

```
cps-int-closures-k-reg.ml:
```

```
let eval_top t =
```

```
  current_environment := empty_env;
```

```
  current_continuation := Stop;
```

```
  current_handler := Error;
```

```
  match t with
```

```
    WithCont (ki, e) ->
```

```
      eval e
```

```
let eval_program (equations, body) =
```

```
  current_environment := empty_env;
```

```
  current_continuation := Stop;
```

```
  current_handler := Error;
```

```
  fenv := List.map (fun (f,rhs) -> (f, eval_top rhs))
```

```
    equations;
```

```
  eval_top body
```

cgen.ml:

```
let emit_cps_program (defs, body) =
  let env =
    let rec loop defs env =
      match defs with
      [] -> env
      | (global, _)::rest ->
          extend_env_with_global (loop rest env) global (Label global) in
    loop defs empty_env in
  let rec loop defs =
    match defs with
    [] -> emit_top (Global (Label "mcmain")) body env remove_stack_frame_and_return
    | (global, cps)::rest ->
        emit_top (env global) cps env (loop rest)
  in
  loop defs
```

cgen.ml:

```
let emit_top (Global glob_label) (WithCont (_, expr)) env
    nextCode =
    let stopLabel = uniqueLabel "Stop" in
    csect_with_label [p (Long (Int32.of_int 0))]
        glob_label RW;
    let stopCode =
        [ p (Long (make_code_vector_header 0));
          i Lwz [r4; IAL (glob_label, rtoc)];
          i Stw [r3; IA (0, r4)]; ] @
        nextCode in
    csect_with_label stopCode stopLabel PR;
```

cgen.ml:

```
[ li [r12; Con (Int32.add continuation_size code_template_size)];
  i Bl [Label "alloc_heap_space"];
  li [r4; Con continuation_header];
  i Stw [r4; IA (header_offset, r12)];
  i Addic [r4; r12; Con (Int32.add continuation_size heap_object_tag)];
  i Stw [r4; IA (cont_ct, r12)];
  li [r5; Con code_template_header];
  i Stw [r5; IA (untag header_offset, r4)];
  i Lwz [r5; IAL (stopLabel, rtoc)];
  enter_pointer r5 r5;
  i Stw [r5; IA (untag ct_cv, r4)];
  li [envr; Con (make_fixnum 0)];
  i Stw [envr; IA (cont_env, r12)];
  li [ehr; Con (make_fixnum 0)];
  i Stw [ehr; IA (cont_exh, r12)];
  i Stw [ehr; IA (cont_cont, r12)];
  enter_pointer conr r12] @
(emit_expr expr 0 env)
```

cgen.ml:

```
let enter_const con target_reg =  
  match con with  
  | Lambda.CInt int ->  
    [li [target_reg; Con (make_fixnum int)]]  
  | Lambda.CChar char ->  
    [li [target_reg; Con (make_char char)]]
```

```
cgen.ml:
```

```
let fetch_ident id reg level env =  
  match env id with  
  | Local loc -> make_access (level - loc.level)  
    loc.offset reg  
  | Global glob -> [ i Lwz [reg ; IAL (glob, rtoc)];  
    i Lwz [reg; IA (0,reg)]]
```

cgen.ml:

```
let enter_builtin builtin target_reg =  
  match builtin with  
  | "()" -> [li [target_reg; Con mc_unit]]  
  | "true" -> [li [target_reg; Con mc_true]]  
  | "false" -> [li [target_reg; Con mc_false]]
```

Codeerzeugung für valexp

cgen.ml:

```
let rec emit_valexp valexpr level target_reg env =  
  match valexpr with  
  | Const c ->  
    enter_const c target_reg  
  | Ident id ->  
    fetch_ident id target_reg level env  
  | Builtin builtin ->  
    enter_builtin builtin target_reg
```

Lambda

cps-int-closures-k-reg.ml:

```
| Lambda(x, ki, e) ->  
    Closure (!current_environment, x, ki, e)
```

cgen.ml:

```
| Lambda (id, contid, body) ->  
    let new_level = level + 1 in  
    let closure_label = newLabel () in  
    let closure_code = p (Long (make_code_vector_header 0)) ::  
        make_env_frame_from_r3 @  
        emit_expr body new_level  
        (extend_env_with_local env id new_level) in  
    csect_with_label closure_code closure_label PR;
```

Installieren der Closure

```
[ li [r12; Con (Int32.add closure_size code_template_size)];  
  i Bl [Label "alloc_heap_space"];  
  li [r4; Con closure_header];  
  i Stw [r4; IA (header_offset, r12)];  
  i Stw [envr; IA (clos_env, r12)];  
  i Addic [r4; r12; Con (Int32.add  
                        closure_size  
                        heap_object_tag)];  
  i Stw [r4; IA (clos_ct, r12)]; (* set it *)  
  li [r5; Con code_template_header];  
  i Stw [r5; IA (untag header_offset, r4)];  
    i Lwz [r5; IAL (closure_label, rtoc)];  
  enter_pointer r5 r5;  
  i Stw [r5; IA (untag ct_cv, r4)];  
  enter_pointer target_reg r12]
```

```
cps-int-closures-k-reg.ml:
```

```
and eval e =
```

```
  match e with
```

```
    Return (ki, ve) ->
```

```
      let v = eval_val ve in
```

```
      return v
```

```
cgen.ml:
```

```
and emit_expr e level env =
```

```
  match e with
```

```
    Return (k, ve) ->
```

```
      (emit_valexp ve level r3 env) @
```

```
      return r3
```

