# Harlequin


## Language

`harlequin` starts with `fer-de-lance` and makes one major addition and a minor deletion

- add **static types**,
- replace unbounded tuples, with **pairs**.

That is, we now have a proper type system and the `Checker` is extended to **infer** types for all sub-expressions.

The code proceeds to `compile` (i.e. `Asm` generation) **only if it type checks**.

This lets us eliminate a whole bunch of the **dynamic tests**

- checking arithmetic arguments are actually numbers,
- checking branch conditions are actually booleans,

- checking tuple accesses are actually on tuples,
- checking that call-targets are actually functions,
- checking the arity of function calls,

etc. as code that typechecks is **guaranteed to pass the checks** at run time.

## Strategy

Lets start with an informal overview of our strategy for type inference; as usual we will then formalize and implement this strategy.

The core idea is this:

1. **Traverse** the `Expr` ...
2. **Generating** fresh variables for unknown types...
3. **Unifying** function input types with their arguments ...
4. **Substituting** solutions for variables to infer types.

Lets do the above procedure informally for a couple of examples!

### Example 1: Inputs and Outputs

```
              in
(defn (incr x) (+ x 1))     (→ (int) int)
                  out
(incr input)
```

### Example 2: Polymorphism

```
          in   out
(defn (id x) x)    (forall (a) (→ (a) a))

(let* ((a1 (id 7))        (→ (int) int)
       (a2 (id true)))    (→ (bool) bool)
  true)
```

### Example 3: Higher-Order Functions

(forall (a)

(→ ((→(a) int) a ) int ))

```
(defn (f it x)
  (+ (it x) 1))
```

(→ (int) int)

```
(defn (incr z)
  (+ z 1))
```

'a=int'

```
(f incr 10)
```

(→ (. . . .) _)

## Example 4: Lists

```
;; --- an API for lists -------------------------------------
(defn (nil) (as (forall (a) (-> () (list a))))
  false)
```

(forall (a b) (→ (a (list b)) (list b)))

```
(defn (cons h t) (as (forall (a) (-> (a (list a)) (list a))))
  (vec h t))

(defn (head l) (as (forall (a) (-> ((list a)) a)))
  (vec-get l 0))

(defn (tail l) (as (forall (a) (-> ((list a)) (list a))))
  (vec-get l 1))

(defn (isnil l) (as (forall (a) (-> ((list a)) bool)))
  (= l false))

;;--- computing with lists ----------------------------------

(defn (length xs)
  (if (isnil xs)
    0
    (+ 1 (length (tail xs)))))
```

(forall (a) (→ ((list a)) int))

```
(defn (sum xs)
  (if (isnil xs)
    0
    (+ (head xs) (sum (tail xs)))))
```

(→ ((list int)) int )

true

```
(let (xs  (cons 10 (cons 20 (cons 30 (nil)))))
  (vec (length xs) (sum xs)))
```

Int (list int)

(cons true (cons 20 (cons 30 (nil))))

list a

## Strategy Recap

cons : bool (list int)

1. **Traverse** the `Expr` ...
2. **Fresh** variables for unknown types...

$e : \boxed{t}$

$\rightarrow (a)\ b \equiv \rightarrow (c)\ int$

$a \equiv c$
$b \equiv int$

3. **Unifying** function input types with their arguments ...
4. **Substituting** solutions for variables to infer types ...
5. **Generalizing** types into polymorphic functions ...
6. **Instantiating** polymorphic type variables at each use-site.

# Plan

1. **Types**
2. Expressions
3. Variables & Substitution
4. Unification
5. Generalize & Instantiate
6. Inferring Types
7. Extensions

# Syntax

First, lets see how the syntax of our `garter` changes to enable static types.

## Syntax of Types

A `Type` is one of:

```
pub enum Ty {
  Int,
  Bool,
  Fun(Vec<Ty>, Box<Ty>),
  Var(TyVar),
  Vec(Box<Ty>, Box<Ty>),
  Ctor(TyCtor, Vec<Ty>),
}
```

int
bool
$t_1 \ldots t_n$   $t$   $(\rightarrow (t_1 \ldots t_n)\ t)$
a
$(vec\ t_1\ t_2)$
"Map" [int, bool]
"list" [int]

here `TyCtor` and `TyVar` are just string names:

```
pub struct TyVar(String);  // e.g. "a", "b", "c"

pub struct TyCtor(String); // e.g. "List", "Tree"
```

