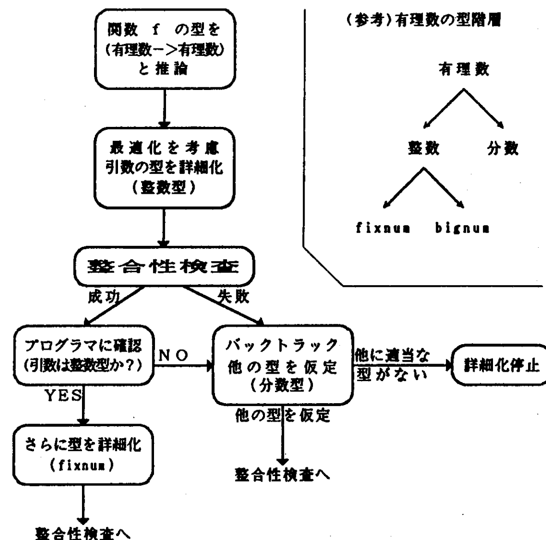
である。

型の詳細化機構の動作は次のようになる。（図3）

(1)推論により得られた引数の型の部分型の1つを、新たに引数の型として仮定する。
(2)仮定された引数の型について、整合性の検査を行う。
(3)関数の結果の型を求め直す。
(4)引数の型として別の部分型を仮定して、(2)の整合性検査から繰り返す。
(5)整合性検査で仮定の誤りが判明した場合、別の部分型を仮定し、(2)の整合性検査からやり直す。

なお、仮定により得られた結果を型宣言としてプログラムへ付加する場合、この仮定についてのプログラマの確認が必要となろう。

またある条件下では、プログラマの確認を得た上で、整合性の検査基準を緩和する必要がある。例えば、x が fixnum 型の場合、式 (1- x) の型は整数型である。しかしただ1つの場合を除いて、この式は fixnum 型のデータを返す。ここで、(1- x) の型が fixnum 型でなければならないとき、プログラマが (1- x) の型を fixnum 型であると認めた場合には、型は整合しているものとして扱う必要がある。



5. おわりに

プログラム中の式(特に変数)のデータ型の推論により、型宣言の付加を支援するシステムについて、型推論の方法を述べた。また、最適化効果を考慮して、対話による型の詳細化機構を導入した。現在、このシステムの試作が進行しており、その推論機構の基本動作が確認されている。

今後の課題として、推論の対象とするプログラムへの制限の緩和が挙げられる。特に次の項目への適用を検討したい。

(1)代入文
(2)リストの要素型

また、プログラム作成支援系としての立場から、次の項目の充実を図りたい。

(3)型詳細化におけるプログラマとの対話方法
(4)型不整合検出時の適切なアドバイス

本研究を進めるにあたり、御指導を頂いている、酒井室長、大野主幹研究員、竹内主幹研究員、ならびに御助言、御協力を頂いたELISグループの各位に感謝致します。

参考文献

[1] Guy.L.Steel Jr.: "COMMON LISP The Lanuage", Digital Press, 1985
[2] N.Suzuki: "Inferring Types in Smalltalk",
Proc. 8th ACM Symp. on Princes of Programming Languages, 1981
[3] A.H.Borning and D.H.H.Ingals:
"A Type declaration and Inference System for Smalltalk",
Proc. 9th ACM Symp. on Princes of Programming Languages, 1982
[4] P.Nishra: "Towards a theory of types in Prolog",
Int. Symp. on Logic Programming, IEEE, 1984
[5] J.D.Sybalsky: "An APL Compiler for tc Production Environment",
APL 80.Int. Conf. on APL, 1984
[6] R.Milner: "A theory of type polymorphism in programming", 1978
Journal of Computer and System Sciences. 17, 1978