型に着目した定式化と実装 (部分評価およびプログラミング支援)

理学専攻 情報科学コース　対馬かなえ　(指導教員：浅井健一)

1　はじめに

プログラミング言語において、型は効率化やランタイムエラーの削減など、多くのことに利用されている。本研究では型に着目して、2 つのことに関する定式化および実装を行った。

1 つ目は継続を含むプログラムの部分評価 (強力な最適化) である。通常の部分評価ではプログラムの構文に従って最適化を行うため、最適化に時間がかかる。そこで、本研究では型を利用する部分評価 [2] を用いることで、プログラム実行と同じ速度で最適化が出来るようにした。

2 つ目は型に関するプログラミング支援である。OCaml などの静的型付け言語では、ランタイムエラーを減らすために型推論を行っている。現在の OCaml などのシステムでは、正しく型付け出来なかったとき、型の衝突に関するエラーメッセージのみが返される。これではプログラマが自分でエラーの原因を探さねばならず、大きな負担になってしまう。そこで、本研究では Chitil の研究 [1] に着目し、対話的な型デバッガの実装および改良を行った。本稿ではこちらの研究についてのみ述べる。

2　合成的な型推論

例として、次のプログラムを考えてみる。二行目は OCaml が実際に返した型エラーメッセージである。

```
# (fun f → f "3") (fun x → int_of_char x);;
This expression has type string but is here used
with type char
```

ここで、プログラマは f は char を受け取る関数であると思っているとする。すると、エラーの原因は左の関数の '3' を "3" と書き間違えていたことである。しかし、二行目のエラーメッセージからは「下線の x より前に x の型が string だと推論されるようなプログラムがあった」ことしか読み取れない。そのため、プログラマ自身がエラーの原因を探さなくてはいけない。

OCaml が採用している型推論アルゴリズム W ではプログラムの前の方から型を確定させながら、型推論を進める。そのため、型推論自体に偏りがあり、このような問題が生じる。

これらを解決するために、Chitil は合成的な型推論を考案した。合成的な型推論とは、プログラムの一部一部の型を独立して推論し、それらを合成していくことで全体の型を求めるような型推論である。

ここで上の例と同じプログラムを、合成的な型推論で推論してみる。

```
# (fun f → f "3") (fun x → int_of_char x);;
```

合成的な型推論では、関数適用の関数部分 (fun f → f "3") と引数部分 (fun x → int_of_char x) は別々に型推論する。すると、下のように推論される。

(fun f → f "3") : (string → α) → α
(fun x → int_of_char x) : char → int

それぞれの推論が終了すると、次に (fun f → f "3") (fun x → int_of_char x) 全体の型を求める。この式は関数適用なので、関数の型は (引数の型 → α) になる必要がある。よって、string は char と同じ型となるはずであると推論されるが、string と char は同じ型にはなり得ない。従ってアルゴリズム W と同様に型エラーだと判断される。このように合成的な型推論では、一方では型が string に、他方では char になっていたことも推論される。エラーメッセージを作る際にそれらの情報も使うことで、偏りがないエラーメッセージを作ることが出来る。

これらは $\Delta \vdash \tau$ という型付けを統合していくことで、定式化および実装されている。Δ は、ある式の型 τ を求める際の前提である。

合成的な型推論とアルゴリズム W と比べた時の特徴として、推論時に代入を行わないことが挙げられる。よって型推論木のある一部分を取り出すと、それのみで完結した型推論になっている。例えば、下の推論木を考えてみる。$\{...\} \vdash$ exp : τ は、$\{...\}$ という前提のもとで、式 exp は τ 型を持つということを意味する。最終的に f の型は int → ϵ だと推論されているが、推論木の上の方では f の型は違う型である。このように、合成的な型推論ではそこまでのプログラムで必要とされた最低限の型を知る事が出来るため、デバッギングに有用である。

$$\dfrac{\dfrac{\{\mathrm{f}{:}\alpha\} \vdash \mathrm{f}{:}\alpha \quad \{\mathrm{x}{:}\beta\} \vdash \mathrm{x}{:}\beta}{\{\mathrm{f}{:}\beta \to \delta,\ \mathrm{x}{:}\beta\} \vdash \mathrm{f\ x} : \delta} \qquad \dfrac{\{\mathrm{f}{:}\ \gamma\} \vdash \mathrm{f}{:}\ \gamma \quad \{\} \vdash 3 : \mathrm{int}}{\{\mathrm{f} : \mathrm{int} \to \epsilon\} \vdash \mathrm{f\ 3} : \epsilon}}{\{\mathrm{f}{:}\ \mathrm{int} \to \epsilon,\ \mathrm{x} : \mathrm{int}\} \vdash (\mathrm{f\ x},\ \mathrm{f\ 3}){:}\ \epsilon * \epsilon}$$

Chitil は Haskell のサブセットで定式化をしていたが、本研究では無名関数や再帰関数を含む OCaml のサブセットを構文として定式化を行った。また、プログラマによる型の定義なども含む構文で実装を行った。

3　Algorithmic Debugging

Algorithmic Debugging とは、Prolog のエラーを見つけるために Shapiro によって考案されたデバッギング法である [3]。

方法としては、プログラムの一部の結果の正否をユーザに問う。そして、ユーザの正否を利用して、プログラム全体の木の内部を移動し、エラーの原因を特定していく。ユーザの正否に従ってエラーの場所を特定するために、以下の方針を用いる。

