平成 29 年度

筑波大学情報学群情報科学類

卒業研究論文

題目 合流点を追加した
コンパイラ中間言語の設計と検証

主専攻　　ソフトウェアサイエンス

著者　　河原 悟

指導教員 亀山 幸義, 海野 広志

要　旨

高階のプログラム言語を対象としたコンパイラでは、 CPS(Continuation Passing Style) や ANF(A-Normal Form) といった中間言語が多く用いられてきた。 CPS と ANF は実行順序の明示など優れた性質を持つ一方、コントロールフローの合流点を扱う仕組みをもたない。そのため、 CPS や ANF に基づいた最適化では、コードサイズの爆発などの問題がおきる。そこで Maurer らは、必要呼びの体系を対象とした、合流点を明示的に扱うことで CPS と ANF の優れた点を模倣できる中間言語 System F_J を提案した。本研究では、 System F_J を踏まえて、明示的な合流点を持つ値呼びの関数型言語に例外機構を追加したものを対象としたコンパイラ中間言語として、 *Joel* を提案する。 Joel が持つ明示的な合流点を利用することで、 CPS や ANF では表現できない最適化を表現できる。 Joel における最適化の効果を検証するため、 CPS を中間言語に採用した場合の最適化との比較実験をおこなった。ターゲット言語に OCaml のサブセットである Core ML という言語を新たに定義し、生成されたコードの実行速度とコードサイズを比較項目とした。実験の結果、同程度の実行速度でサイズの小さなプログラムを生成することに成功した。

目次

図目次

第 1 章

序論

1.1 背景

現代の多くの最適化コンパイラは、ソース言語とターゲット言語のあいだに、 中間言語を用い、中間言語の上で様々な最適化を適用するという構成をとる。これにより、最適化アルゴリズムとその正しさを中間言語上で正確に表現しやすいという利点が得られる。

CPS や SSA などのように、目的や抽象度の異なる様々な中間言語が存在する。なかでも CPS(Continuation Passing Style、継続渡し方式) は Appel の著書『Compiling with Continuation』 [1] にあるように、 ML などの高階関数型言語のコンパイラ中間言語として広く採用されてきた。

また、 ANF(A-Normal Form、 A 正規形) も関数型言語コンパイラの中間言語として広く知られている。直接形式で人間にとって直感的な表現であり、また途中の計算を全て変数に束縛することにより最適化がおこないやすくなる [3]。

プログラムを制御フローグラフとして表現したとき、分岐したフローが再び合流する点を合流点と呼ぶ。CPS や ANF は合流点を表現するための明示的な仕組みをもたない。合流点を持つプログラムを CPS や ANF で表現すると、コードのコピーが発生してコードサイズが増大するという問題がある。 Maurer らは明示的な合流点を持つ中間言語 *System* F_J [6] を提案し、合流点を明示的に扱うことで、 CPS と ANF が抱える問題を解決した。そして CPS や ANF でおこなえないような最適化により、高速に動作するターゲット言語へのコンパイルが可能となった。

1.1.1 合流点

プログラム 1.1の `j4`、 `j5`のように、分岐した制御フローが再び合流する部分が存在する。これを**合流点** (*join point*) と呼び、プログラムの中の異なる制御フローが合流する地点を指す。また合流点は、制御フローグラフにおける、複数の基本ブロックからのパスをを持つ基本ブロックと考えることができ (図 1.1)、 1 つのプログラム中にも合流点は多く存在することが分かる。合流点はコンパイラの研究においてもよく取り扱われており、例えば、 SSA(Static Single Assignment、静的単一代入) 形式は、変数に対する代入がたかだか一回という性質を保つため、合流点において ϕ 関数という特別な関数を挿入したものである。 SSA は主に手続き型言語を対象としたコンパイラ中間言語として広く用いられており、 LLVM [5] や COINS [9] では内部表現として

SSA を採用している。

プログラム 1.1　プログラムにおける合流点の例

```
let j = fun x -> M in
if e1 then j e2
else j e3
```

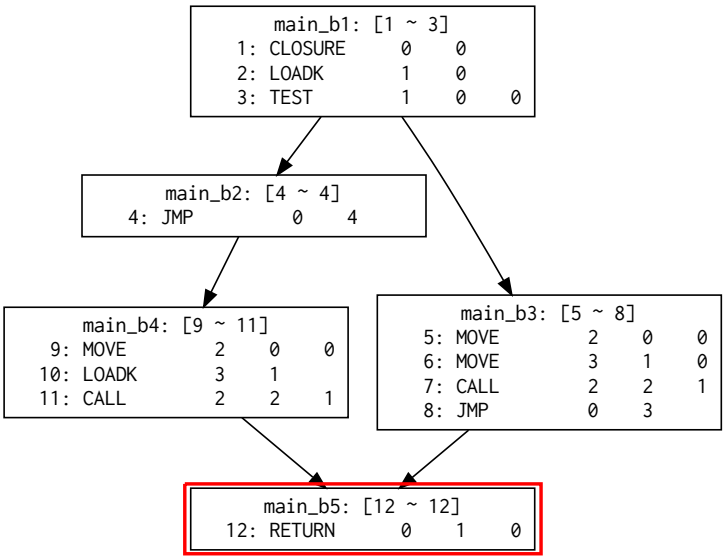

図 1.1　制御フローグラフにおける合流点の例

1.1.2　関数型言語における合流点の表現

関数型言語の中間言語には CPS や ANF がよく用いられている。その一方で CPS と ANF には合流点を扱う仕組みがない。そのため、 CPS や ANF に基づいた最適化では、合流点に相当する関数の展開などにより、コードサイズの爆発などの問題がおきる。

Maurer らは、明示的に合流点を扱うことができるコンパイラ中間言語 *System F_J* を提案した。 System F_J は合流点を通常の関数とは構文的に区別している。これにより、合流点を持つプログラムを、 JUMP 命令のような軽量なコントロール操作にコンパイルすることができる。 CPS による制御フローのエンコードを System F_J は合流点で表現でき、さらには合流点を考慮することで、 CPS と同等以上の最適化がおこなえる。また、System F_J は ANF と同様に直接形式であるため、 CPS ではおこないにくい最適化、例えば共通部分式除去 (Common Subexpression Elimination) も可能となる。そして、可換変換 (commuting conversion) によるコードの肥大化を、合流点を導入することにより抑えることができる (図 1.2、プログラム 1.2)。一方で、 System F_J は Haskell 言語、つまり必要呼びでかつ純粋な (副作用のない) 言語を対象としたコンパイラ中間言語として設計されている。そのため、値呼びの非純粋な言語において明示的な合流点を用いた最適化が有効かどうかにつ

いては触れられていない。

```
e1 + e2 + (match e3 with
           | P -> e4
           | Q -> e5)
```

commuting conversion ⇒

```
match e3 with
| P -> e1 + e2 + e4
| Q -> e1 + e2 + e5
```

図 1.2　可換変換の例

プログラム 1.2　合流点によるコード肥大化の解消

```
join j e = e1 + e2 + e in
match e3 with
| P -> j e4
| Q -> j e5
```

1.2　本研究の目的

本研究では、非純粋な値呼びの関数型言語およびその処理系において、 Maurer らの手法が有効かどうかを検証する。検証の方法として、コンパイラの中間言語に合流点を追加した場合のパフォーマンスおよび出力されるターゲット言語のコードサイズを測定する。 Maurer らが対象とした Haskell 言語のように必要呼びの評価戦略を用いる関数型言語の CPS 変換 [7] は値呼びの言語を CPS 変換するよりも複雑である。一方で値呼びの評価戦略を用いる関数型言語では、中間言語に CPS を選択することは自然である。値呼びの言語において中間言語に CPS を用いる場合との比較をおこない、新たな中間表現の選択肢として有効かどうかを検討する。

検証をおこなうために、非純粋な値呼びの言語 *Joel* を設計、実装する。 System F_J が対象としている純粋な言語と広く使われている非純粋な言語との隔たりを埋める第一歩として、コントロールエフェクトがあり多くのプログラム言語が持つ機能である例外機構を副作用の一例として導入した。また、 System F_J における操作的意味論や最適化の規則を値呼びの場合において妥当となるように変更を加えた。

Joel による最適化の評価のため、 CPS による最適化で得られるプログラムとの性能比較をおこなう (図 1.3)。ソース言語を Joel/CPS に変換し、それぞれ最適化をおこなう。そして最適化されたプログラムからターゲット言語を生成し、生成されたプログラムのコードサイズ、実行速度を測定することで、 Joel による最適化の評価をおこなう。今回はソース言語として Core ML (4.1 節)、ターゲット言語として OCaml を選択した。

本論文の構成

本論文の構成は以下の通りである。第 2 章では本研究で用いる言語 Joel の構文と意味論を定義する。第 3 章では Joel 上でおこなう最適化について説明し、第 4 章では本研究で用いたソース言語 Core ML および比較対象となる中間言語 CPS の定義と、 Joel/CPS 上での最適化を経たプログラムの性能比較をおこなう。第 5 章ではまとめと今後の課題について述べる。

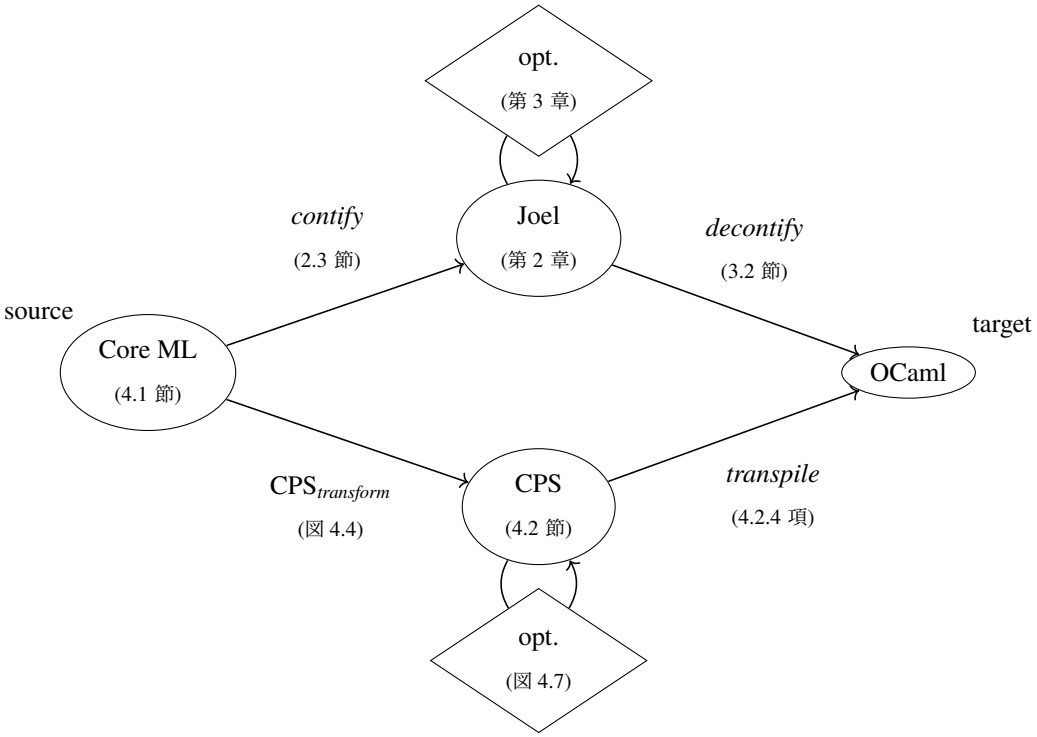

図 1.3　概略図

本研究では Joel およびその実験コードを OCaml により実装した。ソースコードは `https://github.com/Nymphium/joel/tree/bachelor_thesis_poc` にある。