Finally, a **polymorphic type** is represented as:

```
pub struct Poly {
  pub vars: Vec<TyVar>,
  pub ty: Ty,
}
```

forall $(a_1, a_2 \ldots a_n)$
ty

## Example: Monomorphic Types

A function that

- takes two input `Int`
- returns an output `Int`

Has the *monomorphic* type `(-> (Int Int) Int)`

Which we would represent as a `Poly` value:

$\phi$

```
forall(vec![], fun(vec![Ty::Int, Ty::Int], Ty::Int))
```

**Note:** If a function is **monomorphic** (i.e. *not polymorphic*), we can just use the empty vec of `TyVar`.

## Example: Polymorphic Types

Similarly, a function that

- takes a value of **any** type and
- returns a value of **the same** type

Has the *polymorphic* type `(forall (a) (-> (a) a))`

Which we would represent as a `Poly` value:

```
forall(
    vec![tv("a")],
    fun(vec![Ty::Var(tv("a"))], Box::new(Ty::Var(tv("a")))),
)
```

Similarly, a function that takes two values and returns the first, can be given a `Poly` type `(forall (a b) (-> (a b) a))` which is represented as:

arg1 arg2 arg1

```
forall(
    vec![tv("a"), tv("b")],
    fun(vec![Ty::Var(tv("a")), Ty::Var(tv("b"))], Ty::Var(tv("a"))))
)
```

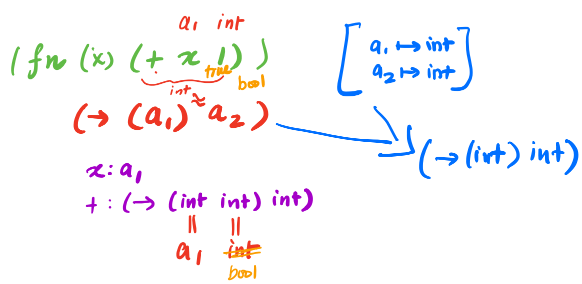
## Syntax of Expressions

To enable inference `harlequin` simplifies the language a little bit.

- *Dynamic tests* `isNum` and `isBool` are removed,
- *Tuples* always have exactly *two* elements; you can represent `(vec e1 e2 e3)` as `(vec e1 (vec e2 e3))`.
- *Tuple* access is limited to the fields `Zero` and `One` (instead of arbitrary expressions).

```
pub enum ExprKind<Ann>{
  ...
  Vek(Box<Expr<Ann>>, Box<Expr<Ann>>),  // Tuples have 2 elems
  Get(Box<Expr<Ann>>, Index),           // Get 0-th or 1-st elem
  Fun(Defn<Ann>),                       // Named functions
}

pub enum Index {
    Zero,                               // 0-th field
    One,                                // 1-st field
}
```




## Plan

1. Types
2. Expressions
3. **Variables & Substitution**
4. Unification
5. Generalize & Instantiate
6. Inferring Types
7. Extensions

# Substitutions

Our informal algorithm proceeds by

$a_1, a_2$

- Generating **fresh type** variables for unknown types,
- Traversing the `Expr` to **unify** the types of sub-expressions,
- By **substituting** a type *variable* with a whole *type*.

Lets formalize *substitutions*, and use it to define unification.

## Representing Substitutions

We represent substitutions as a record with two fields:

```
struct Subst {
    /// hashmap from type-var |-> type
    map: HashMap<TyVar, Ty>,
    /// counter used to generate fresh type variables
    idx: usize,
}
```

- `map` is a **map** from type variables to types,
- `idx` is a **counter** used to generate **new** type variables.

For example, `ex_subst()` is a substitution that maps `a`, `b` and `c` to `int`, `bool` and `(-> (int int) int)` respectively.

```
let ex_subst = Subst::new(&[
  (tv("a"), Ty::Int),
  (tv("b"), Ty::Bool),
  (tv("c"), fun(vec![Ty::Int, Ty::Int], Ty::Int)),
]);
```

$$\left[\begin{array}{l} a \mapsto int \\ b \mapsto bool \\ c \mapsto (\to (int\ int)\ int) \end{array}\right]$$

## Applying Substitutions

The main *use* of a substition is to **apply** it to a type, which has the effect of *replacing* each occurrence of a type variable with its substituted value (or leaving it untouched if it is not mentioned in the substitution.)

We define an interface for "things that can be substituted" as:

```
trait Subable {
  fn apply(&self, subst: &Subst) -> Self;
  fn free_vars(&self) -> HashSet<TyVar>;
}
```

and then we define how to `apply` substitutions to `Type`, `Poly`, and lists and maps of `Type` and `Poly`.

