A Syntax Directed Register Allocator

BY CHENCHAO DING

cd17@iu.edu

1 Motivation

The original idea and the big picture of this project are from Wang and Dybvig's work [2]. It includes a gentle introduction to model transformers, thus omitted here. There are two main motivations for implementing and justifying this register allocator: aesthetics and performance.

Most important compiler passes, like ANF transformation and explicating control, are syntax directed and structural recursive, while graph-coloring based register allocation introduces knowledges from another branch of mathematics (graph theory), which in some way "breaks the uniformity". An syntax directed register allocator can bring back such kind of uniformity.

There are two interesting analogies within:

$$\textit{static} \text{ register allocation} \sim \textit{dynamic} \text{ cache replacement}$$
$$\texttt{env} \text{ of an interpreter} \sim \texttt{model} \text{ of a register allocator}$$

Static analysis, or specifically, abstract interpretations can be applied to the allocation pass. It's the application of program analysis that achieves correctness preserving program transformation.

The goal of this register allocator is to minimize register-memory and register-register traffic, while most graph coloring algorithms aim at minimizing the number of spills (or the cost of spills). The difference between these two goals can be subtle and intriguing.

Viewing registers as caches of the stack and allowing multi-homing of variables may optimize some loads and saves, thus reducing register-memory traffic. In Section 3, several test cases are prepared to preliminarily justify this idea.

2 Implementation

Model transformers, including `load`, `save`, `shuffle`, and a structural recursive abstract interpreter `ts` (compound transformer) are implemented in Racket based on Wang and Dybvig's work. The strategy for "cache replacement", i.e., selecting a victim to save, is "first in, first out" (FIFO), which is the simplest way to avoid the worst scenario where a variable is selected as a victim immediately after getting loaded to a register.

2.1 Representation of Models

For simplicity, models are represented as a `struct` with 3 fields: an alist `reg-map` that maps variables to registers, an alist `fv-map` that maps variables to frame locations, and a list `regs` that consists of all available (unused) registers in the current "state" of abstract interpretation.

```
(struct M (reg-map fv-map regs) #:transparent)
```

For each procedure, there is an **initial model** that binds its arguments to corresponding registers, according to the calling conventions. E.g.

```
(M '((x . rdi) (y . rsi)) '() R@) ;; R@ = R - '(rdi rsi)
```

is the initial model for procedures with exactly 2 arguments: `x` and `y`.

2.2 Liveness Analysis

Unlike most graph coloring algorithms, MTS doesn't need a full interference graph, but it still needs to know where does a variable die, so that a `remove-dead` operation can be applied to `unbind` all variables that are no longer live and free the corresponding registers. So the liveness analysis is basically a backward ends (death positions) marking process.

2.2.1 The input language

The input programs for liveness analysis are still in the syntax-tree representation:

```
program := (ProgramDefs '() (list defs defs ...))
defs := (Def var ([var : type] ...) type '() exp)
type := Integer | Bool | (Vector type ...)
exp := (begin exp exp ...)
    | (if exp exp exp)
    | (set! var exp)
    | (fun-ref var)
    | (fun-app var var ...)
    | (bop atm atm)
    | atm
atm := var | int | #t | #f
bop := + | - | eq? | < | > | <= | >=
```

We allocate registers on syntax-trees, before `explicate-control` and `select-instructions` passes. The equivalent operation of transposing a control flow graph would be a `flip`:

```
(define (flip exp)
  (match exp
    [`(begin . ,s*) `(begin ,@(flip s*))]
    [`(if ,cnd ,s1 ,s2)
     (match cnd
       [`(if . ,ss)
        `(if ,(flip cnd) ,(flip s1) ,(flip s2))]
       [else `(if ,cnd ,(flip s1) ,(flip s2))])]
    [`(,s . ,s*)
     (match s
       [`(if . ,ss) `(,@(flip s*) ,(flip s))]
       [else `(,@(flip s*) ,s)])]
    ['() '()]))
```

