```
module compiler where

open import source
open import target
open import lib


infixr 1 _⇒ₛ_
infixl 2 _×_

-- Product and projection function
data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B

  : ∀ {A B} → A × B → A
  (a , _) = a

  : ∀ {A B} → A × B → B
  (_ , b) = b

--  Type Interpretation
Compl : SD → Set
Compl sd = I sd

_×ₛ_ : (SD → Set) → (SD → Set) → SD → Set
(P ×ₛ Q) sd = P sd × Q sd

_⇒ₛ_ : (SD → Set) → (SD → Set) → SD → Set
(P ⇒ₛ Q) sd = ∀{sd'} → (sd ≤ₛ sd') → P sd' → Q sd'

Intcompl : SD → Set
Intcompl = R ⇒ₛ Compl


⟦_⟧ty : Type → SD → Set
⟦ comm ⟧ty = Compl ⇒ₛ Compl
⟦ intexp ⟧ty = Intcompl ⇒ₛ Compl
⟦ intacc ⟧ty = Compl ⇒ₛ Intcompl
⟦ intvar ⟧ty = ⟦ intexp ⟧ty ×ₛ ⟦ intacc ⟧ty
⟦   ⇒   ⟧ty = ⟦   ⟧ty ⇒ₛ ⟦   ⟧ty

-- Unit type for empty context
data   : Set where
  unit :

-- Context Interpretation
⟦_⟧ctx : Context → SD → Set
⟦ · ⟧ctx _ =
⟦ Γ , A ⟧ctx sd = ⟦ Γ ⟧ctx sd × ⟦ A ⟧ty sd

get-var : ∀ {Γ A sd} → A ∈ Γ → ⟦ Γ ⟧ctx sd → ⟦ A ⟧ty sd
get-var Zero (_ , a) = a
get-var (Suc x) ( , _) = get-var x


-- Functorality
fmap-⇒ : ∀ {P Q sd sd'} → (P ⇒ₛ Q) sd → sd ≤ₛ sd' → (P ⇒ₛ Q) sd'
fmap-⇒   p p' x =   (≤ₛ-trans p p') x

fmap-Compl : ∀ {sd sd'} → Compl sd → sd ≤ₛ sd' → Compl sd'
fmap-Compl {sd} c (<-f f<f') = popto sd (<-f f<f') c
```

```
fmap-Compl {  f , d  } {  f , d'  } c (≤-d d≤d') =
  adjustdisp-dec ((d' - d) d≤d') (-→≤ d≤d')
    (sub I (-ₛ≡
      {n = (d' - d) d≤d'}
          (n-[n-m]≡m d≤d')) c)

fmap-L : ∀ {sd sd'} → L sd → sd ≤ₛ sd' → L sd'
fmap-L (l-var sdᵛ sdᵛ≤ₛsd) sd≤ₛsd' = l-var sdᵛ (≤ₛ-trans sdᵛ≤ₛsd sd≤ₛsd')
fmap-L (l-sbrs) _ = l-sbrs

fmap-S : ∀ {sd sd'} → S sd → sd ≤ₛ sd' → S sd'
fmap-S (s-l l) sd≤ₛsd' = s-l (fmap-L l sd≤ₛsd')
fmap-S (s-lit lit) _ = s-lit lit

fmap-A : ∀ {A sd sd'} → ⟦ A ⟧ty sd → sd ≤ₛ sd' → ⟦ A ⟧ty sd'
fmap-A {comm} = fmap-⇒ {Compl} {Compl}
fmap-A {intexp} = fmap-⇒ {Intcompl} {Compl}
fmap-A {intacc} = fmap-⇒ {Compl} {Intcompl}
