プログラミング－言語・基礎・実践－研究報告　9－8
（1992．　10．　30）

項書換え系を用いたコード最適化とその拡張

澤田潤
sawada@kurims.kyoto-u.ac.jp
京都大学数理解析研究所

本論文では、コンパイラが生成する中間コードの基本ブロックが等式論理で表され得るという考えに基づき、最適化のための新たな枠組を与えた。この枠組では、等式に変形されたコードは項書換え系の技法である Associative Commutative な Knuth-Bendix の完備化の変形アルゴリズムによって書き換え規則に変換され、その書き換え規則から最適化されたコードが生成される。また、この技法を拡張することにより constant folding やループの最適化を行なうことを試みた。また、このシステムの実装に際しての高速化も方法も考察した。

A Framework for Code Optimization using Term Rewriting and its Extension

Jun Sawada
sawada@kurims.kyoto-u.ac.jp
Kyoto University, Research Institute for Mathematical Sciences

In this paper, we give a new framework for code optimization on the ground that a basic block can be expressed by an equational theory. In this system, code is converted into a set of equations, and then simplified and enriched into a set of rewriting rules by a variant of AC completion procedure. Finally, optimized code is retrieved from the resulting rewriting rules. Constant folding and optimizing techniques related to loops using this system are also considered. We also discuss tuning up the system in its implementation.

1 はじめに

コードの最適化にはさまざまな手法が存在し、一般にはそれらは別々のものと考えられインプリメントされている．その中でももっとも簡単な技法だけも copy propagation や dead code elimination、common subexpression、algebraic transformation などが挙げられる．これらの技法は単純さにも関わらず強力であって重要であるが、それらを統一的に見るような視点はなかった．

ここでは、項書換え系を用いた新たな最適化の手法を与える．この技法によるオプティマイザはこれらの技法をすべて適応したのと同程度の最適化能力を持ちながら、それぞれの技法を個別にインプリメントしたものとは根本的に異なるものである．

このオプティマイザはまず中間コードを等式論理に直し、その等式の集合を変換し、その結果より最適化されたコードを生成する．次節以後、この過程を順に説明していく．

2 コードの等式論理への変換

まず、コードの等式論理への変換を考える．

とりあえず、5節までで考える最適化の対象はコンパイラの理想的な中間コードの基本ブロックとし、演算結果を変数に代入していく代入文の列と考えることにする．また、ある変数がコードのある位置で live であるとは、その変数の値が次に書き換えられる前に参照されることとする．

このような基本ブロックは次のような点に注意して等式の集合に変換される．

1. 各代入は一つの等式に対応する．等式の両辺は代入の値を表すような適当な項である．
2. コードの中の変数 (配列などの変数を含む) は等式論理の中では定数記号として取り扱われる．
3. ある変数 (配列などの変数を含む) がある代入文で値を代入される場合、その代入以後のこの変数には等式論理の別の定数記号が割り当てられる．

実際に基本ブロックとその変換の例を挙げる．

```
t := i * 200
t := t + 4
x := A[t]
A[t] := y
```

$\Rightarrow$

$$t1 = \times(i, 200)$$
$$t2 = +(t1, 4)$$
$$x = aref(A1, t2)$$
$$A2 = assign(A1, t2, y)$$

1に述べているように、各代入文は等式に変換される．例えば `t := i * 200` の右辺は等式論理の項 $\times(i, 200)$ に置き換えられ、この代入に対応して等式 $t1 = \times(i, 200)$が生成される． ここで注意しなければならないのは、2で述べているように等式の中の t や i などは定数記号であるということである．また、3で述べているように、変数の値が代入によって変更され得る場合は、その変数には異なる定数記号が割り当てられる．例えば、`t := t + 4` では右辺の `t` と左辺の `t` は異なる値を持つので、それぞれに等式論理の異なる定数記号 $t1, t2$ を割り当て、等式 $t2 = +(t1, 4)$ が対応する等式となる．また、配列の参照、代入などは少し変わった形に変形される．例えば、`x := A[t]` は $x = aref(A1, t2)$ と変換され、配列への値の代入 `A[t] := y` には等式 $A2 = assign(A1, t2, y)$ が対応させられている．

