慶應義塾大学学術情報リポジトリ
Keio Associated Repository of Academic resouces

Title	関数型プログラミング言語の実装手法
Sub Title	
Author	柏木, 力哉
Publisher	慶應義塾大学AI・高度プログラミングコンソーシアム
Publication year	2023
Jtitle	AICカンファレンス予稿集 (2023.) ,p.13- 18
JaLC DOI	
Abstract	本論文は、関数型プログラミング言語をラムダ計算をベースとしたコア言語に翻訳する過程で必要な実装手法を説明する。オフサイドルールや演算子の結合性解決といった構文解析上のテクニックから、型検査やパターンマッチの翻訳などまで、一連の過程を隈なく記述する。本研究では、明示的に型付けされたコア言語を出力するため、Damas-Milner型システムを拡張した、型抽象と型適用を挿入する型推論アルゴリズムを考案した。加えて、任意階の多相やモジュールシステムによるデータ抽象など、現代的な関数型言語には必須である、高度な型システムの実装も明らかにした。
Notes	会議名 : AICカンファレンス2023 開催地 : 慶應義塾大学日吉キャンパス 日時 : 2023年3月4日 第1章研究論文 論文-2
Genre	Conference Paper
URL	https://koara.lib.keio.ac.jp/xoonips/modules/xoonips/detail.php?koara_id=KO11003001-20230304-0013

慶應義塾大学学術情報リポジトリ(KOARA)に掲載されているコンテンツの著作権は、それぞれの著作者、学会または出版社/発行者に帰属し、その権利は著作権法によって保護されています。引用にあたっては、著作権法を遵守してご利用ください。

The copyrights of content available on the KeiO Associated Repository of Academic resources (KOARA) belong to the respective authors, academic societies, or publishers/issuers, and these rights are protected by the Japanese Copyright Act. When quoting the content, please follow the Japanese copyright act.

関数型プログラミング言語の実装手法

柏木力哉
慶應義塾大学理工部電気情報学科

Abstract: 本論文は、関数型プログラミング言語をラムダ計算をベースとしたコア言語に翻訳する過程で必要な実装手法を説明する。オフサイドルールや演算子の結合性解決といった構文解析上のテクニックから、型検査やパターンマッチの翻訳などまで、一連の過程を限なく記述する。本研究では、明示的に型付けされたコア言語を出力するため、Damas-Milner 型システムを拡張した、型抽象と型適用を挿入する型推論アルゴリズムを考案した。加えて、任意階の多相やモジュールシステムによるデータ抽象など、現代的な関数型言語には必須である、高度な型システムの実装も明らかにした。

Keywords: language design, functional programming languages, compiler, lambda calculus, type systems

1. 序論

関数型プログラミング言語の実装は、Lisp や ML の開発を通して長い間研究されてきた。今日では、Haskell や Agda、Idris など、より強力な型システムを備えた言語の開発が進んでおり、定理証明や型駆動開発の期待が高まっている。しかし、Lisp や ML が多くの開発者に言語機能の実験の場を与えてきた一方で、現代的な関数型言語では型システムの複雑さのために、開発者による自作は珍しくなり、実装法に関する議論も不十分となっている。そこで本論文では、純粋関数型言語「Plato[*1]」の開発における筆者の経験を通して、関数型言語の仕様に関する議論とその実装手法を提示する。

2. 概要

本稿では Fig. 1 に示す Haskell に似た構文のプログラムを構文解析し、型付きラムダ計算をベースとした**コア言語**に翻訳する過程で必要な実装手法を説明する。これは ML 系や Haskell など多くの関数型言語で取られているコンパイル手法であり、抽象構文木をより小さなコア言語に翻訳することで、言語仕様の拡張性や、フロントエンドとバックエンドの独立性を保つことが可能となる。そのため、コア言語に翻訳した後の最適化やマシン言語への翻訳は本研究とは独立しており、議論の対象外とする。

```
1  -- Main module
2  import Bool
3
4  data Nat = Zero | Succ Nat
5  data List a = Nil | a :: List a
6  infixr 5 ::
7
8  iseven Zero = True
9  iseven (Succ n) = not (isodd n)
10
11 isodd Zero = False
12 isodd (Succ n) = not (iseven n)
13
14 filter : {a} (a -> Bool) -> List a -> List a
15 filter f Nil = Nil
16 filter f (x :: xs) = case f x of
17     True -> x :: filter f xs
18     False -> filter f xs
19
20 filter iseven (Zero :: Succ Zero :: Succ (Succ Zero))
```

Fig. 1 Sample code of the Source Language

このサンプルコードを記述する言語を**ソース言語**と本稿では呼ぶ。ソース言語は、再帰型、型演算子、モジュールシステム、高階関数、再帰関数など、関数型言語における重要な言語仕様を多く含んでいる。言語仕様は正確に定まっているわけではなく、実装の議論を通して吟味していくが、以下にサンプルコードを例に、大まかな仕様を示す。

