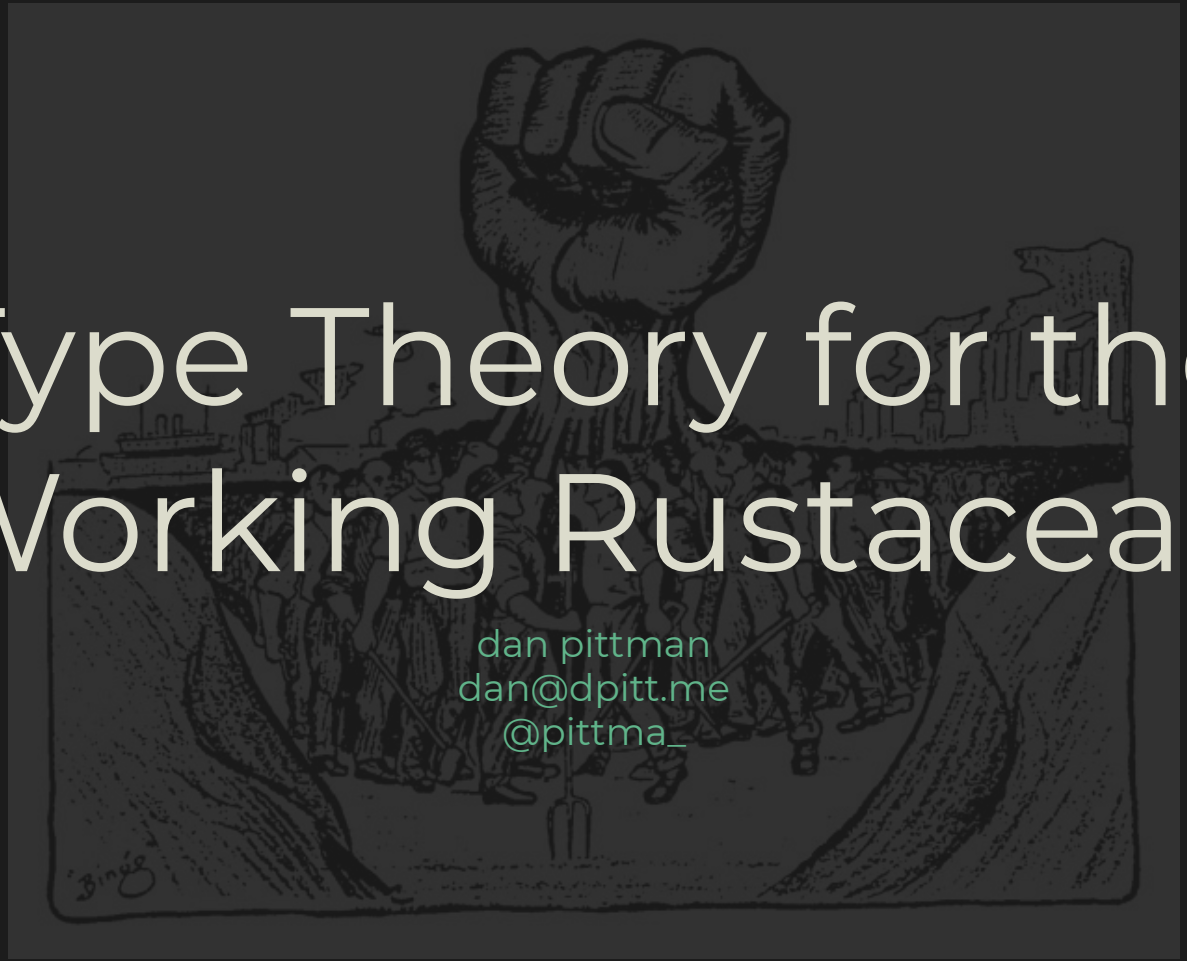
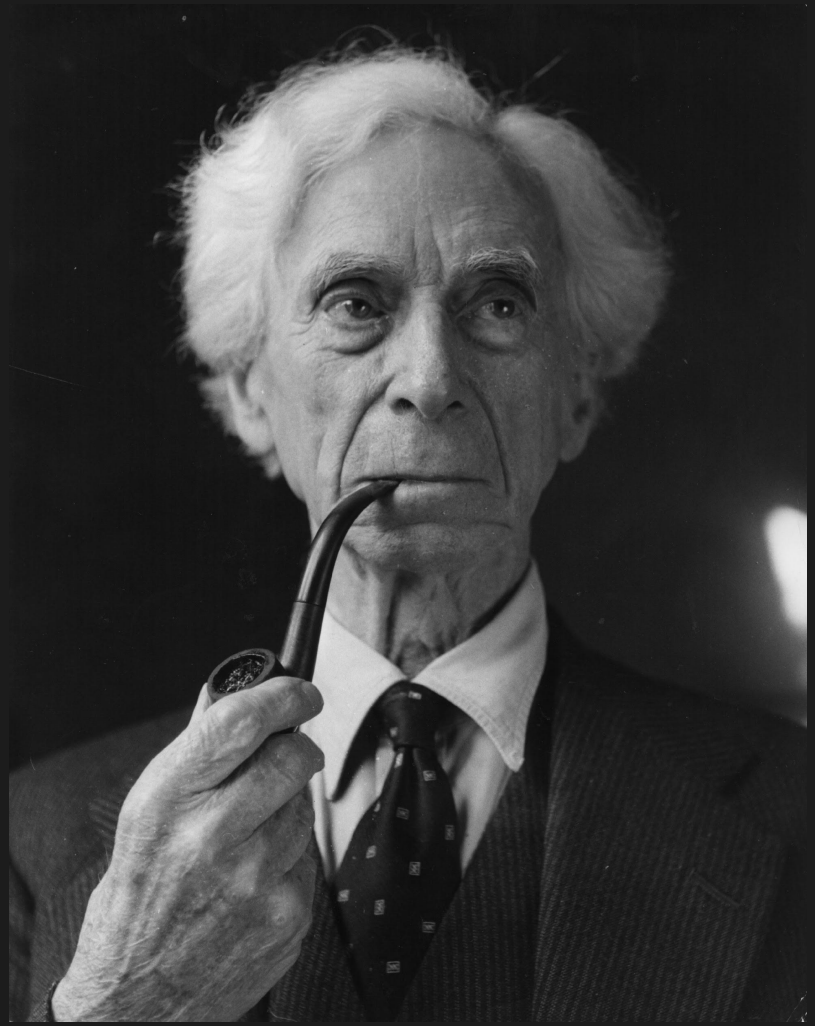


