Minecraft 言語開発入門

単純型付きラムダ計算をデータパックにコンパイルする

intsuc

1　序論

データパックとは，Minecraft Java Edition[1] の拡張機能です．json ファイルによって宣言的にデータを定義したり，mcfunction ファイルによって Minecraft のワールドに作用する文 (コマンド) をまとめることができます．Minecraft 言語とは，Minecraft 上で実行されることを目的としたプログラミング言語です．本記事では，単純型付きラムダ計算と呼ばれるプログラミング言語を Minecraft 上で実行するためのコンパイラを実装する方法について解説します．

2　単純型付きラムダ計算

コンパイラのソース言語となる，単純型付きラムダ計算 ($\lambda^{\to}$)[2] を定義します．

2.1　構文

ブール型 (`bool`) を基本型とする，標準的な単純型付きラムダ計算です．本記事を通して，x は変数を表すメタ変数とします．

$$
\begin{array}{rrll}
\text{型環境} & \Gamma & ::= & \cdot & \text{空} \\
& & \mid & \Gamma, x : \tau & \text{拡張} \\
\\
\text{型} & \tau & ::= & \texttt{bool} & \text{ブール} \\
& & \mid & \tau \to \tau & \text{関数} \\
\\
\text{式} & e & ::= & x & \text{変数} \\
& & \mid & \texttt{true} & \text{真} \\
& & \mid & \texttt{false} & \text{偽} \\
& & \mid & \texttt{if } e \texttt{ then } e \texttt{ else } e & \text{条件} \\
& & \mid & x \to e & \text{抽象} \\
& & \mid & e\ e & \text{適用} \\
& & \mid & e : \tau & \text{型注釈}
\end{array}
$$

図 1　$\lambda^{\to}$ の構文

2.2　型付け規則

$\boxed{\Gamma \vdash e : \tau}$ 型環境 Γ の下で式 e は型 τ を持つ.

$$\frac{(x : \tau \in \Gamma)}{\Gamma \vdash x : \tau}\ \textsc{Var}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ \textsc{True} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}\ \textsc{False}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau}\ \textsc{If}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash x \to e : \tau_1 \to \tau_2}\ \textsc{Abs} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \textsc{App}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (e : \tau) : \tau}\ \textsc{Anno}$$

図 2　$\lambda^{\to}$ の型付け規則

2.3　操作的意味論

これ以上評価できない式 (値) の定義をします.

$$
\begin{array}{rrll}
\text{値} & v & ::= & \texttt{true} & \text{真} \\
& & \mid & \texttt{false} & \text{偽} \\
& & \mid & x \to e & \text{関数}
\end{array}
$$

図 3　$\lambda^{\to}$ の値

- `true` は定数なので，これ以上評価できません．
- `false` は定数なので，これ以上評価できません．
- $x \to e$ の内部の e は評価しないものとします．

代入の定義をします．

$$
\begin{aligned}
[e'/x]x &= e' && (1)\\
[e'/x]\texttt{true} &= \texttt{true} && (2)\\
[e'/x]\texttt{false} &= \texttt{false} && (3)\\
[e'/x](\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) &= \texttt{if } [e'/x]e_1 \texttt{ then } [e'/x]e_2 \texttt{ else } [e'/x]e_3 && (4)\\
[e'/x](x' \to e) &= x' \to e \qquad (x = x') && (5)\\
[e'/x](x' \to e) &= x' \to [e'/x]e \qquad (x \neq x') && (6)\\
[e'/x](e_1\ e_2) &= [e'/x]e_1\ [e'/x]e_2 && (7)\\
[e'/x](e : \tau) &= [e'/x]e : \tau && (8)
\end{aligned}
$$

図 4　$\lambda^{\to}$ の代入

基本的には部分式に対して帰納的に代入を行うだけです．代入される変数 x と $x' \to e$ が束縛する変数 x' が等しい場合 (5)，代入はその内部の e に対しては行われません．