(→ (int z) bool)

For example,

(→ (a z) b)

```
let ty = fun(vec![tyv("a"), tyv("z")], tyv("b"));
ty.apply(&ex_subst)
```

[a ↦ int
b ↦ bool
c ↦ (→ (int int) int)]

returns a type like

```
fun(vec![Ty::Int, tyv("z")], Ty::Bool)
```

by replacing `"a"` and `"b"` with `Ty::Int` and `Ty::Bool` and leaving "z" alone.

## QUIZ / Handout

Recall that `let ex_subst = ["a" |-> Ty::Int, "b" |-> Ty::Bool]` ...

What should be the result of

```
let ty = forall(vec!["a"], fun(vec![tyv("a")], tyv("a")));
ty.apply(ex_subst)
```

1. `forall(vec!["a"], fun(vec![Ty::Int ], Ty::Bool)`
2. `forall(vec!["a"], fun(vec![tyv("a")], tyv("a"))`
3. `forall(vec!["a"], fun(vec![tyv("a")], Ty::Bool)`
4. `forall(vec!["a"], fun(vec![Ty::Int ], tyv("a"))`
5. `forall(vec![   ], fun(vec![Ty::Int ], Ty::Bool)`

## Bound vs. Free Type Variables

Indeed, the type

```
(forall (a) (-> (a) a))
```

[a ↦ int]

is identical to

```
(forall (z) (-> (z) z))
```

- A **bound** type variable is one that appears under a `forall`.
- A **free** type variable is one that is **not** bound.

We should only substitute **free type variables**.

## Applying Substitutions

Thus, keeping the above in mind, we can define `apply` as a recursive traversal:

```
fn apply(&self, subst: &Subst) -> Self {
  let mut subst = subst.clone();
  subst.remove(&self.vars);
  forall(self.vars.clone(), self.ty.apply(&subst))
}

fn apply(ty: &Ty, subst: &Subst) -> Self {
  match ty {
    Ty::Int => Ty::Int,
    Ty::Bool => Ty::Bool,
    Ty::Var(a) => subst.lookup(a).unwrap_or(Ty::Var(a.clone())),
    Ty::Fun(in_tys, out_ty) => {
      let in_tys = in_tys.iter().map(|ty| ty.apply(subst)).collect();
      let out_ty = out_ty.apply(subst);
      fun(in_tys, out_ty)
    }
    Ty::Vec(ty0, ty1) => {
      let ty0 = ty0.apply(subst);
      let ty1 = ty1.apply(subst);
      Ty::Vec(Box::new(ty0), Box::new(ty1))
    }
    Ty::Ctor(c, tys) => {
      let tys = tys.iter().map(|ty| ty.apply(subst)).collect();
      Ty::Ctor(c.clone(), tys)
    }
  }
}
```

where `subst.remove(vs)` **removes** the mappings for `vs` from `subst`

## Creating Substitutions

We can start with an **empty substitution** that maps no variables

subst $[a \mapsto \cancel{b}\ int]$

new $b \mapsto int$

new subst $\begin{bmatrix} a \mapsto int \\ b \mapsto int \end{bmatrix}$

```
fn new() -> Subst {
  Subst { map: Hashmap::new(), idx: 0 }
}
```

## Extending Substitutions

we can **extend** the substitution by assigning a variable `a` to type `t`

a ty

$a \mapsto ty$

```
fn extend(&mut self, tv: &TyVar, ty: &Ty) {
  // create a new substitution tv |-> ty
  let subst_tv_ty = Self::new(&[(tv.clone(), ty.clone())]);
  // apply tv |-> ty to all existing mappings
  let mut map = hashmap! {};
  for (k, t) in self.map.iter() {
      map.insert(k.clone(), t.apply(&subst_tv_ty));
  }
  // add new mapping
  map.insert(tv.clone(), ty.clone());
  self.map = map
}
```

### Telescoping

Note that when we extend `[b |-> a]` by assigning `a` to `Int` we must take care to also update `b` to now map to `Int`. That is why we:

0. Create a new substitution `[a |-> Int]`
1. Apply it to each binding in `self.map` to get `[b |-> Int]`
2. Insert it to get the extended substitution `[b |-> Int, a |-> Int]`

## Plan

1. Types
2. Expressions
3. Variables & Substitution ✓
4. **Unification**
5. Generalize & Instantiate
6. Inferring Types
7. Extensions

unify $(t_1, t_2) \rightarrow$ UNIFIER + error

# Unification

Next, lets use `Subst` to implement a procedure to `unify` two types, i.e. to determine the conditions under which the two types are *the same*.

