令和3年度 学士論文

# 副作用を持つ<br>プログラミング言語を<br>型安全に定義するための<br>ライブラリの開発に向けて

東京工業大学 情報理工学院数理・計算科学系
学籍番号 18B10155

津山勝輝

指導教員

増原 英彦 教授

令和3年2月28日

# 概要

型安全インタプリタとはプログラミング言語の意味論仕様の実装方法のひとつである。型安全インタプリタの重要な特徴は、メタ言語の型検査を通して対象言語の型安全性を証明できることである。対象言語を実用的な言語機能で拡張するとき、型安全インタプリタに証明項を追加して、その機能が型安全に動作することを証明する必要がある。こうした証明項はインタプリタの実装を繁雑にするという問題がある。本研究の目的は、エフェクトやコエフェクトを持つ言語の型安全インタプリタの簡潔な実装を支援するライブラリの開発である。本論文では、特にエフェクト言語の型安全インタプリタの実装に有効な抽象化を分析するため、限定継続命令 (`shift`/`reset`) を持つ言語 $\lambda_{s/r}$ と、代数的エフェクトとハンドラを持つ言語 $\lambda_e$[21] の型安全インタプリタを依存型言語 Agda[14] を使って実装した。実装を通じて、抽象構文木の変数束縛に PHOAS[8] を用い、インタプリタを継続渡しスタイル (CPS) で実装することで、CPS 意味論の書き下しとして型安全インタプリタを実装できることを確認した。この方法による実装によって、インタプリタに複雑な証明項が出現しない簡潔な実装が可能である。また、CPS インタプリタが引数として受け取る継続やハンドラを、モナドによって抽象化することで、継続やハンドラを直接扱わずに手続き的に意味論を記述できることが期待できる。

# 謝辞

本研究を進めるにあたり、増原英彦教授、叢悠悠助教授に多くのアドバイスやご指導をいただきました。また、増原研究室の学生の皆様にも様々な知見や、研究におけるアドバイスをいただきました。本論文は以上の方々のご支援がなければ存在しえませんでした。この場を借りて感謝申し上げます。

# Contents

# List of Figures

# 第1章　はじめに

型付きのプログラミング言語の定義には、型規則と評価規則が含まれる。型規則は、プログラムを、予想される実行結果の値の種類で分類する (型付けする) ときの規則の定義である。型規則によって何らかの型を付けることができるプログラムを、型付け可能なプログラムという。評価規則は、その言語で書かれたプログラムがどのように評価 (実行) されるかを定義するものである。

プログラムに型付けする目的は、エラーを持つプログラムを実行前に検出することである。値の種類に不整合 (例えば、数値でないものを使った足し算) などがあるプログラムの実行は失敗することがある。このようなエラーは、型エラーと言われる、型付けによって検出できるエラーの一種である。

型規則と評価規則の間には、「型付け可能なプログラムは評価に失敗しない」という性質が求められる。この性質を、本論文では健全性と呼ぶ。もし型付けされたプログラムに型エラーが存在し実行に失敗するならば、型付けの意味が失われてしまうので、健全性は型規則と評価規則を定義する際に、満たされるべき重要な性質である。

Rouvoet[18] は、型付きプログラミング言語の実装の構成を、フロントエンドとバックエンドに分けて説明している。フロントエンドの役割はその言語で記述されたプログラムが与えられたとき、プログラムがその言語上で正しく動作するかどうかを検査することである。プログラミング言語が型付きであるとき、その実装のフロントエンドには**型検査器**が含まれる。型検査器は、プログラミング言語の型規則に従って、プログラムに型付けを行う。これによってプログラムの型エラーを検出することができる。バックエンドの役割は、フロントエンドを通過したプログラムを、評価したり、機械語等に翻訳することでプログラムを実行可能にすることである。**インタプリタ**はバックエンドのひとつで、プログラムを直接評価することで実行する。

型検査器とインタプリタの間には、「型検査器で型付けされたプログラムは、インタプリタによって同じ型の値に評価される」という性質が求められる。この性質を本論文では**型安全性**と呼ぶ。型安全性が成り立たなければ、型検査器を通過したにも関わらず、インタプリタがエラーを起こ

したり、ある型のプログラムを異なる型の値に評価してしまう可能性がある。これでは型検査の利点が失われてしまう。そのため、型安全性はインタプリタと型検査器を実装する上で確かめるべき重要な性質である。

型安全性を保証するには、テストを書いて実行するだけでは不十分である。テストすることで示せるのは型安全性に反するケースが存在することだけで、そのようなケースが存在し得ないことを証明することはできない。なので、型安全性を保証するには、その証明を与えることが必要である。型安全性を証明するには、紙とペンによる証明、もしくは Coq や Agda などの証明アシスタントによる仕様の形式化が必要である。従来の証明方法では、型検査器・インタプリタの仕様と型安全性の証明を分離して記述していた。証明が分離されていることで、型検査器・インタプリタの仕様を変更するたびに新たな証明を記述する必要がある。そのため、この方法は言語実装者にとっての負担であった。

これを解決するために提唱されたのが、**型安全インタプリタ** (intrinsically-typed inrterpreter)[1] である。型安全インタプリタとは、型付け可能なプログラムのみを表す**型付き抽象構文木**に対するインタプリタである。型安全インタプリタは、インタプリタの仕様の記述方法の一種である。型安全インタプリタの利点は主に以下の二つである。

- 実行可能な形式なので、意味論のテストが可能である。
- インタプリタがメタ言語の型検査を通過すれば、対象言語の型安全性が証明される。

特に重要なのは 2 つ目である。これにより、型安全性の証明を個別に記述をする必要がない。メタ言語上の型安全インタプリタの型で対象言語の型安全性の言明を表すことで、カリー=ハワード同型により、型安全インタプリタの本体は、対象言語の型安全性の証明とみなすことができる。

実用的な言語機能を持った対象言語の型安全インタプリタを簡潔に記述するためのライブラリを開発する研究 [4, 19] が存在する。これらのライブラリは、追加する機能が型安全に動作することを示す証明項をインタプリタ定義から隠蔽し、型安全インタプリタの簡潔な記述を可能にする。Poulsen ら [4] は、可変状態を持つ言語の型安全インタプリタを実装するためのライブラリを開発した。この言語の型安全インタプリタは、可変状態の数が評価に従って増えることを示す証明項を持つ。Poulsen らのライブラリは、モナドを利用した抽象化で証明項をカプセル化している。ライブラリを使用せず証明項を明示的に扱う実装では、このような証明項が、評価計算に直接関係がないにもかかわらず、インタプリタの動作定義に頻繁に出現することで、記述が繁雑になってしまう。

本研究の目的は、エフェクトやコエフェクトを持つ言語のインタプリタを簡潔に実装するためのライブラリを開発することである。本論文は特に

エフェクトに注目し、エフェクトを持つ言語の型安全インタプリタにどのような抽象化が有効かを分析するため、限定継続命令 (`shift`/`reset`) を持つ言語 $\lambda_{s/r}$ と、代数的エフェクトとハンドラを持つ言語 $\lambda_e$[21] の型安全インタプリタを実装した。実装には依存型言語の Agda[14] を用い、2 つの言語の型付き抽象構文木の変数束縛には PHOAS[8] を用いた。また、$\lambda_{s/r}$、$\lambda_e$ は明示的に継続を操作する機構があるため、インタプリタは CPS で記述した。

- $\lambda_{s/r}$ の型付き抽象構文木は、実行前と実行後の文脈が返す型の 2 つを保持する。型安全インタプリタは、`shift`/`reset` のための CPS 意味論 [11] に基づいて実装した。
- $\lambda_e$ の型付き抽象構文木は、項が実行する可能性のあるオペレーションの集合を保持する。代数的エフェクトとハンドラのための CPS 変換 [12] に基づき、$\lambda_e$ の CPS 意味論を構築し、それに対応する CPS インタプリタを記述した。

また、抽象構文木定義に PHOAS を使うことによる利点は主に以下の 2 つである。

- 対象言語における変数への代入がメタ言語の関数適用によって型安全に行われるので、メタ言語で代入規則を定義する必要がない。
- インタプリタの継続と、対象言語の関数が同じメタ言語レベルで扱われるため、継続の操作のために特別な命令をメタ言語で定義する必要がなく、CPS 意味論さえあれば、それを直接書き下す形でインタプリタを記述できる。

本論文の構成は以下の通りである。 2 章で、STLC のための型安全インタプリタの具体的な実装を通じて、本研究のアプローチについて説明する。 3 章で、STLC を限定継続命令で拡張した言語のための実装を説明する。 4 章で、STLC を代数的エフェクトとハンドラで拡張した言語のための実装を説明する。 5 章で関連研究を紹介し、最後に 6 章でまとめと今後の課題を述べる。

