令和2年度 学士論文

代数的エフェクトを備えた関数型言語 Koka に対する、エフェクト割り当て最適化

東京工業大学 情報理工学院　数理・計算科学系
学籍番号 17B13400
古殿　直也

指導教員
増原　英彦　教授

2021年2月26日

概要

本研究では Koka 言語のためのコンパイラ最適化として open floating を提案する。Koka 言語は代数的エフェクトとそれを解析するエフェクトシステムを備えた関数型言語である。それらを備えることで例外や非決定計算などの計算エフェクトを使ったプログラムを安全かつ柔軟に書くことができる。代数的エフェクトを効率よく実現するためには、言語の意味論やコンパイラの実装方式を工夫する必要があり、Koka 言語ではエフェクト割り当てに基づいた変換であるエビデンス変換をコンパイラに採用している。そこで、open floating はエビデンス変換を施す前の中間表現のエフェクト割り当てを最適化することで、コンパイラの生成コードの実行速度を向上させる。本稿では Koka コンパイラの中間言語の形式化である F^{ϵ}+open に対して open floating を設計する。

謝辞

まず、本研究を進めるにあたり多くのアドバイスやご指導をいただいた増原英彦教授、叢悠悠助教、Microsoft Research の Daan Leijen 氏に心より感謝をいたします。

また、議論を交わし、多くの助言をしていただいている増原英彦研究室のみなさまにも感謝を申し上げます。特に池守君との議論は本研究を進めるために欠かせないものでした。

最後に家族のみなさまに感謝を申し上げます。

目次

図目次

第 1 章

導入

関数型言語では足し算などの純粋な計算と計算エフェクトを区別する。計算エフェクトには例外、破壊的代入、非決定計算などがある。計算エフェクトは言語の表現力を高める反面、プログラマが意図しない動作を引き起こす原因となる。この問題はエフェクトシステムを用いて解決できる。

エフェクトシステムは型システムの一種で、プログラムの型だけではなく、実行時に起きる計算エフェクトを静的に解析する。それによって計算エフェクトに関する安全性—例えば、ハンドルされない例外が存在しないこと—を保証する。

エフェクトシステムを備えた言語に新しい計算エフェクトを追加する場合、それに応じてエフェクトシステムを拡張する必要があり、言語実装者の負担となる。たとえば言語に非決定計算の機構を追加するとき、そのための演算を定義するだけではなく、エフェクトシステムを拡張し、定義した演算をエフェクトシステムの解析対象に追加しなければいけない。代数的エフェクトが言語に備わっていれば、この問題は解決する。代数的エフェクト [12] は言語機能の一つで、限定継続を扱える例外のようなものである。代数的エフェクトを使うことで、プログラマは例外や非決定計算などの計算エフェクトを定義し、それを使うプログラムを書くことができる。したがって、代数的エフェクトを備えた言語では例外や非決定計算などの機構を組み込みの演算として定義する必要がない。よって、エフェクトシステムは代数的エフェクトの解析を定義するだけで多くの計算エフェクトを扱うことができ、エフェクトシステムの定義がコンパクトになる。そのため言語実装者の負担が軽減される。

このように言語が代数的エフェクトを備えることには利点がある一方で、プログラムを高速に実行するためには、意味論の定義やコンパイル手法に工夫が必要である。エフェクトシステムと代数的エフェクトを備えた言語として Effekt[1], Eff[13], Koka[8, 9, 14] などが提案されている。本研究では Koka 言語の最適化手法を提案する。

Koka 言語コンパイラの処理の概要を図 1.1 に示す。Koka 言語コンパイラは表面言語のプログラムに型を割り当て、その情報を用いることで代数的エフェクトを備えた中間言語 Core へ変換する。プログラムは最適化を経て、代数的エフェクトを持たない低水準言語へと変換される。この変換をエビデンス変換 [14] と呼ぶ。エビデンス変換は Core

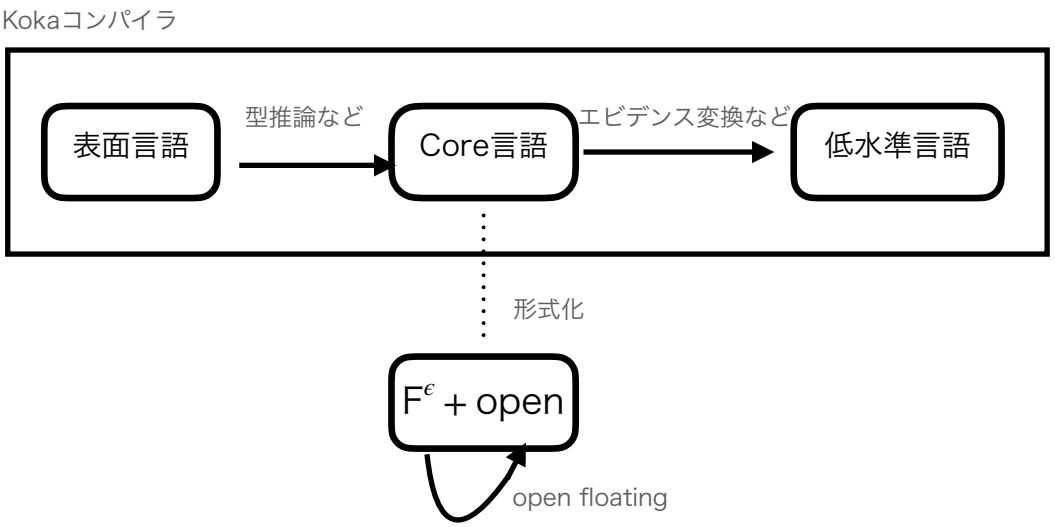

図 1.1　Koka コンパイラの概要

のエフェクト割り当てに基づいた変換手法であるから、Core でのエフェクト割り当てを最適化することで、効率の良いプログラムを生成できる。

本研究では中間言語 Core の形式化である、F^{ϵ}+open 上での最適化 open floating を提案する。Open floating は Koka 言語プログラムの型を部分的に保つ変換で、多くの関数適用のオーバーヘッドを減らすことが期待される。本稿では形式言語上での定義だけを与える。Core 言語への実装は未完成で、最適化の評価が必要である。

以降の章では、2 章では代数的エフェクトや言語の記述法を説明する。説明には F^{ϵ}+open から型をのぞいた、値呼びラムダ計算を代数的エフェクトで拡張した言語を用いる。3 章では、F^{ϵ}+open の説明をする。3 章で F^{ϵ}+open でのエフェクト割り当ての実行時への影響を述べたのちに、4 章で本研究の提案である open floating を定義する。5 章で今後の研究方針を述べ、6 章で関連研究を紹介する。

第 2 章

λ^{ϵ}: ラムダ計算の代数的エフェクトによる拡張

この章では値呼びラムダ計算の代数的エフェクトによる拡張である λ^{ϵ} の構文や意味、代数的エフェクトの使い方を説明する。F^{ϵ}+open は λ^{ϵ} の拡張で、構文や評価規則の多くを共有するため、λ^{ϵ} の理解は F^{ϵ}+open の理解のために必要である。

2.1 構文、評価規則

図 2.1 は Xie ら [14] によって提案された λ^{ϵ} 言語の定義である。値呼びラムダ計算を、オペレーション (perform op) とハンドラ式、ハンドル式 (handler h、 handle h e) で拡張することで代数的エフェクトを実現している。代数的エフェクトを用いたプログラミングでは、オペレーション呼び出しをすることで計算エフェクトを引き起こす。オペレーションの例として、変更可能な変数を扱う get と set や例外を投げる raise などが考えられる。オペレーション呼び出しの意味は対応するハンドラによって定まる。プログラマはハンドラを適切に定義することで、計算エフェクトを定義することができる。

λ^{ϵ} 言語では二つの簡約規則 handler, handle が計算をハンドルすることの意味を、簡約規則 perform がオペレーション呼び出しの意味を定義する。プログラマはハンドラ式 (handler h) を定義することでハンドラを定義する。ハンドラ (h) はオペレーションの名前と関数の組の集合 ($\{op_1 \to f_1, \ldots, op_n \to f_n\}$) で、オペレーション呼び出しをハンドルするときには、呼び出された名前に対応する関数が用いられる。これらの関数 ($f_1, \ldots, f_n$) のことをオペレーション節とよぶ。

ハンドラ式にサンクを渡すことで、そのハンドラの下でサンクを解凍し計算エフェクトをハンドルできる。簡約規則 handler で定義されているように、サンクを解凍するだけでなく、ハンドラ式がハンドル式に簡約される。そのハンドル式の下で、サンク v を意味を持たない値 () に適用することでサンクを解凍する。ハンドル式はハンドルする計算が値になるまで評価文脈に残り続け ($E ::=$ handle h E)、値になったとき簡約されその値を計算結果とする (簡約規則 return)。

式	e	$::= v$	(値)
		$\mid e\ e$	(関数適用)
		$\mid \text{let}\ \ x = e \,\text{in}\ \ e$	(let 式)
		$\mid \text{handle}\ \ h\ e$	(ハンドル式)
値	v	$::= x$	(変数)
		$\mid \lambda x.e$	(関数 f)
		$\mid \text{handler}\ \ h$	(ハンドラ式)
		$\mid \text{perform}\ \ op$	(オペレーション)
ハンドラ	h	$::= \{op_1 \to f_1, \ldots, op_n \to f_n\}$	

評価文脈　E　$::= \square \mid E\ e \mid v\ E \mid \text{let}\ \ x = E \,\text{in}\ \ e \mid \text{handle}\ \ h\ E$

簡約規則

(app)	$(\lambda x.e)\ v$	$\longrightarrow$	$e[x := v]$
(handler)	$(\text{handler}\ \ h)\ v$	$\longrightarrow$	$\text{handle}\ \ h\ v\ ()$
(return)	$\text{handle}\ \ h\ v$	$\longrightarrow$	v
(perform)	$(\text{handle}\ \ h\ E)[\text{perform}\ \ op\ v]$	$\longrightarrow$	$f\ v\ k$
	where $\ op \notin \text{bop}(E) \wedge (op \to f) \in h,$		$k = \lambda x.(\text{handle}\ \ h\ E)[x]$
(let)	$\text{let}\ \ x = v \,\text{in}\ \ e$	$\longrightarrow$	$e[x := v]$