| T1 | T2 | Unified | Substitution |
|---|---|---|---|
| `Int` | `Int` | `Int` | `empSubst` |
| `a` | `Int` | `Int` | `a` $a \mapsto int$ |
| `a` | `b` | `b` | `a` $a \mapsto b$ |
| `a -> b` | `a -> d` | `a->d` | `b` $b \mapsto d$ |
| `a -> Int` | `Bool -> b` | `Bool->Int` | `a` $a \mapsto bool, b \mapsto int$ |
| `Int` | `Bool` | *Error* | *Error* |
| `Int` | `a -> b` | "occurs check" *Error* | *Error* |
| `a` (a) | $\boxed{\texttt{a -> Int}}$ ($b \rightarrow int$) | *Error* | *Error* $a \mapsto (b \rightarrow int)$ |

- The first few cases: unification is possible,
- The last few cases: unification fails, i.e. type error in source!

### Occurs Check

- The very last failure: `a` in the first type **occurs inside** free inside the second type!
- If we try substituting `a` with `a -> Int` we will just keep spinning forever! Hence, this also throws a unification failure.

### Exercise

Can you think of a program that would trigger the *occurs check* failure?

## Implementing Unification

We implement unification as a function:

```
fn unify<A: Span>(ann: &A, subst: &mut Subst, t1: &Ty, t2: &Ty) -> Result<(),
Error>
```

such that

```
unify(ann, subst, t1, t2)
```

- either **extends** `subst` with assignments needed to make `t1` the same as `t2`,
- or returns an **error** if the types cannot be unified.

The code is pretty much the table above:

```
fn unify<A: Span>(ann: &A, subst: &mut Subst, t1: &Ty, t2: &Ty) -> Result<(),
Error> {
  match (t1, t2) {
    (Ty::Int, Ty::Int) | (Ty::Bool, Ty::Bool) => Ok(()),
    (Ty::Fun(ins1, out1), Ty::Fun(ins2, out2)) => {
      unifys(ann, subst, ins1, ins2)?;
      let out1 = out1.apply(subst);
      let out2 = out2.apply(subst);
      unify(ann, subst, &out1, &out2)
    }
    (Ty::Ctor(c1, t1s), Ty::Ctor(c2, t2s)) if *c1 == *c2 => unifys(ann, subst,
t1s, t2s),
    (Ty::Vec(s1, s2), Ty::Vec(t1, t2)) => {
      unify(ann, subst, s1, t1)?;
      let s2 = s2.apply(subst);
      let t2 = t2.apply(subst);
      unify(ann, subst, &s2, &t2)
    }
    (Ty::Var(a), t) | (t, Ty::Var(a)) => var_assign(ann, subst, a, t),
    (_, _) =>
    {
        Err(Error::new(
            ann.span(),
            format! {"Type Error: cannot unify {t1} and {t2}"},
        ))
    }
  }
}
```

The helpers

- `unifys` recursively calls `unify` on *sequences* of types:
- `var_assign` extends `su` with `[a |-> t]` if **required** and **possible**!

```rust
fn var_assign<A: Span>(ann: &A, subst: &mut Subst, a: &TyVar, t: &Ty) ->
Result<(), Error> {
  if *t == Ty::Var(a.clone()) {
    Ok(())
  } else if t.free_vars().contains(a) {
    Err(Error::new(ann.span(), "occurs check error".to_string()))
  } else {
    subst.extend(a, t);
    Ok(())
  }
}
```

We can test out the above table:

```rust
#[test]
fn unify0() {
  let mut subst = Subst::new(&[]);
  let _ = unify(&(0, 0), &mut subst, &Ty::Int, &Ty::Int);
  assert!(format!("{:?}", subst) == "Subst { map: {}, idx: 0 }")
}

#[test]
fn unify1() {
  let mut subst = Subst::new(&[]);
  let t1 = fun(vec![tyv("a")], Ty::Int);
  let t2 = fun(vec![Ty::Bool], tyv("b"));
  let _ = unify(&(0, 0), &mut subst, &t1, &t2);
  assert!(subst.map == hashmap! {tv("a") => Ty::Bool, tv("b") => Ty::Int})
}

#[test]
fn unify2() {
  let mut subst = Subst::new(&[]);
  let t1 = tyv("a");
  let t2 = fun(vec![tyv("a")], Ty::Int);
  let res = unify(&(0, 0), &mut subst, &t1, &t2).err().unwrap();
  assert!(format!("{res}") == "occurs check error: a occurs in (-> (a) int)")
}

#[test]
fn unify3() {
  let mut subst = Subst::new(&[]);
  let res = unify(&(0, 0), &mut subst, &Ty::Int, &Ty::Bool)
        .err()
        .unwrap();
  assert!(format!("{res}") == "Type Error: cannot unify int and bool")
}
```

let (id (fn (x) $\underbrace{x}_{a_1}$))