After the backward liveness-analysis, the control flow is `flip` back:

```
(flip (uncover-live-exp (flip exp)))
```

2.2.2 The output language

Every single statement is now paired with a set (or a list), `ends`, that consists of variables die immediately after this statement. Notice that this transformation preserve control flow structures after `flip` twice.

```
...
stmt := (set! var atm)
     | (set! var (bop atm atm))
```

```
      | (set! var (fun-ref var))
      | (set! var (fun-app var var ...))
      | (fun-app var var ...)
      | (bop atm atm)
      | atm
exp* := (Pair stmt ends)
      | (begin exp* exp* ...)
      | (if exp* exp* exp*)
ends := (list var var ...)
```

This output language also serves as the input language of the next pass: `allocate-registers`.

2.3 Allocate Registers

There are 3 main operations on models, namely `load`, `save`, and `shuffle`. Other operations like `bind`, `unbind`, `remove-dead`, etc. serve as helper functions. These 3 operations are essential to the goal of minizing register-memory and register-register traffic.

2.3.1 `load`

```
;; model * (var U (Pair var var)) -> model * (Listof instr)
(load model vars)
```

One or two variables are loaded to registers. `load` first tries searching them in the current register bindings: if they are in the register, do nothing; if there are available registers to bind, then bind them; else select a victim from the current register bindings, `save` it to the stack. One or more move instructions are emitted simultaneously.

2.3.2 `save`

```
;; model * var -> model * (Listof instr)
(save model var)
```

One variable is saved to the stack. `save` first tries searching it in the current stack location bindings: if the variable is in the stack, do nothing; else generate a new stack location and bind it to the variable, and emit the corresponding move instruction.

2.3.3 `shuffle`

There are two scenarios that `shuffle` is needed for consistency. One is when a function is called, and the arguments are initialized with registers according to calling conventions:

```
(define (f [x : Integer] [y : Integer] [z : Integer]) : Integer
  (begin   ;; initial model ((x . rdi) (y . rsi) (z . rdx))
    (Pair (fun-app g z x y) ends*)
    ...))
```

In this example, we need to move `rdx` to `rdi`, `rdi` to `rsi`, and `rsi` to `rdx`, which forms a loop. A temporary register, `rax` is used to accomplish this rotate-like algorithm:

```
(set! rax rdx)
(set! rdx rsi)
(set! rsi rdi)
(set! rdi rax)
```

The other is when two conditional branches join. In Section 3, test cases will show some interesting results originated from shuffling code generation.

3 Performance Tests & Analysis

There are no empirical data for this method so far. Based on features implemented in P523 class, several programs on the scheme benchmark can be used to test and (statically) analyze the performance. One of the state-of-the-art graph coloring algothims, the Chaitin-Briggs algorithm [1], is implemented as a reference for comparison.

3.1 Scheme Benchmark Tests

`sum`, `fib`, `ack`, and `tak` are all recursive functions. So far only "static" analysis of performance is applied. There are four metrics for the output program: memory reads, memory writes, register reads, and register writes.

3.1.1 `sum`

```
(define (sum [n : Integer] [r : Integer]) : Integer
  (if (eq? n 0)
      r
      (tail-sum (- n 1) (+ n r))))
```

MTS's output:

```
'(begin
   (if (eq? rdi 0)
       (begin rsi)
       (begin
         (set! rcx (fun-ref sum))
         (set! rdx (- rdi 1))
         (set! rsi (+ rdi rsi))
         (set! rdi rdx)
         (tail-call rcx))))

'(memory-read 1)
'(memory-write 0)
'(register-read 8)
'(register-write 5)
```

CB's output:

```
'(begin
    (set! r9 rdi)
    (set! rcx rsi)
    (if (eq? r9 0)
      (begin rcx)
      (begin
        (set! r10 (fun-ref tail_sum))
        (set! rdi (- r9 1))
        (set! r9 (+ r9 rcx))
        (set! rdi rdi)          ;; 0 reads, 0 writes
        (set! rsi r9)
        (tail-call r10))))

'(memory-read 1)
'(memory-write 0)
'(register-read 10)
'(register-write 7)
```

In this test, MTS defeats CB.

