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**

\[
\text{(defn (incr x) (+ x 1)) (\Rightarrow (\text{int}) \text{int})}
\]

**(incr input)**

---

**Example 2: Polymorphism**

\[
\text{(defn (id x) x) (\Rightarrow (\text{forall})(a) (\Rightarrow (a) a))}
\]

\[
\text{(let* ((a1 (id 7)) (\Rightarrow (\text{int}) \text{int})}
\text{(a2 (id true))) (\Rightarrow (\text{bool}) \text{bool})}
\text{true)}
\]

---

**Example 3: Higher-Order Functions**
Example 4: Lists

;; --- an API for lists ------------------------------------------
(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))

;; --- computing with lists -------------------------------------
(defn (length xs) (if (isnil xs)
  0
  (+ 1 (length (tail xs)))))
(defn (sum xs) (if (isnil xs)
  0
  (+ (head xs) (sum (tail xs)))))
(let (xs (cons 10 (cons 20 (cons 30 (nil)))))
  (vec (length xs) (sum xs)))

Strategy Recap

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

\[ e \Rightarrow t \quad \Rightarrow (a \ b) \equiv \Rightarrow (c) \ int \]
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, let's see how the syntax of our *garter* changes to enable static types.

**Syntax of Types**

A *Type* is one of:

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

Here *TyCtor* and *TyVar* are just string names:

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

pub struct TyCtor(String); // e.g. "List", "Tree"
```
Finally, a **polymorphic type** is represented as:

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

**Example: Monomorphic Types**

A function that
- takes two input Int
- returns an output Int

Has the *monomorphic* type \((\forall a \rightarrow (\text{Int} \times \text{Int}) \rightarrow \text{Int})\)

Which we would represent as a Poly value:

```rust
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 \((\forall a \rightarrow (a) \rightarrow (a) a)\)

Which we would represent as a Poly value:

```rust
code
forall(a, vec![], 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 \((\forall a b \rightarrow (a) (b) a)\) which is represented as:
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 \((\text{vec } e1 \ e2 \ e3)\) as \((\text{vec } e1 \ (\text{vec } e2 \ e3))\).
- **Tuple** access is limited to the fields zero and one (instead of arbitrary expressions).

```rust
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

\[ \alpha_1, \alpha_2 \]

- 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.

Let's formalize substitutions, and use it to define unification.

**Representing Substitutions**

We represent substitutions as a record with two fields:

```rust
struct Subst {
    map: HashMap<TyVar, Ty>,
    idx: usize,
}
```

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

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

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

**Applying Substitutions**

The main use of a substitution 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:
and then we define how to apply substitutions to Type, Poly, and lists and maps of Type and Poly.

For example,

\[
(\rightarrow (\text{int } z) \text{ bool})
\]

\[
(\rightarrow (a z) b)
\]

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

returns a type like

\[
\begin{bmatrix}
  a & \rightarrow & \text{int}
  \\
  b & \rightarrow & \text{bool}
  \\
  c & \rightarrow & (\rightarrow (\text{int } \text{int}) \text{int})
\end{bmatrix}
\]

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

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

**QUIZ**

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")));
```

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

\[
(\forall a . (\rightarrow (a) a))
\]

\[
[a. \rightarrow \text{int}]
\]
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:

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

```rust
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

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

**Extending Substitutions**

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

```rust
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
Unification

Next, let's use `subst` to implement a procedure to `unify` two types, i.e. to determine the conditions under which the two types are the same.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>Unified</th>
<th>Substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>Int</td>
<td>Int</td>
<td>empSubst</td>
</tr>
<tr>
<td><code>a</code></td>
<td>Int</td>
<td>Int</td>
<td><code>a</code> a → Int</td>
</tr>
<tr>
<td><code>a</code></td>
<td><code>b</code></td>
<td><code>b</code></td>
<td>a → b</td>
</tr>
<tr>
<td>a → b</td>
<td>a → d</td>
<td>a → d</td>
<td>b → d</td>
</tr>
<tr>
<td>a → Int</td>
<td>Bool → b</td>
<td>Bool → Int</td>
<td><code>a</code> a → Bool, b → Int</td>
</tr>
<tr>
<td>Int</td>
<td>Bool</td>
<td>Error</td>
<td>Error</td>
</tr>
<tr>
<td>Int</td>
<td>a → b</td>
<td>Error</td>
<td>Error</td>
</tr>
<tr>
<td><code>a</code></td>
<td>a → Int</td>
<td>Error</td>
<td>Error</td>
</tr>
</tbody>
</table>

- 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:

```rust
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 \( t_1 \) the same as \( t_2 \),
• or returns an error if the types cannot be unified.

The code is pretty much the table above:

```rust
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)) => {
            unifyys(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 => unifyys(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

• unifyys recursively calls unify on sequences of types:
• var_assign extends su with \([a |\rightarrow t]\) if required and possible!
fn var_assign<\(\text{ann}: \&\text{A}, \text{subst}: \&\text{mut} \ \text{Subst}, \ a: \&\text{TyVar}, \ t: \&\text{Ty}\) -> Result<\(), \ \text{Error}> 
\{
    if *t == Ty::Var(a.clone()) {
        Ok::<\(), \ \text{Error}>(())
    } else if t.free_vars().contains(a) {
        Err(\(\text{Error::new(ann.span(), "occurs check error".to_string())}\))
    } else {
        subst.extend(a, t);
        Ok::<\(), \ \text{Error}>(())
    }
\}

We can test out the above table:

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

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

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

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

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

Generalize and Instantiate

Recall the example:

\[
\text{id: } (\forall (a) \rightarrow (a) a) \\
(\text{defn (id x) x}) \\
(\text{let* ((a1 (id 7)) (a2 (id true))) true}) \\
\Rightarrow (\forall (a) \rightarrow (a) a)
\]

For the expression \((\text{defn (id x) x})\) we inferred the type \((\rightarrow (a0) a0)\)

We needed to generalize the above:

- to assign \(\text{id}\) the Poly-type: \((\forall (a0) \rightarrow (a0) a0)\)

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

- at \((\text{id 7})\) the function \(\text{id}\) has type \(\rightarrow (\text{int}) \text{int}\)
- at \((\text{id true})\) the function \(\text{id}\) has type \(\rightarrow (\text{bool}) \text{bool}\)

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
i.e. a Map from program variables \( \text{Id} \) to their (inferred) Poly type.

**Generalize**

We can now implement `generalize` as:

```rust
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:
Instantiate

Next, to instantiate a Poly of the form

\[\forall(a_1, \ldots, a_n, ty)\]

we:

1. Generate fresh type variables, \(b_1, \ldots, b_n\) for each "parameter" \(a_1 \ldots a_n\)
2. Substitute \([a_1 \mapsto b_1, \ldots, a_n \mapsto b_n]\) in the "body" \(ty\).

For example, to instantiate

\[\forall(\text{vec![}tv("a")]\text{), fun(\text{vec![}tyv("a")]\text{, tyv("a")})}\]

we

1. Generate a fresh variable e.g. "a66",
2. Substitute \(["a\mapsto "a66"]\) in the body \(["a"\mapsto "a"]\)
to get

\[
\text{fun}(\text{vec!}[\text{tyv}("a66")], \text{tyv}("a66"))
\]