この節では、コード上の表現と等式論理における項とを区別するために、項を prefix 形式で書いたが、次節からは場合に応じて infix 形式で項を表記する．

3 AC-完備化による等式論理の変形

等式論理の集合に変換された基本ブロックは AC (associative commutative) 完備化の変形アルゴリズムによって変換される．AC完備化アルゴリズムとは Knuth-Bendix 完備化アルゴリズムの変形アルゴリズムである．一般の項書換え系では、交換律の成り立つような $+$ のような関数記号を許す場合、$x + y = y + x$ は向きづけ不能であり、Knuth-Bendix の完備化アルゴリズムは適応できない．ところが、書き換え規則の適応方法を AC unification を用いて定義し直し、かつ、危険対の定義を拡張することによって、交換律と結合律を満たすような関数記号を許しても完備化が可能となる．ここで用いる AC 完備化の変形アルゴリズムは、一般的な AC 完備化アルゴリズムとは異なるが、ここでの目的であるコード最適化には十分なものである．

今、前節で定義したコードから等式論理への変換によって生成された等式の集合を E で表すとする．また、T と E_{aux} を次に与えられるような等式の集合とする．

$$T = \left\{ \begin{array}{l} (X + Y) + Z = X + (Y + Z) \\ X + Y = Y + X \\ (X \times Y) \times Z = X \times (Y \times Z) \\ X \times Y = Y \times X \end{array} \right\}$$

$$E_{\mathrm{aux}} = \{(X \times Z) + (Y \times Z) = (X + Y) \times Z\}$$

さらに、T によって項 s、t の間に等式が成り立つ時、$s =_T t$ と書く．また、書き換え規則の集合 R に含まれる書き換え規則によって s が t に普通の意

味で書き換えられる時、$s \to_R t$ と書く．この時、書換えを次のように定義する．

定義 1 T を等式の集合とし、R を書き換え規則の集合とする．項 s、t に対して、項 s'、t' が存在して、$s =_T s', t =_T t'$ かつ、$s' \to_R t'$ であるとき s は (R,T) によって t に書き換えらるとし、$s \to t(R,T)$ と書く．(s'、t' はそれぞれ s、t 自身であっても良い)

さらに、同値関係 $=_T$ に関する項 t の同値類を $[t]_T$ と書き、項 $t' \in [t]_T, s' \in [s]_T$ について書き換え $t' \to s'(R,T)$ が成り立つ時、項の同値類 $[t]_T$ が (R,T) によって $[s]_T$ に書き換えられるとし、$[t]_T \to [s]_T(R,T)$ と書く．

次に、項の集合 $Terms$ を次のように定義する．

定義 2 $Terms_0$、$Terms$ はそれぞれ以下のように定義される．

$$Terms_0 \stackrel{def}{=} \{t | \exists s, t = s \in E \ or \ s = t \in E\}$$
$$Terms \stackrel{def}{=} \{t | \exists s \in Terms_0, s =_{Eq} t\}$$
(ただし $Eq = E \cup E_{aux} \cup T$)

$Terms$ は $Terms_0$ の等式論理 $E \cup E_{\rm aux} \cup T$ に関する閉包であって、グラウンドな項の有限集合である．

定義 3 T を等式の集合とし、R を書き換え規則の集合とする．$Terms$ の任意の項 t、t' に対し、$t =_T t'$ が成り立ち、かつ $t \to_R s$ と $t' \to_R s'$ が成り立つ時、$\langle s, s' \rangle$ を $Terms$ 上の R と T に関する危険対とする．

例えば、$R = \{a \times b \to s, (X \times Z) + (Y \times Z) \to (X+Y) \times Z\}$ とし、かつ $(a \times b) + (a \times c)$ が $Terms$ の項である時、$(a \times b) + (a \times c) =_T (b \times a) + (c \times a)$ であり、かつ、$(a \times b) + (a \times c) \to_R s + (a \times c)$ と $(b \times a) + (c \times a) \to_R (b + c) \times a$ が成り立つから、$\langle s + (a \times c), (b + c) \times a \rangle$ は $Terms$ 上の R と T に関する危険対である．