- `import` 文で他のモジュールのコンポーネントをインポートする。(2 行目)
- `data` $id = constructors$ で代数的データ型を宣言する。(4、5 行目)
- `(infix/infixl/infixr)` $int\ op$ で中置演算子の結合性を宣言する。(6 行目)
- 関数宣言は型署名と関数本体からなり、型署名は省略できる。(8、9 行目など)
- 型署名において、型変数は明示され、$\{\bar{id}\}$ で宣言される。(14 行目)
- コード中に生で置かれた式は評価され、結果をコンソールに出力する。(20 行目)

コンパイルのフローは Fig. 2 の通り。各ノードに対応する節番号を付している。

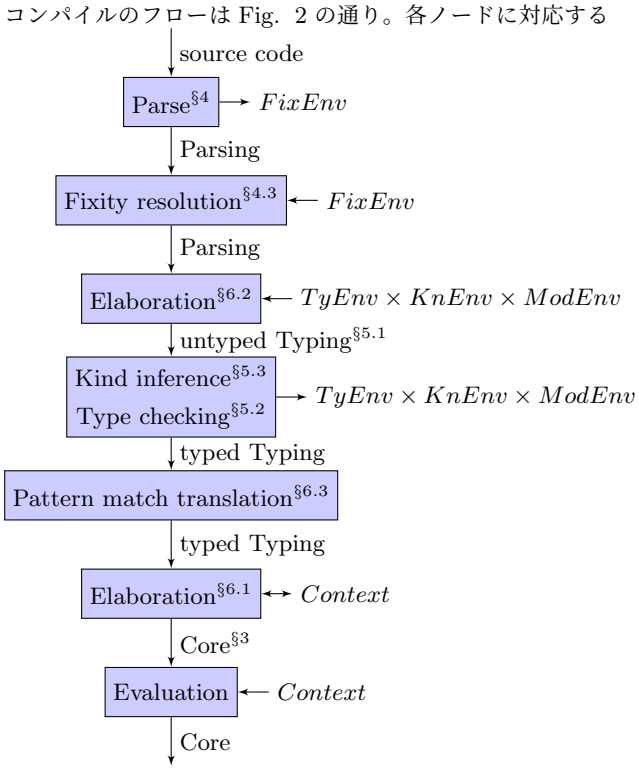

Fig. 2 Compilation pipeline

[*1] https://github.com/ksrky/Plato

3. コア言語

コア言語の構文を Fig. 3 に示す。コア言語は System F_ω

name	$x,\ X$	
label	l	
term	$e ::=$	$x \mid e\ e \mid \lambda x\colon t.\ e \mid e\ t \mid \Lambda X\colon k.\ e \mid$
		$\texttt{let}\ x\texttt{=}e\ \texttt{in}\ e \mid \texttt{fix}\ e \mid e.l \mid \{\overline{l\texttt{=}e}\} \mid$
		$\texttt{case}\ e\colon t\ \texttt{of}\{\overline{l \to e}\} \mid \texttt{pack}\ \{T, t\}\ \texttt{as}\ T \mid$
		$\texttt{unpack}\ \{X, x\}\texttt{=}t\ \texttt{in}\ t \mid \langle l,\ \overline{t}\rangle\ \texttt{as}\ t \mid$
type	$t ::=$	$X \mid t \to t \mid \forall X\colon k.\ t \mid \exists X\colon k.\ t \mid t\ t \mid$
		$\lambda X\colon k.\ t \mid \{\overline{l\colon t}\} \mid \langle \overline{l,\ t}\rangle$
kind	$k ::=$	$* \mid k \to k$
context	$\Gamma ::=$	$\emptyset \mid \Gamma,\ x\colon t \mid \Gamma,\ x\colon t\texttt{=}e \mid \Gamma,\ X\colon k \mid$
		$\Gamma,\ X\colon k\texttt{=}t$

Fig. 3 Syntax of core language

をベースとしており、不動点コンビネータやレコードなども備えている。変数の名前 (name) とタグやレコードのフィールドのラベル (label) には定義が書かれていないが、これは実装上の問題だからである。name と label は以下の性質を満たすようなデータであれば何を用いても良い。

- name はラムダ式や let 式などで束縛された環境中で一意に参照できる。
- label は同一モジュール内で一意に参照できる。

name には識別子を用いてもよいし、de Bruijn インデックスを用いる方法もある。詳しい実現方法については [14] が参考になる。

4. 構文解析

4.1 関数型プログラミング言語の構文

プログラミング言語の構文は、できる限り少ないルールで複数の意味を表現できるものであることが望ましい。これは開発者が実装しやすくなるだけでなく、プログラマにとっても覚えるべき構文を減らすというメリットがある。また、予約語は最小限にし、構文にとって自明なキーワードやシンボルはできる限り取り除くべきである。関数型言語の構文は特に以上のような設計思想が反映されており、ソース言語の構文もその例外ではない。

