速攻 MinCaml コンパイラ概説

住井 英二郎
sumii@acm.org

2005 年 9 月 26 日

これは何？

この文書は、プログラミング言語 ML のサブセット「MinCaml」のコンパイラについて解説したチュートリアルです。MinCaml コンパイラは、単純かつ強力なプログラミング言語である ML の普及を目標とし、「ML はよく知らないから使いたくない」という社会的慣性の悪循環を打ち破るべく、未踏ソフトウェア創造事業「美しい日本の ML コンパイラ」により開発されました。実装言語としては ML の一種である Objective Caml を使用しています。多少の ML プログラミング経験を前提としていますので、ML そのものについて知りたい方は、まず「超特急: 一時間でわかる ML 超入門」のほうを見ていただければ幸いです。

なお、もしソースコードを読むときは、Emacs というエディタの caml-mode というプラグインを使う等して、色のついたテキストで見ることを強くお勧めします。また、横幅を無理に 80 桁未満とするような努力はしていませんので、120 桁ぐらいのウィンドウでご覧ください。

[2008 年 9 月 17 日更新: お茶の水女子大学の増子さんと浅井先生のご貢献により、SPARC に加え PowerPC もサポートされました。**make** の前に**./to_ppc** か**./to_sparc** のいずれかを実行してください。]

[2012 年 8 月 27 日更新: Pentium 4 以降（SSE2 対応）の x86 をサポートしました。**make** の前に**./to_x86** を実行してください。]

MinCaml コンパイラの Makefile

MinCaml コンパイラをダウンロードしたら、まず **Makefile** を見てください。**Makefile** というのは、MinCaml 自体のコンパイル方法を記述したファイルです。

MinCaml コンパイラの **Makefile** は、**OCamlMakefile** という便利なファイルを利用しています（下のほうの **include OCamlMakefile** という行です）。**SOURCES =** …という行に MinCaml で使うファイルを書いておくだけで、「**make byte-code**」というコマンドを実行すればバイトコード実行形式 **min-caml** が生成されます。同様に、「**make native-code**」でネイティブコード実行形式 **min-caml.opt** ができます。また、「**make top**」を実行すれば、MinCaml のモジュールが最初からロードされた OCaml トップレベル対話環境 **min-caml.top** が生成されます。これはデバッグや実験に便利です。

MinCaml の **Makefile** には自動テストの機能もあります。ディレクトリ **test** の下に「プログラム名**.ml**」という形の ML プログラムをおき、**TESTS =** …という行にプログラム名を書いて、「**make do_test**」を実行してみてください。すると、MinCaml でコンパイルしたアセンブリ（プログラム名**.s**）の実行結果（プログラム名**.res**）と、OCaml で実行した結果（プログラム名**.ans**）とを自動比較し、差分を「プログラム名**.cmp**」に出力します。正常にコンパイル・実行されていれば、違いはないので空になるはずです。

アセンブリを実行するには SPARC 環境のコンパイラ **gcc** やアセンブラ **as** が必要です。ただし、実行しないで生成するだけならば、SPARC 環境ではなくても OK です。

メインルーチン

次に MinCaml のメインルーチン **main.ml** を見てください。末尾の **let () =** …という部分から、コンパイラの実行が開始されます。

まず、OCaml の標準ライブラリ **Arg** を利用して、オプションを処理しています。**-inline** でインライン展開する関数のサイズを、**-iter** で最適化処理のループ回数を指定します（詳しくは後で述べます）。

それ以外のコマンドライン引数は、コンパイルするプログラムの名前と解釈され、関数 **Main.file** により「プログラム名**.ml**」という MinCaml ソースコードから「プログラム名**.s**」という SPARC アセンブリが生成されます。

なお、デバッグや実験のために、引数の文字列をコンパイルして、標準出力に結果を表示す

る **Main.string** という関数も用意されています。

main.ml の中核は関数 **Main.lexbuf** です。引数として受け取ったバッファに対し、順に字句解析（**Lexer.token**）、構文解析（**Parser.exp**）、型推論（**Typing.f**）、K 正規化（**KNormal.f**）、α 変換（**Alpha.f**）、最適化（**iter**）、クロージャ変換（**Closure.f**）、仮想マシンコード生成（**Virtual.f**）、SPARC の 13 bit 即値最適化（**Simm13.f**）、レジスタ割り当て（**RegAlloc.f**）、アセンブリ生成（**Emit.f**）を行います。

最適化関数 **iter** は、β 簡約（**Beta.f**）、ネストした let の簡約（**Assoc.f**）、インライン展開（**Inline.f**）、定数畳み込み（**ConstFold.f**）、不要定義削除（**Elim.f**）の 5 つを、**-iter** で指定した回数を上限として、結果が不変になるまで適用します。