また項 t の重みづけ weight(t) を、関数記号の重みづけ $weight(f)$ を用いて、

$$\text{weight}(t) = \begin{cases} 0 & \text{if } t \text{ is a variable or a constant.} \\ \text{weight}(f) + \sum_{i=1}^{n} \text{weight}(t_i) & \text{if } t = f(t_1, \cdots, t_n) \\ & \text{and } f \text{ is not an AC function symbol.} \\ \text{weight}(f) \times (n-1) + \sum_{i=1}^{n} \text{weight}(t_i) & \text{if } t = f(t_1, \cdots, t_n) \\ & \text{and } f \text{ is an AC function symbol.} \end{cases}$$

と定める．この重みづけによって項の間の半順序を定め、完備化における向きづけにはそれを用いる．

Let P be E. Let $R := \phi$.
loop:
While $P \neq \phi$,
  Let r be an equation $s = t$ in P.
  Let $P := P - \{r\}$.
  Let s' be s fully rewritten
    by $\to (R \cup R_{\rm aux}, T)$.
  Let t' be t fully rewritten
    by $\to (R \cup R_{\rm aux}, T)$.
  If $s' =_T t'$ then goto "loop".
  If $s' > t'$ in the given order,
    then let r' be $s' \to t'$,
    else let r' be $t' \to s'$.
  For every rule $s_i \to t_i$ in R,
    Let s_i' be s_i rewritten by $\to (\{r'\}, T)$.
    Let t_i' be t_i rewritten by $\to (\{r'\}, T)$.
    If $s_i \neq s_i'$ or $t_i \neq t_i'$,
      then let $R := R - \{s_i \to t_i\}$,
      and let $P := P \cup \{s_i' = t_i'\}$.
  End of For.
  Let $R := R \cup \{r'\}$.
End of While.
For every new critical pair $\langle u, v \rangle$
    for $R \cup R_{\rm aux}$ and T on $Terms$,
  $P := \{u = v\} \cup P$ and goto "loop".
End of For.
Return R.

アルゴリズム 1

以上の定義を用いて AC 完備化アルゴリズムをアルゴリズム 1 に示すように定義する．ただし、$R_{\rm aux}$ は $E_{\rm aux}$ の等式を向きづけた

$$R_{\rm aux} = \{(X \times Z) + (Y \times Z) \to (X + Y) \times Z\}$$

なる書き換え規則である．

このアルゴリズムは中間コードの基本ブロックから生成された E によらず停止する．その結果として返す書き換え規則の集合 R は、グラウンドな項をグラウンドな項に書き換える書き換え規則の有限集合で、その書き換え規則の右辺は定数記号であることが容易に確かめられる．また、次の意味で合流性を満たす．

事実 1 R をアルゴリズム 1 によって返された書き換え規則の集合とする．この時、$\to (R \cup R_{aux}, T)$ は $\{[t]_T | t \in Terms\}$ 上で合流性を満たす．すなわち、任意の項 $t \in Terms$ に対し、$[t]_T$ が

$\rightarrow (R \cup R_{aux}, T)$ によって異なる項の同値類に書き換えられ、$[t]_T \rightarrow [t_1]_T(R \cup R_{aux}, T)$、$[t]_T \rightarrow [t_2]_T(R \cup R_{aux}, T)$ となる場合、項 $t' \in Terms$ が存在して、$[t_1]_T \rightarrow [t']_T(R \cup R_{aux}, T)$、$[t_2]_T \rightarrow [t']_T(R \cup R_{aux}, T)$ となる.

この事実は $Terms$ の中の効率的な計算を示す項をその値を持つべき変数を表す定数項に書き換えられることを示しており、次に説明するコード生成で最適化されたコードが取り出されることを意味している.

