MinCamlコンパイラ

住井 英二郎

約 2000 行の ML コードにより実装された、教育目的のコンパイラ MinCaml について議論する。対象言語は値呼び、非純粋、暗な単相型を持つ高階関数型言語である。レイトレーシングを含むいくつかのアプリケーション・プログラムに対し、実行速度において Objective Caml や GCC とほぼ同等のコードを生成することができた。

We present an educational compiler, MinCaml, implemented in 2000 lines of ML. The target language is a strict, impure, implicitly typed, higher-order functional language. The compiler produces as fast code as Objective Caml and GCC do for several application programs including a ray tracer.

1 序論

1.1 背景

東京大学理学部情報科学科では、1992 年度より、「CPU 実験」ないし「プロセッサ実験」と呼ばれる演習（正式には「情報科学実験 II」の一部）が行われている。この演習では、学生が 5〜6 人程度のグループを作り、FPGA（プログラム可能な集積回路の一種）を用いて、独自の CPU を設計・実装する。また、その CPU をターゲットとするクロスコンパイラを作成し、与えられたレイトレーシング・プログラムの実行速度を競う[†1]。

MinCaml[†2]は、情報科学実験 II において学生が作成するコンパイラの参照実装として、2001 年から 2005 年にかけて筆者が作成したコンパイラである。このような背景から、MinCaml は以下の要件を中心に設計された。

1. 一学期間の解説（1 週 30 分〜60 分程度、計 14 週）と演習により、学部学生（3 年後期）がコンパイラの実装を詳細に理解し、自力で移植・改良できるようになること。
2. CPU 実験の課題であるレイトレーシング・プログラムを、十分な効率で実行できること。

1.2 節以降に述べる MinCaml の設計は、これらの要件の系として比較的自然に決定された。

表 1 に 2001 年度〜2002 年度の情報科学実験 II で、MinCaml を用いた部分のカリキュラム例を示す。2003 年度以降は、教員および TA の交代により、内容が変化している。また、2005 年度以降は、MinCaml のソースコードの公開にともない、多相関数の実装方法や代数的データ型のパターンマッチングなど、より高度な内容が追加されている。

1.2 MinCamlの概要

MinCaml の対象言語は、非純粋[†3]・値呼び[†4]で、暗な[†5]単相型[†6]を持つ、必要最小限の関数型言語（2 章）を定義した。これは、（変数への破壊的代入がな

The MinCaml Compiler.

Eijiro Sumii, 東北大学大学院情報科学研究科, Graduate School of Information Sciences, Tohoku University.

コンピュータソフトウェア, Vol.??, No.? (????), pp.??–??.
xxxx 年 yy 月 zz 日受付.

†1 http://www.is.s.u-tokyo.ac.jp/vu/jugyo/processor/

†2 http://min-caml.sf.net/

†3 副作用があること

†4 変数に束縛される式の値が、束縛の際に直ちに計算されること

†5 ソースコード上で型が明示されず、推論されること

†6 多相型がないこと

表 1 カリキュラム例

週	内容
1	イントロダクション、字句・構文解析
2	K 正規化
3	α 変換、β 変換、let 式の A 正規化
4	インライン展開、定数畳み込み、不要定義除去
5	クロージャ変換、既知関数呼び出し最適化
6	仮想機械語生成
7	関数呼び出し規約
8	レジスタ割り当て
9	レジスタ溢れ
10	アセンブリ生成
11	末尾呼び出し最適化、継続渡し形式
12	型推論、浮動小数点演算
13	ごみ集め（実装は必須でない）
14	型を応用したプログラム解析の例：エスケープ解析（実装は必須でない）

いという意味での）関数型言語のほうが、（はじめから一種の単一代入形式になっているという意味で）最適化等の処理が単純になると考えたためである。ただし、レイトレーシングにともなう配列への破壊的代入や入出力を簡単にするため、それらの副作用は容認した。また、サンク†7を生成・更新するオーバーヘッド（ないし正格性†8・線形性†9解析の必要）を避けるため、評価戦略は必要呼び（遅延評価）ではなく値呼びとした。さらに、浮動小数点演算と整数演算の両方を単純かつ高速に実現するため、単相型推論を導入した。

コンパイラの実装言語としては Objective Caml†10を用いた。これは以下の 2 つの理由による。

- 代数データ型やパターンマッチング、静的型検査（パターンマッチングの網羅性検査を含む）といった ML の機能が、関数型言語コンパイラの実装に適していること
- すでに「情報科学実験 I」（3 年前期）において、8 週にわたり Objective Caml によるプログラミング演習が行われ、多相型推論を含む ML サブセットのインタプリタなど、言語処理系に関連する比較的高度な課題が出されていること