これらの処理の内容については、以下で説明していきます。

MinCaml の構文と型

MinCaml は ML のサブセットです。優先順位や括弧などの細かいことを別にすると、以下のような構文をもっています。

e ::=	式
c	定数
op(e_1, ..., e_n)	プリミティブ演算
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 ... y_n **=** e_1 **in** e_2	再帰関数定義
e e_1 ... e_n	関数呼び出し
(e_1**,** ..., e_n**)**	組の作成
let (x_1**,** ..., x_n**) =** e_1 **in** e_2	組の読み出し
Array.create e_1 e_2	配列の作成
e_1**.(**e_2**)**	配列の読み出し
e_1**.(**e_2**) <-** e_3	配列への書き込み

これを ML のデータ型 **Syntax.t** として表したモジュールが **syntax.ml** です。ただし、**let** や **let rec** のように、新しい変数が定義される式では、その型も（上の構文にはありませんが）含まれています。この型を表す **Type.t** は **type.ml** で定義されています。

T ::=	型
π	プリミティブ型
$T_1 \rightarrow \ldots \rightarrow T_n \rightarrow T$	関数型
$T_1 \times \ldots \times T_n$	組の型
T **array**	配列型
α	型変数

最後の「型変数」は、型推論のところで使用します。

なお MinCaml コンパイラでは、いわゆるカリー化関数の部分適用を自動ではサポートしていません。つまり、関数呼び出しではすべての引数を与えなければいけません。部分適用をしたいときは、たとえば **let rec f_123 x = f 123 x in f_123** のように自分で関数定義をする必要があります（Scheme 言語を知っている人は、それと同じだと思ってください）。

また、命令型プログラミングに必要な参照(reference)もありませんが、これは要素が一個しかないような配列で代用できます。具体的には、**ref e** は **Array.create 1 e**、**!e** は **e.(0)**、e_1 **:=** e_2 は e_1**.(0) <-** e_2 とすれば OK です。

コンパイラの方針

一般にコンパイルとは、ある高水準言語のプログラムを、より低水準な言語に変換することです。たとえば、二つの非負整数の最大公約数を求める MinCaml 関数

```
let rec gcd m n =
  if m = 0 then n else
  if m <= n then gcd m (n - m) else
  gcd n (m - n)
```

は、以下のような SPARC アセンブリにコンパイルされます。

```
gcd.7:
        cmp     %i2, 0
        bne     be_else.18
        nop
        mov     %i3, %i2
        retl
        nop
be_else.18:
        cmp     %i2, %i3
        bg      ble_else.19
        nop
        sub     %i3, %i2, %i3
        b       gcd.7
        nop
ble_else.19:
        sub     %i2, %i3, %o5
        mov     %i3, %i2
        mov     %o5, %i3
        b       gcd.7
        nop
```

これは一見すると、ものすごいギャップです。MinCaml コンパイラでは適切な中間言語を設定し、単純な変換を順番に適用していくことにより、このギャップを一つずつ埋めていきます。MinCaml と SPARC アセンブリの主なギャップは次の 5 つです。

1. 型。MinCaml には型があり、型チェックが行われますが、アセンブリにはそのような仕組みがありません。
2. ネストした式。たとえば MinCaml では `1+2-(3+(4-5))` のように、いくらでも複雑な式を書くことができますが、アセンブリでは一つの命令につき一つの演算しかできません。
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
```

のように関数の中で別の関数を定義できますが、アセンブリではトップレベルの「ラベル」しか使用できません。

4. MinCaml には組や配列などのデータ構造がありますが、アセンブリにはありません。
5. MinCaml ではいくつでも変数を使用できますが、アセンブリには有限のレジスタしかありません。

これらのギャップを埋めるために、MinCaml では

1. 型推論
2. K 正規化
3. クロージャ変換
4. 仮想マシンコード生成
5. レジスタ割り当て

といった処理を順番に行います。以下では、そのような変換や各種最適化について説明していきます。

字句解析(`lexer.mll`)

コンピュータにとっては、ML プログラムといえども、はじめはただの文字列です。たとえばさっきの **`gcd`** だったら、

「**`l`**」「**`e`**」「**`t`**」「 」「**`r`**」「**`e`**」「**`c`**」「 」「**`g`**」「**`c`**」「**`d`**」「 」「**`m`**」「 」「**`n`**」「 」「**`=`**」...

のように見えるわけです。このままでは何もできないので、まず

「**`let`**」「**`rec`**」「**`gcd`**」「**`m`**」「**`n`**」「**`=`**」...

のような**字句**に区切ります。この処理を**字句解析**といいます。

字句解析にはいろいろな方法がありますが、ここでは ocamllex という、まさに OCaml で字句解析をするためのツールを利用します。そのファイルが **`lexer.mll`** です。ocamllex についての詳細はマニュアル（ないし和訳）を参照してもらうことにして、概要だけ説明すると、