Haskell の Language Report[8] や Standard ML の定義 [10] にあるように、関数型言語の構文は多くの場合、文脈自由文法で記述される。LALR(1) パーサーを用いると、これらの定義をそのままの形で実装に落とし込める。詳細については [2] を参照せよ。

4.2 オフサイドルール

ソース言語の構文解析において、実装が最も複雑であるのがオフサイドルールである。C や Java のような言語では、コード中のブロックはブレースで囲われ、文はセミコロンによって区切られるが、オフサイドルールを持つ言語ではブロックはインデントによって、文は改行によって表現される。オフサイドルールの実装は字句解析器を煩雑にするが、Fig. 1 に示されている通り、それに見合うだけ構文は簡潔となる。

Haskell のオフサイドルールの正確な定義を Fig. 4 に示す ([8], §10.3)。Haskell では、トークン列にブレースやセミコロンを挿入することによって、文やブロックを明示している。

$L\ (<n>{:}\ ts)\ (m : ms)$	$=$	$;\ :\ (L\ ts\ (m : ms))$	if $m = n$
	$=$	$\}\ :\ (L\ (<n>{:}\ ts)\ ms)$	if $n < m$
$L\ (<n>{:}\ ts)\ ms$	$=$	$L\ ts\ ms$	
$L\ (\{n\} : ts)\ (m : ms)$	$=$	$\{\ :\ (L\ ts\ (n : m : ms))$	if $n > m$ (*Note* 1)
$L\ (\{n\} : ts)\ []$	$=$	$\{\ :\ (L\ ts\ [n])$	if $n > 0$ (*Note* 1)
$L\ (\{n\} : ts)\ ms$	$=$	$\{\ :\ \}\ :\ (L\ (<n>{:}\ ts)\ ms)$	(*Note* 2)
$L\ (\} : ts)\ (0 : ms)$	$=$	$\}\ :\ (L\ ts\ ms)$	(*Note* 3)
$L\ (\} : ts)\ ms$	$=$	parse-error	(*Note* 3)
$L\ (\{ : ts)\ ms$	$=$	$\{\ :\ (L\ ts\ (0 : ms))$	(*Note* 4)
$L\ (t : ts)\ (m : ms)$	$=$	$\}\ :\ (L\ (t : ts)\ ms)$	if $m' = 0$ and parse-error(t) (*Note* 5)
$L\ (t : ts)\ ms$	$=$	$t\ :\ (L\ ts\ ms)$	
$L\ []\ []$	$=$	$[]$	
$L\ []\ (m : ms)$	$=$	$\}\ :\ L\ []\ ms$	if $m \neq 0$ (*Note* 6)

Fig. 4 Haskell's definition of offside rule

この定義は非常にシンプルであるが、下から 4 行目のルールが構文解析エラーに依存するため、単純に字句解析の後処理として実装することはできず、実装は字句解析、構文解析の両方にまたがる。すなわち、Fig. 4 の定義から、直接実装を導くことはできない。他のオフサイドルールを持つ言語でも同様に、オフサイドルールは構文解析とは対照的に、アドホックな実装となっている。これに対して、[1] は、レイアウトルールのアルゴリズムを文脈自由文法の中で扱うことを試みた数少ない論文の一つである。

4.3 演算子の結合性解決

OCaml や Haskell などの関数型言語では中置演算子をプログラム中で定義することができる。このような特徴を構文に含めるのは簡単で、中置演算子を 2 つの引数をとる関数とみなし、演算子の結合性を宣言するための構文を追加すれば良い。中置演算子の結合性とは、右結合、左結合、結合性なしのいずれかと、その結合性の強さである。しかし、結合性を含む中置演算子の構文解析をするには、パーサーを書き換える必要があり、もし演算子が宣言されるたびにパーサーを動的に変更するというのであれば、当然煩雑な処理が伴うだろう。そのため、**演算子の結合性解決 (Fixity resolution)** は、構文解析の後処理と実装するのが好ましい。このようにすると、演算子が使用された後に結合性が宣言されていても、その結合性をもとにパースすることができる。結合性解決のアルゴリズムは、[8] の 10.6 節で実装付きで解説されているため、そちらを参照されたい。

4.4 構文的制約

構文解析では検出できない**構文的制約 (Syntactic restriction)** が必要となることがある。構文的制約は、人間にとってわかりにくいプログラムの記述を制限するという点で有用である。例えば、多くの関数型言語に共通すると思われる構文的制約を以下に示す。これは [10] に倣っている。