また、OCamlLex と OCamlYacc による字句・構文解析を除き、いわゆるコンパイラ・フレームワーク†11の類は用いないこととした。これは、

- 学生がコンパイラの実装を細部まで理解するため
- フレームワーク自体の仕組みや使い方を学習するオーバーヘッドを避けるため

の 2 点による。

同様の理由（および性能の要求）により、生成するコードはバイトコードではなくネイティブコードとした。ターゲットの命令セットアーキテクチャとしては、学科でもっとも簡単に利用できる 32 ビット RISC プロセッサである SPARC V8†12を選択した。（SPARC 以外の RISC プロセッサへの移植は比較的容易なはずである。実際、たとえばお茶の水女子大学の浅井健一氏による PowerPC への移植が存在する [私信, 2005 年 11 月]。一方、たとえば IA-32 のような 2 オペランド命令セットアーキテクチャや、SSE 以前の IA-32 の FPU のようなスタックマシンへの対応は、より複雑になる。ただし、学生による IA-32 への移植はいくつか存在する。）

2 対象言語

図 1 に MinCaml コンパイラの対象言語（それ自体も MinCaml と呼ぶ）の抽象構文と型を示す。MinCaml の表層文法は、Objective Caml の極めて小さなサブセットとなっている。このサブセットは以下の理由によって設計された。

まず、通常の実用的関数型言語と同様に、MinCaml もプリミティブとして整数、浮動小数点数、論理値、組、関数等を持つ。それぞれについて、少なくとも一つの構成子（定数、関数定義、組の作成など）と、少

†7 遅延評価において、式とその環境を保存するためのデータ構造

†8 遅延評価において、式が必ず評価される（ので遅延する必要がない）という性質

†9 遅延評価において、式の値が高々一回しか使用されない（ので保存する必要がない）という性質

†10 http://caml.inria.fr/

†11 コンパイラを作成するための専用のソフトウェア

†12 http://www.sparc.com/standards/V8.pdf

e ::=	式
c	定数
$op(e_1, \ldots, e_n)$	算術演算
`if` e_1 `then` e_2 `else` e_3	条件分岐
`let` $x = e_1$ `in` e_2	変数定義
x	変数の読み出し
`let rec` $x\ y_1\ \ldots\ y_n = e_1$ `in` e_2	再帰関数定義
$e\ e_1\ \ldots\ e_n$	関数呼び出し
$(e_1, \ldots, e_n)$	組の作成
`let` $(x_1, \ldots, x_n) = e_1$ `in` e_2	組からの読み出し
`Array.create` $e_1\ e_2$	配列の作成
$e_1.(e_2)$	配列からの読み出し
$e_1.(e_2) \leftarrow e_3$	配列への書き込み
τ ::=	型
π	プリミティブ型
$\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$	関数型
$\tau_1 \times \ldots \times \tau_n$	組型
τ `array`	配列型

図 1　MinCaml の抽象構文と型

なくとも一つの破壊子（算術演算、条件分岐、関数呼び出し、組からの読み出しなど）がある。

MinCaml には一般的な代数的データ型やパターンマッチングがないため、論理値と条件分岐はプリミティブとした[†13]。また、組からの読み出しは、$\#_i(M)$ のような射影ではなく、一種の特殊なパターンマッチングの構文とした。これは、レコード多相性 [9] のない型システムにおける、いわゆる flex record の問題[†14]を回避するためである。

MinCaml は、変数と関数の名前空間を分離せず、さらに自由変数を持つネストした関数定義も許すことにより、高階関数をサポートしている。これはレイトレーシングには必要とされないが、（最近は関数型言語だけでなく各種言語にも導入されている）クロージャの概念を学習するためである。

ただし、MinCaml の関数定義は非カリー化されており、部分適用をサポートしない。部分適用が必要な場合は、適当な名前の関数を定義する必要がある。たとえば f が二引数関数の場合、$f\ 3$ のような部分適用は、`let rec` $f_3\ y = f\ 3\ y$ `in` f_3 のように書かなければならない（この点は ML より Scheme の関数定義・関数適用に近い）。

レイトレーシングにおけるベクトルや行列の演算を単純に実現するため、要素への破壊的代入が可能な配列をサポートした。MinCaml には多相型がないが、配列の操作は要素型について多相的でなければならないので（Objective Caml のようなライブラリ関数ではなく）プリミティブとした。また、要素が一個であるような配列により、破壊的代入の可能な参照セルも実現している。なお、境界検査は（演習の課題となっているが）標準では実装されていない。

MinCaml は一般的な代数的データ型や、その値に対するパターンマッチングをサポートしていない。もし MinCaml を「ML の実装」とみなすならば、これは重要な機能の欠如といえる。しかし、以下の理由により、標準の MinCaml では代数的データ型とパターンマッチングを省略した（ただし、それらを実装した学生もいる）。