```
| '-'? digit+
    { INT(int_of_string (Lexing.lexeme lexbuf)) }
```

といったパターンマッチングのような構文により、「正規表現**'-'? digit+**にマッチしたら字句 **INT** を返す」等のルールを並べて書けば OK です。字句を表すデータ型（**INT** など）は、次に述べる **parser.mly** で定義されています。**Lexing.lexeme lexbuf** という部分は、解析中の文字列を表す「おまじない」だと思ってください。

構文解析(parser.mly)

さて、字句解析が終わると

「**1**」「**2**」「**3**」「**-**」「**4**」「**5**」「**6**」「**+**」「**7**」「**8**」「**9**」

のような文字列のかわりに

「**123**」「**-**」「**456**」「**+**」「**789**」

のような字句の列が得られます。が、このように平らな列のままでは、まだ高度な処理はできません。たとえば「**123-456+789**」だったら **123-(456+789)** ではなく **(123-456)+789** という意味であることを認識しないといけないからです。**syntax.ml** で定義したデータ型 **Syntax.t** で表現すると、

```
Add(Sub(Int 123, Int 456), Int 789)
```

のような**構文木**として解釈する必要があるわけです。このように字句の列を構文木に変換する処理を**構文解析**といいます。MinCaml コンパイラでは、ocamlyacc というツールを利用して、**parser.mly** というファイルで構文解析を実装しています。

parser.mly の中身は **lexer.mll** と類似しており、字句の列から構文木を表すデータ型へのパターンマッチングが並んでいます。たとえば

```
| exp PLUS exp
    { Add($1, $3) }
```

という感じです。**$1** や**$3** というのは、1 番目や 3 番目の構文要素（ここでは両方とも **exp**）という意味です。

構文の定義は、ほとんど先に述べた式 e の通りなのですが、一点だけ注意があります。ML

では式を並べるだけで関数適用になるので、**x - y** と書いたときに、**x** から **y** を引き算しているのか、関数 **x** を引数 **-y** に適用しているのか、曖昧になってしまうのです！　そこで、括弧をつけなくても関数の引数になれる式 **simple_exp** と、一般の式 **exp** を区別しています。たとえば **-y** は **simple_exp** ではないので、先の例は関数適用ではなく引き算であるとわかるわけです。

また、いろいろな構文や二引数演算子の優先順位、（**lexer.mll** で出てきた）字句を表すデータ型も **parser.mly** で定義されています。

なお、変数の型が必要なところ（**let** など）は、とりあえず未定義の新しい型変数 **Var(ref None)** で埋めています。これについては次の型推論で述べます。

型推論(typing.ml)

普通の ML と同様に、MinCaml では変数や関数の型を書かなくても自動で推論してくれます。これを**型推論**といい、多相関数や高階関数のあるプログラムでは非常に便利です。

MinCaml の型推論の本体は関数 **Typing.g** です。この関数は、**型環境**（変数の名前から、その型への写像）**env** と、式 **e** とを受け取り、**e** の型を推論して返します。また、式の中に出てくる変数の型が合っているかどうかも調べます。もし未定義の型変数があったら、適切な型を代入します。このような代入により型を合わせる関数が **Typing.unify** です。

たとえば、足し算 **e1 + e2**（**Syntax.t** に合わせて書くと **Add(e1, e2)** ）だったら、まず **g env e1** により部分式 **e1** の型を推論し、その型が **int** であることを **unify** により確かめます。部分式 **e2** についても同様です。そして、式全体の型として **int** を返します。

少しややこしい例としては、関数適用 **e e1 ... en** があります。**e** が関数で、**e1** から **en** までが引数です。この場合は、関数の型は **g env e**、引数の型は **g env e1**, ..., **g env en** のように推論できますが、返値の型がわかりません。そこで、未定義の新しい型変数 **t** を作り、**g env e** が「**g env e1**, ..., **g env en** を受け取って **t** を返す」関数型と等しくなるように **unify** を呼んでいます。そして、式全体の型として **t** を返します。

let や **let rec** のように新しい変数が導入されるところでは、型環境 **env** が拡張されま

す。逆に変数 **x** が出てきたら、型環境 **env** を引いて型を得ます。ただし、型環境に含まれていない変数が出てきたら、プログラムの外から与えられる**外部変数**とみなし、未定義の新しい型変数を与え、外部変数のための特別な環境 **extenv** に追加します。これは普通の ML では実現されてない、MinCaml に特有の機能です。これにより、宣言しなくても外部変数を使用することができます。

関数 **Typing.unify** は、与えられた二つの型が等しいかどうかを中まで調べていき、一方が未定義の型変数 **Type.var(ref None)** だったら、他方と等しくなるように代入を行います。ただし一方の型変数が、他方の型の中に現れていないかどうかチェックします（occur check と呼ばれます）。もし現れていると、代入結果の型にサイクルができてしまうからです。たとえば、もし（occur check をしないで）型変数 α と関数型 int→α を unify してしまうと、α = int→α なので、int→int→int→...という無限長の型になってしまいます！　このようなことを防止するために(ちょっとややこしいですが)occur check は必要なのです。

型推論が終了したら（エラーでも正常終了でも)、結果をわかりやすくするために、すべての型変数をその中身でおきかえます（**Typing.deref_typ**)。もしまだ未定義の型変数があったら、型が決まらないということなので、勝手に **int** としてしまいます。これも MinCaml に特有の仕様です。

K 正規化(kNormal.ml)

「コンパイルとは高水準言語と低水準言語のギャップを埋めることである」と述べましたが、そのようなギャップの一つに「ネストした式」がありました。たとえば ML や大概の言語では **a + b + c - d** のような式の値を一発で計算することができますが、普通のアセンブリにそのような命令はありません。

このギャップを埋めるために、計算の途中結果もすべて変数として定義するのが **K 正規化**という変換です(ちなみにK正規化という名前は、ML Kit というコンパイラに由来します)。たとえば、さっきの例は

