エフェクトハンドラを持つ言語に対する依存型付きコンパイラ

津山勝輝 [1], 叢悠悠 [2], 増原英彦 [3]

東京工業大学情報理工学院数理・計算科学系

[1] s-tsuyama@prg.is.titech.ac.jp

[2] cong@c.titech.ac.jp

[3] masuhara@acm.org

概要　Pickard と Hutton は、コンパイラの正当性を表す等式からコンパイラの動作を導出する方法を依存型言語 Agda を用いて拡張した。この拡張の利点は、型付きソース言語とターゲット言語の型付け可能な項を表現する構文木を定義し、型安全なコンパイラを形式化することが可能な点である。本研究では、先行研究のコンパイラがサポートしている言語機能を組み合わせかつ拡張することで、エフェクトハンドラを持つソース言語に対するコンパイラの実装を行う。準備段階として、既存のコンパイラを組み合わせることで、例外処理を追加したラムダ計算に対応するコンパイラを実装する。これを第一級継続で拡張することにより、目的のコンパイラを得る。我々のコンパイラは、依存型を用いることにより、コンパイル前後のエフェクト安全性を保証している。

1　はじめに

依存型付き言語を用いたプログラミング言語の形式化では、**型付き構文木**を定義することで、表現可能な項の型付け可能性を保証することができる。インタプリタやコンパイラなどを型付き構文木を入出力とする関数として定義すると、その型宣言は自身の型安全性を表すため、Curry-Howard 対応により、関数がホスト言語の型検査を通過すればこの性質が保証されることになる。これにより、言語の定義と性質の証明を同時に行うことができる。

これまでに、様々な言語機能を持つ言語に対して上記の方法でインタプリタやコンパイラを定義する研究が行われている [1, 2, 10, 11, 9]。我々はエフェクトに注目し、エフェクトハンドラを持つ言語に対してこの方法を適用可能かを分析する。

本研究では、依存型付き言語 Agda を用いて、エフェクトハンドラ [5] を持つソース言語 λ_{eff} に対するコンパイラを作成する。我々はまず、エフェクトハンドラに対するコンパイラを得るための準備として、ラムダ計算に例外処理機構を加えたソース言語 λ_{ex} に対してコンパイラを記述する。そして、λ_{ex} のコンパイラを第一級継続で拡張することで、λ_{eff} に対するコンパイラを得る。

先行研究 [3, 9] にはコード継続というターゲット言語における継続を表す概念が存在する。我々はこれを利用して継続を操作する計算のコンパイルを行う。しかし先行研究のコード継続は捕捉・再開されることを想定していない。そのためスタック結合の正しさを保証できず、そのまま λ_{eff} のコンパイラ定義に用いることはできない。我々の λ_{eff} のコンパイラでは前述の問題に対して、継続再開時に行われるスタックの結合が型安全に行われるように継続の型を再定義する。結果として得られたコンパイラは、型安全性と同時に、純粋なプログラムに対してコンパイル前後のエフェクト安全性 (実行時に発生するエフェクトに対し、それを処理するハンドラが存在すること) を保証する。

本論文の構成は以下の通りである。まず第 2 節で、ラムダ計算に例外処理機構を加えたソース言語 λ_{ex} に対するコンパイラを示す。次に第 3 節で、エフェクトハンドラを持つソース言語 λ_{eff} に対するコンパイラを示す。第 4 節で関連研究を紹介し、第 5 節でまとめと今後の課題を述べる。

また、本論文に掲載する Agda のコードは、スペースの都合上省略した部分が多い。完全なソースコードは GitHub で公開している [13] 。

2　関数と例外処理を持つソース言語のコンパイラ

この節では、ラムダ計算に例外処理機構を加えた言語である λ_{ex} のコンパイラを定義する。我々は Pickard と Hutton [9] の例外処理を持つ言語に対するコンパイラと、Bahr と Hutton [4] のラムダ計算に対するコンパイラを基に、これらを組み合わせることで λ_{ex} のコンパイラを得る。

以下、2.1 節でソース言語の型付き構文木とインタプリタを定義し、2.2 節でターゲット言語の型付き構文木とインタプリタを定義し、2.3 節で型安全なコンパイラを得る。

2.1　ソース言語

はじめに λ_{ex} の型付き構文木を定義する。λ_{ex} は fine-grain call-by-value [8] に基づいているため、項と型がそれぞれ値と計算に分かれているほか、let 束縛で計算の順序が明示化されている。

データ型 VTy は自然数型と関数型からなる λ_{ex} の値の型を表し、CTy は λ_{ex} の計算の型を表す。CTy の要素は値の型と真偽値のペアであり、左の要素は計算の結果の型、右の要素はその計算がハンドルされていない例外を起こすか否かを表す。true ならばハンドルされていない例外を起こす可能性がある計算であり、false ならば起こさない、すなわち純粋な計算である。CTy の定義は Pickard と Hutton [9] のソース言語の項の型に基づいている。また、Ctx は型コンテキストを表す型である。ここでは、型コンテキストを VTy 型のリストとして表現する。

以下に Agda のコードを記す。Agda では、相互再帰するデータを定義するときに、それらの型宣言を本体の定義よりも先に連続して記述する。例えば以下のコードでは、VTy と CTy が相互再帰している。また、Agda の variable 宣言は今後のコードの読みやすさのために使用する。ここで宣言した変数を型宣言中で使用する際に必要な全称量化子を省略することが可能となる。例えば今後、変数 A と B は型宣言中に VTy 型の変数として扱うことができる。

```
data VTy : Set
CTy : Set

data VTy where
  N : VTy -- 自然数型
  _⇒_ : VTy → CTy → VTy -- 関数型

CTy = VTy × Bool
```

```
Ctx : Set
Ctx = List VTy

variable
  a b : Bool
  A B : VTy
  C D : CTy
  Γ : Ctx
```

λ_{ex} の項は値を表す Val 型と、計算を表す Cmp 型のデータ型として定義する。これらのデータ型は Ctx 型の値と VTy または CTy 型の値でインデックスされており、これにより型付け可能性を保証できる。例えば Val Γ A は、型コンテキスト Γ の元で A 型に型付けされる項を表す。

```
data Val (Γ : Ctx) : VTy → Set
data Cmp (Γ : Ctx) : CTy → Set
```

