計算機科学実験 レポート3

1029-28-9483 勝田 峻太朗

2018 年 11 月 12 日

目 次

課題 11 (任意: データフロー解析)

生存変数解析モジュール `live.ml` を参考に，(a) 到達可能定義解析，(b) 到達コピー解析，の中から一つ以上を実装しなさい．`cfg.ml`，`dfa.ml` 等の依存モジュールは汎用なフレームワークとして設計してあるので，修正することなくそのまま利用できるはずである．

到達可能定義解析

到達可能定義解析は, プログラムの `d` 行目の定義が何行目まで有効か調べるための解析である．これにより, 例えば `d'` 行目以降で, ある定義が使われていない場合, その定義を削除するなどという最適化が可能になる．到達可能定義解析は, 以下の伝達関数に従い, 不動点反復を繰り返すことで実行できる．

$$
\begin{aligned}
gen(d) &= \{d\text{行目で新たに生成される定義の集合}\} \\
kill(d) &= d\text{行目で消去される定義の集合} \\
\text{のとき,} & \\
f_j(x) &= gen(j) \cup (X - kill(j))
\end{aligned}
$$

解析器の定義

```
open Vm
open Cfg
open Dfa
module Set = MySet

(* TODO *)
let filter_vars vs =
  Set.from_list (List.filter (fun v ->
      match v with
        Param _ | Local _ -> true
      | Proc _ | IntV _ -> false
    ) (Set.to_list vs))

let transfer (entry_vars: Vm.operand Set.t) (stmt: Vm.instr): Vm.operand Set.t =
  let gen vs =
    lub
      (filter_vars (match stmt with
             Move (dst, src) -> Set.singleton src
           | BinOp (dst, op, l, r) -> Set.from_list [l; r]
           | BranchIf (c, l) -> Set.singleton c
           | Call (dst, tgt, args) -> Set.insert tgt (Set.from_list args)
           | Return v -> Set.singleton v
           | Malloc (dst, vs) -> Set.from_list vs
           | Read (dst, v, i) -> Set.singleton v
           | _ -> Set.empty
         ))
      vs in
```

```
28    let kill vs =
29      match stmt with
30        Move (dst, _)
31      | BinOp (dst, _, _, _)
32      | Call (dst, _, _)
33      | Malloc (dst, _)
34      | Read (dst, _, _) -> Set.remove (Local dst) vs
35      | _ -> vs in
36    let kill' = kill entry_vars in
37    (* TODO *)
38    lub (gen entry_vars) (Set.from_list(List.filter (fun x -> not (MySet.member x kill')) (MySet.to_list
39
40  (* make reachability analyzer *)
41  let make (): Vm.operand Set.t Dfa.analysis =
42    {
43      direction = FORWARD;
44      transfer = transfer;
45      compare = compare;
46      lub = lub;
47      bottom = Set.empty;
48      init = Set.singleton dummy;
49      to_str = string_of_vars;
50    }
```

課題 12 (任意: レジスタ機械コード生成)

> 生存変数解析モジュールによる解析結果を用いて，仮想機械コードをレジスタ機械コードに変換しなさい．Reg.trans 関数の第 1 引数で指定される整数は利用可能な汎用レジスタの個数である．なるべく多くの局所変数を汎用レジスタに割り付けるよう工夫すること．

実装の方針

変換におけるレジスタ割付は, 生存変数解析と制御フローグラフのデータを用いて行う．具体的には, 制御フローの基本ブロックを順番に生存変数を参照しながら変換していく．

変数は空いているレジスタが存在すればレジスタに優先的に割り付けられ, 空いていなければフレーム上に割り付けられる．一度設定された割付はなるべく各段階で引き継がれ, 一度フレームに割り付けられた変数は, レジスタに空きができたとしても新たにレジスタに再割付されることはない．

ただし, レジスタへの割当が必要な場合で, レジスタが満杯の場合は, レジスタに割当られている変数との割当のスワップを行う．スワップ対象となるレジスタに割り当てられている変数は, 確率的に選択する．

フレームの管理

フレームには, 以下の領域を確保する必要性がある．

(1) 呼び出し規約に関わる `saed fp, lr, saved a1, saved a2` の領域.

