修 士 論 文

正格な関数型言語における
遅延評価を用いたループ不変式削除最適化

北陸先端科学技術大学院大学
情報科学研究科情報処理学専攻

奥谷 謙一

2006 年 3 月

修 士 論 文

正格な関数型言語における
遅延評価を用いたループ不変式削除最適化

指導教官　大堀 淳 教授

審査委員主査　大堀 淳 教授
審査委員　日比野 靖 教授
審査委員　小川 瑞史 教授

北陸先端科学技術大学院大学
情報科学研究科情報処理学専攻

410027 奥谷 謙一

提出年月: 2006 年 2 月

概 要

ループ不変式削除は，手続き型言語では多くのコンパイラにおいて実装されている最適化の１つである．関数型言語ではループは再帰関数で表される．また正格な言語では評価順序も厳密に定義されている．本論文では、遅延評価を用いることで評価順序を変えないループ不変式削除を，正格な関数型言語において実装しその効果について調査する．

目 次

第1章　はじめに

1.1　背景

近年のソフトウェアの複雑化により，バグのない信頼性の高いソフトウェアの開発手法が求められている．このようなソフトウェア開発において，プログラミング言語の選択は最も重要な点である．

近代関数型言語の代表である Standard-ML[1] では，静的型システムをもとに，従来発生していた実行時エラーをコンパイル時に検出することを可能とし，より安全なプログラミングを実現している．また，多相型，型推論，高階関数などの機能は，柔軟かつ簡潔なプログラミングを可能としている．

このように，関数型言語には多くの有効な特徴を持つ言語であるが，実行速度に関しては，実行時環境の必要性などにより，C 言語などの手続き型言語と比較すると，十分な水準には達してはいない．この欠点の改善には，実行時環境の改善など様々な方法があるが，手続き型言語で用いられる最適化を実現することは有効な改善方法の 1 つである．

このような最適化は数多くの種類があるが，ループ不変式削除を本研究では扱うこととした．これは，ループ内で値の変化のない計算を，ループ外で変数に束縛し，その変数でループ内の計算を置き換える変換である．これを関数型言語において実装することにより，手続き型言語のように実行速度の向上が期待できると考えられる．

1.2　最適化

現在実用的なコンパイラは多くの場合，ある種の最適化処理を備えている．最適化は処理内容を変えず，より効率的な実行を可能にすることである．

最適化には主に 2 つの目的がある．

- メモリの使用量を抑える．
- 実行速度を速くする．

プログラム実行時のメモリ使用量は一般的にコードの大きさに依存する．そのためコンパイル後のコード量を減らすことでメモリの使用量を抑制することは，効率的な実行につながる．しかし近年ではメモリ量は十分なことが多いのでそれほど重視されなくなっている．

また，JAVA などの抽象機械コードを目的コードとする言語ではコードを実行するために，抽象機械が必要である．このようなものでは，抽象機械を改善にするといった，動的な最適化を行う場合もある．

それとは逆に，実行コードを実行前にあらかじめ解析し，変換を施すコード最適化を本研究では扱う．コード最適化は，局所的最適化，大域的最適化の 2 種類に分けられる．

局所的最適化には，コストがかかる演算を，等価でコストのかからない演算に置き換える（強さの軽減），や定数式をコンパイル時に評価する（定数畳み込み）などがある．

大域的最適化は，ループなどプログラムの流れを解析することで可能になる最適化である．プログラム実行時間の多くはループに費やされる傾向にあるので，ループに対する最適化は特に有効である．ループの最適化は，コード移動，誘導変数の削除，強さの軽減が重要なものとしてあげられる．

このなかでコード移動は，ループ不変式削除とも呼ばれ，ループ中のコード量を減らす点で重要である．この変換は，ループの実行回数とは独立に，常に同じ値を返す式（ループ不変式）を見つけ出し，それをループの前に移す．例として，次のプログラムを考えると，

```
while (i <= limit -2) {
  処理;  /* ループ中では limit の値が変わらないとする */
}
```

ここで，$limit - 2$ はループ不変式である．その部分をループ外に移動することで，次のように最適化される．

```
t = limit -2;
while (i <= t){
  処理;
}
```

本研究では，このループ不変式削除を対象とする．

1.3 目的

本研究の目的は，関数型言語でループ不変式削除を行うアルゴリズムの構築，その評価である．しかし，ML をはじめとする関数型言語では，ループは再帰関数で表され，またパターンマッチなどを利用した分岐が多用されるため，単純な変換では適用できない．この問題を解決し，単純な変形でループ不変式削除を行うことができるアルゴリズムを構築する．

具体的には，以下について述べる．

- 遅延評価を利用したループ不変式削除遅延評価を利用することで，ループ外で式を束縛する再に式を評価せずに束縛する．非正格関数型言語においてはこのような最適化も考えられている [4]．この遅延評価を正格関数型言語に導入することで，評価順序に影響を与えないループ不変式削除アルゴリズムを構築する．
- 提案するアルゴリズムの実装上記のアルゴリズムを実装してその実用性を評価する．実装対象は Standard ML の拡張コンパイラである，大堀らが開発中の SML♯[7] とした．

1.4 構成

本論文の構成を述べる．2 章では手続き型言語におけるループ不変式削除について説明する．3 章では，関数型言語における一般的な最適化について説明する．4 章では，今回実装した遅延評価を利用したループ不変式削除のアルゴリズムについて説明する．5 章では，4 章で説明したアルゴリズムの実装，評価について説明する．6 章では，まとめと，今後の課題について説明する．

第2章　手続き型言語でのループ不変式削除

まず手続き型言語でのループ不変式削除について簡単に述べる.

2.1　ループ不変式削除の例

ループ不変式削除とは，ループ内で値の変化のない計算であるループ不変式を，計算の意味を変化させないようにループ外に移動することである.

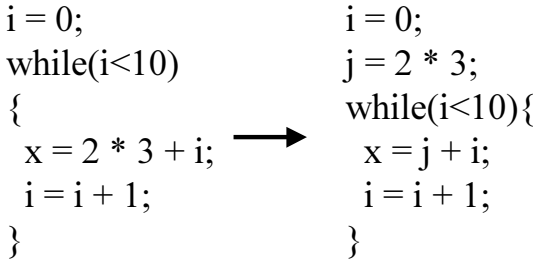

図 2.1: 手続き型言語でのループ不変式削除

図 2.1 は手続き型言語におけるループ不変式削除前後のそれぞれのコードである．ループ内に現れるループ不変式 (2*3) をループ外に移動している.

このような変形を行うことで計算量を抑制することができる.

2.2　ループ不変式の検出

以下 AHO によって説明されているループ不変式の検出法について簡単に書く.

- 定数，またはすべての定義がループの外側から到達する変数だけで構成される文に「不変」の印をつける.
- 「不変」の印の付いた新しい文が出尽くすまで次を繰り返す.

