λ計算に対する仕様に基づいたコンパイラの導出

横関 茉衣　(指導教員：浅井 健一)

1　はじめに

コンパイラに誤りがないことは重要である反面、正しいコンパイラを書くことは難しいものであり、技術と手間を要する。

そこで、プログラム変換の分野では初期の頃から、「正しさが保証されたコンパイラを導出する」手法が考えられてきた。正しさが保証されたコンパイラを導出するとは、コンパイラを作り、その後に正しさを証明するのではない。ソース言語の高レベルな意味論を、等式と帰納法を用いながら、コンパイラ・ターゲット言語・仮想機械に正しく変換するのである。この変換によって得られたコンパイラは構成により正しい (correct by construction) ので、その後の正しさの証明を必要としない。

Bahr と Hutton によりコンパイラの導出方法が洗練される [1] までは、コンパイラの導出は、継続や非関数化などの概念に関する知識が必要な高度なトピックとみなされてきた。そこで、Bahr と Hutton は、ソース言語の意味論から、数学で方程式の解を導出するのと同じようにコンパイラを導出する手法を提案した [1]。そして、Pickard と Hutton はその手法に対して、ソース言語に型を付けることを、依存型プログラミング言語を使用することにより可能にした [2]。本研究では、ソース言語に型を付けつつ、言語機能としてλ計算を導入した言語のコンパイラを導出した。

Bahr と Hutton も言語機能としてλ計算が導入された言語のコンパイラを導出しており、本研究ではその導出を、ソース言語に型を付けるために依存型プログラミング言語 Agda に移した。

2　コンパイラが満たすべき等式

正しさが保証されたコンパイラを導出する手法 [1] では以下のような、コンパイラの仕様を記述した、導出されたコンパイラが満たすべき等式により導出が進む。

$$exec\ (comp\ e\ c)\ s\ =\ exec\ c\ (eval\ e\ \triangleright\ s) \qquad (1)$$

この等式は、ソース言語の意味論 (*eval* 関数) とコンパイル (*compile* 関数) とターゲット言語の意味論 (*exec* 関数) の関係を結んでいる。ソース言語からターゲット言語へコンパイルしてもその意味論は変わらないという直観的な考えを形式化している。なお、e はソースプログラム、c は継続、s は実行前のスタックを表す。

exec 関数はターゲット言語 *code* とスタック s を引数として受け取ると、スタックが s の状態でターゲット言語 *code* を実行した後のスタックを返す。したがって両辺は全ての計算を終えたあとのスタックとなっている。

この等式おいて継続 c はターゲット言語で表されたその後の仕事（計算）である。左辺の $exec\ (comp\ e\ c)\ s$ は、c に仕事として e が表す計算を追加したターゲット言語である。そして、右辺の $eval\ e \triangleright s$ はスタック s の先頭に e の評価値を載せたスタックである。つまり右辺は e の実行を済ませたものとなる。

なお、この等式はソース言語に合わせて形を変える場合がある。

3　λ計算に対するコンパイラの導出

正しさが保証されたコンパイラを導出する手法 [1] に則り、λ計算が導入された型付きの言語のコンパイラを導出する。

誌面の都合上、関数適用以外の定数・λ抽象・変数については、構文の定義からコンパイル結果の導出まで終始省略した。定数・λ抽象・変数のコンパイル結果の導出は再帰を使わないため、関数適用と比較して容易に導出できる。

3.1　ソース言語の定義

3.1.1　構文

ソース言語の式は `Expr` 型とした。`Expr` 型の第一引数は式の評価値の型である。$\alpha 2 \Rightarrow \alpha 1$ は関数の引数として $\alpha 2$ 型を受け取り、$\alpha 1$ 型を返す関数の型を表す。`Expr` 型の第二引数は環境の型とした。このように、式の型に式の評価値の型を引数として与えることにより、ソース言語に型がついている。

```
data Expr : Ty → List Ty → Set where
  App : (e1 : Expr (α2 ⇒ α1) E) (e2 : Expr α2 E)
        → Expr α1 E
```