第 2 章

対象言語の構成

本研究では、 Maurer らが提案した System F_J をもとにして、例外処理機構を含む値呼びのソース言語に対応した中間言語 *Joel* を設計し、後の実験に用いる。

System F_J からの Joel の主な変更点は、評価戦略が値呼びであること、例外処理機構を持つことである。System F_J は Haskell 言語のように純粋な必要呼びの言語を想定している。一方で現在広く使われているプログラム言語は非純粋な値呼びの言語が主流である。 System F_J のキーコンセプトである明示的な合流点を用いた最適化が Python や Scala、 OCaml などといった非純粋な値呼びのプログラム言語でも有効かどうかを検証するため、合流点を持つ非純粋な値呼びの中間言語を定義し、その言語について論じる。

2.1 構文

Joel の抽象構文を図 2.1に示す。 単純ラムダ計算にデータ型およびパターンマッチ、 **let** 式、再帰的な定義を表す **let rec** 式、例外処理を追加した言語となっている。加えて合流点を明示する **join** 式、再帰的な合流点を表す **join rec** 式、合流点へのジャンプを表す **jump** 式がある。 e, u は項、 v は値、 K はデータコンストラクタを表し、 Ex は例外を表すメタ変数である。 x は項における変数、 j はラベルと呼ばれる、合流点を示すための記号を表す。 E は評価文脈、 E_{free} は後述する制限された評価文脈、 L は末尾文脈を表す。 s は後述する操作的意味論の定義で用いる評価スタックを表す。 vb は **let** 式および **let rec** 式で束縛される値に関して言及しない場合に用いる。 jb も同様に、 **join** 式および **join rec** 式で束縛される項に言及しない場合に用いる。なお、再帰的定義の **let rec** 式および **join rec** 式も含めて **let** 式、 **join** 式と呼ぶ。 $\overrightarrow{c}$ は c の列を表しており、例えば $\lambda\overrightarrow{x}.e$ は $\lambda x_1.\lambda x_2.\ldots\lambda x_n.e$ を表す。 alt は **match** 式のマッチケース $k\ \overrightarrow{x} \to u$ を表している。 exc は **try-with** 式の持つ例外ハンドラ $Ex\ \overrightarrow{x} \to u$ を表している。

Joel は System F_J に基づいて設計されており、以下が変更を加えた点である。まず例外の送出 **raise** $(Ex\ \overrightarrow{e})$、 **try-with** 式を追加した。評価文脈の定義に、 **try-with** を含まない F_{free} および E_{free} を追加し、値呼びの言語に沿うように変更した。

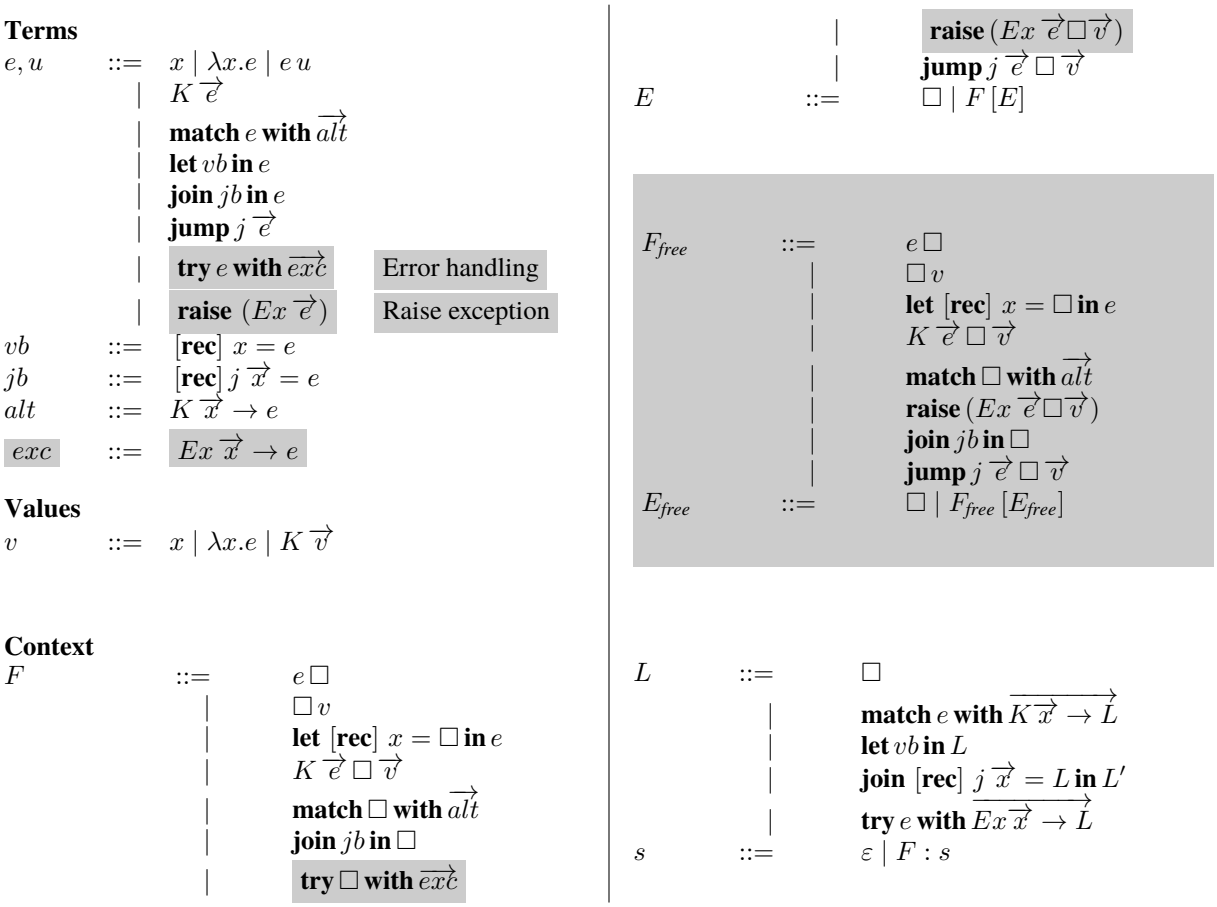

図 2.1　Joel の抽象構文

評価文脈、末尾文脈

評価文脈は、ある項を評価するときの、最初に計算がおこなわれる項を除いた、プレースホルダーを持つ項である。 `3 + 5`を例に取ると、最初に評価される `3`と評価文脈である `□ + 5`に分けることができる。

本論文では Maurer らによる System F_J の定義に則り、フレーム F、フレームを用いた評価文脈 E を再帰的に定義する。評価順序もここで定められている。 F の定義中の $e\,\square$、$\square\,v$ は、関数呼出しの項においてまず引数を評価し、そのあと関数を評価することを示している。また $K\ \overrightarrow{e}\ \square\ \overrightarrow{v}$ はデータコンストラクタの引数を右から順に評価していくことを表しており、この評価順序は OCaml の主要な処理系に倣っている。また評価戦略にも依存しており、例えば **let** 式で束縛される項も $\textbf{let}\,x = \square\,\textbf{in}\,e$ のように、束縛時に評価されることがここにより定められる。

末尾文脈は項の中で最後に評価される部分を指す。 $\textbf{let}\,x = e\,\textbf{in}\,u$ を例に考えると、値呼びであるため最初に e が評価され、その次に u が評価される。最後に評価される u と、それ以外の $\textbf{let}\,x = e\,\textbf{in}\,\square$ に分けられる。

評価文脈のうち $\textbf{try}\,\square\,\textbf{with}\,\overrightarrow{exc}$ を含まない E_{free} を本研究で新たに定義した。 E_{free} を新たに定義した理由お

よび Joel における利用方法は第 3 章で改めて述べる。

2.2 操作的意味論

Joel の操作的意味論は図 2.2のようになる。 Maurer らによる System $\mathrm{F_J}$ の定義を基に、値呼びの評価規則を定義した。また、例外処理に関する操作も定義した。 @ はリストの結合、 :: はリストへの要素の追加を表す

$$\langle F\,[e]\,;\ s;\ \Sigma\rangle \mapsto \langle e;\ F :: s;\ \Sigma\rangle \qquad \text{(Push)}$$

$$\langle v;\ F :: s;\ \Sigma\rangle \mapsto \langle F\,[v]\,;\ s;\ \Sigma\rangle \qquad (\text{Pop}_v)$$

$$\langle \lambda x.e;\ \square\, v :: s;\ \Sigma\rangle \mapsto \langle e;\ s;\ \Sigma, x = v\rangle \qquad (\beta)$$

$$\langle \textbf{let}\ [\textbf{rec}]\ x = v\ \textbf{in}\ e;\ s;\ \Sigma\rangle \mapsto \langle e;\ s;\ \Sigma, x = v\rangle \qquad \text{(Bind)}$$

$$\langle x;\ s;\ \Sigma\,[x = v]\rangle \mapsto \langle v;\ s;\ \Sigma\,[x = v]\rangle \qquad \text{(Look)}$$

$$\left\langle K\ \overrightarrow{v};\ \textbf{match}\ \square\ \textbf{with}\ \overrightarrow{alt} :: s;\ \Sigma\right\rangle \mapsto \begin{array}{c}\langle u;\ s;\ \Sigma, \overrightarrow{x = v}\rangle \\ \text{if mem}\left(\overrightarrow{alt}, (K\ \overrightarrow{x} \to u)\right)\end{array} \qquad \text{(Match)}$$

$$\left\langle \begin{array}{c}\textbf{jump}\ j\ \overrightarrow{v}; \\ s' @ (\textbf{join}\ [\textbf{rec}]\ j\ \overrightarrow{x} = u\ \textbf{in}\ \square :: s)\,; \\ \Sigma\end{array} \right\rangle \mapsto \begin{array}{c}\langle u;\ \textbf{join}\ [\textbf{rec}]\ j\ \overrightarrow{x} = u\ \textbf{in}\ \square :: s;\ \Sigma, \overrightarrow{x = v}\rangle \\ \text{if not mem}\,(s', \textbf{join}\ [\textbf{rec}]\ j\ \overrightarrow{x} = e\ \textbf{in}\ \square)\end{array} \qquad \text{(Jump)}$$

$$\langle v;\ \textbf{join}\ jb\ \textbf{in}\ \square :: s;\ \Sigma\rangle \mapsto \langle v;\ s;\ \Sigma\rangle \qquad \text{(Ans)}$$

$$\langle v;\ \textbf{try}\ \square\ \textbf{with}\ \overrightarrow{exc} :: s;\ \Sigma\rangle \mapsto \langle v;\ s;\ \Sigma\rangle \qquad \text{(Try-Value)}$$

$$\left\langle \begin{array}{c}\textbf{raise}\ (Ex\ \overrightarrow{v})\,; \\ s' @ (\textbf{try}\ \square\ \textbf{with}\ \overrightarrow{exc} :: s)\,; \\ \Sigma\end{array} \right\rangle \mapsto \begin{array}{l}\langle u;\ s;\ \Sigma, \overrightarrow{x = v}\rangle \\ \text{if mem}\,(\overrightarrow{exc}, (Ex\ (\overrightarrow{x}) \to u)) \\ \text{and not mem}\left(s', \left(\textbf{try}\ \square\ \textbf{with}\ \overrightarrow{exc'}\right)\right) \\ \textit{where} \\ \overrightarrow{exc'}\ \text{such that mem}\left(\overrightarrow{exc'}, (Ex\ \overrightarrow{x} \to u)\right)\end{array} \qquad \text{(Try-Catch)}$$

$$\langle \textbf{raise}\ (Ex\ \overrightarrow{v})\,;\ s;\ \Sigma\rangle \mapsto \begin{array}{l}\text{exit} \\ \text{if not mem}\,(s, (\textbf{try}\ \square\ \textbf{with}\ \overrightarrow{exc})) \\ \textit{where} \\ \overrightarrow{exc}\ \text{such that mem}\,(\overrightarrow{exc}, (Ex\ \overrightarrow{x} \to u))\end{array} \qquad \text{(Uncaught-Raise)}$$