これらを用いて $\lambda^{\to}$ の大ステップ操作的意味論を定義します．

$\boxed{e \Downarrow e'}$　式 e は式 e' へと評価される．

$$
\frac{}{\texttt{true} \Downarrow \texttt{true}}\ \textsc{True} \qquad \frac{}{\texttt{false} \Downarrow \texttt{false}}\ \textsc{False}
$$

$$
\frac{e_1 \Downarrow \texttt{true} \quad e_2 \Downarrow v_2}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Downarrow v_2}\ \textsc{If-True} \qquad \frac{e_1 \Downarrow \texttt{false} \quad e_3 \Downarrow v_3}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Downarrow v_3}\ \textsc{If-False}
$$

$$
\frac{}{x \to e \Downarrow x \to e}\ \textsc{Abs} \qquad \frac{e_1 \Downarrow x \to e \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{e_1\ e_2 \Downarrow v}\ \textsc{App}
$$

$$
\frac{e \Downarrow v}{e : \tau \Downarrow v}\ \textsc{Anno}
$$

図 5　$\lambda^{\to}$ の値呼びの大ステップ操作的意味論

- 規則 TRUE, FALSE, ABS において，帰結の左辺は値なので，右辺は左辺と同じになります．
- 規則 IF-TRUE において，条件の e_1 は `true` へと評価されているため，then 節の e_2 が評価され，else 節の e_3 は評価されません．
- 規則 IF-FALSE において，条件の e_1 は `false` へと評価されているため，else 節の e_3 が評価され，then 節の e_2 は評価されません．
- 規則 APP において，オペランドの e_2 が値の v_2 に評価されてから代入されているため，評価戦略は値呼び (call-by-value) となります．
- 規則 ANNO において，型注釈の τ は無視されます．

3 コンパイラフェイズ

$\lambda^{\to}$ のコンパイル処理の定義をします．コンパイルは基本的に，抽象構文木 (AST) 等の中間表現を段階 (phase) 毎に変換しながら，最終的なターゲットを生成する処理です．本記事では，入力を文字列 (Source) とし，5 つの段階 (構文解析，名前解決，型付け，脱関数化，データパック化) を経て，データパック (Datapack) を出力するコンパイラを考えます．

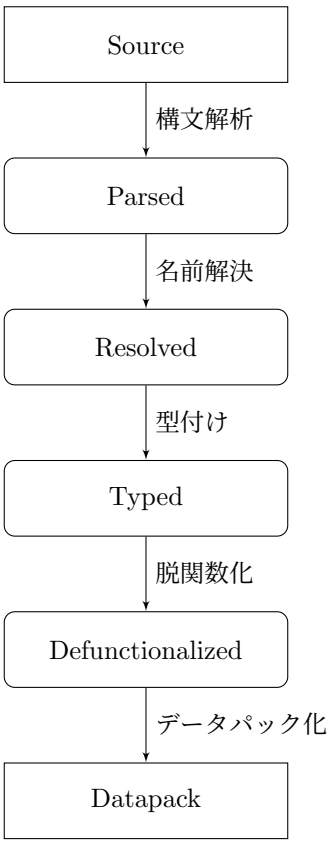

図 6　$\lambda^{\to}$ のコンパイラフェイズ

3.1 構文解析

構文解析では，文字列の入力 (Source) を Parsed に変換します．s を文字列を表すメタ変数とします．

型	τ	::=	`bool` ブール
		$\mid$	$\tau \to \tau$ 関数
式	e	::=	s 変数
		$\mid$	`true` 真
		$\mid$	`false` 偽
		$\mid$	`if` e `then` e `else` e 条件
		$\mid$	$s \to e$ 抽象
		$\mid$	$e\ e$ 適用
		$\mid$	$e : \tau$ 型注釈

図 7　構文解析後の AST(Parsed)