3.1.2　意味論

評価値は自然数か関数値とした。

$$\text{値}\quad Value\ :=\ Num\ \mathbb{N}\ |\ Clo\ Expr\ Env$$

意味論は評価関数ではなく、データ型 `eval` の構成子として定義した。つまり、functional semantics ではなく relational semantics とした。

$$\frac{\texttt{e1} \Downarrow_{env} Clo\ \texttt{e}\lambda\ env\lambda \quad \texttt{e2} \Downarrow_{env} v2 \quad \texttt{e}\lambda \Downarrow_{v2:env\lambda} v\lambda}{\texttt{App e1 e2} \Downarrow_{env} v\lambda}\ (\texttt{eApp})$$

これを Agda で記述すると以下のようになる。`eval e env v` は環境が `env` である下で `e` を評価した結果が `v` であることを意味する。

```
data eval : (Expr α E) → Env E → Value α
            → Set where
  eApp : {env : Env E} (envλ : Env Eλ)
         (eλ : Expr α1 (α2 :: Eλ))
         {e1 : Expr (α2 ⇒ α1) E} (vλ : Value α1)
         {e2 : Expr (α2 E} (v2 : Value α2)
         → eval e1 env (Clo eλ envλ)
         → eval e2 env v2
         → eval eλ (cons v2 envλ) vλ
         → eval (App e1 e2) env v
```

3.2　コンパイラ・仮想機械・ターゲット言語の導出

3.2.1　実行結果の関数値

3.1.2 節の $Value$ の定義では、関数値 Clo は $Expr$ 型と Env 型、つまり、ソースコードと環境で構成されていた。しかし、実行結果の値として 3.1.2 節の $Value$ を使うことには問題がある。

例えば、$(\lambda x.\, x+1)\,1$ を表すソースプログラム `App (Abs Var zero + 1) 1` をコンパイルし、実行した

場合、Abs (Var zero + 1) の実行結果は Clo (Var zero + 1) ϵ となる。よって、実行時に Var zero + 1 をコンパイルする必要が生じてしまう。

したがって、コンパイラに通した後、すべてのソースコードはターゲット言語に置き換わるべきである。よって、実行結果の値は以下のように定める。

値　$Value\text{-}c \quad := \quad Num\text{-}c\ \mathbb{N} \mid Clo\text{-}c\ Code\ Env\text{-}c$

意味論における値 $Value$ と実行結果の値 $Value\text{-}c$ はコンパイラの導出の過程で共存する。以下、conv 関数は $Value$ から $Value\text{-}c$ への変換を行う関数、conve 関数は $Value$ からなる環境から $Value\text{-}c$ からなる環境への変換を行う関数とする。

3.2.2　コンパイラが満たすべき等式

ターゲット言語の意味論を関数ではなく、データ型の構成子として記述したため、等式 (1) 右辺の $eval\ e$ 部分を変える必要がある。また、実行関数の引数をスタックから、スタックと環境のペアに変更する。すると等式 (1) は以下のようになる。

$$\begin{aligned} & exec\ (comp\ e\ c)\ \langle\ s\ ,\ conve\ env\ \rangle \\ = & exec\ c\ \langle\ EVal\ (conv\ v)\ \triangleright\ s\ ,\ conve\ env\ \rangle \quad (2) \\ & if\ e \Downarrow_{env} v \end{aligned}$$

最後の行の $e \Downarrow_{env} v$ は、式 e が環境 env のもとで v として意味論から評価されることを意味する。つまり、この等式は評価が終わらないようなλ計算を表す式は対象にしていない。

この等式に correct という名前を付け、以下のようにして Agda に乗せた。5 行目の correct の引数 $eval\ e\ env\ v$ は、等式 (2) の最後の $if\ e \Downarrow_{env} v$ に相当している。e が評価が終わらない式である場合、correct の引数 eval e env v は存在しないため、そのような e は引数となることができない。

```
correct :
  (e : Expr α E)
  (c : Code ⟨ VAL α :: S , E ⟩ ⟨ S' , E' ⟩)
  (s : Stack S) (env : Env E) {v : Value α}
  → eval e env v
  → exec (comp e c) ⟨ s , conve env ⟩
    ≡ exec c ⟨ EVal (conv v) ▷  s , conve env ⟩
```