# 第2章　単純型付きλ計算の型安全インタプリタ

単純型付きλ計算は純粋関数型言語の小さなモデルである。この章では、単純型付きλ計算のための型安全インタプリタの実装を通じて、このアプローチについて説明する。

## 2.1　単純型付きλ計算

単純型付きλ計算の構文を以下に示す。

$$\begin{array}{rrcl} \text{(型)} & \tau & ::= & \mathit{Unit} \mid \tau \to \tau \\ \text{(項)} & t & ::= & \mathit{unit} \mid x \mid \lambda x.t \mid t\ t \\ \text{(値)} & v & ::= & \mathit{Unit} \mid \lambda x.t \\ \text{(型環境)} & \Gamma & ::= & \varnothing \mid \Gamma, x : t \end{array}$$

型は *Unit* 型と関数型からなる。項は変数 $x$、ラムダ抽象 $\lambda x.t$、関数適用 $t\ t$ と、*unit* からなる。値はラムダ抽象 $\lambda x.t$ と *unit* のみからなり、項はこれらの値のうちのどれかに評価されることになる。

型環境 $\Gamma$ は、束縛された変数と、それが参照する値の型のペアの集合である。型付け判断 $\Gamma \vdash t : \tau$ は型環境 $\Gamma$ のもとで項 $t$ は $\tau$ 型を持つという意味である。型規則は、型付け判断を導出するための推論規則である。STLC の型規則を図 2.1 に示す。規則 T-Unit は *unit* がどの型環境においても *Unit* 型を持つことを表す。規則 T-Var は、変数が型環境 $\Gamma$ で束縛されているならば、対応する型をその変数の型にする規則である。規則 T-Abs は、関数 $\lambda x.t$ に型を付ける規則である。関数本体 $t$ が、引数の型 $x : \tau_1$ を加えた型環境の元で $\tau_2$ 型になるとき、$\lambda x.t$ は $\tau_1 \to \tau_2$ 型を持つ。規則 T-App は $\tau_1 \to \tau_2$ 型の関数に $\tau_1$ の項を適用すれば $\tau_2$ 型になることを表す。

## 2.2　CPS 意味論

この節では、STLC の意味論を継続渡しスタイル [15] で記述する。意味論は、STLC の項がどのように評価されるかを表す。ここでは、評価戦略

型規則

$$\Gamma \vdash \mathit{unit} : \mathit{Unit} \qquad \text{(T-Unit)}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \text{(T-Var)}$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau_2} \qquad \text{(T-App)}$$

Figure 2.1: STLC の型規則

として左から右への値呼びをする。CPS で記述された評価関数は、「現在行っている評価の結果の値を受け取り、残りの評価を行う関数」を引数に受け取ることによって評価順序を規定する。この残りの評価を行う関数のことを継続というが、継続については 3 章で詳細に説明する。

STLC の CPS 意味論を図 2.2 に示す。$E[e]\rho\kappa$ は「環境 $\rho$ と継続 $\kappa$ のもとで項 $e$ を評価した結果」を表す。環境 $\rho$ は、変数が参照する値の集合であり、継続 $\kappa$ は、残りの評価を表す。$\mathit{unit}$ の評価は、継続 $\kappa$ に $\mathit{unit}$ を渡し計算を再開させる。変数 $x$ の評価は、環境 $\rho$ から対応する値を探し出し、その値を継続に与える。関数 $\lambda x.t$ は、メタ言語における関数へ評価される。その関数は引数の値 $v$ と新しい継続 $\kappa'$ を受け取り、$x$ の値を現在の環境に追加して $\kappa'$ のもとで関数本体 $t$ を評価する。関数適用 $e_1\ e_2$ の評価は、$e_1$、$e_2$ の順に評価し、最後にそれぞれの結果 $f$、$v$ について、$f$ に $v$ と現在の継続 $\kappa$ を渡す。型規則より、$f$ はラムダ抽象の評価結果であるはずなので、$f\ v\ \kappa$ がどのように処理されるかを理解するには、$E[\lambda x.e]\rho\kappa$ のケースを見るとよい。これによると、$f\ v\ \kappa$ は、環境に $v$ を追加し、関数 $t_1$ の本体を $\kappa$ のもとで評価する。関数適用の評価順序は $E$ に渡す継続によって制御している。よってメタ言語の評価順序に依存しない。

次の節からは、STLC のための型付き抽象構文木と型安全インタプリタの実装を示す。

$$
\begin{aligned}
E[unit]\rho\kappa &= \kappa(unit) \\
E[x]\rho\kappa &= \kappa(\rho(x)) \\
E[\lambda x.e]\rho\kappa &= \kappa(\lambda v.\lambda\kappa'.E[e]\rho[v/x]\kappa') \\
E[e_1\ e_2]\rho\kappa &= E[e_1]\rho(\lambda f.E[e_2]\rho(\lambda v.f\ v\ \kappa))
\end{aligned}
$$

Figure 2.2: STLC の CPS 意味論

```
data Ty : Set where
  Unit : Ty
  _⇒_ : Ty → Ty → Ty
```

Figure 2.3: 型の構文実装

## 2.3　構文の実装

まず、型の定義にしたがって STLC の型を表す Agda のデータ型 Ty を図 2.3 に定義する。各コンストラクタは、*unit* 型と関数型を表す。関数型のコンストラクタは「引数型と返り値型の 2 つを受け取り、型になる」と読むことができる。

次に、型付き抽象構文木を以下に定義する。

```
data Expr (Var : Ty → Set) : Ty → Set where
```

型付き抽象構文木の Agda における型は、STLC の型判断の言明 $\Gamma \vdash t : T$ を表す。すなわち、Expr *Var* $T$ は、STLC の $T$ 型の項を表す型である。Expr は Ty 型の値を受け取って型になる。これは Agda が持つ依存型の特徴である。

抽象構文木の実装にあたり、変数の名前解決には PHOAS[8] を使用する。Expr 型が受け取る *Var* は PHOAS 特有のパラメータで、変数が参照する値の型を表す。これによって、無限再帰[1] を起こすことなく、メタ言語の束縛機構を使って、オブジェクト言語の変数を管理できる。

[1] PHOAS の前身の HOAS では、以下のような定義になる。

```
data Expr : Set where
  lam : (Expr → Expr) → Expr
```

Expr 型の各コンストラクタ定義を、STLC の型規則にしたがって実装する。ここからは、Expr 型のコンストラクタを、対応する型規則と共に示す。特に、これらのコンストラクタの型は、引数型が型規則の仮定を表し、返り値型が結論を表すということに注目すべきである。

unit コンストラクタは、型規則 T-Unit を表し、自明に Unit 型を持つ。

$$\mathsf{unit} : \mathsf{Expr}\ \mathit{Var}\ \mathsf{Unit} \qquad \Gamma \vdash \mathit{unit} : \mathit{Unit}\ (\text{T-Unit})$$

var コンストラクタは、参照先の $\mathit{Var}\ T$ 型の値を受け取って $T$ 型の変数を表す。$\mathit{Var}\ T$ 型の値は、STLC における $T$ 型の値に対応する。このコンストラクタの型は型規則 T-Var を表す。この規則の仮定 $x : \tau \in \Gamma$ は、var コンストラクタが参照先の値を引数に受け取っていることで保証される。$\forall\{T\}$ で囲まれた変数 $T$ は明示的に与える必要はなく、Agda によって推論させる。

$$\mathsf{var} : \forall\ \{T\} \to \mathit{Var}\ T \to \mathsf{Expr}\ \mathit{Var}\ T$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad (\text{T-Var})$$

lam コンストラクタは、「$A$ 型の値を受け取り $B$ 型の項を返す関数」を受け取って、$A \Rightarrow B$ 型の項になる。このコンストラクタが受け取る関数は、メタ言語 (ここでは Agda のこと) の関数であることに注意すべきである。対象言語 (ここでは STLC) の変数束縛を、メタ言語の変数束縛で表現している。これは PHOAS の特徴である。

$$\mathsf{lam} : \forall\ \{A\ B\} \to (\mathit{Var}\ A \to \mathsf{Expr}\ \mathit{Var}\ B) \to \mathsf{Expr}\ \mathit{Var}\ (A \Rightarrow B)$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2} \qquad (\text{T-Abs})$$

app コンストラクタは関数適用を表す。このコンストラクタは、$A \Rightarrow B$ 型の項と $A$ 型の項を受け取り $B$ 型の項になる。

---

このような定義は Agda に許容されない。このコンストラクタによって以下のような無限再帰を起こすプログラムが書けるからである。

```
p : Expr → Expr → Expr
p (lam f) = f
w : Expr → Expr
w x = p x x
```

```
~t : Ty → Set
~t Unit = ⊤
~t (A ⇒ B) = ∀{ α : Set} → (~t A) → (~t B → α ) → α
```