評価規則

$$\frac{e \longrightarrow e'}{E[e] \longmapsto E[e']}$$

評価文脈がハンドルするオペレーション

$$\begin{aligned}
\text{bop}(\square) &= \emptyset \\
\text{bop}(E\ e) &= \text{bop}(E) \\
\text{bop}(v\ E) &= \text{bop}(E) \\
\text{bop}(\text{let}\ \ x = E \,\text{in}\ \ e) &= \text{bop}(E) \\
\text{bop}(\text{handle}\ \ h\ E) &= \text{bop}(E) \cup \{op \mid op \to f \in h\}
\end{aligned}$$

図 2.1　λ^{ϵ}: ラムダ計算の代数的エフェクトによる拡張

オペレーション呼び出しの簡約は perform 規則で定義される。オペレーション op を呼び出すときオペレーション呼び出しをハンドルするハンドラ handle h はもっとも内側にある op を実装するハンドラである。たとえば次のオペレーション呼び出しはハンドルするのはオペレーション節が f_2 のハンドラである。

```
handle {raise →f1}
  handle {raise → f2}
    perform raise 0
```

これは perform 規則の一つ目の付帯条件 $(op \notin \mathrm{bop}(E) \wedge (op \to f) \in h)$ から定まる。$\mathrm{bop}(E)$ は評価文脈 E が捕捉するオペレーションの名前の集合である。したがってこの条件は、ハンドルするハンドラ handle h はその内側の評価文脈は op をハンドルせず、なおかつ h が op に対するオペレーション節を定義するものであることを意味する。

オペレーション呼び出しを簡約すると、オペレーション節に呼び出しの引数 v とともに限定継続 k が渡される。この限定継続 k は「オペレーション呼び出しの結果」を受け取り、オペレーションをハンドルしたハンドル式の計算結果を返す関数である。次の式を考える。

```
handle {choose →λx.λk.(k (fst x))}
    let x = perform choose (1, 3) in x + 10
```

choose オペレーションの呼び出しは 1 行目のハンドル式がハンドルする。このとき捕捉される限定継続は

```
λy.handle choose→λx.λk.(k (fst x))
  let x = y in x + 10
```

である。

ここでオペレーション節への関数適用はオペレーション呼び出しの評価文脈ではなく、ハンドル式の評価文脈で行われることに注意が必要である。

λ^{ϵ} 言語ではハンドル式 (handle h e) を評価の中間状態を表すためだけに用いることを想定している。つまりプログラマが書くプログラムにはハンドル式は現れないことを仮定し、F^{ϵ}+open でも同様の仮定をする。これらの仮定によって、コンパイラは入力のプログラムにハンドル式が含まれないことを前提としたプログラム変換を行うことができる。例えば本稿が提案する最適化、open floating は F^{ϵ}+open の式で handle 式を含まないものだけに対して定義する。

簡約規則 app、let では、式 e に自由出現する変数 x に値 v を代入する。let 式を関数や関数適用を用いて表すことはできるが、let 式が F^{ϵ}+open にあると、本研究の貢献である open floating の最適化の機会が増える。そのため F^{ϵ}+open には let 式を定義し、それに合わせて λ^{ϵ} でも let 式を定義する。

2.2　例

代数的エフェクトを使ったプログラムの例を評価規則と照らし合わせながら説明する。説明のために言語に整数、組、リストやそれらを操作する演算が提供されていることを仮定する。

```
let f =
  λ_. [(perform choose (1, 3)) + (perform choose (10, 30))] in
let always-first =
  handler {choose → λx.λk.k (fst x)} in
100 :: (always-first f)
```

このプログラムは非決定計算 `f` と、それに対するハンドラ式 `always-first` を定義し、そのハンドラの下で非決定計算 `f` を実行するプログラムである。ハンドラ式 `always-first` は、ハンドルされる式に現れる `choose` オペレーションの意味を定義し、常に第一要素を選択する。そのため、`always-first f` を評価すると `11` が得られる。

評価は次のように進む。

```
100 :: [(always-first f)]
= 100 ::
  [handler {choose →λx.λk.k (fst x)}
    λ_.(perform choose (1, 3)) + (perform choose (10, 30))]
```

ハンドラ式を簡約し、ハンドル式の下で非決定計算を実行する。

```
⟼handler 100 ::
  [handle {choose → λx.λk.k (fst x)}
    (perform choose (1, 3)) + (perform choose (10, 30))]
```

オペレーション呼び出しが起きる。ハンドラ式の評価文脈 100::□ で、choose ハンドラのオペレーション節に引数 (1, 3) と限定継続を渡す。

```
⟼perform 100 ::
  [λx.λk.k (fst x))
    (1,3)
    (λy.handle {choose → λx.λk.k (fst x)}
        y + (perform choose (10, 30))]
⟼* 100 :: (k 1)
  (ここで k = (λy.handle {choose → λx.λk.k (fst x)}
                  y + (perform choose (10, 30)))。)
```

限定継続 k に値 1 を適用することで、中断されていた計算が「再開」する。これによってプログラムは choose オペレーションの呼び出しが 1 を返したかのように振る舞う。

```
⟼app 100 ::
  [handle {choose → λx.λk.k (fst x)}
    1 + (perform choose (10, 30))]
```

同様に評価が進み非決定計算の値が得られる。

```
⟼* 100 :: [handle {choose → λx.λk.k (fst x)} 11]
```

ハンドル式の計算結果は、ハンドルされる式の計算結果である。

```
⟼return 100 :: [11] ⟼ [100, 11]
```

オペレーション呼び出しの意味はハンドラが定めるため、異なるハンドラの下で計算を評価することで、異なる計算結果を得ることができる。

```
let f =
  λ_. [(perform choose (1, 3)) + (perform choose (10, 30))] in
let try-all =
  handler {choose → λx.λk. append(k (fst x), k (snd x))} in
100 :: (try-all f)
```

このプログラムはオペレーション呼び出しを含む関数 f の定義は変えずに、異なるハンドラの下で実行するプログラムである。オペレーション節で k に束縛される限定継続を 2 回使うことで、全ての選択肢を計算する。評価は以下のように進む。

```
100 :: (try-all f)
= 100 :: (handler {...}
    λ_.[(perform choose (1, 3)) + (perform choose (10, 30))])
⟼handler 100 ::
  (handle {...}
    [(perform choose (1, 3)) + (perform choose (10, 30))])
⟼perform 100 ::
  (λx.λk. append (k (fst x), k (snd x)))
    (1,3)
    (λy.handle {choose → λx.λk.k (fst x)}
      [y + (perform choose (10, 30))])
⟼* 100 :: append((k 1), (k 3))
      (where k = λy.handle {choose → λx.λk.k (fst x)}
                     [y + (perform choose (10, 30))])
```

```
⟼→app 100 :: append((handle {...}
          [1 + (perform choose (10, 30))]), (k 3))
⟼→* 100 :: append(append([11], (handle {...} [1 + 30])), (k 3))
⟼→* 100 :: append([11, 31], [13, 33])
⟼→* [100, 11, 31, 13, 33]
```

ハンドラがない場合、オペレーション呼び出しの評価は行き詰まる。

```
let f =
  λ_. [(perform choose (1, 3)) + (perform choose (10, 30))] in
f ()
⟼→ (perform choose (1, 3)) + (perform choose (10, 30))
↛
```

第 3 章

F^{ϵ}+open

F^{ϵ}+open は代数的エフェクトとエフェクトシステムを備えた言語で、Koka コンパイラの中間言語に対する形式化である。F^{ϵ}+open は Xie ら [14] によって定義された言語 F^{ϵ} を open、under というエフェクト変換で拡張した体系である。本研究は open floating を定義、実装することが主要な目標であり、その一環として F^{ϵ}+open の健全性を池守が示した [15]。

また F^{ϵ}+open は型を明示的に扱う言語であるからプログラムの型を操作する変換を定義しやすい。 そのため open floating はこの言語に対して定義される。

3.1 構文

F^{ϵ}+open は λ^{ϵ} の拡張である。追加される構文は図 3.1 にグレーで示した項に型が現れる構文と、図 3.2 で定める型の構文である。

F^{ϵ}+open で計算エフェクトを解析するためにエフェクトラベル (l)、エフェクトシグネチャ (sig)、それらの対応 (Σ) を扱う。エフェクトラベル (l) は計算エフェクトの名前である。例として、例外、非決定計算、状態を扱うことを考える。例外、非決定計算、状態のエフェクトラベルとして exn、 amb、 state というエフェクトラベルを割り当てる。エフェクトシグネチャは計算エフェクトを引き起こすオペレーションの名前と型の対応を表す。例外、非決定計算、状態のエフェクトシグネチャ (sig) として、次のようなものが考えられる。

$$\{raise : \forall\alpha.int \to \alpha\}$$
$$\{choose : \forall\alpha.(\alpha, \alpha) \to \alpha\}$$
$$\{get : () \to int, set : int \to ()\}$$

例外を引き起こすオペレーションは例外を投げる $raise$ のみで、そのオペレーションは int 型を入力とし、出力の型は任意とする。非決定計算を引き起こすオペレーションは非決定的に値を返す $choose$ のみで、任意の型に対して、その型の値のペアを受け取り、その一方を返す。状態を引き起こすオペレーションは値を取得するオペレーション get

と値を更新するオペレーション set があり、それぞれのオペレーションに対して型が定められている。

エフェクトラベルやエフェクトシグネチャは、それぞれの計算エフェクトに対応して与えられる。エフェクトシステムがそれらの情報を扱うためには、それぞれのエフェクトラベルとエフェクトシグネチャが対応しているという情報が必要である。その対応は Σ として与えれる。今回の例では次のようになる。

$$\begin{aligned}\Sigma = \{&exn : \{raise : \forall\alpha.int \to \alpha\}\\ &amb : \{choose : \forall\alpha.(\alpha, \alpha) \to \alpha\}\\ &state : \{get : () \to int, set : int \to ()\}\}\end{aligned}$$

エフェクトラベル l に対応するエフェクトシグネチャを $\Sigma(l)$ と書く。エフェクトラベルとエフェクトシグネチャの対応 Σ は外部から与えられると仮定し、Σ を定義したり、拡張したりする機能を F^{ϵ}+open では考えない。ハンドラやオペレーション呼び出しの型付けは、型システムの説明で後述するように、与えられた Σ に基づいて決定される。