- 関数のパラメータやラムダ式に現れるパターン、データ型のパラメータは同じラベルの変数を含まない。
- 同じレベルで宣言された関数及びデータ型は互いに同じ

名前を持たない。

- データコンストラクタ名は同一モジュール内で一意である。

関数宣言やラムダ式では、一度に複数の引数をとることが可能であり、このときに同名の変数が存在するとどれを指しているのかわからなくなる。このために 1 つ目の条件が必要となる。また、同様にトップレベルや、同じ let 束縛内で宣言された関数宣言や型宣言に、同一のラベルを持つことを許すと、プログラム中で先に宣言したか後に宣言したかでプログラムの意味が変わってしまうという欠点がある。もちろん、異なるレベルで宣言された関数や変数は同名であってもシャドイングされるが、これが 2 つめの条件の理由である。3 つ目はデータコンストラクタに付く型注釈を省略するために必要である。データ型は型理論的には直和型として解釈されるが、直和型を追加したラムダ計算の体系では、純粋なラムダ計算の体系で成り立つ、型の一意性の定理が成り立たなくなる [14]。コンストラクタの構文に型注釈を追加することでこれを避けることもできるが、それよりもラベルの一意性を制約として加えたほうがプログラマに対する要求は少なくて済む。

5. 型検査

5.1 型付け構文

プログラムを構文解析し、抽象構文木に変換できたら、次は型検査のフェーズとなる。サンプル言語では、関数に型署名があってもなくても良いため、型署名がない場合は、型を推論する必要がある。また、最終的には明示的に型付け・カインド付されたコア言語を出力しなければならないため、型検査中、構文木に型情報・カインド情報を付加する操作も必要となる。型検査は効率性の観点からは、抽象構文木上で行う方が良いであろうが、その分、構文解析と型検査という異なるプロセスのモジュール性が低下し、開発と保守を難しくする。そのため、ここでは、抽象構文木から翻訳された型検査専用の構文上で、型検査を含めた意味解析の全てを行うこととする。型検査専用の構文をここでは**型付け構文 (Typing)** と呼び、以下にその定義を示す。[ ] で囲われた部分はオプショナルであることを表している。つまり、抽象構文木から翻訳された時点では [ ] の中は空であるが、型検査が終わると、全ての [ ] の中に型が付く。

name	$x,\ c$	
type variable	α	
term	$e ::=$	$x \mid e\ e \mid \lambda x[:t].\ e \mid \lambda p[:t].\ e \mid e\ \bar{t} \mid \Lambda\overline{\alpha[:k]}.\ e \mid \texttt{let}\ \mathrm{B}\ \texttt{in}\ e \mid \texttt{case}\ e[:t]\ \texttt{of}\ \{\overline{p \to e}\}$
pattern	$p ::=$	$x \mid _ \mid c\ \overline{p}$
type	$t ::=$	$\alpha \mid c \mid t \to t \mid \forall\overline{\alpha[:k]}.\ t \mid t\ t \mid \lambda x[:k].\ t \mid \mu x:k.\ t \mid \langle\overline{c,\ \bar{t}}\rangle$
kind	$k ::=$	$* \mid k \to k$
binds	$\mathrm{B} ::=$	$\{\overline{x = e}\}$

Fig. 5 Syntax of Typing

5.2 Damas-Milner 型システム

Damas-Milner 型システムとは、一階の多相型を持つ、型の注釈を必要としないラムダ計算の型システムである。すなわち、明示的な型注釈を持たないプログラミング言語の型推論アルゴリズムを与える。Damas-Milner 型システムの具体的なアルゴリズムは、Milner によって [3] や [9] で示されており、Algorithm $\mathcal{W}$ と呼ばれている。Algorithm $\mathcal{W}$ の Haskell による実装は、[4] や [6] が参考になる。また、型推論においてより良いエラーメッセージを出力するため、Algorithm $\mathcal{M}$ と呼ばれる別のアルゴリズムが Lee と Yi [7] によって研究された。Algorithm $\mathcal{M}$ は、型情報を推論関数の内部に伝播させることによって、エラーメッセージを改善する。ただし、Algorithm $\mathcal{W}$ と Algorithm $\mathcal{M}$ には欠点があり、置換 (substitution) を逐次計算しなければならないため、効率性に欠けることが知られている。計算量を改善するアルゴリズムは既に Milner が [9] で紹介しており、Algorithm $\mathcal{J}$ と呼ばれる。Algorithm $\mathcal{J}$ は Algorithm $\mathcal{W}$ に加えて、型変数をミュータブルに扱い、型変数自体に型を記録するという副作用を伴うことによって、置換を出力せずに済むようにしている。以上の Damas-Milner の型推論アルゴリズムをまとめると、Table 1 のようになる。

Table 1 Type inference algorithms of Damas-Milner type system

副作用	型と置換を生成	型を内部に伝播
なし	Algorithm $\mathcal{W}$	Algorithm $\mathcal{M}$
あり	Algorithm $\mathcal{J}$	

