
Weak normalization and consistency for PicoF* using logical relations

---

Features of PicoF*:
- dependent function types
- the F* weakest precondition (WP) calculus
  - Computing WPs at the meta level and turning them into formulas
    using 2nd-order quantification and more meta-level tricks
- logical formulas and validity judgment
  - complete consistency proof, we don't assume _anything_ about
    the syntactic validity judgment!
- fancy fixpoints (let rec) with metrics and semantic termination check
- well-founded ordering on naturals, extended to expressions
- case analysis on nats (predecessor)
- subtyping (really easy)

Syntax:

    e,δ ::= x | λ(x:t).e | e_1 e_2 | O | S e | pred e eO eS | let rec (f^{δ}:t) x = e

    n ::= O | S n

    v ::= n | λ(x:t).e | let rec (f^{δ}:t) x = e

    k ::= $pred_t$

    t ∈ typ ::= nat | x:t→c

    c ∈ cmp ::= Pure t wp

    wp ∈ wpre ::= bind wp_1 >>= (x:t). wp_2 | return e
                | tot | up φ | and wp_1 wp_2 | ite φ wp_1 wp_2 | forall (x:t). wp

    φ ∈ form ::= e_1<e_2 | e_1=e_2 | $\varphi_1 \Rightarrow \varphi_2$ | $\varphi_1 \wedge \varphi_2$ | ∀x:t. φ | ∀α:k. φ | α(e)
                          = on nats only   2nd-order ∀

    Γ ::= • | Γ, x:t | Γ, α:k

    γ : exp_var -> exp   – expression substitution

Reduction relation (call by value):

    (λ(x:t).e) v ~> e[v/x]   (S-Beta)

    e_1 ~> e_1'
    ———————————— (S-App_1)
    e_1 e_2 ~> e_1' e_2

    e_2 ~> e_2'
    ———————————— (S-App_2)
    v_1 e_2 ~> v_1 e_2'

    e ~> e'
    ———————— (S-Succ)
    S e ~> S e'

    pred O e0 eS ~> e0   (S-PredZero)

```
    pred (S v) e0 eS ~> eS v   (S-PredSucc)

    e ~> e'
    ---------------------------------- (S-Pred)
    pred e e0 eS ~> pred e' e0 eS

    (let rec (f^δ:t) x = e) v ~>* e[v/x][(let rec (f^δ:t) x = e)/f] (S-LetRec)
```

```
Turning WPs to formulas (at the meta level)

    p ∈ syn_post : exp -> form  — syntactic post-condition (meta-level function)

    w2f : cmp -> syn_post -> form   — meta-level function desugaring WPs
    w2f (Pure t (bind wp₁ >>= (x:t'). wp₂)) p =
         w2f (Pure t' wp₁) (fun e -> (w2f (Pure t wp₂[e/x]) p))
    w2f (Pure t (return e)) p = p e
    w2f (Pure t tot) p = ∀x:t. p x
    w2f (Pure t (up φ)) p = φ
    w2f (Pure t (and wp₁ wp₂)) p = w2f (Pure t wp₁) p ∧ w2f (Pure t wp₂) p
    w2f (Pure t (ite φ wp₁ wp₂)) p = ( φ ⇒ w2f (Pure t wp₁) p) ∧
                                     (¬φ ⇒ w2f (Pure t wp₂) p)
    w2f (Pure t (forall (x:t'). wp)) = ∀x:t'. w2f (Pure t wp) p
```

```
Validity (Γ ⊢ φ)

    Γ ⊢ e : Pure t wp
    ------------------------------------------------------------------ (V-Construct)
    Γ ⊢ ∀α:post_t. (w2f (Pure t wp) (fun e' -> α(e'))) ⇒ α(e)

    ⊢ e₁ : nat
    ⊢ e₂ : nat
    e₁ ~>* n
    e₂ ~>* n
    ---------- (V-NatEq)
    ⊢ e₁ = e₂

    Γ ⊢ e : Pure nat tot
    -------------------- (V-EqReflE)
    Γ ⊢ e = e

    Γ ⊢ e₁ = e₂
    Γ ⊢ e₂ = e₃
    ------------ (V-EqTranE)
    Γ ⊢ e₁ = e₃

    Γ ⊢ e₁ = e₂
    ------------ (V-EqSymE)
    Γ ⊢ e₂ = e₁

    ⊢ e₁ : t₁                    closed expressions only, not very expressive
    ⊢ e₂ : t₂
    e₁ ~>* v₁
    e₂ ~>* v₂
    v₁ << v₂                     where v₁ << v₂ = { n₁ < n₂, if v₁ = n₁, v₂ = n₂
    ---------- (V-PrecedesIntro)                 { False, otherwise
    ⊢ e₁ < e₂

    Γ ⊢ e₁ < e₂
    Γ ⊢ e₂ < e₃
    ------------ (V-PrecedesTrans)
```

```
    Γ ⊢ e₁ < e₃

    Γ ⊢ φ₁    Γ ⊢ φ₂
    ──────────────── (V-AndIntro)
    Γ ⊢ φ₁ ∧ φ₂

    Γ ⊢ φ₁ ∧ φ₂
    ──────────── (V-AndElimᵢ)
    Γ ⊢ φᵢ

    Γ, φ ⊢ φ₂                          where we use formulas φ as binders to mean:
    ──────────── (V-ImplIntro)                    _:nat→Pure nat (and (up φ) tot)
    Γ ⊢ φ₁ ⇒ φ₂

    Γ ⊢ φ₁ ⇒ φ₂    Γ ⊢ φ₁
    ──────────────────── (V-ImplElim)
    Γ ⊢ φ₂

    Γ, x:t ⊢ φ
    ────────────── (V-ForallIntro)
    Γ ⊢ ∀(x:t). φ

    Γ ⊢ ∀(x:t). φ    Γ ⊢ e : t
    ──────────────────────────── (V-ForallElim)
    Γ ⊢ φ[e/x]

    Γ, φ₁ ⊢ φ₂    Γ, ¬φ₁ ⊢ φ₂                  where ¬φ = φ ⇒ false
    ──────────────────────── (V-ExMiddle)       false = 0 < 0
    Γ ⊢ φ₂

    Γ ⊢ false
    ────────── (V-FalseElim)
    Γ ⊢ φ

    Γ, α:predₜ ⊢ φ
    ──────────────── (V-ForallPIntro)
    Γ ⊢ ∀(α:predₜ). φ

    Γ ⊢ ∀(α:predₜ). φ    Γ, x:t ⊢ φ' ok
    ──────────────────────────────────── (V-ForallPElim)
    Γ ⊢ φ[(φ'[e/x])/(α(e))]
```

Sub-computation (Γ ⊢ Pure t' wp' <: Pure t wp)

```
    Γ ⊢ t' <: t
    Γ ⊢ ∀α:postₜ'. w2f (Pure t' wp') (fun e -> α(e))
                ⇐ w2f (Pure t  wp ) (fun e -> α(e))
    ──────────────────────────────────────────────── (S-Comp)
    Γ ⊢ Pure t' wp' <: Pure t wp
```

Subtyping (Γ ⊢ t₁ <: t₂)

```
    Γ ⊢ nat <: nat (Sub-Nat)

    Γ ⊢ t <: t'    Γ, x:t ⊢ Pure s' wp' <: Pure s wp
    ──────────────────────────────────────────────── (Sub-Fun)
    Γ ⊢ x:t' → Pure s' wp' <: x:t → Pure s wp
```

Expression typing (Γ ⊢ e : Pure t wp)

```
    Γ(x) = t
    ──────────────────── (T-Var)
    Γ ⊢ x : Pure t tot

    Γ, x:t ⊢ e : c
    ──────────────────────────────── (T-Abs)
    Γ ⊢ λ(x:t).e : Pure (x:t→c) tot

    Γ ⊢ e₁ : Pure (x:t₂→Pure t wp) wp₁    Γ ⊢ e₂ : Pure t₂ wp₂
    ──────────────────────────────────────────────────────────────────── (T-App)
    Γ ⊢ e₁ e₂ : Pure t[e₂/x] (bind wp₁ >>= (_:t₂→Pure t wp₁)
                              bind wp₂ >>= (x:t₂) wp)

    Γ ⊢ 0 : Pure nat tot (T-Zero)

    Γ ⊢ e : Pure nat tot
    ──────────────────────── (T-Succ)
    Γ ⊢ S e : Pure nat tot

    Γ ⊢ e : Pure nat wp    Γ ⊢ e0 : Pure t wp0
    Γ ⊢ eS : Pure (x:nat → Pure t wpS) wp'
    ──────────────────────────────────────────────────────────────────────────── (T-Pred)
    Γ ⊢ pred e e0 eS : Pure t (bind wp >>= (y:nat).
                         ite (y = 0) wp0 (bind wp' >>= _.
                                          forall (x:nat). and (up (x < e)) wpS))

    Γ ⊢ δ : Pure (x:tₓ → Pure t'' tot) tot  [only interesting case is t'' = nat]
    t = x:tₓ → Pure t' wp
    Γ, x:tₓ, f:(y:tₓ → Pure t'[y/x]
                            (and (up (δ y < δ x)) wp[y/x])) ⊢ e : Pure t' wp
    ──────────────────────────────────────────────────────────────────────────── (T-Fix)
    Γ ⊢ let rec (fᵟ:t) x = e : Pure t tot

    Γ ⊢ e : Pure t tot
    ──────────────────────── (T-Ret)
    Γ ⊢ e : Pure t (return e)

    Γ ⊢ e : c    Γ ⊢ c <: c'
    ──────────────────────── (T-Sub)
    Γ ⊢ e : c'
```