図 2.2　Joel の操作的意味論

メタ演算子である。スタックはリストで表現され、スタック s_1 と s_2 の結合は $s_1 @ s_2$、フレーム F のスタック s のスタックトップへの追加は $F :: s$ として表現される。

$\text{mem}\,(l, m)$ はリスト状のデータ構造 l に m が含まれているかどうかを表す述語である。 $\text{fv}\,(e)$ はプログラム中の項 e に含まれる自由変数の集合を表す。

計算の状態は項 e、スタック s、ヒープ Σ の三つ組 $\langle e;\ s;\ \Sigma\rangle$ からなる。評価される項が $F\,[e]$ のとき、PUSH 規則により、スタックの先頭に F を追加して e の評価に移る。スタックトップが $\square\,v$ かつ評価される項が $\lambda x.e$ のとき、β 規則により、まずスタックをポップし、ヒープに $x = v$ という情報が追加されて e が評価される。

評価される項が v かつスタックトップが F のとき、POP_v 規則によりスタックトップをポップして $F\,[v]$ を新たに評価する。$\langle v;\ (K\,\square) :: s;\ \Sigma\rangle$ のように、データコンストラクタの引数の評価が値になるまでおこなわれた三つ組を例に取り、この評価を一つ進めると、スタックトップの $K\,\square$ をポップして v をこのフレームに埋め込み、$\langle K\,v;\ s;\ \Sigma\rangle$ となる。

2.2.1 合流点の評価

合流点の操作もスタックの操作により表現される。**join** $j\ \overrightarrow{x} = e$ **in** u をスタック s、ヒープ Σ で評価することを考える。

$$\langle \textbf{join}\ j\ \overrightarrow{x} = e\ \textbf{in}\ u;\ s;\ \Sigma\rangle$$

まず **join** $j\ \overrightarrow{x} = e$ **in** $\square$ がスタックに積まれ、u を評価する。

$$\langle u;\ (\textbf{join}\ j\ \overrightarrow{x} = e\ \textbf{in}\ \square) :: s;\ \Sigma\rangle$$

u の評価を進めると、u の評価フレームがスタックに積まれていく。この u による評価フレームのスタックを s' とすると、評価スタックは s'@ (**join** $j\ \overrightarrow{x} = e$ **in** $\square :: s$) となる。u が **jump** 式を持つとき、評価が進むと計算の状態は以下のようになる。

$$\langle \textbf{jump}\ j\ \overrightarrow{v};\ s'@\,(\textbf{join}\ j\ \overrightarrow{x} = e\ \textbf{in}\ \square :: s);\ \Sigma'\rangle$$

ここで Σ' は Σ にいくつかの変数と値の組が追加、更新されたものである。**jump** $j\ \overrightarrow{v}$ を評価すると、JUMP 規則によりスタック s' が捨てられ、Σ' に $\overrightarrow{x = v}$ を追加して j に束縛されている e の評価がおこなわれる。

$$\langle e;\ \textbf{join}\ j\ \overrightarrow{x} = e\ \textbf{in}\ \square :: s;\ \Sigma', \overrightarrow{x = v}\rangle$$

スタック s' の破棄が合流点の操作の要となっており、s' に蓄えられている計算を破棄して合流点までジャンプすることができる。

再帰的な合流点についても同様の操作をおこなう。**join rec** $j\ \overrightarrow{x} = e$ **in** u をスタック s、ヒープ Σ で評価することを考える。u が **jump** 式を含むとき、非再帰的な合流点の操作と途中までは同じである。

$$\langle \textbf{jump}\ j\ \overrightarrow{v};\ s'@\,(\textbf{join rec}\ j\ \overrightarrow{x} = e\ \textbf{in}\ \square :: s);\ \Sigma'\rangle$$

同様に、JUMP 規則によりラベル j に束縛されている項 e を評価する。

$$\langle e;\ \textbf{join rec}\ j\ \overrightarrow{x} = e\ \textbf{in}\ \square :: s;\ \Sigma', \overrightarrow{x = v}\rangle$$

ここで **join rec** $j\ \overrightarrow{x} = e$ **in** $\square$ をスタックから破棄しないことにより、e に含まれる j への**再帰的なジャンプ**が可能である。

2.2.2 例外の評価

新たに追加した例外も合流点の操作と同様に、スタックの操作によりコントロールの制御を表現する。**try** E [**raise** $(Ex\ \overrightarrow{e})$] **with** $\overrightarrow{exc}$ をスタック s、Σ で評価することを考える。

$$\langle \textbf{try}\ E\,[\textbf{raise}\ (Ex\ \overrightarrow{e})]\ \textbf{with}\ \overrightarrow{exc};\ s;\ \Sigma\rangle$$

まず **try** □ **with** $\overrightarrow{exc}$ がスタックに積まれ、E [**raise** $(Ex\ \overrightarrow{e})$] を評価する。

$$\langle E\,[\textbf{raise}\ (Ex\ \overrightarrow{e})]\,;\ \textbf{try}\ \square\ \textbf{with}\ \overrightarrow{exc} :: s;\ \Sigma\rangle$$

評価を進めると、E から成るフレームがスタックに積まれる。E から成るフレームにより新たに積まれたスタックを s' とすると、

$$\langle \textbf{raise}\ (Ex\ \overrightarrow{e})\,;\ s' @ (\textbf{try}\ \square\ \textbf{with}\ \overrightarrow{exc} :: s)\,;\ \Sigma\rangle$$

となる。例外 Ex の引数 $\overrightarrow{e}$ が全て値になるまで評価されると

$$\langle \textbf{raise}\ (Ex\ \overrightarrow{v})\,;\ s' @ (\textbf{try}\ \square\ \textbf{with}\ \overrightarrow{exc} :: s)\,;\ \Sigma\rangle$$

となる。例外 Ex に対する例外ハンドラ $Ex\ \overrightarrow{x} \to u$ を持つようなフレーム **try** □ **with** $\overrightarrow{exc}$ をスタックトップから順に走査する。条件に合致するフレームおよび例外ハンドラ $Ex\ \overrightarrow{x} \to u$ が見つかった場合、スタックトップからそのフレームまでのフレームをすべて破棄し、例外ハンドラの仮引数 $\overrightarrow{x}$ を値 $\overrightarrow{v}$ に束縛し、u を評価する。

上記の例の $\langle$**raise** $(Ex\ \overrightarrow{v})\,;\ s' @ (\textbf{try}\ \square\ \textbf{with}\ \overrightarrow{exc} :: s)\,;\ \Sigma\rangle$ を評価すると、スタックの **try** □ **with** $\overrightarrow{exc}$ が持つ $\overrightarrow{exc}$ がハンドラ $Ex\ \overrightarrow{x} \to u$ を持つとき、スタックトップから **try** □ **with** $\overrightarrow{exc}$ までを破棄して u を評価する (Try-Catch 規則)。

$$\langle u;\ s;\ \Sigma, \overrightarrow{x = v}\rangle$$

再帰的な合流点とは異なり、スタックから **try** □ **with** $\overrightarrow{exc}$ は破棄される。このため例外ハンドラのボディ u で起きた例外はその **try-with** 式では捕捉されない。

また、スタック中に例外 Ex を捕捉するハンドラがなかった場合、プログラムが終了する (Uncaught-Raise 規則)。

このスタックの破棄が例外送出によるコントロールの制御を実現している。

2.3 合流点への変換

合流点となる部分を探し出し、**join/jump** 式へと変換する *contify* アルゴリズムを図 2.3に示す。

示した contify アルゴリズムは Maurer らが [6] の 4 章で示したアルゴリズムと同様の動作をする。*contify* は非再帰関数の **join/jump** 式への変換、$contify_{rec}$ が再帰関数の **join/jump** 式への変換を示している。ここで ρ は、contify 可能な変数 f とその引数と、対応するラベル j によるジャンプの対応を表す環境である。また dom (ρ) は ρ に登録されている変数の集合を返す。contify アルゴリズムを項および部分項に適用することで **join/jump** 式を含む Joel のプログラムに変換する。

$$
\begin{aligned}
\mathit{contify}\,(\mathbf{let}\ f = \lambda\overrightarrow{x}.u\ \mathbf{in}\ L\,[\overrightarrow{e}]) \quad &= \quad \mathbf{join}\ j\ \overrightarrow{x} = u\ \mathbf{in}\ L\left[\overrightarrow{\mathtt{tail}_\rho\,(e)}\right] \\
&\qquad \text{if}\ \rho\,(f\ \overrightarrow{x}) = \mathbf{jump}\ j\ \overrightarrow{x}\ \text{and}\ f \notin \text{fv}\,(L) \\
\mathit{contify}_{rec}\,(\mathbf{let\ rec}\ f = \lambda\overrightarrow{x}.L\,[u]\ \mathbf{in}\ L'\,[\overrightarrow{e}]) \quad &= \quad \mathbf{join\ rec}\ j\ \overrightarrow{x} = L\left[\overrightarrow{\mathtt{tail}_\rho\,(u)}\right]\ \mathbf{in}\ L'\left[\overrightarrow{\mathtt{tail}_\rho\,(e)}\right] \\
&\qquad \text{if}\ \rho\,(f\ \overrightarrow{x}) = \mathbf{jump}\ j\ \overrightarrow{x} \\
&\qquad \text{and}\ f \notin \text{fv}\,(L)\ \text{and}\ f \notin \text{fv}\,(L')
\end{aligned}
$$

$$
\textit{where} \qquad \mathtt{tail}_\rho\,(e) \quad = \quad \begin{cases}
\mathbf{jump}\ j\ \overrightarrow{u} & \text{if}\ e \equiv f\ \overrightarrow{u} \\
& \text{and}\ \rho\,(f\ \overrightarrow{x}) = \mathbf{jump}\ j\ \overrightarrow{x} \\
& \text{and}\ |\overrightarrow{u}| = |\overrightarrow{x}| \\
& \text{and dom}\,(\rho) \cap \text{fv}\,(\overrightarrow{u}) = \emptyset \\
e & \text{if dom}\,(\rho) \cap \text{fv}\,(e) = \emptyset \\
\text{undefined} & \textit{otherwise}
\end{cases}
$$

図 2.3　contify アルゴリズム

$\mathtt{tail}_\rho\,(e)$ は項を **jump** 式に変換する補助関数である。$\mathtt{tail}_\rho\,(e)$ は環境 ρ、変換の対象となる e を取り、以下の条件を満たす場合に変換がおこなわれる。

- $e \equiv f\ \overrightarrow{u}$ であり、f に対応する **jump** 式が ρ に登録されている
- アリティ $|\overrightarrow{x}|$ が適用している項の数 $|\overrightarrow{u}|$ と一致する
- $\text{dom}\,(\rho)$ と $\text{fv}\,(\overrightarrow{u})$ の積が空、つまり contify 可能な変数が $\overrightarrow{u}$ に自由変数として現れていない

このとき $e \equiv f\ \overrightarrow{u}$ を $\mathbf{jump}\ j\ \overrightarrow{u}$ に変換できる。