Figure 2.4: 型の変換の定義

```
app : ∀ {A B} → Expr Var (A ⇒ B) → Expr Var A → Expr Var B
```

$$\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau_2} \qquad \text{(T-App)}$$

このように、型付き抽象構文木は型システムの仕様を表現する。

## 2.4　型の変換の実装

この章の最終目的は、型付き抽象構文木に対する型安全インタプリタを定義することである。インタプリタは STLC の項を Agda の値へ評価するように実装する。したがって、インタプリタ実装のために STLC の型を Agda の型に対応させる関数が必要である。型の変換 ~t の実装を図 2.4 に示す。⊤ は Agda 上の *Unit* 型と解釈してよい。STLC の関数型の変換結果は、~t *A* 型の引数と ~t *B* → α 型の関数で表される、関数適用後の継続を受け取り、α 型の値を返す Agda の関数型である。~t の本来の型は Ty → $Set_1$ であるが、Agda の {-# –type-in-type #-} オプションを使うことで *Set* と $Set_1$ の違いを無視している。

## 2.5　インタプリタの実装

これまでの定義を使うことで、型安全インタプリタを記述できる。まず、インタプリタの型定義を以下に示す。

```
eval : ∀ {T} → Expr ~t T → ~t T
evale : ∀ {T α } → Expr ~t T → (~t T → α ) → α
```

型安全インタプリタ eval は、トップレベルの項に対する評価関数であり、*T* 型の項を受け取り、~t *T* 型の値へ評価する。これは、STLC の型安全性

「$T$ 型の項は、$T$ 型の値へ評価される」を表している。したがって、eval 関数がホスト言語の型検査を通れば、STLC の型安全性を証明したことになる。eval の本体は、evale を用いて実装される。evale は $T$ 型の項と、˜t $T \to \alpha$ 型の継続を受け取り、$\alpha$ 型の値へ評価する関数である。$\alpha$ は、継続が最終的に返す値の型を表す。eval の本体は以下である。eval は与えられた項を空の継続 (受け取った値をそのまま返す) のもとで評価する。

$$\mathsf{eval}\ e = \mathsf{evale}\ e\ (\lambda\ x \to x)$$

evale の実装は CPS 意味論に基づいている。CPS 意味論の評価関数 $E$ を evale とし、$E$ が行う評価計算を Agda で書き下すことで、evale の実装が得られる。ここからは、各コンストラクタに対する evale の動作を対応する CPS 意味論と共に示す。特に、CPS 意味論の式とインタプリタのプログラムが類似していることに注目すべきである。

unit コンストラクタは、Agda の $\top$ 型の値である tt に評価される。なので、evale は継続 $k$ を tt で再開させる。$E$ が受け取る継続 $\kappa$ が、Agda における evale の引数 $k$ として意味論と同様に扱える点に注目すべきである。

$$\mathsf{evale}\ \mathsf{unit}\ k = k\ \mathsf{tt}$$

$$E[unit]\rho\kappa = \kappa(unit)$$

var コンストラクタの評価は、単に参照先の値を返す。CPS 意味論の、環境から変数の参照先の値を取る $\rho(x)$ は、var コンストラクタから引数 $x$ を取り出す操作に相当する。よって evale は環境 $\rho$ に対応する引数を必要としない。

$$\mathsf{evale}\ (\mathsf{var}\ x)\ k = k\ x$$

$$E[x]\rho\kappa = \kappa(\rho(x))$$

lam コンストラクタの評価結果は「引数の値を受け取り、ラムダ抽象の本体に代入して生成される項を評価する関数」である。CPS 意味論の、環境に変数を追加する操作 $\rho[v/x]$ は、lam コンストラクタの引数で、関数本体を表す $f$ を $v$ に適用することに相当する。PHOAS による定義によって、変数への値の代入を Agda の関数適用で表現でき、別途の代入規則などを定義する必要がなくなっている。

$$\mathsf{evale}\ (\mathsf{lam}\ f)\ k = k\ \$\ \lambda\ v\ k' \to \mathsf{evale}\ (f\ v)\ k'$$

$$E[\lambda x.e]\rho\kappa = \kappa(\lambda v.\lambda\kappa'.E[e]\rho[v/x]\kappa')$$

app コンストラクタの評価は、これまでのコンストラクタと同様に、CPS 意味論の書き下しとして見ることができる。通常の型システムを持つ言語では、$f$ が関数でないことによるインタプリタの例外を考慮する必要がある。依存型言語の Agda は、依存型パターンマッチングによって、$e_1$ の評価後の型が $t\ A \to t\ B$ であることを保証するため、このような例外を除外できる。これは、メタ言語に依存型言語を使うことによる利点のひとつである。

```
evale (app e1 e2) k =
  evale e1 $ λ f →
  evale e2 $ λ v →
  f v k
```

$$E[e_1\ e_2]\rho\kappa = E[e_1]\rho(\lambda f.E[e_2]\rho(\lambda v.f\ v\ \kappa))$$

以上のように、型付き構文木を PHOAS で定義することによって、CPS 意味論の書き下しとして、インタプリタを定義することができる。以降の章では、限定継続命令や代数的エフェクトとハンドラを持つ言語についての、この方法による実装を説明する。

# 第3章　限定継続命令による拡張

この章では、STLC を限定継続命令 `shift`/`reset`[11] で拡張した言語を実装する方法を説明する。まず、3.1 節, 3.2 節で、継続の概念と答えの型ついて説明する。次に 3.3 節で対象言語の構文と型システムを定義し、Agda による構文実装を示す。最後に 3.4 節で CPS 意味論とインタプリタ実装を示す。

## 3.1　継続

継続とは「ある時点での残りの計算」を表す概念である。具体的には、ある計算 $5 * (2 + 3) - 4$ において、部分式 $(2 + 3)$ の継続は $5 * x - 4$ である。ただし、$x$ は $(2 + 3)$ の計算結果で置き換えられる。また、限定継続は、継続の範囲が定まっているものである。例えば、継続の範囲を $< \cdot >$ で区切るとすると、計算 $5 * < (2 + 3) - 4 >$ において、$(2 + 3)$ の継続は $x - 4$ となる。

`shift`/`reset`[11] は、限定継続をプログラム中で明示的に扱うための命令である。`reset` は、継続の範囲を区切る命令である。$reset(e)$ で囲まれた式 $e$ の意味は変わらないが、継続は $reset$ の内部までに限定される。例えば、$5 * reset((2 + 3) - 4)$ において、$(2 + 3)$ の継続は $x - 4$ である。

ここまでに例示した継続は「引数 x を受け取り、残りの計算を評価する関数」と読むこともできる。もうひとつの命令 `shift` は、その時点での継続を関数として取る命令である。$shift(\lambda k.e)$ は評価される時点の継続を関数として変数 $k$ に束縛して式 $e$ を評価する。例えば、$5 * shift(\lambda k.k(2+3)) - 4$ について、変数 $k$ に束縛される継続は $5 * x - 4$ である。この計算の評価過程を以下に示す。

```
reset(5 * shift(fun k -> k (2 + 3)) - 4)
--> reset((fun k -> k (2 + 3)) (fun x -> reset(5 * x - 4)))
--> reset((fun x -> reset(5 * x - 4)) (2 + 3))
--> reset(5 * (2 + 3) - 4)
```

`shift`/`reset` には様々な応用がある。ここでは、非決定計算を `shift`/`reset` を使って記述する例を示す。

```
coin () = shift (fun k -> [k true , k false])
```

関数 coin は、自身の呼び出しを true と false の 2 通りに置き換えた上で、残りの計算を評価し、それぞれの結果をリストにする関数である。関数としての継続を異なる値にたいして 2 回呼び出すことで非決定性が表現されている。具体的には以下のように動作する。

```
reset( if coin() then 1 else 0 )
--> (fun k -> [k true , k false]) (fun x -> if x then 1 else 0)
--> [if true then 1 else 0 , if false then 1 else 0]
--> [1 , 0]
```

