

```
(* identifiers *)
type ident =
  Strid of string
  | Numid of string * int
type kident =
  ident
```

```
let fresh_counter =
  ref 0
```

```
let fresh_int () =
```

```
  begin
```

```
    fresh_counter := !fresh_counter + 1!
```

```
  end
```

```
let fresh_prefix =
```

```
  Numid (prefix, fresh_int ())
```

```
let fresh_ps =
```

```
  List.map fresh ps
```

type exp =

Var of ident

Const of ident

Pure of ident * exp list

Oper of ident * exp list

Vector of exp list

App of exp * exp

Let of ident list * exp * exp

If of exp * exp * exp

Close of exp * exp

type definition =

ident * ident list * ident * exp

type scheme =

definition list * exp

(* target calculus *)

type trivial =

TVar of ident

TConst of ident

TPure of ident * trivial list

TClose of ident * ident

type header =

HTriv of trivial

HOper of ident * ident list

HApp3 of ident * ident * ident

HVect of ident list

type serious =

SReturn of ident

SApp3 of ident * ident * ident

SLet of ident list * header * serious

SIf of ident * serious * serious

SLetCont of ident * ident list * serious * serious

SField of ident * ident

type xDefinition =

ident * ident * ident * serious

type xScheme =

xDefinition list * serious

```

(* utilities *)
let rename i subst =
  try
    List.assoc i subst
  with x →
    i
let rec remove γ xs =
  match xs with
  | [] → []
  | x::xs → if x=γ then xs else x::remove γ xs
let rec removeL γ xs =
  match γs with
  | [] → []
  | γ::γs → remove γ (removeL γ xs)
let rec union xs γs =
  match xs with
  | [] → γs
  | x::xs → union xs (union γs in
    if List.mem x xs' then xs' else x::xs')

```


and

```

(* toTrivial e *)
let rec toTrivial subst e c =
  match e with
  | Var i →
    c (TVar (rename i subst))
  | Const p →
    c (TConst p)
  | Pure (p, es) →
    (fun ts →
      toTrivialL subst es
      (fun ts →
        Pure (p, es) →
          c (TPure (p, ts)))
      - →
        toSerious subst e
        (fun xx →
          c (TVar xx))
        and toTrivialL subst es c =
          match es with
          | [] → c []
          | e::es →
            toTrivial subst e
            (fun t →
              toTrivialL subst es
              (fun ts →
                c (t::ts)))
            )
    )

```

```

(* toSerious e (Fun i -> SReturn i) *)
(* toSerious : ... * exp * (ident -> serious) -> serious *)
toSerious subst e c =
  match e with
  | Const p ->
    let fr = fresh "t" in
      Slet ([fr], HTRiv (TConst p), c fr)
  | Pure (p, es) ->
    toTrivialL subst es
  | Fun ts ->
    let fr = fresh "t" in
      Slet ([fr], HTRiv (TPure (p, ts)), c fr))
  | Oper (o, es) ->
    let fr = fresh "t" in
      toSeriousL subst es
      (Fun is ->
        Slet ([fr], HOper (o, is), c fr))
  | App (close (e1, e2), e) ->
    toSerious subst e1
    (Fun x1 ->
      toSerious subst e2
      (Fun x2 ->
        toSerious subst e
          (Fun x3 ->
            let fr = fresh "t" in
              Slet ([fr], HApp3 (x1, x2, x3), c fr))))
  | App (f, e) ->
    toSerious subst f
    (Fun ff ->
      toSerious subst e
        (Fun ee ->
          let [frf: frc] = freshL ["f": "t": "t": "t"] in
            Slet ([frf: frc], HTRiv (TVar ff),
              Slet ([fr], HApp3 (frf, frc, ee), c fr))))
  | Let ([x], e1, e2) ->
    toSerious subst e1
    (Fun xx ->
      toSerious ((x, xx) :: subst) e2 c)
  | Let (xs, e1, e2) ->
    toSerious subst e1
    (Fun xx ->
      Slet (xs, HTRiv (TVar (xx)), toSerious subst e2 c))
  | IF (e1, e2, e3) ->
    toSerious subst e1
    (Fun x1 ->
      let [k: xx] = freshL ["k": "t"] in
        let body = c xx in
          SletCont
            (k, remove xx (free_serious body), xx, body,
              SIF
                (x1,
                  toSerious subst e2 (Fun i -> SYield (k, i)),
                  toSerious subst e3 (Fun i -> SYield (k, i))))))

```

```

| Close (e1, e2) →
  toSerious subst e1
  (fun x1 →
    toSerious subst e2
    (fun x2 →
      let fr = fresh "t" in
        Slet ([fr], HTriv (TClose (x1, x2)), c fr)))
  | Vector (es) →
    toSeriousL subst es
    (fun xs →
      let fr = fresh "t" in
        Slet ([fr], HVect (xs), c fr))

```

```

and
toSeriousL subst es c =
  match es with
  [] -> c []
  | (e :: es) ->
    toSerious subst e
      (fun x ->
        toSeriousL subst es
          (fun xs ->
            c (x :: xs)))