非再帰関数を対象とする *contify* は、変数 f で定義された関数が末尾文脈 L では用いられず、$\overrightarrow{e}$ の各要素を tail_ρ に渡すことで **join/jump** 式に変換する。f が末尾呼び出しであり、全ての f の呼び出しにおいて f に適用した項の数がアリティと一致する場合のみ contify 可能となる。以下のような項を例に取る。

$$
\begin{aligned}
&\mathbf{let}\ f = \lambda x.\lambda y.K\ x\ y\ \mathbf{in} \\
&\mathbf{let}\ g = \lambda x.\lambda y.y\ \mathbf{in} \\
&f\ m\ (g\ n)
\end{aligned}
$$

contify アルゴリズムにより以下のように変換できる。

$$
\begin{aligned}
&\mathbf{join}\ j_f\ x\ y = K\ x\ y\ \mathbf{in} \\
&\mathbf{let}\ g = \lambda x.\lambda y.y\ \mathbf{in} \\
&\mathbf{jump}\ j_f\ m\ (g\ n)
\end{aligned}
$$

f は末尾位置でのみ呼び出されており、適用している項の数 $(|m, (g\ n)| = 2)$ が f のアリティ $(|x, y| = 2)$ と一致するため contify ができる。一方 g は末尾位置以外で呼び出されており、なおかつ適用している項の数 $(|n| = 1)$ が g のアリティ $(|x, y| = 2)$ と一致しないため contify による変換がおこなわれない。

再帰関数を対象とする $\mathit{contify}_{rec}$ は、変数 f で定義された変数が L および L' では用いられず、u および $\overrightarrow{e}$ の各要素を tail_ρ に渡すことで **join/jump** 式に変換する。f が自身の定義および **let** 式の末尾において末尾位置

でのみ呼び出され、全ての f の呼び出しにおいて f に適用した項の数がアリティと一致する場合のみ contify 可能となる。以下のような項を例に取る。

$$
\begin{array}{l}
\textbf{let}\ fold = \lambda f.\lambda z.\lambda ls. \\
\quad \textbf{let rec}\ work = \lambda z.\lambda xs. \\
\quad\quad \textbf{match}\ xs\ \textbf{with} \\
\quad\quad\quad \mid Cons(x, xs') \quad \rightarrow \quad work\,(f\,z\,x)\,xs' \\
\quad\quad\quad \mid Nil \quad \rightarrow \quad z \\
\quad \textbf{in}\ work\,z\,ls \\
\textbf{in}\ g\ (fold\,plus\,zero\,ys)
\end{array}
$$

contify アルゴリズムにより以下のようになる。

$$
\begin{array}{l}
\textbf{let}\ fold = \lambda f.\lambda z.\lambda ls. \\
\quad \textbf{join rec}\ work\,z\,xs = \\
\quad\quad \textbf{match}\ xs\ \textbf{with} \\
\quad\quad\quad \mid Cons(x, xs') \quad \rightarrow \quad \textbf{jump}\ work\,(f\,z\,x)\,xs' \\
\quad\quad\quad \mid Nil \quad \rightarrow \quad z \\
\quad \textbf{in jump}\ work\,z\,ls \\
\textbf{in}\ g\ (fold\,plus\,zero\,ys)
\end{array}
$$

$work$ の呼び出しはすべて末尾位置でおこなわれており、 $work$ に適用している項の数 ($|(f\,z\,x)\,, xs'| = |z, ls| = 2$) が $work$ のアリティ ($|z, xs| = 2$) と一致するため contify ができる。 $fold$ は関数呼出しにおいて適用している項の数がアリティと一致しているが末尾呼び出しでないため変換はできない。

第 3 章

最適化

図 3.1に Joel 上でおこなう最適化を示す。

$$
\begin{array}{rcll}
(\lambda x.e)\ v & = & \textbf{let}\, x = v\, \textbf{in}\, e & (\beta) \\
(\lambda x.E_{free}\,[x])\ e & = & E_{free}\,[e] & \text{if } x \notin \text{fv}\,(E_{free}) \quad (\beta_\Omega) \\
\textbf{let}\, x = v\, \textbf{in}\, e & = & \textbf{let}\, x = v\, \textbf{in}\, e\,\{v/x\} & (\textit{inline}) \\
\textbf{let}\, x = e\, \textbf{in}\, u & = & u & \text{if } x \notin \text{fv}\,(u) \text{ and } e = v \quad (\textit{drop}_{\textit{value}}) \\
\textbf{join}\, j\ \overrightarrow{x} = e\, \textbf{in} & & \textbf{join}\, j\ \overrightarrow{x} = e\, \textbf{in} & \\
L\left[\overrightarrow{u}, \textbf{jump}\, j\ \overrightarrow{e'}, \overrightarrow{u'}\right] & = & L\left[\overrightarrow{u}, \textbf{let}\, \overrightarrow{x = e'}\, \textbf{in}\, e, \overrightarrow{u'}\right] & (\textit{jinline}) \\
\textbf{join}\, j\ \overrightarrow{x} = e\, \textbf{in}\, u & = & u & \text{if } j \notin \text{fl}\,(u) \quad (\textit{jdrop}) \\
\textbf{match}\, K\ \overrightarrow{v}\ \textbf{with}\, \overrightarrow{alt} & = & \textbf{let}\, \overrightarrow{x = v}\, \textbf{in}\, e & \text{if } (K\ \overrightarrow{x} \to e) \in \overrightarrow{alt} \quad (\textit{case}) \\
E_{free}\,[\textbf{let}\ [\textbf{rec}]\ x = e\, \textbf{in}\, u] & = & \textbf{let}\ [\textbf{rec}]\ x = e\, \textbf{in}\, E_{free}\,[u] & \begin{array}{l}\text{if bl}\,(E_{free}) \cap \text{fl}\,(e) = \emptyset \\ \text{and } x \notin \text{fv}\,(E_{free})\end{array} \\
 & & & (\textit{float}) \\
E_{free}\left[\textbf{match}\, e\, \textbf{with}\, \overrightarrow{K\ \overrightarrow{x}\ \to u}\right] & = & \textbf{match}\, e\, \textbf{with}\, \overrightarrow{K\ \overrightarrow{x} \to E_{free}\,[u]} & \begin{array}{l}\text{if } \overrightarrow{x} \cap \text{fv}\,(E_{free}) = \emptyset \\ \text{and bl}\,(E_{free}) \cap \text{fl}\,(e) = \emptyset\end{array} \\
 & & & (\textit{casefloat}) \\
E_{free}\left[\textbf{match}\, e\, \textbf{with}\, \overrightarrow{K\ \overrightarrow{x}\ \to u}\right] & = & \begin{array}{l}\textbf{join}\, j\, y = E_{free}\,[y]\ \textbf{in} \\ \textbf{match}\, e\, \textbf{with}\, \overrightarrow{K\ \overrightarrow{x} \to \textbf{jump}\, j\, u}\end{array} & \begin{array}{l}\text{if } j \notin \text{bl}\,(E_{free}) \\ \text{and } y \notin \text{fv}\,(E_{free}) \\ \text{and } (\text{fl}\,(e) \cup \text{fl}\,(\overrightarrow{u})) \cap \text{bl}\,(E_{free}) = \emptyset\end{array} \\
 & & & (\textit{contify}_{\textit{commute}}) \\
E\,[\textbf{join}\ [\textbf{rec}]\ j\ \overrightarrow{x} = u\, \textbf{in}\, e] & = & \textbf{join}\ [\textbf{rec}]\ j\ \overrightarrow{x} = E\,[u]\ \textbf{in}\, E\,[e] & \begin{array}{l}\text{if } j \notin \text{bl}\,(E) \\ \text{and } \overrightarrow{x} \cap \text{fv}\,(E) = \emptyset\end{array} \\
 & & & (\textit{jfloat}) \\
E_{free}\,[\textbf{jump}\, j\ \overrightarrow{e}] & = & \textbf{jump}\, j\ \overrightarrow{e} & \begin{array}{l}\text{if } j \notin \text{bl}\,(E_{free}) \\ \text{and bl}\,(E_{free}) \cap \text{fl}\,(\overrightarrow{e}) = \emptyset\end{array} \\
 & & & (\textit{abort})
\end{array}
$$

図 3.1　Joel における最適化

$\text{fl}\,(e)$ は e に含まれる束縛されていないラベルの集合を表し、 $\text{bl}\,(e)$ は e が束縛するラベルの集合を表す。

Maurer らにより定義された System F_J 上の最適化に基づいて、値呼びの言語に沿って設計をおこなった。また、 CPS 上でおこなわれる最適化と同等の最適化がおこなえるように、 β_Ω を追加した [8]。

Maurer らによる System F$_J$ 上での最適化では、必要呼びの言語ではいつでも可能だが値呼びの言語では必ずしも成り立たない変換があるため、それらを値呼びでも成り立つように変更した。例えば *drop*$_{value}$ は、Maurer らによる最適化の一つ *drop* を変更したものであり、 **let** 式で束縛される項が値のときにのみ変換ができるように制約が加えられている。 この制約は Joel が値呼びの言語であることに由来する。束縛される項は

$$\textbf{let}\ [\textbf{rec}]\ x = e\ \textbf{in}\ u \quad = \quad u \quad \text{if}\ x \notin \text{fv}\,(e)$$

図 3.2　Maurer らによる *drop* の定義

必要呼びでは束縛時に評価がおこなわれず、値呼びの場合は束縛時に評価がおこなわれる。このとき束縛される項が例外を発生するか評価が停止しない可能性があり、そのとき *drop* はプログラムの意味を変えることになる。

casefloat を例にとると、 Maurer らにより定義された *casefloat* とは評価文脈が異なる。 対象となる項を

$$E\left[\textbf{match}\ e\ \textbf{with}\ \overrightarrow{alt}\right] \quad = \quad \textbf{match}\ e\ \textbf{with}\ \overrightarrow{K\ \overrightarrow{x}\ \rightarrow E\,[u]}$$

図 3.3　Maurer らによる *casefloat* の定義

$E\left[\textbf{match}\ e\ \textbf{with}\ \overrightarrow{alt}\right]$ とすると、 E が $\textbf{try}\ \square\ \textbf{with}\ \overrightarrow{exc}$ を持つとき、 Maurer らによる定義では e が起こしうる例外を捕捉できない形に変換されるため、プログラムの意味を変える変換となってしまう。そのため Joel 上でおこなう *casefloat* は E_{free} を評価文脈に持つ項のみに対象を限定している。

また、 System F$_J$ にはなかった最適化規則 *contify*$_{commute}$ を本研究では追加した。コードサイズを肥大化させ得る *casefloat* に対して、 *contify*$_{commute}$ はコード肥大化を抑える規則となっている。次のようなプログラム図 3.4を考える。 *casefloat* を適用すると図 3.5外側の評価文脈が内側のマッチケースに展開される。そして

$$\begin{array}{l}\textbf{match}\\ \quad \textbf{begin match}\ v\ \textbf{with}\\ \qquad \mid P_1 \rightarrow e\\ \qquad \mid P_2 \rightarrow R\\ \quad \textbf{end}\\ \textbf{with}\\ \mid Q\,x \rightarrow x\\ \mid R\ \ \rightarrow m\end{array}$$

図 3.4　最適化前のプログラム

図 3.5の内側の **match** 式に *case* を適用することを考える。 **match** 式に渡される項 e が値であるとき、 *case* を適用することができる。 $e \equiv Q\,v'$ のとき、図 3.5に *case* を適用すると、図 3.6のようなプログラムが得られる。 項 e が値でないとき、 *case* 規則を適用できない。そのため P_1 のマッチケース内の **match** 式は解消されず、コードサイズが肥大したままになる。このように、 *casefloat* をおこなったが *case* 規則を適用できな

```
match v with
  | P1 →
    begin match e with
      | Q x  → x
      | R    → m
    end
  | P2 →
    begin match R with
      | Q x  → x
      | R    → m
    end
```

図 3.5　*casefloat* を適用したプログラム

```
match v with
| P1 → let x = v' in x
| P2 → m
```