## 3.2　答えの型

STLC を shift/reset で拡張すると、型システムに新たに答えの型という概念が必要になる。答えの型とは「現時点の継続が返す型」である。例えば、$(2+2) == 5$ において、$(2+2)$ を評価するときの継続は $x == 5$ であり、その継続の返り値の型は *Bool* 型であるので、$(2+2)$ の答えの型は *Bool* 型である。このように評価の前後で答えの型が変化しない計算を、「純粋である」と言う。STLC の項に限れば、評価の前後で答えの型が変わることはないが、shift/reset で拡張したとき、答えの型が変化する項を書くことができる。例えば、先ほど示した関数 coin について、以下の式で coin() を評価する直前の継続は *Int* 型を返すが、coin() を評価したあとは、*List Int* 型を返すようになる。これが答えの型の変化である。型システムはこの答えの型の変化を検出する必要がある。

```
coin () = shift (fun k -> [k true , k false])
reset( if coin() then 1 else 0 )
```

## 3.3　構文の定義

この章で扱う、STLC を shift/reset で拡張した言語を、本稿では $\lambda_{s/r}$ と呼ぶことにする。$\lambda_{s/r}$ の構文定義を図 3.1 に示す。$t_1/\alpha \to t_2/\beta$ は $t_1$ から $t_2$ への関数で、呼び出すと答えの型が $\alpha$ から $\beta$ へ変化することを示す。項は STLC の項に shift と reset を追加したものである。

ここで、型の定義にしたがって、型の構文を Agda で以下のように定義できる。

```
data Ty : Set where
  Unit : Ty
  _/_⇒_/_ : Ty → Ty → Ty → Ty → Ty
```

$$
\begin{array}{lrcl}
\text{(型)} & \tau & ::= & \mathit{Unit} \mid \tau/\tau \to \tau/\tau \\
\text{(項)} & t & ::= & \mathit{unit} \mid x \mid \lambda x.t \mid t\ t \mid \mathit{shift}(\lambda k.t) \mid \mathit{reset}(t)
\end{array}
$$

Figure 3.1: $\lambda_{s/r}$ の構文

型判断は $\Gamma; \alpha \vdash e : \tau; \beta$ の形をとる。これは、「型環境 $\Gamma$ のもとで項 $e$ は $\tau$ 型を持ち、$e$ を評価すると答えの型が $\alpha$ から $\beta$ へ変化する」ことを意味する。この型判断に対応するため、Agda の実装において、型付け可能な項の型である Expr 型は評価前と評価後の答えの型に依存するよう以下のように定義する。

```
data Expr (Var : Ty → Set) (α : Ty) : Ty → Ty → Set where
  -- ...
```

Expr *Var* α *T* β 型は、$\lambda_{s/r}$ において $\Gamma; \alpha \vdash e : T; \beta$ と型付けされる項を表す型である。

次に、$\lambda_{s/r}$ 型規則を図 3.2 に示す。型規則の定義は、Danvy と Filinski ら [10] の型システムに基づいている。純粋な項の型規則では、評価の前後で answer type が同じであることに注目すべきである。規則 T-Reset は、*reset* で囲まれた式 $t$ の評価後の答えの型 $\tau$ が、$\mathit{reset}(t)$ の型になることを表す。規則 T-Shift は、文脈内で $\mathit{shift}(\lambda k.t)$ に期待される型 $\tau$ と文脈の答えの型 $\alpha$ で継続に型を付けて $k$ に束縛し、その上で $t$ の型検査を行う。

STLC と同様、型規則にもとづいて、Expr 型のコンストラクタを以下のように定義することができる。

```
unit : Expr Var α Unit α
var  : ∀ {T} → Var T → Expr Var α T α
fun  : ∀ {A B β γ} →
       (Var A → Expr Var β B γ) →
       Expr Var α (A / β ⇒ B / γ) α
app  : ∀ {A B β γ δ} →
       Expr Var γ (A / α ⇒ B / β) δ →
       Expr Var β A γ →
       Expr Var α B δ

reset : ∀ {T β} → Expr Var β β T → Expr Var α T α
```

型規則

$$\frac{x : \tau \in \Gamma}{\Gamma; \alpha \vdash x : \tau; \alpha} \quad \text{(T-Var)}$$

$$\frac{\Gamma, x : \tau_1; \alpha \vdash t : \tau_2; \beta}{\Gamma; \gamma \vdash \lambda x.t : \tau_1/\alpha \to \tau_2/\beta; \gamma} \quad \text{(T-Abs)}$$

$$\frac{\Gamma; \gamma \vdash t_1 : \tau_1/\alpha \to \tau_2/\beta; \delta \qquad \Gamma; \beta \vdash t_2 : \tau_1; \gamma}{\Gamma; \alpha \vdash t_1\ t_2 : \tau_2; \gamma} \quad \text{(T-App)}$$

$$\frac{\Gamma, k : \tau/\delta \to \alpha/\delta; \gamma \vdash t : \gamma; \beta}{\Gamma; \alpha \vdash \mathit{shift}(\lambda k.t) : \tau; \beta} \quad \text{(T-Shift)}$$

$$\frac{\Gamma; \alpha \vdash t : \alpha; \tau}{\Gamma; \beta \vdash \mathit{reset}(t) : \tau; \beta} \quad \text{(T-Reset)}$$

Figure 3.2: $\lambda_{s/r}$ の型規則

```
shift : ∀ { T β γ δ } →
        ( Var ( T / γ ⇒ α / γ) → Expr Var δ δ β ) →
        Expr Var α T β
```

## 3.4　CPS 意味論とインタプリタ

図 3.3 に CPS 意味論を示す。純粋な項の評価は、最終的に、現在の継続に評価結果の値を渡すことで計算を再開させる。限定継続命令の評価では、評価時の継続の取り扱いが STLC と比べて変化する。$\mathit{reset}(e)$ の評価は、$e$ を空の継続のもとで評価し、その結果を現在の継続に渡して計算を再開する。$\mathit{shift}(\lambda k.e)$ の評価は、現在の継続をメタ言語上の関数として k に束縛し、$e$ に代入したものをさらに評価する。$e$ を評価するときの継続は空の継続 (受け取った値をそのまま返す) である。

型システムと意味論を定義したので、$\lambda_{s/r}$ の型安全性を定義する。$\lambda_{s/r}$ の型安全性は、$\Gamma; \alpha \vdash e : \tau; \beta$ なる $e$ に関して、$E[e]\rho(\lambda v.v)$ が $\beta$ 型になることである。

ここで、CPS 意味論に基づく型安全インタプリタの実装を示す。まず、STLC の実装と同様に型の変換の定義が必要である。CPS 意味論で示した、評価の結果の型に対応させる。型の変換 ~t の実装を以下に示す。

$$
\begin{aligned}
E[unit]\rho\kappa &= \kappa(unit) \\
E[x]\rho\kappa &= \kappa(\rho(x)) \\
E[\lambda x.e]\rho\kappa &= \kappa(\lambda v.\lambda\kappa'.E[e]\rho[v/x]\kappa') \\
E[e1\ e2]\rho\kappa &= E[e1]\rho(\lambda f.E[e2]\rho(\lambda v.f\ v\ \kappa)) \\
E[reset\ e]\rho\kappa &= \kappa(E[e]\rho\ id) \\
E[shift(\lambda k.e)]\rho\kappa &= E[e]\rho[\lambda v.\lambda\kappa'.\kappa'(\kappa(v))/k]
\end{aligned}
$$

Figure 3.3: $\lambda_{s/r}$ の意味論

```
~t : Ty → Set
~t Unit = ⊤
~t (A / α ⇒ B / β) =
  (~t A) → ((~t B) → (~t α)) → (~t β)
```

次に、インタプリタの実装を示す。以下はインタプリタの型である。

```
eval : ∀ { α β } → Expr ~t α α β → ~t β
evale : ∀ {T α β } → Expr ~t α T β → (~t T → ~t α) → ~t β
```

STLC の実装と同様に、eval はトップレベルの項のインタプリタであり、型安全インタプリタ eval は $\lambda_{s/r}$ の型安全性を表す。eval が受け取る項の実行前の答えの型は項自身の型と一致する。(~t $T$ →~t α) は継続の型である。evale は T 型の式と現在の継続を受け取り、計算を行う。

インタプリタ本体の実装を以下に示す。eval は項を空の継続のもとで評価する。STLC の実装と同様、evale は CPS 意味論の仕様を反映している。

```
eval e = evale e (λ x -¿ x)

evale unit k = k tt
evale (var x) k = k $ x
evale (fun f) k = k $ λ x → λ k' → evale (f x) k'
evale (app e1 e2) k =
  evale e1 $ λ f →
  evale e2 $ λ v →
  f v k

evale (reset e) k =
```

```
    k $ evale e (λ x → x)

evale (shift f) k =
  evale (f $ λ v k' → k' $ k v) (λ x → x)
```

対象言語に限定継続命令が追加された場合でも、適切な CPS 意味論と、PHOAS の型付き抽象構文木を与えれば、以上のように CPS 意味論に直接対応する型安全インタプリタを記述できる。この方法は `shift`/`reset` 以外の限定継続命令に対しても有効である [9]。