すなわち、エラーメッセージが改善され、かつ効率的なアルゴリズムが、上の表で空欄部分にあたる。ところが、これには決まった名前が与えられていないようなので、本稿では **Algorithm** $\mathcal{Z}$ と呼ぶことにする。Algorithm $\mathcal{Z}$ の形式的定義を Lee&Yi [7] のスタイルで Fig. 6 に示す。FV は型

$$\mathcal{Z} : TypeEnv \times Term \times Type \to Unit$$

$$\begin{aligned}
&\mathcal{Z}(\Gamma,\ x,\ \rho) = \\
&\quad \mathcal{U}(\rho,\ \{\bar{\beta}/\bar{\alpha}\}\tau) \text{ where } x : \forall\bar{\alpha}.\tau \in \Gamma,\ \text{new } \bar{\beta} \\
&\mathcal{Z}(\Gamma,\ e_1\ e_2,\ \rho) = \\
&\quad \mathcal{Z}(\Gamma,\ e_1,\ \beta \to \rho) \text{ where new } \beta; \quad \mathcal{Z}(\Gamma,\ e_2,\ \beta) \\
&\mathcal{Z}(\Gamma,\ \lambda x.e,\ \rho) = \\
&\quad \mathcal{U}(\rho,\ \beta_1 \to \beta_2) \text{ where new } \beta_1,\ \beta_2; \\
&\quad \mathcal{Z}(\Gamma \cup x : \beta_1,\ e,\ \beta_2) \\
&\mathcal{Z}(\Gamma,\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2,\ \rho) = \\
&\quad \mathcal{Z}(\Gamma,\ e_1,\ \beta) \text{ where new } \beta; \\
&\quad \mathcal{Z}(\Gamma \cup x : \forall\bar{\alpha}.\beta_1,\ e_2,\ \rho) \text{ where } \bar{\alpha} = FV(\Gamma) \backslash FV(\beta_1)
\end{aligned}$$

Fig. 6 Algorithm $\mathcal{Z}$

($Type$) や環境 ($TypeEnv$) に含まれる自由変数の集合を返す関数である。

型変数への型の記録は**単一化 (Unification)**$\mathcal{U}$ によって行われる。$\mathcal{U}$ は型変数に型を代入するという副作用をもち、

また単一化できなかった場合、失敗する。$\mathcal{Z}$ や $\mathcal{U}$ のアルゴリ

$$\mathcal{U}\colon Type \times Type \to Unit$$
$$\mathcal{U}(\alpha_1,\, \alpha_2) = \text{if } \alpha_1 = \alpha_2 \text{ then } unit \text{ else } \alpha_2 := \alpha_1$$
$$\mathcal{U}(c_1,\, c_2) = \text{if } c_1 = c_2 \text{ then } unit \text{ else } fail$$
$$\mathcal{U}(t_{11} \to t_{12},\, t_{21} \to t_{22}) = \mathcal{U}(t_{11},\, t_{21});\ \mathcal{U}(t_{12},\, t_{22})$$
$$\mathcal{U}(\alpha_1,\, t_2) = \text{if } \alpha_1 \notin FV(t_2) \text{ then } \alpha_1 := t_2 \text{ else } fail$$
$$\mathcal{U}(t_1,\, \alpha_2) = \text{if } \alpha_2 \notin FV(t_1) \text{ then } \alpha_2 := t_1 \text{ else } fail$$
$$\mathcal{U}(t_1,\, t_2) = fail$$

Fig. 7 Unification algorithm with side effects

ズムは最小限の構文を用いて記述しているが、これらは容易に型付け構文に拡張できる。

5.3 構文主導翻訳

本節では、型検査と同時にどのように項に型情報を付加するかを述べる。明示的に型付けされたコア言語では、Fig. 1 の `filter` 関数は以下のように表現される。

$$filter = \Lambda a\colon *.\ \lambda f\colon a \to Bool.\ \lambda xs\colon List\ a.\ ...$$

すなわち、型署名を分解して、ラムダ抽象に型を注釈するだけでなく、量化された型変数を型抽象として項の中に明示しなければならない。また、同じく 20 行目の式は以下のように、`Nat` 型が明示的に適用されなければならない。