図 3.6　*case* を適用したプログラム

かった場合、コードサイズが外側の **match** 式のマッチケースの数に比例する。図 3.4に $contift_{commute}$ を適用すると図 3.7のようになる。 *casefloat* がコードサイズを肥大化させる一方、 $contify_{commute}$ は外側の **match** 式

```
join j y =
  match y with
    | Q x  → x
    | R    → m
in
match v with
| P1 → jump j e
| P2 → jump j R
```

図 3.7　$contify_{commute}$ を適用したプログラム

を **join** 式で括り、内側の **match** 式のマッチケースを **jump** 式に渡すことでコードの肥大化を抑えることができる。

3.1　J-正規形 (JNF、J-Normal Form)

abort により、 **jump** $j\ \vec{e}$ に付属する評価文脈が取り払われる。このため、図 3.1で示した最適化、特に *abort* を適用できる限り適用すると **jump** $j\ \vec{e}$ が全て末尾位置になることが予想される。全ての **jump** $j\ \vec{e}$ が末尾位置にあるような Joel 言語の式を ***J*-正規形** (JNF、 J-Normal Form) と呼び、その構文を 3.1 節に示す。

$$
\begin{array}{lcl}
\textbf{Terms} & & \\
e, u & ::= & x \mid \lambda x.e \mid eu \\
& \mid & K\ \overrightarrow{e} \\
& \mid & \textbf{match}\ e\ \textbf{with}\ \overrightarrow{K\ \overrightarrow{x} \to u} \\
& \mid & \textbf{let}\ x = e\ \textbf{in}\ u \\
& \mid & \textbf{let rec}\ x = e\ \textbf{in}\ u \\
& \mid & \textbf{join}\ j\ \overrightarrow{x} = e\ \textbf{in}\ e_L \\
& \mid & \textbf{join rec}\ j\ \overrightarrow{x} = e_L\ \textbf{in}\ u_L \\
& \mid & \textbf{try}\ e\ \textbf{with}\ \overrightarrow{Ex\ \overrightarrow{x} \to u} \\
& \mid & \textbf{raise}\ (Ex\ \overrightarrow{e}) \\
e_L, u_L & ::= & e, u \\
& \mid & L\,[\textbf{jump}\ j\ \overrightarrow{e}]
\end{array}
$$

図 3.8　JNF の抽象構文

3.2 節で述べる、 Joel から無条件ジャンプのようなコントロールの操作がない言語への変換において、最適化を可能な限り適用した Joel の項が JNF で表せることは必須条件となる。

3.2　合流点からの変換

本研究での実装において、ターゲット言語を OCaml に設定した。 Joel から OCaml への変換にあたり、合流点を OCaml の項で表現するために *decontify* という操作をおこなう (図 3.9)。 decontify 操作は項およびその

$$
\textit{decontify}\,(e) \quad = \quad \begin{cases}
\textbf{let}\ j = \lambda\overrightarrow{x}.u\ \textbf{in}\ u' & \text{if } e = (\textbf{join}\ j\ \overrightarrow{x} = u\ \textbf{in}\ u') \\
\textbf{let rec}\ j = \lambda\overrightarrow{x}.u\ \textbf{in}\ u' & \text{if } e = (\textbf{join rec}\ j\ \overrightarrow{x} = u\ \textbf{in}\ u') \\
j\ \overrightarrow{u} & \text{if } e = (\textbf{jump}\ j\ \overrightarrow{u}) \\
e & \textit{otherwise}
\end{cases}
$$

図 3.9　decontify アルゴリズム

部分項へ再帰的に適用される。

次のような Joel の項を考える。 図 3.10は decontify により次のように変形できる。 $\textbf{join rec}\ work\ z\ xs = \ldots$

$$
\begin{array}{l}
\textbf{let}\ fold = \lambda f.\lambda z.\lambda xs. \\
\quad \textbf{join rec}\ work\ z\ xs = \\
\quad\quad \textbf{match}\ xs\ \textbf{with} \\
\quad\quad\quad \mid Cons(x, xs') \to \textbf{jupm}\ work\ (f\ z\ x)\ xs' \\
\quad\quad\quad \mid Nil \to z \\
\quad \textbf{in jump}\ work\ z\ xs \\
\textbf{in}\ fold\ (\lambda x.\lambda y.x + y)\ 0\ ls
\end{array}
$$

図 3.10　decontify 前のプログラム

は **let** 式 $\textbf{let rec}\ work = \lambda z.\lambda xs.\ldots$ に変形される。そして $\textbf{jump}\ work\ \ldots$ は関数呼出し $work\ \ldots$ に変形される。

$$
\begin{array}{l}
\textbf{let}\ fold = \lambda f.\lambda z.\lambda xs. \\
\quad \textbf{let rec}\ work = \lambda x.\lambda xs. \\
\qquad \textbf{match}\ xs\ \textbf{with} \\
\qquad\quad \mid Cons(x, xs') \rightarrow work\,(f\,z\,x)\,xs' \\
\qquad\quad \mid Nil \rightarrow z \\
\quad \textbf{in}\ work\,z\,xs \\
\textbf{in}\ fold\ (\lambda x.\lambda y.x + y)\ 0\,ls
\end{array}
$$

decontify は JNF である Joel のみを対象としている。次のような Joel の項を考える。 ラベル j に束縛され

$$\textbf{join}\,j\,x = x\,\textbf{in}\,3 + \textbf{jump}\,j\,10$$

図 3.11　Joel の項

ている合流点は任意の型の値 x を取り x を返す恒等関数となっており、その型は $\forall\tau.\tau \rightarrow \tau$ と考えられ、多相性を持っている。そしてこの項における **jump** 式は末尾位置になく、評価文脈 $3 + \square$ を持つため、 JNF ではなくかつ *abort* が適用できる。 OCaml において **jump** 式のようなコントロールの制御を実現するためには、例外処理以外にも大域脱出などで用いられる **try-with** 式を使うことが考えられるかもしれない。

```
exception Jump of 'a;;
try 3 + (raise (Jump 10)) with
  Jump x -> x;;
```

しかし、 OCaml の例外は多相を許していない。例外の引数の型が多相の場合、例外のハンドラの仮引数も多相である必要がある。例外の引数が多相の場合を仮定する (プログラム 3.1)。

プログラム 3.1　多相型を持つ例外の例

```
exception Poly of 'a;;
let f = function
  | 0 -> raise @@ Poly false
  | n -> n + 1
let g n =
  try f n with Poly x -> x + 1
```

型`'a -> exn`を持つ例外 `Poly`、内部で例外 `Poly false`を送出する関数 `f`、 `f`を `try-with`式の中で呼び出し `Poly`をハンドルする関数 `g`を定義する。 OCaml の型システムは `g`の `try-with`式が持つ例外ハンドラ `Poly x -> x + 1`の型付けを次のようにおこなう。最初に `x`の型を任意の単相型`'_a`とする。型 `int -> int -> int`を持つ関数 `(+)`に `x`を渡しているため、 `x`を `int`型に定める。例外の送出である `raise`関数は任意の型`'b`を返し例外の引数の型を伝搬しないため、例外ハンドラは例外の引数の任意の型`'a`を知ることができない。そのため `g`には正しい型 `int -> int`が付くが、 `g 0`を実行すると型の不一致による実行時エラーが起こる。このことから、`exception Jump of 'a -> 'a`のような定義はできない。

図 3.11に *abort* を適用すると以下のようになる。

$$\textbf{join}\, j\, x = x\, \textbf{in}\, \textbf{jump}\, j\, 10$$

評価文脈 $3 + \square$ が取り払われ、 **jump** 式が末尾位置にあるため JNF である。この場合 decontify 可能であり、次のような OCaml プログラムに変換することができる。

```
let j = fun x -> x in j 10
```

jump 式が末尾位置にあると OCaml におけるコントロールの制御を考慮する必要がなく、また多相性を保ちながら、関数とその呼出しに落とし込むことができる。したがって、合流点の多相性を保ちながら OCaml への変換をおこなうには OCaml の関数だけで表現できるように **jump** 式が末尾位置である必要がある。

第 4 章

性能測定

Joel による最適化を評価するために、ターゲット言語へ変換し、そのターゲット言語の実行速度、コードサイズを測定する。今回はターゲット言語を OCaml とした。また、比較対象として、中間言語として広く知られる CPS を用いる。

4.1 Core ML

今回の実験においてソース言語となる Core ML は、 Joel から合流点を除いた言語となる (図 4.1)。 Core

Terms

$$
\begin{array}{lcl}
e, u & ::= & x \mid \lambda x.e \mid e\,u \mid K\ \overrightarrow{e} \\
& \mid & \textbf{match}\ e\ \textbf{with}\ \overrightarrow{alt} \\
& \mid & \textbf{let}\ vb\ \textbf{in}\ e \\
& \mid & \textbf{try}\ e\ \textbf{with}\ \overrightarrow{exc} \\
& \mid & \textbf{raise}\ (Ex\ \overrightarrow{e})
\end{array}
$$

$$
\begin{array}{lcl}
vb & ::= & [\textbf{rec}]\ x = e \\
alt & ::= & K\ \overrightarrow{x} \to e \\
exc & ::= & Ex\ \overrightarrow{e}
\end{array}
$$

Values

$$
\begin{array}{lcl}
v & ::= & x \mid \lambda x.e \mid K\ \overrightarrow{v}
\end{array}
$$

図 4.1 Core ML の抽象構文

ML は今回のターゲット言語である OCaml のサブセットでもある。そのため Core ML の項は、 Joel や後述の CPS のような OCaml への変換はおこなわずに、 OCaml プログラムとして実行することができる。

4.2 CPS

比較対象として、関数型言語のコンパイラにおける代表的な中間言語である CPS を選択した。 CPS(Continuation Passing Style、継続渡し形式) はプログラムのコントロールを全て継続で表現する形式である。 [1] により詳細な性質や長所が述べられている。

本研究では、 Core ML のように例外を持つ言語を表現するために、 Thielecke による Double-Barreled CPS [11] に着想を得た、複数の例外ハンドラを持つような CPS を設計した。

4.2.1 抽象構文

本研究で用いる CPS 言語の抽象構文を図 4.2に示す。先述の Core ML から CPS 言語への変換 (図 4.4) によ

Terms

$$
\begin{array}{lll}
e, u & ::= & x \mid \lambda x.e \mid e\,u \mid K\,\overrightarrow{e} \\
& \mid & \textbf{let}\, x = e\, \textbf{in}\, u \\
& \mid & \textbf{let rec}\, x = e\, \textbf{in}\, u \\
& \mid & \textbf{match}\, e\, \textbf{with}\, \overrightarrow{K\,\overrightarrow{x} \to u} \\
& \mid & \left[\overrightarrow{(Ex, e)}\right] \mid e :: u \mid e @ u \\
& \mid & \textbf{lookup}\,(Ex, e) \\
\textbf{Values} & & \\
v & ::= & x \mid \lambda x.e \mid K\,\overrightarrow{v}
\end{array}
$$

図 4.2　CPS の抽象構文

り、 Core ML における例外よおび例外のハンドリングは ***lookup*** 関数および例外ハンドラのリストに変換される。 $\left[\overrightarrow{(Ex, e)}\right]$ は例外名と例外ハンドラのペアのリスト、 :: はリストへの要素の追加、 @ はリストの結合を表している。図 4.3に **lookup** 関数の定義を示す。リスト h の左側から例外名 Ex に対応する例外ハンドラを探し

$$
\textbf{lookup}\,(Ex, e) = \begin{cases} u & \text{if } e = (Ex, u) :: e' \\ \textbf{lookup}\,(Ex, e') & \text{if } e = (Ex', u) :: e' \\ \text{undefined} & \textit{otherwise} \end{cases}
$$