# 第4章　代数的エフェクトによる拡張

この章では、STLC を代数的エフェクト [16] によって拡張した言語を実装する方法について説明する。まず、4.1 節で代数的エフェクトの概念について簡単に説明する。次に 4.2 節, 4.3 節で構文と型規則を説明し、4.4 節で意味論を説明する。最後に 4.5〜節 4.7 で Agda による実装を示す。

## 4.1　代数的エフェクトとハンドラ

代数的エフェクトとは、例外・状態などの副作用を表すための機能である [16]。代数的エフェクトを持つ言語では、オペレーション呼び出しによって副作用を発生させ、副作用はハンドラによって処理される。

代数的エフェクトを用いたプログラム例を示す。ここでは、3 章で例示した非決定性計算を扱う。非決定性計算を表すエフェクト `NDet` と、そのオペレーション `coin` は Eff 言語 [5] に似た構文を使って以下のように定義できる。

```
effect NDet {
  control coin() : bool
}
```

オペレーション `coin` の定義は、「呼び出されると *Bool* 型の値を返す」と読むことができる。ただし、この時点で `coin` の振る舞いは定義されていないことに注意すべきである。`coin` の振る舞いはハンドラによって決定される。`coin` を用いた計算の例を以下に示す。

```
handle {
  if (do coin()) then (return 1) else (return 0)
}
with {
  return x → x
  coin () k → [k true , k false]
}
```

これは、handle の後に続くオペレーション呼び出しを含む計算を、with の後に続くハンドラのもとで実行するプログラムである。ここでのハンドラの定義によって、coin オペレーションが呼び出されたときには、その時点での継続を true、false の 2 通りの値で再開させた結果をリストにして返すようになる。この計算の実行過程を以下に示す。

```
if (do coin()) then (return 1) else (return 0)
--> (fun _ k -> [k true , k false])
      () (fun x -> if x then (return 1) else (return 0))
--> [return 1 , return 0]
--> [1 , 0]
```

オペレーションの動作は、ハンドラの定義によって自由に変えることができる。この特徴によって、計算を再利用しやすくなるという利点がある。例えば、以下のプログラムでハンドラは coin オペレーションの動作を、「継続に true を渡して再開する」と定義している。よってプログラム全体の結果は先ほどの例と違い、1 となる。

```
handle {
  if (do coin()) then (return 1) else (return 0)
}
with {
  return x → x
  coin () k → k true
}
```

## 4.2　構文

この章で扱う、STLC を代数的で拡張した言語を本稿では $\lambda_e$ と呼ぶことにする。この言語の定義は、Hillerström ら [12] の言語に基づいている。項がエフェクトを起こさない純粋な「値」と、エフェクトを起こす可能性のある「計算」の 2 つからなることがこの言語の特徴である。$\lambda_e$ の構文を図 4.1 に示す。

まず型について説明する。値の型は、*Unit* 型と、$A$ 型の値を受け取って $C$ 型の計算を行う関数型 $A \to C$ からなる。エフェクトシグネチャは、オペレーションの型を表す。エフェクトシグネチャは引数の型と、結果の型の 2 つの情報を持つ。エフェクトは、エフェクトシグネチャの集合である。計算の型 $A!E$ は、計算結果の値の型 $A$ と、その計算が起こす可能性のあるエフェクト $E$ のペアである。$C \Longrightarrow D$ 型のハンドラは、$C$ 型の計算を処理して $D$ 型の計算に変換する。

$$
\begin{array}{rrcl}
\text{(値の型)} & A, B & ::= & \mathit{Unit} \mid A \to C \\
\text{(エフェクトシグネチャ)} & S & ::= & l : A \to B \\
\text{(エフェクト)} & E & ::= & \emptyset \mid \{S\} \uplus E \\
\text{(計算の型)} & C, D & ::= & A!E \\
\text{(ハンドラの型)} & F & ::= & C \Rightarrow D \\
\text{(値)} & V, W & ::= & \mathit{unit} \mid x \mid \lambda x.C \\
\text{(計算)} & M, N & ::= & V\ W \mid \mathit{return}\ V \mid \mathit{do}\ l\ V \mid \\
 & & & \mathit{let}\ x = M\ \mathit{in}\ N \mid \mathit{handle}\ M\ \mathit{with}\ H \\
\text{(ハンドラ)} & H & ::= & \{\mathit{return}\ x \to M\} \mid \{l\ p\ k \to M\} \uplus H
\end{array}
$$

Figure 4.1: $\lambda_e$ の構文

次に値と計算とハンドラについて説明する。値は *unit* と変数と関数からなる。計算は関数適用、return 式、オペレーション呼び出し、let 束縛、エフェクトハンドリングからなる。return 式 $\mathit{return}\ V$ は、値 $V$ を結果として返す計算である。オペレーション呼び出し $\mathit{do}\ l\ V$ は、$l$ とラベル付けされたオペレーションを引数 $V$ で呼び出す計算である。let 束縛 ($\mathit{let}\ x = M\ \mathit{in}\ N$) は、まず計算 $M$ を実行し、結果の値を $x$ に束縛して、$N$ を実行する計算である。エフェクトハンドリング ($\mathit{handle}\ M\ \mathit{with}\ H$) は、計算 $M$ が起こす副作用をハンドラ $H$ で処理する。ハンドラ $H$ は、*return* の処理と、0 個以上のオペレーションの処理の定義の集合からなる。

## 4.3　型規則

値、計算、ハンドラの型判断はそれぞれ $\Gamma \vdash V : A$、$\Gamma \vdash M : C$、$\Gamma \vdash H : C \Rightarrow D$ の形をとる。値の型判断は、「型環境 $\Gamma$ のもとで値 $V$ は $A$ 型を持つ」ことを意味する。計算とハンドラの型判断についても同様である。

型規則を図 4.2 に示す。値や関数適用の型付けは STLC と同様である。規則 T-Return は任意のエフェクト $E$ について、返す値の型を結果の型とする規則である。規則 T-Do は、エフェクト $E$ に含まれるオペレーション $l : A \to B$ を $A$ 型の値で呼び出すと、結果として $B$ 型の値が返ることを表す。規則 T-Let は、$A!E$ 型の計算 $M$ の結果、すなわち $A$ 型の値を $x$ に束縛したときの $N$ の型を検査する。規則 T-Handle は、$C$ 型の計算が $C \to D$ 型のハンドラによって $D$ 型の計算になることを表す。ハ

値の型規則

$$\Gamma \vdash \mathit{unit} : \mathit{Unit}\ (\text{T-Unit}) \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ (\text{T-Var}) \qquad \frac{\Gamma, x : A \vdash M : C}{\Gamma \vdash \lambda x.M : A \to C}\ (\text{T-Abs})$$

計算の型規則

$$\frac{\Gamma \vdash V : A \to C \qquad \Gamma \vdash W : A}{\Gamma \vdash V\ W : C}\ (\text{T-App}) \qquad \frac{\Gamma \vdash V : A}{\Gamma \vdash \mathit{return}\ V : A!E}\ (\text{T-Return})$$

$$\frac{(l : A \to B) \in E \qquad \Gamma \vdash V : A}{\Gamma \vdash \mathit{do}\ l\ V : B!E}\ (\text{T-Do}) \qquad \frac{\Gamma \vdash M : A!E \qquad \Gamma, x : A \vdash N : B!E}{\Gamma \vdash \mathit{let}\ x = M\ \mathit{in}\ N : B!E}\ (\text{T-Let})$$

$$\frac{\Gamma \vdash M : C \qquad \Gamma \vdash H : C \Rightarrow D}{\Gamma \vdash \mathit{handle}\ M\ \mathit{with}\ H : D}\ (\text{T-Handle})$$

ハンドラの型規則

$$\frac{\begin{array}{c} H = \{\mathit{return} x \to M\} \uplus \{l\ p\ k \to N_l\}_l \qquad C = A!l : A_l \to B_{ll} \\ \Gamma, x : A \vdash M : D \qquad \forall l.\ \Gamma, p : A_l, k : B_l \to D \vdash N_l : D \end{array}}{\Gamma \vdash H\ C \Rightarrow D}\ (\text{T-Handler})$$

Figure 4.2: $\lambda_e$ の型規則

ンドラの型規則 T-Handler は、$return$ と計算の型 $C$ 中のオペレーションとシグネチャ $\{l\ p\ k \to N_l\}_l$ を $D$ 型の計算で処理するように定義されたハンドラに $C \Rightarrow D$ 型を付ける。