- 「不変」の印の付いていない文について，その変数がすべて，定数，ループの外側からの到達定義をもつもの，あるいは到達定義が 1 つだけで，その定義にはループの中ですでに「不変」の印が付いているもののどれかであれば，その文に「不変」の印をつける．

定義がループの外側から到達する変数とは，変数がループ内で代入操作が行われていないものである．

2.3 ループ不変式削除の定義

ループ不変式として検出されたものは条件を満たせばループの前に移動することが可能である．

その条件を説明する．$x = y + z$ がループ不変であるとしたとき，

- x への代入がループ内に無い．
- 式の定義がすべてのループの出口に対して到達している．

この 2 つの条件を満たすループ不変式はループの前に移動し，ループ内の計算を減らすことができる．

第3章　関数型言語でのループ不変式削除

関数型言語で行われているループ不変式削除と，適用が困難と考えられる例を次に述べる．

3.1　関数型言語の例

関数型言語ではループは再帰関数として表される．

```
fun f (0,y) = 1
  | f (x,y) = f (x-1,a+b+y)
```

このような関数の場合 $a+b$ はループ内で変化せず，副作用を持たないので，

```
fun f (x,y) =
  if x = 0 then 1
  else let val tmp = a + b
           fun f' (0,y) = 1
             | f' (x,y) = f' (x-1,tmp+y)
       in f' (x,y)
       end
```

関数外で $a+b$ を一時変数 tmp に束縛し，それを用いた関数 f' を定義することでループ不変式削除を行うことができる．この例では $x=0$ の場合には a+b が実行されることはないので，その場合についてはあらかじめ判定し無駄な演算を行わないように配慮されている．

このような単純な例では関数型言語でも問題なく手続き型と同様にループ不変式削除を行うことができる．

3.2　ループ不変式削除が困難な例

次の関数のなかに含まれるループ不変式（ここではIV{1,2,3}）を関数定義の外に出す最適化を考える．

```
fun f x =
  case x of
    D1 y => ... IV1 ... f y
    D2 y => ... IV2 ... f y
    D3 y => ... IV3
```

副作用を考えなければ以下の変形が可能であるが, 引数にD2が含まれない場合 ($D1(D1(D3))$ のような場合）に, IV2が無駄に実行されてしまう．また実際には算術式のような基本的な演算にも例外 (副作用) が起こりうるためにこのような変形は正しくない.

```
fun f x = let
    val tmp1 = IV1
    val tmp2 = IV2
    val tmp3 = IV3
    fun f' x =
      case x of
      D1 y => ... tmp1 ... f y
      D2 y => ... tmp2 ... f y
      D3 y => ... tmp3
  in f' x end
```

また, 以下のような変形では無駄な演算は行われないが, ソース量が爆発的に増えてしまう.

```
fun f x =
  case x of
    D1 y => ... IV1 ... let
      val tmp1 = IV1
      fun f'' x =
        case x of
          D1 y => ... tmp1 ... f'' y
          D2 y => ... IV2 ... let
            val tmp2 = IV2
            fun f''' x =
              case x of
                D1 y => ... tmp1 ... f''' y
                D2 y => ... tmp2 ... f''' y
                D3 y => ... IV3 ... let
                    ...
```

そこでIV1,IV2,IV3の実行を実際にその値が必要とされるまで遅延させることでこの問題を解決する．ここで，遅延評価を行うための関数lazyとforceを導入する．

lazyは$(unit \rightarrow' a) \rightarrow' a\ lazy$の型を持つlazyな値の生成関数である．評価を遅延させたい式expがあるときに，$val\ delayedexp = lazy(fn() => exp)$とすることで，$delayedexp$にはexpの評価を遅延した値が入ることになる．

forceは$'a\ lazy \rightarrow' a$の型を持つlazyな値の評価関数とする．ある式expの評価を遅延させた式$delayedExp$があるとき，$force(delayedExp)$とされたときに，expを評価する．一度目のforceでの式expを評価する際，その値は保存され，2度目の$force(delayedExp)$の際には保存された値を返すものとする．

そして上記の関数を用い，以下のように変形することで意味の変わらない最適化を行う．

```
fun f x =
  let
    val tmp1 = lazy(fn () => IV1)
    val tmp2 = lazy(fn () => IV2)
    val tmp3 = lazy(fn () => IV3)
    fun f' x =
      case x of
      D1 y => ... force(tmp1) ... f y
      D2 y => ... force(tmp2) ... f y
      D3 y => ... force(tmp3)
  in f' x
  end
```

第4章　遅延評価を利用したループ不変式削除

本論文では，正格な関数型言語でのループ不変式削除を単純な変形で行うために，遅延評価を利用することを提案する．提案するアルゴリズムでは，ループ内で条件分岐が深くネストしている場合でも，単純な変形により評価順序を損なわないループ不変式削除を行うことができる．

4.1　遅延評価

4.1.1　遅延評価の説明

遅延評価について説明する．

関数型言語は評価順序について，正格と非正格にわけることができる．Standard ML[1] などの正格な言語では，引数がまず評価され，その値が束縛され関数が実行される．これを call-by-value という (図 4.1)．

Haskell[5] など非正格な言語では，引数の評価を遅延し，関数には引数へのポインタが渡される．そして必要になったときに評価される，これを call-by-need という (図 4.2)．

これを用い，ループ不変式をループ外で束縛するときに，実際に値が使われるまでその評価を遅延することで，無駄な演算を抑制し，効率的な変換が可能だと考えた．

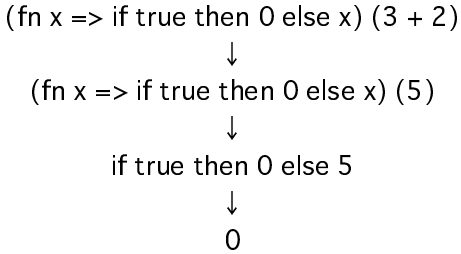

図 4.1: call-by-value での引数の束縛

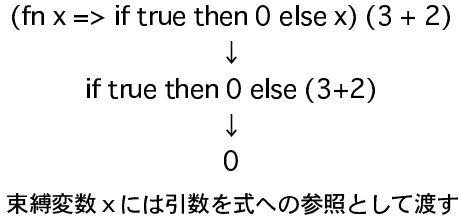

図 4.2: call-by-need での引数の束縛

4.1.2 遅延評価を利用する利点，欠点

遅延評価を導入することにより，式の評価を一時変数に束縛する時点ではなく，もとの評価位置で評価することができるようになる．そのため，副作用のないループ不変式を自由にループ外へ出すことができる．しかし正格な関数型言語に遅延評価を導入するためには，遅延した式がまだ評価されていないのか，または評価されているのかを判断するための評価機構を追加する必要がある．そのため，遅延した式の評価に多少のオーバーヘッドが存在すると考えられる．

4.1.3 遅延評価機構の概要

式の評価を遅延するための構文 lazy とその処理を行う force を定義する．

例として図 4.3 を示す．この例で，2*3 は x に束縛される時点では処理されず，lazy により式への参照として x に束縛される．

そして force が 1 度目に遅延した式に適用された際に 2*3 を計算し，x の参照をその結果 6 に付け替える．force の 2 度目の適用に際には 2*3 の計算は行われず，結果である 6 が取り出されるだけである．