図 4.3　**lookup** 関数の定義

ていく関数である。

4.2.2 Core ML からの変換規則

Core ML から CPS への変換規則を図 4.4に示す。全ての変換は継続 k に加え、例外ハンドラのリスト h を受け取るようになっている。 **try-with** 式の変換で例外ハンドラ $\overrightarrow{Ex\,\overrightarrow{x} \to u}$ が $\left[\overrightarrow{(Ex, \lambda\overrightarrow{x}.[\![u]\!]\,k\,h)}\right]$ に変換され、例外ハンドラ h の先頭に追加される。例外の送出 **raise** $(Ex\,\overrightarrow{e})$ は $\lambda k.\lambda h.[\![\overrightarrow{e}]\!]\left(\lambda\overrightarrow{e'}.(\textbf{lookup}\,(Ex, h))\;\overrightarrow{e'}\right)\,h$ に変換され、 **lookup** 関数により例外ハンドラのリストから例外 Ex に対応するハンドラを探し、そのハンドラに $\overrightarrow{e}$ を CPS 変換したものを渡す継続が $[\![\overrightarrow{e}]\!]$ に渡される。

Core ML の項 (図 4.5) を変換することを考える。 図 4.4により、図 4.5を図 4.6のように変換することができる。 図 4.6をプログラムとして実行するには、この項に継続として恒等関数 $\lambda i.i$、空の例外ハンドラのリスト [] を渡せばよい。

$$
\begin{aligned}
\llbracket \lambda x.e \rrbracket &= \lambda k.\lambda h.k\,\lambda x.\lambda k'.\lambda h'.\llbracket e \rrbracket\,k'\,h' \\
\llbracket x \rrbracket &= \lambda k.\lambda h.k\,x \\
\llbracket e\,u \rrbracket &= \lambda k.\lambda h.\llbracket u \rrbracket\,(\lambda u'.\llbracket e \rrbracket\,(\lambda e'.e'\,u'\,k\,h)\;h)\;h \\
\llbracket \textbf{let}\;[\textbf{rec}]\;x = e\;\textbf{in}\;u \rrbracket &= \begin{cases} \lambda k.\lambda h.\textbf{let}\;[\textbf{rec}]\;x = v^\star\;\textbf{in}\;\llbracket u \rrbracket\,k\,h & \text{if}\; e = v \\ \lambda k.\lambda h.\llbracket e \rrbracket\;(\lambda x.\llbracket u \rrbracket\,k\,h)\;h & \textit{otherwise} \end{cases} \\
\llbracket K\;\overrightarrow{u} \rrbracket &= \lambda k.\lambda h.\llbracket \overrightarrow{u} \rrbracket\left(\lambda\overrightarrow{u'}.k\;\left(K\;\overrightarrow{u'}\right)\right)\;h \\
\llbracket \textbf{match}\,e\,\textbf{with}\;\overrightarrow{K\;\overrightarrow{x}\;\rightarrow u} \rrbracket &= \lambda k.\lambda h.\llbracket e \rrbracket\left(\lambda e'.\textbf{match}\;e'\;\textbf{with}\;\overrightarrow{K\;\overrightarrow{x}\;\rightarrow \llbracket u \rrbracket\,k\,h}\right)\;h \\
\llbracket \textbf{try}\,e\,\textbf{with}\;\overrightarrow{Ex\;\overrightarrow{x}\;\rightarrow u} \rrbracket &= \lambda k.\lambda h.\llbracket e \rrbracket\,k\;\left(\left[\overrightarrow{(Ex, \lambda\overrightarrow{x}.\llbracket u \rrbracket\,k\,h)}\right]\;@h\right) \\
\llbracket \textbf{raise}\;(Ex\;\overrightarrow{e}) \rrbracket &= \lambda k.\lambda h.\llbracket \overrightarrow{e} \rrbracket\left(\lambda\overrightarrow{e'}.(\textbf{lookup}\;(Ex, h))\;\overrightarrow{e'}\right)\;h \\
\llbracket \overrightarrow{e} \rrbracket\left(\lambda\overrightarrow{e'}.m\right)\;h &\equiv \llbracket e_n \rrbracket\;\left(\lambda e'_n.\llbracket e_{n-1} \rrbracket\;\left(\lambda e'_{n-1}.\cdots\cdots\llbracket e_1 \rrbracket\;(\lambda e'_1.m)\;h\cdots\cdots h\right)\;h\right)\;h \\
(\lambda x.e)^\star &\equiv \lambda k.\lambda h.\llbracket e \rrbracket\,k\,h \\
(K\;\overrightarrow{v})^\star &\equiv K\;\overrightarrow{v}
\end{aligned}
$$

図 4.4　CPS 変換

```
try
  raise (Ex, e)
with Ex x → f x
```

図 4.5　変換対象となる項

```
λk.λh.
  (λk'.λh'. (λk2.λh2.k2 e) (λe'. (lookup (Ex, h')) e') h)
    k ((Ex, λx. (λk3.λh3. (λk4.λh4.k4 x) (λx'. (λk5.λh5.k5 f) (λf'.f' x' k3 h3) h3) h3) k h) :: h)
```

図 4.6　変換後の項

4.2.3　CPS 上の最適化

CPS における最適化の規則を図 4.7に示す。

$$
\begin{array}{rcll}
(\lambda x.e)\ u & = & e\,\{u/x\} & & (\beta) \\
\textbf{let}\, x = e\, \textbf{in}\, u & = & \textbf{let}\, x = e\, \textbf{in}\, u\,\{e/x\} & & (\mathit{inline}) \\
\textbf{let}\, x = e\, \textbf{in}\, u & = & u & \text{if } x \notin \mathrm{fv}\,(u) & (\mathit{drop}_{\mathit{value}}) \\
\textbf{match}\, K\ \overrightarrow{e}\ \textbf{with}\, \overrightarrow{alt} & = & \textbf{let}\, \overrightarrow{x = e}\, \textbf{in}\, u & \text{if } (K\ \overrightarrow{x} \to u) \in \overrightarrow{alt} & (\mathit{case}) \\
\left[\overrightarrow{(Ex, e)}\right] \mathbin{@} \left[\overrightarrow{(Ex', u)}\right] & = & \left[\overrightarrow{(Ex, e)}; \overrightarrow{(Ex', u)}\right] & & (\mathit{append}) \\
\textbf{lookup}\,(Ex, (Ex', \_) :: u) & = & \textbf{lookup}\,(Ex, u) & & (\mathit{lookup}_0) \\
\textbf{lookup}\,(Ex, (Ex, e) :: u) & = & e & & (\mathit{lookup})
\end{array}
$$

図 4.7　CPS 上の最適化

4.2.4　OCaml への変換

CPS 言語における例外ハンドラの処理を OCaml で実現するため、 OCaml に以下のようなモジュールを追加する。

プログラム 4.1　C モジュール

```
module C = struct
  type ('k, _, _) assoc_hlist =
    | Nil  : ('k, 'z, 'z) assoc_hlist
    | Cons : ('k * 't) * ('k, 'u, 'x) assoc_hlist ->
             ('k, 't -> 'u, 'x) assoc_hlist
  let rec (@) : type k ty1 ty2 v.
      (k, ty1, ty2) assoc_hlist ->
      (k, ty2, v) assoc_hlist ->
      (k, ty1, v) assoc_hlist
    = fun a b ->
    match a with
    | Nil -> b
    | Cons(x, xs) -> Cons(x, xs @ b)
  let rec lookup : type k a b.
      k -> (k, a, b) assoc_hlist -> b
    = fun key -> function
    | Cons((key', x), ls) ->
      if key = key' then
        Obj.magic x
      else lookup key ls
    | Nil -> failwith "not found"
end;;
```

OCaml で一般に使われるリスト`'a list`は特定の型`'a`の要素のみを持つリストに対応する。 CPS から OCaml への変換にあたり、引数の異なる例外ハンドラを一つのリストで表現するために、 GADTs(Generalized

Algebraic Data Types、一般代数的データ型) という機能を用い、キーとなる`'k`型 (ここでは例外名として`string`型) と任意の型のペア型を要素に持つ連想リスト`('k, _, _) assoc_hlist`型を定義した。`assoc_hlist`のデータコンストラクタ`C.Cons`、`C.Nil`は CPS における ::、[] にそれぞれ対応する。`C.(@)`は CPS における @ に対応する。CPS における ::、[] および @ を`C.Cons`、`C.Nil`、`C.(@)`に、**lookup** を`C.lookup`に変換することで OCaml プログラムとして実行することができるようになる。この操作を *transpile* と呼び、図 4.8に示す。*transpile* を全ての項および部分項におこなうことで、CPS の項から OCaml プログラムへ変換することが

$$
\begin{aligned}
\textit{transpile}\,(e :: u) &= C.Cons(e, u) \\
\textit{transpile}\,(e@u) &= C.(@)\,e\,u \\
\textit{transpile}\,(\textbf{lookup}\,(Ex, e)) &= C.lookup\,Ex\,e \\
\textit{transpile}\,(e) &= e
\end{aligned}
$$

図 4.8　*transpile* アルゴリズム

できる。

4.3　コードサイズの評価

本研究での性能比較として、生成された OCaml プログラムの実行速度と、そのプログラムのサイズの評価をおこなう。今回はプログラムのサイズの評価方法として、生成された OCaml プログラムをコンパイルして得られるオブジェクトファイルのファイルサイズを測定する。

次のような例を考える。OCaml プログラムが記述されているファイル `bench_foo.ml` がある。このファイルをコンパイルすると、`bench_foo.o` というファイルが得られる。この `bench_foo.o` のファイルサイズを測定する。

4.4　実験

4.4.1　実験環境

本研究では Joel を OCaml により実装した。また、ベンチマークライブラリには Core_bench[*1] を用いた。

以下の環境で実験をおこなった (表 4.1)。

4.4.2　実験内容

Core ML で記述された、自由変数`__arg__`を持つ各実験プログラムを Joel ないしは CPS に変換し、各言語上での最適化をおこなう。そして最適化によって得られたプログラムを OCaml に変換し、コードサイズを測定する。最後に各プログラムの`__arg__`に任意の長さの`int list`を渡してその実行速度を測る。

[*1] `https://github.com/janestreet/core_bench`

CPU	Intel Xeon Processor E3-1270
メモリ	DDR3 8GB
オペレーティングシステム	Arch Linux
OCaml 処理系	OCaml version 4.05.0

表 4.1　実験環境

4.4.3　実験結果と考察

実験結果は以下のようになった。図 4.9、 4.10、 4.11、 4.12、 4.13 は実行速度、表 4.2はコードサイズを表している。また、 Core ML で記述された実験プログラムおよび生成されたプログラム、 4.2.4 項で述べた C モジュールなどの設定は Aに添付した。