(2) レジスタに割付しきれなかったローカル変数を置く領域.
(3) 関数呼び出しをするときに, 呼び出し後に必要で, かつレジスタに割り当てられている変数を退避する領域.

(2) が n 個, (3) が m 個ある時のフレームの状態が**図 1** となる用に実装した.

Figure 1: フレームの使用

`offset` 型の変数へのアクセスは, 用意されている `Load, Store` 命令で行うが, レジスタ退避領域へのアクセスは, 簡単のため, 新たに命令 `Save, Restore: (reg*offset) -> instr` を追加した. たとえば `Save r (m-2)` は, **図 1** のレジスタ退避 `m-2` に当たる箇所へのストアを行い,`Restore` はロードを行う.

プログラムの構造

`trans_decl`

`Vm.decl` を `decl` に変換する. これは, 生存変数解析の結果と基本ブロックを参照しながら実行する. 変換の主要な操作は, 各基本ブロックに対して呼び出される `process_block`(下で説明) において行われる.

生存変数解析関係

`live_bblock` 生存変数解析によって得られた基本ブロックの配列.
`get_property` 命令と `Cfg.side` を受け取り, その時点での生存変数を得るための関数.

生成命令関係

基本方針として, 各 Vm.decl の変換過程で生じた命令は, 順番に ref list に追加していく.

instrs_list 生成された命令を順番に追加していくための ref instr list 型の変数.
append_instr 命令を追加する

フレームに逃がす変数の量を管理

レジスタに割り当てられず, フレームに逃がす変数分の領域を計算する必要がある.

save_space 別関数への呼び出しをする際にフレームに退避する必要のあるレジスタ変数の合計を出す.
update_savespace: int -> unit ある関数内で複数の関数呼び出しをする場合, 退避するレジスタの個数の最大値をフレームに退避領域として確保する必要がある. この関数は, 受け取った値 (ある関数呼び出し前に対比する必要のあるレジスタの個数) が現在の save_space より小さければ,save_space の値を更新する.

```
(* manage local memory space *)
  let save_space = ref 0 in
  let update_save_space n =
    if !save_space < n then save_space := n;
    debug_string ("updated save_space to " ^ string_of_int n )
  in
```

基本ブロックごとのレジスタ割当の管理

図 2 は. 以下のコードの実行時の生存変数解析の結果の例である.

```
loop v = (1, 0) in
if v.1 < 101 then
  recur (v.1 + 1, v.1 + v.2)
else
  v.2;;
```

このとき,loop3 の基本ブロック A から出ている 2 つのブロック B と C について VM コードでは,A -> B -> C の順番であるが, 単純にこの順番で前に述べたように実装すれば,C に処理が到達したときに, 最初の生存変数である t4 の割付が受け継がれない. これを防ぐためには, 基本ブロック A の割当情報を保存しておき,C に到達したときにロードする必要がある.

これを実現するために, 各ブロックに対して, ブロック最後の命令におけるレジスタ割当情報を保持しておく仕組みを作る.

past_allocations: (V.id, dest) MyMap.t array ref i 番目のブロック末尾の割付情報を配列の i 番目に保存する. 初期値は空データである.
set_past_alloc 割付情報を i 番目に保存する.
get_past_alloc i 番目のブロックの割付情報を取得する.

```
  (* save allocations in other blocks *)
  let past_allocations = ref (Array.make (Array.length live_bblock) MyMap.empty: (V.id, dest) MyMap.t a
  let set_past_alloc alloc i =
    Array.set !past_allocations i alloc in
  let get_past_alloc i =
    Array.get !past_allocations i in
```