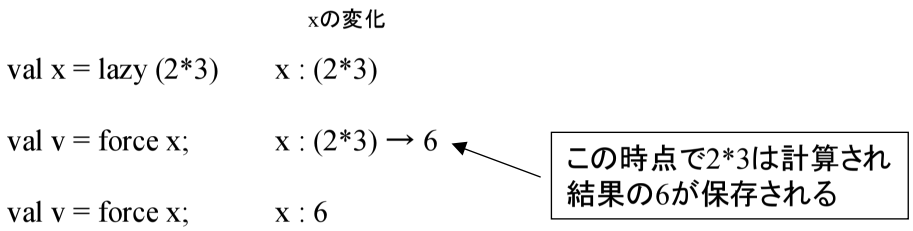

図 4.3: 遅延評価の例

4.2 対象言語の定義

4.2.1 型付ラムダ式の定義

ここで，ループ不変式削除の対象となる型付ラムダ式を示す．対象とする言語の持つ式の定義を示す．

$$
\begin{array}{rcl}
M & := & c(c \in Constant) \\
 & | & x(x \in Variable) \\
 & | & \lambda x.M \\
 & | & \lambda x.PrimFunc \\
 & | & M_1\ M_2 \\
 & | & switch\ M\ of\ (c_1 \Rightarrow M_1|...) \\
 & | & let\ x = M_1\ in\ M_2\ end \\
 & | & let\ rec\ x = M_1\ in\ M_2\ end
\end{array}
$$

各式の意味を示す．

- c は定数を表す．
- x は変数を表す．
- $\lambda x.M$ は x を受け取り M を計算する関数を表す．
- $\lambda x.PrimFunc$ は x を引数とする組み込み関数を表す．
- $M_1\ M_2$ は関数 M_1 への M_2 の適用を表す．
- $switch\ M\ of\ (c_1 \Rightarrow M_1|...)$ は M の結果により場合わけを行い，c_n に当てはまる場合に M_n を実行する．
- $let\ x = M_1\ in\ M_2\ end$ は，M_1 を x に束縛した上で M_2 を実行する．M_1 には x は含まれない．
- $let\ rec\ x = M_1\ in\ M_2\ end$ は，再帰関数 M_1 を x に束縛した上で M_2 を実行する．M_1 には x は含まれる．

またグローバルな変数束縛を次のような書式であらわすとする．

- let x = e で x という名前の変数に e を束縛する．
- let rec x = e で x という名前の e をボディとする再帰関数を束縛する．

例として階乗関数の定義は以下のようになる．

```
let rec factrial =
  fn a =>
    switch a of
       0 => 1
     | n => n * factrial(n-1)
```

このラムダ式の定義に対する，遅延可能関数検出アルゴリズム，ループ不変式検出アルゴリズム，ループ不変式置換アルゴリズム，ループ不変式削除をそれぞれ示す.

4.3 ループ不変式検出法

定義した型付ラムダ式を対象とする，ループ不変式削除のアルゴリズムを示す.

4.3.1 ループ不変式の定義

まず，ループ不変式の定義について考える．手続き型言語の場合は2.2で述べたように，定義されている．この考え方を関数型言語に対して適用することを考えた．また，本研究の対象は純粋関数型言語ではなく，副作用のあるものとする．そこで（破壊的な代入，入出力など）副作用の持たない関数を「遅延可能関数」と定義する.

- 定数，または遅延可能関数への定数の適用は「不変」の印を付ける.
- 「不変」がでつくすまで次を繰り返す.
- 「不変」の印がついていない文に対して，それが「不変」の印がついた式が束縛された変数の遅延可能関数への適用であれば「不変」の印をつける.

4.3.2 ループ不変式削除の条件

手続き型言語の場合には2.3で述べた条件を満たさなければループ不変式を移動することはできない．関数型言語において，それぞれの条件を考える.

- x への代入がループ内に無い.

変数への代入は基本的に関数型言語にないので問題ない.

- 式の定義がすべてのループの出口に対して到達している.

この条件は式の計算を無駄にやらないことを保証するためのものである．これは，本研究で導入する遅延評価を用いることで回避することができるので，問題ない.

4.3.3 遅延可能関数・変数検出アルゴリズム

まず，遅延可能関数・変数検出アルゴリズム DF を示す．このアルゴリズムは，ref 型の値の操作，入出力といった，副作用のない関数を検出するものである．

DF は 2 つの値，DEV, e を引数として動作する．それぞれについて説明する．

- DEV はこれまでに定義されている遅延可能関数・変数の集合である．
- e は let x = e または let rec x = e の e である．

また，遅延可能関数である組み込み関数は $DPrimFunc$ という集合に含まれているとする．

アルゴリズム DF は e が遅延可能関数であれば true そうでなければ false を返す．

再帰関数に適用する際には，その関数自体は遅延可能関数であると仮定して処理を行う，すなわちその時点の DEV をその関数名で拡張したものを用いる．これにより，再帰関数も正しく遅延可能であるかを判定することができる

$$
\begin{aligned}
DF(DEV, c) &= true \\
DF(DEV, x) &= \begin{cases} true & x \in DEV \parallel x \in DPrimFunc \\ & \parallel x\ is\ notFunction \\ false & otherwise \end{cases} \\
DF(DEV, \lambda x.M) &= \begin{cases} DF(DEV, M) & x\ is\ function \\ DF(DEV \cup x, M) & otherwise \end{cases} \\
DF(DEV, \lambda x.PrimFunc) &= \begin{cases} true & PrimFun \in DPrimFunc \\ false & otherwise \end{cases} \\
DF(DEV, M_1\ M_2) &= \begin{cases} DF(DEV, M_2) & M_1 \in DEV \\ false & otherwise \end{cases} \\
DF(DEV, switch\ M\ of & \\
(c_1 \Rightarrow M_1 | ...)) &= DF(DEV, M)\ and\ DF(DEV, M_1)\ and\ \ldots \\
DF(DEV, let\ x = M_1 & \\
in\ M_2\ end) &= DF(DEV, M_1)\ and\ DF(DEV \cup x, M_2) \\
DF(DEV, let\ rec\ x = M_1 & \\
in\ M_2\ end) &= DF(DEV \cup x, M_1)\ and\ DF(DEV \cup x, M_2)
\end{aligned}
$$

引数 e の各場合について説明する．

- $e = c$ の場合
  常に true.

- $e = x$ の場合
  x が DEV に含まれるまたは，$LIPrimFunc$ に含まれる，あるいは関数でなければ true，そうでなければ false.

- $e = \lambda x.M$ の場合
  x が関数であれば $DF(DEV, M)$ の結果，そうでなければ $DF(DEV \cup x, M)$ の結果となる．

- $M_1\ M_2$ の場合
  M_1 が DEV に含まれ $DF(DEV, M_2)$ が true であれば true，そうでなければ false.

- $switch\ M\ of\ (c_1 \Rightarrow M_1|...)$ の場合
  $DF(DEV, M), DF(DEV, M_1), DF(DEV, M_2), \ldots$ がすべて true であれば true，そうでなければ false.