### Implementing Instantiate

We implement the above as:

```rust
def 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!

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

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

let ty0 = subst.instantiate(&t_id); (a \rightarrow b) \sim (\forall (b : \text{int})\ a : \text{bool})
let ty1 = subst.instantiate(&t_id); b : \text{int}
let ty2 = subst.instantiate(&t_id);

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

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

### Plan

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

The top-level type-checker looks like this:

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

\[
\text{Num}(\_) \mid \text{Input} \Rightarrow \text{Ty::Int}, \\
\text{True} \mid \text{False} \Rightarrow \text{Ty::Bool},
\]

Inference: Variables

\[
\text{id} : (\forall a \to (a \to a))
\]

\[
x : \text{int}
\]

\[
(id \ x)
\]
For identifiers, we

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

\[
\text{Var}(x) \Rightarrow \text{subst.instantiate}(	ext{env.lookup}(x)?)
\]

Why do we **instantiate**? Recall the **id** example!

**Inference: Let-bindings**

Next, lets look at let-bindings:

\[
\text{Let}(x, e_1, e_2) \Rightarrow \{
\begin{align*}
&\text{let } t_1 = \text{infer}(	ext{env}, \text{subst}, e_1)\? ; \quad // (1) \\
&\text{let } \text{env}_1 = \text{env}.\text{apply}(\text{subst}) ; \quad // (2) \\
&\text{let } s_1 = \text{generalize}(\&\text{env}_1, t_1) ; \quad // (3) \\
&\text{let } \text{env}_2 = \text{env}_1.\text{extend}(\&[(x.\text{clone}(), s_1)]) ; \quad // (4) \\
&\text{infer}(\&\text{env}_2, \text{subst}, e_2)\? ; \quad // (5)
\end{align*}
\]

In essence,

1. **Infer** the type \( t_1 \) for \( e_1 \),
2. **Apply** the substitutions from (1) to the **env** ,
3. **Generalize** \( t_1 \) to make it a **Poly** type \( s_1 \),
   ○ *why? recall the id example*
4. **Extend** the env to map \( x \) to \( s_1 \) and,
5. **Infer** the type of \( e_2 \) in the extended environment.

**Inference: Function Definitions**

Next, lets see how to infer the type of a function i.e. \( \text{Lam} \)

\[
(\text{let } (\text{id } (\text{fn } (x) x)) )
\]

\[
\begin{align*}
&\text{0 } (\rightarrow (a_0) a_0) \quad // \text{"template"} \\
&\text{1 } \text{using } \text{env } + x_1 \rightarrow a_1 \ldots x_n \rightarrow a_n \\
&\text{INFER (body)} \Rightarrow \text{out} \\
&\rightarrow (a_1 \ldots a_n) \text{ out}
\end{align*}
\]
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)

Inference: Function Calls

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

\[
\text{Ex}_0 \quad (\text{id} : (\forall a. (\rightarrow (a) a)))
\]

\[
\begin{align*}
\text{id} & \equiv (\rightarrow (b) b) \\
(id & \ 10) & \equiv (\rightarrow (\text{int}) \text{ out}) \\
\text{out} & \equiv \text{int}
\end{align*}
\]

\[
\text{Ex}_0 \quad (\text{if} \ e_1 \ e_2 \ e_3)
\]

\[
\begin{array}{c}
t \ 1 \ 2 \ 3
\end{array}
\]
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))
}

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.

### Plan

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

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

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

**Extensions: Primitives**

What about *primitives*?

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

What about *branches*?

- if cond: e1 else: e2

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:

\( (\rightarrow \text{(int)} \text{ int}) \)

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

\( (\rightarrow \text{(int int)} \text{ int}) \)

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.

- 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:
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.

\textit{Garter} allows for specifications on functions using the \texttt{is} operator. For example, you may want a special function that just compares two \texttt{Int} for equality:

\begin{verbatim}
(defn (eqInt x y) (is (-> (int int) bool))
  (= x y))
(eqInt 17 19)
\end{verbatim}

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

\begin{verbatim}
(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)))
\end{verbatim}

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

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

This is a bit tricky, and so am leaving it as \textbf{Extra Credit}.

\section*{Recommended TODO List}

1. Copy over the relevant compilation code from \texttt{fdl}
   \begin{itemize}
   \item Modify tuple implementation to work for pairs
   \item You can remove the dynamic tests (except overflow!)
   \end{itemize}

2. Fill in the signatures to get inference for \texttt{add1}, \texttt{+}, \texttt{(if \ldots)} etc

3. Complete the cases for \texttt{vec} and \texttt{vec-get} to get inference for pairs.

4. Extend \texttt{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**)