Parsed の時点では，変数はただの文字列でしかありません．例えば，$\lambda^{\to}$ の式

$$x \to x \to x$$

において，x と x は異なる変数ですが，Parsed の式

$$\texttt{"x"} \to \texttt{"x"} \to \texttt{"x"}$$

にはそのような区別はありません．これは次のフェイズの名前解決で解決されます．

なお，具体的な構文解析の方法については本旨から外れるため説明しません．

3.2 名前解決

名前解決では，同じ変数に同じ固有のシンボルを与え，Parsed を Resolved に変換します．

$$
\begin{array}{rcll}
\text{シンボル環境} \quad \Sigma & ::= & \cdot & \text{空} \\
& | & \Sigma, s \mapsto x & \text{拡張} \\
\\
\text{式} \quad e & ::= & x & \text{変数} \\
& | & \texttt{true} & \text{真} \\
& | & \texttt{false} & \text{偽} \\
& | & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e & \text{条件} \\
& | & x \to e & \text{抽象} \\
& | & e\ e & \text{適用} \\
& | & e : \tau & \text{型注釈}
\end{array}
$$

図 8　名前解決後の AST(Resolved)

拡張 $(\Sigma, s \mapsto x)$ は，既に Σ 内に $s \mapsto x'$ が存在すれば，それを $s \mapsto x$ で上書きした新しいシンボル環境を表します．型は Parsed のものと同様です．x `fresh` は x がフレッシュ (任意の他のシンボルと区別できる) であることを表します．

$\boxed{\Sigma \vdash e \rightsquigarrow e'}$ シンボル環境 Σ の下で Parsed の式 e は Resolved の式 e' へと名前解決される．

$$\frac{s \mapsto x \in \Sigma}{\Sigma \vdash s \rightsquigarrow x}\ \text{R-Var}$$

$$\frac{}{\Sigma \vdash \texttt{true} \rightsquigarrow \texttt{true}}\ \text{R-True} \qquad \frac{}{\Sigma \vdash \texttt{false} \rightsquigarrow \texttt{false}}\ \text{R-False}$$

$$\frac{\Sigma \vdash e_1 \rightsquigarrow e_1' \quad \Sigma \vdash e_2 \rightsquigarrow e_2' \quad \Sigma \vdash e_3 \rightsquigarrow e_3'}{\Sigma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rightsquigarrow \texttt{if}\ e_1'\ \texttt{then}\ e_2'\ \texttt{else}\ e_3'}\ \text{R-If}$$

$$\frac{x\ \texttt{fresh} \quad \Sigma, s \mapsto x \vdash e \rightsquigarrow e'}{\Sigma \vdash s \to e \rightsquigarrow x \to e'}\ \text{R-Abs} \qquad \frac{\Sigma \vdash e_1 \rightsquigarrow e_1' \quad \Sigma \vdash e_2 \rightsquigarrow e_2'}{\Sigma \vdash e_1\ e_2 \rightsquigarrow e_1'\ e_2'}\ \text{R-App}$$

$$\frac{\Sigma \vdash e \rightsquigarrow e'}{\Sigma \vdash e : \tau \rightsquigarrow e' : \tau}\ \text{R-Anno}$$

図 9　$\lambda^{\to}$ の名前解決規則

基本的には部分式を帰納的に名前解決するだけです．

- 規則 R-Var では，シンボル環境 Σ から文字列 s に対応する変数 x を探し，s を x に変換します．
- 規則 R-Abs では，フレッシュ x を生成し，$s \mapsto x$ で拡張したシンボル環境下で内部の式 e の名前解決を行います．

3.3 型付け

型付けでは，Resolved が正しく型付けされるか確認し，されるならば Resolved を Typed に変換します．

型環境	Γ	::= $\cdot$	空
		$\mid \Gamma, x : \tau$	拡張
式	e	::= x	変数
		$\mid$ `true`	真
		$\mid$ `false`	偽
		$\mid$ `if` e `then` e `else` e	条件
		$\mid x \to e$	抽象
		$\mid e\ e$	適用

図 10　型付け後の AST(Typed)

型は Resolved のものと同様です．型注釈 $(e : \tau)$ はこれ以降不要なので無くなっています．

型付けのアルゴリズムを得るために，双方向化のレシピ [4] に従って図 2 の型付け規則を双方向化 [3] します．このレシピはあくまで指標なので，遵守する必要はありません．双方向型付けでは，型付け関係 $\boxed{\Gamma \vdash e : \tau}$ は以下の 2 つのモード:

- $\boxed{\Gamma \vdash e \Leftarrow \tau}$ チェック (checking): 環境 Γ の下で式 e の型と型 τ は一致する．
- $\boxed{\Gamma \vdash e \Rightarrow \tau}$ 合成 (synthesis): 環境 Γ の下で式 e は型 τ を合成する．

に分けられます．それぞれ以下のような関数であると考えられます．

- チェック: 型環境 Γ，式 e，型 τ を入力として，正しく型付け可能かを判定する関数．
- 合成: 型環境 Γ，式 e を入力として，型 τ を返す関数．

$\boxed{\Gamma \vdash e \Leftarrow \tau}$ 環境 Γ の下で Resolved の式 e の型と型 τ は一致する．

$\boxed{\Gamma \vdash e \Rightarrow \tau}$ 環境 Γ の下で Resolved の式 e は型 τ を合成する．

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

$$\frac{}{\Gamma \vdash \texttt{true} \Leftarrow \texttt{bool}}\ \text{T-True}\Leftarrow \qquad \frac{}{\Gamma \vdash \texttt{true} \Rightarrow \texttt{bool}}\ \text{T-True}\Rightarrow$$

$$\frac{}{\Gamma \vdash \texttt{false} \Leftarrow \texttt{bool}}\ \text{T-False}\Leftarrow \qquad \frac{}{\Gamma \vdash \texttt{false} \Rightarrow \texttt{bool}}\ \text{T-False}\Rightarrow$$

$$\frac{\Gamma \vdash e_1 \Leftarrow \texttt{bool} \quad \Gamma \vdash e_2 \Rightarrow \tau \quad \Gamma \vdash e_3 \Leftarrow \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow \tau}\ \text{T-If}\Rightarrow$$

$$\frac{\Gamma, x : \tau_1 \vdash e \Leftarrow \tau_2}{\Gamma \vdash x \to e \Leftarrow \tau_1 \to \tau_2}\ \text{T-Abs}\Leftarrow \qquad \frac{\Gamma \vdash e_1 \Rightarrow \tau \quad \tau = \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 \Leftarrow \tau_1}{\Gamma \vdash e_1\ e_2 \Rightarrow \tau_2}\ \text{T-App}\Rightarrow$$

$$\frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash (e : \tau) \Rightarrow \tau}\ \text{T-Anno}\Rightarrow \qquad \frac{\Gamma \vdash e \Rightarrow \tau \quad \tau = \tau'}{\Gamma \vdash e \Leftarrow \tau'}\ \text{T-Sub}\Leftarrow$$

図 11　$\lambda^{\to}$ の双方向型付け規則

- 規則 T-Var⇒:
  1. 規則 T-Var は導入規則でも除去規則でもないため，主要判断は存在しません．
  2. 帰結の判断を双方向化します．型 τ は型環境 Γ に存在するので合成にします．

     $$\Gamma \vdash x \Rightarrow \tau$$

- 規則 T-True⇐:
  1. 規則 T-True は導入規則なので，主要判断は帰結

     $$\Gamma \vdash \texttt{true} : \texttt{bool}$$

     となります．
  2. 主要判断を双方向化します．導入規則なのでチェックにします．

     $$\Gamma \vdash \texttt{true} \Leftarrow \texttt{bool}$$

  3. 他の判断を双方向化します．規則 T-True は公理なので，他の判断はありません．
- 規則 T-True⇒:
  1. 規則 T-True は導入規則ですが，利便性のために合成規則を作ります．
  2. 帰結の判断を双方向化します．`true` の型が `bool` であることは事前に分かっているため合成にします．

     $$\Gamma \vdash \texttt{true} \Rightarrow \texttt{bool}$$

  3. 他の判断を双方向化します．規則 T-True は公理なので，他の判断はありません．
- 規則 T-False⇐:

1. 規則 T-False は導入規則なので，主要判断は帰結

$$\Gamma \vdash \texttt{false} : \texttt{bool}$$

となります．

2. 主要判断を双方向化します．導入規則なのでチェックにします．

$$\Gamma \vdash \texttt{false} \Leftarrow \texttt{bool}$$

3. 他の判断を双方向化します．規則 T-False は公理なので，他の判断はありません．

- 規則 T-False⇒:
  1. 規則 T-False は導入規則ですが，利便性のために合成規則を作ります．
  2. 帰結の判断を双方向化します．`false` の型が `bool` であることは事前に分かっているため合成にします．

$$\Gamma \vdash \texttt{false} \Rightarrow \texttt{bool}$$

  3. 他の判断を双方向化します．規則 T-False は公理なので，他の判断はありません．
- 規則 T-If⇒:
  1. 規則 T-If は除去規則なので，主要判断は前提

$$\Gamma \vdash e_2 : \tau$$

  となります．(レシピ通りならば最初の前提 $\Gamma \vdash e_1 : \texttt{bool}$ になるのですが，`bool` と分かっているものを合成する必要は無いため，2 つ目の前提にしました．)
  2. 主要判断を双方向化します．除去規則なので合成にします．

$$\Gamma \vdash e_2 \Rightarrow \tau$$

  3. 他の判断を双方向化します．
     (a) 型 `bool` は事前に分かっているためチェックにします．

$$\Gamma \vdash e_1 \Leftarrow \texttt{bool}$$

     (b) 型 τ は $\Gamma \vdash e_2 \Rightarrow \tau$ で得られているためチェックにします．

$$\Gamma \vdash e_3 \Leftarrow \tau$$

     (c) 型 τ は $\Gamma \vdash e_2 \Rightarrow \tau$ で得られているため合成にします．

$$\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow \tau$$

- 規則 T-Abs⇐:
  1. 規則 T-Abs は導入規則なので，主要判断は帰結

$$\Gamma \vdash x \to e : \tau_1 \to \tau_2$$

  となります．
  2. 主要判断を双方向化します．導入規則なのでチェックにします．

$$\Gamma \vdash x \to e \Leftarrow \tau_1 \to \tau_2$$

3. 他の判断を双方向化します．
   (a) 型 τ_1，τ_2 は $\Gamma \vdash x \to e \Leftarrow \tau_1 \to \tau_2$ で得られているのでチェックにします．

$$\Gamma, x : \tau_1 \vdash e \Leftarrow \tau_2$$

- 規則 T-App⇒:
  1. 規則 T-App は除去規則なので，主要判断は最初の前提

$$\Gamma \vdash e_1 : \tau_1 \to \tau_2$$

     となります．
  2. 主要判断を双方向化します．除去規則なので合成にします．

$$\Gamma \vdash e_1 \Rightarrow \tau_1 \to \tau_2$$

  3. 他の判断を双方向化します．
     (a) 型 τ_1 は $\Gamma \vdash e_1 \Rightarrow \tau_1 \to \tau_2$ で得られているためチェックにします．

$$\Gamma \vdash e_2 \Leftarrow \tau_1$$

     (b) 型 τ_2 は $\Gamma \vdash e_1 \Rightarrow \tau_1 \to \tau_2$ で得られているため合成にします．

$$\Gamma \vdash e_1\ e_2 \Rightarrow \tau_2$$

- 規則 T-Anno⇒: 規則 T-Anno⇒ はモードを合成からチェックへ変更するための規則です．
- 規則 T-Sub⇐: 規則 T-Sub⇐ はモードをチェックから合成へ変更するための規則です．

3.4 脱関数化

脱関数化 [5][6] では，第一級オブジェクト [7] としての関数 $x \to e$ を含む Typed を，含まない Defunctionalized に変換します．データパックにはそのような関数を直接表現する方法は無いので，このフェイズは重要です．X を x を要素とする集合とします．

定義環境	Δ	$::=$	$\cdot$ 空
		$\mid$	$\Delta, \langle x; X \rangle = e$ 拡張
式	e	$::=$	x 変数
		$\mid$	`true` 真
		$\mid$	`false` 偽
		$\mid$	`if` e `then` e `else` e 条件
		$\mid$	$\langle x; X \rangle$ クロージャ
		$\mid$	$e(e)$ 呼び出し

図 12　脱関数化後の AST(Defunctionalized)