## 4.4　CPS 意味論

代数的エフェクトとハンドラのための CPS 変換 [12] に基づいて、図 4.3 のように CPS 意味論を定義する。計算を評価する $E_c$ は新たにハンドラ $h$ を引数に受け取るようになる。ハンドラ $h$ は、オペレーションのラベル、パラメーター、継続を受け取ってオペレーションを処理する。純粋な値はメタ言語の値に評価される。$return\ V$ の評価は、継続に $V$ の評価結果を渡して再開させる。$do\ l\ V$ の評価は、ハンドラ $h$ にラベル $l$、引数 $V$、継

$$
\begin{array}{rcl}
E_v[unit]\rho & = & unit \\
E_v[x]\rho & = & \rho(x) \\
E_v[\lambda x.M] & = & \lambda v.\lambda\kappa.\lambda h.E_c[M]\rho[v/x]\kappa h \\
E_c[V\ W]\rho & = & (E_v[V]\rho)\ (E_v[W]\rho) \\
E_c[return\ V]\rho\kappa & = & \kappa\ (E_v[V]\rho) \\
E_c[do\ l\ V]\rho\kappa h & = & h(l, E_v[V]\rho, \lambda x.\kappa\ x\ h) \\
E_c[let\ x = M\ in\ N]\rho\kappa & = & E_c[M]\rho(\lambda v.E_c[N]\rho[v/x]\kappa) \\
E_c[handle\ M\ with\ H]\rho\kappa h & = & E_c[M]\rho(E_h[H_{ret}])(E_h[H_{ops}]) \\
 & where & H_{ret} = \{return\ x \to N\} \\
 & & H_{ops} = \{l\ p_l\ k_l\ \to N_l\}_l \\
 & & E_h[H_{ret}] = \lambda x.\lambda h.E_c[N] \\
 & & E_h[H_{ops}] = \lambda z.case\ z\ with\ \{(l,p,k) \to E_c[N_l]\rho[p/p_l][k/k_l]\} \\
\top[M] & = & E_c[M]\emptyset(\lambda x.\lambda h.x)(\lambda z.())
\end{array}
$$

Figure 4.3: $\lambda_e$ の CPS 意味論

続 $\lambda x.\kappa\ x\ h$ を渡してオペレーションを処理する。$let\ x = M\ in\ N$ の評価は、$M$ を評価して、その結果を環境 $\rho$ に追加して $N$ を評価する。この評価順序は $E_c$ に渡す継続によって制御している。$handle\ M\ with\ H$ の評価は、$H$ に基づいて生成された継続とハンドラで $M$ を評価する。このときの継続とハンドラは、ハンドラ $H$ の $return$ 節の $H_{ret}$、$operation$ 節の $H_{ops}$ から作られる。

型システムと意味論を定義したので、$\lambda_e$ の型安全性を定義する。$\lambda_e$ の型安全性は、$\Gamma \vdash M : A!\emptyset$ なる $M$ に対して、$\top[M]$ が $A$ 型の値になることである。

## 4.5　構文の実装

ここでは、$\lambda_e$ を Agda で実装する方法を示す。

まず、型の定義にしたがって型の構文を以下のように定義する。VTy、CTy、HTy はそれぞれ値の型、計算の型、ハンドラの型を表す。Sig はエフェクトシグネチャを表し、エフェクトを表す Eff は、Sig のリストとして定義される。

```
-- Value types
data VTy : Set where
  Unit : VTy
  _⇒_ : VTy → CTy → VTy

-- Effect signatures
data Sig : Set where
  op : VTy → VTy → Sig

-- Effects
Eff : Set
Eff = List Sig

-- Computation types
CTy : Set
CTy = VTy × Eff

-- Handler types
data HTy : Set where
  _⇒_ : CTy → CTy → HTy
```

次に、定義にしたがって値、計算、ハンドラの構文を以下のように定義する。STLC と同様に変数束縛には PHOAS を使用している。Do コンストラクタに渡すオペレーションラベルは、シグネチャのリストである Eff 中の要素のインデックスを表す型 _∈_ で表現している。ハンドラの operation 節は、型レベルの map 関数である All を使って、エフェクト $E$ 中の各シグネチャに対応する動作定義を受け取る。

```
mutual
  -- Values
  data Val (Var : VTy → Set) : VTy → Set where
    unit : Val Var Unit
    var  : ∀{T} → Var T → Val Var T
    fun  : ∀{A C} → (Var A → Cmp Var C) → Val Var (A ⇒ C)

  -- Computations
  data Cmp (Var : VTy → Set) : CTy → Set where
    app    : ∀{A C} → Val Var (A ⇒ C) → Val Var A → Cmp Var C
    Return : ∀{E A} → Val Var A → Cmp Var (A , E)
    Do     : ∀{E A B} → (op A B) ∈ E → Val Var A → Cmp Var (B , E)
    Let_In_ : ∀{E A B} → Cmp Var (A , E) →
```

```
                (Var A → Cmp Var (B , E)) → Cmp Var (B , E)
  Handle_With_ : ∀ {C D} → Cmp Var C → Handler Var (C ⇒ D) → Cmp Var D

-- Handlers
data Handler (Var : VTy → Set) : HTy → Set where
  return_/ops_ : ∀{A E C} →
    -- return clause
    (Var A → Cmp Var C) →
    -- operation clauses
    All (λ {(op A' B') →
       Var A' → (Var (B' ⇒ C)) → Cmp Var C }
    ) E →
    Handler Var ((A , E) ⇒ C)
```

## 4.6　型の変換

STLC の実装と同様に、型の変換を実装する。ここでは、値の型の変換 ˜t と、計算の型の変換 ˜c を定義する。˜t は自明な型の変換である。˜c による計算の型の変換先は、それが持つエフェクトに基づく Cmp 型の継続と、Hand 型のハンドラを受け取り、答えの型 α の値を返す関数型である。˜c の本来の型は $Cty \to Set_1$ である[1] が、Agda の {-# –type-in-type #-} オプションを使うことで $Set$ と $Set_1$ の違いを無視している。また、ハンドラの型 Hand の定義は Agda の停止性検査を通過しないため、{-# TERMINATING #-} オプションを使って停止性検査を省略している。

```
mutual
  ˜t : VTy → Set
  ˜t Unit = ⊤
  ˜t (A ⇒ C) = ˜t A → ˜c C

  ˜c : CTy → Set
  ˜c (A , E) = ∀{ α : Set} → Cont (A , E) α → Hand E α → α

  Cont : CTy → Set → Set
  Cont (A , E) α = ˜t A → Hand E α → α
```

[1] Girard のパラドックスは、Agda の型理論において $Set : Set$ は矛盾しているということを示した。現在 Agda は $Set$ 型にレベルをつけてこれを解決している。例えば、$Set : Set_1$ であり、$Set_1 : Set_2$ である。また、$(Set \to Set) : Set_1$ でもある。

```
{-# TERMINATING #-}
Hand : Eff → Set → Set
Hand E α = ∀{A B} → (op A B) ∈ E → ˜t A → (˜t B → α) → α
```

## 4.7　インタプリタ

ここでは 3 つのインタプリタを以下に定義する。eval はトップレベルの $\lambda_e$ プログラムの評価関数であり、これが型安全インタプリタそのものである。eval の型は $\lambda_e$ の型安全性の言明を表しており、その第一引数は空のエフェクトを持つ計算の型である。これは、トップレベルのプログラムがハンドラによって全ての副作用を処理する必要があることを示している。

eval は evalv と evalc の 2 つのインタプリタを使って定義される。evalv は値のための、evalc は純粋でないものを含む任意の計算のためのインタプリタである。

```
eval : ∀{A} → Cmp ˜t (A , []) → ˜t A
evalv : ∀{A} → Val ˜t A → ˜t A
evalc : ∀{C} → Cmp ˜t C → ˜c C
```

3 つのインタプリタの本体の実装を以下に示す。STLC と同様 evalv、evalc、eval はそれぞれ 4.4 節で示した $E_v$、$E_c$、$\top$ の定義に対応する。

```
evalv unit     = tt
evalv (var x) = x
evalv (fun f) = λ x → evalc (f x)


evalc (app v1 v2) k =
  let f = evalv v1 in
  let v = evalv v2 in
  f v k
evalc (Return v) k =
  k $ evalv v
evalc (Do label v) k h =
  h label (evalv v) (λ x → k x h)
evalc (Let e In f) k =
  evalc e $ λ x →
  evalc (f x) k
evalc {C = C} (Handle e With (return ret /ops calls)) =
  evalc e { α = ˜c C} (λ x _ → evalc (ret x)) (λ l p k → evalc (lookup calls l p k))
```

```
eval c = evalc c init-k pure-h
  where
    init-k : ∀{A} → Cont (A , []) (˜t A)
    init-k x _ = x
    pure-h : ∀{A} → Hand [] (˜t A)
    pure-h ()
```