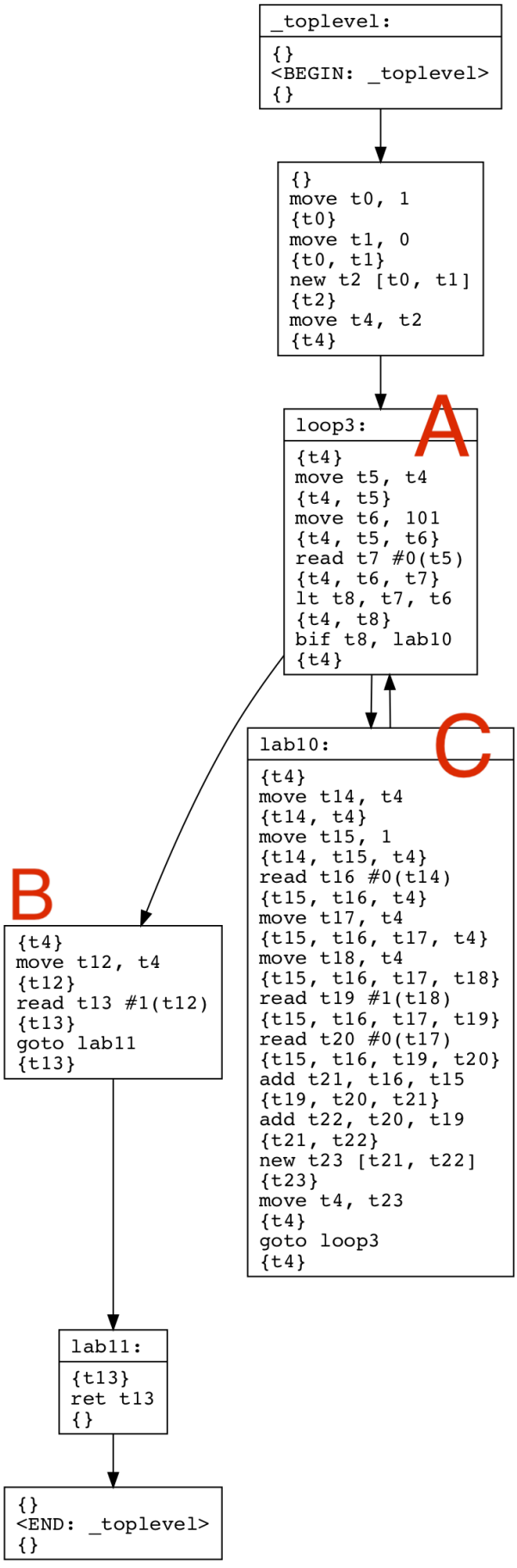

Figure 2: 生存変数解析の結果

オフセットの管理

レジスタに割付しきれなかった変数は offset 型として, フレーム内に割付される. この offset 型の個数を管理する必要がある.

offset_counter 現在の offset 型の個数を保持する. 初期値 0 に設定してある.
get_new_offset 新しい offset のインデックスを返し,offset_counter を更新する.

```
(* manage offset of local *)
  let offset_counter = ref 0 in
  let get_new_offset () =
    let r = !offset_counter in
    offset_counter := !offset_counter + 1;
    r in
```

レジスタ割当の管理, 更新

各命令の変換前に, その命令の前後の生存変数を参照し, レジスタ割当を決定してから値を更新する. 具体的な手順は以下.(ただし, 命令の直前の生存変数を BEFORE, 直後の生存変数を AFTER とする.)

1. BEFORE に入っている生存変数の割当をを前の命令評価時の割当から引き継ぐ.
2. 新しく生成される変数 ($S_{AFTER} \backslash S_{BEFORE}$) の割り付けを決定し, 割当に追加する.

allocation 現在の変数割当を保持する.
get_used_reg 現在の変数割当のうち, 使用済みのレジスタのリスト
get_free_reg 現在変数割当で使用されていないレジスタのリストを返す.
get_nlocal 現在フレームに割り当てられている変数の総数を返す.

```
1   (* manage allocation of Vm.id to dest >>> *)
2   let allocation = ref (MyMap.empty: (V.id, dest) MyMap.t) in
3   let append_alloc (id, d) = allocation := MyMap.append id d !allocation in
4   let search_alloc id = MyMap.search id !allocation in
5   let string_of_alloc () =
6     String.concat ", "
7       (List.sort String.compare
8          (List.map (fun (id, ds) -> "(t" ^ string_of_int id  ^ ", " ^ string_of_dest ds ^ ")")
9             (MyMap.to_list !allocation))) in
10  let alloc_status () =
11    let rec loop l raccum laccum =
12      (match l with
13       | (_, R i):: tl -> loop tl (i :: raccum) laccum
14       | (_, L i):: tl -> loop tl raccum (i:: laccum)
15       | [] -> (raccum, laccum)) in
16    loop (MyMap.to_list !allocation) [] [] in
17  let get_used_reg () =
18    let (r_usage, _) = alloc_status () in
19    r_usage
20  in
21  let get_nlocal () =
22    let (_, l_usage) = alloc_status () in
23    List.length l_usage
24  in
```

```
25    let get_free_reg () =
26      let used = MySet.from_list (get_used_reg ()) in
27      let all = MySet.from_list (List.init nreg (fun x -> x)) in
28      MySet.to_list (MySet.diff all used) in
29    (* >>> manage allocation of Vm.id to dest *)
```