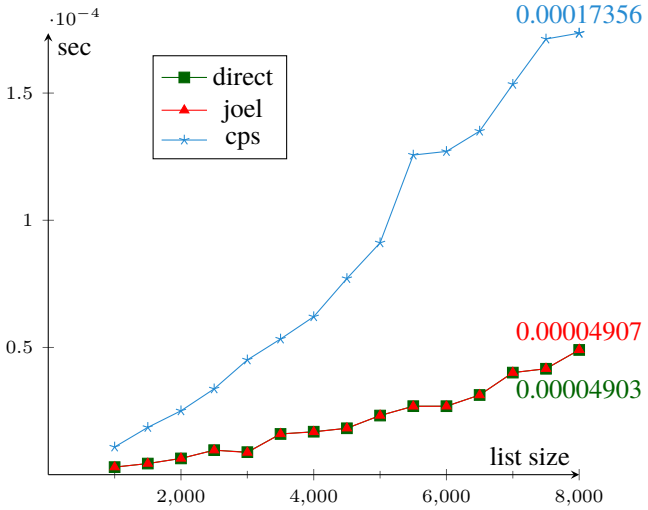

図 4.9　rev の実行速度

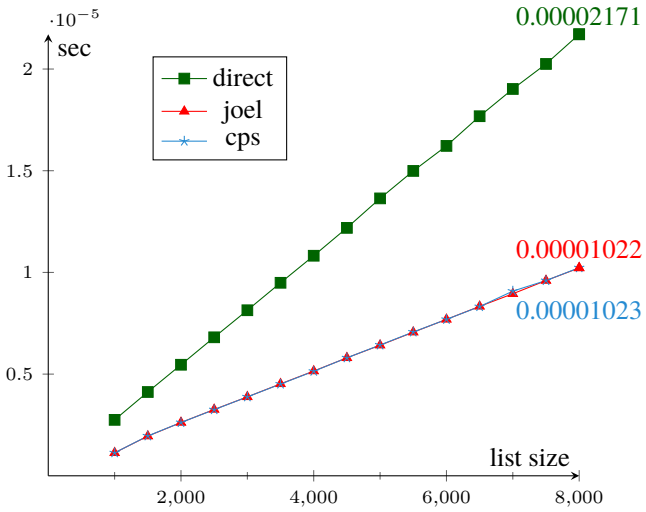

図 4.10　exists の実行速度

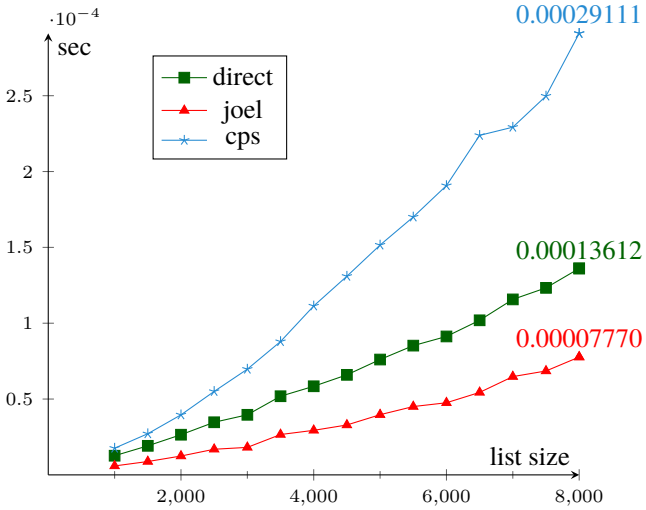

図 4.11　mapfold の実行速度

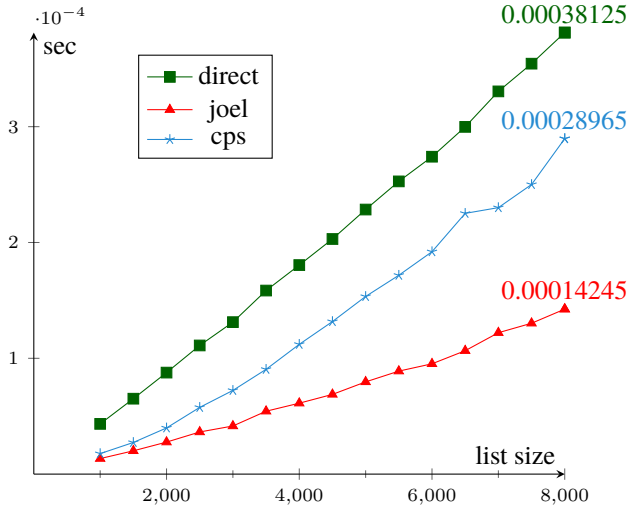

図 4.12　try-mapfold の実行速度

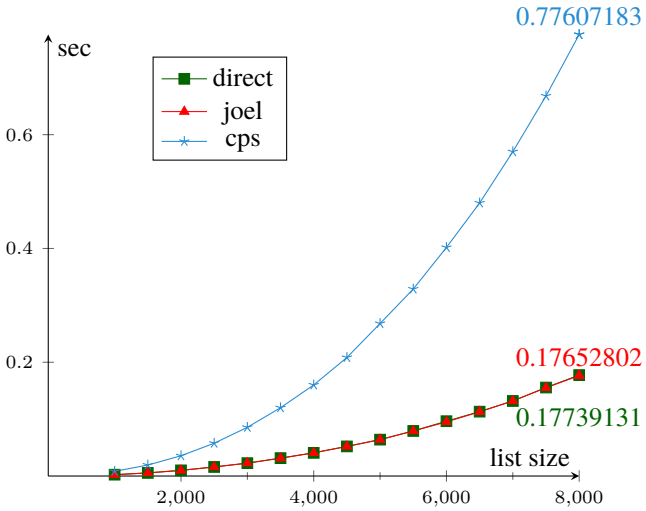

図 4.13　stream の実行速度

	Direct	Joel	CPS
rev	3040	2768	4144
exists	4136	2576	3120
mapfold	5000	3384	5168
try-mapfold	5880	3816	5264
stream	7312	4256	7680

表 4.2　コードサイズ

実行速度に関して、今回の実験では Joel 上での最適化による実行速度の低下は見られなかった。 mapfold、try-mapfold では実行速度の向上が見られた。

いくつかのケースでは CPS 上で最適化をおこない生成されたプログラムが Core ML を直接実行したケースと比較して遅くなった。プログラム中の多くの関数呼出しにより、生成されたプログラムに継続や例外ハンドラのリストを取る多くの関数クロージャがある。そのために Core ML や Joel にはなかった余分な関数呼出しおよびメモリの確保によるオーバーヘッドがかかっている予想できる。

コードサイズに関しては、いずれのテストプログラムにおいても Joel は縮小化に成功している。 Joel 上での最適化により、 [6] に示されている再帰的な合流点を利用した融合変換が exists ではおこなわれている。融合変換により中間データ構造を作る関数とそれを利用する関数が 1 つの関数にまとめられ、コンパクトなプログラムが生成されていることが確認できた。

第 5 章

結論

本研究では、 Maurer らの提案した System F_J に基づき、明示的な合流点をもつ値呼びの非純粋な言語 Joel を定義し、コンパイラ中間言語としての有用性を検証した。 CPS と ANF が抱える合流点に関する問題を、 Maurer らは明示的な合流点を持つ中間言語 System F_J を設計することにより解消した。再帰的な合流点により、 CPS や ANF では表現し難い最適化を System F_J でおこなうことができ、高速に動作するターゲット言語へのコンパイルが可能となった。しかし System F_J は必要呼びの評価戦略を持つ副作用のない言語をターゲット言語としたコンパイラ中間言語として提案された。今日において広く用いられているプログラム言語は値呼びで非純粋な言語が多く、 Maurer らによるアイデアを広く適用するには隔たりがある。本研究では、値呼びの評価戦略で副作用のある言語をターゲット言語とした、 System F_J のキーコンセプトである明示的な合流点を持つ中間言語 Joel を定義した。 Joel の持つ非純粋性として、多くのプログラム言語に備わっている機能である例外の送出、例外ハンドリングを追加した。

Joel とその上での最適化を定義することで、コントロールエフェクトのある非純粋な値呼びの言語において、明示的な合流点を用いた中間言語における最適化が有効に働くかどうかを検証した。検証方法として、Joel をコンパイラ中間言語とした場合に生成されるターゲット言語の実行速度およびコードサイズの評価をおこなった。 OCaml のサブセットである Core ML を新たに定義し、ターゲット言語として用いた。 CPS 言語を中間言語に採用した場合の最適化との比較実験では、実行速度の向上は見られなかったがサイズの小さなプログラムの生成に成功した。コードサイズの縮小化は、組み込みシステムのようにストレージが小容量の場合に対して恩恵がある。また、プログラムを CPU キャッシュメモリへ格納できる可能性が高くなり、実行速度の向上が期待される。

join/jump 式は、全ての出現位置で末尾呼び出しになっている関数およびその呼び出しの明示化であり、*jfloat* により評価文脈を取り入れる操作は末尾呼び出し最適化と考えることができる。また、 $contify_{commute}$ により評価文脈を合流点として表現することで、プログラムの分岐位置においてコードサイズが肥大化することを抑えられる。

5.1 関連研究

K 正規形 [2] は合流点と類似する部分に関する操作により、コードサイズの肥大化を抑えている。しかし、Maurer らが提案した合流点の操作は、コードサイズの縮小だけでなく、より進んだ最適化をおこなうことを目

的としている点で異なる。上野、大堀も明示的な合流点を持つ計算体系 [12] を提案している。ただし、彼らの貢献は CPS や ANF におけるコードサイズ爆発を抑えるための let 挿入を形式化したものであり、最適化に関しては触れられていない。

5.2 課題

ソース言語から Joel への変換、 Joel 上での最適化、および Joel からターゲット言語への変換がプログラムの意味を保存することは示すべき性質である。また、 Joel の最適化が進むと JNF になることも、本論文は予想の範疇を出ていない。最適化が進んだ Joel の項が JNF に含まれるということは、 Joel から、今回は OCaml とした、ターゲット言語への変換の正当性を示す道具となる。つまり、本研究で定義した Joel に求められる性質は次に挙げる定理が成り立つことである。

- contify 操作がプログラムの意味を保存する
  ソース言語 L の項 e_l が $e_l \to e'_l$ となるならば、 contify により得られた Joel の項 $e \equiv \mathit{contify}\,(e_l)$ について、 $e \to e'$ かつ $e' \equiv \mathit{contify}\,(e'_l)$ である。
- 最適化がプログラムの意味を保存する
  変換の等式 $e = e'$ について、 $\langle e;\ s;\ \Sigma\rangle \mapsto^\star \langle v;\ s^\star;\ \Sigma^\star\rangle$ ならば $\langle e';\ s';\ \Sigma'\rangle \mapsto^\star \langle v;\ s'^\star;\ \Sigma'^\star\rangle$ である。
- Joel の項の最適化がこれ以上できないならば JNF である
  変換の等式 $e = e'$ について $e \equiv e'$ ならば e' は JNF に含まれる。
- decontify 操作がプログラムの意味を保存する
  Joel の項 e が $e \to e'$ ならば、 decontify により得られたターゲット言語 L のプログラム $e_l \equiv \mathit{decontify}\,(e)$ について、 $e_l \to e'_l$ かつ $e'_l \equiv \mathit{decontify}\,(e')$ である。
- 最適化が CPS と等価
  第 3 章で示した最適化を J とすると

$$J \vdash joel\,(M) = joel\,(N) \qquad iff \qquad \lambda\beta\eta \vdash cps\,(M) = cps\,(N)$$

謝辞

本研究の進行および本論文の執筆にあたり様々な助言および指導をしてくださった亀山幸義先生並びに海野広志先生、プログラム論理研究室の皆様に深く感謝致します。論文執筆における組版技術を始めとした種々の議論に付き合っていただいた WORD 編集部の在学生および OB の皆様に感謝致します。 5 年間の生活を支えてくださった家族に感謝致します。

参考文献

[1] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA, 2007.

[2] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From Region Inference to Von Neumann Machines via Region Representation Inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 171–183, New York, NY, USA, 1996. ACM.

[3] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The Essence of Compiling with Continuations. *SIGPLAN Not.*, 39(4):502–514, April 2004.

[4] Andrew Kennedy. Compiling with Continuations, Continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 177–190, New York, NY, USA, 2007. ACM.

[5] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[6] Luke Maurer, Paul Downen, Zena M. Ariola, and Simon Peyton Jones. Compiling Without Continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 482–494, New York, NY, USA, 2017. ACM.

[7] Chris Okasaki, Peter Lee, and David Tarditi. Call-by-need and continuation-passing style. *LISP and Symbolic Computation*, 7(1):57–81, Jan 1994.

[8] Amr Sabry and Matthias Felleisen. Reasoning About Programs in Continuation-passing Style. *SIGPLAN Lisp Pointers*, V(1):288–298, January 1992.

