項書換による証明付き最適化

池渕 未来[1,a)

概要：プログラムの最適化には，たとえば short cut fusion のように項書換として実現できるものが多くある．書換による最適化が実用されている例として，Haskell のコンパイラである GHC は，Haskell プログラムに対してプログラマの指定した書換規則を適用し最適化する機能を持つ．一方，証明支援系 Coq は自動証明機能の一環として，等式証明のための自動的な書換の機能を持っている．今回の発表では，その機能を利用して Coq で記述されたプログラムを最適化する方法について示す．この方法ではプログラムは最適化されるだけではなく，最適化前のプログラムと最適化後のプログラムが同じ振舞いをすることが自動的に証明され，最適化の安全性が保証される．具体例として，リストを操作する関数に対する簡単な書換規則による最適化や，さらに高速フーリエ変換の一般的な場合に対する実装から特別な場合での効率的な実装の導出を見る．

キーワード：形式検証，Coq，項書換，最適化

1. はじめに

項書換による最適化は，Haskell のコンパイラである GHC などで採用されている [3]．たとえば，リストに対する関数 `map` を考える．関数 `f`, リスト `xs` に対し，`map f xs` は `xs` の各要素に `f` を適用したリストを返す．このとき，さらにもう一つの関数 `g` が与えられたとき，以下の式が成り立つ．

```
map g (map f xs) = map (fun x ⇒ g (f x)) xs
```

左辺は中間的にリスト `map f xs` を生み出すため，右辺のほうが効率がよい．ソースコード中に左辺の形の項が現れたときは右辺に書き換えるとより効率的なプログラムになる．実際，GHC はこのような書換による最適化を施す機能を持つ．書換のための規則はユーザーも定義できる．

本稿では，Coq 内で書換による最適化を行う方法を示す．Coq はプログラムを記述できると同時にそのプログラムの性質などに関する論理的命題やその証明を記述し証明の正しさを機械的にチェックできる．Coq 内には証明の対象となるプログラムを記述する言語 (Gallina と呼ばれる) に加えて，証明を対話的に進めるためのタクティクと呼ばれるコマンドを持つ．さらに，ユーザーがタクティクを組み合わせて独自の自動証明コマンドを与えるための Ltac と呼ばれる言語も実装されている．標準の自動証明タクティクや Ltac を用いると，与えた性質をみたすようなプログラムを自動的に導出できることが知られている [1]．本稿で紹介する内容は，そういった自動証明によるプログラム導出のうち書換に特化したものである．この方法では用いる書換規則にその等式が正しいことの証明をつければ，最適化前のプログラムと最適化後のプログラムが等価であることが証明され，最適化の正しさが機械的に保証される．

[1] マサチューセッツ工科大学
[a)] ikebuchi@mit.edu

二章でまずリストを操作する簡単なプログラムに対して最適化の例を見た後，三章で高速フーリエ変換 (FFT) の一般的な実装から整数環の剰余環 $\mathbb{Z}/(2^{2^N}+1)\mathbb{Z}$ でのより効率的な実装を導出する．著者の実装した最適化のための Ltac プログラム optimize, optimize' の詳細な説明についてはより多くの Coq の知識を必要とするためここでは省略するが，それらの実装や今回扱う最適化対象のプログラム全体を載せたコード は https://gist.github.com/mir-ikbch/6df11b5ba869c1d17fed5681104f5eb5 で見ることができる．

2. 簡単な例

以下の Coq プログラムを考える．

```
Definition sqr_list xs :=
  map (fun x ⇒ x * x) xs.
Definition sqr_add2_list xs :=
  map (fun x ⇒ x + 2) (sqr_list xs).
```

このプログラムは，Definition キーワードを使って二つの関数 sqr_list と sqr_add2_list を定義している．関数 map は前章で説明した通りである．関数 sqr_add2_list は，sqr_list の定義を展開すると

```
map (fun x ⇒ x + 2) (map (fun x ⇒ x * x) xs)
```

となり，前章で述べたように中間リストを生成する分非効率である．それでは，Coq でこの関数を最適化することを考えよう．まず，最適化に使う規則を自動書換のためのデータベースに登録する．

```
Hint Rewrite map_map : my_db.
Hint Unfold sqr_list sqr_add2_list : my_db.
```

Hint Rewrite rule : db_name は規則 rule を書換データベース db_name に登録し，Hint Unfold ident : db_name は定義 ident をデータベース db_name に登録する．map_map は標準ライブラリで証明済みの前章で述べた定理

```
map g (map f xs) = map (fun x ⇒ g (f x)) xs
```

である．著者の実装した Ltac プログラム optimize を用いて以下のように入力すると最適化が行われる．

```
Definition sa_optimized_sig :
{ f | forall xs, sqr_add2_list xs = f xs } :=
  ltac:(optimize my_db).
```

