型条件を緩和した型推論アルゴリズムを適用するトランスレータの試作

An Implementation of a Program Translator using an Extention of Type Inference Algorithm

児玉 靖司 †

Yasushi KODAMA

† 南山大学数理情報学部情報通信学科

yas@nanzan-u.ac.jp

Dept. of Information and Telecommunication Engineering, Nanzan University

コンパイラの最適化処理におけるコード移動の一手法である巻上げを用い，型条件を緩和した型推論アルゴリズムを適用するトランスレータを試作した．命令型言語のための型推論アルゴリズムは Tofte により提案され Standard ML の型推論システムとして実現されている．しかし，このアルゴリズムは，入力した原始プログラムが実行時エラーとならなくても，型エラーとなる場合が存在し型条件が厳しい．本稿で用いる手法は，値グラフを用い変数の参照点で単一化される型を推論した後，定義点の型を推論し，結果として，値グラフ上の ϕ 節まわりに注目して巻上げを行い上記の型条件を緩和する手法である．本稿では，Standard ML プログラムを原始プログラムとして入力し，型条件を緩和した Standard ML プログラムへのトランスレータを試作し，考察を行う．

1 はじめに

型宣言のないプログラムから，式の意味を解析し型を推論することができる．関数型言語に対しては，Hindley/Milner 型推論アルゴリズム $\mathcal{W}$ [6] が最初に提案され，さらなる改良，考察がされている．参照型を含んだ命令型言語に対しては，[3] で最初に提案され，その後 [9][10][8] により正確に推論するアルゴリズムが提案されている．現在では Tofte の型推論アルゴリズム [4] が知られており，Standard ML (以下，SML という) の型推論システムとして実現されている．関数型言語のための型推論アルゴリズムは，型システムに対して健全性，完全性が成立している．しかし，命令型言語のための型推論アルゴリズムは，型システムに対して健全性は成立しているが完全性は成立しない．つまり，原始プログラムを入力として型推論アルゴリズムを適用した際に,「実行時エラー」とならない原始プログラムに対し，型推論アルゴリズムが「型エラー」となる場合があり **型条件が厳しい**．我々は，コンパイラの最適化処理におけるコード移動の一手法である巻上げを用いて，型条件を緩和する手法を考案した．本手法は，1. 巻上げ候補の発見，2. 巻上げ処理 (コード移動) から成る．

以下，2 節では，この研究を提案するに至った背景を SML の型推論アルゴリズムを用いて説明し，3 節で，本稿で試作したトランスレータを説明する．4 節で実行例と議論を行い，最後に 5 節で今後の課題を述べる．

2 背景

本節では，標準的な型推論システムをもつプログラミング言語 SML を使って，副作用をもつ変数の記述を説明する．

型付きラムダ計算では，型付けできなかったラムダ式 $\lambda x.xx$ を型付けするために，Hindley/Milner 型推論アルゴリズム [3][6] は，新しい構文として `let` 式を導入した．簡約意味として (`fn x=>`式$_1$) 式$_2$ と同じ構文 `let val x=`式$_2$ `in` 式$_1$ `end` を導入した．ラムダ式 $\lambda x.xx$ は以下のように書くことにより型付け可能となる．

```
let val f=fn y=>y in f f end
```

この式が型付け可能なのは，型推論方法が異なるためである．`let` 式では，式 `fn y=>y` より推論された $t \to t$ 型を**一般化** (generalization) し，変数 `f` の型を $\forall t.t \to t$ とする．この $\forall t$ の意味は，変数 `f` の参照点で新たな型変数が生成されることを示す．`in` と

end の間の f f の左の f のために $t_1 \to t_1$ 型 (t_1 は新しく生成する型変数) が **実体化** (instantiation) され，右の f のために $t_2 \to t_2$ 型が実体化される．結果として式 f f は，$t_2 \to t_2$ つまり $t \to t$ 型と型推論される．

型の一般化/実体化を追加した let 式は，プログラム中の各変数に対して「参照の透過性」[1]が成立している場合には有効である．例えば，以下のプログラム，

```
1: let val l=[] in
2:    [1,2,3]@l;
3:    ["a","b"]@l
4: end
```