- MinCaml は、特定の言語の実装を説明することだけを目指してはいない。
- 代数的データ型のパターンマッチングは、必ずしも多くのプログラミング言語に導入されていない。
- MinCaml の他の言語機能と比べ、パターンマッチングは実装が複雑になりやすい。
- 代数的データ型がなくとも、ある程度のアプリケーション・プログラムを記述できる。

代数的データ型を省略したため、多相データ型も標準ではサポートされていない（ただし、毎年一名〜数名程度の学生は、浮動小数点数用とそれ以外用の 2 つにコードを複製することで、多相関数を実現して

†13　もし代数的データ型とパターンマッチングがあれば、論理値は `type bool` $=$ *true* | *false*、条件分岐は `match` e_1 `with` *true* $\rightarrow e_2$ | *false* $\rightarrow e_3$ と表現できる。

†14　たとえば「組 x を受け取り、x の第一要素を返す」という関数に最も一般的な型が存在せず、型推論ができない

いる)。

また、ごみ集めについても授業では解説しているが、標準の MinCaml には実装されていない。レイトレーシングのプログラムは、メインループでヒープ確保を行なわないよう、注意して書かれている(ただし、一部の学生はごみ集めも実装し、オセロの思考ルーチンなど、ごみ集めを必要とするアプリケーション・プログラムを独自 CPU で実行している)。

3 内部の設計と実装

MinCaml コンパイラは、2 節の対象言語を、SPARC アセンブリに変換する。しかし、これらの言語の間には以下のように大きなギャップが存在する。

1. 型。MinCaml は暗な単相型を持つが、アセンブリには静的型がない。
2. ネストした式。MinCaml では(たとえば $a*x+b*y+1$ などのように)いくらでも深くネストした一つの式が許されるが、通常のアセンブリでは一つの命令につき一つの演算しかできない。
3. ネストした関数定義。MinCaml では

```
let rec make_adder x =
  let rec adder y = x + y in
  adder in
(make_adder 3) 7
```

のようなネストした定義が可能だが、アセンブリにはトップレベルの「ラベル」しかない。
4. MinCaml には組や配列などのデータ構造があるが、アセンブリにはない。
5. MinCaml では任意の個数の変数を使用できるが、アセンブリでは有限のレジスタしか使用できず、メモリへの退避と復帰が必要となる。

MinCaml は、適切な中間言語を定義し (1) 型推論 (2) K 正規化 (3) クロージャ変換 (4) 仮想機械語生成 (5) レジスタ割り当て という 5 つの主な処理を順番に行うことにより、これらのギャップを一つずつ埋めている。

MinCaml コンパイラを構成する主要モジュール(Objective Caml のストラクチャ)の一覧と行数を表 2 に示す。モジュールのインターフェース(Objective Caml のシグネチャ)や補助モジュールも合わせると、現在の MinCaml コンパイラ(バージョン 0.0.11)のソースコードは 2040 行である。ただし、その他に SPARC アセンブリと C で書かれた、215 行のランタイム・ライブラリがある。

表 2 MinCaml の主要モジュールと行数

モジュール	行数
字句解析(OCamlLex ファイル)	100
構文解析(OCamlYacc ファイル)	170
型推論	163
K 正規化	179
α 変換	46
β 簡約	38
`let` 式の A 正規化	18
インライン展開	32
定数畳み込み	50
不要定義削除	32
クロージャ変換	106
仮想機械語生成	164
13 ビット即値最適化	42
レジスタ割り当て	251
アセンブリ生成	255

以下、これらのモジュールの概要を説明する。より詳細な疑似コードは `http://min-caml.sf.net/min-caml.pdf` から、ソースコードは `http://prdownloads.sourceforge.net/min-caml/min-caml-0.0.11.tar.gz?download` からダウンロードできる。

3.1 字句・構文解析

通常の言語処理系と同様、MinCaml もプログラムを単なる文字列からトークンの列に変換し(字句解析)、それから構文木を構築する(構文解析)。これらのアルゴリズム自体はコンパイラに関する講義ですでに扱われているため、MinCaml では立ち入らないこととし、単に OCamlLex と OCamlYacc を用いた。MinCaml の文法記述は、全体としては容易であった。ただし、通常の Objective Caml の字句・構文解析と同様、`x-3` のような整数減算を、`x(-3)` のような関数

適用と解釈されないよう、「括弧をつけなくても関数の引数になりうる式」(simple expressions) と通常の式を区別した（たとえば、`-3` は simple expression でないため、`x-3` は関数適用とならない）。