Logical relation:

```
    π ∈ sem_post : val -> Prop  — semantic post-condition (meta-level predicate)
      where we additionally require that the πs are invariant under
      reduction or expansion within values (lambdas), formally:
        e₁ <~> e₂ ⇒ π v[e₁/x] ⇔ π v[e₁/x]

    σ : typ_var -> sem_post     — semantic type variable substitution
      under the same invariance constrains as above

    E⟦c, σ⟧ = { e | ∀π∈W⟦c, σ⟧. ∃v. e ~>* v  ∧  v ∈ V⟦typ(c),σ⟧  ∧  π v }
                     where typ(Pure t wp) = t and wp(Pure t wp) = wp
    — intuitive reading: E⟦c, σ⟧ is the set of all expressions e so
      that for all (semantic) post-conditions π for which the
      pre-condition wp(c) e reduces to a value v of type typ(c) and
      such that π v holds

    V⟦nat,σ⟧ = {n}
    V⟦x:t→c,σ⟧ = { λx.e | ∀v∈V⟦t,σ⟧. e[v/x] ∈ E⟦c[v/x], σ⟧ }
              ∪ { let rec (fᵟ:_) x = e |
```

```
                ∀v ∈ V⟦t⟧. e[v/x][(let rec (f^δ:_) x = e)/f] ∈ E⟦c[v/x], σ⟧ }

W⟦ Pure t (return e), σ ⟧ = { π | ∀v. e ~>* v ⇒ π v }
W⟦ Pure t (bind wp₁ >>= (x:t'). wp₂), σ ⟧ =
  { π | W⟦Pure t' wp₁, σ⟧ ∋ (fun v -> π∈W⟦Pure t wp₂[v/x],σ⟧) }
W⟦ Pure t tot, σ ⟧ = { π | ∀v∈V⟦t,σ⟧. π v }
W⟦ Pure t (up φ), σ ⟧ = { π | P⟦φ, σ⟧ }
W⟦ Pure t (and wp₁ wp₂), σ ⟧ = W⟦Pure t wp₁,σ⟧ ∩ W⟦Pure t wp₂,σ⟧
W⟦ Pure t (ite φ wp₁ wp₂), σ ⟧ = { π | (P⟦φ, σ⟧ ⇒ π ∈ W⟦Pure t wp₁, σ⟧) ∧
                                       (¬P⟦φ, σ⟧ ⇒ π ∈ W⟦Pure t wp₂, σ⟧) }
W⟦ Pure t (forall (x:t'). wp) ⟧ = { π | ∀v∈V⟦t',σ⟧. π ∈ W⟦Pure t wp[v/x], σ⟧ }
– a semantic translation of WPs to predicates; intuitive reading:
  W⟦ c, σ ⟧ is the set of all post-conditions for which wp(c) holds
– W can also be obtained from w2f and P, but need to define it this
  way to ensure termination of the logical relation

P⟦ e₁ < e₂, σ ⟧ = ∃v₁∃v₂. e₁ ~>* v₁ ∧ e₂ ~>* v₂ ∧ v₁ << v₂
P⟦ e₁ = e₂, σ ⟧ = ∃n. e₁ ~>* n ∧ e₂ ~>* n
P⟦ φ₁ ⇒ φ₂, σ ⟧ = P⟦φ₁, σ⟧ ⇒ P⟦φ₂, σ⟧
P⟦ φ₁ ∧ φ₂, σ ⟧ = P⟦φ₁, σ⟧ ∧ P⟦φ₂, σ⟧
P⟦ ∀x:t. φ, σ ⟧ = ∀v∈V⟦t, σ⟧. P⟦φ[v/x], σ⟧
P⟦ α(e), σ ⟧    = ∀v. e ~>* v ⇒ σ(α)(v)
P⟦ ∀α:pred_t. φ, σ ⟧ = ∀π. P⟦ φ, σ[α↦π] ⟧

Note: These functions are defined by mutual recursion, where
      we decrease the lex tuple %[size(t), ξ] or %[size(c), ξ]
      or %[size(φ), ξ], where ξ ∈ {C,V,W,P} and W < C

  size(nat) = 0
  size(x:t→c) = 1 + size(t) + size(c)

  size(Pure t wp) = 1 + size(t) + size(wp)

          ↓ we have ∀γ. size(γ(X)) = size(X)
  size(return e) = size(tot) = 1                    ↓ need to count this!
  size(bind wp₁ >>= (x:t'). wp₂) = 1 + size(wp₁) + size(t') + size(wp₂)
  size(up φ) = 1 + size(φ)
  size(and wp₁ wp₂) = 1 + size(wp₁) + size(wp₂)
  size(ite φ wp₁ wp₂) = 1 + size(φ) + size(wp₁) + size(wp₂)
  size(forall (x:t). φ) = 1 + size(t) + size(φ)

        we have ∀γ. size(γ(X)) = size(X)
        ↓                 ↓          ↓
  size(e₁ < e₂) = size(e₁ = e₂) = size(α(e)) = 1
  size(φ₁ ⇒ φ₂) = size(φ₁ ∧ φ₂) = 1 + size(φ₁) + size(φ₂)
  size(∀x:t. φ) = 1 + size(t) + size(φ)
  size(∀α:pred_t. φ) = 1 [+ size(t)] + size(φ)

G⟦Γ⟧ = { (γ, σ) | dom(γ) = domx(Γ)  ∧  ∀x ∈ domx(Γ). γ(x) ∈ V⟦γ(Γ(x))⟧
                  dom(σ) = domα(γ) }

Note: we have that ∀γ. size(γ(t)) = size(t) ∧ size(γ(φ)) = size(φ)
      since expressions don't influence the size; otherwise
      V⟦x:t₁→t₂⟧ calls E⟦t₂[v/x]⟧ on something larger (if this
      ever becomes untrue we can look at Trellys/Zombie for a nice solution)

Note: Here we only work with closed expressions, values, types, formulas
      (closed with respect to expression variables).
- So "∀v ∈ V⟦t₁⟧ ..." above should be read as
  "∀v closed values so that v ∈ V⟦t₁⟧ ..."
```

- "∃v_1∃v_2 ..." should be read as "∃v_1 and v_2 closed values"
- "∀γ ∈ G⟦Γ⟧" should be read as "∀γ closed value substitutions ..."

Definition (Semantic "Judgments"):
  Γ ⊨ φ = ∀(γ,σ) ∈ G⟦Γ⟧. P⟦γ(φ), σ⟧
  Γ ⊨ t_1 <: t_2 = ∀(γ,σ) ∈ G⟦Γ⟧. V⟦γ(t_1), σ⟧ ⊆ V⟦γ(t_2), σ⟧
  Γ ⊨ c_1 <: c_2 = Γ ⊨ typ(c_1) <: typ(c_2) ∧ ∀(γ,σ) ∈ G⟦Γ⟧. W⟦γ(c_1),σ⟧ ⊇ W⟦γ(c_2),σ⟧
  Γ ⊨ e : c = ∀(γ,σ) ∈ G⟦Γ⟧. γ(e) ∈ E⟦γ(c), σ⟧

Lemma (Determinism): ∀e e_1 e_2. e ~> e_1 ∧ e ~> e_2 ⇒ e_1 = e_2

Lemma (Expand): e ~>* e' and e' ∈ E⟦t⟧ then e ∈ E⟦t⟧
Proof: By the definition of E⟦t⟧ and the transitivity of ~>* □

Lemma (Substitutivity of reduction): ∀e,e'. e~>e' ⇒ ∀γ. γ(e) ~> γ(e')
Proof: Straightforward induction on typing derivations, using the fact
       that substitution cannot turn values into non-values
       (needed with CBV) □

Definition (Evaluation Contexts):
  E ::= [] | E e_2 | v_1 E | S E | pred E e0 eS

Property (Substitution preserves evaluation contexts):
  If E is an evaluation, so is γ(E), for any substitution γ.