その他補助関数

convert_id Vm.id に対する dest を生成する. 空いているレジスタがあればそこに格納し, なければ,get_new_offset を用いて„ 新しい offset 型の場所に割り付ける.

```
  (* >>> manage allocation of Vm.id to dest *)
  (* helper functions for converting statements >>> *)
  let convert_id id =
    match search_alloc id with
    | Some d -> d
    | None ->
      let free_reg = get_free_reg () in
      if List.length free_reg = 0
      then let ret = get_new_offset () in
        (append_alloc (id, L ret);
         L ret)
      else let ret =  List.hd free_reg in
        (append_alloc (id, R ret);
         R ret)
  in
```

convert_op Vm.operand 型を operand 型へ変換する. Vm.Local 型の場合は,

1. すでにレジスタに割り付けてあれば, そのレジスタを返す.
2. すでに offset 型に割り付けてあれば, 以下で説明する swap に従って割付のスワッピングを行う.
3. 割り付けられていない場合で, レジスタが空いている場合は, 新たにレジスタに割りつける.
4. 割り付けられていない場合で, レジスタが空いていない場合は, スワッピングによってレジスタに割り付ける.

```
  let convert_op op =
    match op with
    | V.Param i -> Param i
    | V.Local id ->
      let dest = (convert_id id) in
      (match dest with
       | R i -> Reg i
       | L o ->
         (* swap offset with some register *)
         let swap_r = select_random (get_used_reg ()) in
         swap o swap_r;
         Reg swap_r)
    | V.Proc l -> Proc l
    | V.IntV i -> IntV i
  in
```

move_to_reg レジスタにある値を Vm.operand に移動する命令列を生成する.

mode_to_dest レジスタにある値を dest 型の場所に移動する命令列を生成する.

```
(* move data in s to register d *)
  let move_to_reg (d: reg) (s: V.operand): instr list =
    match s with
    | V.Param i -> [Move(d, convert_op s)]
    | V.Local id ->
      let dest = convert_id id in
      if is_register dest
      then [Move(d, Reg (reg_of_dest dest))]
      else [Load(d, offset_of_dest dest)]
    | V.Proc l -> [Move(d, Proc l)]
    | V.IntV i -> [Move(d, IntV i)]
  in
  (* move data in register to dest (either reg or offset) *)
  let move_to_dest (d:dest) (s:reg) =
    match d with
    | R r -> [Move(r, Reg s)]
    | L o -> [Store(s, o)]
  in
```

process_block

この関数は,Cfg.bblock と自身の live_bblock の配列におけるインデックスを受け取り, 変換命令を生成する (instrs_list に追加する).

手順

1. bblock.label があれば, 現在変換している関数のポインタやトップレベル出ない場合, Label 型の命令を追加する.
2. bblock.pred があれば, その基本ブロック変換の最後の変数割付データを得て, 変数割付を初期化する. 複数あれば, それを統合する.
3. その後, 各命令 bblock.stmts について, 命令 1 つについて変数割付を更新しながら, 変換する.
4. 最後の命令が変換し終わったら. その時点での変数割付データを past_allocations に追加する.