3.2.3　導出

関数適用のコンパイル方法のみ導出する。帰納法を利用しながら、コンパイラ・仮想機械・ターゲット言語の導出を行う。導出は上記の等式 (2) の左辺と右辺が繋がるように式変形することによって進む。つまり、下記の Agda プログラムの ? 部分を繋ぐことにより進む。? 部分の下から上へと式変形する。

```
correct (App e1 e2) c s env
        (eApp eλ envλ vλ v2 p1 p2 pλ) =
  begin
    exec (comp (App e1 e2) c) ⟨ s , conve env ⟩
  ≡⟨ ? ⟩
    exec c ⟨ EVal (conv vλ) ▷ s , conve env ⟩
  □
```

まず、λ式内での環境から、関数適用が行われた場所での環境へ戻す命令を RET とすると、最後の式は次のように変形できる。

```
  exec c ⟨ EVal (conv vλ) ▷ s , conve env ⟩
= { RET の定義 }
  exec RET
    ⟨ EVal (conv vλ) ▷ EClo c (conve env) ▷ s
    , cons-c (conv v2) (conve envλ) ⟩
```

ここで、スタックの先頭にλ式の中身を実行した値 conv vλ が積まれていることに注目する。等式 (2) の e に eλ を代入した時の等式より、λ式のボディ部分を実行した直後の段階から実行する手前の段階へ式変形することができる。帰納法に基づいて導出しているため、correct の証明の途中で一つ小さい式に対する correct を使用することが可能となっている。

```
= { correct eλ RET (EClo c (conve env) ▷ s)
            (cons v2 envλ) pλ }
    exec (comp eλ RET)
    ⟨ EClo c (conve env) ▷ s
    , cons-c (conv v2) (conve envλ) ⟩
```

次に、APP と conv を定義することにより、以下のように変形できる。APP はスタックに積まれた、関数適用のλ式部分と引数部分の実行結果から、λ式のボディ部分を実行する段階へ移行するための命令である。

```
= { APP の定義 }
    exec (APP c)
    ⟨ EVal (Clo-c (comp eλ RET) (conve envλ))
        ▷ EVal (conv v2) ▷ s
    , conve env ⟩
= { conv の定義 }
      exec (APP c)
    ⟨ EVal (conv (Clo eλ envλ)) ▷ EVal (conv v2) ▷ s
    , conve env ⟩
```

最後に、再度帰納法を 2 回使用し、ソースコード App e1 e2 のコンパイル方法を定義することにより、目標の式に式変形できる。

```
= { correct e1 (APP c) (EVal (conv v2) ▷ s) env p1 }
    exec (comp e1 (APP c))
    ⟨ EVal (conv v2) ▷ s , conve env ⟩
= { correct e2 (comp e1 (APP c)) s env p2 }
    exec (comp e2 (comp e1 (APP c)))
    ⟨ s , conve env ⟩
= { comp (App e1 e2) c の定義 }
    exec (comp (App e1 e2) c) ⟨ s , conve env ⟩
```

4　まとめ

本研究では言語機能としてλ計算を導入した言語のコンパイラを導出した。依存型プログラミング言語 Agda を使用したことにより、ソース言語は型が付いたものとなった。今後はソース言語に言語機能として shift/rest を導入したい。

参考文献

[1] Patrick Bahr and Graham Hutton. Calculating Correct Compilers. *Journal of Functional Programming*, Vol. 25, p. e14, September 2015.

[2] Mitchell Pickard and Graham Hutton. Calculating dependently-typed compilers (functional pearl). *Proc. ACM Program. Lang.*, Vol. 5, No. ICFP, August 2021.