(→ ($a_1$) $a_2$) [$a_2 \mapsto a_1$]

$x : a_1$

(forall (a) (→ (a) a))

## Plan



# Generalize and Instantiate

Recall the example:

id: (forall (a) (→ (a) a))

```
(defn (id x) x)
```

(→ ($a_1$) $a_1$)

```
(let* ((a1 (id 7))
       (a2 (id true)))
  true)
```

(→ ($a_2$) $a_2$)

For the expression `(defn (id x) x)` we inferred the type `(-> (a0) a0)`

We needed to **generalize** the above:

- to assign `id` the `Poly`-type: `(forall (a0) (-> (a0) a0))`

We needed to **instantiate** the above `Poly`-type at each *use*

- at `(id 7)` the function `id` has type `-> (int) int`
- at `(id true)` the function `id` has type `-> (bool) bool`

Lets see how to implement those two steps.

## Type Environments

To `generalize` a type, we

1. Compute its (free) type variables,
2. Remove the ones that may still be constrained by *other* in-scope program variables.

We represent the types of **in scope** program variables as **type environment**

```
struct TypeEnv(HashMap<String, Poly>);
```

i.e. a `Map` from program variables `Id` to their (inferred) `Poly` type.

## Generalize

We can now implement `generalize` as:

```
fn generalize(env: &TypeEnv, ty: Ty) -> Poly {
    // 1. compute ty_vars of `ty`
    let ty_vars = ty.free_vars();
    // 2. compute ty_vars of `env`
    let env_vars = env.free_vars();
    // 3. compute unconstrained vars: (1) minus (2)
    let tvs = ty_vars.difference(env_vars).into_iter().collect();
    // 4. slap a `forall` on the unconstrained `tvs`
    forall(tvs, ty)
}
```

The helper `freeTvars` computes the set of variables that appear inside a `Type`, `Poly` and `TypeEnv`:

```
// Free Variables of a Type
fn free_vars(ty:&Ty) -> HashSet<TyVar> {
  match ty{
    Ty::Int | Ty::Bool => hashset! {},
    Ty::Var(a) => hashset! {a.clone()},
    Ty::Fun(in_tys, out_ty) =>
free_vars_many(in_tys).union(out_ty.free_vars()),
    Ty::Vec(t0, t1) => t0.free_vars().union(t1.free_vars()),
    Ty::Ctor(_, tys) => free_vars_many(tys),
  }
}

// Free Variables of a Poly
fn free_vars(poly: &Poly) -> HashSet<TyVar> {
  let bound_vars = poly.vars.clone().into();
  poly.ty.free_vars().difference(bound_vars)
}

// Free Variables of a TypeEnv
fn free_vars(env: &TypeEnv) -> HashSet<TyVar> {
  let mut res = HashSet::new();
  for poly in self.0.values() {
      res = res.union(poly.free_vars());
  }
  res
}
```

## Instantiate

Next, to **instantiate** a `Poly` of the form

```
forall(vec![a1,...,an], ty)
```

we:

1. Generate **fresh** type variables, `b1,...,bn` for each "parameter" `a1...an`
2. Substitute `[a1 |-> b1,...,an |-> bn]` in the "body" `ty` .

For example, to instantiate

```
forall(vec![tv("a")], fun(vec![tyv("a")], tyv("a")))
```

we

1. Generate a fresh variable e.g. `"a66"` ,
2. Substitute `["a" |-> "a66"]` in the body `["a"] |->> "a"`

to get

```
fun(vec![tyv("a66")], tyv("a66"))
```

## Implementing Instantiate

We implement the above as:

```
fn instantiate(&mut self, poly: &Poly) -> Ty {
  let mut tv_tys = vec![];
  // 1. Generate fresh type variables [b1...bn] for each `a1...an` of poly
  for tv in &poly.vars {
      tv_tys.push((tv.clone(), self.fresh()));
  }
  // 2. Substitute [a1 |-> b1, ... an |-> bn] in the body `ty`
  let su_inst = Subst::new(&tv_tys);
  poly.ty.apply(&su_inst)
}
```