アルゴリズム 1 の実行の例として簡単な例をあげておく.

$$E = \left\{ \begin{array}{rcl} w & = & b + c \\ y & = & b \times a \\ z & = & a \times c \\ x & = & y + z \\ v & = & a + b \\ u & = & v + c \end{array} \right\}$$

をアルゴリズム 1 に与えると次のような書き換え規則が得られる.

$$R = \left\{ \begin{array}{rcl} b + c & \rightarrow & w \\ b \times a & \rightarrow & y \\ a \times c & \rightarrow & z \\ y + z & \rightarrow & x \\ a \times w & \rightarrow & x \\ a + b & \rightarrow & v \\ w + a & \rightarrow & u \\ v + c & \rightarrow & u \end{array} \right\}$$

この結果に見られるように、R に含まれる書き換え規則には $a \times w \rightarrow x$ や $w + a \rightarrow u$ などのように、もとのコードにはなかった x、u などの計算方法を示すものがある. これらの書き換え規則からもっとも計算コストの小さい計算方法を探し、コードを再生成する方法を次節で示す.

4 等式論理からの最適化コード抽出

この節では、前節で示した方法によって生成された書き換え規則から最適化されたコードを抽出する方法について述べる.

まず、書き換え規則よりグラフを生成する. R をアルゴリズム 1 によって返された書き換え規則とし、R より生成されるグラフ G を次のように定める.

> R のすべての書き換え規則 $s \rightarrow t$ に対し、
>
> Case 1. s が定数項である場合、
> t も定数項であるので、グラフ G にノード s、t と、t から s への有向辺を加える.
> Case 2. s が定数項でない場合、
> この場合も t は定数項であるので、グラフ G に t と s に含まれる定数記号をノードとして加え、s に含まれる定数記号それぞれから t への有向辺を加える.

書き換え規則 R より生成されるグラフ G はデータの流れを表しているといえる. このグラフに簡単なデータフロー解析を施してコードを生成する.

まず、元のコードの基本ブロックの実行前に live であった変数、もしくは定数などに対応するグラフ G のノードをインプットノードと呼ぶことにする. また、基本ブロックの実行終了時に live である変数に対応するノードをアウトプットノードと呼ぶことにする. ところで、同じ変数にも等式論理における異なる定数項が割り当てられていたため、二つ以上のノードが同一の変数に対応することになるが、当然ながら、基本ブロックの実行前の状態に対応するノードがインプットノードとなり、基本ブロックの実行終了時の状態に対応するノードがアウトプットノードとなる.

ここで、グラフ G のノードの集合 N として次のようなものを考える.

> G のすべてのアウトプットノードは N に含まれ、N に含まれる任意のノード n に対し次のいずれかが成り立つ.
>
> 1: n はインプットノードである.
> 2: n に対応する記号 s に対して、ある書き換え規則 $s \rightarrow t \in R$ が存在して、かつ、t に対応するノードが N に含まれている.
> (この時 t は定数記号からなる項である.)
> 3: n に対応する記号 s に対して、ある書き換え規則 $t \rightarrow s \in R$ が存在して、かつ、t に含まれる記号に対応するノードはすべて N に含まれる.

すなわち、N はその要素 n がインプットノードそのものか、もしくはグラフ G における上流の適当なノードが N 自身に含まれているようなノードの集合である. このようなノードの集合 N は一意には定まらない. 例えば、グラフ G に現れるノードすべての集合も N の条件を満たす. ここでは、上の 3 のケースに現れる項 t の重みの和

$$\sum_{\text{3: における t}} \text{weight}(t)$$

ができるだけ小さくなるようなNを選ぶ．この和$\sum_t \mathrm{weight}(t)$が計算をするコストに相当し、$N$のノードが最適化されたコードで計算される値に相当する．このような$\sum_t \mathrm{weight}(t)$が小さくなるようなノードの集合$N$を求めるアルゴリズムの実現に際してははグラフのアウトプットノードからの再帰的な探索によるのが妥当であろう．