fmap-A {intvar} ( e , a ) sd≤ₛsd' = ( fmap-A {intexp} e sd≤ₛsd' , fmap-A
fmap-A {A ⇒ B} = fmap-⇒ {⟦ A ⟧ty} {⟦ B ⟧ty}

fmap-Γ : ∀ {Γ sd sd'} → ⟦ Γ ⟧ctx sd → sd ≤ₛ sd' → ⟦ Γ ⟧ctx sd'
fmap-Γ { · } unit _ = unit
fmap-Γ {Γ , A} (  , a) p = fmap-Γ   p , fmap-A {A} a p


sd≤ₛsd'→sd≤ₛsd'-ₛ[d'-[suc-d]] : ∀ {sd sd'} → sd ≤ₛ sd' → (δ ≤δ  : suc (SD
sd≤ₛsd'→sd≤ₛsd'-ₛ[d'-[suc-d]] {  f , _  } {  f' , _  } (<-f f<f') _ = <-f f<
sd≤ₛsd'→sd≤ₛsd'-ₛ[d'-[suc-d]] {  f , d  } {  f , d'  } (≤-d d≤d') δ ≤δ  = ≤-d

assign : (sd : SD) → (sd' : SD) → (S ⇒ₛ Compl) sd → sd ≤ₛ sd' → R sd'
assign   f , d     f' , d'     sd≤ₛsd' r with (≤-compare {suc d} {d'})
... | leq δ ≤δ  = assign-dec ((d' - (suc d)) δ ≤δ )
                    (-→≤ δ ≤δ ) (l-var   f , d   (sd≤ₛsd'→sd≤ₛsd'-ₛ[d'-[suc-d
                    (  ((sd≤ₛsd'→sd≤ₛsd'-ₛ[d'-[suc-d]] sd≤ₛsd' δ ≤δ ))
                        (s-l (l-var   f , d   ((sd≤ₛsd'→sd≤ₛsd'-ₛ[d'-[suc-d]] sd


... | geq δ ≤δ  = assign-inc (((suc d) - d') δ ≤δ )
                    (l-var   f , d   (≤ₛ-trans sd≤ₛsd' +ₛ→≤ₛ)) r
                    (  ((≤ₛ-trans sd≤ₛsd' +ₛ→≤ₛ))
                        (s-l (l-var   f , d   ((≤ₛ-trans sd≤ₛsd' +ₛ→≤ₛ)))))

use-temp : ∀ {sd sd'} → (S ⇒ₛ Compl) sd → sd ≤ₛ sd' → R sd' → I sd'
use-temp   sd≤ₛsd' (r-s s) =   sd≤ₛsd' s
use-temp {sd} {sd'}   sd≤ₛsd' (r-unary uop s) =
  assign sd sd'   sd≤ₛsd' (r-unary uop s)
use-temp {sd} {sd'}   sd≤ₛsd' (r-binary s  bop s ) =
  assign sd sd'   sd≤ₛsd' (r-binary s  bop s )


⟦_⟧ : ∀ {Γ A} → (e : Γ ⊢ A) → (sd : SD) → ⟦ Γ ⟧ctx sd → ⟦ A ⟧ty sd
⟦ Var x ⟧ _   = get-var x
⟦ Lambda f ⟧ sd   {sd' = sd'} p a = ⟦ f ⟧ sd' (fmap-Γ   p , a)
⟦ App f e ⟧ sd   = ⟦ f ⟧ sd   (≤-d ≤-refl) (⟦ e ⟧ sd   )
⟦ Skip ⟧ _ _ _   =
⟦ Seq c  c ⟧ sd   sd' p = ⟦ c  ⟧ sd   sd' (⟦ c  ⟧ sd   sd' p)
⟦ Lit i ⟧ _ _ _   =   ≤ₛ-refl (r-s (s-lit i))
⟦ Neg e ⟧ sd   p   = ⟦ e ⟧ sd   p (use-temp   p s →   p (r-unary UNeg s))
⟦ Plus e  e ⟧ sd   p   = ⟦ e  ⟧ sd   p (use-temp (  p' s  → ⟦ e  ⟧ sd   (≤ₛ-tra
```