Lemma (Context Auxiliary)
  ∀e_1 closed. ∀e. fv(e)={x} ⇒ e[e_1/x] ~> e'' ⇒
     (a) ∃e'. e ~> e' (open reduction, no x involved in evaluation)
  or (b) ∃E. e = E[x] (one x is in evaluation position)
  or (c0) ∃E. e = E[(λ(y:t).eb) x] and e_1=v (one x creates new redex; CBV)
  or (c1) ∃E. e = E[x v] (one x creates new redex)
     and   (c1.1) e_1 = λ(y:t). e_1'
         or (c1.2) e_1 = let rec (f^δ:t) y = e_1'
  or (c2) ∃E. e = E[x x] and e_1=(λ(y:t).eb) (one x creates new redex)
  or (c3) ∃E. e = E[pred x e0 eS] and e_1=n (one x creates new redex)
Proof:
  by induction on the derivation of e[e_1/x] ~> e''
  Common Subcase below: e=x, E=[] – choose (b)
  (S-Beta) e[e_1/x] = (λ(y:t).eb) v and e'' = eb[v/y]
    Subcase e=(λ(y:t).eb) x, e_1 = v – choose E=[] in (c0)
    Subcase e=x v, e_1 = (λ(y:t).eb) – choose E=[] in (c1.1)
    Subcase e=x x, e_1 = (λ(y:t).eb) – choose E=[] in (c2)
    Subcase e=e^1 e^2 where neither e^1 nor e^2 are variables
      Have e^1[e_1/x] = (λ(y:t).eb)
        which implies e^1=(λ(y:t').eb') and eb=eb'[e_1/x] and t=t'[e_1/x]
        so e'' = eb'[e_1/x][v/y]
      Have e^2[e_1/x] = v, which implies e^2 is also a value (can't be variable)
      Choose e' = eb'[e^2/y] and show (a): e~>e'
        (λ(y:t').eb') e^2 ~> eb'[e^2/y] by (S-Beta)
  (S-App$_1$) e[e_1/x] = e^1' e^2', e^1' ~> e^1'', e'' = e^1'' e^2'
  Subcase e=e^1 e^2, e^1'=e^1[e_1/x], e^2'=e^2[e_1/x],
          e[e_1/x] = e^1[e_1/x] e^2[e_1/x], e'' = e^1'' e^2[e_1/x]
    By IH for e^1[e_1/x] ~> e^1'' we get 5 sub-subcases:

```
    (a) ∃e¹*. e¹'' = e¹*[e₁/x] and e¹ ~> e¹*
        we chose e'=e¹* e² and show that
        (1) e'' = e¹'' e²[e₁/x] = e¹*[e₁/x] e²[e₁/x] = (e¹* e²)[e₁/x]
        (2) e = e¹ e² ~> e¹* e² by (S-App₁)
    (b) ∃E. e¹ = E[x]
        we show (b) by picking E'=E e² so that e = E'[x] since e¹ e² = E[x] e²
    (c0) ∃E. e¹ = E[(λ(y:t).eb) x] and e₁=v
         we show (c0) by picking E'=E e² so that e = E'[(λ(y:t).eb) x]
    (c1) ∃E. e = E[x v] and    (c1.1) e₁ = λ(y:t). e₁'
                           or (c1.2) e₁ = let rec (fᵟ:t) y = e₁'
         we show (c1) by picking E'=E e² so that e = E'[x v]
    (c2) ∃E. e¹ = E[x x] and e₁=(λ(y:t).eb)
         we show (c2) by picking E'=E e² so that e = E'[x x]
    (c3) ∃E. e = E[pred x e0 eS] and e₁=n
         we show (c3) by picking E'=E e² so that e = E'[pred x e0 eS]
  All other inductive cases (context rules) are exactly the same as (S-App₁).
  Cases (S-PredZero), (S-PredSucc), and (S-LetRec) are similar to (S-Beta) □
```

```
Lemma (Reduce in Context): ∀e₁ and e₂ closed. ∀e. fv(e)={x} ⇒
  e₁ ~> e₂ ∧ e[e₁/x] ~>* v₁ ⇒ e[e₂/x] ~>* v₂ ∧ ∃v. v₁ = v[e₁/x] ∧ v₂ = v[e₂/x]
Proof.
  By induction on ~>*.
  Case e[e₁/x] = v₁ ⇒ e and e[e₂/x] are also values
    (e can't be a variable since then e₁ is a value so irreducible)
    pick v₂=e[e₂/x] and v=e to show the conclusion
  Case e[e₁/x]~>e'' and e'' ~>* v₁
    by (Context Auxiliary) lemma above:
    Subcase (a):  ∃e'. e ~> e'
      by (Substitutivity of reduction) lemma e[e₂/x] ~> e'[e₂/x]
      and e[e₁/x] ~> e'[e₁/x] so by determinism e''= e'[e₁/x]
      by IH: e'[e₂/x] ~>* v₂ ∧ ∃v. v₁ = v[e₁/x] ∧ v₂ = v[e₂/x]
      the first gives us by transitivity: e[e₂/x] ~>* v₂
    Subcase (b): ∃E. e = E[x]
      by context rules and determinism: e'' = (E[e₁/x])[e₂]
                                            = (E[e₂])[e₁/x]  since e₂ closed
      by IH: (E[e₂])[e₂/x] ~>* v₂ ∧ ∃v. v₁ = v[e₁/x] ∧ v₂ = v[e₂/x]
           = e[e₂/x]
    All (cX) subcases are contradictory because they would require e₁
    to be a value, thus irreducible, but we already have e₁ ~> e₂. □
```

```
Corollary (Reduce in Context for nats): ∀e₁ and e₂ closed. ∀e. fv(e)={x} ⇒
  e₁ ~> e₂ ∧ e[e₁/x] ~>* n ⇒ e[e₂/x] ~>* n
```

```
Lemma (Expand in Context): ∀e₁ and e₂ closed. ∀e. fv(e)={x} ⇒
  e₂ ~> e₁ ∧ e[e₁/x] ~>* v₁ ⇒ e[e₂/x] ~>* v₂ ∧ ∃v. v₁ = v[e₁/x] ∧ v₂ = v[e₂/x]
Proof.
  By well-founded induction on the lexicographic ordering of:
  - the length of the e[e₁/x] ~>* v₁ derivation
  - the number of occurrences of x in e.
  Case e[e₁/x] = v₁
    Subcase e = x, e₁=v₁, e[e₂/x]=e₂ ~> e₁ = v₁,
      so we pick v=v₁ and since it is closed it satisfies the conclusion
    Subcase e is a value, and so is e[e₂/x] ~>* e[e₂/x]
      we then pick v=e to show the conclusion
  Case e[e₁/x]~>e'' and e'' ~>* v₁
    by (Context Auxiliary) lemma above:
    Subcase (a) ∃e'. e ~> e'
      by (Substitutivity of reduction) lemma e[e₂/x] ~> e'[e₂/x]
      and e[e₁/x] ~> e'[e₁/x] so by determinism e''= e'[e₁/x]
```

```
      by IH: e'[e₂/x] ~>* v₂ ∧ ∃v. v₁ = v[e₁/x] ∧ v₂ = v[e₂/x]
      the first gives us by transitivity e[e₂/x] ~>* v₂
    Subcase (b) ∃E. e = E[x]
      By IH on (E[e₁])[e₁/x] = e[e₁/x] ~>* v₂ ∧ ∃v. v₁ = v[e₁/x] ∧ v₂ = v[e₂/x]
      we get (E[e₁])[e₂/x] ~>* v₂
      e[e₂/x] = (E[e₂/x])[e₂] ~> (E[e₂/x])[e₁] = (E[e₁])[e₂/x]
      by transitivity: e[e₂/x] ~> (E[e₁])[e₂/x] ~>* v₂
    Subcase (c0, S-Beta) e = E[(λ(y:t).eb) x] and e₁=v
      e[e₁/x] = (E[(λ(y:t).eb) x])[e₁/x]
              = (E[e₁/x])[(λ(y:t[e₁/x]).eb[e₁/x]) e₁]
      e'' = (E[e₁/x])[eb[e₁/x][v/y]] = (E[eb[x/y]])[e₁/x]
      by IH (E[eb[x/y]])[e₂/x] ~>* v₂ ∧ ∃v. v₁ = v[e₁/x] ∧ v₂ = v[e₂/x]
      e[e₂/x] = (E[(λ(y:t).eb) x])[e₂/x]
              = (E[e₂/x])[(λ(y:t[e₂/x]).eb[e₂/x]) e₂]
      by (S-Beta) + context rules: e[e₂/x] ~> (E[e₂/x])[eb[e₂/x][v/y]]
                                           = (E[eb[x/y]])[e₂/x]
      by transitivity: e[e₂/x] ~> (E[eb[x/y]])[e₂/x] ~>* v₂
    Subcase (c1.1, S-Beta) e = E[x v] and e₁ = λ(y:t). e₁'
      e[e₁/x] = (E[e₁/x])[e₁ v[e₁/x]]
      by S-Beta e'' = (E[e₁/x])[e₁'[v[e₁/x]/y]] = (E[e₁'[v/y]])[e₁/x]
      by IH (E[e₁'[v/y]])[e₂/x] ~>* v₂ ∧ ∃v. v₁ = v[e₁/x] ∧ v₂ = v[e₂/x]
      e[e₂/x] = (E[e₂/x])[e₂ v[e₂/x]]
             ~> (E[e₂/x])[e₁ v[e₂/x]]      by e₁ ~> e₂ + context rules
             ~> (E[e₂/x])[e₁'[v[e₂/x]/y]]  by (S-Beta) + context rules
              = (E[e₁'[v/y]])[e₂/x]
      by transitivity e[e₂/x] ~>₂ (E[e₁'[v/y]])[e₂/x] ~>* v₂
    Subcase (c1.2, S-LetRec) e = E[x v] and e₁ = let rec (fᵟ:t) y = e₁'
      similar to (c1.1, S-Beta) above
    Subcase (c2, S-Beta) e = E[x x] and e₁ = λ(y:t). e₁'
      e[e₁/x] = (E[e₁/x])[e₁ e₁]
      by S-Beta e'' = (E[e₁/x])[e₁'[e₁/y]] = (E[e₁'[x/y]])[e₁/x]
      by IH (E[e₁'[x/y]])[e₂/x] ~>* v₂ ∧ ∃v. v₁ = v[e₁/x] ∧ v₂ = v[e₂/x]
      e[e₂/x] = (E[e₂/x])[e₂ e₂]
             ~> (E[e₂/x])[e₁ e₂]      by e₁ ~> e₂ + context rules
             ~> (E[e₂/x])[e₁ e₁]      by e₁ ~> e₂ + context rules
             ~> (E[e₂/x])[e₁'[e₁/y]]  by (S-Beta) + context rules
              = (E[e₁'[x/y]])[e₂/x]]
      by transitivity e[e₂/x] ~>₂ (E[e₁'[v/y]])[e₂/x] ~>* v₂
    Subcase (c3.1, S-PredZero) e = E[pred x e0 eS] and v=0
      similar to (c1.1, S-Beta) above
    Subcase (c3.2, S-PredSucc) e = E[pred x e0 eS] and v=S n
      similar to (c1.1, S-Beta) above  □

─────────────────────────────────────────────────────────────────────────────

Corollary (Expand in Context for nats): ∀e₁ and e₂ closed. ∀e. fv(e)={x} ⇒
  e₂ ~> e₁ ∧ e[e₁/x] ~>* n ⇒ e[e₂/x] ~>* n

─────────────────────────────────────────────────────────────────────────────

Lemma (Interpretations closed under reduction and expansion):
  ∀e₁ and e₂ closed expressions. e₁ ∈ E⟦c,σ⟧ ⇒ e₂ ∈ E⟦c,σ⟧ ⇒
  e₁ ~> e₂ ⇒ (1) ∀c. fv(c)={x} ⇒ (E⟦c[e₁/x],σ⟧ = E⟦c[e₂/x],σ⟧)
           ∧ (2) ∀t. fv(t)={x} ⇒ (V⟦t[e₁/x],σ⟧ = V⟦t[e₂/x],σ⟧)
           ∧ (3) ∀φ. fv(φ)={x} ⇒ (P⟦φ[e₁/x],σ⟧ ⇔ P⟦φ[e₂/x],σ⟧)
           ∧ (4) ∀c. fv(c)={x} ⇒ (W⟦c[e₁/x],σ⟧ = W⟦c[e₂/x],σ⟧)
Proof. by induction the sizes
  (1) Have: ∀π∈W⟦c[e₁/x], σ⟧. ∃v. e ~>* v  ∧  v ∈ V⟦typ(c[e₁/x]),σ⟧  ∧  π v
      Let π∈W⟦c[e₂/x], σ⟧, by IH(4) we get π∈W⟦c[e₁/x], σ⟧
      so ∃v. e ~>* v  ∧  v ∈ V⟦typ(c[e₁/x]),σ⟧  ∧  π v
      Take same v and need to show v ∈ V⟦t[e₂/x],σ⟧, this follows from IH(2)
  (2) Case t = nat. trivial
      Case t = y:t₁→c₂ and v=λ(y:_).e.
```

```
        Have ∀v' ∈ V⟦t₁[e₁/x],σ⟧. e[v'/y] ∈ E⟦c₂[e₁/x][v'/y],σ⟧
        Let v' ∈ V⟦t₁[e₂/x],σ⟧. To show: e[v'/y] ∈ E⟦c₂[e₂/x][v'/y],σ⟧
        By IH part (2⇐): v' ∈ V⟦t₁[e₁/x],σ⟧ so also e[v'/y] ∈ E⟦c₂[e₁/x][v'/y],σ⟧
        By IH part (1⇒): e[v'/y] ∈ E⟦c₂[e₂/x][v'/y],σ⟧.
      Case t = y:t₁→t₂ and v=let rec — similar to previous case
(3) Case φ=f₁ < f₂, to show P⟦f₁[e₁/x] < f₂[e₁/x],σ⟧ ⇔ P⟦f₁[e₂/x] < f₂[e₂/x],σ⟧
        From e₁ ∈ E⟦t,σ⟧ we get e₁ ~>* v₁, from e₂ ∈ E⟦t,σ⟧ we get e₂ ~>* v₂
        (⇒) direction:
          Assume P⟦f₁[e₁/x] < f₂[e₁/x],σ⟧.
          By definition of P: f₁[e₁/x] ~>* v₁  and f₂[e₁/x] ~>* v₂  and v₁ << v₂
          By definition of <<: v₁ = n₁, v₂ = n₂ so
                               f₁[e₁/x] ~>* n₁  and f₂[e₁/x] ~>* n₂  and n₁  < n₂
          To show:             f₁[e₂/x] ~>* n₁' and f₂[e₂/x] ~>* n₂' and n₁' < n₂'
          Seems that the only way of showing this is showing that