cps-int-closures-k-reg.ml:

```
let return v =
  match !current_continuation with
  | Stop -> v
  | Continuation (env, h, x, k, e) ->
    current_continuation := k;
    current_handler := h;
    current_environment := extend_env env x v;
    eval e
```

cgen.ml:

```
let return value_reg =
  [ i Lwz [envr; IA (untag cont_env, conr)];
    i Lwz [ehr; IA (untag cont_exh, conr)];
    i Lwz [ctr; IA (untag cont_ct, conr)];
    i Lwz [conr; IA (untag cont_cont, conr)];
    i Lwz [r6; IA (untag ct_cv, ctr)];
    extract_pointer_after_header r6 r6;
    i Mtlr [r6];
    i Mr [r3; value_reg];
    i Blr []]
```

cps-int-closures-k-reg.ml:

```
| TailCall(ve1, ve2, ki) ->
  let v1 = eval_val ve1 in
  let v2 = eval_val ve2 in
  match v1 with
  | Closure (closure_env, x, ki, e) ->
    current_environment := extend_env closure_env x v2;
    eval e
```

cgen.ml:

```
and emit_tailcall func arg level env =
  emit_valexp func level r20 env @
  emit_valexp arg level r3 env @

[ i Lwz [envr; IA (untag clos_env,r20)];
  i Lwz [ctr; IA (untag clos_ct,r20)];
  i Lwz [r6; IA (untag ct_cv, ctr)];
  extract_pointer_after_header r6 r6;
  i Mtlr [r6];
  i Blr []]
```

```
cps-int-closures-k-reg.ml:
```

```
| Call(ve1, ve2, c) ->
```

```
    let v1 = eval_val ve1 in
```

```
    let v2 = eval_val ve2 in
```

```
    let k = eval_cont c in
```

```
    current_continuation := k;
```

```
    match v1 with
```

```
      Closure (closure_env, x, ki, e) ->
```

```
        current_environment := extend_env closure_env x v2;
```

```
        eval e
```

```
cgen.ml:
```

```
| Call (func, arg, cont) ->
```

```
    emit_call func arg cont level env
```

Behandlung der Primitiva

Wissen: Anwendungen der Primitiva sind immer vollständig.

Wissen nicht: Wann erfolgt Anwendung bei binären Primitiva.

Bsp: $x + \text{sum}(x - 1)$

⇒

Cps.Call

```
(Cps.Builtin "+", Cps.Ident "x___3",
 ("a___7",
  Cps.Call (Cps.Builtin "-", Cps.Ident "x___3",
   ("a___8",
    Cps.Call (Cps.Ident "a___8", Cps.Const (Lambda.CInt 1),
     ("a___9",
      Cps.Call (Cps.Ident "sum___2", Cps.Ident "a___9",
       ("a___10",
        Cps.TailCall (Cps.Ident "a___7", Cps.Ident "a___10",
         "k___1")))))))))))
```

Unäre Primitiva

- | Call (Builtin b, arg, (x, cont_expr)) ->
let Some p_info = Camlprim.maybe_primitive_info b in
begin
 match p_info.Camlprim.arity with
 1 ->
 emit_valexp arg level r20 env @
 emit_cont (x, cont_expr) level env @
 emit_unary_builtin b r20 r3 @
 return r3
- | TailCall (Builtin b, arg, _) ->
 emit_valexp arg level r20 env @
 emit_unary_builtin b r20 r3 @
 return r3

Binäre Primitiva: Erste, unvollständige Anwendung

```
| Call (Builtin b, arg, (x, cont_expr)) ->
  let Some p_info = Camlprim.maybe_primitive_info b in
  begin
    match p_info.Camlprim.arity with
    | 1 -> ...
    | 2 ->
      let c level =
        emit_valexp arg level r20 env @
        emit_binary_builtin b r20 r21 r3 @
        return r3 in
      emit_expr cont_expr level
      (extend_env env x (IncompleteBinary c))
```

Binäre Primitiva: Zweite Anwendung mittels TailCall

```
| TailCall (Ident func, arg, _) ->
  begin
    match env func with
      IncompleteBinary c ->
        emit_valexp arg level r21 env @
          c level
      | _ ->
        emit_tailcall (Ident func) arg level env
  end
```

Binäre Primitiva: Zweite Anwendung mittels Call

```
| Call (Ident func_id, arg, (x, cont_expr)) ->
  begin
    match env func_id with
    | IncompleteBinary c ->
      emit_valexp arg level r21 env @
      emit_cont (x, cont_expr) level env @
      c level
    | _ ->
      emit_call (Ident func_id) arg (x, cont_expr)
              level env
  end
```

```
let emit_unary_builtin builtin arg_reg target_reg =  
  match builtin with  
  "dec" ->  
    [ i Subi [target_reg; arg_reg; Con (make_fixnum 1)]]
```

```
let emit_binary_builtin builtin arg1_reg arg2_reg target_reg =  
  match builtin with  
  "+" ->  
    [i Add [target_reg; arg1_reg; arg2_reg]]
```