Typed にあった抽象 $x \to e$ と適用 $e\ e$ が無くなり，それぞれクロージャ $\langle x; X\rangle$ と呼び出し $e(e)$ になっています．クロージャ $\langle x; X\rangle$ の x はクロージャを識別するためのタグ，X はクロージャが捕獲した変数 (環境) を表します．定義環境の $\langle x; X\rangle = e$ は，名前が x，引数が X，本体が式 e の関数定義を表します．

Typed の式 e 内の自由変数の集合を返す関数 fv を定義します．クロージャが捕獲する変数を得るために必要になります．

$$
\begin{aligned}
fv(x) &= x \\
fv(\texttt{true}) &= \{\} \\
fv(\texttt{false}) &= \{\} \\
fv(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) &= fv(e_1) \cup fv(e_2) \cup fv(e_3) \\
fv(x \to e) &= fv(e) \setminus x \\
fv(e_1\ e_2) &= fv(e_1) \cup fv(e_2)
\end{aligned}
$$

図 13　関数 fv の定義

$\boxed{e \rightsquigarrow e' \dashv \Delta}$ Typed の式 e は Defunctionalized の式 e' へと脱関数化され，定義環境 Δ を出力する．

$$\frac{}{x \rightsquigarrow x \dashv \cdot}\ \text{D-Var}$$

$$\frac{}{\texttt{true} \rightsquigarrow \texttt{true} \dashv \cdot}\ \text{D-True} \qquad \frac{}{\texttt{false} \rightsquigarrow \texttt{false} \dashv \cdot}\ \text{D-False}$$

$$\frac{e_1 \rightsquigarrow e_1' \dashv \Delta_1 \quad e_2 \rightsquigarrow e_2' \dashv \Delta_2 \quad e_3 \rightsquigarrow e_3' \dashv \Delta_3}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rightsquigarrow \texttt{if}\ e_1'\ \texttt{then}\ e_2'\ \texttt{else}\ e_3' \dashv \Delta_1, \Delta_2, \Delta_3}\ \text{D-If}$$

$$\frac{e \rightsquigarrow e' \dashv \Delta}{x \to e \rightsquigarrow \langle x; fv(x \to e)\rangle \dashv \Delta, \langle x; fv(x \to e)\rangle = e'}\ \text{D-Clos} \qquad \frac{e_1 \rightsquigarrow e_1' \dashv \Delta_1 \quad e_2 \rightsquigarrow e_2' \dashv \Delta_2}{e_1\ e_2 \rightsquigarrow e_1'(e_2') \dashv \Delta_1, \Delta_2}\ \text{D-Call}$$

図 14　$\lambda^{\to}$ の脱関数化規則

基本的には部分式を帰納的に脱関数化するだけです．

- 規則 D-Clos:
  1. 抽象 $x \to e$ の x をクロージャのタグとし，$x \to e$ 内の自由変数 ($fv(x \to e)$) をクロージャの環境とします．
  2. 名前が x，引数が $fv(x \to e)$，本体が e を脱関数化した e' の関数定義を追加して出力します．

3.5 データパック化

データパック化では, 一階関数的なプログラムを表す Defunctionalized を, ついにデータパック (Datapack) に変換します．Minecraft 上で効率良く実行できるように，単一のグローバルスタックを用いてメモリ管理を行います．スタックを用いたメモリ管理を行うには，ブロック構造によって定められる変数のスコープと，変数の生存期間が一致する必要があります [8]．抽象 $x \to e$ をブロックとして，このブロックから出る際に変数 x を解放するために，**ブロック内のクロージャによる x の捕獲はコピーによって行われる**ようにします．

f を変数を表すメタ変数とします．

シンボル環境	Σ	$::=$ $\cdot$	空
		$\mid$ Σ, x	拡張
定義環境	Δ	$::=$ $\cdot$	空
		$\mid$ $\Delta, f \mapsto C$	拡張
コマンド	C	$::=$ $\cdot$	空
		$\mid$ C, s	拡張

図 15　データパック (Datapack)

定義環境の拡張 $(\Delta, f \mapsto C)$ は，既に Δ 内に $f \mapsto C'$ が存在する場合，それを $f \mapsto C$ で上書きした定義環境を表します．Δ 内に $f \mapsto C$ が存在する場合，$\Delta(f)$ で C を表します．

