マルチステージプログラミングのための計算体系の実装

杉浦 啓介† 亀山 幸義‡
筑波大学情報学類† 筑波大学 コンピュータサイエンス専攻‡

1 はじめに

マルチステージプログラミング（以下 MSP）とは、プログラムを生成する段階や、生成したプログラムを実行する段階など、複数のステージを持つプログラミングの手法である。MSP を用いることで、様々な環境に特化したプログラムを生成できる等の利点が考えられる。また、domain-specific 言語の効率の良い実装なども MSP によって可能となる[2]。

プログラムの効率化という観点から見ると、部分計算と MSP はほぼ似た目標を持っている。自動化されている点で部分計算が優れているように思われるが、システムに効率化を全て任せると、細かい部分にまで手が回らないことや効率化のパターンが一通りに定まってしまうこと等もある。MSP は、プログラマに効率化したい部分を書かせる仕様になっているため、多少の労力を伴う代わりに細かな調整が可能になるという利点がある。

Walid Taha らは既存の関数型言語である OCaml を拡張し、実用性のある MSP 言語 MetaOCaml を開発して公開している[1]。

MSP 言語には通常のプログラミング言語に加えて主に４つのコンストラクトが存在する。

1. Bracket

MetaOCaml では .<...>. という形で表現されるコンストラクトで、 .< と >. で囲まれた項はすぐには評価されず、コードとして出力される。

(MetaOCaml での例)
```
# let a = .<1+2>.;;
val a : ('a, int) code = .<(1 + 2)>.
```

2. Escape

MetaOCaml では .~(...) という形で表現されるコンストラクトで、Bracket をキャンセルする働きを持つ。

```
# let b = .<3 + .~a>.;;
val b : ('a, int) code = .<(3 + (1 + 2))>.
```

3. Run

MetaOCaml では .!(...) という形で表現されるコンストラクトで、生成したコードを実行する働きを持つ。

```
# let c = .! b;;
val c : int = 6
```

4. Cross-Stage Persistence

あるステージの関数や値を、その未来のステージでも再利用可能にするコンストラクトで、MetaOCaml では明示的な表現は持たないが、λ^{α} では %(...)という形で表現している[1]。

```
# let sq x = x * x;;
val sq : int -> int = <fun>
# let d = .<4 + (sq 5)>.;;
val d : ('a, int) code =
  .<(4 + (((* cross-stage persistent value
(as id: sq) *)) 5))>.
```

Taha らは MSP 言語の計算体系としてλ^{α}を提案し[1]、MSP 言語である MetaOCaml を開発した。しかし、MetaOCaml はλ^{α}を完全にカバーしているわけではなく、また MetaOcaml の型システムの定式化を行った論文は現時点では存在していない。従って、MetaOcaml で実現されている MSP の有効性の範囲や、型の保存性などを厳密に検証することはできない。

本研究では、 λ^{α}の体系を完全にカバーした処理系を実装することを目的とした。具体的には、与えられた項における型の整合性を検査して、その型を返す型推論器と、型が整合した項の計算を行う評価器の２つを設計・実装した。本研究の処理系は、MetaOCaml のように既存の言語を拡張するという手法ではなく、λ^{α}の計算体系に直接対応する型推論器と評価器(インタープリタ)の作成を行うという手法を取ることにより、自然かつ確実にλ^{α}を実現することができた。

Implementation of a Calculus for Multi-stage programming
† Keisuke Sugiura, College of Information Sciences, University of Tsukuba
‡ Yukiyoshi Kameyama, Department of Computer Science, University of Tsukuba

2 方針

λ^{α}を実装するにあたり、ベースとなる言語として OCaml を使用する。型付きの MSP 言語の作成において重要である“正しい型付け”を考慮して、今回の研究の実装方針としては、完全にλ^{α}の文法通りの動作をするような実装を行うことを第一の目標とした。また、MetaOCaml を直接拡張する形ではなく、ベースとなる言語そのものには手を加えないように MSP 機能を持つプログラムを追加するような形で実装を行った。

3 実装

λ^{α}は、通常の型付きλ計算の体系に Environment Classifier という、ステージごとにつける識別子のような概念を追加し、MSP を可能にするように拡張した計算体系である。今回作成したプログラムは、λ^{α}の文法規則を満たすかどうかを調べることで、入力された式がマルチステージプログラムとして正しい型が付いているかをチェックするプログラム infer.ml と、実際にその式を評価し、値を求めるプログラム eval.ml の二つである。

4.1 型推論器(infer)

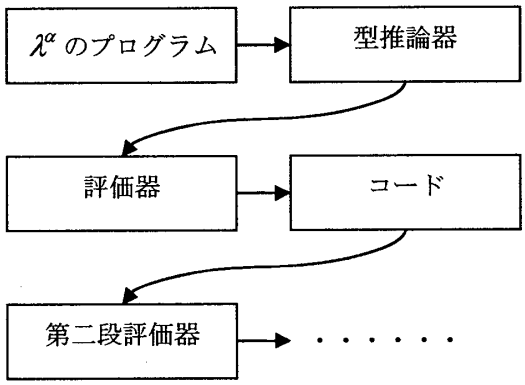
$$\Gamma \overset{A}{\mapsto} e : t$$

infer は、環境と Environment Classifier のリストΓと、式 e を与えると、Environment Classifier のリスト A が空集合という条件で型 t を見つけるプログラムである。型 t が見つからなかった場合は型が付かないのでエラーとなる。プログラムの動きとしては、式 e がどんな形であるかパターンマッチングを行い、それぞれの場合でλ^{α}の文法に従った処理を再帰的に行い、最後に返ってきたそれぞれの式の型の unification をすることで型が付くかどうか、型が付くならばどのような型であるのかを推論するようにしている。

4.2 評価器(eval)

eval は、infer によって型推論された式 e を受け取り、λ^{α}の評価規則に従って式の評価を行うプログラムである。型付きλ計算の評価器と同様に Closure を使用し、λ^{α}の形に沿うように拡張したものである。

現時点の実装で、簡単なプログラムについては型推論と評価が、λ^{α}の定義通りに実行されることを確認している。ただし、対応している構文は単純型付きλ^{α}の計算のみで、四則演算や条件文、再帰や let 文の実装は行っていない。また、λ^{α}の仕様に沿うことを第一としたため、実行速度の高速化については考慮していない。



4 まとめと今後の課題

今回の研究では、単純型付きλ^{α}の計算体系に正確に従った型推論プログラムと式評価プログラムを作成した。
前章で述べたように、四則演算や条件文などのプログラミング言語としては標準的な機能をまだ実装していないので、それらの実装や多相型の導入などによって実用的なプログラムへと進展させるといった点は今後の課題である。
また、実行速度の効率化や、速度の検証などは今回は行わなかったので、これも今後の研究課題である。

参考文献

[1] W. Taha and M. F. Nielsen, Environment Classifiers, Annual Symposium on Principles of Programming Language, pp. 26-37, 2003.

[2] W. Taha, A Gentle Introduction to Multi-stage Programming, Domain-Specific Program Generation, pp. 30-50, 2003.

[3] Y. Kameyama, O. Kiselyov, C.-c. Shan, Closing the stage: from Staged Code to Typed Closures, ACM PEPM, 2008.