```
let tmp1 = a + b in
let tmp2 = tmp1 + c in
tmp2 - d
```

という風に変換できます。

MinCaml では、**KNormal.g** が K 正規化の実装本体、**KNormal.t** が K 正規化した後の式を表すデータ型です。**KNormal.g** は変数の型環境 **env** と K 正規化前の式とを受け取り、K 正規化後の式と、その型とを組にして返します（型環境を受け取ったり型を返したりするのは、**let** で変数を定義するときに型も付加する必要があるためで、あまり K 正規化の本質とは関係がありません）。

たとえば **e1 + e2** という式だったら、まず **g env e1** のように **e1** を K 正規化し、その結果を **let** により変数 **x** として定義します。それから **g env e2** のように **e2** も K 正規化し、その結果を **let** により変数 **y** として定義し、**x + y** という式を（その型 **int** とともに）返します。

このような **let** を挿入するために、**insert_let** という補助関数を使用しています。**insert_let** は、式 **e** を受け取り、新しい変数 **x** を作って、**let x = e in** ...という式を返します（ただし **e** が最初から変数のときは、それを **x** として利用し、**let** は挿入しません）。**in** の中を作るための関数 **k** も引数として受け取り、**k** を **x** に適用した結果を...の部分として利用します。

なお、**KNormal.g** では、K 正規化のついでに、論理値 **true**, **false** を整数 **1**, **0** に変換する処理もやっています。また、比較や条件分岐も、**e1 <= e2** や **if e then e1 else e2** ではなく、**if e1 <= e2 then e3 else e4** のように比較と分岐が一体となった特殊な形に直します。通常のアセンブリでは比較と分岐が一体となっているので、そのギャップを早めに埋めてしまうためです。そのために、もし条件 **e** が比較になっていなかったら、**if e <> 0 then e1 else e2** のように、条件の部分を比較に変換します。また、**if (not e) then e1 else e2** を **if e then e2 else e1** のように変換していくことにより、最後は