さて、このようなノードの集合Nに関して最適化されたコードの生成を行なう．

> Nのノードnに対し次のようなコードを対応させる．
>
> 1: nがインプットノードである場合、対応するコードは無し．
> 2: nがNの定義における2:を満たす場合 グラフ生成時のCase 1のtの値をそのままsにコピーするようなコード's := t'を対応させる．
> 3: nがNの定義における3:を満たす場合 nの計算コストを出す時に求めたnを計算に用いる書き換え規則$s \to t$に対しsを計算しtに代入するコード't := s'を対応させる．

Nの中でグラフの上流にあるノードから対応するコードを書き出していくと、最適化されたコードが生成される．

以上が、前節で生成した書き換え規則からコードを取り出す方法の概要である．ただし、正しいコードを出すためには変数の生死の情報を付け加えるために疑似的な有向辺を付け加え、さらに、それによって生じるかも知れない有向グラフのループを解消しなければならない．

前節で完備化の例としてあげた等式の集合Eの元の基本ブロックはこのコードの実行終了時にliveな変数がu、w、xであるという条件下で以下のように最適化される．

```
w := b + c                 w := b + c
y := a * b                 x := a * w
z := c * a        ⇒        u := w + a
x := y + z
v := a + b
u := v + c
```

5 インプリメントにおける問題

前節までで最適化の方法の概要を説明したが、実際にインプリメントして実行してみると実行時間のかなりの部分を完備化の段階で費やす．したがって、その高速化が問題となる．

まず、3節で述べたようにこの完備化アルゴリズムはAC unificationを使わない．Eとして与える書き換え規則がグラウンドであり、危険対をグラウンドな項の上で生成しているため、完備化はAC matchingアルゴリズムによるグラウンドな項の書換えのみで行なうことができる．このことはかなりの高速化に役立っていると思われるが、その一方で高速なAC matchingアルゴリズムを用いることが必要となる．

また、グラウンドな項の集合*Terms*を定義して、危険対をその上で計算しているが、実際には危険対を生成する可能性がある項は*Terms*の中のごく一部である．特に、等式集合Tや書き換え規則R_{aux}に現れる関数記号を持たない項からは危険対が生まれる可能性がないので*Terms*から除外して構わない．また、ある項の部分項がR_{aux}に現れる関数記号を持っていても、その部分項が*Terms*に他の項として現れているのであれば、同じような理由で元の項は*Terms*から除いても構わない．このような危険対生成に直接関係ない項の*Terms*からの除去は、危険対の計算のみでなく、*Terms*に含まれる項の計算を大きく高速化する．

しかし、*Terms*に含まれる項はコードの基本ブロックの大きさに関して指数的に増えるので、上に挙げたような項を*Terms*から除去するだけでは完備化の手続きを十分速くするのには不十分な場合も存在する．この問題を解決するために*Terms*を項の重み$\mathrm{weight}(t)$や項の深さなどで制限することが考えられる．実際に項の重みが大きい項を*Terms*から除去して同様の完備化手続きを行なうと、実行にかかる時間は大きく削ることができる一方、複雑で自明でない場合を除けば同様に最適化することができる．例えば、前節の最適化の例では$+$に3、$\times$に5の重みをつけて14以上の重みを持つ項を*Terms*から除いても同様の最適化ができる．

6 システムの拡張

前節までで説明した最適化のためのシステムを拡張することを考える．前節までのシステムではconstant foldingができることが自明な場合でも、システムそのものに定数からなる式を計算をする機構を加えてなかったためにこういった最適化を行なえなかった．また、最適化する対象は基本ブロックであって、大域的な最適化については考えてなかった．この節では前節までのシステムを大きく変えることなくこれらの最適化ができるように拡張することを考える．

6.1 単純な拡張

まず、constant folding について考える. constant folding はアルゴリズム 1 を少し変えることのみで実現できる.

いま、コードの定数に対応する記号のみを葉に持つような項を考えると、こうした項は実際に値を計算することができる. 特に、項 s の部分項 s' がこのような項である時に s の中の s' をそれを計算した値で置き換えて得られる項が t である場合、$s \rightarrow_{\mathrm{Calc}} t$ と書くとする. 例えば、

$$(a+1+(2\times 3)) \rightarrow_{\mathrm{Calc}} (a+1+5)$$

である. この時、定数からなる項の計算を行なえるように 定義 1で定めた書き換え規則を次のように定義し直す.