Val 型のコンストラクタを以下のように定義する。var は λ_{ex} の変数を構成する。構成される変数項は型付け可能である必要があるため、型コンテキスト Γ に A 型が存在することの証明をコンストラクタ引数に受け取る。lam はラムダ抽象を構成する。引数には、A 型で拡張されたコンテキストのもとで型付け可能な関数本体を渡している。

```
data Val Γ where
  val : ℕ → Val Γ N -- 自然数値
  var : A ∈ Γ → Val Γ A
  lam : Cmp (A :: Γ) C → Val Γ (A ⇒ C)
```

Cmp 型のコンストラクタは以下のように定義する。return は λ_{ex} において、値を計算に持ち上げる項である。コンストラクタ引数は計算の結果として返す値を与える。throw は、λ_{ex} において例外を発生させる計算を表す。例外は計算の任意の箇所で発生させることができるので、throw は任意の型の計算として扱う。catch は、第一引数の計算を実行し、その中で例外が起きた場合に第二引数の例外ハンドラを実行する。

```
data Cmp Γ where
  add : Val Γ N → Val Γ N → Cmp Γ (N , false) -- 加算
  app : Val Γ (A ⇒ C) → Val Γ A → Cmp Γ C -- 関数適用
  return : Val Γ A → Cmp Γ (A , false)
  throw : Cmp Γ (A , true)
  catch : Cmp Γ (A , a) → Cmp Γ (A , b) → Cmp Γ (A , a ∧ b)
  Let_In_ : -- let 束縛
    Cmp Γ (A , a) → Cmp (A :: Γ) (B , b) → Cmp Γ (B , a ∨ b)
```

次に、λ_{ex} の意味論を形式化するためにインタプリタを定義する。インタプリタは評価する対象に応じて evalv,evalc?,evalc の 3 種類を定義する。各インタプリタは受け取った値環境のもとで値や計算を評価し、評価後の値を表す Result 型の値に変換する。それぞれの型宣言は、型安全性「型環境 Γ の元で A 型に型付けされる項は、Γ と合同な値環境のもとで評価すると A 型の値になる」を表している。そのためインタプリタが Agda の型検査を通過すれば、この性質が成立すると言える。

```
evalv : Val Γ A → Env Γ → Result A
evalc? : Cmp Γ (A , a) → Env Γ → Maybe (Result A)
evalc : b ≡ false → Cmp Γ (A , b) → Env Γ → Result A
```

Result 型の値は num とクロージャを表す clos の 2 つから構成される。clos は関数本体と、それが実行される時の値環境を受け取る。値環境は Env Γ という形の型を持ち、それは型コンテキスト Γ と合同であることを意味する。例えば、値環境 1 :: 2 :: [] は型コンテキスト N :: N :: [] と合同である。Env Γ 型の定義には All 型を使用し、Γ に含まれる各変数が格納する値を保持している。

```
data Result : VTy → Set
Env : Ctx → Set

data Result where
  num : ℕ → Result N
  clos : Cmp (A :: Γ) C → Env Γ → Result (A ⇒ C)

Env Γ = All (λ T → Result T) Γ
```

evalv は値を評価する関数である。evalv は Val 型の数値を Result 型の数値に、ラムダ抽象をクロージャに変換し、変数項の評価では対応する値を値環境から読み出す。Env Γ 型の定義によって、変数に対応する値が確実に存在することが保証される。

```
evalv (val n) _ = num n
evalv (lam f) env = clos f env
evalv (var x) env = lookup env x
```

evalc? は計算を評価する関数である。評価する計算は例外を起こす場合があるため、Maybe モナドを使用する。例外が発生した場合には nothing を返し、正常に終了した場合は結果の値を just に包んで返却する。evalc? は throw を評価するときに例外が発生したとみなし nothing を返す。return v を評価するときには v を just に包んで返す。catch c h の評価は、まず c を評価する。それが正常終了すれば結果をそのまま返し、例外が発生した場合は h の評価を行う。関数適用 app f e の評価

は f を先に評価してクロージャを得て、クロージャに保存されている値環境に e の評価結果を追加して関数本体を評価する。evalv の型によって f の評価後の値も関数型であることが保証されているため、evalc? f の評価結果がクロージャでない場合は考慮しない。

```
evalc? throw env = nothing
evalc? (return v) env = just (evalv v env)
evalc? (catch c h) env with evalc? c env
... | just v = just v
... | nothing = evalc? h env
evalc? (app f e) env with evalv f env
... | clos b env' = evalc? b ((evalv e env) :: env')
```

evalc は純粋な計算を評価する関数である。evalc に渡される計算は例外を起こさないので、Result 型の値へ必ず評価される。また、例外を起こさないことを型で表現する目的で evalc に b≡false の証明を追加の引数として与えている。evalc のアルゴリズムは evalc? とほぼ同等であるが、catch c h の c が例外を起こしたかどうかを部分評価の結果で判断するために evalc? を補助的に使用している。

```
evalc _ (return v) = evalv v
evalc _ (catch {a = true} {b = false} c h) env with evalc? c env
... | just v = v
... | nothing = evalc refl h env
evalc _ (catch {a = false} c h) env = evalc refl c env
evalc p (app f e) env with evalv f env
... | clos b env' = evalc p b ((evalv e env) :: env')
```

このように、依存型言語を使用することにより、型安全性が保証され、さらに動的な型エラーの検査が必要なくなるため簡潔な実装が可能になる。

2.2 ターゲット言語

次に、ターゲット言語の構文と意味論を定義する。ターゲット言語はスタックマシンベースの抽象機械に対する命令群である。我々は先行研究の [9, 4] の例外処理を持つ言語とラムダ計算それぞれのターゲット言語を基に、これらを組み合わせてターゲット言語を定義した。

抽象機械が操作するスタックを以下のように定義する。SValTy 型はスタックが保持する値の型を表現する。SValTy のコンストラクタはそれぞれ値、ハンドラ、継続を表す。ValTy は項の評価結果の値がスタックに追加されるときその値につけられる型である。HandTy は例外に対するハンドラにつけられる型である。ContTy は関数呼び出しの際に、関数本体の実行後に再開されるプログラム (継続) につけられる型である。HandTy, ContTy が受け取る引数は、本体の実行前後のスタックと実行時環境の形状を表す。StackTy はスタックの型であり、スタックがどの型の値をどの順番で持っているかを表す。スタック本体を表す Stack を StackTy 型の値でインデックスすることにより、スタックの形状を型レベルで表現している。

```
data SValTy : Set
StackTy : Set

data SValTy where
  ValTy : VTy → SValTy
  HandTy : Ctx → StackTy → StackTy → SValTy
  ContTy : Ctx → StackTy → StackTy → SValTy
```

```
StackTy = List SValTy

Stack : StackTy → Set
Stack S = All (λ A → StackVal A) S
```

```
variable
  Γ' Γ₁ : Ctx
  T : SValTy
  S S' S₁ S₂ S₃ : StackTy
```