Type Theory for the Working Rustacean

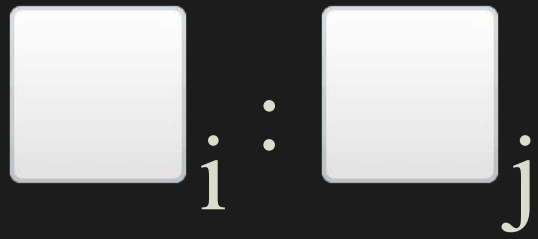
dan pittman
dan@dpitt.me
@pittma_

$$a : \mathrm{A}$$

```
x: u8
```



A : ★



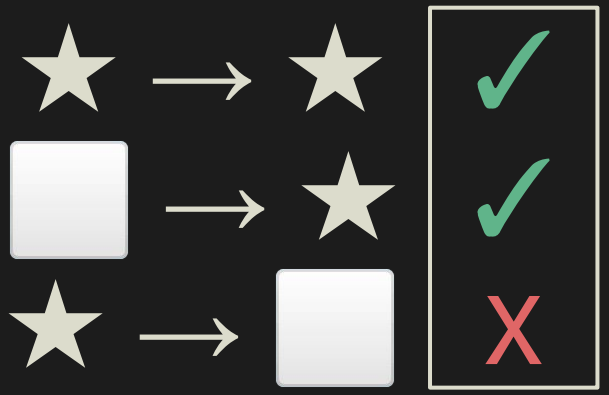

a : A : ★ : □i : □j : ...