```
if e1 = e2 then e3 else e4
```

あるいは

```
if e1 <= e2 then e3 else e4
```

という二つの形に帰着できます。これらは K 正規化とは無関係ですが、もし別々のモジュールにしてしまうと途中のデータ型を定義せねばならず面倒なので、あくまで「ついで」と

して **KNormal.g** で一緒に実装されています。

また、外部変数の使用は、外部関数の呼び出し（**KNormal.ExtFunApp**）か、外部配列の参照（**KNormal.ExtArray**）のどちらかに限ることとします。大抵のアプリケーションでは、この 2 つで必要十分だからです。これも K 正規化のついでに実現されています。

α変換(alpha.ml)

さて、K 正規化が済んだら最適化を行うのですが、その前に「異なる変数には異なる名前をつける」**α変換**を行います。違う変数に同じ名前がついていると、いろいろと処理が面倒になるためです。たとえば **let x = 123 in let x = 456 in x + x** という式だったら、**let x1 = 123 in let x2 = 456 in x2 + x2** と直してしまいます。

これを実装した本体が **Alpha.g** です。変換前の変数名から変換後の変数名への写像 **env** と、変換前の式 **e** とを受け取り、変換後の式を返します。たとえば **let x = e1 in e2** という式があったら、まず **e1** を変換してから、新しい変数 **x'** を作り、**x** から **x'** への対応を **env** に追加して、**e2** を変換するという具合です。**let rec** や **LetTuple** の場合も、一見するとややこしそうですが、やっていることは一緒です。

なお、外部変数の名前は **env** に登録されないので、α変換の対象になりません（関数 **Alpha.find** 参照）。これは意図された動作です。もし外部変数の名前が変わってしまったら正しくリンクできなくなるからです。

β簡約(beta.ml)

いきなり例ですが、たとえば **let x = y in x + y** のような式があったら、**x** と **y** が等しいことは明らかですから、**x + y** を **y + y** と置き換えてしまうことができます。このような変換を K 正規形の**β簡約**といいます（λ計算という理論でもβ簡約という言葉がありますが、それの特殊な場合になっています）。普通のプログラムだと、あまり必要な処理でもないのですが、他の最適化等をした後のプログラムでは効果がある場合もあります。

MinCaml では、関数 **Beta.g** がβ簡約の処理をしている本体です。ある変数からそれに等しい変数への写像 **env** と、式 **e** とを受け取り、**e** をβ簡約した式を返します。やはりポイ

ントは **let x = e1 in e2** の場合で、**e1** を β 簡約した結果が変数 **y** だったら、**x** から **y** への対応を **env** に追加して、**e2** を β 簡約します。そして、変数 **x** が出てきたら、**env** を引いて、変数 **y** に置き換えてしまいます。ただし K 正規形では、ありとあらゆる式に変数が出てくるので、この置換のための関数 find を定義・使用しています。

ネストした let の簡約(assoc.ml)

次に、式の形を見やすくするために、**let x = (let y = e1 in e2) in e3** のようにネストした **let** を、**let y = e1 in let x = e2 in e3** のように平たくします。これは MinCaml でコンパイルされたプログラムの性能に（直接は）影響しませんが、コンパイラのデバッグや実験のときに、人間にとってわかりやすくするためです。

この簡約は **Assoc.f** で実装されています。**let x = e1 in e2** という形の式があったら、まず再帰により **e1** を **e1'**に、**e2** を **e2'**に簡約します。そして、**e1'**が **let** ... **in e** のようになっていたら、最後の **in** の直後に **let x = e in e2'**を挿入して、**let** ... **in let x = e in e2'**という式を返します。ちょっとトリッキーですが、できてしまえば非常に簡単です（**assoc.ml** はわずか 21 行です）。

インライン展開(inline.ml)

次は、最適化処理の中でもっとも効果のある**インライン展開**です。これは、小さな関数の呼び出しを、その本体で置き換えてしまう、という変換です。MinCaml では関数 **Inline.g** により実装されています。

まず、関数定義 **let rec f x1 ... xn = e in ...**があったら、関数 **f** の本体 **e** のサイズを、**Inline.size** によって計算します。もしそのサイズが整数参照 **Inline.threshold** 以下だったら、関数名 **f** から仮引数 **x1**, ..., **xn** および本体 **e** への対応を、写像 **env** に追加します。そして、関数呼び出し **f y1 ... yn** があったら、さっきの仮引数 **x1**, ..., **xn** および本体 **e** を写像 **env** から引いてきて、本体 **e** の中の仮引数 **x1**, ..., **xn** を実引数 **y1**, ..., **yn** で置き換えた式を返します。

ただし、インライン展開した式は関数の本体を複製した式ですから、変数が重複しているかもしれないので、また α 変換する必要があります。偶然か必然かわかりませんが、**Alpha.g**

を流用すれば、上述の「仮引数を実引数で置き換える」処理とα変換は同時に実現できてしまいます。**Alpha.g** の写像 **env** として空の写像を用いるかわりに、**x1**, ..., **xn** を **y1**, ..., **yn** に対応させるだけで OK です。

定数畳み込み(constFold.ml)

関数をインライン展開すると、たとえば **let x = 3 in let y = 7 in x + y** における **x + y** のように、すでに値がわかっている変数についての計算がよく出てきます。これを実際に計算して、コンパイル時に「**10**」のような定数に置き換えてしまう最適化処理が**定数畳み込み**です。MinCaml では関数 **ConstFold.g** にて実装されています。

ConstFold.g は、変数の名前から値への写像 **env** と、式 **e** とを受け取り、定数畳み込みを行って返します。たとえば **x + y** という式を受け取ったら、**x** と **y** の値が整数定数かどうか調べ、もしそうだったら直ちに計算して結果を返します。逆に **let x = e in ...** という変数定義を見つけたら、**x** から **e** への対応を **env** に追加します。整数だけでなく、浮動小数や組についても同じです。

不要定義削除(elim.ml)

定数畳み込みを行うと、**let x = 3 in let y = 7 in 10** における **x** や **y** のように、使われない変数定義や関数定義が出てきます。MinCaml では、これを **Elim.f** で取り除きます。

一般に、**e1** に副作用がなく、**x** が **e2** に出現していなければ、**let x = e1 in e2** という式を単なる **e2** に変換することができます。この「副作用がない」という条件を実装したのが **Elim.effect**、「変数が式に出現する」という条件を実装したのが **KNormal.fv** です。ただし、副作用が本当にあるかどうかは決定不能なので、「配列への書き込みか、関数呼び出しがあったら副作用がある」と判定しています。

ちなみに **KNormal.fv** という名前は**自由変数**(free variable)という用語に由来します。たとえば **let x = 3 in x + y** という式には二つの変数 **x**, **y** がありますが、**x** は式の中で整数 **3** に定義（束縛）されているので**束縛変数**といわれ、**y** は束縛されていないので自由変数といいます。

クロージャ変換(closure.ml)

まだ MinCaml とアセンブリの間に残っているギャップとして「ネストした関数定義」があります。これを平らにするのが**クロージャ変換**で、関数型言語のコンパイラではもっとも重要な処理の一つです。

「ネストした関数定義を平らにする」といっても、やさしいケースと難しいケースがあります。たとえば