**Question** Why does `instantiate` **update** a `Subst` ?

Lets run it and see what happens!

```
let t_id = forall(vec![tv("a")], fun(vec![tyv("a")], tyv("a")));

let mut subst = Subst::new(&[]);

let ty0 = subst.instantiate(&t_id);
let ty1 = subst.instantiate(&t_id);
let ty2 = subst.instantiate(&t_id);

assert!(ty0 == fun(vec![tyv("a0")], tyv("a0")));
assert!(ty1 == fun(vec![tyv("a1")], tyv("a1")));
assert!(ty2 == fun(vec![tyv("a2")], tyv("a2")));
```

(-> (a) b) ~ (-> (bool) int)
a ~ bool
b ~ int

- The `fresh` calls **bump up** the counter (so we *actually* get fresh variables)

## Plan

1. Types
2. Expressions
3. Variables & Substitution
4. Unification

int
bool } base

(-> (int int) int)

(forall (a) (-> (a) a))

x: vec a b

(fun (x) (vec (vec-get x 1)$^b$
(vec-set x 0)$^a$))

(vec ba)



(vec 2 true)

(vec int bool)

(∀(a b) (→ ((vec a b)) (vec b a)))

# Inference

[a ↦ int , b ↦ bool]

The top-level *type-checker* looks like this:

Finally, we have all the pieces to implement the actual **type inference** procedure `infer`

```
fn infer<A: Span>(env: &TypeEnv, subst: &mut Subst, e: &Expr<A>) -> Result<Ty,
Error>
```

which takes as *input*:

1. A `TypeEnv` ( `env` ) mapping in-scope variables to their previously inferred ( `Poly` )-types,
2. A `Subst` ( `subst` ) containing the *current* substitution/fresh-variable-counter,
3. An `Expr` ( `e` ) whose type we want to infer,

and

- *returns* as output the **inferred type** for `e` (or an `Error` if no such type exists), and
- *updates* `subst` by
  - generating **fresh type** variables and
  - doing the **unifications** needed to check the `Expr` .

Lets look at how `infer` is implemented for the different cases of expressions.

## Inference: Literals

For numbers and booleans, we just return the respective type and the input `Subst` without any modifications.

```
Num(_) | Input => Ty::Int,
True | False => Ty::Bool,
```

2

## Inference: Variables

id : (∀(a) (→ (a) a))

x : int

(id x)

For identifiers, we

1. **lookup** their type in the `env` and
2. **instantiate** type-variables to get *different types at different uses.*

(let (add (fn (x1 x2) (+ x1 x2)) (add 10 20))

e1 e2

```
Var(x) => subst.instantiate(env.lookup(x)?),
```

(→ (a) b)

Why do we *instantiate*? Recall [the `id` example!]

ENV

let x = e1 in ENV + x ↦ s1 e2 s1

(λx. e2) e1

## Inference: Let-bindings

let id = (fn (x) x)

in ENV + id ↦ (∀(a0)(→ (a0) a0))

(vec (id 10) (id true))

Next, lets look at let-bindings:

(fn (id) (vec (id 10) (id true)))

(....)

```
Let(x, e1, e2) => {
  let t1   = infer(env, subst, e1)?;           // (1)
  let env1 = env.apply(subst);                 // (2)
  let s1   = generalize(&env1, t1);            // (3)
  let env2 = env1.extend(&[(x.clone(), s1)]);  // (4)
  infer(&env2, subst, e2)?                     // (5)
}
```

In essence,

1. **Infer** the type `t1` for `e1`,
2. **Apply** the substitutions from (1) to the `env`,
3. **Generalize** `t1` to make it a `Poly` type `s1`,
   - why? [recall the id example]
4. **Extend** the env to map `x` to `s1` and,
5. **Infer** the type of `e2` in the extended environment.

3 env

## Inference: Function Definitions

Next, lets see how to infer the type of a function i.e. `Lam`

→ a

(let (id (fn (x) x))

...)

① (→ (a0) a0) // "template"

② using env + x1 ↦ a1 ... xn ↦ an

INFER(body) ~ out

(→ (a1...an) out)

```
fn infer_defn<A: Span>(env: &TypeEnv, subst: &mut Subst, defn: &Defn<A>) ->
Result<Ty, Error> {
  // 1. Generate a fresh template for the function
  let (in_tys, out_ty) = fresh_fun(defn, subst);

  // 2. Add the types of the params to the environment
  let mut binds = vec![];
  for (x, ty) in defn.params.iter().zip(&in_tys) {
    binds.push((x.clone(), mono(ty.clone())))
  }
  let env = env.extend(&binds);

  // 3. infer the type of the body
  let body_ty = infer(&env, subst, &defn.body)?;

  // 4. Unify the body type with the output type
  unify(&defn.body.ann, subst, &body_ty, &out_ty.apply(subst))?;

  // 5. Return the function's template after applying subst
  Ok(fun(in_tys.clone(), out_ty.clone()).apply(subst))
}
```