- 全ての木のノードは正しいか誤っているかのどちらかである
- あるノードが誤っていて、それの子ノードが全て正しいならば、そのノードがエラーの原因である

Algorithmic Debugging は対象とする木の中身に左右されることがない。そのため、型推論などとは独立して実装を行うことが出来る。Algorithmic Debugging を前節で求めた型推論木に適用する事で、対話的なデバッガが実装出来る。

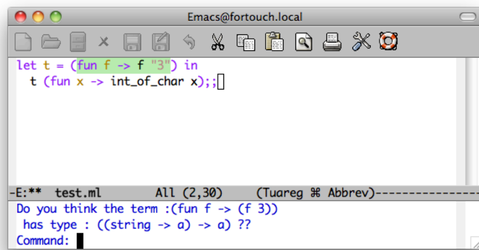

図 1: Emacs におけるデバッガの実行例

Algorithmic Debugging を使って、前節で求めた型推論木のデバッグを行ってみる。プログラマの意図した型としては `f` が char を受け取ってくる関数で、`(fun f → ...)` は $(\mathrm{char} \to \alpha) \to \alpha$ という型を持つような関数であると仮定する。

```
input > (fun f ->f "3")(fun x ->int_of_char x);;
Error is found in
    ((fun f ->(f "3"))(fun x ->(int_of_char x))).

Do you think the term : (fun f -> (f "3"))
                  has type ((string -> a) -> a)?
input > no
Do you think the term : f has type (string -> a)?
input > no
Do you think the term : "3" has type string?
input > yes
Error located : (f "3")
```

プログラマはデバッガからの質問に yes か no で答え、デバッガはその答えを使って型推論木の内部を移動する。それにより、最終的にエラーの原因は `f` が string を受け取っているところだと特定されている。これはプログラマの意図した型 (`f` は char を受け取ってくる関数) に反しているところを見つけた結果である。

4 Emacs での実装

本研究では、プログラムを見ながら対話的なデバッグが出来るように Emacs を使った拡張を行った。

図 1 に Emacs におけるデバッガの実行例を示した。プログラマは通常の Emacs と同様に、プログラムを入力する。そしてプログラムを書き終わったら、登録されている型検査のコマンドを入力することで、型推論プログラムが動く。もし型エラーがあるならば、前節のような対話的なデバッグが始まる。

Emacs 上での対話的なデバッグでは、デバッガからの問いはミニバッファ (下の小さなウィンドウ) に表示される。プログラマは自分が書いたプログラム中の色の付いた部分を参照しながら、問いに回答していく。

5 改良

前節までのシステムを使うことで、エラーの原因を探せるようになった。しかしエラーの原因に辿り着くまでに、多くの質問に答える必要があり、プログラマに負担となる。よって答える必要がない質問を減らす改良を行う。本研究では以下の 2 つの改良を行った。

5.1 回答の保存による自動回答

`(fun f → (true && (false || (true && f))) 1`

というプログラムを考えてみる。プログラマは `f` が int であると考えていると仮定する。しかし、型推論では `(true && f)` の時点で、 `f` は bool だと推論されている。よって多くの質問が `f` が bool であることを前提にしており、これは `f` は bool でないという共通の理由で全て答える事が出来る。

現在のシステムでは型付け $(\Delta \vdash \tau)$ に関する質問を行っている。しかし、ある型付けに対するプログラマからの回答が no だった場合に、原因が環境の中のどこか一部なのか、プログラムの型なのか、特定することが出来ない。そのため、環境の一つ一つと型を独立して質問されるようにした。これによって個々の質問に対して回答が保存出来るようになる。

また、この改良では Algorithmic Debugging の拡張も必要になる。構文木を移動する際に、既に回答されたリストを常に持ち歩き、それを質問と回答によって更新していくことで、実装されている。

5.2 意図した型の入力による自動推論

例として、下のプログラムを考えてみる。プログラマは関数 `make_double` が、あるリストに対して全ての要素を二倍にしたリストを作る関数 (型は int list → int list) だと考えているとする。

```
let rec make_double list = match list with
  | [] → []
  | fst :: rest →
    (fst ^ fst) :: (make_double rest)
in make_double [1;2;3]
```

しかし、`make_double` は `^` という string の結合関数を使っているため、string list → string list の関数だと推論される。このような例ではプログラマの意図した型を受け取ることで、質問回数を減らす事が出来る。

`make_double` の型が int list → int list であるという情報を受け取ると、構文に従って `fst` は int であり、`rest` は int list であることが分かる。すると、`^` は string の結合関数であるが、int を受け取っていることが分かり、原因を特定することが出来る。

6 まとめと今後の課題

本研究では、対話的な型デバッガの実装とその改良を行った。実装に関しては型の定義なども含む構文で実装されている。また、質問回数を減らすことを目的とした改良も行った。

今後は、ユーザテストを行うことで有用性について議論し、改善したいと考えている。また、質問回数をより削減するような手法についても検討したい。

参考文献

[1] Chitil O. “Compositional Explanation of Types and Algorithmic Debugging of Type Errors,” *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP’01)*, pp. 193–204 (2001).

[2] Danvy, O. “Type-Directed Partial Evaluation,” *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages (POPL’96)*, pp. 242–257 (1996).

[3] Shapiro, E. Y. *Algorithmic Program Debugging* Cambridge:MIT Press (1983).