StackVal はスタックが保持する値を表す。スタックは、val コンストラクタによって EnvVal 型の値を保持できる他、hand コンストラクタや cont コンストラクタによって前述したハンドラと継続のコードを保持できる。

```
data StackVal : SValTy → Set
data StackVal where
  val : EnvVal A → StackVal (ValTy A)
  cont : Code Γ S S' → RuntimeEnv Γ → StackVal (ContTy Γ S S')
  hand : Code Γ S S' → RuntimeEnv Γ → StackVal (HandTy Γ S S')
```

EnvVal 型は実行時環境が保持する値を表す。Result 型の評価結果の値に対応しており、数値とクロージャが存在する。クロージャは関数本体のコードとそれを実行する際の実行時環境を保持する。ターゲット言語のクロージャを本論文では**コードクロージャ**と呼ぶ。コードクロージャは clos と closImpure の 2 種類を定義している。clos のコンストラクタ引数は、関数本体のコードとその実行時環境である。関数本体のコードは、スタックの先頭に呼び出し元の継続が存在することを要求する。closImpure の定義は、関数本体のコードがスタックにハンドラが存在することを要求することや、値の型で関数本体が例外を起こしうることを表現している点で clos と異なる。このような区別が必要な理由は、ターゲット言語の実行において、関数本体が例外を起こす可能性の有無を判断する必要があるためである。詳細は Code 型の説明の際に述べる。RuntimeEnv は実行時環境を表す。これは Ctx 型の値でインデックスされており、RuntimeEnv Γ 型の実行時環境は Γ の各変数に対応する EnvVal 型の値を保持する。

```
data EnvVal : VTy → Set
RuntimeEnv : List VTy → Set

data EnvVal where
  num : ℕ → EnvVal N
  clos : ( ∀{ Γ₁ S₁ S₂} → Code (A :: Γ) ((ContTy Γ₁ (ValTy B :: S₁) S₂) :: S₁) S₂)
         → RuntimeEnv Γ → EnvVal (A ⇒ (B , false))
  closImpure :
    (∀{S₁ S₂ S₃ Γ Γ₁ Γ'₁} → Code (A :: Γ)
      ((ContTy Γ Γ₁ (ValTy B :: (S₁ ++ HandTy Γ Γ'₁ S₃ S₂ :: S₃)) S₂)
        :: (S₁ ++ HandTy Γ Γ'₁ S₃ S₂ :: S₃)) S₂)
    → RuntimeEnv Γ → EnvVal (A ⇒ (B , true))

RuntimeEnv Γ = All (λ T → EnvVal T) Γ
```

次にターゲット言語の構文を定義する。Code 型はターゲット言語の型付き構文木であり、各コンストラクタはスタックに対する命令を表す。Code 型は Ctx 型と 2 つの StackTy 型の値でインデックスされている。Ctx 型の値のインデックスは命令が読み出すことのできる変数の集合を表し、2 つの StackTy 型はそれぞれ実行前後のスタックの形状を表す。つまり、Code Γ S S′ 型の命令を Γ と合同な実行時環境と S 型のスタックに対し実行すると、S′ 型のスタックを得る。ターゲット言語のインタプリタである exec の型宣言はこの性質を表している。

```
data Code (Γ : Ctx) : StackTy → StackTy → Set where
```

```
exec : Code Γ S S' → Stack S → RuntimeEnv Γ → Stack S'
```

Code 型の重要なコンストラクタを抜粋して説明する。わかりやすさのために、コンストラクタ定義とそれに対応する exec のケースを並べて示す。一部を除いて、各コンストラクタは**コード継続**と呼ばれる、その命令の後に実行される命令を受け取る。

MARK は第一引数に例外ハンドラのコードを受け取り、それをスタックに追加し、UNMARK は必要なくなったハンドラをスタックから消去する。

```
MARK : Code Γ S S' → Code Γ (HandTy Γ S S' :: S) S' → Code Γ S S'

exec (MARK h c) s env = exec c (hand h env :: s) env

UNMARK : Code Γ (T :: S) S' → Code Γ (T :: HandTy Γ 1 S S' :: S) S'

exec (UNMARK c) (x :: h :: s) = exec c (x :: s)
```

THROW は例外を発生させ、スタックにあるハンドラコードを実行する。この命令は、自身が起こす例外を処理するハンドラがスタックに存在することを型によって要求している。THROW の実行時には、fail 関数を実行する。fail 関数は、ハンドラより前の要素からなる *S1* 型のプレフィックスを破棄し、ハンドラコードを実行する。

```
THROW : Code Γ (S1 ++ HandTy Γ 1 S S' :: S) S'

exec THROW s _ = fail s -- 自身を処理するハンドラコードを実行
```

LOOKUP は環境から変数に対応する値を読み出しスタックにプッシュする。

```
LOOKUP : A ∈ Γ → Code Γ (ValTy A :: S) S' → Code Γ S S'

exec (LOOKUP x c) s env = exec c ((val $ lookup env x) :: s) env
```

RET は関数本体の実行から呼び出し元の実行を再開する。実行前のスタックの先頭には関数の返り値 (val *v*) と再開するコード (cont *c env*) が存在している必要がある。

```
RET : Code Γ (ValTy A :: ContTy Γ 1 (ValTy A :: S) S' :: S) S'

exec RET (val v :: cont c env :: s) _ = exec c (val v :: s) env
```

ABS は第一引数に関数本体を受け取り、そこから生成されたコードクロージャをスタックにプッシュする。この命令が扱う関数本体は純粋なものに限られる。ABSImpure は ABS と動作は同じであるが、関数本体が例外を起こす可能性があるため、関数本体のコードは実行前のスタックにハンドラが存在することを要求する。

```
ABS :
  (∀{S1 S2 Γ 1} → Code (A :: Γ) (ContTy Γ 1 (ValTy B :: S1) S2 :: S1) S2)
  → Code Γ (ValTy (A ⇒ (B , false)) :: S) S' → Code Γ S S'

exec (ABS c' c) s env = exec c (val (clos c' env) :: s) env

ABSImpure :
  (∀{S1 S2 S3 Γ 1 Γ '1} → Code (A :: Γ)
    ((ContTy Γ 1 (ValTy B :: (S1 ++ HandTy Γ '1 S3 S2 :: S3)) S2)
      :: (S1 ++ HandTy Γ '1 S3 S2 :: S3))
    S2) →
  Code Γ (ValTy (A ⇒ (B , true)) :: S) S' → Code Γ S S'
```

```
exec (ABSImpure c' c) s env = exec c (val (closImpure c' env) :: s) env
```