定義 4 T を等式の集合、R を書き換え規則の集合とする. 項 s、t に対して、項 s'、t' が存在して、$s =_T s', t =_T t'$ かつ、$s' \rightarrow_R t'$ もしくは $s' \rightarrow_{\mathrm{Calc}} t'$ であるとき s は $(R \cup\ Calc, T)$ によって t に書き換えらるとし、$s \rightarrow t(R \cup\ Calc, T)$ と書く.

これらの定義は Calc を項の計算を表す無限の書き換え規則の集合を表していると考えれば、容易に理解できるであろう. さらに $Terms$ の代わりに $Terms_{\mathrm{Calc}}$ を次のように定める.

定義 5 $Terms$ を定義 2で定義した通りとして、$Terms_{\mathrm{Calc}}$ を次の用に定める.

$$Terms_{\mathrm{Calc}} \stackrel{def}{=} \{t | \exists s \in Terms, s \rightarrow_{\mathrm{Calc}} t\} \cup Terms$$

この $Terms_{\mathrm{Calc}}$ を用いて危険対を次のように定義する.

定義 6 T を等式の集合とし、R を書き換え規則の集合とする. $Terms_{\mathrm{Calc}}$ の任意の項 t、t' に対し、$t =_T t'$ が成り立ち、かつ $t \rightarrow_{(R \cup \mathrm{Calc})} s$ と $t' \rightarrow_{(R \cup \mathrm{Calc})} s'$ が成り立つ時、$\langle s, s' \rangle$ を $Terms$ 上の $R \cup\ Calc$ と T に関する危険対とする.

この定義によれば $4 \times (x+1) \in Terms_{\mathrm{Calc}}$ の時、

$$4\times(i+1) \leftarrow_{R_{\mathrm{aux}}} (4\times i)+(4\times 1) \rightarrow_{\mathrm{Calc}} (4\times i)+4$$

であるから、$\langle 4 \times (i+1), (4 \times i) + 4 \rangle$ は $R_{\mathrm{aux}} \cup$ Calc と T に関する危険対となる.

さて、以上の定義を用いてアルゴリズム 1 を変形をする. アルゴリズム 1 の $(R \cup R_{\mathrm{aux}}, T)$ による書換えを、$(R \cup R_{\mathrm{aux}} \cup\ \mathrm{Calc}, T)$ による書き換えに置き換える. また、アルゴリズム 1 では $Terms$ 上の $R \cup R_{\mathrm{aux}}$ と T に関する危険対を用いたが、その代わりに $Terms_{\mathrm{Calc}}$ 上の $R \cup R_{\mathrm{aux}} \cup$ Calc と T に関する危険対を用いる.

このような変形されたアルゴリズムによる完備化の例を挙げる.

```
j := i + 1        j = i+1
x := j * 4        x = j×4
k := 2 * 3        k = 2×3
y := x + k        y = x+k
```

図 1　　図 2

図 1 の基本ブロックを最適化する際に得られる等式は図 2 のようになる. 上で述べたアルゴリズム 1 の変形をこの等式に適応した結果得られる書き換え規則の集合は次のようなものとなる.

$$\begin{aligned} i+1 &\rightarrow j \\ j*4 &\rightarrow x \\ k &\rightarrow 6 \\ x+6 &\rightarrow y \\ 4+(4*i) &\rightarrow x \\ 10+(4*i) &\rightarrow y \end{aligned}$$

元のコードでは `y := x + k` とあった y の値を求めるのに `x + 6` を計算することによっても求められることが、これらの書き換え規則の一つ $x+6 \rightarrow y$ によって示されている. さらに、このコードで値を求めたい変数が y だけの時には次のようなコードが生成される.

```
t := 4 * i
y := 10 + t
```

なぜなら、上述の最後の書き換え規則 $10+(4*i) \rightarrow y$ から得られるこのコードの方が `y := x + 6` なる代入を含むコードより効率的だからである. この例に見られるように、単なる constant folding ではなく、constant folding と他の最適化の技法を組み合わせてできる最適化が自然に実現される. さらに、ここに説明した constant folding の方法は 6.2で述べる induction variable に関する最適化と組み合わせることにより適用範囲が広がる.