データパック化のアルゴリズムを定義します．

$\boxed{\Sigma; f \vdash e \dashv \Delta}$ シンボル環境 Σ，関数環境 f の下で Defunctionalized の式 e はデータパック化され，定義環境 Δ を出力する．

$$\frac{(x \in \Sigma)}{\Sigma; f \vdash x \dashv f \mapsto \texttt{data modify storage \_ stack append from storage \_ stack[-2].}x}\ \textsc{Var-Data}$$

$$\frac{(x \notin \Sigma)}{\Sigma; f \vdash x \dashv f \mapsto \texttt{data modify storage \_ stack append from storage \_ stack[-1]}}\ \textsc{Var-Arg}$$

$$\frac{}{\Sigma; f \vdash \texttt{true} \dashv f \mapsto \texttt{data modify storage \_ stack append value \{\_:true\}}}\ \textsc{True}$$

$$\frac{}{\Sigma; f \vdash \texttt{false} \dashv f \mapsto \texttt{data modify storage \_ stack append value \{\_:false\}}}\ \textsc{False}$$

$$\frac{f_2\ \texttt{fresh} \quad f_3\ \texttt{fresh} \quad \Sigma; f \vdash e_1 \dashv \Delta_1 \quad \Sigma; f_2 \vdash e_2 \dashv \Delta_2 \quad \Sigma; f_3 \vdash e_3 \dashv \Delta_3}{\begin{array}{l}\Sigma; f \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \dashv \\ \quad \Delta_1, \Delta_2, \Delta_3, \\ \quad\quad f \mapsto (\Delta_1(f), \\ \quad\quad\quad \texttt{data modify storage \_ \_ set from storage \_ stack[-1].\_}, \\ \quad\quad\quad \texttt{data remove storage \_ stack[-1]}, \\ \quad\quad\quad \texttt{execute if data storage \_ \{\_:true\} run function } f_2, \\ \quad\quad\quad \texttt{execute if data storage \_ \{\_:false\} run function } f_3) \\ \quad\quad f_2 \mapsto (\Delta_2(f_2), \\ \quad\quad\quad \texttt{data modify storage \_ \_ set value true})\end{array}}\ \textsc{If}$$

$$\frac{}{\begin{array}{l}\Sigma; f \vdash \langle x; X \rangle \dashv \\ \quad f \mapsto (\texttt{data modify storage \_ stack append value \{\_:}x\texttt{\}} \\ \quad\quad (\texttt{data modify storage \_ stack[-1].}x'\ \texttt{set from storage \_ stack[-3].}x')_{\forall x' \in X \cap \Sigma}, \\ \quad\quad (\texttt{data modify storage \_ stack[-1].}x'\ \texttt{set from storage \_ stack[-2]})_{\forall x' \in X \setminus \Sigma})\end{array}}\ \textsc{Clos}$$

$$\frac{\Sigma; f \vdash e_1 \dashv \Delta_1 \quad \Sigma; f \vdash e_2 \dashv \Delta_2}{\begin{array}{l}\Sigma; f \vdash e_1(e_2) \dashv \\ \quad \Delta_1, \Delta_2, \\ \quad\quad f \mapsto (\Delta_1(f), \\ \quad\quad\quad \Delta_2(f), \\ \quad\quad\quad \texttt{function apply})\end{array}}\ \textsc{Call}$$

図 16　$\lambda^{\to}$ のデータパック化規則

Defunctionalized の出力環境 Δ の名前の部分のみを含む集合 F に対して，

```
data modify storage _ _ set from storage _ stack[-2]._,
(execute if data storage _ {_:f} run function f)_{∀f∈F},
data remove storage _ stack[-2],
data remove storage _ stack[-2]
```