APP はスタックの先頭の値を引数として環境に追加しスタックの 2 番目に存在するコードクロージャを実行する。この命令が扱うコードクロージャは純粋である必要がある。APPImpure は APP と動作は同じであるが、コードクロージャ本体が例外を起こす可能性がある。

```
APP : Code Γ (ValTy B :: S) S' → Code Γ (ValTy A :: ValTy (A ⇒ (B , false)) :: S) S'

exec (APP c) (val v :: val (clos c' env') :: s) env = exec c' (cont c env :: s) (v :: env')

APPImpure :
  Code Γ (ValTy B :: (S₁ ++ HandTy Γ ₁ S₃ S₂ :: S₃)) S₂ →
  Code Γ (ValTy A :: ValTy (A ⇒ (B , true)) :: (S₁ ++ HandTy Γ ₁ S₃ S₂ :: S₃)) S₂

exec (APPImpure c) (val v :: val (closImpure c' env') :: s) env =
  exec c' (cont c env :: s) (v :: env')
```

ABS と ABSImpure、APP と APPImpure の動作に差異はない。それにもかかわらずこれらの命令が必要なのは、関数と例外処理を組み合わせたときに起こる問題のためである。仮に命令が ABS, APP のみであれば、関数本体が例外を起こした場合にハンドラがスタックに存在することを保証できない。また ABSImpure, APPImpure のみであれば、純粋な関数であってもスタックにハンドラが存在することを強制し、ソース言語の表現力を損なう。これらの命令をひとつにまとめることができれば実装はさらに簡潔になるが、これは今後の課題である。

2.3 コンパイラ

前節で定義した λ_{ex} からターゲット言語へのコンパイラを定義する。compile はトップレベルの純粋なソース項をターゲットプログラムへコンパイルする。コンパイル結果のプログラムは、入力スタックにソース項と同じ型の値をプッシュするものになる。これがコンパイラの型安全性であり、compile の型はそれを表している。

```
compile : b ≡ false → Cmp Γ (A , b) → Code Γ S (ValTy A :: S)
```

compile の補助関数として、インタプリタと同様に compileV,compileC?,compileC の 3 つのコンパイラを定義する。各コンパイラは、コンパイル対象の項の他にコード継続を受け取る。コード継続は入力項を実行した後に実行される命令を表す。

```
compileV : Val Γ A → Code Γ (ValTy A :: S) S' → Code Γ S S'
compileC? : Cmp Γ (A , a) → Code Γ (ValTy A :: (S₁ ++ HandTy Γ ₁ S S' :: S)) S' →
            Code Γ (S₁ ++ HandTy Γ ₁ S S' :: S) S'
compileC : (b ≡ false) → Cmp Γ (A , b) → Code Γ (ValTy A :: S) S' → Code Γ S S'
```

compileV は値に対するコンパイラである。特徴的な入力ケースはラムダ抽象である。ラムダ抽象のコンパイル結果は関数本体をコンパイルしコードクロージャを生成する命令である。関数本体のコンパイル時のコード継続として、呼び出し元に実行を戻す命令である RET を使用する。ここで、関数本体が例外を起こす可能性の有無によって ABS, ABSImpure のどちらを使用するかが異なる。

```
compileV {A = A ⇒ (B , true)} (lam cmp) c =
  ABSImpure (λ {S₁ S₂ S₃ Γ ₁ Γ '₁} → compileC?
    {S₁ = (ContTy Γ ₁ (ValTy B :: (S₁ ++ HandTy Γ '₁ S₃ S₂ :: S₃)) S₂) :: _}
    cmp RET) c
compileV {A = A ⇒ (B , false)} (lam cmp) c = ABS (compileC refl cmp RET) c
```

compileC? は計算に対するコンパイラである。コンパイル対象の計算は例外を起こす可能性があるので、生成される命令は入力スタックにハンドラが存在することを要求する。catch e h に対しては、MARK を使用して h からなるハンドラをプッシュし e を実行する命令を生成する。MARK のコード継続には、e の実行後にハンドラを破棄するため UNMARK 命令を渡す。app に対しては、関数本体が例外を起こす可能性の有無によって APP, APPImpure のどちらを使用するかが異なる。

```
compileC? throw _ = THROW
compileC? (catch e h) c = MARK (compileC? h c) (compileC? {S1 = []} e $ UNMARK c)
compileC? {a = false} (app f e) c = compileV f $ compileV e $ APP c
compileC? {a = true} (app f e) c = compileV f $ compileV e $ APPImpure c
```

compileC は純粋な計算に対するコンパイラである。こちらが扱う計算は純粋であるため、生成されるコードは入力スタックにハンドラの存在を要求しない。動作は compileC? とほぼ同様であるが、注目すべきは catch e h のケースである。型によって e が例外を起こす可能性の有無がわかるので、それによって生成するコードを変えている。例外を起こさない場合にはハンドラが実行されないため、ハンドラを無視して e をコンパイルする。そうでない場合は compileC? と同様である。

```
compileC _ (catch {a = false} e h) c = compileC refl e c
compileC _ (catch {a = true} {b = false} e h) c =
  MARK (compileC refl h c) (compileC? {S1 = []} e $ UNMARK c)
```

このように、依存型を使用してコンパイラを定義することで、コンパイル前後で型が保存されることと、例外が発生してもそれを処理するハンドラがスタックに存在することを保証できる。

3　エフェクトハンドラを持つソース言語のコンパイラ

この節では、λ_{ex} のコンパイラを拡張することで、エフェクトハンドラを持つソース言語 λ_{eff} のコンパイラを定義する。以下、3.1 節でソース言語の型付き構文木とインタプリタを定義し、3.2 節でターゲット言語の型付き構文木とインタプリタを定義し、3.3 節で型安全なコンパイラを定義する。最後に 3.4 節で λ_{ex} のコンパイラから拡張する際の要点についてまとめる。

3.1　ソース言語

λ_{eff} は Hillerström ら [6] の言語を基にした、fine-grained call-by-value の言語である。λ_{eff} の項は値と計算とハンドラの 3 つに分類される。