3.1.2 fib

```
(define (fib [n : Integer]) : Integer
  (if (< n 2)
      n
      (+ (fib (- n 1))
         (fib (- n 2)))))
```

MTS's output:

```
'(begin
   (if (< rdi 2)
       (begin rdi)
       (begin
         (set! rcx (fun-ref fib))
         (set! rdx (- rdi 1))
         (set! rbx rdi)
         (set! rdi rdx)
         (set! r12 (call rcx))
         (set! rdx (fun-ref fib))
         (set! rbx (- rbx 2))
         (set! rdi rbx)
         (set! rbx (call rdx))
         (+ r12 rbx))))

'(memory-read 2)
'(memory-write 0)
'(register-read 12)
'(register-write 10)
```

CB's output:

```
'(begin
   (set! rbx rdi)
   (if (< rbx 2)
       (begin rbx)
       (begin
         (set! r10 (fun-ref fib))
         (set! r12 (- rbx 1))
         (set! rdi r12)
         (set! r12 (call r10))
         (set! r10 (fun-ref fib))
         (set! rbx (- rbx 2))
         (set! rdi rbx)
         (set! rbx (call r10))
         (+ r12 rbx))))

'(memory-read 2)
'(memory-write 0)
'(register-read 12)
'(register-write 10)
```

In this test, MTS generates exactly the same efficient code as CB.

3.1.3 ack

```
(define (ack [m : Integer] [n : Integer]) : Integer
```

```
  (if (eq? m 0)
      (+ n 1)
      (if (eq? n 0)
          (ack (- m 1) 1)
          (ack (- m 1) (ack m (- n 1))))))
```

MTS's output:

```
'(begin
   (if (eq? rdi 0)
       (begin (+ rsi 1))
       (begin
         (if (eq? rsi 0)
             (begin
               (set! rcx (fun-ref ack))
               (set! rdi (- rdi 1))
               (set! rsi 1)
               (tail-call rcx))
             (begin
               (set! rbx (fun-ref ack))
               (set! r12 (- rdi 1))
               (set! rcx (fun-ref ack))
               (set! rsi (- rsi 1))
               (set! rsi (call rcx))
               (set! rdi r12)
               (tail-call rbx))))))

'(memory-read 3)
'(memory-write 0)
'(register-read 11)
'(register-write 10)
```

CB's output:

```
'(begin
   (set! rdi rdi)
   (set! r9 rsi)
   (if (eq? rdi 0)
       (begin (+ r9 1))
       (begin
         (if (eq? r9 0)
             (begin
               (set! r10 (fun-ref ack))
               (set! r9 (- rdi 1))
               (set! rdi r9)
               (set! rsi 1)
               (tail-call r10))
             (begin
               (set! r12 (fun-ref ack))
               (set! r13 (- rdi 1))
               (set! r10 (fun-ref ack))
               (set! r9 (- r9 1))
               (set! rdi rdi)
               (set! rsi r9)
               (set! rbx (call r10))
               (set! rdi r13)
               (set! rsi rbx)
```

```
                (tail-call r12))))))

'(memory-read 3)
'(memory-write 0)
'(register-read 16)
'(register-write 15)
```

In this test, MTS defeats CB.

3.1.4 `tak`

```
(define (tak [x : Integer] [y : Integer] [z : Integer]) : Integer
  (if (>= y x)
      z
      (tak (tak (- x 1) y z)
           (tak (- y 1) z x)
           (tak (- z 1) x y))))
```

`tak` is an interesting example where MTS generates many shuffling lines, since each recursive call in the body has different "orders" of arguments.