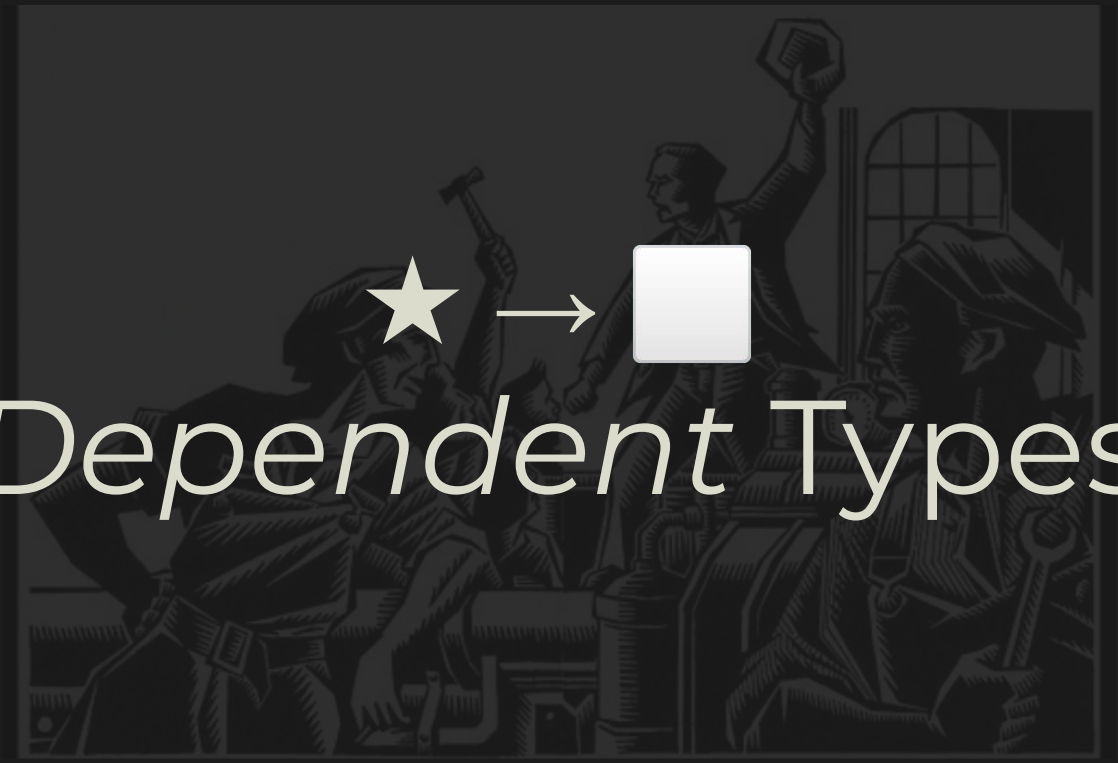
```
fn f(x: u8) -> String;
```

```
fn poly<T>(x: T) -> String;
```

```
struct Vec<T, n : u8>;

let v: Vec<&str, 1> = vec!["hi"];
```





★ → ⬜

(*Dependent* Types)

“Dependently typed programs are, by their nature, proof carrying code.”