3.2 型推論

すでに述べたように、MinCaml は（主に実行時性能のため）暗な単相型を持つ。MinCaml の型推論は、型への参照 (reference) のオプション値 (`None` または `Some r`) により型変数を表現する、標準的な単一化により実現されている†15。

ただし、単相型であるため、具体化されなかった型変数は、∀で量化するのではなく `int` を代入する。具体化されないということは、その型の値が定義されることも使用されることもないので、任意の型を代入しても（閉じたプログラムの）健全性が損われることはない。`unit` や `void`（空の型）ではなく `int` を選択した理由は、次に述べる外部関数・外部配列に現れる型として、`int` が最も多かったためである。

単相型であるため、ML の多相型と異なり、プログラム中に現れる自由変数（実際には外部関数ないし外部配列）の型も推論することができる [7]。ただし、この場合、「具体化されなかった型変数に `int` を代入する」ことは一般には健全でない。たとえば、`print_float (read_float ())` のようなプログラムを与えると、`read_float` の返り値（すなわち `print_float` の引数）の型は、`float` ではなく `int` と推論されてしまう。このような自由変数の型は、型推論の際の初期型環境において、陽に指定しなければならない。

3.3 K 正規化

型推論の後、MinCaml は構文木を K 正規形 [3] に変換する。K 正規形とは、ネストしている式の評価途中の値をすべて一時変数に束縛した形式である。図 2 に MinCaml の K 正規形の抽象構文を示す。

たとえば、整数加算 $e_1 + e_2$ は、`let` $x_1 = e_1'$ `in let` $x_2 = e_2'$ `in` $x_1 + x_2$ のように K 正規化

†15 このような実現法の直観的な概要については、たとえば `http://itpro.nikkeibp.co.jp/article/COLUMN/20070717/277580/` などを参照されたい。

e ::=
- c
- $op(x_1, \ldots, x_n)$
- `if` $x = y$ `then` e_1 `else` e_2
- `if` $x \leq y$ `then` e_1 `else` e_2
- `let` $x = e_1$ `in` e_2
- x
- `let rec` $x\ y_1\ \ldots\ y_n = e_1$ `in` e_2
- $x\ y_1\ \ldots\ y_n$
- $(x_1, \ldots, x_n)$
- `let` $(x_1, \ldots, x_n) = y$ `in` e
- $x.(y)$
- $x.(y) \leftarrow z$

図 2 MinCaml の K 正規形

される（ただし、e_1' は e_1 の、e_2' は e_2 の K 正規形とする）。一般的な K 正規化は、抽象構文の再帰処理により自然に実現できる。

ただし、たとえば e_1 や e_2 が最初から y や z などの変数だった場合、`let` $x_1 = y$ `in let` $x_2 = z$ `in` $x_1 + x_2$ ではなく単純に $y + z$ と変換できるよう、以下のような補助関数 `insert_let` を利用している。

```
let insert_let e k =
  match e with
  | Var(x) -> k x
  | _ ->
      let x = gensym () in
      Let(x, e, k x)
```

ここで、`e` は変数に束縛される式、`k` は `e` が束縛される変数の名前を引数とする、一種の部分継続 [4] [8] である。具体的には、たとえば整数加算の場合であれば

```
let rec f = function
  | Add(e1, e2) ->
      insert_let (f e1)
        (fun x1 -> insert_let (f e2)
            (fun x2 -> Add(x1, x2)))
  | ...
```

のようなパターンマッチングにより、K 正規化を実装する再帰関数 `f` を定義している。

なお、以降の処理を簡単にするため、K 正規化の「ついで」として、if 式は if $x = y$ then e_1 else e_2 または if $x \leq y$ then e_1 else e_2 のいずれかの形に変換される。たとえば、if $x < y$ then e_1 else e_2 は if $y \leq x$ then e_2 else e_1 に、if x then e_1 else e_2 は let $z = \mathit{false}$ in if $x = z$ then e_2 else e_1（ただし z は e_1 や e_2 に出現しない変数）に変換する。これは SPARC を含め、通常のアセンブリ言語においては、比較命令と分岐命令を連続して一緒に使用する必要があるためである。

関数型言語処理系の中間言語としては、継続渡し形式 (CPS) [11] [1] や A 正規形[†16] [5] が有名である。K 正規形は A 正規形と類似しているが、let 式の = の右辺にも let 式が出現する（let 式が末尾位置以外に出現する）ことを許しており、関数のインライン展開などが単純になる。また、CPS や A 正規形では if 式が末尾位置にしか出現できず、そうでない場合は継続生成かコード複製が必要となるが、K 正規形にはそのような制限はない。