関数呼び出し (Call) の変換におけるレジスタ退避

関数呼び出し時には, 関数呼び出し後にレジスタに割り当てられている値が失われないように, フレーム内に退避する必要がある. 退避するのは, 呼び出し命令実行後の生存変数のうち, レジスタに割り当てられいるものである. また, 呼び出し後には, 退避した値をレストアする必要がある. 下のコードでは,regs_to_save で, 退避する必要のあるレジスタを探し出し, 各レジスタに対し,Save と Restore を呼び出し前と後に行っている.

```
 | V.Call(id, op, opl) as ins ->
        (* list of live operands after call *)
        let live_ops_after = MySet.to_list (get_property ins Cfg.AFTER) in
        let rec loop ops accum =
          match ops with
          | (V.Local id')::: tl ->
            if id <> id' then
```

```
            (match search_alloc id' with
             | Some (R x) -> loop tl (x::accum)
             | _ -> loop tl accum)
          else loop tl accum
        | _:: tl -> loop tl accum
        | [] -> accum
      in
      let regs_to_save = loop live_ops_after [] in
      update_save_space (List.length regs_to_save);
      append_instr(List.mapi (fun i r -> Save(r, i)) regs_to_save);
      append_instr [Call(convert_id id, convert_op op, List.map convert_op opl)];
      append_instr(List.mapi (fun i r -> Restore(r, i)) regs_to_save);
```

機能: スワップ

変数を新たにレジスタに割り付ける必要があり, レジスタが満杯の場合は, 確率的にレジスタを選び, スワッピングを行う.

具体的なステップは以下.

1. レジスタ割当のデータを交換する.
2. 置換対象のフレーム上のデータを予約レジスタ (Ip) にロードする.
3. 置換対象のレジスタのデータを置換対象のフレーム上にストアする.
4. 予約レジスタの値を置換対象のレジスタに移動する.

```
(* swap allocation for given offset and dest *)
  let swap (off: offset) (r: reg) =
    (* === swap alloc data === *)
    let id_off' = MyMap.reverse_search (L off) !allocation in
    let id_reg' = MyMap.reverse_search (R r) !allocation in
    let (id_off, id_reg) =
      match (id_off', id_reg') with
      | Some o, Some r -> (o, r)
      | _ -> err "failed to swap" in
    allocation := MyMap.assoc id_off (R r) !allocation;
    allocation := MyMap.assoc id_reg (L off) !allocation;
    (* === change data === *)
    (* move offset to reserved_reg *)
    append_instr [Load(reserved_reg, off)];
    (* move reg to offset *)
    append_instr [Store(r, off)];
    (* move reserved_reg to reg *)
    append_instr [Move(r, Reg reserved_reg)];
```

課題 13 (任意: ARM コード生成その 2)

arm_reg.ml の codegen 関数を完成させることにより，課題 12 で生成したレジスタ機械コードから ARM アセンブリコードを生成しなさい.

実装の方針

前に実装した arm_noreg.ml を参考に, 実装を行った.

プログラムの構造

補助的な関数, 値

normal_regs 汎用レジスタ V1~V7 までのリスト
param_to_reg パラメータ番号をレジスタに変換する.
reg_of_operand オペランドをレジスタに変換する.
gen_dest レジスタと Reg.dest を受け取り, レジスタの値を Reg.dest に格納する命令列を生成する.
convert_op Reg.operand を Arm_spec.addr に変換する.

```
let normal_regs = List.filter (fun r -> r <> Ip) (List.map (fun (_, r) -> r) reg_mappings)

let reg_of r = List.assoc r reg_mappings

(* 「reg 中のアドレスから offset ワード目」をあらわす addr *)
let mem_access reg offset = RI (reg, offset*4)

let local_access i = mem_access Fp (-i-2)

let param_to_reg = function
    0 -> A1
  | 1 -> A2
  | i -> err ("invalid parameter: " ^ string_of_int i)

let reg_of_operand op =
  match op with
  | Reg.Param i -> param_to_reg i
  | Reg.Reg r -> reg_of r
  | Reg.Proc l -> err "cannot convert proc to reg"
  | Reg.IntV i -> err "cannot convert immediate to reg"

(* Reg.operand から値を取得し，レジスタ rd に格納するような
   stmt list を生成 *)
let gen_operand rd = function
    Reg.Param i ->
    let rs = param_to_reg i in
    if rd = rs then [] else [Instr (Mov (rd, R rs))]
  | Reg.Reg r ->
```

```
29      let rs = reg_of r in
30      if rd = rs then [] else [Instr (Mov (rd, R rs))]
31    | Reg.Proc lbl -> [Instr (Ldr (rd, L lbl))]
32    | Reg.IntV i -> [Instr (Mov (rd, I i))]
33
34  (* create instrs to store register rs to destination *)
35  let gen_dest (rs: reg) = function
36    | Reg.R r -> [Instr(Mov(reg_of r, R rs))]
37    | Reg.L l -> [Instr(Str(rs, local_access l))]
38
39  let convert_op op =
40    match op with
41    | Reg.Param i -> R (param_to_reg i)
42    | Reg.Reg r -> R (reg_of r)
43    | Reg.Proc l -> L l
44    | Reg.IntV i -> I i
```