MTS's output (static performance analysis only):

```
'(memory-read 6)
'(memory-write 2)
'(register-read 28)
'(register-write 28)
```

CB's output (static performance analysis only):

```
'(memory-read 9)
'(memory-write 2)
'(register-read 23)
'(register-write 26)
```

MTS's memory writes and reads are less than CB's, but its register writes and reads are more than CB's.

3.2 Extra Tests

3.2.1 `many-variables`

This test is aimed at telling the difference between minimizing spills and minimizing memory traffic.

```
(define (test) : Integer
  (let ([a 1])
    (let ([b 2])
      (let ([c 3])
        (let ([d 4])
          (let ([e 5])
            (let ([f 6])
              (let ([g 7])
                (let ([h 8])
                  (let ([i 9])
                    (let ([j 10])
                      (let ([k 11])
                        (let ([l 12])
```

```
                                (let ([m 13])
                                  (let ([n 14])
                                    (let ([o 15])
                                      (let ([p 16])
                                        (+ (+ a (+ b (+ c (+ d (+ e (+ f (+ g (+ h (+
  i (+ j (+ k (+ l (+ m (+ n (+ o (+ p 20))))))))))))))))
                                           1))))))))))))))))
```

MTS's output (static performance analysis only):

```
'(memory-read 5)
'(memory-write 5)
'(register-read 37)
'(register-write 37)
```

CB's output (static performance analysis only):

```
'(memory-read 7)
'(memory-write 7)
'(register-read 26)
'(register-write 26)
```

MTS's memory writes and reads are less than CB's, but its register writes and reads are more than CB's.

4 Observations & Conclusion

For minimizing register-register traffic, MTS can generate relatively more efficient code if:

- arguments of a procedure occur many times in its body.
- arguments of function calls in a procedure's body matches the order and the positions of the procedure's arguments themselves (e.g. `sum` and `ack`).

MTS may generate less efficient code if:

- arguments of a procedure are call-live (thus may cause many invokes of `shuffle` and `save`).

For minimizing register-memory traffic, there is no safe and solid conclusion now, since only 2 examples with spills are tested. However, there are some observations on MTS's behaviors:

- The only form of register-memory traffic is `(set! fv r)` or `(set! r fv)`, and they are generated and inserted by `save` and `load`, respectively.
- Thus no need to patch instructions like `(set! fv.1 fv.2)`.

5 Limitations & Future Work

Loops are not included temporarily since the approximation method now is less accurate than full-fledged, lattice-based data flow analysis. However, benchmark tests involving loops, like `sum`, have some workarounds like converting to a tail call. In Wang and Dybvig's work [2], loops were not discussed, either.

Model operations like `bind`, `unbind`, `remove-dead` are frequently in this implementation, while the main data structure for models now is the association list. Structures like the hash table can be used to improve compile-time efficiency.

The current version of `load` is limited to 1 or 2 variables. An reasonable extension is to allow multiple variables to be loaded to registers at once.

In this project, both MTS and Chaitin-Briggs' are implemented before instruction selection.

	MTS	Chaitin-Briggs'
before `select`	✓	✓
after `select`	?	×

Table 1. advanced comparison between MTS and CB's

Safe conclusion can not be reached unless MTS defeats CB's in both "before `select`" and "after `select`" sets. The ? mark for MTS in the after `select` set means it's not easy to get a neat implementation.

The Chaitin-Briggs algorithm implemented here is the version without live-range splitting, while MTS implies an online live-range splitting which cannot be separated from the main algorithm. To be more fair, a CB's version with live-range splitting should be in consideration.

There are some issues in the current implementation that will generate unnecessary shuffling code (like in `tak` example). More investigations and detective works are needed to find the bottleneck.

Bibliography

[1] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16:428–455, 1994.

[2] Yin Wang and R. Kent Dybvig. Register allocation by model transformer semantics. *CoRR*, abs/1202.5539, 2012.