図 17　`apply` の実装

- 規則 Call: オペレータ e_1 とオペランド e_2 を順にデータパック化します．これらのコマンドが実行された時点で，スタックの先頭には e_2，その 1 つ下には e_1 が乗るので，`function apply` によって e_1 を e_2 に適用します．
  `apply` では，まずオペレータを識別するためのタグ_をレジスタ_に保存し，条件分岐によって目的の関数を呼び出します．呼び出した関数から帰ってくると，スタックの先頭にはその戻り値が乗っているので，その 1 つ下のオペランドと 2 つ下のオペレータを解放します．
- 規則 Var-Data: Σ はオペレータのクロージャの環境を表します．その中に x が含まれるということは，x は現在のオペレータの環境の要素ということになります．よって，オペランドの環境から x を複製し，スタックに乗せます．
- 規則 Var-Arg: Σ はオペレータのクロージャの環境を表します．その中に x が含まれないということは，x は現在のオペランドということになります．よって，オペランドを複製し，スタックに乗せます．
- 規則 True: {_:true} を複製し，スタックに乗せます．`true` でないのは，スタックを単一の CompoundTag の ListTag で表現するためです．
- 規則 False: {_:false} を複製し，スタックに乗せます．`false` でないのは，スタックを単一の CompoundTag の ListTag で表現するためです．
- 規則 If: 条件 e_1 と then 節 e_2 と else 節 e_3 を順にデータパック化します．関数 f の末尾に条件分岐を行うためのコマンドを追加します．e_1 を評価すると，スタックの先頭にその結果が乗っているので，レジスタ_に保存し，以降不要なのでスタックの先頭から除きます．レジスタ_の内容によって，e_2 もしくは e_3 を実行します．レジスタ_の内容が書き換わることで then 節から帰ってきた後に else 節に入らないように，then 節である関数 f_2 の末尾にレジスタ_を `true` にするコマンドを追加します．
- 規則 Clos: タグ x のクロージャを複製し，スタックに乗せます．規則 Var-Data, Var-Arg の要領で，変数を複製して捕獲します．スタックの先頭にはクロージャが乗っているので，オフセットが 1 ずれています．

最後に，Defunctionalized の式 e に対応する関数の最初に，

```
data modify storage _ stack set value []
```

によってスタックの初期化を行い，Defunctionalized の出力環境 Δ のそれぞれの要素 $\langle x; X\rangle = e$ に対して，e に対応する関数の最後に

```
data modify storage _ _ set value x
```

を追加すれば全ての関数生成が完了します (規則 If と同様に，どの関数から帰ってきたか判別するために必要です)．

4 結論

Minecraft 言語を開発する際の入門として，$\lambda^{\to}$ をデータパックにコンパイルする方法について解説しました．なるべく self-contained にしようとしたのですが，前提知識が要求される箇所があるかもしれません．

$\lambda^{\to}$ は純粋な言語なので，Minecraft のワールドに作用することがありません (ストレージを除く)．例えば，ある座標のブロックを調べたり，ある座標にブロックを置いたりできないのです．本記事では扱いませんでしたが，実用的な Minecraft 言語ではこれらの機能は充実しているべきでしょう．

なお，本記事で解説したコンパイラの実装は https://github.com/intsuc/stlc にあります．Scala3 によって実装されていますが，本記事の解説は特定のプログラミング言語に依存せずに，かつできるだけ宣言的に書かれているため，他のプログラミング言語でも比較的容易に実装できるはずです．

本記事が Minecraft 言語へのさらなる関心と発展に寄与すれば幸いです．

参考文献

[1] Minecraft Java Edition. https://www.minecraft.net/en-us/store/minecraft-java-edition

[2] Alonzo Church. A Formulation of the Simple Theory of Types. https://doi.org/10.2307/2266170

[3] Jana Dunfield, Neel Krishnaswami. Bidirectional Typing. https://www.cl.cam.ac.uk/~nk480/bidir-survey.pdf

[4] Joshua Dunfield, Frank Pfenning. Tridirectional Typechecking. https://doi.org/10.1145/964001.964025

[5] John C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. https://doi.org/10.1145/800194.805852

[6] François Pottier, Nadji Gauthier. Polymorphic Typed Defunctionalization and Concretization. https://doi.org/10.1007/s10990-006-8611-7

[7] Christopher Strachey. Fundamental Concepts in Programming Languages. https://doi.org/10.1023/A:1010000313106

[8] Anindya Banerjee, David A. Schmidt. Stackability in the Simply-Typed Call-By-Value Lambda Calculus. https://doi.org/10.1016/S0167-6423(96)00040-8