[9] Masataka Sassa, Toshiharu Nakaya, Masaki Kohama, Takeaki Fukuoka, and Masahito Takahashi. Static single assignment form in the COINS compiler infrastructure. *Proc. SSGRR 2003w*, 2003.

[10] Eijiro Sumii. MinCaml: A Simple and Efficient Compiler for a Minimal Functional Language. In *Proceedings of the 2005 Workshop on Functional and Declarative Programming in Education*, FDPE '05, pages 27–38, New York, NY, USA, 2005. ACM.

[11] Hayo Thielecke. Comparing Control Constructs by Double-Barrelled CPS. *Higher-Order and Symbolic Computation*, 15(2):141–160, Sep 2002.

[12] 雄大 上野 and 淳 大堀. 制御フローの合流のための計算系. **情報処理学会論文誌プログラミング(*PRO*)**, 1(3):19–33, oct 2008.

付録 A

実験プログラム

A.1 節は第 4 章の実験のプログラムで参照しているモジュールである。 A.2 節、 A.3 節、 A.4 節、 A.5 節、A.6 節は実験の性能測定に使ったプログラムである。

A.1　実験プログラムで用いた型やモジュール等の定義

```
(* header {{{ *)
type 'a mylist =
  | Nil
  | Cons of 'a * 'a mylist;;
let rec of_list = function
  | [] -> Nil
  | x :: xs -> Cons(x, of_list xs);;

exception Exit of int
exception ZeroDouble

(* CPS module {{{ *)
module C = struct
  type ('k, _, _) assoc_hlist =
    | Nil  : ('k, 'z, 'z) assoc_hlist
    | Cons : ('k * 't) * ('k, 'u, 'x) assoc_hlist -> ('k, 't -> 'u, 'x) assoc_hlist
  let rec (@) : type k ty1 ty2 v.
    (k, ty1, ty2) assoc_hlist ->
    (k, ty2, v) assoc_hlist ->
    (k, ty1, v) assoc_hlist  = fun a b ->
    match a with
    | Nil -> b
    | Cons(x, xs) -> Cons(x, xs @ b)
  let rec lookup : type k a b.
    k -> (k, a, b) assoc_hlist -> b = fun key -> function
    | Cons((key', x), ls) -> if key = key' then Obj.magic x else lookup key ls
```

```
    | Nil -> raise Not_found
end;;
(* }}} *)
(* for stream {{{ *)
type ('a, 'b) stream_shape =
  | Empty
  | Block of 'a * 'b;;
type _ stream = E : 'b * ('b -> ('a, 'b) stream_shape) -> 'a stream;;
type ('a, 'b) pair = P of 'a * 'b;;
(* }}} *)
(* }}} *)
```

A.2 rev

プログラム A.1 Direct

```
let rev = fun l ->
  let rec work = fun l1 l2 ->
    match l1 with
    | Nil -> l2
    | Cons(a, l) -> work l (Cons(a, l2))
  in work) l Nil
in rev __arg__
```

プログラム A.2 Joel

```
let rec work1 = fun l1 l2 ->
  match l1 with
  | Nil -> l2
  | Cons(a, l) -> work1 l (Cons(a, l2))
in work1 __arg__ Nil
```

プログラム A.3 CPS

```
let rec work = fun l1 __k33 __h32 ->
  __k33 (fun l2 __k36 __h37 ->
    match l1 with
    | Nil -> __k36 l2
    | Cons(a, l) -> work l (fun __fv45 -> __fv45 (Cons(a, l2)) __k36 __h37) __h37)
in work __arg__ (fun __fv20 -> __fv20 Nil (fun __kinit -> __kinit) C.Nil) C.Nil
```

A.3 exists

プログラム A.4　Direct

```
let find = fun p xs ->
  let rec go = fun x109 ->
    match x109 with
    | Cons(x, xs') ->
      (match p x with
       | true -> Some(x)
       | false -> go xs')
    | Nil -> None
  in go xs
in
let exists = fun p xs ->
  match find p xs with
  | Some(x) -> true
  | None -> false
in exists (fun x110 -> match x110 with 5 -> true| _ -> false) __arg__
```

プログラム A.5　Joel

```
let rec go114 = fun x111 ->
  match x111 with
  | Cons(x, xs') ->
    (match x with
     | 5 -> true
     | _ -> go114 xs')
  | Nil -> false
in go114 __arg__
```

プログラム A.6　CPS

```
let rec go = fun x121 __k196 __h195 ->
  match x121 with
  | Cons(x, xs') ->
    (match x with
     | 5 -> __k196 Some((x))
     | _ -> go xs' __k196 __h195)
  | Nil -> __k196 None
in go __arg__ (fun __c160 -> match __c160 with Some(x) -> true| None -> false) C.Nil
```

A.4　mapfold

プログラム A.7　Direct

```
let map = fun f xs ->
```

```
  let rec workm = fun x332 ->
    match x332 with
    | Cons(x, xs) -> Cons(f x, workm xs)
    | Nil -> Nil
  in workm xs
in
let fold = fun f z xs ->
  let rec workf = fun z x333 ->
    match x333 with
    | Cons(x, xs) -> workf (f z x) xs
    | Nil -> z
  in workf z xs
in fold (fun x y -> x + y) 0 (map (fun x -> x * x) __arg__)
```

プログラム A.8　Joel

```
let rec workm = fun x334 ->
  match x334 with
  | Cons(x, xs) -> Cons(x * x, workm xs)
  | Nil -> Nil
in
let xs338 = workm __arg__
in
let rec workf336 = fun z x335 ->
  match x335 with
  | Cons(x, xs) -> workf336 z + x xs
  | Nil -> z
in workf336 0 xs338
```

プログラム A.9　CPS

```
let rec workm = fun x346 __k490 __h489 ->
  match x346 with
  | Cons(x, xs) -> workm xs (fun __a497 -> __k490 (Cons(x * x, __a497))) __h489
  | Nil -> __k490 Nil
in workm __arg__ (fun __av355 ->
  let rec workf = fun z __k435 __h434 ->
    __k435 (fun x347 __k438 __h439 ->
      match x347 with
      | Cons(x, xs) -> workf (z + x) (fun __fv445 -> __fv445 xs __k438 __h439)
          __h439
      | Nil -> __k438 z)
  in workf 0 (fun __fv422 -> __fv422 __av355 (fun __kinit -> __kinit) C.Nil) C.Nil)
     C.Nil
```

A.5 try-mapfold

プログラム A.10 Direct

```
let map = fun f xs ->
  let rec work = fun x1353 ->
    match x1353 with
    | Cons(x, xs) -> Cons(f x, work xs)
    | Nil -> Nil
  in work xs
in
let fold = fun f z xs ->
  let rec work = fun z x1354 ->
    match x1354 with
    | Cons(x, xs) -> work (f z x) xs
    | Nil -> z
  in work z xs
in
let double = fun x1355 ->
  match x1355 with
  | 0 -> raise ZeroDouble
| x -> x * x
in
let safe_double = fun x ->
  try double x with ZeroDouble -> 1
in fold (fun x y -> safe_double x + safe_double y) 0 (map safe_double __arg__)
```

プログラム A.11 Joel

```
let rec work = fun x1356 ->
  match x1356 with
  | Cons(x, xs) -> Cons(try x * x with ZeroDouble -> 1, work xs)
  | Nil -> Nil
in
let xs1367 = work __arg__
in
let rec work1359 = fun z x1357 ->
  match x1357 with
  | Cons(x, xs) -> work1359 (try z * z with ZeroDouble -> 1 + (try x * x with
      ZeroDouble -> 1)) xs
  | Nil -> z
in work1359 0 xs1367
```

プログラム A.12 CPS

```
let rec work = fun x1375 __k1557 __h1556 ->
  match x1375 with
  | Cons(x, xs) -> work xs (fun __a1564 -> __k1557 (Cons(x * x, __a1564))) __h1556
  | Nil -> __k1557 Nil
in work __arg__ (fun __av1389 ->
  let rec work = fun z __k1502 __h1501 ->
    __k1502 (fun x1376 __k1505 __h1506 ->
      match x1376 with
     | Cons(x, xs) -> work (z * z + x * x) (fun __fv1512 -> __fv1512 xs __k1505
         __h1506) __h1506
     | Nil -> __k1505 z)
  in work 0 (fun __fv1489 -> __fv1489 __av1389 (fun __kinit -> __kinit) C.Nil) C.Nil
     ) C.Nil
```

A.6 stream

プログラム A.13 Direct

```
let of_mylist =
  let rec list_length = fun x686 ->
    match x686 with
    | Nil -> 0
    | Cons(_, xs) -> 1 + list_length xs
  in
  let rec list_nth = fun i x687 ->
    match x687 with
    | Nil -> raise Not_found
    | Cons(x, xs) ->
      (match i with
       | 0 -> x
       | _ -> list_nth (i - 1) xs)
  in
  let step = fun p ->
    match p with P(i, lst) ->
     (match i < list_length lst with
      | true -> Block(list_nth i lst, P(i + 1, lst))
      | false -> Empty)
   in fun lst -> E(P(0, lst), step)
in
let fold = fun f z str ->
  match str with E(s, step) ->
    let rec loop = fun z s ->
      match step s with
      | Empty -> z
```

```
      | Block(a, t) -> loop (f z a) t
    in loop z s
in
let map = fun f str ->
  match str with E(s, step) ->
    let new_step = fun s ->
      match step s with
      | Empty -> Empty
      | Block(a, t) -> Block(f a, t)
    in E(s, new_step)
in fold (fun x y -> x + y) 0 (map (fun x -> x * x) (of_mylist __arg__))
```

プログラム A.14　Joel

```
let rec list_length = fun x688 ->
  match x688 with
  | Nil -> 0
  | Cons(_, xs) -> 1 + list_length xs
in
let rec list_nth = fun i x689 ->
  match x689 with
  | Nil -> raise Not_found
  | Cons(x, xs) ->
    (match i with
     | 0 -> x
     | _ -> list_nth (i - 1) xs)
in
let rec loop690 = fun z s ->
  match s with P(i, lst) ->
    (match i < list_length lst with
     | true ->
       (match Block(list_nth i lst, P(i + 1, lst)) with
        | Empty -> z
        | Block(a, t) ->
          let a701 = a * a in loop690 (z + a701) t)
     | false -> z)
in loop690 0 (P(0, __arg__))
```

プログラム A.15　CPS

```
let rec list_length = fun x708 __k1009 __h1008 ->
  match x708 with
  | Nil -> __k1009 0
  | Cons(_, xs) -> list_length xs (fun __e21017 -> __k1009 (1 + __e21017)) __h1008
in
let rec list_nth = fun i __k969 __h968 ->
```

```
 __k969 (fun x709 __k972 __h973 ->
   match x709 with
   | Nil -> C.lookup "Not_found" __h973
   | Cons(x, xs) ->
     (match i with
      | 0 -> __k972 x
      | _ -> list_nth (i - 1) (fun __fv986 -> __fv986 xs __k972 __h973) __h973))
in
let rec loop = fun z __k852 __h851 ->
  __k852 (fun s __k855 __h856 ->
    match s with P(i, lst) ->
      list_length lst (fun __e2954 ->
        match i < __e2954 with
        | true -> list_nth i (fun __fv938 ->
          __fv938 lst (fun __a920 ->
            match Block(__a920, P(i + 1, lst)) with
            | Empty -> (__k855) (z)
            | Block(a, t) ->
              (match Block(a * a, t) with
               | Empty -> __k855 z
               | Block(a, t) -> loop (z + a) (fun __fv864 -> __fv864 t __k855 __h856
                   ) __h856)) __h856) __h856
        | false -> __k855 z) __h856)
in loop 0 (fun __fv839 -> __fv839 (P(0, __arg__)) (fun __kinit -> __kinit) C.Nil) C.
    Nil
```