$$filter\ Nat\ iseven\ (Zero\ ::\ Succ\ Zero\ ::\ ...)$$

Fig. 8 に項の書き換えを含んだ Algorithm $\mathcal{Z}$ の定義を示す。

$$\mathcal{Z}\colon TypeEnv \times Term \times Type \to Term$$
$$\mathcal{Z}(\Gamma,\, x,\, \rho) =$$
$$\quad \mathcal{U}(\rho,\, \{\bar{\beta}/\bar{\alpha}\}\tau) \text{ where } \Gamma(x) = \forall\bar{\alpha}.\tau,\ \text{new } \bar{\beta};$$
$$\quad \text{return } (x\ \bar{\beta})$$
$$\mathcal{Z}(\Gamma,\, e_1\ e_2,\, \rho) =$$
$$\quad e_1' = \mathcal{Z}(\Gamma,\, e_1,\, \beta \to \rho) \text{ where new } \beta;$$
$$\quad e_2' = \mathcal{Z}(\Gamma,\, e_2,\, \beta);\quad \text{return } (e_1'\ e_2')$$
$$\mathcal{Z}(\Gamma,\, \lambda x.\ e,\, \rho) =$$
$$\quad \mathcal{U}(\rho,\, \beta_1 \to \beta_2) \text{ where new } \beta_1,\ \beta_2;$$
$$\quad e' = \mathcal{Z}(\Gamma \cup x\colon \beta_1,\, e,\, \beta_2);\quad \text{return } (\lambda x\colon \beta_1.\ e')$$
$$\mathcal{Z}(\Gamma,\, \texttt{let}\ x \texttt{=} e_1\ \texttt{in}\ e_2,\, \rho) =$$
$$\quad e_1' = \mathcal{Z}(\Gamma,\, e_1,\, \beta) \text{ where new } \beta;$$
$$\quad e_2' = \mathcal{Z}(\Gamma \cup x\colon \forall\bar{\alpha}.\beta_1,\, e_2,\, \rho) \text{ where } \bar{\alpha} = FV(\Gamma)\backslash FV(\beta_1)$$
$$\quad \text{return } (\texttt{let}\ x\colon \forall\bar{\alpha}.\ \beta \texttt{=} \Lambda\bar{\alpha}.\ e_1'\ \texttt{in}\ e_2')$$

Fig. 8 Algorithm $\mathcal{Z}$ with syntax-directed translation

全称量化は冠頭でしか許されていないため、型適用を挿入する可能性があるのは、変数が現れる場所のみであり、型抽象を挿入する可能性があるのは、トップレベルで宣言された関数の本体と let 束縛の中身だけである。Damas-Milner 型システムにおける省略された型抽象、型適用の挿入は自明であるが、任意階の多相を許した型システムではより複雑になる (§8.1)。

5.4 カインド推論

カインドは型の正当性を確認するのに有用である。例えば、Fig. 1 の 5 行目の `List` は、$* \to *$ というカインドを持つ。ところが、14 行目の型署名で、`List a` が `List` となっていた場合、`List` に引数が足りないことは、カインドが真の型のカインド $(*)$ でないことによって確かめられる。しかし、型の正当性を確認するために、いちいちカインドを書くのは面倒である。そのため、データ型のカインドを推論し、型署名におけるカインドが正しいかどうかを検査する必要がある。

カインド推論の実装はそれほど難しくなく、単相型の型推論の問題と同等であるため、5.2 節で説明した方法がそのまま適用できる。

6. エラボレーション意味論

本節では、ソース言語の意味論を、コア言語への翻訳 (**エラボレーション**と言う) によって記述する。

6.1 代数的データ型の翻訳

代数的データ型の直感的解釈は、「直積型の直和型」である。しかし、代数的データ型の内部表現にはいくつか方法があり、存在型との対応関係を考慮すると、より高度な設計技法が求められる [5][17]。

ここでは、データ型の宣言をトップレベルに限ることによって、エラボレーションの単純化を試みる。データ型は、その型自体と、型を構成するためのコンストラクタ、すなわち関数へと翻訳される。Fig. 1 における List 型は型理論的には、$List = \mu X.\ \lambda a.\ Unit + a \times X\ a$ 表現されるが、もし、List 型と同様の定義をした List2 型があった場合、List 型と List2 型は区別されるべきであろうか? ソース言語上では両者は区別されるべきである。なぜならある型 X と Y がプログラム中で異なる意味で用いられているのに、コア表現が同じだからという理由で型の等価性が認められてしまうのは厄介な問題を引き起こす可能性があるからである。この点で、データ型はそのラベルによって区別されており、型の内部表現とは無関係であることが分かる。あるデータ型の値はそのコンストラクタによってのみアクセスされればよい。従って、List 型を例にすると、データ型は以下のように、型本体を持たない抽象型として翻訳される。

$$List\colon * \to *$$
$$Nil\colon \forall a.\ List\ a \qquad = \Lambda a.\ \langle Nil\ a\rangle$$
$$(::)\colon \forall a.\ a \to List\ a \to List\ a = \Lambda a.\ \lambda x.\ \lambda xs.\ \langle ::\ a\ x\ xs\rangle$$

Fig. 9 Elaboration of abstract data types

6.2 再帰関数の翻訳

再帰関数をアセンブリ言語などのマシン依存の言語に翻訳する時、一般的には特別の配慮を要することはない。なぜなら、低レベル言語には JUMP 命令があるからである。文字列のラベルを介して、プログラム中の任意の場所に飛ぶことができ、それによって、自己再帰も相互再帰も実現できる。

一方で、ラムダ計算は状態を持たず、ラムダ式のスコープ中でしか変数を扱えないため、純粋なラムダ計算で再帰関数を扱うことはできない。そこで、不動点コンビネータを導入し、コア言語上で再帰関数を評価できるようにする。Fig. 1 の `iseven` 関数と `isodd` 関数の相互再帰は、fix とレコードを使って Fig. 10 のように表せる。

```
ieio =
  fix (\ieio: {iseven: Nat -> Bool,
               isodd: Nat -> Bool}.
    {iseven = \Zero -> True
            | \Succ n -> not (ieio.isodd n),
      isodd = \Zero -> False
            | \Succ n -> not (ieio.iseven n)})
iseven = ieio.iseven
isodd = ieio.isodd
```