λ_{eff} のエフェクトシグネチャはデータ型 Sig で表現されている。Sig 型のコンストラクタ op は、オペレーションの引数の型とそれが返す型を引数に受け取る。λ_{eff} のエフェクトは Eff 型で表される。λ_{eff} のエフェクトはエフェクトシグネチャの集合であり、Eff 型は Sig 型のリストとして定義する。

値、計算、ハンドラの型はそれぞれ VTy, CTy, HTy 型で表現する。値の型は自然数型と関数型に加えて、値としてのハンドラに付けられるハンドラ値型が存在する。λ_{eff} では、インタプリタの参考実装 [7] の言語に基づき、ハンドラを第一級の値として扱うためである。Hand コンストラクタはハンドラの型を値の型に持ち上げる。CTy 型は VTy 型と Eff 型のペアである。これらはそれぞれ計算結果の値の型と、計算が起こす可能性があるエフェクトを表す。HTy 型の唯一コンストラクタはハンドル前後の計算の型を受け取る。

```
data Sig where
  op : VTy → VTy → Sig

Eff = List Sig

data VTy where
  Nat  : VTy
  _⇒_  : VTy → CTy → VTy
  Hand : HTy → VTy

CTy = VTy × Eff

data HTy where
  _⇒_ : CTy → CTy → HTy

Ctx = List VTy

variable
  A B A' B' : VTy
  E E' E₁ E₂ : Eff
  C D : CTy
  H : HTy
  Γ Γ' Γ₁ Γ₂ : Ctx
```

次に型付き構文木を定義する。Val は λ_{eff} の値を表現するデータ型である。λ_{ex} の値に加えて、ハンドラを値にするためのコンストラクタ handler が追加されている。Cmp は λ_{eff} の計算を表現するデータ型である。λ_{ex} の throw や catch の代わりに、オペレーション呼び出しとハンドラが含まれる。Handler は λ_{eff} のハンドラを表現するデータ型である。ハンドラは return 節と operation 節から成る。operation 節は OperationClauses 型によって表現されており、エフェクトが持つ各オペレーションに対してそれを処理する計算を要求する。

```
data Val (Γ : Ctx) : VTy → Set
data Cmp (Γ : Ctx) : CTy → Set
data Handler (Γ : Ctx) : HTy → Set
OperationClauses : Ctx → Eff → CTy → Set

data Val Γ where
  handler : Handler Γ H → Val Γ (Hand H)

data Cmp Γ where
  Return : Val Γ A → Cmp Γ (A , E) -- 値を返すだけの計算
  Do : (op A B) ∈ E → Val Γ A → Cmp Γ (B , E) -- オペレーション呼び出し
  Handle_With_ : Cmp Γ C → Val Γ (Hand (C ⇒ D)) → Cmp Γ D -- ハンドラ

data Handler Γ where
  λx_|λx,r_ :
    Cmp (A :: Γ) C → -- return 節
    OperationClauses Γ E C → -- operation 節
    Handler Γ ((A , E) ⇒ C)

OperationClauses Γ E₁ D = All (λ {(op A' B') → Cmp ((B' ⇒ D) :: A' :: Γ) D}) E₁
```

次に、λ_{eff} の意味論を形式化するためにインタプリタを定義する。インタプリタは Kawahara ら [7] の CEK マシン形式の意味論を基にしている。この形式のインタプリタは、評価対象の項、値環境、継続を引数に受け取る。我々の λ_{eff} のインタプリタの継続は、**純粋な継続**と**メタ継続**の 2 種類に分けられる。純粋な継続は現在ハンドルされている計算の範囲に限定した継続であり、オペレーション呼び出しの際にハンドラへ渡される継続である。メタ継続は現在の計算を処理しているハンドラの外側の継続である。

継続を表現する方法として**フレーム**を使用する。フレームは評価中の項がどの式の部分項であるのかを表す。フレームは、ある計算の一部に穴を空けたものとみなせる。Frame A C は A 型の値を受け取ると C 型の計算になるフレームを表す型である。継続本体はこのフレームの連なりで構成される**スタックフレーム**として表現する。PureStackFrame A C は、A 型の値によって再開され

C 型の計算を行う純粋な継続を表す。MetaStackFrame C D は、C 型の計算を処理するエフェクトハンドリングの後に D 型の計算を行うメタ継続を表す。

```
data Result : VTy → Set
Env : Ctx → Set
```

```
data Frame : VTy → CTy → Set
data PureStackFrame : VTy → CTy → Set
data MetaStackFrame : CTy → CTy → Set
```

```
data Frame where
  app□_,_ : Val Γ A → Env Γ → Frame (A ⇒ C) C
  app_□ : Result (A ⇒ C) → Frame A C
  Do_□ : (op A B) ∈ E → Frame A (B , E)
  Handle_With □,_ :
    Cmp Γ (A , E) → Env Γ → Frame (Hand ((A , E) ⇒ (B , E'))) (B , E')

data PureStackFrame where
  empty : PureStackFrame A (A , E)
  extend : Frame A (A' , E) → PureStackFrame A' (B , E) →
             PureStackFrame A (B , E)

data MetaStackFrame where
  empty : MetaStackFrame (A , []) (A , [])
  _[_[Handle □ With_]] :
    MetaStackFrame (B , E') D → PureStackFrame A' (B , E') →
    Result (Hand ((A , E) ⇒ (A' , E'))) → MetaStackFrame (A , E) D
```

インタプリタは評価する対象に応じて 2 種類用意されている。evalv, evalc は値と計算に対するインタプリタである。補助関数として計算の評価後の値で継続を再開する resumeCont を定義する。

```
evalv : Val Γ A → Env Γ → PureStackFrame A (A' , E) →
        MetaStackFrame (A' , E) (B , []) → Result B
evalc : Cmp Γ (A , E) → Env Γ → PureStackFrame A (A' , E) →
        MetaStackFrame (A' , E) (B , []) → Result B
resumeCont : Result A → PureStackFrame A (A' , E) →
             MetaStackFrame (A' , E) (B , []) → Result B
```

evalv は与えられた値を Result 型の値へ変換し、resumeCont を呼び出すことで評価を続行する。evalc は与えられた計算に対応するフレームをスタックフレームにプッシュし、部分項の評価を行う。

```
evalv (fun f) env      = resumeCont $ clos f env
evalv (handler h) env = resumeCont $ hand h env

evalc (Return v) env K = evalv v env K
evalc (Do l v) env K = evalv v env $ (extend (Do l □) K)
evalc (Handle e With h) env K = evalv h env $ extend (Handle e With □, env) K
```

resumeCont はスタックフレームの状態によって処理が異なる。純粋な継続が空の場合、メタ継続が空であるならば評価が終了したことを意味し、メタ継続が空でなければ次にハンドラの return 節が実行されることを意味する。

```
resumeCont v empty empty = v
resumeCont v empty (H [ K [Handle □ With h ]]) with h
... | (hand (λx ret |λx,r _) env') = evalc ret (v :: env') K H
```