- $let\ x = M_1\ in\ M_2\ end$ の場合
  まず M_1 がループ不変関数かを判定する．ループ不変関数の場合には DEV を x で拡張したもので M_2 を判定し結果を得る．

- $let\ rec\ x = M_1\ in\ M_2\ end$ の場合
  まず M_1 がループ不変関数かを判定する．その際には DEV を x で拡張した上で DF を呼ぶ．ループ不変関数の場合には DEV を x で拡張したもので M_2 を判定し結果を得る．

4.3.4 ループ不変式検出アルゴリズム

4.3.1 で定義したループ不変式を検出するためのアルゴリズム LI を示す．LI は 2 つの値,LIV,e をとる．それぞれについて説明する．

- LIV はループ不変な変数・関数の集合である．

- e はループ内の式である．

またこれまで定義されている遅延可能関数・変数は DEV に保存されているものとする．
アルゴリズム LI は e がループ不変式であれば $true$, そうでなければ $false$ を返す関数である．

$$
\begin{aligned}
LI(LIV, c) &= true \\
LI(LIV, x) &= \begin{cases} true & x \in LIV \\ false & otherwise \end{cases} \\
LI(LIV, \lambda x.M) &= true \\
LI(LIV, \lambda x.PrimFunc) &= true \\
LI(LIV, M_1\ M_2) &= DF(DEV, M_1)\ andalso\ LI(LIV, M_2) \\
LI(LIV, switch\ M\ of\ (c_1 \Rightarrow M_1|...)) &= LI(LIV, M)\ andalso\ LI(LIV, M_1)\ andalso\ ... \\
LI(LIV, let\ x = M_1\ in\ M_2\ end) &= if\ LI(LIV, M_1)\ andalso\ DF(LIV, M_1) \\
&\quad then\ LI(LIV \cup x, M_2) \\
&\quad else\ LI(LIV, M_2) \\
LI(LIV, let\ rec\ x = M_1\ in\ M_2\ end) &= if\ LI(LIV, M_1)\ andalso\ DF(LIV, M_1) \\
&\quad then\ LI(LIV \cup x, M_2) \\
&\quad else\ LI(LIV, M_2)
\end{aligned}
$$

引数 e の各場合について説明する.

- $e = c$,$e = x$,$e = \lambda x.M$ の場合
  常に true.

- $M_1\ M_2$ の場合
  M_1 が遅延可能関数，あるいは副作用のないプリミティブ演算であり，M_2 がループ不変の場合に true.

- $switch\ M\ of\ (c_1 \Rightarrow M_1|...)$ の場合
  M がループ不変かつ，M_n がすべてループ不変の場合 true.

- $let\ x = M_1\ in\ M_2\ end$ の場合
  M_1 がループ不変式かつ遅延可能関数・変数の場合，LIV を x で拡張して M_2 がループ不変．そうでない場合は $LI(LIV, M_2)$ が true であればループ不変.

- $let\ rec\ x = M_1\ in\ M_2\ end$ の場合
  M_1 がループ不変式かつ遅延可能関数・変数の場合，LIV を x で拡張して M_2 がループ不変．そうでない場合は $LI(LIV, M_2)$ が true であればループ不変.

4.3.5 ループ不変式削除アルゴリズム

次に，ループ不変式削除アルゴリズムを示す.

本研究では，ループ不変式を 4.3.4 で示したアルゴリズムによって検出を行う．ループ不変式置換アルゴリズム LIS の概要を示す.

- 入力された式全体にアルゴリズム LI を適用する.
- それがループ不変であると判定されれば新しい変数を作り置き換え，置き換えた式，変数とループ不変式の組を返す.
- そうでなければ式の部分式があれば再帰的にアルゴリズム LIS を適用し，置き換えた式，返り値の変数とループ不変の組をマージしたものの組を返す.

ループ不変式置換アルゴリズム

まずループ不変式置換アルゴリズム LIS を示す.
LIS は 2 つの値,LIV, e をとる．それぞれについて説明する.

- LIV はこれまでに定義されているループ不変式，ループ不変関数の集合である.
- e は再帰関数の body である.

アルゴリズム LIS は再帰関数のボディ e の中にあるループ不変式を発見するたびに新たな変数 x を生成し，$force(x)$ で置き換える処理をする．そして置換後の式，生成した変数とループ不変式の組のリストを返す.

アルゴリズム中で使われている LIE はループ不変式を置換した後の式と，置換リストを用い，ループ不変式削除を完了するアルゴリズムである．これは，次項で説明する.

$$
\begin{array}{rcl}
LIS(LIV, c) & = & (c, []) \\
LIS(LIV, x) & = & (x, []) \\
LIS(LIV, \lambda x.M) & = & let\ (newM, subst) = LIS(LIV, M) \\
 & & in\ (\lambda x.newM, subst) \\
 & & end \\
LIS(LIV, M_1\ M_2) & = & if\ LI(LIV, M_1\ M_2) \\
 & & then\ (force\ x, [(x, M_1\ M_2)]) \\
 & & else\ \ let\ \ (newM_1, subst_1) = LIS(LIV, M_1) \\
 & & \qquad\ \ (newM_2, subst_2) = LIS(LIV, M_2) \\
 & & \quad in\ \ (newM_1\ newM_2, subst_1@subst_2) \\
 & & \quad end \\
LIS(LIV, switch\ M\ of & = & if\ LI(LIV, switch\ M\ of c_1 \Rightarrow M_1|...)) \\
(c_1 \Rightarrow M_1|...)) & & then\ (force\ x, [(x, exp)]) \\
 & & else\ \ let\ \ (newM, subst) = LIS(LIV, M) \\
 & & \qquad\ \ (newM_n, subst_n) = LIS(LIV, M_n) \\
 & & \quad in\ \ (newexp, subst@subst_n) \\
 & & \quad end \\
LIS(LIV, let\ x = M_1 & = & if\ LI(LIV, let\ x = M_1\ in\ M_2\ end) \\
in\ M_2\ end) & & then\ (force\ y, [(y, let\ x = M_1\ in\ M_2\ end)])\quad (y\ is\ fleshname) \\
 & & else\ \ let\ \ (newM_1, subst_1) = LIS(LIV, M_1) \\
 & & \qquad\ \ (newM_2, subst_2) = LIS(LIV, M_2) \\
 & & \quad in\ \ (let\ x = newM_1\ in\ newM_2\ end, subst@subst_n) \\
 & & \quad end
\end{array}
$$

$$
\begin{array}{rcl}
LIS(LIV, let\ rec\ x = M_1 & = & let\ \ newM_1 = LIE(LIS(LIV, M_1)) \\
in\ M_2\ end) & & \qquad (newM_2, subst) = LIS(LIV, M_2) \\
 & & in\ \ (let\ rec\ x = newM_1\ in\ newM_2\ end, subst) \\
 & & end
\end{array}
$$

引数 e の各場合について説明する.

- $e = c$,$e = x$ の場合
  特に操作なし.

- $e = \lambda x.M$ の場合
  M のループ不変式置換を行い (newM,subst) を求め，($\lambda x.newM$,subst) を返す.