の 1 行目で，変数 l は，$\forall t.t\ list$ 型と推論され，2 行目の変数 l は $t_1\ list$ 型と実体化され $int\ list$ 型に単一化される．3 行目の変数 l は，2 行目とは独立に $t_2\ list$ 型と実体化され $string\ list$ 型に単一化される．このように，let 式で宣言された変数 l に一旦 [] を代入した後，変数のスコープ中では，常に変数 l が値 [] を束縛することが保証されているために (参照の透過性)，変数の参照点で $t\ list$ 型の型変数 t に，どのような型を単一化してもよい．

しかし，参照の透過性が一般に成立しない副作用がある場合には，この一般化によって，型推論アルゴリズムは，型システムに対して「健全」ではなくなる[2]．そこで，SML [4][5] および SML/NJ[11] では，let 式での変数の代入に対して，**参照型** (referential type) つまり，**弱い型** (weak type[3]) の型変数に対して，一般化できる式を，識別子，リテラル，ラムダ抽象の**文法的な値** (syntactic value)[9][10] または，**展開不可能式** (non-expansive expression)[7] に限定することによりこの問題を解決している．まとめると，現在よく知られている命令型言語に型推論システムを導入するアプローチは，以下のようになる．

1. 参照型の型変数に対しては，弱い型とする．
2. let 式において，弱い型を一般化するのは文法的な値に限定する．

しかし，この型推論アルゴリズムの制約は厳し過ぎる．以下の例を考える．

[1]純粋な関数型言語では，各変数に対して副作用が許されないのでどの場所で参照しても同じ値を示す．

[2]型推論アルゴリズムが「型エラー」とならなくても，入力した原始プログラムが「実行時エラー」となる場合がある．

[3][7] では imperative type という．

```
1: let val lg=ref 0; val d=true in
2:   let val l=ref [] in
3:      if d then l:=[1,2,3]
4:           else l:=["a","b"];
5:      lg:=length (!l)
6: end end
```

このプログラムは，3，4 行目で $(t\ list)ref$ 型の変数 l に対して，if 式の条件に応じて $int\ list$ が代入される場合 (then 部) と，$string\ list$ が代入される場合 (else 部) がある．しかし 5 行目で length 関数より，変数 l の値であるリストの長さを求めるため $string\ list$ に対しても，$int\ list$ に対しても適用可能である．つまり,「実行時エラー」は生じない．しかし，従来の型推論アルゴリズムでは「型エラー」となる．2 行目で，変数 l の型は参照型であるため，一般化されず $t\ list$ となり，3 行目の then 部と，4 行目の else 部の同一の型変数 t ($t\ list$ 型の t) に対して単一化が行われるため「型エラー」となる．仮に 2 行目で 変数 l の型推論の際に一般化されれば，この場合は,「型エラー」とはならないが，一般には参照の透過性が成立しないため，一般化してはならない．この問題を解決するために，参照型を含む変数 l を削除することができる点に注目する．事実，以下のように 5 行目の length 関数への適用を 3 行目 (then 部) と，4 行目 (else 部) に巻上げ，変数 l を削除することができる．よって，参照型に関する型条件を緩和することができる．

```
1: let val lg=ref 0; val d=true in
2:      if d then lg:=length([1,2,3])
3:           else lg:=length(["a","b"])
4: end
```

本稿では，以上の巻上げ (コード移動) を利用し，参照型を含んだ変数の削除により型推論アルゴリズムの型条件緩和を行う手法を適用したトランスレータを試作した．次節でより詳しく説明する．

3 考案した手法とトランスレータの実現

巻上げを利用し以下の 2 点に分けて解析する．1. 巻上げ候補の発見，2. 巻上げ処理，である．1. 巻上げ候補を解析するためには参照型を含む変数に関して，参照点で単一化される型を解析する必要がある．我々の手法は，変数の定義-参照関係を明らかにする値グラフ上での型推論を行う．各変数の定義点での