F^{ϵ}+open のエフェクトシステムは式が起こすことのできる計算エフェクトを表現するためにエフェクト列を用いる。エフェクト列は型の一つとして定義され、空のエフェクト列（$\langle\rangle$）、型変数（μ）、それらのエフェクトラベルによる拡張（$\langle l \mid \epsilon\rangle$）のいずれかである。エフェクト列の等価性は図 3.2 で定義するように、ラベルの順序の違いを無視して判定する。

関数型は引数の型と返り値の型に加えて、関数の本体が起こせるエフェクト列から構成される。

たとえばエフェクト $\texttt{state} : \{get : () \to int, set : int \to ()\} \in \Sigma$ を仮定したとき、次の関数 `f` は $\forall\mu.() \to \langle state\rangle int$ 型をもつ。

```
f = Λ μ^eff.λ^⟨state⟩ _:().perform^⟨state⟩ get() + 3
```

エフェクト列の等価性をラベルの順序を無視して判定する動機は、エフェクト列に関する多相性から生じる。次の型が付く関数を考える。

$$\begin{aligned}&f : \forall\mu_1.() \to \langle l_1 \mid \mu_1\rangle(), && g : \forall\mu_2.() \to \langle l_2 \mid \mu_2\rangle()\\ &h : \forall\alpha.\alpha \to \langle\rangle(\alpha \to \langle\rangle\alpha)\end{aligned}$$

h を f, g に適用するために、f と g の型が一致することをエフェクトシステムは要求する。このとき μ_1 を $\langle l_2\rangle$ で、μ_2 を $\langle l_1\rangle$ で具体化すると、$() \to \langle l_1, l_2\rangle(), () \to \langle l_2, l_1\rangle()$ が得られ、ラベルの順序の違いを無視することで二つの関数型は一致する。

関数型がつく項 (関数、オペレーション、ハンドラ式) にはエフェクト列注釈がつく。またハンドラのオペレーション節は多相型を持つ事があり、それに対応してオペレーションに型パラメータを適用する。たとえば先程の例であげた非決定計算のハンドラは次のように多相的に定義される。

```
handler^⟨⟩{
```

式	e	$::= v$	(値)
		$\mid e\ e$	(関数適用)
		$\mid$ let $x = e$ in e	(*let* 式)
		$\mid$ handle$^{\epsilon}$ $h\ e$	(ハンドル式)
		$\mid e[\sigma]$	(型適用)
		$\mid \mathrm{open}[\langle l_1, \ldots, l_n \rangle, \langle l_1, \ldots, l_n \mid \epsilon \rangle](e)$	(*open* 式)
		$\mid \mathrm{under}[\langle l_1, \ldots, l_n \rangle, \langle l_1, \ldots, l_n \mid \epsilon \rangle](e)$	(*under* 式)

値	v	$::= x$	(変数)
		$\mid \lambda^{\epsilon} x : \sigma.e$	(関数 f)
		$\mid$ handler$^{\epsilon}$ h	(ハンドラ式)
		$\mid$ perform$^{\epsilon}$ $op\ \overline{\sigma}$	(オペレーション)
		$\mid \Lambda\alpha^{k}.v$	(型抽象)
ハンドラ	h	$::= \{op_1 \to f_1, \ldots, op_n \to f_n\}$	

図 3.1　F$^{\epsilon}$+open の構文（式、値）

$$\texttt{choose} \to \Lambda\,\alpha^{*}.\lambda^{\langle\rangle}\texttt{x:}(\alpha,\alpha).\lambda^{\langle\rangle}\texttt{k:}\alpha \to\ \langle\texttt{amb}\rangle\ \alpha.\ \texttt{k (fst x) }\}$$

open、under はエフェクト列の変換をする。詳しくは後述の型規則とともに説明する。

エフェクトに関係しない型やカインドは System F$_{\omega}$ と同様に定義される。

3.2　型システム

本節ではエフェクトシステムの型付け関係と保証される安全性を説明した後、それぞれの型規則を詳細に説明する。

エフェクトシグネチャ	sig	$::= \{op_1 : \forall\overline{\alpha}_1.\sigma_1 \to \sigma'_1, \ldots, op_n : \forall\overline{\alpha}_n.\sigma_n \to \sigma'_n\}$	
エフェクトラベルとシグネチャの対応	Σ	$::= \{l_1 : \mathrm{sig}_1, \ldots, l_n : \mathrm{sig}_n\}$	
エフェクト列	$\langle\rangle$	(空のエフェクト列)	
	$\langle \_ \mid \_ \rangle$	(エフェクト列の拡張)	

エフェクトに関するメタ変数

ϵ	$::= \sigma^{\mathtt{eff}}$	(エフェクト列)
μ	$::= \alpha^{\mathtt{eff}}$	(エフェクト列の型変数)
l	$::= c^{\mathtt{lab}}$	(エフェクトラベル)

エフェクト列に関する略記

$$\langle l_1, \ldots, l_n \rangle \stackrel{\triangle}{=} \langle l_1 \mid \langle \ldots \mid \langle l_n \mid \langle\rangle \rangle \rangle \rangle$$
$$\langle l_1, \ldots, l_n \mid \epsilon \rangle \stackrel{\triangle}{=} \langle l_1 \mid \langle \ldots \mid \epsilon \rangle \rangle$$

エフェクト列の等価性

$$\frac{}{\epsilon \equiv \epsilon}\ \text{(反射)} \qquad \frac{\epsilon_1 \equiv \epsilon_2 \qquad \epsilon_2 \equiv \epsilon_3}{\epsilon_1 \equiv \epsilon_3}\ \text{(推移)}$$

$$\frac{l_1 \neq l_2 \qquad \epsilon_1 \equiv \epsilon_2}{\langle l_1, l_2 \mid \epsilon_1 \rangle \equiv \langle l_2, l_1 \mid \epsilon_2 \rangle}\ \text{(交換)} \qquad \frac{\epsilon_1 \equiv \epsilon_2}{\langle l \mid \epsilon_1 \rangle \equiv \langle l \mid \epsilon_2 \rangle}\ \text{(拡張)}$$

型	σ	$::= \alpha^k$	(カインド k の型変数)
		$\mid c^k \sigma \ldots \sigma$	(カインド k の型構築子)
		$\mid \sigma \to \epsilon\sigma$	(関数型)
		$\mid \forall\alpha^k.\sigma$	(全称型)
カインド	k	$::= *$	(値)
		$\mid k \to k$	(型構築子)
		$\mid \mathtt{eff}$	(エフェクト列)
		$\mid \mathtt{lab}$	(エフェクトラベル)

図 3.2　F^{ϵ}+open の構文（型）

$$
\begin{aligned}
\text{評価文脈}\quad E \quad ::= {} & \Box \mid E\ e \mid v\ E \mid \text{let}\ x = E \,\text{in}\ e \mid \text{handle}\ h\ E \mid E\ [\sigma] \\
& \mid \text{open}[\langle l_1, \ldots, l_n\rangle, \langle l_1, \ldots, l_n \mid \epsilon\rangle](E) \\
& \mid \text{under}[\langle l_1, \ldots, l_n\rangle\langle l_1, \ldots, l_n \mid \epsilon\rangle](E)
\end{aligned}
$$

簡約規則

(app)	$(\lambda^{\epsilon} x : \sigma.e)\ v$	$\longrightarrow$	$e[x := v]$
(handler)	$(\,\text{handler}^{\,\epsilon}\ h)\ v$	$\longrightarrow$	$\text{handle}^{\,\epsilon}\ h \cdot v\ ()$
(return)	$\text{handle}^{\,\epsilon}\ h\ v$	$\longrightarrow$	v
(perform)	$(\,\text{handle}^{\,\epsilon}\ h\ E)[\,\text{perform}^{\,\epsilon'}\ op\ \overline{\sigma}\ v]$	$\longrightarrow$	$f[\overline{\sigma}]\ v\ k$
where	$op \notin bop(E) \wedge (op \to f) \in h,$		
	$op : \forall\overline{\alpha}.\sigma_1 \to \sigma_2 \in \Sigma(l)\,,$		
$\{op \mid op \to f \in h\}$	$= \{op \mid op : \forall\overline{\alpha}.\sigma_1 \to \sigma_2 \in \Sigma(l)\}$		
	$k = \lambda^{\epsilon} x : \sigma.(\,\text{handle}^{\,\epsilon}\ h\ E)[x]$		
(let)	$\text{let}\ x = v \,\text{in}\ e$	$\longrightarrow$	$e[x := v]$
(tapp)	$\Lambda\alpha^k v[\sigma]$	$\longrightarrow$	$e[\alpha^k ::= \sigma]$
(open)	$\text{open}[\langle l_1, \ldots, l_n\rangle, \langle l_1, \ldots, l_n \mid \epsilon\rangle](v)\ v'$		
	$\longrightarrow \text{under}[\langle l_1, \ldots, l_n\rangle, \langle l_1, \ldots, l_n \mid \epsilon\rangle](v\ v')$		
(under)	$\text{under}[\langle l_1, \ldots, l_n\rangle, \langle l_1, \ldots, l_n \mid \epsilon\rangle](v)$	$\longrightarrow$	v

評価規則

$$\frac{e \longrightarrow e'}{E[e] \longmapsto E[e']}$$

評価文脈がハンドルするオペレーション

$$
\begin{aligned}
\text{bop}(\Box) &= \emptyset \\
\text{bop}(E\ e) &= \text{bop}(E) \\
\text{bop}(v\ E) &= \text{bop}(E) \\
\text{bop}(\,\text{let}\ x = E \,\text{in}\ e) &= \text{bop}(E) \\
\text{bop}(\,\text{handle}^{\,\epsilon}\ h\ E) &= \text{bop}(E) \cup \{op \mid op \to f \in h\}
\end{aligned}
$$

図 3.3　F$^{\epsilon}$+open の評価規則

3.2.1　型付け関係

型付け関係は式、値、ハンドラのそれぞれに対しての定義と、型に対するカインドつけ関係からなる。それぞれ次のような形式である。

$$\Gamma \vdash_{exp} e : \sigma \mid \epsilon$$
$$\Gamma \vdash_{val} v : \sigma$$
$$\Gamma \vdash_{ops} h : \sigma \mid l \mid \epsilon$$
$$\Gamma \vdash_{wf} \sigma : k$$