- Altenkirch, McBride, & McKinna, *Why Dependent Types Matter*

```
data Nat : Type where
  zero : Nat
  suc : Nat → Nat

data Vec (X : Type) : (n : Nat) → Type where
  empty : Vec X zero
  cons : {n : Nat} → X → Vec X n → Vec X (suc n)
```

```
data Nat : Type where
  zero : Nat
  suc : Nat → Nat

data Vec (X : Type) : (n : Nat) → Type where
  empty : Vec X zero
  cons : {n : Nat} → X → Vec X n → Vec X (suc n)
```

```
data Nat : Type where
  zero : Nat
  suc : Nat → Nat

data Vec (X : Type) : (n : Nat) → Type where
  empty : Vec X zero
  cons : {n : Nat} → X → Vec X n → Vec X (suc n)
```

```
data Nat : Type where
  zero : Nat
  suc : Nat → Nat

data Vec (X : Type) : (n : Nat) → Type where
  empty : Vec X zero
  cons : {n : Nat} → X → Vec X n → Vec X (suc n)
```

```
data Nat : Type where
  zero : Nat
  suc : Nat → Nat

data Vec (X : Type) : (n : Nat) → Type where
  empty : Vec X zero
  cons : {n : Nat} → X → Vec X n → Vec X (suc n)
```

```
data Nat : Type where
  zero : Nat
  suc : Nat → Nat

data Vec (X : Type) : (n : Nat) → Type where
  empty : Vec X zero
  cons : {n : Nat} → X → Vec X n → Vec X (suc n)
```

```
data Nat : Type where
  zero : Nat
  suc : Nat → Nat

data Vec (X : Type) : (n : Nat) → Type where
  empty : Vec X zero
  cons : {n : Nat} → X → Vec X n → Vec X (suc n)
```

```
data Nat : Type where
  zero : Nat
  suc : Nat → Nat

data Vec (X : Type) : (n : Nat) → Type where
  empty : Vec X zero
  cons : {n : Nat} → X → Vec X n → Vec X (suc n)
```

```
data Nat : Type where
  zero : Nat
  suc : Nat → Nat

data Vec (X : Type) : (n : Nat) → Type where
  empty : Vec X zero
  cons : {n : Nat} → X → Vec X n → Vec X (suc n)
```

```
data Nat : Type where
  zero : Nat
  suc : Nat → Nat

data Vec (X : Type) : (n : Nat) → Type where
  empty : Vec X zero
  cons : {n : Nat} → X → Vec X n → Vec X (suc n)
```

```
data Nat : Type where
  zero : Nat
  suc : Nat → Nat

data Vec (X : Type) : (n : Nat) → Type where
  empty : Vec X zero
  cons : {n : Nat} → X → Vec X n → Vec X (suc n)
```

```
struct Vec<T, n : u8>;
```

```
struct Vec<T, N : Nat>;
```

```
trait Nat {}
struct Zero {}
impl Nat for Zero {}
struct Suc<N: Nat> {
    _pred: PhantomData<N>,
}

impl<N: Nat> Nat for Suc<N> {}
```

```
trait Nat {}
struct Zero {}
impl Nat for Zero {}
struct Suc<N: Nat> {
    _pred: PhantomData<N>,
}

impl<N: Nat> Nat for Suc<N> {}
```

```
struct SizeProofVec<T, N: Nat> {
    v: Vec<T>,
    _size: PhantomData<N>,
}

impl<T, N: Nat> SizeProofVec<T, N> {
    fn push(mut self, x: T) -> SizeProofVec<T, Suc<N>>
    {
        self.v.push(x);
        SizeProofVec {
            v: self.v,
            _size: PhantomData,
        }
    }
}
```

```
struct SizeProofVec<T, N: Nat> {
    v: Vec<T>,
    _size: PhantomData<N>,
}

impl<T, N: Nat> SizeProofVec<T, N> {
    fn push(mut self, x: T) -> SizeProofVec<T, Suc<N>>
    {
        self.v.push(x);
        SizeProofVec {
            v: self.v,
            _size: PhantomData,
        }
    }
}
```

```
struct SizeProofVec<T, N: Nat> {
    v: Vec<T>,
    _size: PhantomData<N>,
}

impl<T, N: Nat> SizeProofVec<T, N> {
    fn push(mut self, x: T) -> SizeProofVec<T, Suc<N>>
    {
        self.v.push(x);
        SizeProofVec {
            v: self.v,
            _size: PhantomData,
        }
    }
}
```

```
struct SizeProofVec<T, N: Nat> {
    v: Vec<T>,
    _size: PhantomData<N>,
}

impl<T, N: Nat> SizeProofVec<T, N> {
    fn push(mut self, x: T) -> SizeProofVec<T, Suc<N>>
    {
        self.v.push(x);
        SizeProofVec {
            v: self.v,
            _size: PhantomData,
        }
    }
}
```

```
struct SizeProofVec<T, N: Nat> {
    v: Vec<T>,
    _size: PhantomData<N>,
}

impl<T, N: Nat> SizeProofVec<T, N> {
    fn push(mut self, x: T) -> SizeProofVec<T, Suc<N>>
    {
        self.v.push(x);
        SizeProofVec {
            v: self.v,
            _size: PhantomData,
        }
    }
}
```

```
struct SizeProofVec<T, N: Nat> {
    v: Vec<T>,
    _size: PhantomData<N>,
}

impl<T, N: Nat> SizeProofVec<T, N> {
    fn push(mut self, x: T) -> SizeProofVec<T, Suc<N>>
    {
        self.v.push(x);
        SizeProofVec {
            v: self.v,
            _size: PhantomData,
        }
    }
}
```

```
struct SizeProofVec<T, N: Nat> {
    v: Vec<T>,
    _size: PhantomData<N>,
}

impl<T, N: Nat> SizeProofVec<T, N> {
    fn push(mut self, x: T) -> SizeProofVec<T, Suc<N>>
    {
        self.v.push(x);
        SizeProofVec {
            v: self.v,
            _size: PhantomData,
        }
    }
}
```

```rust
fn copy_from_slice(&mut self, src: &[T])
where
    T: Copy;
```