            f₁[e₂/x] ~>* n₁ and f₁[e₂/x] ~>* n₁
          So it all boils down to (Reduce in Context for nat) lemma:
            e₁ ~> e₂ ∧ e[e₁/x] ~>* n ⇒ e[e₂/x] ~>* n
        (⇐) direction: by (Expand in Context for nat) lemma
      Case φ=f₁ = f₂ — similar to previous case
      Case φ = φ₁ ⇒ φ₂ and φ = φ₁ ∧ φ₂ — trivial from IH
      Case φ = ∀(y:ty).φ
        to show (P⟦∀(y:ty[e₁/x]). φ[e₁/x],σ⟧ ⇔ P⟦∀(y:ty[e₂/x]). φ[e₂/x],σ⟧)
        by definition of P to show:
          ∀v∈V⟦ty[e₁/x]),σ⟧. P⟦φ[e₁/x][v/y],σ⟧
        ⇔ ∀v∈V⟦ty[e₂/x]),σ⟧. P⟦φ[e₂/x][v/y],σ⟧
        by IH (part 1): V⟦ty[e₁/x]),σ⟧ = V⟦ty[e₂/x]),σ⟧
        by IH (part 2): P⟦(φ[v/y])[e₁/x],σ⟧ ⇔ P⟦(φ[v/y])[e₂/x],σ⟧  □
      Case φ=α(e)
        to show (∀v. e[e₁/x] ~>* v ⇒ σ(α)(v)) ⇔ (∀v. e[e₂/x] ~>* v ⇒ σ(α)(v))
        this follows from (Reduce/Expand in Context) and the constraint we make
        on πs that they are invariant under reduction/expansion within values
      Case φ=∀α:predₜ. φ, σ
        to show (∀π. P⟦ φ[e₁/x], σ[α↦π] ⟧) ⇔ (∀π. P⟦ φ[e₂/x], σ[α↦π] ⟧)
        this follows by IH(3)
(4) Case c = Pure t (return e)
        to show ∀π. (∀v. e[e₁/x] ~>* v ⇒ π v) ⇔ (∀v. e[e₂/x] ~>* v ⇒ π v)
        this follows from (Reduce/Expand in Context) and the constraint we make
        on πs that they are invariant under reduction/expansion within values
      The other cases are easy by induction.
```

```
Property (Relating P+w2f and W; take 1) ∀c ∀p ∀σ.
  P⟦ w2f c p, σ ⟧ ⇔ Π⟦p, σ⟧ ∈ W⟦ c, σ ⟧, where Π⟦p, σ⟧ = (fun v -> P⟦p v, σ⟧)

Proof. By induction on the size of c.

  Case: c = Pure t tot
    P⟦ ∀x:t. p x, σ ⟧ ⇔ ∀v∈V⟦t,σ⟧. P⟦p v, σ⟧  — trivial from definitions

  Case: c = Pure t (up φ)
    it is trivial that: P⟦φ, σ⟧ ⇔ P⟦φ, σ⟧

  Case: c = Pure t (and wp₁ wp₂)
    to show: P⟦ w2f (Pure t wp₁) p ⟧ ∧ P⟦ w2f (Pure t wp₂) p ⟧
           ⇔ Π⟦p, σ⟧ ∈ W⟦Pure t wp₁,σ⟧ ∩ W⟦Pure t wp₂,σ⟧
    this follows directly from IH

  Case: c = Pure t (return e)
    to show: P⟦ p e, σ ⟧ ⇔ (∀v. e ~>* v ⇒ P⟦p v, σ⟧)
    this follows from P closed under reduction/expansion lemma
```

```
  Case: c = Pure t (bind wp₁ >>= (x:t'). wp₂)
      P⟦ w2f c p, σ ⟧
    ⇔ P⟦ w2f (Pure t' wp₁) (fun e -> (w2f (Pure t wp₂[e/x]) p)), σ ⟧
      [by induction hypothesis]
    ⇔ Π⟦fun e -> w2f (Pure t wp₂[e/x]) p, σ⟧ ∈ W⟦Pure t' wp₁, σ⟧
    ⇔ (fun v -> P⟦w2f (Pure t wp₂[v/x]) p, σ⟧) ∈ W⟦Pure t' wp₁, σ⟧
    = W⟦Pure t' wp₁, σ⟧ ∋ (fun v -> π∈W⟦Pure t wp₂[v/x],σ⟧)
    ⇔ Π⟦p, σ⟧ ∈ W⟦ c, σ ⟧

  Case: c = Pure t (ite wp wp0 wpS)
      P⟦ w2f c p, σ ⟧
    ⇔ P⟦ (φ ⇒ w2f (Pure t wp₁) p) ∧ (¬φ ⇒ w2f (Pure t wp₂) p), σ ⟧
    ⇔ P⟦φ, σ⟧ ⇒ P⟦w2f (Pure t wp₁) p, σ⟧ ∧ (¬P⟦φ, σ ⟧ ⇒ P⟦w2f (Pure t wp₂) p, σ⟧
    [by induction hypothesis]
    ⇔ (P⟦φ, σ⟧ ⇒ Π⟦p, σ⟧ ∈ W⟦ Pure t wp₁, σ ⟧) ∧
     (¬P⟦φ, σ ⟧ ⇒ P⟦w2f (Pure t wp₂) p, σ⟧)
    ⇔ Π⟦p, σ⟧ ∈ W⟦ c, σ ⟧, done

  Case: c = Pure t (forall (x:t). wp)
      P⟦ w2f c p, σ ⟧
    ⇔ P⟦∀x:t'. w2f (Pure t wp) p⟧
    ⇔ ∀v∈V⟦t,σ⟧. P⟦w2f (Pure t wp[v/x]) p⟧
      [by induction hypothesis]
    ⇔ ∀v∈V⟦t,σ⟧. Π⟦p, σ⟧ ∈ W⟦Pure t' wp[v/x], σ⟧
    ⇔ Π⟦p, σ⟧ ∈ W⟦ c, σ ⟧, done □

Property (Relating P+w2f and W; take 2) ∀c ∀σ ∀α ∉ dom(σ) ∀π.
  P⟦w2f c (fun e' -> α(e')), σ[α↦π]⟧ ⇔ π ∈ W⟦ c, σ ⟧
Proof:
    P⟦w2f c (fun e' -> α(e')), σ[α↦π]⟧
  [by previous property]
  ⇔ Π⟦(fun e' -> α(e')), σ[α↦π]⟧ ∈ W⟦ c, σ ⟧
  ⇔ π ∈ W⟦ c, σ ⟧ □
```

```
Lemma (Dependent Application): ∀e₁ and e₂ closed expressions
                               ∀closed types x:t₂→Pure t wp,
                                and formulas wp₁, wp₂
  e₁ ∈ E⟦Pure (x:t₂→Pure t wp) wp₁⟧ ∧ e₂ ∈ E⟦Pure t₂ wp₂⟧ ⇒
  (e₁ e₂) ∈ E⟦Pure t[e₂/x] (bind wp₁ >> bind wp₂ >>= (x:t₂) wp)⟧
Proof:
  from 1st hyp: ∀π∈W⟦Pure (x:t₂→Pure t wp) wp₁, σ⟧.                    (H1)
                ∃v₁. e₁ ~>* v₁ ∧ v₁ ∈ V⟦x:t₂→Pure t wp,σ⟧ ∧ π v₁
  from 2nd hyp: ∀π∈W⟦Pure t₂ wp₂, σ⟧.                                 (H2)
                ∃v₂. e₂ ~>* v₂ ∧ v₂ ∈ V⟦t₂,σ⟧ ∧ π v₂
  to show: (e₁ e₂) ∈ E⟦Pure t[e₂/x] (bind wp₁ >> bind wp₂ >>= (x:t₂) wp)⟧
        ⇔ ∀π∈W⟦Pure t[e₂/x] (bind wp₁ >> bind wp₂ >>= (x:t₂) wp), σ⟧.
            ∃v. (e₁ e₂) ~>* v  ∧  v ∈ V(⟦t[e₂/x],σ⟧)  ∧  π v
        ⇔ ∀π. W⟦Pure (x:t₂→Pure t wp) wp₁, σ⟧ ∋ (fun _ ->
                  W⟦Pure t₂ wp₂, σ⟧ ∋ (fun x ->
                    π ∈ W⟦Pure t[e₂/x] wp, σ⟧)) ⇒
            ∃v. (e₁ e₂) ~>* v  ∧  v ∈ V(⟦t,σ⟧)  ∧  π v
    Let π so that W⟦Pure (x:t₂→Pure t wp) wp₁, σ⟧ ∋ (fun _ ->
                  W⟦Pure t₂ wp₂, σ⟧ ∋ (fun x ->
                    π ∈ W⟦Pure t[e₂/x] wp, σ⟧))
```

```
    by (H1) we get ∃v₁. e₁ ~>* v₁ ∧ v₁ ∈ V⟦x:t₂→Pure t wp,σ⟧                (H1')
                ∧ W⟦Pure t₂ wp₂, σ⟧ ∋ (fun x ->
                     π ∈ W⟦Pure t[e₂/x] wp, σ⟧)
    by (H2) we get ∃v₂. e₂ ~>* v₂ ∧ v₂ ∈ V⟦t₂,σ⟧                            (H2')
                ∧ π ∈ W⟦Pure t[e₂/x] wp[v₂/x], σ⟧                          (H2'')
    by (W closed under reduction) lemma: π ∈ W⟦Pure t[v₂/x] wp[v₂/x], σ⟧
  By definition of v₁ ∈ V⟦x:t₂→Pure t wp,σ⟧ we get two very similar sub-cases:
  Subcase: v₁=λ(x:_).e
       and ∀v ∈ V⟦t₂,σ⟧ ⇒ e[v/x] ∈ E⟦Pure t[v/x] wp[v/x],σ⟧
    Putting this together with (H2') we get:
      e[v₂/x] ∈ E⟦Pure t[v₂/x] wp[v₂/x],σ⟧
    By instantiating the definition of E
    with π∈W⟦Pure t[v₂/x] wp[v₂/x], σ⟧ (H2'') we get:
      ∃v. e[v₂/x] ~>* v  ∧  v ∈ V⟦t[v₂/x],σ⟧  ∧  π v
    We choose the same v and construct the following reduction
            e₁ e₂ ~>* (λ(x:_).e) e₂
                  ~>* (λ(x:_).e) v₂
                  ~>  e[v₂/x]
                  ~>* v
    We have v ∈ V⟦t[v₂/x],σ⟧ and e₂ ~>* v₂, so by repeated
    applications of the lemma (V closed under expansion): v ∈ V⟦t[e₂/x],σ⟧
  Subcase: v₁=(let rec (fᵟ:_) x = e) – very similar  □
```

```
Lemma (Delayed substitution):
  P⟦ φ, σ[α↦(fun v -> P⟦ φ'[v/x], σ ⟧)] ⟧ ⇔ P⟦ φ[(φ'[e/x])/(α(e))], σ ⟧
```

```
Lemma (Monotonicity of W with respect to π)
  (∀v. π₁ v ⇒ π₂ v) ∧ π₁ ∈ W⟦c,σ⟧ ⇒ π₂ ∈ W⟦c,σ⟧
[this seems very much related to monotonicity of WP]
Proof: by induction on the size of c:
  Case c = Pure t (return e), W⟦c,σ⟧ = { π | ∀v. e ~>* v ⇒ π v }
    have: ∀v. e ~>* v ⇒ π₁ v, which implies ∀v. e ~>* v ⇒ π₂ v
  Case c = Pure t (bind wp₁ >>= (x:t'). wp₂)
    (fun v -> π₁∈W⟦Pure t wp₂[v/x],σ⟧) ∈ W⟦Pure t' wp₁, σ⟧
    by IH: ∀v. π₁∈W⟦Pure t wp₂[v/x],σ⟧ ⇒ π₂∈W⟦Pure t wp₂[v/x],σ⟧
    by IH: (fun v -> π₁∈W⟦Pure t wp₂[v/x],σ⟧) ∈ W⟦Pure t' wp₂, σ⟧
  Case c = Pure t tot
    ∀v∈V⟦t,σ⟧. π₁ v ⇒ ∀v∈V⟦t,σ⟧. π₂ v
  Case c = Pure t (up φ) – trivial
  Case c = Pure t (and wp₁ wp₂) – by induction
  Case c = Pure t (ite φ wp₁ wp₂) – by induction
  Case c =  Pure t (forall (x:t). wp) – by induction  □
```

```
Theorem (Soundness):
  Γ ⊢ φ       ⇒  Γ ⊨ φ
  Γ ⊢ t₁ <: t₂ ⇒  Γ ⊨ t₁ <: t₂
  Γ ⊢ c₁ <: c₂ ⇒  Γ ⊨ c₁ <: c₂
  Γ ⊢ e : c    ⇒  Γ ⊨ e : c

Proof: by mutual induction on derivations.

(V-Construct) Γ ⊢ e : c
  to show: P⟦∀α:predₜ. (w2f γ(c) (fun e' -> α(e'))) ⇒ α(γ(e))⟧
         ⇔ ∀π. P⟦w2f γ(c) (fun e' -> α(e')), σ[α↦π]⟧ ⇒ P⟦α(γ(e)), σ[α↦π]⟧
```

```
          ⇔ ∀π. P⟦w2f γ(c) (fun e' -> α(e')), σ[α↦π]⟧ ⇒ (∀v. γ(e) ~>* v ⇒ π v)
        [by lemma (Relating P+w2f and W; take 2)]
          ⇔ ∀π. π ∈ W⟦γ(c), σ⟧ ⇒ (∀v. γ(e) ~>* v ⇒ π v)
    By IH: γ(e) ∈ E⟦γ(c), σ⟧,
    so ∀π∈W⟦γ(c), σ⟧. ∃v. γ(e) ~>* v  ∧  v ∈ V⟦typ(γ(c)),σ⟧ ∧  π v
    By determinism: ∀π. π ∈ W⟦γ(c), σ⟧ ⇒ (∀v. γ(e) ~>* v ⇒ π v), done

  (V-NatEq) φ = (e₁ = e₂), e₁ ~>* n, e₂ ~>* n
    from this we can immediately show P⟦e₁ = e₂, ∅⟧

  (V-EqReflE) Γ ⊢ e : Pure nat tot
    by IH: e ∈ E⟦Pure nat tot⟧, so ∃n. e~>*n

  (V-EqTranE)
    have: P⟦e₁ = e₂, σ⟧ ∧ P⟦e₂ = e₃, σ⟧
    have: (∃n. e₁ ~>* n ∧ e₂ ~>* n) ∧ (∃n. e₂ ~>* n' ∧ e₃ ~>* n')
    by determinism: n = n' and we get P⟦e₁ = e₃, σ⟧

  (V-EqSymE) trivial

  (V-PrecedesIntro) φ = e₁ < e₂, e₁ ~>* v₁, e₂ ~>* v₂, and v₁ << v₂
    P⟦ e₁ < e₂ ⟧ is immediate from these 3 conditions

  (V-PrecedesTrans) φ = e₁ < e₃ and Γ ⊢ e₁ < e₂ and Γ ⊢ e₂ < e₃
    by IH: P⟦ γ(e₁) < γ(e₂) ⟧ and P⟦ γ(e₂) < γ(e₃) ⟧
    by definition of P: ∃v₁∃v₂. e₁ ~>* v₁, e₂ ~>* v₂ and v₁ << v₂
                        ∃v₂'∃v₃. e₂ ~>* v₂', e₃ ~>* v₃ and v₂' << v₃
    by determinism v₂ = v₂'
    by transitivity of << we get v₁ << v₃ and we're done.

  (V-AndIntro) φ = φ₁ ∧ φ₂, Γ ⊢ φ₁, Γ ⊢ φ₂
    by IH: Γ ⊨ φ₁ and Γ ⊨ φ₂, so also Γ ⊨ φ₁ ∧ φ₂

  (V-AndElim) φ = φᵢ, Γ ⊢ φ₁ ∧ φ₂
    by IH: Γ ⊨ φ₁ ∧ φ₂, so also Γ ⊨ φᵢ

  (V-ImplIntro) φ = φ₁ ⇒ φ₂, Γ, φ₁ ⊢ φ₂
    Assume P⟦γ(φ₁)⟧, to show: P⟦(φ₂)⟧
    by IH: ∀v ∈ V⟦_:nat→Pure nat (and (up φ₁) tot)⟧. P⟦γ(φ₂)⟧
    take v = λx:nat.x and show that v∈V⟦_:nat→Pure nat (and (up φ₁) tot)⟧
    ⇔ ∀n. n ∈ E⟦Pure nat (and (up φ₁) tot)⟧ (x ∉ fv(φ₁)
    ⇔ ∀n. ∀π∈W⟦Pure nat (and (up φ₁) tot), σ⟧. π v
    ⇔ ∀n. ∀π. P⟦γ(φ₁)⟧ ∧ (∀n'. π n') ⇒ π n, which holds

  (V-ImplElim) Γ ⊢ φ₁ ⇒ φ, Γ ⊢ φ₁
    by IH: P⟦γ(φ₁)⟧ and P⟦γ(φ₁)⟧ ⇒ P⟦γ(φ)⟧, so also P⟦γ(φ)⟧.

  (V-ForallIntro) Γ, x:t ⊢ φ