純粋な継続が空でない場合、その先頭にあるフレームの穴に与えられた値を入れ、フレームが表す計算を実行する。エフェクトハンドリングのフレームに対しては、メタ継続を現在の継続と穴に入れるハンドラで拡張し、純粋な継続を empty に初期化してハンドルする対象の計算を評価する。

```
resumeCont h (extend (Handle e With □, env') K) H =
  evalc e env' empty $ (H [ K [Handle □ With h ]])
```

オペレーション呼び出しのケースでは、ラベルを用いてハンドラの operation 節から対応する計算を検索して評価する。

```
resumeCont v (extend (Do l □) K) (H [ K' [Handle □ With h ]]) with h
... | hand (λx ret |λx,r es) env' =
  evalc (lookup es l) ((resump K (hand (λx ret |λx,r es) env')) :: v :: env') K' H
```

このとき、現在の純粋な継続を関数型の値として値環境に追加する。resump は Result の新しいコンストラクタであり、関数としての継続を表す。resump は引数として、捕捉した継続を表すスタックフレームと、それを処理するハンドラを要求する。

```
resump : PureStackFrame A C → Result (Hand (C ⇒ D)) → Result (A ⇒ D)
```

resumeCont の resump K' h' を 値 v に適用するケースでは、純粋な継続を K' に置き換え、メタ継続を h' と呼び出し元の継続で拡張する。

```
resumeCont v (extend (app (resump K' h') □) K) H =
  resumeCont v K' (H [ K [Handle □ With h' ]])
```

3.2 ターゲット言語

スタックに追加する値の種類は λ_{ex} のものと変わらないが、ハンドラの型 HandTy にはハンドルする計算の型の情報が追加されている。また、後述するスタック上のハンドラはメタ継続を保持するように定義されており、2 つの StackTy 型の引数はメタ継続の実行前後のスタックを表す。また、この節では ContTy 型が受け取る引数が λ_{ex} と異なる。StackTy 型の引数は、実行時のスタックの全体ではなく自身を処理するハンドラの前の要素からなるプレフィックスを表す。また、CTy 型の引数は自身が行う計算の型を表す。コード継続の再定義に伴い、ContTy をこのように再定義することとなった。新たなコード継続の定義は後述する。

```
data SValTy where
  ValTy : VTy → SValTy
  HandTy : Ctx → StackTy → StackTy → CTy → SValTy
  ContTy : Ctx → StackTy → CTy → SValTy
```

ContCode 型はコード継続を表し、そのインデックスは ContTy のものと同様である。ContCode 型は第 2 節で述べたコード継続の型とは異なる。注目すべき点は、ハンドラより後のスタック (S_2, S_3) を全称量化している点である。詳細は後述するが、ターゲット言語の命令で継続を捕捉する際には、スタックから継続が使用する部分と継続を処理するハンドラを切り取る。そして、継続を再開する際には、切り取った部分を呼び出し元のスタックに再結合する。第 2 節の設定では、ハンドラより後のスタックの要素は常に一定であった。しかしこの節では、スタックの再結合によってハンドラより後のスタックの要素が変化する場合があり、かつ継続の捕捉をするときには再結合されるスタックに関する情報が得られない。そのため、コード継続をこのように再定義する必要があった。

```
ContCode Γ S₁ (B , E) =
  ∀{ Γ ₁ S₂ S₃} → Code Γ (S₁ ++ HandTy Γ ₁ S₂ S₃ (B , E) :: S₂) S₃
```

スタックが保持する値を表す StackVal 型の新しい定義は以下の通りである。cont は第 2 節と同様にコード継続とその実行時環境を受け取る。hand は、ハンドラ本体のコードの他にメタ継続も保持する。hand の引数の HandlerCode 型は、ハンドラの return 節と operation 節に対応するコードのペアを表す。HandlerCode の 2 つの CTy 型のインデックスは、それぞれハンドル前後の計算の型を表す。id-hand はトップレベルの計算を処理するハンドラである。このハンドラのメタ継続は空であり、return 節は受け取った値をそのまま返すだけである。さらに、トップレベルの計算は純粋なため operation 節も空である。ソース言語ではこのようなハンドラを明示する必要はない。しかし後述のコンパイラの定義では、すべての計算のコンパイル後コードはスタックにそれを処理するハンドラの存在を要求する。そのため、トップレベルの計算のコンパイル後コードを実行する際には、初めに id-hand をスタックにプッシュする。

```
data StackVal : SValTy → Set where
  cont : ContCode Γ S₁ (A , E) → RuntimeEnv Γ → StackVal (ContTy Γ S₁ (A , E))
  hand :
    StackVal (ContTy Γ ₂ (ValTy B :: S₁) (A' , E₂)) -- メタ継続
    → HandlerCode Γ (A , E₁) (B , E₂) -- ハンドラ本体
    → RuntimeEnv Γ -- ハンドラコードの実行時環境
    → StackVal (HandTy Γ (S₁ ++ HandTy Γ ₁ S₂ S₃ (A' , E₂) :: S₂) S₃ (A , E₁))
  id-hand : StackVal (HandTy Γ S (ValTy A :: S) (A , []))

-- operation 節のコード
OperationCodes B E₁ E₂ Γ SS S₃ =
  All ( λ {(op A' B') → Code ((B' ⇒ (B , E₂)) :: A' :: Γ ) SS S₃ }) E₁

HandlerCode Γ (A , E₁) (B , E₂) = (∀{ Γ ₁ Γ ₂ S₁ S₂ S₃ A'} →
  Code (A :: Γ ) (ContTy Γ ₂ (ValTy B :: S₁) (A' , E₂) ::
    S₁ ++ HandTy Γ ₁ S₂ S₃ (A' , E₂) :: S₂) S₃ ×
  OperationCodes B E₁ E₂ Γ (ContTy Γ ₂ (ValTy B :: S₁) (A' , E₂) ::
    S₁ ++ HandTy Γ ₁ S₂ S₃ (A' , E₂) :: S₂) S₃)
```