以上のようにして constant folding が実現されるが、このような拡張のみでは式 `x + 0` を `x` で置き換えることなどはできない. この問題を解決するには、R_{aux} に

$$\begin{aligned} X+0 &\rightarrow X \\ X\times 1 &\rightarrow X \\ X\times 0 &\rightarrow 0 \\ aref(assign(X,Y,Z),Y) &\rightarrow Z \\ assign(X,Y,aref(X,Y)) &\rightarrow X \end{aligned}$$

などの書き換え規則を付け加えるだけでいい.

また、$Terms_{\mathrm{Calc}}$ は $Terms$ にくらべさらに大きな項の集合になるので、実行速度を十分速くするためには、$Terms_{\mathrm{Calc}}$ をより制限することが必要となる.

6.2 大域的最適化について

これまでに考えた最適化はすべて基本ブロックに関する最適化であった. ここでは大域的な最適化の基本的な場合について考察する.

まず、2つ以上の基本ブロックが隣合っている場合を考える. 二つのブロックの間に関数呼び出しなどがあれば、変数の値を変えられる可能性もあるため、copy propagation などの手法は使えないが、分岐などしかない場合にはそうした問題がない. このような基本ブロックの最適化を本論文の最適化システムで行なうためには、後のブロックを最適化する時に前のブロックから得られる等式を一緒に完備化アルゴリズムに掛け、その結果得られる書き換え規則から後のブロックに関する計算を行なう部分のコードを生成すれば良い. これによって、copy propagation や constant folding が二つのブロックにまたがって行なうことができる. この際、後のブロックに対応するコードを生成する時に不当なコードが出ないようにコード生成のプログラムを書き換える技術的な問題がある.

次に単純なループの最適化について考える. ループの最適化としては値の不変な変数に関する code motion や induction variable の除去などがあり、最適化において重要である. そこで、本論文のシステムを用いてこれらの手法の実現することを考える.

まず、コードから等式論理への変換の方法を変更する. 単純なループの中のコードは基本ブロックをなしている. この基本ブロックを2つ順に並べた基本ブロックを考え、その基本ブロックを等式論理に変形する. 例えば、図3のようなループが与えられた時、図4のような基本ブロックを考え、それに対して等式を生成する. 図4の基本ブロックは図3のループを2回繰り返して実行するコードとも考えられる.

```
loop:
    x := 4 * i
    s := x + s
    i := i + 1
    goto loop
```

図3

```
x := 4 * i
s := x + s
i := i + 1
x := 4 * i
s := x + s
i := i + 1
```

図4

$x1 = 4 * i1$
$s2 = x1 + s1$
$i2 = i1 + 1$
$x2 = 4 * i2$
$s3 = x2 + s2$
$i3 = i2 + 1$

図5

その結果得られる等式は図5に示される通りである. ここで、ループの中のコード `x := 4 * i` の x の持つ値は、ループの先の実行と後の実行においてそれぞれ $x1$ と $x2$ で表される. このような対応する記号の組 $\langle x1, x2\rangle$ や $\langle i1, i2\rangle$、$\langle i2, i3\rangle$ などの集合を P とする. また、ループの先の実行直後、変数 x、i の持つ値は記号 $x1$、$i2$ によって表されている. このようなループの先の実行直後の変数の値を表す記号の集合を Q としておく. 記号 s が Q に含まれる場合、s はループの先の実行に対応する等式に現れているはずであり、、後のループの実行時の変数の値を表す t が対応して、$\langle s, t\rangle \in P$ となる.

次に、コードから得られた等式にアルゴリズム1もしくは6.1で述べたアルゴリズム1の拡張をそのまま適応し、完備化された書き換え規則の集合を得る. この集合を R とする.