式に対する型付け関係は型環境、式、型、エフェクト列の間の 4 項関係であり、さらに型環境、エフェクト列、式から型を返す部分関数とみなせる。つまり型環境、式、エフェクト列が与えられたとき、型付け関係を満たす型が高々一つ存在する。言い換えると、型環境 Γ、エフェクト列 ϵ のもとで式 e に型 σ が付く。値に対する型付け関係は型環境、値から型を返し、ハンドラに対する関係は型環境、ハンドラ、エフェクト列から型とラベルの組を返し、カインドつけは型環境、型からカインドを返す部分関数とみなせる。

型環境は項変数と型の組か型変数とカインドの組の列である。

型環境	Γ	::=	$\bullet$	(空の型環境)
		$\mid$	$\Gamma, x : \sigma$	(項変数と型の組による拡張)
		$\mid$	Γ, α^k	(型変数とカインドの組による拡張)

3.2.2　基本的な性質

F$^{\epsilon}$+open の型システムの性質を述べる。ここで取り上げる性質は、すべて池守 [15] によって証明されている。

まず、F$^{\epsilon}$+open は健全である。つまり保存と進行が成り立つ。

定理 1 (保存)**.** 空の型環境、空のエフェクト列のもとで式 e に型 σ がつくとき、e が e' に評価されるならば e' に型 σ がつく。すなわち

$\bullet \vdash_{exp} e : \sigma \mid \langle\rangle$ かつ $e \longmapsto e'$ ならば $\bullet \vdash_{exp} e' : \sigma \mid \langle\rangle$。

定理 2 (進行)**.** 空の型環境、空のエフェクト列のもとで式 e に型がつくならば、e は値であるか評価できる。すなわち

$\bullet \vdash_{exp} e : \sigma \mid \epsilon$ ならば e は値であるか、ある式 e' が存在して $e \longmapsto e'$。

次に、式 e がエフェクト列 ϵ のもとで型付けできるとき、式 e でハンドルされないオペレーションのエフェクトラベルはエフェクト列 ϵ に含まれる。

補題 1 (エフェクト列の意味)**.** $\bullet \vdash_{exp} E[\,\mathrm{perform}^{\,\epsilon'}\ op\ \overline{\sigma}\ v] : \sigma \mid \epsilon$ かつ $op \notin bop(E)$ ならば $op \in \Sigma(l)$ かつ $l \in \epsilon$。

この補題から空のエフェクト列のもとで型がつく式では、全てのオペレーション呼び出しがハンドルされることが導かれる。

3.2.3　型規則

■値　式に対する型付け関係は、値に対するものと、それ以外の式に対するものに分けて定義されるが、値は式の一つであるから、式としての型付けもできる。

$$\frac{\Gamma \vdash_{val} v : \sigma \qquad \Gamma \vdash_{wf} \epsilon : \mathtt{eff}}{\Gamma \vdash_{exp} v : \sigma \mid \epsilon} \tag{Val}$$

値に型がつくとき、その値は式として、任意のエフェク列のもとで同じ型がつく。この規則は、値の評価は計算エフェクトを起こさないため、評価文脈のハンドラに制約を受けないことを表現する。

■変数

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash_{val} x : \sigma} \tag{Var}$$

一般的な型システムと同様に、変数の型付けは型環境から定まる。

■let 式

$$\frac{\Gamma \vdash_{exp} e_1 : \sigma_1 \mid \epsilon \qquad \Gamma, x : \sigma_1 \vdash_{exp} e_2 : \sigma_2 \mid \epsilon}{\Gamma \vdash_{exp} \mathrm{let}\, x = e_1 \,\mathrm{in}\, e_2 : \sigma_2 \mid \epsilon} \tag{Let}$$

エフェクト以外の部分は一般的な型システムと変わらない。エフェクトに関する部分は、全体の評価文脈と部分式の評価文脈が同じ計算エフェクトを捕捉することを反映している。

■型抽象

$$\frac{\Gamma, \alpha^k \vdash_{val} v : \sigma}{\Gamma \vdash_{val} \Lambda\alpha^k.v : \forall\alpha.\sigma} \tag{TAbs}$$

型抽象できるのは値だけである。エフェクトをもつ式に多相型をつけると型システムの健全性が損なわれるためこのような制約が必要である。多相型を付けられる式を値に制限する制約は value restriction[11] と呼ばれる。F^{ϵ}+open のもとになった体系である F^{ϵ}[14] でも同じ制約を課している。[*1]

また、型抽象で導入される型変数にはカインドが注釈としてつく。

[*1] F^{ϵ}+open では式のエフェクトを解析するため、value restriciton を次のように弱められることが期待される:

■型適用

$$\frac{\Gamma \vdash_{exp} e : \forall\alpha^{k}.\sigma_1 \mid \epsilon \qquad \Gamma \vdash_{wf} \sigma_2 : k}{\Gamma \vdash_{exp} e[\sigma_2] : \sigma_1[\alpha := \sigma_2] \mid \epsilon} \quad (\textsc{TApp})$$

型適用の型規則は通常のものと同様である。

■オペレーション

$$\frac{op : \forall\overline{\alpha}.\sigma_1 \to \sigma_2 \in \Sigma(l) \qquad \overline{\alpha} \cap ftv(\Gamma) = \emptyset \qquad \Gamma \vdash_{wf} \epsilon : \mathtt{eff}}{\Gamma \vdash_{val} \text{ perform}^{\epsilon} op\,\overline{\sigma} : \sigma_1[\overline{\alpha} := \overline{\sigma}] \to \langle l \mid \epsilon\rangle\sigma_2[\overline{\alpha} := \overline{\sigma}]} \ (\textsc{Perform})$$

ここで $ftv(\Gamma)$ は型環境 Γ に自由出現する型変数の集合である。オペレーション呼び出しは オペレーションの名前と型パラメータから関数型の値を構成し、その値を引数に適用することによって実行される。簡約規則 perform はオペレーション から関数型の値を構成する操作で、その型はエフェクトシグネチャとラベルの対応 Σ, 適用される型 $\overline{\sigma}$、そして peform についたエフェクト注釈 ϵ から定まる。たとえば $(exn : \{raise : \forall\alpha.int \to \alpha\}) \in \Sigma$ を仮定したとき、 perform$^{\epsilon}$ $raise\ int$ に対する型付けは次のように導出される。

$$\text{(Perform)}\ \frac{raise : \forall\alpha.int \to \alpha \in \Sigma(exn) \qquad \alpha \notin \Gamma \qquad \Gamma \vdash_{wf} \langle exn\rangle : \mathtt{eff}}{\Gamma \vdash_{val} \text{ perform}^{\epsilon}\ raise\ int : () \to \langle exn \mid \epsilon\rangle int)}$$

■ハンドラ・ハンドラ式・ハンドル式　(Handler) 規則と (Handle) 規則は補助的な型付け関係 $\vdash_{ops}$ を用いて定義される。$\vdash_{ops}$ は型規則 (OPs) から定まる関係である。

ハンドラ

$$\frac{\begin{array}{c}\{op_i\}_i = \Sigma(l) \qquad op_i : \forall\overline{\alpha}.\sigma_i^{in} \to \sigma_i^{out} \in \Sigma(l) \qquad \overline{\alpha} \not\cap ftv(\Gamma) \\ \Gamma \vdash_{val} f_i : \forall\overline{\alpha}.\sigma_i^{in} \to \epsilon(\sigma_i^{out} \to \epsilon\sigma) \to \epsilon\sigma \\ \text{for all } i \in \{1,\ldots,n\}\end{array}}{\Gamma \vdash_{ops} \{op_1 \to f_1, \ldots, op_n \to f_n\} : \sigma \mid l \mid \epsilon} \quad (\textsc{OPs})$$

(OPs) は ハンドラ $\{op_1 \to f_1, \ldots, op_n \to f_n\}$ に型とエフェクトラベルを割り当てる規則である。(OPs) 規則の結論は、型環境 Γ、エフェクト ϵ のもとでエフェクトシグネチャ $\Sigma(l)$ に対応し、その返り値型は σ であることを意味する。

$$\frac{\Gamma, \alpha^{k} \vdash_{exp} e : \sigma \mid \langle\rangle}{\Gamma \vdash_{val} \Lambda\alpha^{k}.e : \forall\alpha.\sigma} \quad (\textsc{TAbs (purity)})$$

この規則は型抽象できる式を $\langle\rangle$ エフェクトがつく式に制限する。これは value restriction よりも真に弱い制約で purity restriction と呼ばれる。この型規則を代わりに採用した体系の健全性はまだ示されていない。健全性を保つためには後述する型適用の評価規則を変更する必要があると思われる。この拡張は直近の課題の一つである。

ハンドラ式

$$\frac{\Gamma \vdash_{ops} h : \sigma \mid l \mid \epsilon}{\Gamma \vdash_{val} \text{handler}^{\,\epsilon} h : (() \to \langle l \mid \epsilon \rangle \sigma) \to \epsilon\sigma} \quad (\textsc{Handler})$$

handler 式はサンクを引数にとる関数であった。サンクで起きる計算エフェクトはこのハンドラで捕捉されるため、関数型のエフェクト列として $\langle l \mid \epsilon \rangle$ をもつ。

ハンドル式

$$\frac{\Gamma \vdash_{ops} h : \sigma \mid l \mid \epsilon \qquad \Gamma \vdash_{exp} e : \sigma \mid \langle l \mid \epsilon \rangle}{\Gamma \vdash_{exp} \text{handle}^{\,\epsilon}\ h\ e : \sigma \mid \epsilon} \quad (\textsc{Handle})$$

handle 式は内部表現で、評価文脈にフレームを残すためのものである。したがって e はサンクではなく、そのエフェクトが $\langle l \mid \epsilon \rangle$ になっている。

■関数

$$\frac{\Gamma, x : \sigma_1 \vdash_{exp} e : \sigma_2 \mid \epsilon \qquad \Gamma \vdash_{wf} \sigma_1 : *}{\Gamma \vdash_{val} \lambda^{\epsilon} x : \sigma_1 . e : \sigma_1 \to \epsilon\sigma_2} \quad (\textsc{Lam})$$

関数の型付けは、その注釈（λ^{ϵ}...）に依存する。ラムダ抽象には仮引数の型に加えて、本体のエフェクトが注釈として与えられている。型規則では、本体がそのエフェクトのもとで型付けされることを要求する。