ARM コードへの変換関数

gen_decl: Reg.decl -> Arm_spec.stmt list は,ARM コードを生成する. 内部では,arm_stmts:Arm_spec.stmt list ref に生成した命令を追加していく.

arm_stmts 生成された命令列.
append_stmt arm_stmts に命令を追加する.
stmt_instr Reg.instr を変換し,arm_spec に追加していく関数.

命令の変換

命令の変換において, いくつか取り上げて説明する.

Reg.Malloc 以下の手順の命令を生成する.

1. A1, A2 レジスタを退避する.
2. A1 レジスタに確保したいメモリのサイズを格納する.
3. mymalloc にジャンプする.
4. mymalloc から戻ったあと, 得られたメモリの先頭アドレスは A1 に格納されている. ヒープに値を格納して行くため, 以下の 2 つの手順を Reg.operand list の各要素について繰り返す.

- Reg.operand を Ip レジスタに格納する.
- Ip レジスタの値を A1 レジスタのポインタから適切な offset に格納してく.

5. 確保した領域の先頭アドレスへのポインタ A1 を, 適切な Reg.dest に格納する.
6. 退避した A1,A2 レジスタを復帰する.

```
| Reg.Malloc(dest, opl) ->
      let alloc_size = List.length opl in
      append_stmt([
          Instr(Str(A1, RI(Sp, 0)));
          Instr(Str(A2, RI(Sp, 4)));
        ] @
          (gen_operand A1 (Reg.IntV alloc_size)));
```

```
    (* jump to function head *)
    append_stmt([Instr(Bl "mymalloc");]);
    (* store contents in allocated space *)
    let some_reg = reg_of Reg.reserved_reg in
    List.iteri (fun i  op -> append_stmt(gen_operand some_reg op @ [Instr(Str(some_reg, RI(A1, 4 * i)
    (* Step12: store result *)
    append_stmt (gen_dest A1 dest);
    (* Step13: reset 2 arguments  *)
    append_stmt([
        Instr(Ldr(A1, RI(Sp, 0)));
        Instr(Ldr(A2, RI(Sp, 4)));
      ]);
```

Reg.Read(dest, op, i) 以下の手順で処理する.

- **dest がレジスタの場合**, dest が指定するレジスタにメモリ上の値をロードする.
- **dest がフレーム上にある場合**,

1. 読み出すべき値を Ip レジスタに読み出す.
2. Ip レジスタの値を適切な dest に移動する.

```
 | Reg.Read(dest, op, i) ->
      if Reg.is_register dest
      then append_stmt [Instr(Ldr(reg_of (Reg.reg_of_dest dest), mem_access (reg_of_operand op) i))]
      else append_stmt ([
          (* load to reserved reg *)
          Instr(Ldr(reg_of Reg.reserved_reg, mem_access (reg_of_operand op) i))]
          (* store to dest *)
          @ gen_dest (reg_of Reg.reserved_reg) dest
        )
```

Reg.Save, Reg.Restore 図 1 の offset(x) は, nlocal は n+m であるから, local_access (nlocal - offset - 1) でアクセスできる.

```
| Reg.Save(r, offset) -> append_stmt [Instr(Str(reg_of r, local_access (nlocal - offset - 1)))]
    | Reg.Restore(r, offset) -> append_stmt [Instr(Ldr(reg_of r, local_access (nlocal - offset - 1)))]
  in
```