Shonan Challenge for Staged HPC: (Simplified) Hidden Markov Model

Kenichi Asai

May 23, 2012

This document in pdf.

1 Challenge Specification

Program in OCaml.

1.1 Input Program

A general matrix-vector multiplication program (to obtain the next state in the hidden Markov model):

```
(* f : int array array -> int array -> int array *)
let f a v =
  let v' = Array.make n 0 in
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      v'.(i) <- v'.(i) + a.(i).(j) * v.(j)
    done
  done;
  v'
```

with a concrete sparse adjacency matrix. For example:

```
(* a : int array array *)
let a = [| [| 1; 0; 0; 1; 0 |];
           [| 0; 0; 1; 0; 0 |];
           [| 0; 1; 0; 0; 0 |];
           [| 0; 0; 1; 1; 1 |];
           [| 0; 0; 1; 0; 1 |] |]
```

1.2 Output Program 1

A program where the elements of the array are inlined:

```
(* f : int array -> int array *)
let f v =
  let v' = Array.make 5 0 in
  v'.(0) <- v.(0) + v.(3);
  v'.(1) <- v.(2);
  v'.(2) <- v.(1);
  v'.(3) <- v.(2) + v.(3) + v.(4);
  v'.(4) <- v.(2) + v.(4);
  v'
```

1.3 Output Program 2

A program where the elements of the array are inlined if the number of non-zero elements in a row is less than a threshold (e.g., three):

```
(* f : int array -> int array *)
let f v =
  let v' = Array.make 5 0 in
  v'.(0) <- v.(0) + v.(3);
  v'.(1) <- v.(2);
  v'.(2) <- v.(1);
  for j = 0 to 4 do
    v'.(3) <- v'.(3) + a.(3).(j) * v.(j)
  done;
  v'.(4) <- v.(2) + v.(4);
  v'
```

2 Solution 1

Program in MetaOCaml.

2.1 Hack for handling semicolor properly

The current MetaOCaml does not handle code-producing 'for' loops in a nice way. Here is a hack to handle them properly (due to Ken Shan). It is essentially the CPS version of a 'for' loop.

```
(* for_to : int -> int -> (int -> 'a -> 'a) -> 'a -> 'a *)
let rec for_to n p f init =
    if n > p then init else for_to n (p-1) f (f p init)

(* id : 'a -> 'a *)
let id c = c

(* semi : 'a -> 'b -> 'b *)
let semi c1 c2 = c1; c2
```

2.2 Rewriting the input program

Required rewriting of the input program for staging. Semantically, it is identical to the input program.

```
(* f1 : int array array -> int array -> int array *)
let f1 a = fun v ->
  let v' = Array.make n 0 in
  for_to 0 (n-1) (fun i ->
    for_to 0 (n-1) (fun j ->
      if a.(i).(j) = 1 then
        semi (v'.(i) <- v'.(i) + v.(j))
      else id))
  v'
```

2.3 Staging the rewritten program

Staging the rewritten program to turn it into a code-generating program.

```
(* semi' : ('a, 'b) code -> ('a, 'c) code -> ('a, 'c) code *)
let semi' c1 c2 = .<(.~c1; .~c2)>.

(* f1' : int array array -> ('a, int array -> int array) code *)
let f1' a = .<fun v ->
  let v' = Array.make n 0 in .~(
  for_to 0 (n-1) (fun i ->
    for_to 0 (n-1) (fun j ->
      if a.(i).(j) = 1 then
        semi' .< v'.(i) <- v'.(i) + v.(j) >.
      else id))
  .<v'>.)>.
```

2.4 Generated program

Generated program for the exmaple matrix. Variable names are changed for better readability.

```
# f1' a;;
- : ('a, int array -> int array) code =
.<fun v ->
   let v' = Array.make 5 0 in
   v'.(0) <- v'.(0) + v.(0);
   v'.(0) <- v'.(0) + v.(3);
   v'.(1) <- v'.(1) + v.(2);
   v'.(2) <- v'.(2) + v.(1);
   v'.(3) <- v'.(3) + v.(2);
   v'.(3) <- v'.(3) + v.(3);
   v'.(3) <- v'.(3) + v.(4);
   v'.(4) <- v'.(4) + v.(2);
   v'.(4) <- v'.(4) + v.(4);
   v'>.
```

If we want to optimize sequences of assigments further (to turn the first two assignments to `v'.(0) <- v.(0) + v.(3);`, for example), we need some more elaboration. (Possibly, the compiler can do such a rather simple optimization?)

3 Solution 2

Program in MetaOCaml.

3.1 Rewriting the input program

Required rewriting of the input program for staging (using the same hack for ‘for’ loop).

```
(* non_zeros : int array -> int *)
let rec non_zeros v =
  let n = ref 0 in
  for i = 0 to Array.length v - 1 do
    if v.(i) <> 0 then n := !n + 1
  done;
  !n

(* example threshold *)
let threshold = 3

(* f2 : int array array -> int array -> int array *)
let f2 a = fun v ->
  let v’ = Array.make n 0 in
  for_to 0 (n-1) (fun i ->
    if non_zeros a.(i) < threshold
    then for_to 0 (n-1) (fun j ->
           if a.(i).(j) = 1 then
             semi (v’.(i) <- v’.(i) + v.(j))
           else id)
    else semi
         (for j = 0 to (let n’ = n-1 in n’) do
            v’.(i) <- v’.(i) + a.(i).(j) * v.(j)
          done))
  v’
```

To handle the row with many non-zero elements, we explicitly introduce an if expression into the program. Since both the branches perform the same task, however, it is still semantically identical to the original program. Notice that CPS `for_to` is used in the ‘then’ branch, while the original `for` is used in the ‘else’ branch for the better staging result.

3.2 Staging the rewritten program

Staging the rewritten program to turn it into a code-generating program (`semi’` being the same as before).

```
(* f2’ : int array array -> (’a, int array -> int array) code *)
let f2’ a = .<fun v ->
  let v’ = Array.make n 0 in .~(
  for_to 0 (n-1) (fun i ->
    if non_zeros a.(i) < threshold
    then for_to 0 (n-1) (fun j ->
           if a.(i).(j) = 1 then
             semi’ .< v’.(i) <- v’.(i) + v.(j) >.
           else id)
    else semi’
      .< for j = 0 to .~(let n’ = n-1 in .<n’>.) do
```

```
           v'.(i) <- v'.(i) + a.(i).(j) * v.(j)
         done >.)
  .<v'>.)>.
```

3.3 Generated program

Generated program for the exmaple matrix. Variable names are changed for better readability.

```
# f2' a;;
- : ('a, int array -> int array) code =
.<fun v ->
   let v' = Array.make 5 0 in
   v'.(0) <- v'.(0) + v.(0);
   v'.(0) <- v'.(0) + v.(3);
   v'.(1) <- v'.(1) + v.(2);
   v'.(2) <- v'.(2) + v.(1);
   for j = 0 to 4 do
     v'.(3) <- v'.(3) + a.(3).(j) * v.(j)
   done;
   v'.(4) <- v'.(4) + v.(2);
   v'.(4) <- v'.(4) + v.(4);
   v'>.
```