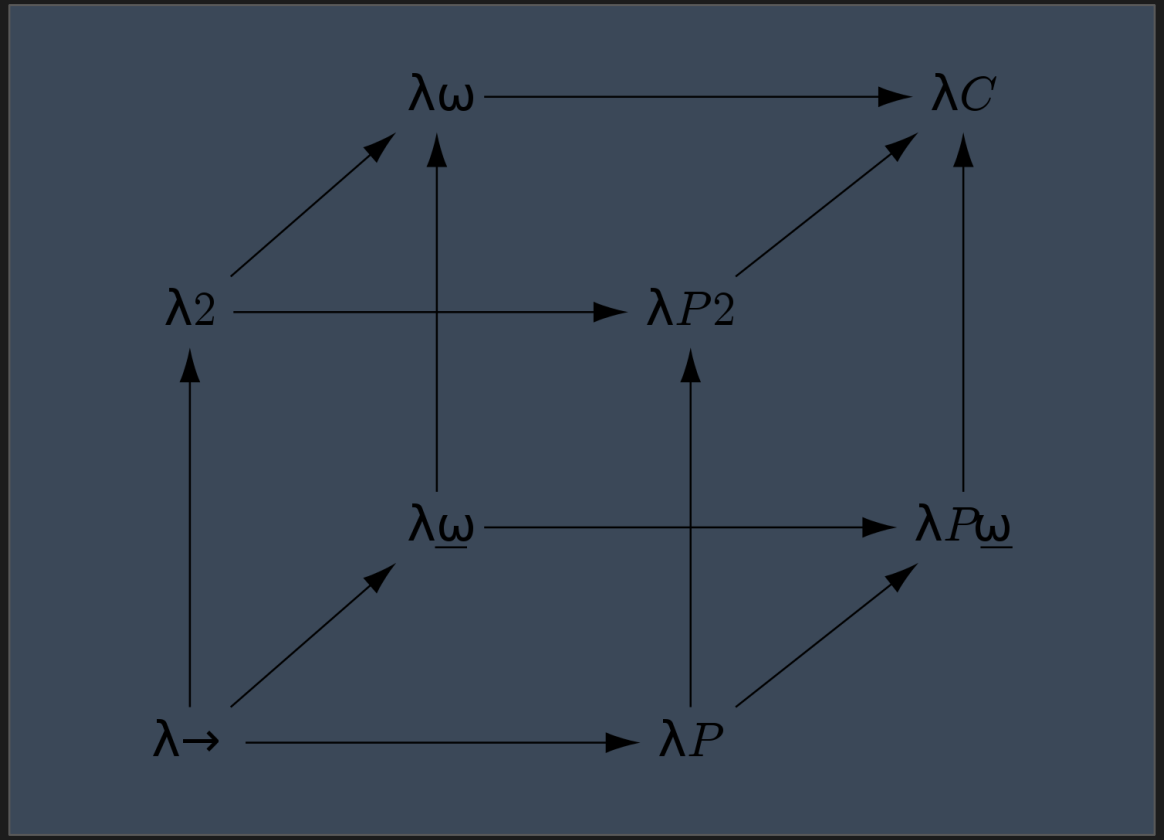
```
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `6`,
 right: `4`: destination and source slices have different
lengths'
```

```rust
fn copy_from<M: Nat>(&mut self, src: &SizeProofVec<T, M>)
where
    T: Copy,
    M: IsLessOrEqualTo<N, Result = True>,
{
    for (idx, item) in  src.vec.iter().enumerate() {
            self.vec[idx] = *item;
    }
}
```

```rust
fn copy_from<M: Nat>(&mut self, src: &SizeProofVec<T, M>)
where
    T: Copy,
    M: IsLessOrEqualTo<N, Result = True>,
{
    for (idx, item) in  src.vec.iter().enumerate() {
            self.vec[idx] = *item;
    }
}
```

```rust
fn copy_from<M: Nat>(&mut self, src: &SizeProofVec<T, M>)
where
    T: Copy,
    M: IsLessOrEqualTo<N, Result = True>,
{
    for (idx, item) in  src.vec.iter().enumerate() {
            self.vec[idx] = *item;
    }
}
```

```
85 |     b.copy_from(&a);
   |       ^^^^^^^^^ expected struct `False`, found struct `True`
   |
   = note: expected type `False`
              found type `True`
```


λω
λC
λ2
λP2
λω
λPω
λ→
λP