    by IH: ∀v ∈ V⟦t⟧. P⟦γ(φ)[v/x]⟧
    by definition of P: P⟦γ(∀x:t.φ)⟧.

  (V-ForallElim) Γ ⊢ ∀(x:t). φ    Γ ⊢ e : t
    by IH: ∀v ∈ V⟦t⟧. P⟦γ(φ)[v/x]⟧
    by IH (Γ ⊢ e : t part): e ∈ E⟦t⟧, so e~>*v and v∈V⟦t⟧, so P⟦γ(φ)[v/x]⟧
    by (P closed under expansion) lemma: P⟦γ(φ[e/x])⟧.

  (V-ExMiddle) Γ, φ₁ ⊢ φ,   Γ, ¬φ₁ ⊢ φ
    by IH: P⟦γ(φ₁)⟧ ⇒ P⟦γ(φ)⟧ (and fiddling with the encoding
                               of Γ, φ₁ as in V-ImplIntro)
    by IH: (P⟦γ(φ₁)⟧ ⇒ False) ⇒ P⟦γ(φ)⟧ (ditto)
```

```
   by excluded middle at the meta level: P⟦γ(φ)⟧

 (V-FalseElim) Γ ⊢ false
   by IH: P⟦false⟧ = False, contradiction.

 (V-ForallPIntro)
   let (γ,σ) ∈ G⟦Γ⟧, to show: P⟦∀(α:pred_t). φ, σ⟧ ⇔ ∀π. P⟦φ, σ[α↦π]⟧
   fix π; since (γ,σ[α↦π]) ∈ G⟦Γ⟧ by IH: P⟦φ, σ[α↦π]⟧ – done

 (V-ForallPElim)
   let (γ,σ) ∈ G⟦Γ⟧, to show: P⟦φ[(φ'[e/x])/(α(e))], σ⟧
   by IH: P⟦∀(α:pred_t). φ, σ⟧ ⇔ ∀π. P⟦ φ, σ[α↦π] ⟧
   we choose π = (fun v -> P⟦ φ'[v/x], σ ⟧)
   – note that this π does satisfy our invariance constraint
     by lemma (P closed under reduction and expansion)
   to obtain P⟦ φ, σ[α↦(fun v -> P⟦ φ'[v/x], σ ⟧)] ⟧
   by (Delayed substitutions) lemma we conclude: P⟦ φ[(φ'[e/x])/(α(e))], σ ⟧

 (S-Comp) Γ ⊢ t' <: t, Γ ⊢ ∀α:post_t'. w2f (Pure t' wp') (fun e -> α(e))
                                     ⇐ w2f (Pure t  wp ) (fun e -> α(e))
   by IH: Γ ⊨ t' <: t
   still to show: W⟦Pure γ(t') γ(wp'),σ⟧ ⊇ W⟦Pure γ(t) γ(wp),σ⟧
   by IH: P⟦∀α:post_t'. w2f (Pure γ(t') γ(wp')) (fun e -> α(e))
                      ⇐ w2f (Pure γ(t)  γ(wp)) (fun e -> α(e)), σ⟧
        ⇔ ∀π. P⟦w2f (Pure γ(t') γ(wp')) (fun e -> α(e)), σ[α↦π]⟧
            ⇐ P⟦w2f (Pure γ(t) γ(wp))   (fun e -> α(e)), σ[α↦π]⟧
       [by (Relating P+w2f and W; take 2) lemma]
        ⇔ ∀π. π ∈ W⟦Pure γ(t') γ(wp'),σ⟧ ⇐ π ∈ W⟦Pure γ(t) γ(wp),σ⟧
        ⇔ W⟦Pure γ(t') γ(wp'),σ⟧ ⊇ W⟦Pure γ(t) γ(wp),σ⟧, done

 (Sub-Nat) set inclusion trivially reflexive

 (Sub-Fun) Γ ⊢ t₂ <: t₁ and Γ, x:t₂ ⊢ c₁ <: c₂
   let (γ,σ) ∈ G⟦Γ⟧, to show: V⟦x:γ(t₁)→γ(c₁)),σ⟧ ⊆ V⟦x:γ(t₂)→γ(c₂)),σ⟧
   let v'∈V⟦x:γ(t₁)→γ(c₁)),σ⟧, to show: v'∈V⟦x:γ(t₂)→γ(c₂),σ⟧
   Subcase: v' = λx.e and ∀v∈V⟦γ(t₁),σ⟧. e[v/x] ∈ E⟦γ(c₁)[v/x], σ⟧ (H)
     Let v ∈ V⟦γ(t₂),σ⟧, to show: e[v/x] ∈ E⟦γ(c₂)[v/x]⟧
     by IH: V⟦γ(t₂),σ⟧ ⊆ V⟦γ(t₁),σ⟧, so v ∈ V⟦γ(t₁),σ⟧,
     so from H: e[v/x] ∈ E⟦γ(c₁)[v/x], σ⟧
     this means that: ∃v''. e[v/x] ~>* v''
        and v'' ∈ V⟦typ(γ(c₁)[v/x]),σ⟧ ∧ ∀π∈W⟦γ(c₁)[v/x], σ⟧. π v''
     by IH: V⟦typ(γ(c₁)[v/x]),σ⟧ ⊆ V⟦typ(γ(c₂)[v/x]),σ⟧
        and W⟦γ(c₁)[v/x],σ⟧ ⊇ W⟦γ(c₂)[v/x],σ⟧
     so v'' ∈ V⟦typ(γ(c₂)[v/x]),σ⟧
     let π ∈ W⟦γ(c₂)[v/x], σ⟧ ⊆ W⟦γ(c₁)[v/x],σ⟧
     by the above we know π v'', done
   Subcase: v' = let rec (f^δ:_) x = e
     and ∀v ∈ V⟦t⟧. e[v/x][(let rec (f^δ:_) x = e)/f] ∈ E⟦c[v/x]⟧ – similar

 (T-Var) e = x, t = Γ(x), to show γ(x) ∈ E⟦Pure γ(Γ(x)) tot, σ⟧
   choose v = γ(x) and get γ(x) ∈ V⟦γ(Γ(x)), σ⟧ directly from (γ,σ) ∈ G⟦Γ⟧
   Let π∈W⟦Pure γ(Γ(x)) tot, σ⟧; still to show: π γ(x)
   by def of W: ∀v∈V⟦γ(Γ(x)),σ⟧. π v, done

 (T-Abs) Γ, x:t ⊢ e₁ : c
   to show: λ(x:t).e₁ ∈ E⟦Pure (x:t→c) tot⟧
          ⇔ ∀π∈W⟦Pure (x:t→c) tot, σ⟧. λ(x:t).e₁ ∈ V⟦x:t→c,σ⟧  ∧  π (λ(x:t).e₁)
          ⇔ ∀π. (∀v∈V⟦x:t→c,σ⟧. π v) ⇒ λ(x:t).e₁ ∈ V⟦x:t→c,σ⟧  ∧  π (λ(x:t).e₁)
```

```
    suffices to show: λ(x:t).e₁ ∈ V⟦x:t→c,σ⟧
                    ⇔ ∀v∈V⟦t,σ⟧. e₁[v/x] ∈ E⟦c[v/x], σ⟧
    this follows immediately from the IH

  (T-App) Γ ⊢ e₁ : Pure (x:t₂→Pure t wp) wp₁, Γ ⊢ e₂ : Pure t₂ wp₂
    by IH: γ(e₁) ∈ E⟦Pure (x:γ(t₂)→Pure γ(t) γ(wp)) γ(wp₁)⟧
       and γ(e₂) ∈ E⟦Pure γ(t₂) γ(wp₂)⟧
    we put these together using (Dependent Application) lemma.

  (T-Zero) e = 0, t = nat, to show 0 ∈ E⟦Pure nat tot,ϙ⟧
    still to show: ∀π∈W⟦Pure nat tot,ϙ⟧. π 0
                 ⇔ ∀π. (∀v∈V⟦nat,σ⟧. π v). π 0 — trivial

  (T-Succ) Γ ⊢ e : Pure nat tot
    to show: S γ(e) ∈ E⟦Pure nat tot,ϙ⟧
    By induction hypothesis:
      γ(e) ∈ E⟦nat⟧, so ∃n. γ(e) ~>* n
    This can be used to produce a reduction:
      S γ(e) ~>* S n
    still to show: ∀π∈W⟦Pure nat tot,ϙ⟧. S n ∈ V⟦nat,ϙ⟧ ∧ π (S n)
                 ⇔ ∀π. (∀v∈V⟦nat,σ⟧. π v). π (S n) — trivial

  (T-Pred) Γ ⊢ e : Pure nat wp ∧ Γ ⊢ e0 : Pure t wp0
                               ∧ Γ ⊢ eS : Pure (x:nat → Pure t wpS) wp'
    by IH: γ(e) ∈ E⟦Pure nat wp⟧,
    so ∀π∈W⟦Pure nat wp, σ⟧. ∃n. γ(e') ~>* n ∧ π n (Hn)
    To show: γ(e) ∈ E⟦Pure t (bind wp >>= (y:nat).
              ite (y = 0) wp0 (bind wp' >> forall (x:nat). and (up (x < e)) wpS))⟧
    ⇔ ∀π∈W⟦Pure t (bind wp >>= (y:nat). ite (y = 0) wp0
              (bind wp' >> forall (x:nat). and (up (x < e)) wpS)), σ⟧.
         ∃v. γ(e) ~>* v ∧ v ∈ V⟦t⟧  ∧  π v
    ⇔ ∀π. W⟦Pure nat wp⟧ ∋ π'(π) ⇒ ∃v. γ(e) ~>* v ∧ v ∈ V⟦t⟧ ∧ π v
                where π'(π) = (fun y -> (P⟦y = 0⟧ ⇒ π ∈ W⟦Pure t wp0⟧) ∧
                                (¬P⟦y = 0⟧ ⇒ W⟦Pure _ wp'⟧ ∋ (fun _ ->
                                   ∀v∈V⟦nat,σ⟧. P⟦v < e⟧ ⇒ π ∈ W⟦Pure t wpS[v/x]⟧)))

    We prove this by case analysis on n:

    Case n = 0:
      Let π st. W⟦Pure nat wp⟧ ∋ π'(π)
      We instantiate (Hn) with π'(π) and get π'(π)(0) ⇔ π ∈ W⟦Pure t wp0⟧

      by IH: γ(e0) ∈ E⟦Pure t wp0⟧
      we instantiate this with π ∈ W⟦Pure t wp0⟧ to get:
        ∃v. γ(e0) ~>* v  ∧  v ∈ V⟦t⟧  ∧  π v
      We finally construct the following reduction:
        γ(e) ~> (pred 0 γ(e0) γ(eS)) ~> γ(e0) ~>* v   ∧   v ∈ V⟦t⟧

    Case n = S n':
      Let π st. W⟦Pure nat wp⟧ ∋ π'(π)
      We instantiate (Hn) with π'(π) and get: π'(π)(S n')
      ⇔ W⟦Pure _ wp'⟧ ∋ (fun _ -> ∀v∈V⟦nat,σ⟧. P⟦v < e⟧ ⇒ π ∈ W⟦Pure t wpS[v/x]⟧)
      by IH: γ(eS) ∈ E⟦Pure (x:nat → Pure t wpS) wp'⟧
      we also have that n' ∈ V⟦nat⟧, and also n' ∈ E⟦Pure nat (return n')⟧
      From these two by the (Dependent Application) lemma we get that
        γ(eS) n' ∈ E⟦Pure t[n'/x] (bind wp' >> bind (return n') >>= (x:nat) wpS)⟧
      So by the Expand lemma
        pred (S n') γ(e0) γ(eS)
```

```
        ∈ E⟦Pure t (bind wp' >> bind (return n') >>= (x:nat) wpS)⟧
    ⇔ ∀π. W⟦Pure _ wp'⟧ ∋ π''(π) ⇒
        ∃v. (pred n γ(e0) γ(eS)) ~>* v ∧ v ∈ V⟦t⟧ ∧ π v
        where π''(π) = (fun _ -> W⟦Pure nat (return n')⟧
                                  ∋ (fun x -> π ∈ W⟦Pure t wpS⟧))
    By (Monotonicity of W wrt π) lemma it suffices to show that:
      (∀v∈V⟦nat,σ⟧. P⟦v < e⟧ ⇒ π ∈ W⟦Pure t wpS[v/x]⟧) ⇒
       W⟦Pure nat (return n')⟧ ∋ (fun x -> π ∈ W⟦Pure t wpS⟧)
    Assume that: ∀v∈V⟦nat,σ⟧. P⟦v < e⟧ ⇒ π ∈ W⟦Pure t wpS[v/x]⟧
    By def W to show: π ∈ W⟦Pure t wpS[n'/x]⟧
    We instantiate assumption with v = n' and obtain what we want
    provided we can show P⟦n' < e⟧, which is immediate from
    (P closed under reduction) lemma and definition of P and <<

(T-Fix) e = let rec (f^δ:t) x = e'  ∧  t = x:t_x → Pure t' wp
  The (outer) induction hypotheses look as follows:
    γ(δ) ∈ E⟦Pure (x:γ(t_x) → Pure γ(t'') tot) tot,σ⟧
    ∀v^x ∈ V⟦t,σ⟧.
      ∀v^f ∈ V⟦y:γ(t_x) → Pure γ(t')[y/x] (and (up (δ y < δ v^x)) γ(wp)[y/x]),σ⟧.
        γ(e')[v^x/x][v^f/f] ∈ E⟦Pure γ(t')[v^x/x] γ(wp)[v^x/x],σ⟧
  To show: γ(e) ∈ V⟦Pure γ(t) tot,σ⟧

  We will show by a nested induction on n that
    ∀n. γ(e) ∈ V⟦x:γ(t_x) → Pure γ(t') (and (up (δ x < n)) γ(wp)),σ⟧  (G)

  Subcase n = 0:
    Let v ∈ V⟦γ(t_x),σ⟧
    To show: γ(e')[v/x][γ(e)/f] ∈ E⟦Pure γ(t')[v/x]
                                    (and (up (δ v < 0)) γ(wp)[v/x]), σ⟧
    Suffices to show:
      ∀π. π ∈ W⟦Pure γ(t')[v/x] (and (up (δ v < 0)) γ(wp)[v/x]), σ⟧ ⇒ ...
    ⇔ ∀π. (P⟦δ v < 0,σ⟧ ∧ π ∈ W⟦Pure γ(t')[v/x] γ(wp)[v/x], σ⟧) ⇒ ...
    ⇔ ∀π. (False ∧ π ∈ W⟦Pure γ(t')[v/x] γ(wp)[v/x], σ⟧) ⇒ ...
    – this holds vacuously

  Subcase n = S n'
    Let v ∈ V⟦γ(t_x),σ⟧
    To show: γ(e')[v/x][γ(e)/f] ∈ E⟦Pure γ(t')[v/x]
                                    (and (up (δ v < S n')) γ(wp)[v/x]), σ⟧
    ⇔ ∀π. π ∈ W⟦Pure γ(t')[v/x] (and (up (δ v < S n')) γ(wp)[v/x]), σ⟧ ⇒
        ∃v''. γ(e')[v/x][γ(e)/f] ~>* v'' ∧ v'' ∈ V⟦γ(t')[v/x], σ⟧ ∧ π v''
    Let π ∈ W⟦Pure γ(t')[v/x] (and (up (δ v < S n')) γ(wp)[v/x]), σ⟧
      ⇔ P⟦δ v < S n',σ⟧ ∧ π ∈ W⟦Pure γ(t')[v/x] γ(wp)[v/x], σ⟧
    We will apply the outer induction hypothesis for v and γ(e),
    but for that we first have to show that:
      γ(e) ∈ V⟦y:γ(t_x) → Pure γ(t')[y/x] (and (up (δ y < δ v)) γ(wp)[y/x]),σ⟧
    Let v' ∈ V⟦γ(t_x),σ⟧,
    to show: γ(e')[v'/x][γ(e)/f] ∈ E⟦Pure γ(t')[v'/x]
                                    (and (up (δ v' < δ v)) γ(wp)[v'/x]), σ⟧
    ⇔ ∀π'. π' ∈ W⟦Pure γ(t')[v'/x] (and (up (δ v' < δ v)) γ(wp)[v'/x]), σ⟧ ⇒
        ∃v''. γ(e')[v'/x][γ(e)/f] ~>* v'' ∧ v'' ∈ V⟦γ(t')[v'/x]), σ⟧ ∧ π' v''
    Let π' ∈ W⟦Pure γ(t')[v'/x] (and (up (δ v' < δ v)) γ(wp)[v'/x]), σ⟧
      ⇔ P⟦δ v' < δ v,σ⟧ ∧ π' ∈ W⟦Pure γ(t')[v'/x] γ(wp)[v'/x], σ⟧

    From P⟦δ v < S n',σ⟧ and P⟦δ v' < δ v,σ⟧ we obtain P⟦δ v' < n',σ⟧
    so also π' ∈ W⟦Pure γ(t')[v'/x] (and (up (δ v' < n')) γ(wp)[v'/x]), σ⟧
```