- $M_1\ M_2$ の場合
  まず，式全体がループ不変かどうかを判定する．ループ不変式であれば，新しい変数

x を生成し，$(force\ x, (x, M_1\ M_2))$ を返す．そうでなければ，M_1，M_2 のループ不変式置換を行い，$(newM_1, subst_1)$，$(newM_2, subst_2)$ をそれぞれ求め，$newM_1\ newM_2$，$subst_1@subst_2$ を返す．

- $switch\ M\ of\ (c_1 \Rightarrow M_1|...)$ の場合
  まず，式全体がループ不変かどうかを判定する．ループ不変式であれば，新しい変数 x を生成し，$(force\ x, (x, switch\ M\ of\ (c_1 \Rightarrow M_1|...)))$ を返す．そうでなければ，M，M_1，M_2，... のループ不変式置換を行い，$(newM, subst)$，$(newM_1, subst_1)$，$(newM_2, subst_2)$，... をそれぞれ求め，$switch\ newM\ of\ \ (c_1 \Rightarrow newM_1|...)$，$subst@subst_1@subst_2@...$ を返す．

- $let\ x = M_1\ in\ M_2\ end$ の場合
  まず，式全体がループ不変かどうかを判定する．ループ不変式であれば，新しい変数 x を生成し，$(force\ x, (x, let\ x = M_1\ in\ M_2\ end))$ を返す．そうでなければ，M_1, M_2 のループ不変式置換を行い，$(newM_1, subst_1), (newM_2, subst_2)$ をそれぞれ求め，$let\ x = newM_1\ in\ newM_2\ end$，$subst_1@subst_2$ を返す．

- $let\ rec\ x = M_1\ in\ M_2\ end$ の場合
  まず M_1 を対象としてループ不変式削除を行い $newM_1$ を得る．その後 M_2 のループ不変式置換を行い $(newM_2, subst)$ を求め，$let\ rec\ x = newM_1\ in\ newM_2\ end$，$subst$ を返す．

ループ不変式遅延束縛

前項で説明したアルゴリズム LIS は，ループ不変式置換後の式と，置換したもののリストを返す．置換したループ不変式を遅延させてバインドし，その後で新たに関数定義を作り直す必要がある．この処理をするアルゴリズムを LIE とする．

以下に例を示しながら説明する．

```
rec f =
  fn x =>
    switch x of
       0 => 3 * 4
     | _ => 2 * 3 + f (a-1)
```

という再帰関数 f のループ不変式を行う場合，アルゴリズム LIS を適用することにより，

```
rec f =
  fn x =>
    switch x of
       0 => force tmp1
     | _ => force tmp2 + f (a-1)
```

と，$(tmp1, 3*4), (tmp2, 2*3)$ を得る．これらの情報よりループ不変式を lazy にて遅延させバインドし，フレッシュな名前 f' を生成しそれで関数名と関数内部の同じな前の変数名を付け替える，そしてもとの名前と引数で全体をくくることで以下を得る．

```
rec f =
  fn x =>
    let tmp1 = lazy (3*4)
      in let tmp2 = lazy (2*3)
        in let rec f' =
          fn x' =>
            switch x' of
                  0 => force tmp1
                | _ => force tmp2 + f' (x'-1)
    in f' x
    end
```

第5章　実装と実験

5.1　実装環境

実装は大堀らが現在開発中の sml♯[7] の中間言語である型付ラムダ式に対して行った.

5.1.1　sml♯ コンパイラの概要

sml♯ の処理の流れを図 5.1 に示す.

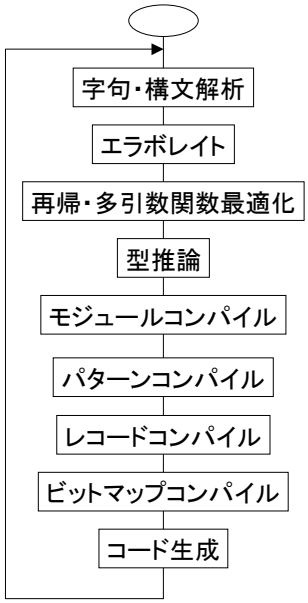

図 5.1: コンパイルの流れ

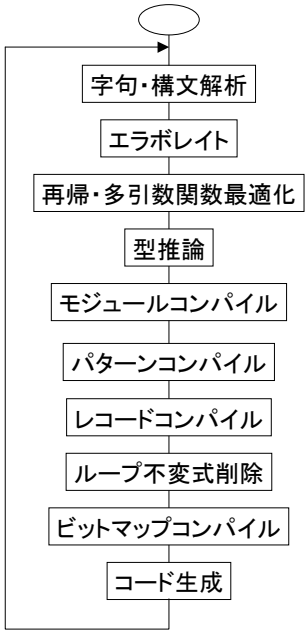

図 5.2: 最適化追加後の流れ

各処理について簡単に説明する.

- 字句・構文解析は，入力されたプログラムを解析し構文木に変換する処理である.
- エラボレイトは，型に依存しない前処理である．例としては結合度の処理などである.

- 再帰・多引数関数最適化は，相互再帰関数の最適化，多引数関数のアンカリー化である．
- 型推論は，変数，関数の型を推論する．
- モジュールコンパイルは，ストラクチャ，シグネチャのコンパイルを行う．
- パターンコンパイルは，パターンマッチングを switch へと変換する．
- レコードコンパイルは，多相型レコード演算をコンパイルする．
- ビットマップコンパイルは，外部関数をガーベージコレクションのための情報を付加する．
- コード生成は，ここまでの結果を目的言語である sml♯ バイトコードへと変換する．

これらの処理を繰り返すことでコンパイルを行っている．コンパイル中で，関数・変数の型などの情報は保存されそれぞれのステップ中で参照することが可能である．

このコンパイラに最適化処理を追加する場所として，レコードコンパイルの後を選択した．一般にコンパイラではコンパイルのステップが進むにつれより単純な命令の集合で表現されるようになる．そのため，よりコンパイルの進んだ段階に最適化処理を追加することが望ましいと考えた．

sml♯ では，ビットマップコンパイルにて変数などにメモリの割り当ての情報が付加されてしまい，その中間言語に対する操作が困難，または不可能になってしまう．そのためレコードコンパイル後の中間言語に対して最適化処理を加えることとした (図 5.2)．