3.4 α 変換

K 正規化に続き、インライン展開等の各種最適化を容易にするため、束縛変数の名前をすべて相異なるものにつけかえる（α 変換）。後に述べるインライン展開で利用するため、変数名から変数名への対応 ε を引数として受け取り、ε に従って（束縛変数だけでなく）自由変数の名前もつけかえる関数を実装・公開している。

3.5 β 簡約

let $x = y$ in e のように、変数の別名を定義しているだけの let 式はコンパイル時に展開する（λ 計算における β 簡約の特殊な場合となっている）。これは、特に他の最適化後に有効な場合がある。

3.6 let 式の A 正規化

let $x =$ (let $y = e_1$ in e_2) in e_3 のようにネストした let 式は、let $y = e_1$ in let $x = e_2$ in e_3 のように平らにすることができる（let 式の A 正規化）。これ自体はコンパイル結果の性能に影響しないが、中間コードが人間にとってわかりやすくなるだけでなく、他の最適化の可能性を高めることができる。

MinCaml における let 式の A 正規化は、以下のように実装されている。

```
let rec f = function
  | Let(x, e1, e2) ->
      let rec insert = function
        | Let(y, e3, e4) ->
            Let(y, e3, insert e4)
        | LetRec(fundef, e) ->
            LetRec(fundef, insert e)
        | LetTuple(ys, z, e) ->
            LetTuple(ys, z, insert e)
        | e -> Let(x, e, f e2) in
      insert (f e1)
  | ...
```

すなわち、let $x = e_1$ in e_2 のような let 式に対しては、まず e_1 について（let 式の）A 正規化を行い、その結果を let ... in e_1' とする。ただし、let ... in は 0 個以上の let の列とし、e_1' は let 式でないとする。また、e_2 の（let 式に関する）A 正規形を e_2' とおく。すると、元の let 式の A 正規形は let ... in let $x = e_1'$ in e_2' となる。

3.7 インライン展開

一定サイズ以下の関数の呼び出しを、その関数の本体で置き換える。インライン展開する関数のサイズの上限は、コマンドライン・オプション (-inline) により指定できる（中間コードをわかりやすくするため、デフォルト値は 0、すなわちインライン展開をまったく行わないとしている）。関数のサイズは、本体の K 正規形構文木のノード数としている。

K 正規形におけるインライン展開は、3.4 節で述べた ε として仮引数名から実引数名への対応をとり、α 変換関数を呼び出すだけで容易に実現できる。これは

†16 let 式の = の右辺に let 式や if 式が出現しないよう、制限された K 正規形のこと。ただし歴史的には、まず A 正規形が提案され [5]、それから制限を緩和した K 正規形が提案された [3]。

関数呼び出しの実引数が、すべて変数名となっているためである。

3.8 定数畳み込み

let $x = 3$ in let $y = 7$ in $x + y$ における $x + y$ のように、変数の値によってコンパイル時に計算できる算術演算（および組からの読み出し）があれば、let $x = 3$ in let $y = 7$ in 10 のように計算して置き換える（定数畳み込み）。実装としては、$x = 3$ や $y = 7$ など let 式による束縛を記憶し、$x + y$ のような演算があれば引数の値を調べて、可能ならば計算して置き換えている。

3.9 不要定義削除

定数畳み込み等の後、let $x = 3$ in let $y = 7$ in 10 における let $x = 3$ や let $y = 7$ のように、使用されていない変数定義（および関数定義）は削除する。一般に、e_1 に副作用がなく、x が e_2 に現れなければ、let $x = e_1$ in e_2 を e_2 に置き換える。「副作用がない」という意味論的条件は、「関数適用と配列への書き込みがない」という構文論的条件により、保守的に近似している。

なお、β 簡約から不要定義削除までの最適化は相互に依存する可能性があるので、コマンドライン・オプション (-iter) で指定された一定の回数を上限として、プログラムが変化しなくなるまで反復する（デフォルトは 1000）。

3.10 クロージャ変換

K 正規形での最適化処理が完了したら、ネストした関数定義を、トップレベルのみの関数定義とクロージャ生成に置き換える（クロージャ変換）。ただし、クロージャを生成する必要がない場合は生成しない。「クロージャを生成する必要がない場合」とは、関数定義に自由変数がなく、かつ、関数自体が値として渡されない場合としている。（後者の条件は、値として渡された関数がクロージャで表現されているかいないか、という解析を省略するためである。）

このために、まず（let rec f x = x + x in ... (f y) ... の f のように）自由変数のない関数があれば記憶しておく。そして、（f y のように）自由変数がないと記憶されている関数の呼び出しは、クロージャを用いる関数呼び出しと区別する（前者は *apply_direct*、後者は *apply_closure* という特殊構文に変換する）。その上で、f が *apply_direct* 以外に変数として現れなければ、f のクロージャ生成を省略する。