$\lambda_e$ では、対象言語の CPS 意味論にハンドラが追加されたが、CPS 意味論、PHOAS の型付き抽象構文木、型の変換を適切に与えれば、今までと同様に CPS 意味論に直接対応する型安全インタプリタを記述できる。型の変換の実装の際に {-# –type-in-type #-} と {-# TERMINATING #-} の Agda オプションを使用したが、これらを取り除くことは今後の課題である。

# 第5章　関連研究

## 5.1　型付き抽象構文木と型安全インタプリタ

**定義インタプリタ**　定義インタプリタ [17] は、プログラミング言語の意味論を記述することを目的にするインタプリタである。型安全インタプリタも定義インタプリタのひとつであり、プログラミング言語の意味論を記述するとともに、その型安全性も同時に証明することができるという点で、より魅力的である。

**依存型**　依存型言語を用いて Intrinsically-typed language を定義するという方法は以前から提唱されてきた。例えば、Altenkirch と Reus[1, 3] は STLC のための型付き抽象構文木に対する型安全インタプリタを依存型言語で定義することを提唱した。

### 5.1.1　GADT

一般化された代数的データ型 (GADT)[22] とは、代数的データ型の拡張であり、コンストラクタの型を明示的に記述することを可能にする。GADT を使えば、型付き抽象構文木は依存型を使わず記述することも可能である。

例えば、数値と真理値からなる算術式の抽象構文木を Haskell の通常の代数的データ型で以下のように表現できる。

```
data Exp = Num Int | Bl Bool | Add Exp Exp
```

*Add* コンストラクタは、任意の *Exp* 型の値を引数に取る。これにより型付けできない項を表現することができるので、この *Exp* は型付き抽象構文木ではない。$a$ 型の項を表現するために、*Exp* 型に型パラメータを追加し *Exp* $a$ 型として定義することは、各コンストラクタの結果の型が自動的に任意の型 $a$ に対して *Exp* $a$ 型になるので無意味である。よってこの表現では、型付き抽象構文木を記述することはできない。次に、同じ言語の型付き抽象構文木を GADT で以下のように記述できる

```
data Exp a where
  Num : Int  -> Exp Int
```

```
Bl  : Bool -> Exp Bool
Add : Exp Int -> Exp Int -> Exp Int
```

GADT ではコンストラクタの結果の型を明示的に書くことができる。例えば、*Bl* コンストラクタは必ず *Exp Bool* 型の値を生成するので、真理値が *Add* コンストラクタに渡されるエラーは型検査で除外できる。よってこの *Exp* は型付き抽象構文木である。

GADT は変数のない算術式に対しては十分有効な方法であるが、対象言語の機能をさらに拡張する際には不十分と言える。例えば、変数を対象言語に導入する場合、型付き抽象構文木に現れる変数が未定義でないという証明を追加する必要がある。例えば、変数束縛を de Brujin インデックスで表現する場合、その変数が型環境の中に存在しているという証明項が必要である。このような証明項を扱うのであれば、通常の型システムよりも依存型をシステムを使う方が簡潔に記述できる。無限再帰するプログラムが書けるという問題があることを無視すれば、依存型を使用せずとも HOAS による変数束縛の表現は可能であるが、変数以上に高度な機能を対象言語に追加することを考えれば、メタ言語に依存型言語を使用することは優れた方法である。

### 5.1.2　Tagless Final

依存型を使わない型安全なインタプリタ実装の一例に、Tagless Final が存在する。Tagless Final[7] とは、メタ言語上に対象言語を設計する際に用いられる統合的手法である。

Tagless Final の利点は、型安全インタプリタと同様に対象言語とその評価を型安全に定義でき、対象言語のプログラムの型付けをメタ言語の型システムに委譲できる点である。さらに、Tagless Final は依存型などの発展的な型システムを必要とせず、Scala や Haskell などの通常のプログラミング言語上で実装できる。Tagless Final では HOAS を使用することができ、対象言語の変数束縛をメタ言語で表現できる。また、対象言語のプログラムの評価には、インタプリタ、CPS 変換、部分評価器などの実装をサポートしている。

ただし、Tagless Final で実装できる対象言語に可変状態などの副作用を扱う機能を追加することは難しいと思われる。可変状態を持つ言語のインタプリタを型安全に定義するためには、可変状態の数が評価に従って増加することを示す証明項をインタプリタに追加する必要がある [4]。しかし証明項を取り扱うにはやはり依存型が必要である。本研究や後述する先行研究では、依存型言語をメタ言語として使用し、様々な機能に対する型

安全インタプリタの簡潔な実装方法や、そのためのライブラリを開発している。

## 5.2　外在的・内在的検証

ある関数が目的の仕様を満たしているかを確認する方法に、内在的検証と外在的検証が存在する。

外在的検証では、関数は内在的検証の場合よりも弱い仕様で記述し、その関数が仕様を満たすことを付属の補題として追加する。つまり、関数の動作の定義と、仕様を満たすことの証明を分離して記述する。例えば、インタプリタが型安全性を満たしていることを外在的検証で示す場合、まずインタプリタを型付きでない通常の抽象構文木の変換として定義し、そのインタプリタが型安全性を満たすという証明を付属する。

一方、内在的検証では、検証する関数の型シグネチャで満たすべき仕様を記述する。このように記述することで、仕様を満たすことの証明が関数の実装に統合される。例えば、型安全インタプリタは内在的検証の一例である。この場合、検証したい関数はインタプリタであり、満たすべき仕様は対象言語の型安全性である。型安全インタプリタの実装では、インタプリタの型で対象言語の型安全性を表している。

## 5.3　高度な機能の型安全な実装

本論文では、限定継続命令・代数的エフェクトとハンドラを持つ言語のための型安全インタプリタを実装したが、先行研究 [4, 19] は、他の機能を持つ言語の型安全インタプリタを簡潔に実装する方法を示している。

Poulsen ら [4] は、可変状態や動的束縛の機能を持つ言語を実装するためのライブラリを開発した。このライブラリは型安全な可変状態ストアに対する操作を行う命令を含んでおり、これによってインタプリタの簡潔な実装が可能になる。本研究と違い、対象言語の構文は de Brujin インデックスを用いて実装されており、インタプリタにはモナドが使用されている。de Brujin インデックスとは、変数束縛の表現のひとつであり、変数はその名前の代わりに、その変数を束縛する $\lambda$ が何階層外側にあるかを表す数値を使う。例えば $\lambda x.\lambda y.x\ y$ は、de Brujin インデックスを使うと $\lambda.\lambda.1\ 0$ と表現される。PHOAS による実装との違いは、変数コンストラクタは参照先の値を受け取らず、型環境 $\Gamma$ の要素へのポインタで束縛関係を表現しており、代入規則を定義する必要があるという点が挙げられる。Poulsen らのインタプリタは、型環境と対応する値環境を評価時に使用し、変数コンストラクタの評価は、引数のポインタで値環境中の値を取り出す。ま

た、関数適用の評価においては、値環境に引数部分の評価結果を挿入してから、関数本体を評価することで変数への代入を表現している。

また、Rouvoet ら [19] は、線形型やセッション型を持つ言語を実装するためのライブラリを開発した。ライブラリは分離論理に基づいて型付き抽象構文木とインタプリタの実装を抽象化した。この抽象化によって実装から線形性の証明に関わる項を隠し、簡潔な実装を支援している。こちらもインタプリタにはモナドを使用しており、線形的な値の取り扱いを強制している。

# 第6章　まとめと今後の課題

## 6.1　まとめ

本論文では、限定継続命令を持つ言語 $\lambda_{s/r}$ と、代数的エフェクトとハンドラを持つ言語 $\lambda_e$ の型安全インタプリタを定義した。これらのインタプリタは、Agda の型検査を通し、それぞれの対象言語の型安全性を証明している。

$\lambda_{s/r}$ と $\lambda_e$ の型安全インタプリタは、特別な抽象化を必要とせずとも簡潔な実装は可能であった。これは、型付き抽象構文木を PHOAS で実装することによって、CPS 意味論の率直な書き下しとして、インタプリタを実装することができたからである。この方法は try-catch などの継続を操作するようなエフェクトに対しても有効であると考えられる。

また、本論文のインタプリタは、継続やハンドラを明示的に引数に受け取っていた。これらを隠蔽し、継続やハンドラに関する操作を抽象化するモナドを開発できると考えられる。このモナドにより継続やハンドラを直接扱うことなく、手続き的に意味論を記述できることが期待できる。