5.1.2 型付ラムダ式の定義

レコードコンパイル後の中間言語である，型付ラムダ式の定義を以下に示す．

```
datatype tlexp =
    TLFOREIGNAPPLY of { funExp:tlexp, instTyList:ty list,
                        argExpList:tlexp list, argTyList: ty list, loc:loc }
  | TLCONSTANT of constant * loc
  | TLVAR of varIdInfoWithType * loc
  | TLGETGLOBAL of string * ty * loc
  | TLGETFIELD of { arrayExp:tlexp, indexExp:tlexp, elementTy:ty, loc:loc}
  | TLSETFIELD of { valueExp:tlexp, arrayExp:tlexp,
                    indexExp:tlexp, elementTy: ty, loc: loc}
  | TLARRAY of { sizeExp:tlexp, initialValue:tlexp,
                 elementTy:ty, resultTy:ty, loc:loc}
```

```
  | TLPRIMAPPLY of { prim:primInfo, instTyList: ty list,
                     argExpList:tlexp list, loc:loc}
  | TLAPP of { funExp:tlexp, funTy:ty, argExp:tlexp, loc:loc}
  | TLAPPM of { funExp:tlexp, funTy:ty, argExpList:tlexp list, loc:loc}
  | TLMONOLET of (varIdInfo * tlexp) list * tlexp * loc
  | TLLET of tldecl list * tlexp list * ty list * loc
  | TLRECORD of { expList:tlexp list, internalTy:ty,
                  externalTy:ty option, loc:loc}
  | TLSELECT of {recordExp:tlexp, indexExp: tlexp, recordTy:ty, loc:loc}
  | TLMODIFY of {recordExp:tlexp, recordTy:ty,
                 indexExp:tlexp, valueExp:tlexp, elementTy:ty, loc:loc}
  | TLRAISE of {argExp:tlexp, resultTy:ty, loc:loc}
  | TLHANDLE of tlexp * varIdInfo * tlexp * loc
  | TLFNM of {varList:varIdInfo list, bodyTy:ty, bodyExp:tlexp, loc:loc}
  | TLFN of {var:varIdInfo, bodyTy:ty, bodyExp:tlexp, loc:loc}
  | TLPOLY of {boundtvars:btvKind IEnv.map, bodyTy:ty, bodyExp:tlexp, loc:loc}
  | TLTAPP of {polyExp:tlexp, polyTy:ty, instTyList:ty list, loc:loc}
  | TLSWITCH of { exp:tlexp, expTy:ty, ruleList:(constant * tlexp) list,
                  defaultExp:tlexp, loc:loc}
  | TLSEQ of tlexp list * ty list * loc
  | TLCAST of {exp:tlexp, targetTy:ty, loc:loc}
  | TLOFFSET of ty * string * loc
  | TLFFIVAL of { funExp:tlexp, libExp:tlexp, argTyList:ty list,
                  resultTy:ty, infTy:ty, loc:loc}

and tldecl =
    TLVAL of (valIdent * tlexp) list * loc
  | TLVALREC of (varIdInfo * ty * tlexp ) list * loc
  | TLVALPOLYREC of btvKind IEnv.map * varIdInfoWithType list
                    * (varIdInfo * ty * tlexp ) list * loc
  | TLLOCALDEC of tldecl list * tldecl list * loc
  | TLSETGLOBAL of string * tlexp * loc
  | TLEMPTY of loc
```

これらの説明を最適化処理に用いた部分だけ簡単に説明する.
tlexp のそれぞれの定義について説明する.

- TLCONSTNT は定数を表す.

- TLVAR は変数を表す．varIdInfoWithType は一意に決まる id と，型と表示名を持つ．
- TLPRIMAPPLY は primInfo で示される PrimtiveFunction へ argExpList で表される tlexp のリストの適用を表す．
- TLAPPM は関数である funExp へ引数となる argExpList の適用を表す．複数の引数を持つことができる．
- TLMONOLET は let 構文を表す．varIdInfo で表される変数へ，組になっている tlexp の束縛をしたのちに tlexp を評価することを表す．
- TLLET は tldecl を束縛したのちに tlexp を評価することを表す．
- TLRECORD はレコードを表す．レコードの型は internalTy または externalTy で示される．これは，データタイプはコンパイル中でレコードに変換されるが，変換後の型を externalTy に保存する．その要素は expList である．
- TLSELECT はレコードからの取り出しを表す．レコードである recordExp から indexExp で示される要素を取り出す．
- TLFNM は varList を引数，bodyExp を関数のボディとする無名関数を表す．
- TLPOLY は多相型を表す．boundtvars は型変数の情報，bodyTy はそのボディの型，bodyExp はボディの式が入る．
- TLTAPP は多相型への型適用を表す．polyExp は多相型の式，polyTy はその型，instTyList には適用する型のリストが入る．
- TLSWITCH は分岐構文を表す．SML♯ の case 構文は switch にパターンコンパイルの段階で変換される．exp を評価し，ruluList のルールに適合する式を実行する．適合しない場合には defaultExp を実行する．

tldecl のそれぞれについて説明する．

- TLVAL は val _ = exp あるいは val x = exp を表す．val _ = exp は exp の結果が必要ない場合に用いられる．主に手続き的な処理をする場合に用いる．
- TLVALREC は val rec x = exp を表す．再帰関数かどうかはコンパイル中で判定されるので，ループ不変式削除最適化は TLVALREC に対して行うことになる．
- TLVALPOLYREC は'a. val rec x = exp を表す．これも再帰関数で

5.1.3 関数・変数定義

レコードコンパイル後の関数･変数定義の構造について説明する．SML♯では変数･関数はTOPBOXEDARRAY･TOPATOMARRAY･TOPDOUBLEARRAY と呼ばれる配列に保存される．TOPATOMARRAY にはアトミックな変数が保存される．TOPDOUBLEARRAY には real 型の変数が保存される．TOPBOXEDARRAY にはそれ以外の変数・関数が保存される．変数・関数定義の際には，それらを TLGETGLOBAL を用い変数に束縛し，束縛した変数に対し TLSETFIELD を行うことで保存している．

- TOPBOXEDARRAY・TOPATOMARRAY・TOPDOUBLEARRAY を変数に束縛する
- 自由変数を TOPBOXEDARRAY・TOPATOMARRAY・TOPDOUBLEARRAY を束縛した変数から取り出す
- 変数，再帰でない関数は TLVAL，再帰関数は TLVALREC，多相型関数は TLVALPOLYREC で定義される
- TOPBOXEDARRAY･TOPATOMARRAY･TOPDOUBLEARRAY に TLSETFIELD にて保存する

5.2 ループ不変式削除の実装

再帰関数・変数定義に対して前章で示したアルゴリズムを用いループ不変式削除を行うプログラムを実装した．

5.2.1 遅延評価の実装

遅延評価はデータタイプでシミュレートすることとした．

実装は以下のようである．

delay 関数は，exp を受け取ることで ref 型で lazy な値を返す．force 関数は，一度目に呼ばれたときに遅延された exp を実行し，その値を保存する．二度目からは保存された値を呼び出すだけである．

```
datatype 'a lazy = EXP of (unit -> 'a) | VALUE of 'a

fun delay exp = ref (EXP exp)

fun force lazyExp =
  case !lazyExp of
```

```
  EXP x =>
    let val v = x ()
    in lazyExp := VALUE v; v
    end
 |VALUE v => v
```

5.2.2 遅延評価関数の取り出し

定義した遅延評価関数を型付ラムダ式の段階で扱うには，TLGETFIELDでTOPBOXEDARRAYから取り出し，一時変数に束縛しなくてはならない．TOPBOXEDARRAYからの値の取り出しのためには次の情報が必要である．

- 変数の名前，型
- TOPBOXEDARRAYの何番目に保存されているか（インデックス）

型はTypeContextより取り出すことができ，インデックスはModuleContextより取り出すことができる．

5.2.3 遅延評価関数の適用

遅延評価関数をバインドした変数に対して，ループ不変式を適用するためには，遅延評価関数は多相型関数のため，ループ不変式の型を適用しなければならない．

これは，型適用を表す構文TLTAPPを用い，lazyにループ不変式の型を適用する場合の下記のようにすればよい．

```
TLTAPP{
  polyExp = lazyをバインドした変数,
  polyTy = lazyの型,
  instTyList [ループ不変式の型],
  loc}
```