3.11 仮想機械語生成

クロージャ変換に続き、組・配列およびクロージャの生成や使用に関わるメモリ操作を, すべてロード命令・ストア命令に変換する。結果として出力されるコードは、変数・if 式・関数呼び出しを除き、SPARC アセンブリに近いものとなる。

3.12 13 ビット即値最適化

SPARC アセンブリでは、整数加算など一部の命令において、第 2 オペランドに 13 ビット以下の符号つき整数を即値として記述することができる。これを活用するため、定数畳み込み（および不要定義削除）と同様の方法により、13 ビット以下の符号つき整数の変数定義は展開する。

3.13 レジスタ割り当て

let 式で束縛される変数に対し、貪欲アルゴリズムにより前から順に、干渉のないレジスタを割り当てる。MinCaml には変数への破壊的代入がないため、通常のレジスタ割り当てよりも単純になっている（一部の学生は手続き間レジスタ割り当てなど、より高度なアルゴリズムを実装している）。

ただし、A 正規形や CPS と異なり、K 正規形では let 式や if 式が末尾位置以外に出現しうる。したがって、たとえば

```
let x =
  if a = b
  then (let y = e1 in e2)
  else c
in e3
```

という式の y にレジスタを割り当てる際には、e_2 だけでなく e_3 の自由変数に割り当てられているレジス

タも避けなければならない。このため、let $y = e_1$ の継続に相当する、let $x = e_2$ in e_3 のような式を作り、レジスタ割り当てを行う関数に引数として渡している。

この点は K 正規形よりも、A 正規形や CPS のほうが単純になると言える。しかし、3.3 節で述べたように、MinCaml は

- if 文が末尾位置にしか来られないことによるオーバーヘッドを避けるため
- インライン展開が単純になるため

という二つの理由により、K 正規形を採用した。

レジスタ溢れが起こったときは、継続に相当する式を先読みし、もっとも後まで使われない変数を退避する（条件分岐がある場合は簡単のため、then 節が else 節より先に来るとみなしている）。また、次のアセンブリ生成において、関数呼び出しにおける引数並び替えを減らすため、関数呼び出し規約を考慮して、継続に相当する式を先読みし、ある種の register coalescing (targeting) を行っている。

3.14 アセンブリ生成

最後に、if 式を比較命令と分岐命令に、関数呼び出しを適切な命令の列に変換し、アセンブリを生成する。関数呼び出しの変換は、主に引数並び替えと末尾呼び出し最適化の 2 点からなる。

引数並び替え (shuffling)[†17]では、必要があれば一時レジスタを使用しつつ、関数呼び出し規約にしたがって、所定のレジスタに引数をセットする。これは以下のような再帰関数により実現されている。

```
(* 並列 mov を表現するリストを受け取り、
   逐次 mov を表現するリストに変換 *)
let rec shuffle pmov =
 (* mov 元と mov 先が等しい mov を取り除く *)
 let (_, pmov) =
  List.partition
   (fun (x, y) -> x = y)
   pmov in
(* サイクルのある並列 mov とない mov に分類 *)
match
 (List.partition
   (fun (_, y) -> List.mem_assoc y pmov)
   pmov) with
| [], [] -> []
| (x, y) :: pmov, [] ->
   (* 一時変数を使用してサイクルを解消 *)
   (y, tmp) :: (x, y) ::
   shuffle
    (List.map
      (function
        | (x', y') when x' = y -> (tmp, y')
        | xy -> xy)
      pmov)
| pmov, acyc ->
   (* サイクル以外を逐次 mov *)
   acyc @ shuffle pmov
```

末尾呼び出し最適化は、アセンブリ生成を行う関数に対し、末尾位置を表す Tail か、末尾位置でないことを表す NonTail r のどちらかを渡すことにより実現されている。NonTail r の場合は式の値を計算し、結果をレジスタ r に格納する。Tail の場合は、末尾呼び出しのジャンプをするか、ret 命令で値を返す。

たとえば、NonTail r1 に対する、式 $r_2 + r_3$ の SPARC アセンブリは、add r2, r3, r1 となる。別の例として、式 let $r = e_1$ in e_2 の Tail に対するアセンブリ生成は、まず e_1 の NonTail r に対するアセンブリ生成を行い、それから e_2 の Tail に対するアセンブリ生成を行う。また、一般に算術演算 e の Tail に対するアセンブリは、まず関数呼び出し規約により返り値を格納するレジスタ r_0 について、e の NonTail r0 に対するアセンブリを生成し、それから ret 命令を追加する。

4 性能

MinCaml の生成するコードの性能を簡単に評価するため、表 3 にあるプログラムとコンパイラについて、生成されたコードの実行時間を比較した。プログラムは、それぞれのコンパイラの対象言語において、もっとも自然かつ最適と判断したスタイルで記述した（たとえば、MinCaml ではすべての関数を let rec