By the inner induction hypothesis we get:
γ(e) ∈ V⟦x:γ(tₓ) → Pure γ(t') (and (up (δ x < n')) γ(wp)),σ⟧ (G)
We instantiate this for v' ∈ V⟦γ(tₓ),σ⟧ and obtain:
γ(e')[v'/x][γ(e)/f] ∈ E⟦Pure γ(t')[v'/x]
(and (up (δ v' < n')) γ(wp)[v'/x]),σ⟧
We instantiate this for π' and obtain:
∃v''. γ(e')[v'/x][γ(e)/f] ~>* v'' ∧ v'' ∈ V⟦γ(t')[v'/x], σ⟧ ∧ π' v''
This allows us to finally apply the outer induction hypothesis,
from which we obtain:
γ(e')[v/x][γ(e)/f] ∈ E⟦Pure γ(t')[v/x] γ(wp)[v/x],σ⟧
We instantiate this for π ∈ W⟦Pure γ(t')[v/x] γ(wp)[v/x], σ⟧ to get:
∃v''. γ(e')[v/x][γ(e)/f] ~>* v'' ∧ v'' ∈ V⟦γ(t')[v/x], σ⟧ ∧ π v''
which is the final thing we had to show in this subcase

From (G) we can now show that γ(e) ∈ V⟦Pure γ(t) tot, σ⟧
Let v ∈ ⟦γ(tₓ),σ⟧,
to show: γ(e')[v/x][γ(e)/f] ∈ E⟦Pure γ(t')[v/x] γ(wp)[v/x], σ⟧
Let π ∈ W⟦Pure γ(t')[v/x] γ(wp)[v/x], σ⟧
still to show: ∃v. γ(e')[v/x][γ(e)/f] ~>* v ∧ v ∈ V⟦γ(t')[v/x],σ⟧ ∧ π v
We instantiate (G) with S (δ v) and obtain:
γ(e) ∈ V⟦x:γ(tₓ) → Pure γ(t') (and (up (δ x < S (δ v))) γ(wp)), σ⟧
We instantiate this with v ∈ ⟦γ(tₓ),σ⟧ to get
γ(e')[v/x][γ(e)/f] ∈ E⟦Pure γ(t')[v/x]
(and (up (δ v < S (δ v))) γ(wp)[v/x]), σ⟧ (H)
We want to instantiate (H) with π, so we first need to show that:
π ∈ W⟦Pure γ(t')[v/x] (and (up (δ v < S (δ v))) γ(wp)[v/x]), σ⟧
⇔ P⟦δ v < S (δ v), σ⟧ ∧ π ∈ W⟦Pure γ(t')[v/x] γ(wp)[v/x], σ⟧
the first conjunct is trivial, the second we have as assumption
So we can indeed instantiate (H) with π obtaining:
∃v. γ(e')[v/x][γ(e)/f] ~>* v ∧ v ∈ V⟦γ(t')[v/x],σ⟧ ∧ π v
which completes the proof of this case