Inference works as follows:

1. Generate a *function type* with fresh variables for the unknown inputs ( `in_tys` ) and output ( `out_ty` ),
2. Extend the `env` so the parameters `xs` have types `in_tys` ,
3. Infer the type of `body` under the extended `env` as `body_ty` ,
4. Unify the *expected* output `out_ty` with the *actual* `body_ty`
5. Apply the substitutions to infer the function's type `(-> (in_tys) out_ty)`

1

## Inference: Function Calls

Finally, lets see how to infer the types of a call to a function whose ( `Poly` )-type is `poly` with arguments `in_args`

Ex:

id : (forall (a) (-> (a) a))

b

id = ? (-> (b) b)

(id $\boxed{10}$)

int

(-> (int) out)

out = int

Ex: (if $e_1$ $e_2$ $e_3$)

plus (-> (int int) int)

(-> ($s_1$ $s_2$) out)

(-> (int int) int)

ex1 (plus $\boxed{10}^{\text{int}}$ $\boxed{22}^{\text{int}}$)

ex2 (plus $\boxed{10}$ $\boxed{\text{true}}$) — int, bool

$e_1^{s_1}$ $e_2^{s_2}$ out = t

int ~ int ✓

bool ~ int ✗

```
fn infer_app<A: Span>(
    ann: &A,
    env: &TypeEnv,
    subst: &mut Subst,
    poly: Poly,
    args: &[Expr<A>],
) -> Result<Ty, Error> {
    // 1. Infer the types of input `args` as `in_tys`
    let mut in_tys = vec![];
    for arg in args {
        in_tys.push(infer(env, subst, arg)?);
    }
    // 2. Generate a variable for the unknown output type
    let out_ty = subst.fresh();

    // 3. Unify the actual input-output `(-> (in_tys) out_ty)` with the
expected `mono`
    let mono = subst.instantiate(&poly);
    unify(ann, subst, &mono, &fun(in_tys, out_ty.clone()))?;

    // 4. Return the (substituted) `out_ty` as the inferred type of the
expression.
    Ok(out_ty.apply(subst))
}
```

sign

(-> ($t_1$ ... $t_n$) t)

~

(-> ($s_1$ ... $s_n$) out)

actual args

f :: ∀ (a..) (-> ($t_1$...$t_n$) $\boxed{t}$)

0 Fresh inst for $a_1 \dots a_k$

f $e_1$ $e_2$ ... $e_n$

1 infer $e_1 \dots e_n$

$s_1 \dots s_n \leftarrow$ typ($e_1 \dots e_n$)

2 UNIFY

$s_1 \sim t_1 \dots s_n \sim t_n$

3 apply SUBST to $\boxed{t}$

The code works as follows:

1. Infer the types of the inputs `args` as `in_tys`,
2. Generate a template `out_ty` for the unknown output type
3. Unify the actual input-output `(-> (in_tys) out_ty)` with the expected `mono`
4. Return the (substituted) `out_ty` as the inferred type of the expression.

swap :: ∀ (a b) (-> ((vec a b)) (vec b a))

1 (-> ((vec $a_0$ $b_0$)) (vec $b_0$ $a_0$))

2 (vec int bool)

3 (-> ((vec int bool)) out)

~

(-> ((vec $a_0$ $b_0$)) (vec $b_0$ $a_0$))

[ $a_0 \mapsto$ int, $b_0 \mapsto$ bool, out $\mapsto$ (vec bool int)]

swap (vec 10 true) ⟹ (vec bool int)

## Plan

1. Types
2. Expressions
3. Variables & Substitution
4. Unification
5. Generalize & Instantiate
6. Inferring Types
7. **Extensions**

# Extensions

(vec $e_1$ $e_2$) "vec" ∀ (a b) (-> (a b) (vec a b))

(→ ($a_0$ $b_0$) (vec $a_0$ $b_0$))

~

The above gives you the basic idea, now *you* will have to implement a bunch of extensions.

(→ (int bool) out)

$s_1$ $s_2$

1. Primitives e.g. `add1`, `sub1`, comparisons etc.
2. (Recursive) Functions
3. Type Checking

[$a_0$ ↦ int, $b_0$ ↦ bool,
out ↦ (vec int bool)]

## Extensions: Primitives

bool a a

(if $e_1$ $e_2$ $e_3$)

What about *primitives* ?

(∀ (a) (→ (bool a a) a))

- `add1(e)`, `print(e)`, `e1 + e2` etc.

What about *branches* ?