†17 レジスタ割り当て後に、たとえば f r_4 r_1 r_2 のような関数呼び出しがあった場合、関数呼び出し規約に従って第一引数 r_4 を r_1 に、第二引数 r_1 を r_2 に、第三引数 r_2 を r_3 に移動すること。ただし、それらの移動は「並列に」（あたかも同時であるかのように）行わなければならない。たとえば今の例では、r_4 を r_1 へ移動する前に、r_1 を r_2 へ移動しなければならず、さらにその前に r_2 を r_3 へ移動する必要がある。

表 3　ベンチマーク・プログラムの実行時間（単位：秒）

	MinCaml	OCamlOpt `-unsafe`	GCC4 `-m32`	GCC4 `-m64`	GCC3 `-mflat`
Ackermann	0.3	0.3	1.3	1.8	1.0
Fibonacci	2.5	3.9	1.5	1.4	6.1
Takeuchi	1.6	3.8	3.7	1.6	5.5
Ray Tracing	3.4	7.5	2.3	2.9	2.6
Harmonic	2.6	2.6	2.0	2.0	2.0
Mandelbrot	1.8	4.6	1.7	1.7	1.5
Huffman	4.5	6.6	2.8	3.0	2.9

により定義するが、Objective Caml では `let rec` により定義された関数がインライン展開されないため、再帰関数以外は `let` により定義した）。

実験環境は Sun Fire V880（Ultra SPARC III 1.2 GHz、メインメモリ 8 GB、Solaris 9）である。OCamlOpt は Objective Caml 3.08.3 のネイティブコード・コンパイラ、GCC4 `-m32` は GCC 4.0.0 20050319 の 32 ビット SPARC 用、GCC4 `-m64` は同コンパイラの 64 ビット SPARC 用、GCC3 `-m32 -mflat` は GCC 3.4.3 の 32 ビット SPARC 用だが SPARC のレジスタウィンドウを用いない。OCamlOpt には`-unsafe` を、GCC には`-O3` を、MinCaml には`-inline 100` をオプションとして与えた（なお、MinCaml の`-inline` は、GCC のオプション`--param max-inline-insns-auto=...` と似たような意味だが、後者のデフォルト値は 100 である）。

全体として、表 3 にあるような比較的単純なプログラムにおける MinCaml の性能は、GCC や Objective Caml とほぼ同等のオーダーにある。個々のプログラムにおける差異は、以下のような理由による。

- Objective Caml と GCC3 は再帰関数をインライン展開しない。
- Objective Caml は多相関数をサポートするため、関数の引数や返り値となる浮動小数点数を、レジスタではなくヒープに確保する。
- `-mflat` オプションをつけない GCC は、SPARC の ABI（アプリケーション・バイナリ・インターフェース）規約にしたがいレジスタウィンドウを使用しており、深い再帰関数呼び出しのオーバーヘッドが大きい
- `-mflat` オプションつきの GCC は、eager callee save を採用しており、依然として関数呼び出しのオーバーヘッドが小さくない
- GCC4 は、Fibonacci 関数のインライン展開などで出現する $(n-1)-2$ のような式を $n-3$ などに簡約する
- 32 ビットの GCC は、（SPARC の ABI に従い）関数呼び出しにおいて浮動小数点数を整数レジスタで渡しており、オーバーヘッドがある。
- MinCaml は浮動小数点演算命令のスケジューリングをまったくしていない。

このように、Objective Caml や GCC のほうが遅いケースの多くは、多相関数のサポートや ABI の制限など相応の理由があり、必ずしも MinCaml のほうが優れているというわけではない（ただし、Objective Caml が再帰関数をインライン展開しないという制限は、以前から認識されているようである[†18]）。

5　関連研究

教科書等に述べられているコンパイラを含め、ML のような高級言語やそのサブセットのコンパイラは多数存在するが、MinCaml のように高速なネイティブコードを生成し、かつ小さいコンパイラは筆者の知る範囲ではない。

Hilsdale ら [6] は Scheme サブセットのコンパイラ

†18 `http://caml.inria.fr/pub/old_caml_site/ocaml/speed.html`

を Scheme により実装している。しかし、生成されるコードの性能は目標としておらず、議論されていない。

Sarker ら [10] は Scheme に基づくコンパイラ・フレームワークについて述べている。1.2 節で述べたように、MinCaml は特殊なフレームワークではなく、通常のプログラミング言語である ML で実装されている。また、ML は静的に型つけされているため、中間コードの構文論的な正しさは、コンパイラのコンパイル時に検出される。