```
let rec quad x =
  let rec dbl x = x + x in
  dbl (dbl x) in
quad 123
```

だったら

```
let rec dbl x = x + x in
let rec quad x = dbl (dbl x) in
quad 123
```

のように定義を移動するだけで OK です。しかし

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

に対して同じことをしたら

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

のようにナンセンスなプログラムになってしまいます。これは、関数 **dbl** には自由変数がないのに対し、関数 **adder** には自由変数 **x** があるためです。

このように自由変数のある関数定義を平らにするためには、**adder** のような関数の本体だけでなく、**x** のような自由変数の値も組にして扱わなければいけません。ML のコードとして書くならば、

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

のような感じです。まず、関数 **adder** は、自由変数 **x** の値も引数として受け取るようにします。そして、関数 **adder** を値として扱うときは、**(adder, x)** のように、関数の本体と自由変数の値の組として扱います。この組のことを関数の**クロージャ**といいます。さらに、関数 **adder** を呼び出すときは、クロージャから関数の本体 **f** と自由変数の値 **fv** を読み出し、引数として与えます。

クロージャ変換でややこしいのは、どの関数はクロージャとして扱い、どの関数は普通に呼び出せるのか、区別する方法です。もっとも単純な方法は「すべての関数をクロージャとして扱う」ことですが、あまりにも効率が悪くなってしまいます。

そこで、MinCaml のクロージャ変換 **Closure.g** では、「自由変数がないとわかっていて、普通に呼び出せる」関数の集合 **known** を引数として受け取り、与えられた式をクロージャ変換して返します（**known** と式の他に、型環境 **env** も受け取ります）。

クロージャ変換の結果はデータ型 **Closure.t** で表現されます。K 正規形の **KNormal.t** とほぼ同様ですが、ネストしうる関数定義のかわりに、クロージャ生成 **MakeCls** とトップレベル関数の集合 **toplevel** があります。また、一般の関数呼び出しのかわりに、クロージャによる関数呼び出し **AppCls** と、クロージャによらないトップレベル関数の呼び出し **AppDir** があります。さらに、今後のために変数の名前のデータ型 **Id.t** と、トップレベル関数の名前（ラベル）のデータ型 **Id.l** を区別します。**AppCls** は変数を、**AppDir** はラベ

ルを使用していることに注意してください。クロージャは **MakeCls** で変数に束縛されますが、トップレベル関数はラベルで呼び出されるためです。

Closure.g は、一般の関数呼び出し **x y1 … yn** があったら、関数 **x** が集合 **known** に属してるか調べ、もし属していたら **AppDir** を、属していなければ **AppCls** を返します。

let rec x y1 … yn = e1 in e2 のような関数定義があったら、以下の処理をします。まず、**x** には自由変数がないと仮定して、集合 **known** に追加し、本体 **e1** をクロージャ変換してみます。そして、もし本当に自由変数がなかったら、処理を続行し **e2** をクロージャ変換します。もし自由変数があったら、集合 **known** や参照 **toplevel** を元に戻して、**e1** のクロージャ変換からやり直します。最後に、**e2** に **x** が（ラベルではなく）変数として出現していなければ、クロージャの生成 **MakeCls** を削除します。

最後の部分はちょっとわかりにくいかもしれません。たとえ **x** に自由変数がなかったとしても、**let rec x y1 … yn = … in x** のように値として返すときは、クロージャを生成する必要があります。というのは、**x** を値として受け取る側では、自由変数があるかどうか一般にわからないので、**AppDir** ではなく **AppCls** を使用して、クロージャによる関数呼び出しを行うからです。このような場合は、**x** が変数として **e2** に出現するので、**MakeCls** は削除されません。一方で、**let rec x y = … in x 123** のように関数呼び出しをするだけであれば、**x** は（変数ではなく）**AppDir** のラベルとして出現するだけなので、**MakeCls** は削除されます。

仮想マシンコード生成(virtual.ml)