(vec-get e 0)  ∀(a b) (→ ((vec a b)) a)

- `if cond: e1 else: e2`

(vec-get e 1)  ∀(a b) (→ ((vec a b)) b)

What about *tuples* ?

- `(e1, e2)` and `e[0]` and `e[1]`

All of the above can be handled as **applications** to special functions.

For example, you can handle `add1(e)` by treating it as passing a parameter `e` to a function with type:

```
(-> (int) int)
```

Similarly, handle `e1 + e2` by treating it as passing the parameters `[e1, e2]` to a function with type:

```
(-> (int int) int)
```

Can you figure out how to similarly account for branches, tuples, etc. by filling in suitable implementations?

## Extensions: (Recursive) Functions

Extend or modify the code for handling `Defn` so that you can handle recursive functions.

- You can basically reuse the code as is
- **Except** if `f` appears in the body of `e`

Can you figure out how to modify the environment under which `e` is checked to handle the above case?

## Extensions: Type Checking

While inference is great, it is often useful to *specify* the types.

While inference is great, it is often useful to *specify* the types.

- They can describe behavior of *untyped code*
- They can be nice *documentation*, e.g. when we want a function to have a more *restrictive* type.

## Assuming Specifications for Untyped Code

For example, we can **implement** lists as tuples and tell the type system to:

- **trust the implementation** of the core list library API, but
- **verify the uses** of the list library.

We do this by:

```
;; list "stdlib" (unchecked) -----------------------------------------------------
(defn (nil) (as (forall (a) (-> () (list a))))
  false)

(defn (cons h t) (as (forall (a) (-> (a (list a)) (list a))))
  (vec h t))

(defn (head l) (as (forall (a) (-> ((list a)) a)))
  (vec-get l 0))

(defn (tail l) (as (forall (a) (-> ((list a)) (list a))))
  (vec-get l 1))

(defn (isnil l) (as (forall (a) (-> ((list a)) bool)))
  (= l false))

;; ---------------------------------------------------

(defn (length l)
  (if (isnil l)
      0
      (+ 1 (length (tail l)))))

(let (l0 (cons 0 (cons 1 (cons 2 (nil)))))
  (length l0))
```

The `as` keyword tells the system to **trust** the signature, i.e. to **assume** it is ok, and to **not check** the implementations of the function (see how `ti` works for `Assume` .)

However, the signatures are **used** to ensure that `nil` , `cons` and `tail` are used properly, for example, if we tried

```
(let (xs  (cons 10 (cons true (cons 30 (nil)))))
  (vec (head 10) (tail xs)))
```

we should get an error:

```
error: Type Error: cannot unify bool and int
    ┌─ tests/list2-err.snek:19:20
    │
19  │ (let (xs  (cons 10 (cons true (cons 30 (nil)))))
    │                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

## Checking Specifications

Finally, sometimes we may want to restrict a function be used to some more *specific* type than what would be inferred.

`garter` allows for specifications on functions using the `is` operator. For example, you may want a special function that just compares two `Int` for equality:

```
(defn (eqInt x y) (is (-> (int int) bool))
  (= x y))

(eqInt 17 19)
```

As another example, you might write a `swapList` function that swaps **pairs of lists** The same code would swap arbitrary pairs, but lets say you really want it work just for lists:

```
(defn (swapList p) (is (forall (a b) (-> ((vec (list a) (list b)))  (vec (list
b) (list a)))))
  (vec (vec-get p 1) (vec-get p 0)))

(let*
    ((l0 (cons 1 (nil)))
     (l1 (cons true (nil))))
  (swapList (vec l0 l1)))
```

Can you figure out how to extend the `ti` procedure to handle the case of `Fun f (Check s) xs e`, and thus allow for **checking type specifications**?

**HINT:** You may want to *factor* out steps 2-5 in the `infer_defn` definition --- i.e. the code that checks the `body` has type `out_ty` when `xs` have type `in_tys` --- into a separate function to implement the `infer` cases for the different `Sig` values.

This is a bit tricky, and so am leaving it as **Extra Credit**.

# Recommended TODO List

1. Copy over the relevant compilation code from `fdl`
    - Modify tuple implementation to work for pairs
    - You can remove the dynamic tests (except overflow!)
2. Fill in the signatures to get inference for `add1`, `+`, `(if ...)` etc
3. Complete the cases for `vec` and `vec-get` to get inference for pairs.
4. Extend `infer` to get inference for (recursive) functions.

5. Complete the `Ctor` case to get inference for constructors (e.g. `(list a)`).

6. Complete `check` to implement **checking** of user-specified types **(extra credit)**