■関数適用

$$\frac{\Gamma \vdash_{exp} e_1 : \sigma_2 \to \epsilon\sigma_2 \mid \epsilon \qquad \Gamma \vdash_{exp} e_2 : \sigma \mid \epsilon}{\Gamma \vdash_{exp} e_1\, e_2 : \sigma \mid \epsilon} \quad (\textsc{App})$$

関数適用がエフェクト ϵ のもとで型付けされるためには、部分式のそれぞれ (e_1, e_2) が同じエフェクト (ϵ) のもとで型付けされることに加えて、関数型のエフェクトが同じエフェクトであることを要求する。これらの要求は、項の評価が型付け、エフェクト付けを保存するために必要である。関数の評価は e_1 を評価する、e_2 を評価する、β 簡約をする、のいずれかである。いずれの評価も、その評価文脈に含まれるハンドラは関数適用の評価文脈のハンドラと同じであるから、部分式のエフェクトは関数適用のエフェクト一致することが要求される。

■open

$$\frac{\Gamma \vdash_{exp} e : \sigma_1 \to \langle l_1, \ldots, l_n \rangle \sigma_2 \mid \epsilon \qquad \Gamma \vdash_{wf} \epsilon_i}{\Gamma \vdash_{exp} \text{open}\,[\langle l_1, \ldots, l_n \rangle, \langle l_1, \ldots, l_n \mid \epsilon_i \rangle](e) : \sigma_1 \to \langle l_1, \ldots, l_n \mid \epsilon_i \rangle \sigma_2 \mid \epsilon} \quad (\textsc{Open})$$

$\langle l_1, \ldots, l_n \rangle$ のように末尾が $\langle\rangle$ であるエフェクト列のことを閉じたエフェクト列とよび、$\langle l_1, \ldots, l_n \mid \mu \rangle$ のように末尾が型変数であるエフェクト列のことを開いたエフェクト列とよぶ。(Open) は閉じた関数型のエフェクト列 ($\langle l_1, \ldots, l_n \rangle$) の末尾を、他のエフェクト列 ($\epsilon$) で置き換える変換である。この規則によって、閉じた型がついた関数をさまざまな評価文脈で呼び出すことができる。 open や under を使うプログラムは本研究の貢献である open floating と深い関わりがあるため、後でより詳細に説明する。

■under

$$\frac{\Gamma \vdash_{exp} e : \sigma \mid \langle l_1, \ldots, l_n \rangle \qquad \Gamma \vdash_{wf} \epsilon_i : \mathtt{eff}}{\Gamma \vdash_{exp} \text{ under}\,[\langle l_1, \ldots, l_n \rangle, \langle l_1, \ldots, l_n \mid \epsilon_i \rangle(e) : \sigma \mid \langle l_1, \ldots, l_n \mid \epsilon_i \rangle} \quad (\textsc{Under})$$

(Open) は関数型のエフェクトを変換するのに対して、(Under) は式につくエフェクトをを変換する。

カインド規則は次のように定める。

$$\frac{\alpha^k \in \Gamma}{\Gamma \vdash_{wf} \alpha^k : k} \quad (\textsc{KindVar})$$

$$\frac{\Gamma, \alpha^k \vdash_{wf} \sigma : * \qquad k \neq \mathtt{lab}}{\Gamma \vdash_{wf} \forall \alpha^k . \sigma : *} \quad (\textsc{KindQuant})$$

$$\frac{\Gamma \vdash_{wf} \sigma_1 : k' \to k \qquad \sigma_2 : k'}{\Gamma \vdash_{wf} \sigma_1\ \sigma_2 : k} \quad (\textsc{KindApp})$$

$$\frac{\Gamma \vdash_{wf} \sigma_1 : * \qquad \Gamma \vdash_{wf} \sigma_2 : * \qquad \Gamma \vdash_{wf} \epsilon : \mathtt{eff}}{\Gamma \vdash_{wf} \sigma_1 \to \epsilon \sigma_2 : *} \quad (\textsc{KindArrow})$$

$$\frac{}{\Gamma \vdash_{wf} c^k : k} \quad (\textsc{KindCon})$$

$$\frac{l \in \Sigma}{\Gamma \vdash_{wf} l : \mathtt{lab}} \quad (\textsc{KindLabel})$$

$$\frac{}{\Gamma \vdash_{wf} \langle\rangle : \mathtt{eff}} \quad (\textsc{KindTotal})$$

$$\frac{\Gamma \vdash_{wf} \epsilon : \mathtt{eff} \qquad \Gamma \vdash_{wf} l : \mathtt{lab}}{\Gamma \vdash_{wf} \langle l \mid \epsilon \rangle : \mathtt{eff}} \quad (\textsc{KindRow})$$

3.3　等価な F$^{\epsilon}$+open プログラムとそれらの実行効率

F$^{\epsilon}$+open では同じ意味をもつプログラムに違う型やエフェクトを割り当てることができる。一方で、Koka コンパイラが F$^{\epsilon}$+open に対応する中間言語 Core のプログラムをより低水準な言語に変換すると、F$^{\epsilon}$+open では同じ意味であったプログラムが、実行効率の異なるプログラムに変換されることがある。

本節では実行効率の違いが生じる理由を述べ、より効率的なプログラムの性質を説明する。

3.3.1　型の消去

本稿では F$^{\epsilon}$+open の式が同じ意味をもつことをはかるために、型の消去を用いる。つまり型の消去で得られる λ^{ϵ} プログラムが一致する F$^{\epsilon}$+open プログラムは同じ意味をもつとみなす。open floating はこの意味で、プログラムの意味を保つ変換である。

定義 1 (型の消去)**.** *F$^{\epsilon}$+open* の式に対する型の消去 $\cdot^*$ を次のように定める。

$$
\begin{aligned}
(e_1\ e_2)^* &= e_1^*\ e_2^* \\
(\,\mathrm{let}\ \ x = e_1 \,\mathrm{in}\ \ e_2)^* &= \mathrm{let}\ \ x = e^* \,\mathrm{in}\ \ e_2^* \\
(\,\mathrm{handle}^{\,\epsilon}\ h\ e)^* &= \mathrm{handle}\ \ h^*\ e^* \\
(e[\sigma])^* &= e^* \\
(\mathit{open}[\langle l_1, \ldots, l_n\rangle, \langle l_1, \ldots, l_n \mid \epsilon\rangle](e))^* &= e^* \\
(\mathit{under}[\langle l_1, \ldots, l_n\rangle, \langle l_1, \ldots, l_n \mid \epsilon\rangle](e))^* &= e^* \\
x^* &= x \\
(\lambda^{\epsilon} x : \sigma.e)^* &= \lambda x.e^* \\
(\,\mathrm{handler}^{\,\epsilon}\ h)^* &= \mathrm{handler}\ \ h^* \\
(\,\mathrm{perform}^{\,\epsilon}\ \mathit{op}\ \overline{\sigma})^* &= \mathrm{perform}\ \ \mathit{op} \\
(\Lambda\alpha^k.v)^* &= v^* \\
\{op_1 \to f_1, \ldots, op_n \to f_n\}^* &= \{op_1 \to f_1^*, \ldots, op_n \to f_n^*\}
\end{aligned}
$$

3.3.2　Koka コンパイラの最適化と F$^{\epsilon}$+open プログラムの関係

Koka コンパイラは F$^{\epsilon}$+open に対応する中間言語のプログラムをトランスパイルして、オペレーション呼び出しでハンドラの実装の列を参照するプログラムを生成する [14]。このトランスパイルをエビデンス変換とよび、ハンドラの実装の列をエビデンスベクタとよぶ。生成されるプログラムは参照するエビデンスベクタが中間言語で割り当てられたエフェクト列に対応することを仮定したプログラムである。たとえば $\lambda^{\langle \mathtt{amb,exn,state}\rangle}\mathtt{x.f}$ (perform `set x`) のようなコンパイラの中間表現にエビデンス変換を施すと、参照されるエビデンスベクタが $\langle$`amb,exn,state`$\rangle$ と対応することを仮定した関数に変換される。

F^{ϵ}+open にはエフェクト列を変換するための式として open 式、under 式がある。それに対応する `open[⟨amb,state⟩,⟨amb,exn,state⟩](f) 3` のようなコンパイラの中間表現は、関数 `f` を呼び出す前にエビデンスベクタを ⟨amb,exn,state⟩ から ⟨exn⟩ に編集するプログラムにコンパイルされる。Koka コンパイラでは、このようなコードが多く生成され、生成プログラムの実行速度を低下させる要因になっている。

エビデンスベクタの型が静的に定まれば、適切なハンドラを参照する操作を定数時間で実行できる。そしてエビデンスベクタの型が静的に定まるか否かは、F^{ϵ}+open の項に出現するエフェクト列の末尾が $\langle\rangle$ であるか、型変数であるかで定まる。そのため、F^{ϵ}+open の項は $\langle l_1,\ldots,l_n\rangle$ のような閉じたエフェクト列が $\langle l_1,\ldots,l_n \mid \mu\rangle$ のような開いたエフェクト列よりも望ましい。

以上をまとめると、Koka コンパイラでより効率の良いプログラムに変換される F^{ϵ}+open の関数は次の基準で判断できる。

- 呼び出される open 式、under 式の個数は少ない方が望ましい。
- エフェクト列注釈は閉じていることが望ましい。

3.3.3　実行効率が異なる等価な F^{ϵ}+open プログラム

次の λ^{ϵ} プログラムを考える。

```
let f = λx.1 + perform raise () in
(handler {raise → λx.λk.x}
  (handler {choose → λx.λk.(k (fst x))}
    λ_.let x = (perform choose (20,30) in
       let y = (f x) in
       (f y)
```