Fig. 10 Mutually recursive function

6.3 パターンマッチの翻訳

パターンマッチは、項の構造によって処理を分岐することができる、関数型言語の有用な特徴である。基本的にパターンになりうるのは、データコンストラクタ、リテラル、変数、ワイルドカードなどである。本節では、ソース言語のパターンマッチを変換して、コア言語に翻訳しやすい形に変換する方法を示す。ところが、パターンマッチに関わる性質はラムダ計算には備わっておらず、ソース言語からコア言語に翻訳する際には自明でないテクニックが必要となる。例えば、Haskell のパターンマッチは以下のような特徴を備えている。

1. `Cons x xs` のようなパターンに、`Cons 1 (Cons 2 Nil)` という項がマッチする時、`x` と `xs` にそれぞれ、`1` と `Cons 2 Nil` が代入される。
2. case 式だけでなく、関数の引数に任意個のパターンが現れることができる。(Fig. 5)
3. ネストされたパターンが使用できる。つまり、コンストラクタの引数の中にさらにパターンが現れることができる。
4. 関数の引数や case 式に現れるパターン同士は、互いに重複があってもよく、宣言された順にマッチングする。
5. 網羅されていないパターンマッチは静的に検査され、警告を出す。

これらの特徴を備えたパターンマッチをコア言語に翻訳するためには、パターンを全て、単なる変数か、あるいは、1 つのデータコンストラクタによってタグ付けされたパターンのみに変換する必要がある。形式的に表すと Fig. 11 のようになる。

$$\text{pattern} \quad p ::= \quad x \mid c\ \overline{x}$$

Fig. 11 Pattern syntax of canonicalized form

しかし、1. から 5. の特徴をみたすようなパターンマッチを翻訳するアルゴリズムは複雑である上に、あまり興味深いものではない。そのため、これらのアルゴリズムの説明は [13] の 5 章などに譲るとして、パターンが Fig. 11 の形で与えられた後の変換を解説する。

パターン抽象と case 式はそれぞれ Fig. 12 のように変換される。このような変換を行うため、コア言語における case 式の意味論はソース言語と異なり、式 e を評価した結果のタグが c_i に一致した時、その本体 $\lambda\bar{x}_i.\ e_i$ に引数を適用する。また、どのタグも一致しなかったとき、e_{def} に e を適用する。

$$\begin{aligned}
&\lambda p.\ e && \rightsquigarrow \lambda\bar{x}.\ e \text{ where } \bar{x} = \text{variables in } p \\
&\text{case } e \text{ of } \{\overline{p_i \to e_i}\} && \rightsquigarrow \text{case } e \text{ of } \{\overline{c_i \to \lambda\bar{x}_i.\ e_i}\}\ e_{\text{def}} \\
& && \quad \text{where } \bar{x}_i = \text{variables in } p_i \\
& && \quad\quad \text{and if } p_i = x_i \text{ then } e_{\text{def}} = \lambda x_i.\ e_i
\end{aligned}$$

Fig. 12 Elaboration of pattern matching

7. コンパイラ製作のまとめ

7.1 名前空間の扱い

型名やデータコンストラクタ名、関数名や型変数名など、項や型のレベルで異なる意味をもつ識別子が複数あり、これらは互いに同じ名前を持つこともある。そのため、識別子は単なる文字列だけでなく、どの名前空間に属するかの情報も含まなければならない。サンプル言語では識別子の型 `Name` を Fig. 13 に示す Haskell コードで表すことができる [16]。

```
data NameSpace
        = VarName   -- 変数や関数
        | ConName   -- データコンストラクタ
        | TyvarName -- 型変数
        | TyconName -- 型コンストラクタ
        | ModName   -- モジュール名
data Name = {nameString = String, nameSpace = NameSpace}
```

Fig. 13 Definition of data type 'Name'

7.2 モジュールシステム

ここでいうモジュールシステムとは、ファイルを一つのモジュールとみなし、ファイル間でコンポーネントのインポート・エクスポートを行うための機構である。本稿で扱うサンプル言語は数値や文字列、入出力を持たず、また一切の組み込み関数も備えていないため、まともなプログラムを書くには事前に定義すべきことが多すぎる。そのため、ここで簡単にプログラムをモジュール化する方法を紹介する。

単一ファイル内の型宣言、関数宣言は、存在型を使うことによってまとめられる。Fig. 1 のサンプルコードにおける `Bool` モジュールは、Fig. 14 のような型に翻訳される。ただし、相互再帰関数を扱うために導入される関数などは、外からアクセスできないような名前に変更する必要がある。

$$\exists Bool.\ \{True\colon Bool,\ False\colon Bool,\ not\colon Bool \to Bool\}$$