ターゲット言語の型付き構文木である Code 型のインデックスの意味は前節のものと同じである。エフェクトハンドラを扱うため、いくつか命令を追加・変更している。

HANDLER は return 節、operation 節 をハンドラ型の値としてスタックにプッシュする。

```
HANDLER :
  HandlerCode Γ (A , E₁) (B , E₂) →
  Code Γ (ValTy (Hand ((A , E₁) ⇒ (B , E₂))) :: S) S' → Code Γ S S'

exec (HANDLER h c) s env = exec c (val (fc-hand h env) :: s) env
```

第一級の値としてのハンドラは EnvVal 型の fc-hand コンストラクタとして表し、return 節、operation 節とその実行時環境のみを持つ。

```
fc-hand : HandlerCode Γ (A , E₁) (B , E₂) → RuntimeEnv Γ →
            EnvVal (Hand ((A , E₁) ⇒ (B , E₂)))
```

MARK はスタックからポップしたハンドラ値と引数のメタ継続からなるハンドラをプッシュし、計算の処理を開始する命令である。

```
MARK :
  ContCode Γ (ValTy B :: S₁) (B' , E₂) → -- メタ継続
  -- 処理対象の計算
```

```
  (∀{ Γ '₁} → Code Γ
    (HandTy Γ '₁ (S₁ ++ HandTy Γ ₁ S₂ S₃ (B' , E₂) :: S₂) S₃ (A , E₁)
                :: (S₁ ++ HandTy Γ ₁ S₂ S₃ (B' , E₂) :: S₂)) S₃ ) →
  Code Γ
  (ValTy (Hand ((A , E₁) ⇒ (B , E₂))) :: S₁ ++ HandTy Γ ₁ S₂ S₃ (B' , E₂) :: S₂) S₃

exec (MARK mk c) (val (fc-hand h env') :: s) env =
  exec c (hand (cont mk env) h env' :: s) env
```

また、INITHAND はトップレベルの計算を処理するために必要な命令である。これを実行すると id-hand がスタックにプッシュされる。

```
INITHAND :
  Code Γ (HandTy Γ S (ValTy A :: S) (A , []) :: S) (ValTy A :: S) →
  Code Γ S (ValTy A :: S)

exec (INITHAND c) s env = exec c (id-hand :: s) env
```

UNMARK はスタックの先頭の値を使用してハンドラの return 節を実行する命令である。このときスタック上のハンドラが持つメタ継続をスタックにプッシュする。この継続は、return 節のコードの末尾にある RET により再開される。

```
UNMARK : Code Γ (ValTy A :: HandTy Γ ₁ S S' (A , E₁) :: S) S'

exec (UNMARK) (val x :: (hand mk h env') :: s) env with h
... | (ret , ops) = exec ret (mk :: s) (x :: env')
exec (UNMARK) (val x :: id-hand :: s) env = val x :: s
```

CALLOP は引数のオペレーションのラベルを使用して、ハンドラの operation 節から対応するコードを検索し実行する。オペレーションの実行時には、スタックの先頭にある引数の値と CALLOP の引数の捕捉されたコード継続が実行時環境に追加される。関数型の値としてのコード継続は EnvVal 型の resump コンストラクタで表す。このコンストラクタは、継続本体のコードとその実行時環境とハンドラに加えて、オペレーション呼び出しによって計算が中断された時点のスタックを受け取る。CALLOP の実行時には、split 関数でスタックをハンドラの前後で区切り、ハンドラより前のスタックを resump に渡す。

```
CALLOP :
  (op A B) ∈ E
  → ContCode Γ (ValTy B :: S₁) (A' , E) -- 捕捉されたコード継続
  → Code Γ (ValTy A :: S₁ ++ HandTy Γ ₁ S S' (A' , E) :: S) S'

exec (CALLOP l c) (val v :: s) env with split s
... | (s1 , (hand mk h env') , s2) with h
... | (_ , ops) =
  exec (lookup ops l) (mk :: s2) (resump (c , s1 , env) (fc-hand h env') :: v :: env')

resump :
  -- 継続本体とそれが使用するスタックと環境
  ContCode Γ (ValTy A :: S) (A' , E) × Stack S × RuntimeEnv Γ →
  -- 継続に対するハンドラ
  EnvVal (Hand ((A' , E) ⇒ (B , E'))) →
  EnvVal (A ⇒ (B , E'))
```

resump は関数適用の命令 APP によって再開される。このとき、継続を処理するハンドラと resump が保持している部分スタックを、関数呼び出し時のスタックに再結合する。継続を処理するハンドラは、resump が保持している HandlerCode と呼び出し元の継続をメタ継続として使用し構成する。

```
exec (APP c) (v :: val (resump (c' , s' , env₂) (fc-hand h envh)) :: s) env =
  exec c' (v :: (s' ++s (hand (cont c env) h envh :: s))) env₂

_++s_ : Stack S → Stack S' → Stack (S ++ S')
```

この節の定義は、第 2 節の ABSImpure や APPImpure などの冗長な定義を必要としない。これは、すべての計算のコンパイル後コードがスタックにそれを処理するハンドラの存在を要求するように定義したからである。

3.3 コンパイラ

トップレベルの計算のコンパイラである compile 関数の定義は以下の通りである。型宣言は、第 2 節で述べた純粋な計算に対するコンパイラと同様の性質を表す。トップレベルの計算をコンパイルした後のコードは、id-hand をスタックにプッシュしてからその計算を行い、最後に UNMARK を行う命令となる。

```
compile : Cmp Γ (A , []) → Code Γ S (ValTy A :: S)
compile c = INITHAND (compileC {S₁ = []} c UNMARK)
```

λ_{ex} のものと同様に、値と計算に対するコンパイラ compileV と compileC を定義する。compileC が受け取るコード継続はオペレーション呼び出しの際に捕捉されるため、ContCode としている。

```
compileC : Cmp Γ (A , E) → ContCode Γ (ValTy A :: S₁) (A' , E) →
             ∀{ Γ ₁ S S'} → Code Γ (S₁ ++ HandTy Γ ₁ S S' (A' , E) :: S) S'
```

ハンドラ値のケースを除いて、compileV の動作は λ_{ex} のものと同様である。ハンドラ値に対するコンパイルでは、return 節と operation 節の項をコンパイルして HandlerCode を生成し、HANDLER 命令を出力する。return 節のコンパイルでは、コード継続にメタ継続を呼び出すための RET 命令を渡している。operation 節のコンパイルでは、同様の方法を compileOps によってそれぞれのオペレーションの計算に適用する。

```
compileV (handler {H = (_ , _) ⇒ (B , E₂)} (λx ret |λx,r ops)) c =
  HANDLER (λ { Γ ₁ Γ '₁ S₁ S₂ S₃ A'} →
  compileC {S₁ = ContTy Γ '₁ (ValTy B :: S₁) (A' , E₂) :: _} ret RET , compileOps ops) c

compileC (Do l v) k = compileV v $ CALLOP l k
```