Feeley[†19]は、800 行未満の Scheme で実装された、Scheme サブセットから C へのコンパイラについて述べている。しかし、CPS 変換とクロージャ変換を主眼としており、性能は目標としていない。実際に Gambit-C と比較して、生成されたコードの実行時間は 6 倍程度と報告されている。

Atze Dijkstra（Edsger Dijkstra とは別人）と Swierstra[†20]は、属性文法を基本とする Haskell コンパイラを開発している。コード生成は 2006 年 11 月現在、完成していないとある。Haskell のフルセットをサポートすることを目標としており、最新バージョン (20060419) のソースコードは、すでに数万行以上となっている。

Appel [2] は命令型言語 Tiger のコンパイラを ML で実装し、その拡張として高階関数や型推論も議論している。Tiger には変数への破壊的代入があるため、MinCaml と比較すると、レジスタ割り当てや最適化等が複雑になっている。

Sperber[†21]は CPS 変換による、単純な関数型言語から PowerPC アセンブリへのコンパイラについて概説している。実装やソースコードは公開されておらず、詳細は不明である。

†19 http://www.iro.umontreal.ca/~boucherd/mslug/meetings/20041020/

†20 http://www.cs.uu.nl/wiki/Ehc/WebHome

†21 http://www-pu.informatik.uni-tuebingen.de/users/sperber/compilerbau-2001/

6 結論

本稿では学部 3 年生程度の教育を目的とした、非常に単純な関数型言語のコンパイラである MinCaml について解説した。MinCaml は東京大学理学部情報科学科の授業以外に、ケンブリッジ大学の授業[†22]やお茶の水女子大学の研究室、各種読書会などでも利用されている。率直に言って、MinCaml のような “toy compiler” がこれほど評価されるとは考えていなかったが、強いて言えば

- MinCaml の極端な小ささ・単純さ故に敷居が低かった
- 未踏ソフトウェア創造事業や SourceForge、FDPE 2005 (Workshop on Functional and Declarative Programming in Education)、PPL サマースクール 2006 等で発表し、聴衆に応じて、できるだけ面白くわかりやすい説明を心がけた
- 単純な実装とわかりやすい説明が可能となるよう、対象言語やコンパイラ内部の設計を何度も再考した

といった理由が背景にあったのかもしれない。

謝辞

東京大学理学部情報科学科の学生ならびに教員の皆様には、レイトレーシング・プログラムの移植、MinCaml のバグ報告、授業の引き継ぎや改善など、多大なご貢献をしていただきました。心よりお礼を申し上げます。

査読者各位には、本論文に関する貴重なご意見・ご助言、誤りのご指摘をいただきました。本論文の主要対象読者はプログラミング言語研究者を想定していましたが、査読者様のご提案にしたがい、学生も想定して、用語に関する脚注を加えました。大変詳細なご提案をいただいた査読者様および査読者様の学生の方に深く感謝します。

MinCaml コンパイラの開発は、2004 年度第 1 回未踏ソフトウェア創造事業により補助された。

†22 http://web.archive.org/web/20061011214446/http://www.cl.cam.ac.uk/DeptInfo/CST/node42.html

参 考 文 献

[1] Appel, A. W.: *Compiling with Continuations*, Cambridge University Press, 1992.

[2] Appel, A. W.: *Modern Compiler Implementation in ML*, Cambridge University Press, 1998.

[3] Birkedal, L., Tofte, M., and Vejlstrup, M.: From Region Inference to von Neumann Machines via Region Representation Inference, *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1996, pp. 171–183.

[4] Danvy, O. and Filinski, A.: Abstracting Control, *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, 1990, pp. 151–160.

[5] Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M.: The Essence of Compiling with Continuations, *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, 1993, pp. 237–247. In *ACM SIGPLAN Notices*, 28(6), June 1993.

[6] Hilsdale, E., Ashley, J. M., Dybvig, R. K., and Friedman, D. P.: Compiler Construction Using Scheme, *Functional Programming Languages in Education*, Lecture Notes in Computer Science, Vol. 1022, Springer-Verlag, 1995, pp. 251–267.

[7] 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*, 1996, pp. 42–53.

[8] Lawall, J. L. and Danvy, O.: Continuation-Based Partial Evaluation, *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, ACM SIGPLAN Lisp Pointers, Vol. VII, 1994, pp. 227–238.

[9] Ohori, A.: A Polymorphic Record Calculus and its Compilation, *ACM Transactions on Programming Languages and Systems*, Vol. 17,No. 6(1995), pp. 844–895.

[10] Sarkar, D., Waddell, O., and Dybvig, R. K.: A nanopass infrastructure for compiler education, *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, 2004, pp. 201 – 212.

[11] Steele, G. L.: RABBIT: A compiler for SCHEME, Technical Report TR474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1978.