Fig. 14 Type of Bool module

Bool モジュールの本体が、`boolModule` という名前で束縛されているとすると、`import Bool` は例として Fig. 15 のように翻訳される。

ここで紹介したのは、極めて単純な存在型の利用だが、Rossberg らは、System F_ω へのエラボレーションによって、ML のモジュールシステムの全ての機能を実現した [15]。

$$
\begin{aligned}
&\text{unpack } \{Bool,\ bool\} = boolModule \text{ in} \\
&\text{let } True = bool.True \text{ in} \\
&\text{let } False = bool.True \text{ in} \\
&\text{let } not = bool.not \text{ in } ...
\end{aligned}
$$

Fig. 15 Elaboration of importing module

8. 発展的内容

8.1 任意階の多相型の型検査

Haskell や OCaml の型システムは、Damas-Milner 型システムをベースとしており、全称量化は冠頭でしか許されない。例えば、Haskell における `a -> b -> a` のような型署名は、`forall a b. a -> b -> a` と解釈される。**任意階の多相 (Arbitraly ranked polymorphism)** とは、冠頭だけでなく、型式の途中に全称量化が現れることを許すことをいい、ネストされた括弧の中に現れる `forall` の深さによって、1 階の多相 (Rank-1)、2 階の多相 (Rank-2)... などと呼ばれる。Rank-N 型 σ^n は次の定義をみたす集合である。

$$
\begin{aligned}
\sigma^0 &::= \quad \tau \\
\sigma^{n+1} &::= \quad \sigma^n \mid \sigma^n \to \sigma^{n+1} \mid \forall \alpha. \sigma^{n+1}
\end{aligned}
$$

Fig. 16 Definition of Rank-N types

しかし、型注釈無しで任意階の多相を導入しようとすると、Rank-3 以上では型検査が決定不能になってしまう。どの程度の型注釈が型検査の決定性に必要なのかは不明であるが、ここでは関数の型署名を必須にするという構文的制約を課すことによって、任意階の多相を導入する。任意階の多相の型推論アルゴリズムは、Odersky と Läufer[11] がよく研究しており、Jones ら [12] はそれを発展させ、構文主導の推論規則と、Haskell による実装コードを示した。

任意階の多相型が使えると、例えば次のようなコードが書ける。任意階の多相の元では、コンストラクタの中で型変数

```
data Monad m = Monad ({a} a -> m a)
                     ({a b} m a -> (a -> m b) -> m b)

monadReturn : {m} Monad m -> {a} a -> m a
monadReturn m = case m of Monad r _ -> r

monadBind : {m} Monad m -> {a b} m a -> (a -> m b) -> m b
monadBind m = case m of Monad _ b -> b
```

Fig. 17 Example program of arbitrary-ranked polymorphism

を量化したり、型署名の途中で型変数を量化することが可能となる。これは存在型と組み合わせることで、Haskell でいう型クラスを実現でき、より表現性の高いプログラムを書くことが可能となる。

9. 結論

本稿では、多くの関数型言語で用いられている有用な機構を、型志向の翻訳によって実装する方法を示した。関数型言語を型理論的解釈によって形式的に記述することで、コンパイラの安全性や、より高度な型システムの導入への大きな助けになることが期待される。

参考文献

[1] M. D. Adams. Principled parsing for indentation-sensitive languages: Revisiting landin's offside rule. *SIGPLAN Not.*, 48(1):511–522, jan 2013.

[2] A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, new ed edition, July 2004.

[3] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, page 207–212, New York, NY, USA, 1982. Association for Computing Machinery.

[4] S. Diehl. Write you a haskell, 2015. `https://smunix.github.io/dev.stephendiehl.com/fun/WYAH.pdf`.

[5] R. Harper and C. Stone. *A Type-Theoretic Interpretation of Standard ML*, page 341–387. MIT Press, Cambridge, MA, USA, 2000.

[6] M. P. Jones. Typing haskell in haskell. 1999.

[7] O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4):707–723, jul 1998.

[8] S. Marlow. *Haskell 2010 Language Report*, 2010. `https://www.haskell.org/onlinereport/haskell2010/`.

[9] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[10] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[11] M. Odersky and K. Läufer. Putting type annotations to work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, page 54–67, New York, NY, USA, 1996. Association for Computing Machinery.

[12] S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17:1–82, January 2005. Submitted to the Journal of Functional Programming.

[13] S. L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., USA, 1987.

[14] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[15] A. Rossberg, C. Russo, and D. Dreyer. F-ing modules. *Journal of Functional Programming*, 24(5):529–607, 2014.

[16] T. Tani. Ghc commentary: The compiler, 2021. `https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/`.

[17] J. C. Vanderwaart, D. Dreyer, L. Petersen, K. Crary, R. Harper, and P. Cheng. Typed compilation of recursive datatypes. *SIGPLAN Not.*, 38(3):98–108, jan 2003.