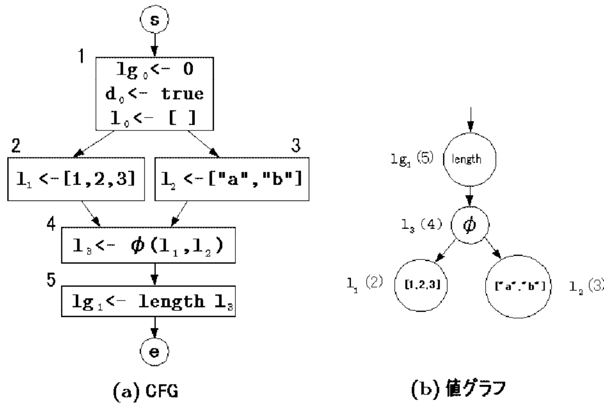

図 1: CFG

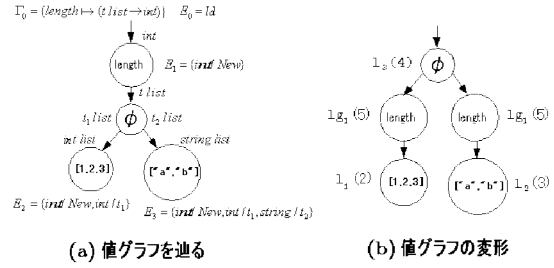

図 2: 値グラフ上の処理

型と，参照点で単一化される型との関係を調べることにより巻上げ候補を発見することができる．2. では，従来の巻上げ処理を用いる．さらに，冗長な変数の削除を行う．

3.1 巻上げ候補の発見

巻上げ候補を発見するために，値グラフ上での型推論を行う．**値グラフ** (value graph，以下 VG という) は，プログラム中の変数の定義-参照を辺で結んだグラフ表現で Alpern, Wegman, Zadeck らによって提案された [1]．最初に，与えられた原始プログラムを以下の集合を使って定義する．

- Var: 変数の集合
- C: 定数の集合
- OP: 演算子の集合

また，原始プログラムに対応する**制御フローグラフ** (control flow graph，以下 CFG という) が作成されているものとする．CFG は，基本ブロックからなる節の集合 N，$E \subset N \times N$ の辺の集合，および特別な節である開始節 s と終了節 e からなる四つ組 $(N, E, \mathrm{s}, \mathrm{e})$ として表される．本稿では，CFG を表す場合に各節が SSA 形式 (静的単一代入形式) [2] に変換されているとする．2 節で示した例題の CFG を示すと図 1(a) となる．VG は，節の集合 V と節から節への辺の集合 A からなる組 (V, A) である．各節は $OP \cup C \cup \Phi$ のラベルを持つ．図 1(a) で示したプログラムの値グラフを図 1(b) に示す (図 1(b) では，各節に対応する変数名 $v \in Var$ を示した)．従来の型推論アルゴリズムでは，開始節 s から終了節 e に向かって推論を行うため各変数の定義点で型を推論する必要があった．一方，本手法では VG 上を辿り型推論することにより，変数の参照点から定義点にさかのぼって型推論する．VG のルート節は，変数の参照関係により一般に複数存在し，各ルート節から深さ優先でグラフを辿るとする．

型推論の方法を説明するために準備をする．本稿で扱う型 τ を以下とする．

$$\tau = \tau^{\mathrm{b}} | t | \tau \to \tau | \tau\ list$$

τ^{b} は基本型 b，t は型変数を示し，$\tau \to \tau$ は関数型を示す．$\tau\ list$ は τ 型の値を要素にもつリスト型とする[4]．型環境 Γ は，識別子 x_{i} から 型 τ_{i} への関数とし，$\Gamma(x_{\mathrm{i}}) = \tau_{\mathrm{i}} (0 \le i \le n)$ と書く．Γ は，定義域が有限の関数であるため $\Gamma = \{x_0 \mapsto \tau_0, \cdots, x_{\mathrm{n}} \mapsto \tau_{\mathrm{n}}\}$ と要素を列挙して書くこともできる．置換 S は $\{\tau_1^0/\tau_1, \cdots, \tau_{\mathrm{n}}^0/\tau_{\mathrm{n}}\}$ と書く．単一化関数は，[12] で提案された一階の単一化関数 $\mathcal{U}$ である．$\mathcal{U}(\tau_1, \tau_2)$ が生ずる置換 U は τ_1，τ_2 の変数のみを含む．$\mathcal{U}$ が生ずる置換 U を mgu(most general unifier) という．本稿では，$\mathcal{U}$ が失敗したとすると，例外 $Error$ が発生し，型推論に失敗した (型エラー) と定義する．以下，単一化関数 $\mathcal{U}$ を $Unify$ と書く．$Unify$ が成功し生成した置換は E と表し広域的に定義する．よって $E^0 = Unify(\tau_1, \tau_2) \cup E$ のように書く．