エフェクトハンドリングに対しては、ハンドラをコンパイルし値としてプッシュした後に、MARK を実行する命令を生成する。MARK に渡すメタ継続はコンパイラに渡されたコード継続である。ハンドラの処理対象の計算のコンパイルでは、その実行後にハンドラの return 節を呼び出すため、コード継続に UNMARK を渡している。他のケースに対する動作は前節のものと同様である。

```
compileC (Handle e With h) k = compileV h $ MARK k (compileC {S₁ = []} e UNMARK)
```

3.4 まとめ

ターゲット言語上での継続の捕捉と再開を型安全に行うには、第 2 節のコード継続の型では不十分であったため再定義した。第 2 節ではスタックの再結合が起こらなかったため、ハンドラより後の要素は常に一定と考えて型定義をしていた。しかし、この節の設定ではハンドラより後の要素が変化する場合がある。CALLOP 命令で継続を捕捉する際には、スタック上にあるオペレーションを処理するハンドラと、ハンドラより前の要素からなるプレフィックスを切り取る。切り取った要素はクロージャとしての継続を表す resump に保持され、ハンドラコードの実行時環境に渡される。そして、関数適用の APP 命令で捕捉した継続を再開する際に、切り取った要素を APP 実行時のスタックに再結合する。つまり、切り取られたハンドラ h について、CALLOP 実行時点のスタック上の h よりも後の要素と、APP 命令でスタックに再結合された h よりも後の要素は異なる。したがって、h の本体は再結合されうる全てのスタック上で実行可能であることを要求する。これを表現するため、ContCode 型はハンドラより後のスタック (S_2, S_3) を全称量化している。

4 関連研究

Pickard と Hutton [9] は、例外機構を持つソース言語と、単純型付きラムダ計算に対するコンパイラを Agda で実装した。彼らの実装は Bahr と Hutton [3] のコンパイラ計算という手法に基づいている。これは、コンパイラの正しさの等式からコンパイラの動作を導出する方法であり、導出されたコンパイラの正しさが保証されるという利点を持つ。Bahr と Hutton は依存型を持たない Haskell 言語を用いて上記の手法を開発したが、Pickard と Hutton はこれを依存型で拡張することで、不正な入力に対する処理の省略を可能にし、導出されたコンパイラの型安全性を保証した。我々のコンパイラは Pickard と Hutton のコンパイラを基にしているが、コンパイラ計算の手法は採用していない。これは、λ_{eff} のインタプリタが継続渡し形式で実装されており、コンパイラ計算の手法を直接適用できなかったためである。

Bahr と Hutton [4] は、コンパイラ計算の方法を、partiality モナドで拡張することによって、発散する計算を含むソース言語のコンパイラを導出した。彼らは、他のモナドを使用して同様の拡張を行うことにより、さまざまな計算効果を持つソース言語に対するコンパイラを導出できると主張している。我々のコンパイラは計算効果の抽象化であるエフェクトハンドラをサポートしているため、モナドによる拡張をせずともソース言語上で様々な計算効果を表現することができる。

5 まとめと今後の課題

本論文では、ラムダ計算に例外処理を加えた言語 λ_{ex} とエフェクトハンドラを持つソース言語 λ_{eff} に対して、型安全なコンパイラを実装した。λ_{ex} ではコンパイラを実装する際に、関数適用で例外が発生するか否かによって異なる命令を生成した。λ_{eff} では継続の再開に伴って起こるスタックの再結合が型安全に行われるようにコード継続の型を再定義した。

今後は、λ_{eff} のコンパイラをコンパイラ計算 [3] の手法に従って導出することで、コンパイラの型安全性だけでなく正しさも保証したいと考えている。具体的な手順としては、コンパイラの正しさを表す等式を定義し、3.1 節で定義した型付き構文木とインタプリタを用いて導出する。さらに、λ_{eff} の構文から、また、ソース言語の型付けに線形型やアフィン型を用いることで、Koka 言語 [12] のように、継続の使われ方に応じて最適化を行うコンパイラの実装も目指している。

謝辞　本論文の執筆にあたり、査読者の方々から有益なコメントを頂きました。また、横関茉衣氏と浅井健一氏からは、ラムダ計算のコンパイルに関して実装のヒントを頂きました。以上の方々に、この場を借りて深く感謝申し上げます。

参考文献

[1] Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. CSL '99, 1999.

[2] Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for imperative languages. *POPL*, 2017.

[3] Patrick Bahr and Graham Hutton. Calculating correct compilers. *Journal of Functional Programming*, Vol. 25, p. e14, 2015.

[4] Patrick Bahr and Graham Hutton. Monadic compiler calculation (functional pearl). *Proc. ACM Program. Lang.*, Vol. 6, No. ICFP, aug 2022.

[5] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of logical and algebraic methods in programming*, Vol. 84, No. 1, pp. 108–123, 2015.

[6] Daniel Hillerström, Sam Lindley, Robert Atkey, and K. C. Sivaramakrishnan. Continuation Passing Style for Effect Handlers. In Dale Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017)*, Vol. 84 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 18:1–18:19, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[7] Satoru Kawahara and Yukiyoshi Kameyama. One-shot algebraic effects as coroutines. In Aleksander Byrski and John Hughes, editors, *Trends in Functional Programming*, pp. 159–179, Cham, 2020. Springer International Publishing.

[8] PaulBlain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Information and Computation*, Vol. 185, No. 2, pp. 182–210, 2003.

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

[10] Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for linear, session-typed languages. CPP 2020, 2020.

[11] Syouki Tsuyama, Youyou Cong, and Hidehiko Masuhara. Intrinsically-typed interpreters for effectful and coeffectful languages. Presented at the first Workshop on the Implementation of Type Systems (WITS 2022), 2022.

[12] Ningning Xie and Daan Leijen. Generalized evidence passing for effect handlers (or, efficient compilation of effect handlers to c). *Proc. ACM Prog. Lang. (ICFP'21)*, Vol. 5, No. ICFP, p. 71, August 2021. doi: 10.1145/3473576.

[13] 津山勝輝. エフェクトハンドラを持つ言語に対する依存型付きコンパイラ (ソースコード). `https://github.com/prg-titech/Effect-Handler-Compiler/tree/main/PPL2023`.