最適化の結果は

```
proj1_sig sa_optimized_sig
```

で取り出すことができる．Eval simpl in コマンドを使うと結果を確認することができる．

```
Eval simpl in proj1_sig sa_optimized_sig.
```

このコマンドを対話環境で実行すると以下を得る．

```
= map (fun x : nat ⇒ x * x + 2)
: list nat → list nat
```

二つの map が一つに融合されたことが確認できた．
もう一つ短い例を考える．

```
Definition sum xs :=
  fold_left Nat.add xs 0.
Definition sum_of_sqr xs :=
  sum (sqr_list xs).
```

ここで fold_left はリストの左畳み込み関数であり，

```
fold_left f [x1; x2; ...;  xn] init =
f (...( f (f init x1) x2)...)  xn
```

のように振る舞う．Nat.add の中置記法+を用いると，

```
fold_left Nat.add [x1; x2; ...;  xn] 0 =
0 + x1 + x2 + ... + xn
```

と書け，sum はリストの要素の総和を計算する関数として定義されていることが分かる．sum_of_sqr は，その定義からリストの各要素を二乗した後に総和を取る関数であることがただちに分かるが，中間リスト sqr_list xs を生成しない実装も存在する．その実装を導出するためには以下の書換規則を使う．

```
fold_left f (map g xs) init
= fold_left (fun x y ⇒ f x (g y)) xs init
```

この定理は Coq の標準ライブラリで証明されていないため自分で証明する必要があるが，ここでは最適化に着目するため証明は省略する．上の規則の名前を fold_left_map とし，これや新たに定義した関数をデータベースに登録して sum_of_square の最適化をする．

```
Hint Rewrite fold_left_map : my_db.
Hint Unfold sum sum_of_sqr : my_db.
Definition ss_optimized_sig :
{ f | forall xs, sum_of_square xs = f xs } :=
  ltac:(optimize my_db).
Eval simpl in proj1_sig ss_optimized_sig.
```

結果は以下となる．

```
= fun xs : list nat ⇒
  fold_left (fun x y : nat ⇒ x + y * y) xs 0
: list nat → nat
```

sum_of_square の中間リストを生成しない実装が得られた．

最適化に使ったコマンドたちの簡単な説明に移る．sa_optimized_sig, ss_optimized_sig の型は一般的に{x|P x}の形で書かれ，「命題 P x をみたす項 x と P x の証明のペア」の型を表す．つまり，sa_optimized_sig は

```
forall xs, sqr_add2_list xs = f xs
```

をみたす f とその証明のペアとして定義されることになる．その定義の部分の ltac(..) は Ltac を使って定義となる (Gallina 言語の) 項を生み出すために使われる．Ltac プログラム optimize については詳細は省略するが，基本的な動作としては引数 db に対して

(1) Hint Unfold で登録した db 内の定義をすべて展開し，
(2) Hint Rewrite で登録した db 内の書換規則をすべて適用し，
(3) 書換後の左辺の項を結果として返す

というものである．

3. 整数での高速フーリエ変換

高速フーリエ変換 (FFT) は離散フーリエ変換を分割統治を用いて高速に計算するアルゴリズムである．離散フーリエ変換は (連続の) フーリエ変換と同様に複素数や実数の上で考えられることが多いが，整数の上でも考えることができる [4][2](4.3.3 C)，高速な整数乗算や多項式乗算の実装に使われている．より一般には 1 の主 n 乗根を持つ環で考えられる．1 の主 n 乗根とは $\omega^n = 1, \sum_{i=0}^{n-1} \omega^{ik} = 0$ $(1 \le k < n)$ をみたす ω のことであり，複素数体では $\omega = e^{-2\pi i/n}$ にあたる．

一般の FFT アルゴリズムは以下のように振る舞う．R を 1 の主 2^N 乗根 ω_N を持つ環とし，各 $0 \le n \le N$ に対して $\omega_n := \omega_N^{2^{N-n}}$ が 1 の主 2^n 乗根になっているとする．FFT は長さ 2^n のリスト l を受け取り，

- l の長さが $1(= 2^0)$ ならば，l を返す．
- そうでなく，$l = [v_0; v_1; \ldots; v_{2^{n+1}-1}]$ ならば

$$
\begin{aligned}
l_0 &:= [v_0; v_2; ...; v_{2^{n+1}-2}] \\
l_1 &:= [v_1; v_3; ...; v_{2^{n+1}-1}] \\
l'_0 &:= \mathrm{FFT}(l_0) \\
l'_1 &:= [\omega_{n+1}^0; \omega_{n+1}^1; \ldots; \omega_{n+1}^{2^n-1}] \cdot \mathrm{FFT}(l_1)
\end{aligned}
$$