巻上げ候補を発見するためには，値グラフ上で各変数に対して，深さ優先で型推論アルゴリズムを適用すればよい．図 1(b) で示した VG を辿って型推論した例を図 2(a) に示す．

VG を辿り ϕ 節を処理する場合，$\tau^0 = Gen(\tau)$ の τ^0 に少なくとも一つの $\forall t$ が出現すると，各後続節に別々の型変数として渡される．図 2(a) では，ϕ 節

[4] この τ の定義は，本稿で説明のために，最低限必要な型を定義した．一般には，$(\tau * \cdots * \tau) Tconst$ ($Tconst$ は型構成子) のように定義すればよい．

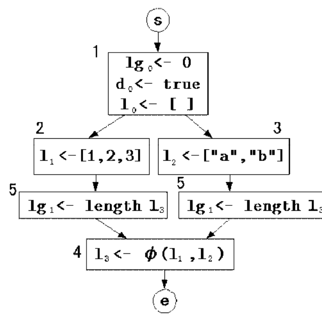

図 3: 巻上げ処理

に渡される型 t *list* に型変数 t が含まれるので，各後続節に t_1 *list*，t_2 *list* と別々の型変数として渡される．この ϕ 節の先行節は，次節で説明するように巻上げ候補であり，図 2(b) のように変形することができる．よって，一般には，以下のように定義する．

型推論の際に，VG 上の ϕ 節に渡される型を調べ，型変数を含む場合，ϕ 節の先行節を巻上げ候補とする．

図 2 では，巻上げ候補は *op* 節である length 節となる．

3.2 巻上げ処理

図 2(a) で発見された巻上げ候補に対して，CFG 上の節 5 を，節 4 (ϕ 節) の先行節に巻き上げることができる (図 3)．一般には，VG 上の ϕ 節の先行節に対応する CFG 上の節を複製し，VG 上の ϕ 節の後続節に対応する CFG 上の先行節に移動することを示している．さらに，冗長な変数 (図 3 では変数 l) を削除する．以上より，従来の型推論アルゴリズムで問題となった変数を削除し，「実行時エラー」がない場合でも「型エラー」となる部分を「型エラー」とならなくすることができる．

4 実行例と議論

実現したトランスレータの実行例は以下のようになる．./hoist が実行プログラムである．

```
prompt% cat -n test.hoi
1 let val l=ref [] in
2  let val c=ref 0 in
3   let val b=false in
4    if b then l:=10::[] else l:="a"::[];
5     c:=(length l)
6     end
7   end
8 end
prompt% ./hoist hoist.hoi > hoist.gen
prompt% cat -n hoist.gen
 1 let val l=ref nil in
 2   let val c=ref 0 in
 3     let val b=false in
 4       if (b) then (
 5         c:=length(10:nil)
 6       ) else (
 7         c:=length("a"::nil)
 8       )
 9     end
10   end
11 end
```

Standard ML のプログラム test.hoi は，従来の型推論アルゴリズムでは「型エラー」となる．我々のプログラム ./hoist で変換後のプログラム test.gen は，巻上げを行っているため「型エラー」とならない．