```

```

let rec toSeriousT subst e =
  let c x = SReturn x in
  match e with
  | Var i →
    c (rename i subst)
  | Const p →
    let fr = fresh "t" in
    Slet ([fr], HTRiv (TConst p), c fr)
  | Pure (p, es) →
    toTrivialL subst es
  | (fun ts →
    let fr = fresh "t" in
    Slet ([fr], HTRiv (TPure (p, ts)), c fr))
  | Oper (o, es) →
    let fr = fresh "t" in
    toSeriousL subst es
  | (fun is →
    Slet ([fr], HOper (o, is), c fr))
  | App (Close (e1, e2), e) →
    (fun x1 →
    toSerious subst e1
    (fun x1 →
    App (Close (e1, e2), e)))
  | App (f, e) →
    toSerious subst f
    (fun ff →
    App (f, e)
    (fun x3 →
    SApp3 (x1, x2, x3))))
  | Let ([x], e1, e2) →
    toSerious subst e1
    (fun xx →
    toSeriousT (x, xx) :: subst) e2
  | Let (xs, e1, e2) →
    toSerious subst e1
    (fun eel →
    Let (xs, e1, e2)
    (TVar eel, toSeriousT subst e2))
  | If (e1, e2, e3) →
    toSerious subst e1
    (fun x1 →
    SIF
    (x1,
    toSeriousT subst e2,
    toSeriousT subst e3))

```

```

| Close (e1, e2) →
  toSerious subst e1
  (fun x1 →
    toSerious subst e2
    (fun x2 →
      let fr = fresh "t" in
        Slet ([fr], HTriv (TClose (x1, x2)), c fr)))
| Vector (es) →
  toSeriousL subst es
  (fun xs →
    let fr = fresh "t" in
      Slet ([fr], HVect (xs), c fr))

```

```
let transDefinition (f, fvs, a, e) =  
  let fr = fresh "t" in  
  (f, fr, a,  
   Slet (fvs, HTriv (TVar fr), toSeriousT [] e))  
let transScheme (defs, e) =  
  (List.map transDefinition defs, toSeriousT [] e)
```

```
(* examples *)
let var s = Var (Strid s)
let const s = Const (Strid s)
let pure (s, es) = Pure (Strid s, es)
let oper (s, es) = Oper (Strid s, es)
let app (e1, e2) = App (e1, e2)
let mlet (s, e1, e2) = Let ([Strid s], e1, e2)
let mif (e1, e2, e3) = If (e1, e2, e3)
let close (e1, e2) = Close (e1, e2)
let vector es = Vector es

let ex1 = If (pure ("=", [var"n"! const"0"]),
  [var"n"!
  pure ("*",
  const"1",
  pure ("*",
  [var"n"!
  pure ("fac", Vector [],
  pure ("-", [var"n"! const"1"])]))]])

let ex2 = If (pure ("=", [var"n"! const"0"]),
  const"1",
  mlet ("e", App (Close (var"fast_exp", Vector[]),
  var"x"),
  pure ("/", [var"n"! const"2"])),
  pure ("*", [var"e"!
  pure ("*", [var"e"!
  pure ("mod", [var"n"!
  const"2"])]))

let ex3 = If (pure ("=", [var"n"! const"0"]),
  const"1",
  App (Close (var"down", Vector[]), pure ("-", [var"n"!
  const"1"])))
```

```

let serious1 = (* toSeriousT [ ] ex1 *)
  Slet ([Numid ("t", 1)],
  HTRiv (TPure (Strid "n"?: TConst (Strid "0"[]))),
  Sif (Numid ("t", 1),
  Slet ([Numid ("t", 6)], HTRiv (TConst (Strid "1"))),
  Slet ([Numid ("t", 6)], HTRiv (TConst (Strid "1"))),
  Sreturn (Numid ("t", 6))),
  Slet ([Numid ("t", 2)], HVect [ ]),
  Slet ([Numid ("t", 3)],
  HTRiv (TPure (Strid "n"?: TConst (Strid "1"[]))
  ),
  Slet ([Numid ("t", 4)],
  HAP3 (Strid "fac", Numid ("t", 2), Numid ("t", 3)),
  Slet ([Numid ("t", 5)],
  HTRiv (TPure (Strid "n*?", [TVar (Strid "n"?: TVar (Numid ("t",
  4))[]])),
  Sreturn (Numid ("t", 5)))))
let serious3 = (* toSeriousT [ ] ex3 *)
  Slet ([Numid ("t", 7)],
  HTRiv (TPure (Strid "n"?: TConst (Strid "0"[]))),
  Sif (Numid ("t", 7),
  Slet ([Numid ("t", 10)], HTRiv (TConst (Strid "1"))),
  Sreturn (Numid ("t", 10))),
  Slet ([Numid ("t", 8)], HVect [ ]),
  Slet ([Numid ("t", 9)],
  HTRiv (TPure (Strid "n"?: TConst (Strid "1"[]))
  ),
  SAPP3 (Strid "down", Numid ("t", 8), Numid ("t", 9)))))

```