このプログラムは例外を起こす関数を定義したのち、二つのハンドラの下で計算を実行する。

以下の 3 つのプログラムは上のプログラムに対応する F^{ϵ}+open プログラムである。これらの F^{ϵ}+open プログラムの型を消去して得られる式は上の λ^{ϵ} プログラムに一致する。$\Sigma = \{amb : \{choose : (int, int) \to int\}, exn : \{raise : () \to int\}\}$ を仮定する。

```
let f = Λ μ^eff.λ^⟨exn | μ⟩ x.1 + perform^⟨exn | μ⟩ raise () in
(handler^⟨⟩ {raise → λ^⟨⟩x.λ^⟨⟩k.x}
  (handler^⟨exn⟩ {choose → λ^⟨exn⟩x.λ^⟨exn⟩k.(k (fst x))}
    λ^⟨amb,exn⟩_.let x = perform^⟨amb,exn⟩ choose (20,30) in
              let y = f[⟨amb⟩] x in
```

```
                f[⟨amb⟩] y))

let f = λ⟨exn⟩x.1 + perform⟨exn⟩ raise () in
(handler⟨⟩ {raise → λ⟨⟩x.λ⟨⟩k.x}
  (handler⟨exn⟩ {choose → λ⟨exn⟩x.λ⟨exn⟩k.(k (fst x))}
    λ⟨amb,exn⟩_.let x = perform⟨amb,exn⟩ choose (20,30) in
                let y = open [⟨exn⟩,⟨amb,exn⟩](f) x in
                open [⟨exn⟩,⟨amb,exn⟩](f) y))

let f = λ⟨exn⟩x.1 + perform⟨exn⟩ raise () in
(handler⟨⟩ {raise → λ⟨⟩x.λ⟨⟩k.x}
  (handler⟨exn⟩ {choose → λ⟨exn⟩x.λ⟨exn⟩k.(k (fst x))}
    λ⟨amb,exn⟩_.let x = (perform⟨amb,exn⟩ choose (20,30)) in
    under [⟨exn⟩,⟨amb,exn⟩](
      let y = f x in
      f y)))
```

これらの中で、最も望ましいプログラムは三つ目のプログラムである。一つ目のプログラムはエフェクト注釈が閉じたエフェクトではない点が、二つ目のプログラムは open が多く呼ばれる点が三つ目に比べて非効率である。

ハンドルされるサンクの本体の型検査はエフェクト列 ⟨amb,exn⟩ の下で行われる。関数適用では文脈のエフェクト列と適用される関数の関数型のエフェクトが一致しなければならない。3 つのプログラムの関数適用はそれぞれの戦略でこの要求を満たすようにしている。

一つ目のプログラムは関数のエフェクト注釈を多相的にし、関数呼び出しでは適切なエフェクト列を型パラメータとして適用する。関数適用の文脈のエフェクト列は ⟨amb,exn⟩ であるから、f の呼び出しの前に ⟨exn⟩ を型適用し、エフェクト列を合わせている。このプログラムは open や under を使わないがエビデンスベクタの参照（オペレーション呼び出し）の効率が悪い。

二つ目のプログラムは関数のエフェクト注釈には閉じたエフェクト列を割り当て、それぞれの関数呼び出しで open を用いてエフェクト変換をする。一つ目と同様に関数適用の文脈のエフェクト列は ⟨amb,exn⟩ である。f の関数型のエフェクト列を文脈のエフェクト列に合わせるために open を用いる。

三つ目のプログラムは、二つ目と同様に閉じたエフェクト注釈をもつ関数を定義する。しかし、open を使ってエフェクト列を変換するのではなく、amb が必要なくなるところで under を使い、文脈のエフェクト列を変換する。これによって関数適用の文脈のエフェクト列が f のエフェクト列に一致して、open を使う必要はなくなる。

現在の Koka コンパイラは二つ目のプログラムにエビデンス変換を適用し低水準なプログラムを生成する。open floating は二つ目のようなプログラムを変換し、文脈のエ

フェクト列が不要なエフェクトラベルを持たないようにする。

第 4 章

Open Floating

4.1 概要

Open floating は F^{ϵ}+open プログラムに出現する open と under の総数を減らすプログラム変換である。例えば、open floating は一つ目のプログラムを二つ目のプログラムに変換する。ここで `calc` の型は `int` $\to \langle\rangle$ `int` で、`set` は `state` エフェクトのオペレーションであると仮定する。

```
let x = open[⟨⟩,⟨state⟩](calc)(
          open[⟨⟩,⟨state⟩]calc(3)) in
perform⟨state⟩ set x
```

```
let x = under[⟨⟩,⟨state⟩](calc(calc(3))) in
perform⟨state⟩ set x
```

一つ目のプログラムでは、関数のエフェクトを変換し、大きくすることで型がつく。そのためそれぞれの関数適用に open がともない、open が多く出現する。

二つ目のプログラムでは、関数のエフェクトを変換するのではなく、文脈のエフェクトを小さくすることで式に型がつく。隣接する部分式が同じエフェクトを必要とするならば、文脈のエフェクトを一回変換することで型がつき、エフェクト変換の回数を減らすことができる。このように、式に型がつくための「最小の環境」を導出する手法は主要な型付け [6] として議論されている。

この観察をもとに open floating を定義する。つまり、open を除去しそれぞれの部分式に最小のエフェクト割り当てることでエフェクト変換の回数を減らす。

4.2 準備

Open floating の定義を与える前に、定義に必要な補助関数などを定義する。

定義 2 (エフェクト要求)**.** エフェクト要求 ϕ を次のように定める。

$$\begin{aligned} \phi ::= & \epsilon && \text{(エフェクト列)} \\ | & \bot && \text{(最小元)} \end{aligned}$$

エフェクト要求は、式に型をつけるために必要なエフェクトを表現するために用いる。$\bot$ は「要求がないこと」を表すエフェクト要求である。

次にエフェクト列とエフェクト要求に順序関係を定める。

定義 3 (エフェクト列の順序)**.** エフェクト列の関係 $\sqsubseteq_\epsilon$ を次の規則から帰納的に定まる関係として定義する。

$$\frac{\epsilon_1 \equiv \epsilon_2}{\epsilon_1 \sqsubseteq_\epsilon \epsilon_2}$$

$$\frac{}{\langle l_1, \ldots, l_n \rangle \sqsubseteq_\epsilon \langle l_1, \ldots, l_n \mid \epsilon \rangle}$$

$\equiv$ は図 *3.2* で定義したエフェクト列の同値関係である。本稿では $\epsilon_1 \sqsubseteq_\epsilon \epsilon_2$ が成り立つことを ϵ_1 は ϵ_2 より小さいという。

定義 4 (エフェクト要求の順序)**.** エフェクト要求の関係 $\sqsubseteq$ を次の規則から帰納的に定まる関係として定義する。

$$\frac{}{\bot \sqsubseteq \phi}$$

$$\frac{\epsilon_1 \sqsubseteq_\epsilon \epsilon_2}{\epsilon_1 \sqsubseteq \epsilon_2}$$

本稿では $\phi_1 \sqsubseteq \phi_2$ が成り立つことを ϕ_1 は ϕ_2 より小さいという。

命題 1 ($\sqsubseteq$ の性質)**.** $\sqsubseteq$ は同値なエフェクト列を同一視したとき、半順序である。つまり、$\sqsubseteq$ はエフェクト列の等しさを $=$ ではなく $\equiv$ で判定するならば、反射律、推移律、反対称律を満たす。

Proof. 定義から明らか。 □

次にエフェクト変換をプログラムに挿入するためのメタな補助関数を定める。

定義 5 (部分関数 under', open')**.** $\phi_1 \sqsubseteq \phi_2$、$e$ を $F^\epsilon + open$ の式とする。

$$under'[\phi_1, \phi_2](e) = \begin{cases} e & (\phi_1 = \phi_2) \\ e & (\phi_1 = \bot) \\ under[\phi_1, \phi_2](e) & (\phi_1 \sqsubseteq \phi_2) \\ undefined & (otherwise) \end{cases}$$

open' はエフェクト列のみに対して定める。

$$open'[\epsilon_1, \epsilon_2](e) = \begin{cases} e & (\epsilon_1 = \epsilon_2) \\ open[\epsilon_1, \epsilon_2](e) & (\epsilon_1 \sqsubseteq \epsilon_2) \\ undefined & (otherwise) \end{cases}$$

命題 2. $open'[\phi_1, \phi_2](e), under'[\phi_1, \phi_2](e)$ は $\phi_1 \sqsubseteq \phi_2$ のとき、F^ϵ*+open* の式を返す。

Proof. まず、$under'[]$(に) 対して示す。$\phi_1 \equiv \phi_2$ のとき、e は F$^\epsilon$+open の式であるから主張は成り立つ。$\phi_1 = \bot$ のときも同様である。それ以外のとき、$\sqsubseteq$ の定義から $\phi_1 \equiv \langle l_1, \ldots, l_n \rangle, \phi_2 \equiv \langle l_1, \ldots, l_n \mid \epsilon \rangle$ とかける。$\mathrm{under}[\langle l_1, \ldots, l_n \rangle, \langle l_1, \ldots, l_n \mid \epsilon \rangle](e)$ は F$^\epsilon$+open の式であるから主張は成り立つ。次に、$open'[]$(に) 対して示す。$\epsilon_1 = \epsilon_2$ のとき、e は F$^\epsilon$+open の式であるから主張は成り立つ。それ以外のとき、$\sqsubseteq$ の定義から $\epsilon_1 \equiv \langle l_1, \ldots, l_n \rangle, \epsilon_2 \equiv \langle l_1, \ldots, l_n \mid \epsilon \rangle$ とかける。$\mathrm{open}[\langle l_1, \ldots, l_n \rangle, \langle l_1, \ldots, l_n \mid \epsilon \rangle](e)$ は F$^\epsilon$+open の式であるから主張は成り立つ。 □

Open floating の定義ではエフェクト要求の最小上界を用いる。

注記 1 (エフェクト要求の最小上界)**.** ϕ がエフェクト要求 $\phi_1, \ldots, \phi_n$ の最小上界であるとは、以下が成り立つことである。

- $\phi_i \sqsubseteq \phi$ *(for all* $i = 1, \ldots, n$*)*
- 任意の ϕ' に対して、$\phi_i \sqsubseteq \phi'$ *(for all* $i = 1, \ldots, n$*)* ならば $\phi \sqsubseteq \phi'$

本項では $\phi = \bigsqcup \phi_1, \ldots, \phi_n$ とかく。

Open floating は F$^\epsilon$+open の型つく式、値、ハンドラのそれぞれに対して補助関数を定義することで定める。

- 式に対しての補助関数 $(flag, \Gamma, e) \rightsquigarrow_E (e', \tau, \phi)$ はフラグ、型環境、式、その型、エフェクト列を引数にとり、変換後の式、その型、エフェクト要求の組みを返す。フラグには二種類の値、**able**, **disable** があり、式の open を除去するか否かを表す。
- 値に対する補助関数 $(\Gamma, v) \rightsquigarrow_V (v', \tau)$ は型環境、値、その型を引数にとり、変換後の値とその型の組を返す。