クロージャ変換が完了したら、いよいよ SPARC アセンブリを生成します。しかし、いきなり本当の SPARC アセンブリを生成するのは大変なので、まず SPARC アセンブリに類似した、仮想マシンコードを生成します。どう「仮想」なのかというと、

- 変数（レジスタ）が無限にある
- ブランチやジャンプのかわりに if-then-else や関数呼び出しがある

の 2 点が主です。

このような仮想アセンブリを定義したのが **sparcAsm.ml** です。**SparcAsm.exp** は、ほぼ SPARC の命令に対応しています（**if** 以外）。命令列 **SparcAsm.t** は、関数の最後で値を

返す **Ans** と、変数代入 **Let** から成り立ちます。**Forget**, **Save**, **Restore** については後で述べます。

クロージャ変換の結果を仮想マシンコードに変換する関数は **Virtual.f**, **Virtual.h**, **Virtual.g** の 3 つです。**Virtual.f** はプログラム全体（トップレベル関数のリストと、メインルーチンの式）を、**Virtual.h** は個々のトップレベル関数を、**Virtual.g** は式を変換します。変換の主要な部分は、クロージャや組・配列の生成や読み出し・書き込みにともなうメモリアクセスを明示することです。クロージャや組・配列などのデータ構造は**ヒープ**というメモリ領域に確保します。ヒープのアドレスは専用のレジスタ **SparcAsm.reg_hp** で記憶することとします。

たとえば、配列の読み出し **Closure.Get** だったら、要素のサイズに応じてオフセットをシフトし、ロードを行います。組の生成 **Closure.Tuple** であれば、浮動小数のアラインメント（8 バイト）に注意しつつ、個々の要素を順番にストアしていき、先頭のアドレスを組そのものの値とします。クロージャの生成 **Closure.MakeCls** も、アラインメントに注意しつつ、関数本体のアドレス（ラベル）と自由変数の値をストアしていき、先頭のアドレスをクロージャそのものの値とします。これに対応して、トップレベル関数の冒頭では、クロージャから自由変数の値をロードします。ただし、クロージャによる関数呼び出し（**AppCls**）を行う際は、そのクロージャのアドレスを必ずレジスタ **SparcAsm.reg_cl** で与えることとします。

なお、SPARC アセンブリでは浮動小数を即値として記述できないので、メモリに定数テーブルを作成する必要があります。そのために**Virtual.g** でグローバル変数**Virtual.data** に浮動小数定数を記録していき、**Virtual.f** でプログラム全体 **SparcAsm.Prog** の一部とします。

13 bit 即値最適化(**simm13.ml**)

SPARC アセンブリでは、ほとんどの整数演算の第二オペランドとして、レジスタだけでなく 13 bit 以下（**-4096** 以上 **4096** 未満）の整数即値をとることができます。そのための最適化処理を **Simm13.g** および **Simm13.g'** で実装しています。対象が仮想 SPARC アセンブリであり、定数が 13 bit 整数に制限されていること以外は、定数畳み込みや不要定義削除とほぼ同様です。

レジスタ割り当て(regAlloc.ml)

[2008 年 9 月 17 日更新: 現在のバージョンのレジスタ割り当てでは、以下の「さかのぼって **Save** を挿入する」処理（**ToSpill** や **NoSpill** など）が省略されています。実装がより単純で、レジスタ割り当て自体が高速になり、コンパイルされたプログラムの性能はほとんど変化しないためです。]

MinCaml コンパイラでもっとも複雑な処理が、無限個の変数を有限個のレジスタで実現する**レジスタ割り当て**です。

まず**関数呼び出し規約**として、引数は番号の小さいレジスタから順に割り当てていくことにします（レジスタで渡しきれないほど数の多い引数は、プログラマが組などで置き換えることとし、MinCaml コンパイラではサポートしません）。返値は第一レジスタにセットすることにします。これらは関数のレジスタ割り当て **RegAlloc.h** で処理されます。

その上で、関数の本体やメインルーチンについてレジスタ割り当てをしていきます。**RegAlloc.g** は、今のレジスタ割り当てを表す（変数からレジスタへの）写像 **regenv** と、命令列とを受け取り、レジスタ割り当てを行って返します。レジスタ割り当ての基本方針は、「まだ生きている（これから使われる）変数が割り当てられているレジスタは避ける」ことです。「まだ生きている変数」は **SparcAsm.fv** により計算されます。ただし、**Let(x, e1, e2)** の **e1** をレジスタ割り当てする際には、現在の式 **e1** だけでなく、その「後」にくる命令列 **e2** も「これから使われる変数」の計算に含めなければいけません。そのために **RegAlloc.g** や、式をレジスタ割り当てする関数 **RegAlloc.g'** では、「これから後」にくる命令列 **cont** も引数として受け取り、生きている変数の計算に用いています。