さらに、ループの先の実行と後の実行における変数の値を表す P の記号の組すべてを、書き換え規則 R を用いて書き換え、同じ項に書き換わるかどうかをテストする. ある記号の組が同じ項に書き換えられるということは、コードから得られる等式論理からその記号が互いに等しいことが導けることを意味し、それらの記号に対応する変数の値がループの実行の後先で変わらないことが示される. このような変数の値はループの外で計算するようにすれば、code motion による最適化が行なわれることになる. このような、不変な変数値を表す記号を次のコード生成の前に求めておくことにする.

さて、最後にコード生成をする訳だが、グラフを生成したり、その探索結果からコードを生成する方法はだいたい同じである. 唯一大きく異なるのが、グラフの探索の方法である. 今、ループの出口で live な変数に対応するグラフノードをアウトプットノードとしよう. この時次のようなノードの集合を N を考える.

> N はすべてのアウトプットノードを含み、かつ、N に含まれるノード n に対し次のいずれかが成り立つ.
>
> 1: n は定数もしくは不変な変数値に対応するノードである.
> 2: n に対応する記号 s に関し、$s \rightarrow t$ が書き換え規則 R に含まれていて、t に対応するノードは N に含まれている.
> 3: n に対応する記号 s に関し、$t \rightarrow s$ が書き換え規則 R に含まれており、t に含まれる記号に対応するノードはすべて N に含まれている.

4: n に対応する記号 s は先のループの実行後の変数の値を表す Q に含まれており、$\langle s,t\rangle \in P$ なる t に対応するノードは N に含まれている.

このようなノードの集合 N は一つに定まらないが、そのうち3のケースの項 t の重みの和 $\sum_t \mathrm{weight}(t)$ のできるだけ小さなものを求める. 実際には、アウトプットノードから有向グラフを上流方向に再帰的に探索することによってこのような N を定める. 最後に上の 2: や 3: のケースのノードについて、それぞれ対応するコード 's := t'、's := t' を生成することによって、ループの最適化されたコードが生成される. 不変な値を持つ変数の値や元のコードではループの入口で live でない変数の初期値を計算するためのコードをループの前に組み合わせることにより、最適化されたコードが得られる.

例に挙げた図3のループの最適化した結果は

```
loop:
    x := x + 4
    s := x + s
    goto loop
```

となる. ここで、ループの前には x の初期値を計算する x := i * 4 などのコードを付け加える必要がある. ところで、x の値を求めるのに induction variable の i は使われていないが、これは 6.1で拡張したアルゴリズム1を図5の等式に適応した結果に $x1+4 \rightarrow x2$ などの書き換え規則が含まれるためである.

7 まとめ

本論文で説明した最適化の方法では一つの統一的な枠組の下でさまざまな最適化の技法と同様の最適化が行なえることがわかった. 特に、constant folding などもシステムをほとんど変更することなく実現することができることが分かった. また、コードから等式への変形、書き換え規則からのコードの生成にいくらか変更を要したものの、基本的には同じ方法を用いることによって、induction variable elimination などもできることを示した.

8 謝辞

この研究を行なうに当たって、有益な助言を与えて下さった元京都大学数理解析研究所助教授 Jim Christian 先生に感謝致します.

参考文献

[1] Aho, A., Sethi, R. and Ullman, J., Compilers, Principles, Techniques, and tools. Addison-Wesley Publishing Company, 1986.

[2] Davidson, J. and Fraser, C., Automatic Generation of Peephole Optimizations, Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices Vol. 19, No. 6, June 1984.

[3] Lankford, D. and Ballantyne, A., Decision Procedures for Simple Equational Theories with Commutative Axioms: Complete Sets of Commutative Reductions. Tech. Rep, Mathematics Dep., Univ. of Texas, Austin, Texas, April 1977.

[4] Hatcher, P., The Equational Specification of Efficient Compiler Code Generation. Computer Language, Vol. 16, No. 1, pp. 81-95, 1991.

[5] Peterson, G. and Stickel, M., Complete Sets of Reductions for Some Equational Theories. Journal of ACM. 28, 2 (April 1981), 233-264.

[6] Sawada, Jun, Code Optimization from the Viewpoint of Equational Theories., 京都大学理学部修士論文