5.2.4 遅延可能関数の判定・保存

ループ不変式削除を行った後に，その関数，変数が遅延可能であるかを判定する．遅延可能関数だと判定された場合，それを保存した配列の番地を保存しておく必要がある．

5.3 実験

5.3.1 遅延評価のオーバーヘッド測定

まず遅延評価の機構のオーバーヘッドを測定することとした．測定は Ultra SPARC llli 1.5GHz の Solaris10 上で行った．

本研究では遅延評価機構は関数として定義されている．そのため，評価関数 force の適用にはコストがかかることになる．このコストがどのくらいかをまず測定した．

ループ不変式として，定数のプリミティブ演算（3*2 のような），定数の遅延可能関数への適用 (f a で f が遅延可能関数，a は定数) を選んだ．

ループ不変式が定数のプリミティブ演算の場合

まずループ不変式として定数のプリミティブ演算を含むような関数について測定した．例としては以下のような関数 f である．この関数に対しループ不変式削除を行う場合，行わない場合について，プログラム中のループ不変式を変化させることによりその処理時間の変化を測定した．

```
fun f x =
    let fun f' (b,0) = b
          | f' (b,x) = f' (ループ不変式 + b,x-1)
    in f' (0,x)
    end
```

以下は，ループ不変式削除最適化適用後の f である．

```
fun f x =
    let fun f' x =
        let val tmp = delay (fn () => ループ不変式)
            fun f'' (b,0) = b
              | f'' (b,x) = f'' (force tmp + b,x-1)
        in f'' x end
    in f' (0,x) end
```

図 5.3 はループ不変式として整数演算を取り出した場合についてのグラフである．横軸はループ不変式内の整数演算数，縦軸は f 1000 としたときの計算時間である．non-opt は最適化を適用しない場合，opt は最適化を適用した場合の結果をそれぞれ表す．各場合について 10 回測定しその平均値をとった．

最適化を行った場合にはループ不変式内の演算数に関わらずほぼ一定の計算時間になっている．最適化を行わない場合にはループ不変式内の演算数に比例して計算時間が増加している．

ここで最適化を行った場合と行わない場合の計算時間がほぼ同じになるのは，取り出す演算数が 48 回の場合である．遅延式の評価には約 2.5ms，整数演算は 1 回あたり約 0.05ms かかっていると推察される．

同様に図 5.4 はループ不変式として実数演算を取り出した場合についてのグラフである．横軸はループ不変式内の実数演算数，縦軸は $f\ 1000$ としたときの計算時間である．

ここで最適化を行った場合と行わない場合の計算時間がほぼ同じになるのは，取り出す演算数が 28 回の場合である．遅延式の評価には約 2.5ms，整数演算は 1 回あたり約 0.09ms かかっていると推察される．

このようにプリミティブ演算だけを取り出す場合には，かなり大きな塊をループ不変として検出した場合のみに限らなければむしろ処理時間が増大してしまうこととなる．

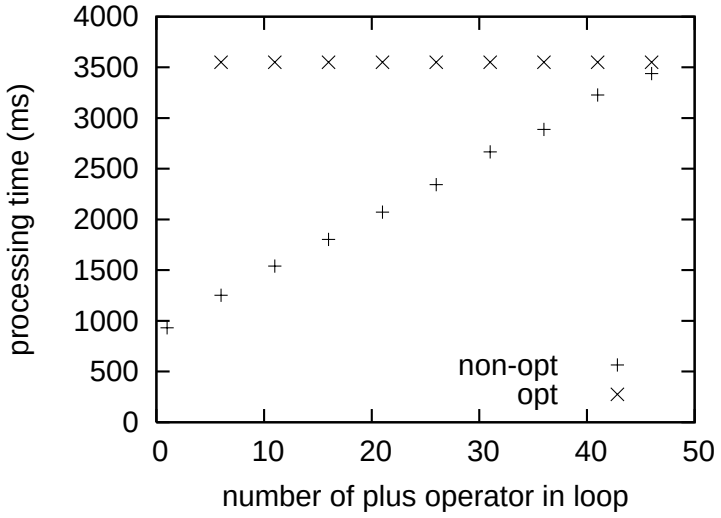

図 5.3: 整数演算の場合

ループ不変式が遅延可能関数への適用の場合

下記プログラムを用いループ不変式が遅延可能関数への適用であった場合について実験した．変数 a は，g 内での f の引数である．これを変化させ f の再帰回数を調節することで，ループ不変式の処理時間を調節した．

```
fun f x =
    let fun f' (b,0) = b
          | f' (b,x) = f' (b,x-1)
    in f' (0,x)
    end
val a = [0 から 10 まで変化]
```

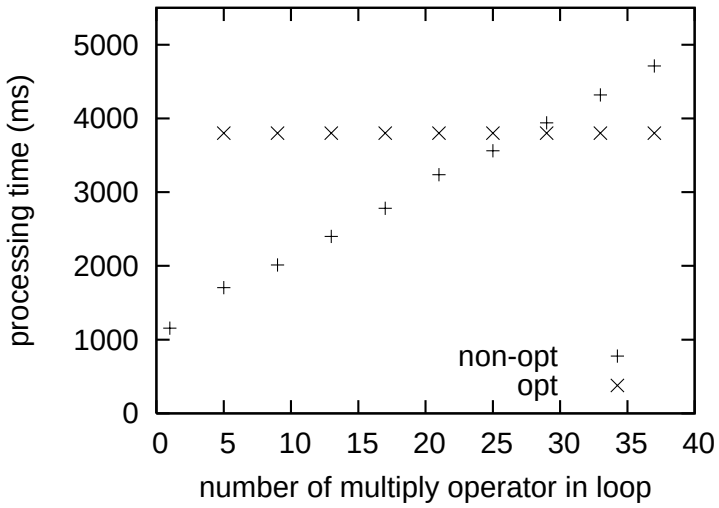

図 5.4: 実数演算の場合

```
fun g x =
    let fun g' (b,0) = b
          | g' (b,x) = g' (f a + b,x-1)
    in g' (0,x)
    end
```

最適化後の g は次のようになる.

```
fun g x =
    let fun g' x =
        let val tmp = delay (fn () => f a)
            fun g'' (b,0) = b
              | g'' (b,x) = g'' (force tmp + b,x-1)
        in g'' x end
    in g' (0,x) end
```

図 5.5 は a の値を 0 から 10 まで変化させながら g 1000 の計算時間の変化をとったグラフである.

最適化をしている場合は，前の例と同様に，a の大きさ（f の再帰回数）にかかわらず，ほぼ一定となった.

ここで最適化を行った場合と行わない場合の計算時間がほぼ同じになるのは，a が 6 の場合である．今回の測定で用いた f は再帰関数としては 1 度の再帰あたりに加算演算を 1 度だけする非常に処理の軽いものである．実際に用いられる関数はこの例よりも重い処理をする．このため再帰している遅延可能関数への適用をループ不変式として取り出すことができる場合は，効果のある場合が多いと考えられる.