次に補助関数の不変条件、命題 3、4、5 を述べる。これらの不変条件から open floating がプログラムの意味を保つことを示せる。

定理 3 (Open floating は意味を保つ)**.** F^ϵ*+open* プログラムに *open floating*$\rightsquigarrow$ を適用したとき、プログラムに付く型と、型の消去で得られる λ^ϵ プログラムは変わらない。つまり

$(\Gamma_0, e) \rightsquigarrow e'$ かつ $\Gamma_0 \vdash_{exp} e : \sigma \mid \langle\rangle$ ならば、

- $\Gamma_0 \vdash_{exp} e : \sigma \mid \sigma$ かつ

- $e^* = e'^*$

が成り立つ。

Proof. 一つ目の結論は命題 3、4 から、二つ目の結論は命題 5 から導かれる。　□

命題 3 (変換後の式の型付け)**.** 変換後の式、値は入力の型環境と出力のエフェクト要求に対応するエフェクト列の下で、出力の型がつく。$\perp$ に対応するエフェクト列は任意のエフェクト列とする。つまり、

- $(\mathit{flag}, \Gamma, e) \rightsquigarrow_E (e', \tau, \phi)$ ならば
  $\Gamma \vdash_{exp} e' : \tau \mid \phi$ *(if* $\phi \neq \perp$*)*
  $\Gamma \vdash_{exp} e' : \tau \mid \epsilon$ *(for any* ϵ*) (if* $\phi = \perp$*)*
  が成り立つ。
- $(\Gamma, v, \tau) \rightsquigarrow_V (v', \tau')$ ならば $\Gamma \vdash_{val} v' : \tau'$ が成り立つ。

命題 4 (変換前後の型付けの関係)**.** *Open floating* を適用することで、割り当てられるエフェクトは変わらないか小さくなる。つまり以下が成り立つ。

- $(\boldsymbol{able}, \Gamma, e) \rightsquigarrow_E (e', \sigma, \phi)$ かつ $\Gamma \vdash_{exp} e : \tau_1 \to \epsilon_{fun}\tau_2 \mid \epsilon$ ならば、ある ϵ'_{fun} に対して $\sigma = \tau_1 \to \epsilon'_{fun}\tau_2$ かつ $\epsilon'_{fun} \sqsubseteq \epsilon_{fun}$ かつ $\phi \sqsubseteq \epsilon$ が成り立つ。
- $(\boldsymbol{disable}, \Gamma, e) \rightsquigarrow_E (e', \sigma', \phi)$ かつ $\Gamma \vdash_{exp} e : \sigma \mid \epsilon$ ならば $\sigma = \sigma'$ かつ $\epsilon' \sqsubseteq \epsilon$ が成り立つ。
- $(\Gamma, v) \rightsquigarrow_V (v', \sigma')$ かつ $\Gamma \vdash_{val} v : \sigma$ ならば $\sigma = \sigma'$ が成り立つ。

命題 5 (変換はプログラムの意味を保存する)**.** 変換前後の式から型を消去して得られる λ^ϵ プログラムは一致する。つまり以下の命題が成り立つ。

- $(\mathit{flag}, \Gamma, e) \rightsquigarrow_E (e', \tau, \phi)$ ならば $e^* = e'^*$
- $(\Gamma, v, \tau) \rightsquigarrow_V (v', \tau')$ ならば $v^* = v'^*$

命題 3 と命題 4 は付録に記した命題**??**とともに、式の構造に関する帰納法で示せると考えられる。それらの結果を用いて命題 5 を示すことができると見込まれる。

4.3　Open Floating の定義

Open floating は適用される関数にかかる open を除去しつつ、命題 3、4、5 を満たすように定義する。

Open floating$\rightsquigarrow$ は型環境 Γ_0 と式 e を引数にとり、式 e' を返す。

$$\frac{(\mathbf{disable}, \Gamma_0, e) \rightsquigarrow_E (e', \tau, \phi)}{(\Gamma_0, e) \rightsquigarrow e'} \qquad (\textsc{open floating})$$

つまり、F^{ϵ}+open の式 e、型環境 Γ_0 に対する open floating はフラグを **disable** として、式に対する補助関数 $\rightsquigarrow_E$ の呼び出しが生成するプログラム e' である。フラグを **disable** とすることで open floating が式の型を保存することが導ける。

補助関数では入力式の部分式に対して補助関数 $\rightsquigarrow_E$ を再帰的に適用し、得られた部分式の変換結果とエフェクト変換を組み合わせることで入力式の変換結果を得る。以降では補助関数の定義で特筆すべき点を説明する。

■open, under

$$\frac{(\mathbf{able}, \Gamma, e) \rightsquigarrow_E (e', \sigma_1 \to \epsilon'\sigma_2, \phi)}{(\mathbf{able}, \Gamma, \mathrm{open}[\langle l_1, \ldots, l_n\rangle, \langle l_1, \ldots, l_n \mid \epsilon\rangle](e)) \rightsquigarrow_E (e', \sigma_1 \to \epsilon'\sigma_2, \phi)} \text{(OPEN の除去)}$$

フラグが **able** のとき、open を除去する。

$$\frac{(\mathbf{disable}, \Gamma, e) \rightsquigarrow_E (e', \sigma_1 \to \epsilon'\sigma_2, \phi)}{\begin{array}{l}(\mathbf{disable}, \Gamma, \mathrm{open}[\langle l_1, \ldots, l_n\rangle, \langle l_1, \ldots, l_n \mid \epsilon\rangle](e)) \\ \rightsquigarrow_E (\mathrm{open}[\langle l_1, \ldots, l_n\rangle, \langle l_1, \ldots, l_n \mid \epsilon\rangle](e'), \sigma_1 \to \epsilon'\sigma_2, \phi)\end{array}} \text{(OPEN の保存)}$$

フラグが **disable** のとき、open をつけたままにする。

$$\frac{(\mathit{flag}, \Gamma, e) \rightsquigarrow_E (e', \sigma, \phi)}{(\mathit{flag}, \Gamma, \mathrm{under}[\langle l_1, \ldots, l_n\rangle, \langle l_1, \ldots, l_n \mid \epsilon\rangle](e)) \rightsquigarrow_E (e', \sigma, \phi)} \quad \text{(UNDER)}$$

under は常に除去する。

■関数適用式

$$\frac{(\mathbf{able}, \Gamma, e_1) \rightsquigarrow_E (e_1', \tau_2 \to \epsilon\tau, \phi_1) \qquad (\mathbf{disable}, \Gamma, e_2) \rightsquigarrow_E (e_2', \_, \phi_2) \qquad \phi = \bigsqcup \epsilon, \phi_1, \phi_2}{(\mathit{flag}, \Gamma, e_1\, e_2) \rightsquigarrow_E (\mathit{under}'[\phi_1, \phi](\mathit{open}'[\epsilon, \phi](e_1'))\, \mathit{under}'[\phi_2, \phi](e_2'), \tau, \phi)} \text{(関数適用)}$$

Open floating では関数（e_1）が open されているとき、その open を除去する。そのため e_1 に対する再帰呼び出しのフラグを、open を除去することを示す **able** とする。例えば $e_1 = \mathrm{open}[\epsilon_1, \epsilon_2](g) : \tau_2 \to \epsilon_2\tau$ のとき、$e_1' = g : \tau_2 \to \epsilon_1\tau$ となり、関数型のエフェクト列が小さくなる。一方 e_2 は関数適用式の型付けを保存するためにフラグを **disable** とする。

変換後の関数適用式のエフェクトは ϕ である。これは変換後の関数型のエフェクト、それぞれの部分式のエフェクトの最小上界である。最小上界が常に定まることは e_1e_2 に対する型付けと命題 4 から導かれると考えられる。証明のアイデアを付録で与える。

■let 式

$$\frac{(\mathbf{disable}, \Gamma, e_1) \rightsquigarrow_E (e_1', \tau_1, \phi_1) \qquad (\mathit{flag}, (\Gamma, x : \tau_1), e_2) \rightsquigarrow_E (e_2', \tau_2, \phi_2) \qquad \phi = \bigsqcup \phi_1, \phi_2}{(\mathit{flag}, \Gamma, \mathrm{let}\, x = e_1 \,\mathrm{in}\, e_2) \rightsquigarrow_E (\mathrm{let}\, x = \mathit{under}'[\phi_1, \phi](e_1') \,\mathrm{in}\, \mathit{under}'[\phi_2, \phi](e_2'), \tau_2, \phi)} \text{(LET 式)}$$

let 式の変換結果は、関数適用と同様に部分式の変換結果とエフェクト変換の組み合わせである。`(let x = 3 in f) 3` のように、let 式が関数適用式のオペレータである場合、let 式の末尾位置である e_2 の評価結果が実行時の関数適用式のオペレータである。よって、e_2 のフラグは let 式のフラグとする。エフェクト要求の最小上界が常に存在することは関数適用式の場合と全く同様に示されると予想される。

■値

$$\frac{(\Gamma, v) \rightsquigarrow_V (v', \tau')}{(\mathit{flag}, \Gamma, v) \rightsquigarrow_E (v', \tau', \bot)} \text{(値)}$$

値を引数として、式に対する補助関数が呼ばれたとき、値に対する補助関数を呼び出すことで変換をする。式としての値は任意のエフェクト列の下で型付けできることに対応して、返すエフェクト要求は何も要求しないことを表す $\bot$ である。

■関数

$$\frac{(\mathbf{disable}, \Gamma, e) \rightsquigarrow_E (e', \tau', \phi)}{(\Gamma, \lambda^{\epsilon} x : \tau.e) \rightsquigarrow_V (\lambda^{\epsilon} x : \tau.\mathit{under}'[\phi, \epsilon](e'), \tau \to \epsilon\tau')} \text{(関数)}$$

関数に対する定義は型付けを保存するためのエフェクト変換の挿入が特徴的である。このエフェクト変換によって、open floating の適用前後で関数のエフェクトを変えずにすむ。