しかし、変数は無限にあってもレジスタは有限なので、どうしても生きているレジスタしか割り当てられないこともあります。その場合は、現在のレジスタの値をメモリに退避する必要があります。これをレジスタ溢れ(register spilling)といいます。命令型言語と違って、関数型言語では変数の値が後から変わることはないので、どうせ退避するなら**できるだけ早く**行うのが得策です。それだけ早くレジスタに空きができるからです。

そこで **RegAlloc.g** では、変数 **x** の退避が必要になったら、そのことを表すデータ型

ToSpill を返し、**x** の定義まで**さかのぼって**退避命令 **Save** を挿入します。また、退避が必要になった時点で **x** は「生きている変数」から除外したいので、その部分に **Forget** という仮想命令を挿入して自由変数の集合から除外します。そのために **ToSpill** は退避すべき変数（のリスト）だけでなく、**Forget** 挿入後の命令列 **e** も保持しています。**x** を **Save** した後は **e** に対してレジスタ割り当てをやり直すわけです。

退避が必要になるのは、レジスタが不足した場合だけではなく、関数呼び出しもあります。MinCaml ではいわゆる caller save の慣習を採用しており、関数を呼び出したらレジスタの値は失われます。したがって、その時点で生きている変数の値は、すべて退避する必要があります。**ToSpill** が一個ではなく複数の変数を（リストとして）保持しているのは、このためです。

なお、もし退避が必要にならなかった場合は、レジスタ割り当てされた命令列 **e'** と、新しい **regenv** をデータ型 **NoSpill** として返します。

退避された変数は、いずれまた使用されます。しかし **regenv** を引いてもレジスタが見つからないので、**RegAlloc.g'**（式をレジスタ割り当てする関数）で例外が発生します。この例外は関数 **RegAlloc.g' and unspill** により処理され、メモリからレジスタへ変数を復帰する仮想命令 **Restore** が挿入されます。

レジスタを割り当てるときは、単に「生きているレジスタを避ける」だけでなく、「将来の無駄な **mov** を削減する」努力もしています（register targeting ないし register coalescing といいます）。たとえば、定義された変数が関数呼び出しの第二引数になるのであれば、できるだけ第二レジスタに割り当てるようにします。また、関数の返値となる変数は、できるだけ第一レジスタに割り当てるようにします。これらを実装したのが **RegAlloc.target** です。この処理のために **RegAlloc.g** や **RegAlloc.g'** は、計算結果（関数の返値）を格納するレジスタ **dest** も引数として要求・使用しています。

アセンブリ生成(emit.ml)

ついに最終段階のアセンブリ生成となりました。もっともややこしいレジスタ割り当てがすでに終わっているので、特に難しいことはなく、**SparcAsm.t** を本当の SPARC アセンブリとして出力するだけです。

ただし、仮想命令はちゃんと実装する必要があります。条件分岐は比較とブランチで実装します。**`Save`** と **`Restore`** は、すでにセーブした変数の集合 **`stackset`** と、変数のスタックにおける位置のリスト **`stackmap`** とを計算しつつ、ストアとロードで実装します。関数呼び出しでは、レジスタの順に引数を並べ替える処理 **`Emit.shuffle`** が少し面倒ですが、それ以外は簡単です。

一点だけ自明でない重要な部分として、**末尾呼び出し最適化**があります。これは「もう後にすることがなく、戻ってこない」ような関数呼び出し（末尾呼び出し）を、Call ではなく、ただのジャンプ命令で実装するという処理です。これにより、たとえば冒頭の gcd のような再帰関数を、ただのループとまったく同様に実装することができます。このために、命令列のアセンブリ生成をする関数 **`Emit.g`** や、各命令のアセンブリ生成をする関数 **`Emit.g'`** では、末尾かどうかを表すデータ型 **`Emit.dest`** も引数として受け取ります。末尾 **`Tail`** だったら、他の関数をジャンプ命令で末尾呼び出しするか、さもなくば計算結果を第一レジスタにセットし、SPARC の **`ret`** 命令で関数を終了します。末尾でない **`NonTail(x)`** だったら、計算結果を **`x`** にセットします。

最後に、ヒープと関数呼び出しスタックを確保するメインルーチン **`stub.c`** と、テストプログラムの実行に必要な外部関数 **`libmincaml.S`** を用意すれば、MinCaml コンパイラの完成です！