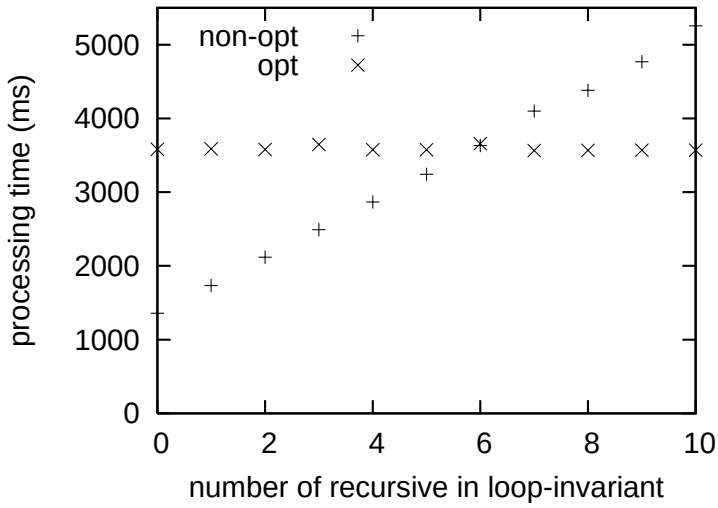

図 5.5: ループ不変式が再帰関数への適用の場合

5.3.2 ループ内で分岐している場合

以下の関数 f のように，case 文を含む再帰関数の場合には，今回提案する手法でなければ簡単にはループ不変式をとりだすことはできない．

```
fun f (b,0) = b
  | f (b,x) = f (b+1,x-1)
fun g x =
    let fun g' (b,0) = b
          | g' (b,x) =
              case x mod 5 of
                0 => g' (f (0, 定数) + b,x-1)
              | 1 => g' (f (0, 定数) + b,x-1)
              | 2 => g' (f (0, 定数) + b,x-1)
              | 3 => g' (f (0, 定数) + b,x-1)
              | 4 => g' (f (0, 定数) + b,x-1)
    in g' (0,x)
    end
```

上記の関数の場合，$g(0, 定数)$ がループ不変式となる．この定数を変化させることでループ不変式の処理の重さを変化させ計算時間を測定した．

図 5.6 は a の値を 0 から 10 まで変化させながら g 1000 の計算時間の変化をとったグラフである．

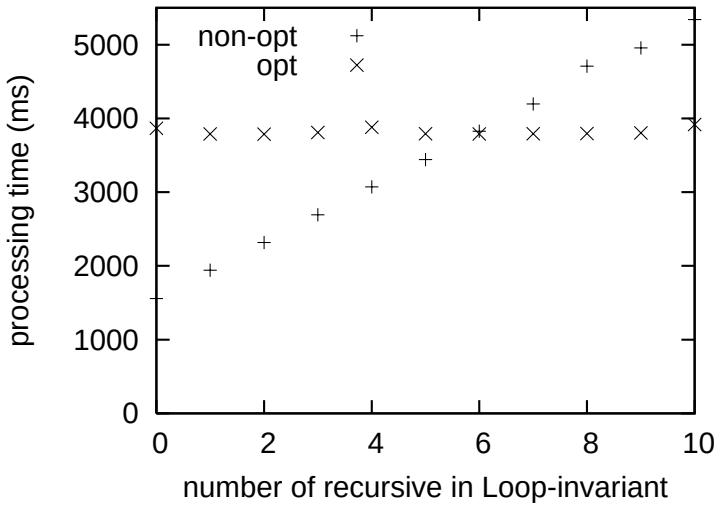

図 5.6: ループ内で分岐がある場合

最適化をしている場合は，この場合も a の大きさ（f の再帰回数）にかかわらず，ほぼ一定となった．そして，最適化を行った場合と行わない場合の計算時間がほぼ同じになるのは，a が 6 の場合である．このように分岐のある場合でも先ほどの例と同様の結果となり，ある程度重い処理を取り出せば効果的であることがわかる．

5.4 考察

本研究で構築した手法の最大の利点はループ内に多数の分岐がある場合でも単純な変換によりループ不変式削除が行えることである．実際に測定した結果ある程度の大きさの処理を取り出す場合には，速度向上を期待できる結果となった．しかしながら，今回の実装ではプリミティブ演算を取り出す場合には遅延評価のオーバーヘッドが大きすぎ，実用的な最適化とはいえない結果となってしまった．

第6章　まとめと今後の課題

6.1　まとめ

正格な関数型言語における，遅延評価を利用したループ不変式削除を実装し，その効果を測定した．ループ中の分岐内にループ不変式がある場合において，従来の手法ではむずかしかったループ不変式削除を遅延評価を用いることで適用することができた．プリミティブ演算など軽い処理を取り出した場合には最適化を適用しない場合に比べてむしろ遅くなってしまう結果になったが，ある程度重い処理をループ不変式として取り出すことができれば効果的に最適化が働くことが確認できた．

6.2　今後の課題

今後の課題としては，より実用的な最適化とすることである．今回の測定結果ではループ不変式として比較的大きな処理のものを取り出さないとむしろ遅くなってしまった．遅くなってしまう原因は，遅延評価機構によるものと考えられる．本研究ではクロージャ生成，評価オーバーヘッドである．

この問題に対する解決策として次のことが考えられる．

１つ目はループ不変式の処理の重さをあらかじめ判定し，オーバーヘッドを考慮しても効果のある場合についてだけループ不変式削除を行うこと．このような処理を行えば確実に早くなる最適化とすることができるが，プログラムを処理する段階で処理の重さを正確に判定することは難しいと考えられる．

２つ目は遅延評価機構を処理系に組み込むことでオーバーヘッドを減らすこと．今回遅延評価機構を関数として実装することで実現したが，そのため処理するたびに関数適用のオーバーヘッドがでてしまった．これを抽象機械でネイティブにサポートすれば高速な遅延評価処理が可能になると考える．

これらを実現できればより実用的にループ不変式削除を行えると考える．

謝辞

本研究を進めるにあたり，ご指導賜りました大堀淳教授に深く感謝いたします．また，日頃よりお世話になりました計算機言語学講座の皆様に深く感謝いたします．

参考文献

[1] Standard ML, http://www.smlnj.org/ .

[2] Alfred V. Aho , Ravi Sethi , and Jeffrey D. Ulman, Compilers: Principles and Techniques and Tools. Addison-Wesley, 1986.

[3] Andrew W. Appel, Compiling with Continuations, Cambridge university press, 1992.

[4] Andrew W. Appel, modern compiler implementation in ML, Cambridge university press, 1998.

[5] Haskell, http://www.haskell.org/.

[6] 大堀　淳，ジャックガリグ，西村　進，アルゴリズムとプログラミング言語，コンピュータサイエンス入門，岩波書店，1999.

[7] SML♯. http://www.pllab.riec.tohoku.ac.jp/smlsharp/index.html .

[8] 中田　育男，コンパイラの構成と最適化，朝倉書店，1999.