残りの定義を以下に示す。ハンドル式 $(\,\mathrm{handle}^{\,\epsilon}\ h\ e)$ に対する定義がないのは、open floating はハンドル式を変換対象にしないからである。ハンドル式は実行時に現れるフレームを表現するための式で、コンパイル時にプログラムに出現することはない。

$$\frac{x : \sigma \in \Gamma}{(\Gamma, x) \rightsquigarrow_V (x, \sigma)} \text{(変数)}$$

$$\frac{((\Gamma, \alpha^k), v) \rightsquigarrow_V (v', \sigma)}{(\Gamma, \Lambda\alpha^k.v) \rightsquigarrow_V (\Lambda\alpha^k.v', \forall\alpha^k.\sigma)} \text{(型抽象)}$$

$$\frac{(\mathbf{disable}, \Gamma, e) \rightsquigarrow_E (e', \forall\alpha^k.\sigma, \phi)}{(\mathit{flag}, \Gamma, e[\sigma']) \rightsquigarrow_E (e'[\alpha := \sigma'], \sigma[\alpha := \sigma'], \phi)} \text{(型適用)}$$

$$\frac{op : \forall\overline{\alpha}.\sigma_1 \to \sigma_2 \in \Sigma(l)}{(\Gamma,\ \mathrm{perform}^{\ \epsilon}\ op\ \overline{\sigma}) \rightsquigarrow_V (\ \mathrm{perform}^{\ \epsilon}\ op\ \overline{\sigma}, \sigma_1[\overline{\alpha} := \overline{\sigma}] \to \langle l \mid \epsilon\rangle \sigma_2[\overline{\alpha} := \overline{\sigma}])}$$

（オペレーション）

$$\frac{\Sigma(l) = \{op_i : \forall\overline{\alpha}.\sigma_i^{in} \to \sigma_i^{out}\}_{i=1,\ldots,n} \qquad (\Gamma, f_i) \rightsquigarrow_V (f_i', \forall\overline{\alpha}.\sigma_i^{in} \to \sigma_i^{out}\})}{(\Gamma,\ \mathrm{handler}^{\ \epsilon}\ \{op_i \to f_i\}_{i=1,\ldots,n}) \rightsquigarrow_V (\ \mathrm{handler}^{\ \epsilon}\ \{op_i \to f_i'\}_{i=1,\ldots,n}, (() \to \langle l \mid \epsilon\rangle \sigma) \to \epsilon\sigma)}$$

（ハンドラ式）

第 5 章

関連研究

5.1 エフェクトシステムと代数的エフェクトを持つ言語

生じる計算エフェクトを表現するために、それぞれの言語のエフェクトシステムは計算エフェクトの「型」を定義する。そのような型 —ここでは「エフェクト」と呼ぶことにする—は集合として表現されるか、あるいは列として表現されることが多い。

エフェクトを集合として表現する言語に Effekt 言語 [1] や Eff 言語 [13] やそれから派生した ImpEff[7] がある。

Effekt[1] のエフェクトシステムや評価規則は Koka と大きく異なるが、コンパイル手法は類似している。Effekt 言語では表面言語のプログラムを変換して、キャパビリティというハンドラの実装を渡すプログラムを生成する。これは Koka 言語のエビデンスベクタに類似している。しかし、エビデンスベクタは複数のハンドラの実装を保持する一方で、キャパビリティは個々のハンドラの実装に対応する。そのため、キャパビリティを変換するようなことはなく、open floating のような最適化を適用する機会はないと考えられる。

Eff や ImpEff は ML のような構文をもつ言語でエフェクトシステムは部分型付けのようなことをする。ImpEff は Eff の最適化を安全に行うために考えられた言語で、Eff と同様に部分型付けベースのエフェクトシステムをもつ。ImpEff に対して中間言語 ExEff が定義されており、型強制を項に明示する。ExEff は MultiCore OCaml[3, 4, 2] をターゲット言語の一つにすることを目標にしている。MultiCore OCaml は代数的エフェクトを備える一方で、エフェクトシステムを備えていない。

Frank[10] や Links[5] もエフェクトシステムと代数的エフェクトを備えた言語である。エフェクトを表現するために、Koka と同様に列型を用いている。Koka では列に重複する要素が現れることを許すが Frank ではそれを許さない。その代わりに、列型にある要素が現れない、などといった制約を表現できるようにすることで、エフェクトシステムを実現している。

5.2　主要な型付け

型付け可能な式に型を付けるために必要な最小の型環境が存在するという性質を、主要な型付け性質という [6]。

open floating のアイデアはこれにならっており、式に必要な最小のエフェクト列を割り当て直す。主要な型付けには、分割コンパイルやエラーメッセージの改善などへの応用が提案されている。

第 6 章

今後の課題

本稿では Koka コンパイラに対する最適化手法として open floating を提案した。

Open floating の定義は F^{ϵ}+open に対してのみしか済んでいない。コンパイラに実装し、最適化を評価することが直近の課題である。

また Koka コンパイラの型推論は形式化も課題の一つである。過去の Koka 言語の型推論の形式化は提案されているが [8]、現在の Koka コンパイラの振る舞いを表すためには不十分である。Koka コンパイラは表面言語での型付けをもとに中間表現を生成するため、型推論の形式化をすることで open floating に関連した更なる議論ができると期待される。

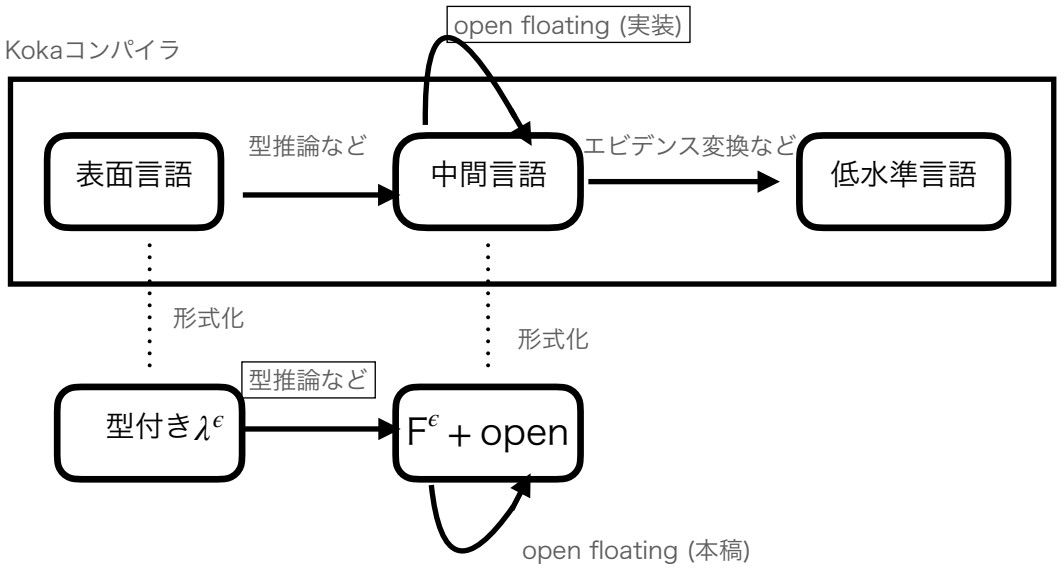

図 6.1　Koka コンパイラと形式化

第 7 章

まとめ

本稿では Koka コンパイラに対する最適化手法として open floating を設計した。Koka コンパイラは代数的エフェクトを高速に実行するために型付けに依存したプログラム変換を行う。コンパイラに open floating を実装することでエフェクト割り当てを最適化し、プログラムの実行速度を向上する事が期待される。

参考文献

[1] Brachthäuser, J. I., Schuster, P. and Ostermann, K.: Effekt: Lightweight Effect Polymorphism for Handlers, Technical report, Technical Report. University of Tübingen, Germany (2020).

[2] Dolan, S., Eliopoulos, S., Hillerström, D., Madhavapeddy, A., Sivaramakrishnan, K. and White, L.: Concurrent system programming with effect handlers, *International Symposium on Trends in Functional Programming*, Springer, pp. 98–117 (2017).

[3] Dolan, S., White, L. and Madhavapeddy, A.: Multicore ocaml, *OCaml Workshop*, Vol. 2 (2014).

[4] Dolan, S., White, L., Sivaramakrishnan, K., Yallop, J. and Madhavapeddy, A.: Effective concurrency through algebraic effects, *OCaml Workshop*, p. 13 (2015).

[5] Hillerström, D. and Lindley, S.: Liberating effects with rows and handlers, *Proceedings of the 1st International Workshop on Type-Driven Development*, pp. 15–27 (2016).

[6] Jim, T.: What are principal typings and what are they good for?, *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 42–53 (1996).

[7] KARACHALIAS, G., PRETNAR, M., SALEH, A. H., VANDERHALLEN, S. and SCHRIJVERS, T.: Explicit effect subtyping, *Journal of Functional Programming*, Vol. 30 (2020).

[8] Leijen, D.: Koka: Programming with Row Polymorphic Effect Types, Proceedings 5th Workshop on *Mathematically Structured Functional Programming,* Grenoble, France, 12 April 2014 (Levy, P. and Krishnaswami, N.(eds.)), Electronic Proceedings in Theoretical Computer Science, Vol. 153, Open Publishing Association, pp. 100–126 (2014).

[9] Leijen, D.: Type directed compilation of row-typed algebraic effects, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pp. 486–499 (2017).

[10] Lindley, S., McBride, C. and McLaughlin, C.: Do be do be do (2017).

[11] Pierce, B. C. and Benjamin, C.: *Types and programming languages*, MIT press (2002).

[12] Plotkin, G. D. and Pretnar, M.: Handling algebraic effects, *arXiv preprint arXiv:1312.1399* (2013).

[13] Pretnar, M.: An introduction to algebraic effects and handlers. invited tutorial paper, *Electronic notes in theoretical computer science*, Vol. 319, pp. 19–35 (2015).

[14] Xie, N., Brachthäuser, J. I., Hillerström, D., Schuster, P. and Leijen, D.: Effect handlers, evidently, *Proceedings of the ACM on Programming Languages*, Vol. 4, No. ICFP, pp. 1–29 (2020).

[15] 池守和槻: 代数的エフェクトを持つ計算体系 System F ε へのエフェクト強制の導入と健全性の証明, 東京工業大学学士論文 (2021).