Tofte の命令型言語のための型推論アルゴリズムでは，参照型を含む変数に対して一般化できる場合が限られていた．しかし，変数の参照点において単一化する型に型変数が含まれる場合，その型変数はあらゆる型と単一化可能である (図 2(a) の t *list* の t)．よって，変数の定義点における t に対応する型がどのような型でもよい (図 2(a) の *string list*，*int list* の *string* と *int*)．VG を用いることにより，変数の参照関係が ϕ 節を通して関係づけられ，VG 上で ϕ 節を上げる (CFG 上では，ϕ 節をまたいで巻き上げる) こと (図 2(b)) により，あいまい性を削除することができる．さらに，従来法により巻上げ処理を行い，冗長な変数を削除することができる．よって，従来の型推論アルゴリズム (以下，Tofte のアルゴリズムという) でも「型エラー」なく推論することができる．

本稿では考案した型推論アルゴリズムを用いたトランスレータを試作した．SML プログラムを原始プログラムとし，巻上げ候補を発見し，巻上げ処理を行い，再び SML プログラムに変換する．変換後は，型条件が緩和されているので Tofte のアルゴリズムで型推論を行うことができる．変換後のプログラムの実行効率の向上は考えていない．本手法では，型推論アルゴリズムの型条件緩和のために巻上げを行

い，原始プログラムでの型推論のみを問題としているためである．巻上げ前の原始プログラムで「実行時エラー」がないということはわかれば，原始プログラムを実行すればよい．

本手法は，1. 巻上げ候補の解析，2. 巻上げ処理に分けることができる．1. では，新たに VG 上を深さ優先で辿る手法を考察した．しかし，このアルゴリズムは，プログラム全体の式を型推論する目的ではなく，巻上げ候補を発見する目的であるため，完全なアルゴリズムである必要はないと考える．さらに，巻上げによりプログラムの意味が変化しないことは一般に知られており，かつ，変換によって型付けに関する問題があった場合でも，本手法では，再度，SML のプログラムに変換し，従来の SML の処理系上で型推論を行うため，新たに問題が出現することはない．

5 今後の課題

本稿で考案した型推論アルゴリズムは，まだ，発展途上であり，今後さらに改良を加え形式的に定義する必要がある．現在のところ VG にループがある場合など，更なる考察が必要であると考えている．

謝辞

本研究は，2003 年度南山大学パッヘ研究奨励金 I-A-2(特定研究助成) より研究助成を受けて行っています．本稿で考案した型推論アルゴリズムに対して，初期段階から有益なご意見をいただいている中央大学理工学部教授土居範久先生，東京理科大学理工学部助手滝本宗宏先生に感謝いたします．

参考文献

[1] Alpern, B., Wegman, M.N. and Zadeck, F.K.: Detecting Equality of Variables in Programs, POPL, ACM, 1988, pp.1-11.

[2] Cytron, R., Ferrante, J., Rosen, B.K. and Wegman, W.W.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph, TOPLAS, Vol.13, No.4, 1991, pp.451–490.

[3] Damas, L. and Milner, R.: Principal Type-Scheme for Functional Programs, In Proc. of the 9th Annual ACM Symposium on Princeples of Programming Languages(POPL), ACM, 1982, pp.207–212.

[4] Milner, R., Tofte, M. and Harpper, R.: The Definition of Standard ML, MIT Press, Cambridge, Massachusetts, 1990.

[5] Milner, R., Tofte, M.: Commentary on Standard ML, MIT Press, Cambridge, Massachusetts, 1991.

[6] Milner, R.: A Theory of Type Polymorphism in Programming, Journal of Computer System Science, 17, 1978, pp.348-375.

[7] Tofte, M.: Type Inference for Polymorphic References, Information and Computation, No. 89, 1990, pp.1-34.

[8] Smith, G. and Volpano, D.: Polymorphic Typing of Variables and References, TOPLAS, Vol. 18, No. 3, 1996, pp.254–267.

[9] Wright, A.K.: Polymorphism for Imperative Languages without Imperative Types, Tech. Rep. TR93-200, Rice University, 1993.

[10] Wright, A.K.: A Syntactic Appoach To Type Soundness, Tech. Rep. TR91-160, Rice University, 1992.

[11] Standard ML of New Jersey Release Notes, AT&T Bell Laboratories, 1997.

[12] Robinson, J. A.: A Machine-Oriented Logic based on the Resolution Principle, Journal of ACM, 12, 1965, pp.23-41.