(T-Ret) Γ ⊢ e : Pure t tot
By IH: γ(e) ∈ E⟦Pure γ(t) tot⟧,
so ∀π∈W⟦Pure γ(t) tot, σ⟧. ∃v. γ(e) ~>* v ∧ v ∈ V⟦γ(t),σ⟧ ∧ π v
⇔ ∀π. (∀v∈V⟦γ(t),σ⟧. π v) ⇒ ∃v. γ(e) ~>* v ∧ v ∈ V⟦γ(t),σ⟧ ∧ π v
We instantiate this with π = fun v -> True to get
∃v. γ(e) ~>* v ∧ v ∈ V⟦γ(t),σ⟧ (H)
To show:
γ(e) ∈ E⟦Pure γ(t) (return γ(e))⟧
⇔ ∀π∈W⟦Pure γ(t) (return γ(e)), σ⟧. ∃v. γ(e) ~>* v ∧ v ∈ V⟦γ(t),σ⟧ ∧ π v
⇔ ∀π. (∀v'. e ~>* v' ⇒ π v') ⇒ ∃v. γ(e) ~>* v ∧ v ∈ V⟦γ(t),σ⟧ ∧ π v
Let π so that ∀v'. γ(e) ~>* v' ⇒ π v'.
By H we can instantiate ∀v' with v to get π v,
and we conclude by putting this together with H.