## 6.2　今後の課題

今後は、エフェクトを持つ言語の型安全インタプリタをモナドで記述しライブラリ化する。本論文で実装したインタプリタは、Agda のオプションを用いて、Agda の型検査器がエラーを検出すべき箇所をいくつか無視させている。これらのオプションを取り除くことも今後の課題である。

本研究の目的は、エフェクトやコエフェクトを持つ言語の型安全インタプリタを簡潔に実装するためのライブラリの開発である。これを達成するため、今後は他のエフェクト・コエフェクトに対する型安全インタプリタの実装も探究する。エフェクトを持つ言語に関しては、トレースエフェクトを持つ言語の型安全インタプリタの実装する。また、コエフェクトを持つ言語に関しては、quantitative types[2] などの機能を持つ言語のためのインタプリタを実装する。

### 6.2.1　Agda のオプション

本論文では、$\lambda_e$ の実装の際に、Agda のオプションを用いて、インタプリタや型の変換は停止するという仮定 ({-# TERMINATING #-} ) と、型のレベルの違いを無視できるという仮定 ({-# –type-in-type #-}) を置いた。これらの仮定を解消することは今後の課題である。

**Terminating**　停止しない関数は 論理における矛盾に対応するため、Agda の型検査は関数が停止することを要求する。{-# TERMINATING #-} を付けた関数に対しては、Agda の停止性検査を省略でき、停止しないことによる矛盾を無視できる。これにより、停止しない関数に Agda で型を付けることができるようになるが、本来ならば Agda の停止性検査をパスしない関数を強制的にパスさせていることになる。証明を記述するために Agda を使用しているという観点から、可能な限りこのようなオプションは取り除くことが理想的である。

**Type-in-Type**　現在 Agda は $Set$ 型にレベルをつけている。例えば、$Set : Set_1$ であるし、$Set_1 : Set_2$ である。レベルに上限はない。{-# –type-in-type #-} は、これらのレベルを無視し、$Set : Set$ を認めるオプションである。しかし、このオプションは矛盾を引き起こすことが知られている。Russell のパラドックスは、すべての集合の集合は正しい集合ではないということを意味する。もしすべての集合の集合 $U$ が存在すれば、その部分集合 $A = \{S | S \notin S\}$ について $A \notin A \Leftrightarrow A \in A$ という矛盾が導ける。同様に、Girard のパラドックスは、Agda の型理論において $Set : Set$ は矛盾しているということを示した。{-# –type-in-type #-} を使うことは $Set : Set$ による矛盾を無視するということなので、このようなオプションは取り除くことが好ましい。

### 6.2.2　トレースエフェクト

イベントトレースとは、プログラムの実行中に起こる副作用などのイベントの順序である。イベントトレースを管理することにより、安全なロック操作などの、イベントの順序が重要なプログラムの性質を静的に強制することができる。トレースエフェクト [20] は、エフェクト型システムとイベントトレースの検査を組み合わせたシステムであり、コンパイル時に自動でプログラムのイベントトレースを予測し、その性質を静的に検証できる。トレースエフェクトの言語モデルはラムダ計算に基づいている。

今後の課題として、トレースエフェクトを持つ言語の型安全インタプリタを実装するために、型付き抽象構文木にイベントトレースに関する証明項を追加する。実装においては、エフェクトの発生した履歴を保持しておく機構が必要になる。この機構の実現には、可変状態を持つ言語の型安全

インタプリタの実装が参考になると考えている。Poulsen ら [4] は、可変状態を持つ言語の型安全インタプリタを実装するためのライブラリを開発し、それを使った実装を示している。この実装では、型安全インタプリタは可変状態を保持するストアを持ち、ストアの長さが評価の前後で増加するという性質を証明に利用している。エフェクトの発生した履歴も評価の前後で増加する性質を持つため、トレースエフェクトとストアの実装には共通点があると考えられる。

### 6.2.3　コエフェクト計算

コエフェクト計算は、コエフェクトと呼ばれる様々なリソースの使用状況を分析するための計算体系である。例えば、線形型システム [13] はコエフェクト計算の一種である。線形型システムでは、変数の使用回数を追跡することにより、各変数がそのスコープ内で正確に一度だけ使用されることを保証する型システムである。これにより、変数が使用されない、もしくは二回以上使用されるプログラムを不正なプログラムとして検出することができる。

quantitative type [2] は、線形型より詳細に変数の使用回数を追跡できる型システムである。quantitative type は、線形型と依存型を組み合わせることで、プロトコルが要求する操作の順番、すなわちどの操作がいつ許可されるのかを型で表現する。これにより、メモリ安全性、プロトコルの正しさを推論できる。quantitative type theory では、各変数に、変数の使用回数を表すための多重度を割り当てる。多重度は任意の半環で定義できる。quantitative type を持つプログラミング言語に Idris2[6] が存在する。Idris2 では、各変数の束縛に 0, 1, $\omega$ の多重度を割り当てることで変数の使用回数を管理する。多重度 0 の変数は型の中で自由に扱えるが、それ以外の場所で使うことができず、多重度 1 の変数は正確に一度だけ使用され、多重度 $\omega$ の変数は無制限に使うことができる。

今後の課題として、quantitative type を持つ言語のための型安全インタプリタを実装するために、変数の多重性を再現するような証明項を追加する。Rouvoet ら [19] は、線形型を持つ言語の型安全インタプリタを実装するためのライブラリを開発し、それを使った実装方法を示している。線形型の性質は Idris2 の多重度 1 の変数に対応するため、実装において参考になる点があると期待できる。

# Bibliography

[1] Altenkirch, T. and Reus, B.: Monadic Presentations of Lambda Terms Using Generalized Inductive Types, CSL '99 (1999).

[2] Atkey, R.: Syntax and Semantics of Quantitative Type Theory, LICS '18 (2018).

[3] Augustsson, L. and Carlsson., M.: An exercise in dependent types: A well-typed interpreter, Workshop on Dependent Types in Programming (1999).

[4] Bach Poulsen, C., Rouvoet, A., Tolmach, A., Krebbers, R. and Visser, E.: Intrinsically-Typed Definitional Interpreters for Imperative Languages, *POPL* (2017).

[5] Bauer, Andrejand Pretnar, M.: An Effect System for Algebraic Effects and Handlers (2013).

[6] Brady, E.: Idris 2: Quantitative Type Theory in Practice (Artifact), *Dagstuhl Artifacts Series* (2021).

[7] Carette, J., Kiselyov, O. and Shan, C.-c.: Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages, *J. Funct. Program.* (2009).

[8] Chlipala, A.: Parametric Higher-Order Abstract Syntax for Mechanized Semantics, ICFP '08 (2008).

[9] Cong, Y., Ishio, C., Honda, K. and Asai, K.: A Functional Abstraction of Typed Invocation Contexts, *ArXiv* (2021).

[10] Danvy, O. and Filinski, A.: A Functional Abstraction of Typed Contexts (1989).

[11] Danvy, O. and Filinski, A.: Abstracting Control, LFP '90 (1990).

[12] Hillerström, D., Lindley, S., Atkey, R. and Sivaramakrishnan, K. C.: Continuation Passing Style for Effect Handlers, FSCD 2017 (2017).

[13] Morris, J. G.: The Best of Both Worlds: Linear Functional Programming without Compromise, *ICFP* (2016).

[14] Norell, U.: *Towards a practical programming language based on dependent type theory*, Vol. 32 (2007).

[15] Plotkin, G. D.: Call-by-name, call-by-value and the $\lambda$-calculus, *Theoretical computer science* (1975).

[16] Pretnar, M.: An Introduction to Algebraic Effects and Handlers. Invited Tutorial Paper, *Electron. Notes Theor. Comput. Sci.*.

[17] Reynolds, J. C.: Definitional Interpreters for Higher-Order Programming Languages, *Proceedings of the ACM Annual Conference - Volume 2* (1972).

[18] Rouvoet, A.: *Correct by Construction Language Implementations*, Delft University of Technology (2021).

[19] Rouvoet, A., Bach Poulsen, C., Krebbers, R. and Visser, E.: Intrinsically-Typed Definitional Interpreters for Linear, Session-Typed Languages, CPP 2020 (2020).

[20] Skalka, C., Smith, S. and Van horn, D.: Types and Trace Effects of Higher Order Programs (2008).

[21] Tsuyama, S., Cong, Y. and Masuhara, H.: Intrinsically-Typed Interpreters for Effectful and Coeffectful Languages, Presented at the first Workshop on the Implementation of Type Systems (WITS 2022) (2022).

[22] Xi, H., Chen, C. and Chen, G.: Guarded Recursive Datatype Constructors, POPL '03 (2003).