を計算し，$(l'_0 + l'_1) +\!\!+ (l'_0 - l'_1)$ を返す．

ここで，リストに対する $\cdot, +, -$ はそれぞれリストの要素ごとの積，和，差を表し，$+\!\!+$ はリストの結合を表す．Strönhage-Strassen [4] は整数環の剰余環 $\mathbb{Z}/(2^{2^N}+1)\mathbb{Z}$ に対する FFT を整数の高速乗算のために用いた．$\mathbb{Z}/(2^{2^N}+1)\mathbb{Z}$ では $\omega_n = 2^{2^{N-n}}$ が 1 の主 2^n 乗根となる．ここで，$2^k \cdot m$ は左シフト演算によってより効率的に計算されることを利用すると上記の一般の FFT アルゴリズム (特にその中の l'_1 の計算) はこの場合に関してさらに高速化される．

この高速化を Coq 上で実現しよう．FFT アルゴリズム内の l'_1 の計算のみを取り出して以下のような関数を考える．

```
Definition prt N n := pow 2 (pow 2 (N−n)).
Definition fft_aux xs N n :=
  let omegas :=
    map (fun k ⇒ pow (prt N (1+n)) k) [0;..;2^n]
  in
```

```
  map2 mul omegas xs.
```

$\mathbb{Z}/(2^{2^N}+1)\mathbb{Z}$ での演算としての乗算 mul，累乗 pow は既に定義されているものとした．また，[a;..;b] はリスト [a; a+1; ...; b] を表し，map2 は関数 f と二つのリスト [x1; ...; xn], [y1; ...; ym] に対して

```
map2 f [x1; ...;  xn] [y1; ...;  ym]
= [f x1 y1; ...;  f xk yk]
```

と振る舞う関数である．(k=min(n,m) としている．) つまり，map2 mul omegas xs は二つのリスト omegas と xs の要素ごとの積を取る．これらをふまえると，prt N m は ω_n，fft_aux xs N n は xs に FFT(l_1) を代入すると l'_1 に対応することが分かる．

使う書換規則として，以下を用いる．

```
shiftl_mul_pow2 :
  forall x y,
    mul (pow 2 x) y = shiftl y x
pow_pow :
  forall x y z,
    pow (pow x y) z = pow x (mul y z)
map2_map :
  forall A A' B C
    (f : A → B → C)(g : A' → A)
    (xs : list A')(ys : list B),
    map2 f (map g xs) ys
    = map2 (fun x y ⇒ f (g x) y) xs ys
```

これまでと同じように上記の書換規則を書換データベースに登録して optimize タクティクを使いたいところだが，Coq の自動書換タクティク (autorewrite) の制限により map や map2 に渡された関数内の書換に一般に失敗するため，その点を改良した optimize' を実装した．書換データベースを使う代わりに，optimize' は使いたい書換規則を列挙したものを受け取る．

```
Definition ffta_optimized_sig :
  { f | forall xs N n,
        fft_aux xs N n = f xs N n} :=
  ltac:(optimize' my_db
                  (shiftl_mul_pow2,
                   pow_pow, map2_map))

Eval simpl in proj1_sig ffta_optimized_sig.
```

結果として以下が表示される．

```
= fun (xs : list Z)(N : Z)(n : Z) ⇒
  map2 (fun x y ⇒
          shiftl y (shiftl x (N+1−(n+1))))
       [0;..;  2 ^ n] xs
: list Z → Z → list Z
```

中間リストの omegas がなくなり，2 の累乗との積が左シフトに置き換えられた．

4. まとめ

本稿では，Coq でリストを操作する関数や FFT の一般な場合から整数環の剰余環の場合の書換による最適化が施されることを見た．この方法では最適化の結果が最適化前と同値であることの形式的証明は，各書換規則の正しさが証明されている限り自動的に得られる．今回は最適化に使う Ltac プログラムの中身については詳しく説明できなかったが，内部での挙動を知らなくても，展開したい関数定義や使いたい書換規則を渡すだけで利用できるものになっている．プログラムの高速化は重要であるがバグを生み出す可能性もある．今回の手法は手軽な形で安全性の保証された最適化を行う際に有用であると考えられる．

参考文献

[1] Chlipala, A.: *Certified Programming with Dependent Types*, MIT Press (2011).

[2] Knuth, D. E.: *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1997).

[3] Peyton Jones, S., Tolmach, A. and Hoare, T.: Playing by the rules: rewriting as a practical optimisation technique in GHC, *2001 Haskell Workshop*, ACM SIGPLAN (2001).

[4] Schönhage, A. and Strassen, V.: Schnelle Multiplikation großer Zahlen, *Computing*, Vol. 7, No. 3, pp. 281–292 (1971).