(T-Sub) Γ ⊢ e : c₁, Γ ⊢ c₁ <: c₂
by IH: e ∈ E⟦γ(c₁), σ⟧
so ∀π∈W⟦γ(c₁), σ⟧. ∃v. e ~>* v ∧ v ∈ V⟦typ(γ(c₁)),σ⟧ ∧ π v
by IH: V⟦typ(γ(c₁)),σ⟧ ⊆ V⟦typ(γ(c₂)),σ⟧ ∧ W⟦γ(c₁),σ⟧ ⊇ W⟦γ(c₂),σ⟧
Let π∈W⟦γ(c₂), σ⟧, to show: ∃v. e ~>* v ∧ v ∈ V⟦typ(γ(c₂)),σ⟧ ∧ π v
since π∈W⟦γ(c₂), σ⟧ ⊆ W⟦γ(c₂), σ⟧ we obtain:
∃v. e ~>* v ∧ v ∈ V⟦typ(γ(c₁)),σ⟧ ∧ π v
by v ∈ V⟦typ(c₁),σ⟧ ⊆ V⟦typ(γ(c₂)),σ⟧, done □

Corollary (Consistency of validity): It is not the case that • ⊢ false

Proof:
  Assume by contradiction that • ⊢ false.
  By Soundness of Γ ⊢ φ we get • ⊨ false, so P⟦0 < 0⟧ holds,
  but P⟦0 < 0⟧ = 0 << 0 = False, contradiction □

Corollary (Weak normalization for closed expressions)
   • ⊢ e : Pure t wp ∧ ∃π ∈ W⟦Pure t wp, •⟧ ⇒ ∃v. e ~>* v
Proof: immediate from the (Soundness) theorem □

Corollary (Consistency of expression typing): ∄e. • ⊢ e : Pure nat (up True)
Proof: We assume by contradiction that ∃e. • ⊢ e : Pure nat (up True)
  Then by the soundness lemma, e ∈ E⟦Pure nat (up True)⟧
    ∀π∈W⟦Pure nat (up True), σ⟧. ∃v. e ~>* v ∧ v ∈ V⟦nat,σ⟧ ∧ π v
  ⇒ ∀π. P⟦True, σ⟧. ∃n. e ~>* n ∧ π n   [we instantiate π to False]
  ⇒ False – contradiction □

We previously used a more syntactic statement of weak normalization

Definition (Γ consistent)

  • ⊢ consistent

  ∃v. Γ ⊢ v : t   Γ ⊢ consistent
  ——————————————————————————
  Γ, x:t ⊢ consistent

  Γ ⊢ consistent
  ——————————————————————————
  Γ, α:pred$_t$ ⊢ consistent

Corollary (Soundness of Γ ⊢ consistent): Γ ⊢ consistent ⇒ ∃(γ,σ) ∈ G⟦Γ⟧

Corollary (More syntactic weak normalization)
  Γ ⊢ e : Pure t wp ∧ Γ consistent ∧ Γ ⊢ ∃α. w2f (Pure t wp) α ⇒
  ∃v γ. γ(e) ~>* v
Proof:
  by soundness on 1st premise: Γ ⊨ e : Pure t wp
  ⇒ ∀(γ,σ) ∈ G⟦Γ⟧. ∃π∈W⟦Pure γ(t) γ(wp), σ⟧ ⇒ ∃v. γ(e) ~>* v
  by soundness on 2nd premise: ∃(γ,σ) ∈ G⟦Γ⟧
  so also: ∃π∈W⟦Pure γ(t) γ(wp), σ⟧ ⇒ ∃v. γ(e) ~>* v
  by soundness on 3nd premise:
  ∃π. P⟦w2f (Pure γ(t) γ(wp)) α, σ[α↦π]⟧
  by (Relating P+w2f and W; take 2) lemma: π∈W⟦Pure γ(t) γ(wp), σ⟧
  so finally: ∃v γ. γ(e) ~>* v □

Corollary (More syntactic weak normalization for closed expressions)
   • ⊢ e : Pure t wp ∧ • ⊢ ∃α. w2f (Pure t wp) α ⇒ ∃v. e ~>* v
Proof: immediate from Weak normalization above taking Γ = • □

Corollary (Total correctness*):
  • ⊢ e : Pure t wp ⇒ ∀p. • ⊢ w2f (Pure t wp) p ⇒ ∃v. e ~>* v ∧ P⟦p v,∅⟧
Proof:
  by soundness: e ∈ E⟦Pure t wp,∅⟧
  so ∀π∈W⟦Pure t wp,∅⟧. ∃v. e ~>* v ∧ v ∈ V⟦typ(c),∅⟧ ∧ π v

Let p so that • ⊢ w2f (Pure t wp) p, by soundness: P⟦w2f (Pure t wp) p,∅⟧
by (Relating P+w2f and W; take 1) lemma: (fun v -> P⟦p v,∅⟧) ∈ W⟦Pure t wp,∅⟧
we instantiate π with (fun v -> P⟦p v,∅⟧) and obtain
∃v. e ~>* v ∧ v ∈ V⟦t,∅⟧ ∧ P⟦p v,∅⟧ □

* Note: the second condition in the conclusion is semantic as opposed
to the syntactic condition in the syntactic proof (• ⊢ p v). We
would need completeness of the validity judgment to obtain exactly
the same statement here as in the syntactic proof.
