CSE 231: Type Inference (Harlequin)

Ranjit Jhala *June 2, 2026*

Harlequin

`fer-de-lance` with two changes:

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

That is,

- extend `checker` with a proper type system,
- `compile` *only* if code type-checks.

Why?

What are some benefits of compile-time type checking?

Strategy

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.

Example 1: Inputs and Outputs

```
(defn (incr x) (+ x 1))

(incr input)
```

Example 2: Polymorphism

```
(defn (id x) x)

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

Example 3: Higher-Order Functions

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

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

(f incr 10)
```

Example 4: An API for Lists

```
(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))
```

Example 5: Using the List API

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

(defn (sum xs)
  (if (= xs nil)
    0
    (+ (head xs) (sum (tail xs)))))

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

Strategy

1. **Traverse** the `Expr` ...
2. **Fresh** variables for unknown types...
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. Substitution
4. Unification
5. Generalize & Instantiate
6. Inferring Types
7. Extensions

Types

First, lets add syntax for **types**

```
<ty> ::= int
       | bool
       | (-> (<ty>*) ty)      ; (-> (int int) int)
       | (vec <ty> <ty>)      ; (vec int bool)
       | <tvar>               ; a
       | (<ctor> <ty>*)       ; (list int)
```

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

```
<poly> ::= (forall (<tvar>+) <ty>)
```

Example: Monomorphic Type

A function that takes two input `int` and returns an output `int` has the *mono-type*

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

Example: Polymorphic Type

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))
```

Similarly, a function that takes two values and returns the first can be given a *polymorphic* type

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

Syntax of Expressions

To enable inference lets *simplify* the language.

- *Dynamic tests* `isNum` and `isBool` are unnecessary,
- *Tuples* have exactly *two* elements,
- *Tuple* access is limited to the fields `Zero` and `One`.

Strategy

Our informal algorithm proceeds by

- 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.

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

Substitutions

A **substitution** is simply a **map** from *type variables* to *types*
For example, a substitution `sub0` that maps `a`, `b` and `c` to `int`, `bool` and `(-> (int int) int)` respectively.

```
[ a := int
, b := bool
, c := (-> (int int) int)
]
```

Operations on Substitutions

1. **Apply**
2. Empty
3. Extend

Applying Substitutions

We will **apply** a *substitution* to a *type* **apply** to
- *replace* each occurrence of a ty-var with substituted value,
- or *preserve* it if not mentioned in the substitution.

QUIZ: Substitution

For example, suppose we have

```
sub0 := [a := int, b := bool, c := (-> (int int) int)]
ty0  := (-> (a z) b)
```

What is the result of *applying* the `sub0` to `ty0`?

```

```

QUIZ: Substitution

For example, suppose we have

```
sub0 := [a := int, b := bool, c := (-> (int int) int)]
ty1  := (forall (a) (-> (a) a))
```

What is the result of *applying* the `sub0` to `ty1`?

1. `(forall (a) (-> (int) bool)`
2. `(forall (a) (-> (a)   a   )`
3. `(forall (a) (-> (a)   bool)`
4. `(forall (a) (-> (int) a   )`
5. `(forall ( ) (-> (Int) bool)`

Bound vs. Free Type Variables

`(forall (a) (-> (a) a))` *is same as* `(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**.

Operations on Substitutions: Empty

1. Apply
2. **Empty**
3. Extend

The **empty** substitution is just an **empty map** (duh.)

Operations on Substitutions: Extend

Extend `sub` by assigning a variable `a` to type `t`

```
sub [ a := t ]
```

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:

1. *Create* a new substitution `[a := int]`
2. *Apply* it to each binding in `sub` to get `[b := int]`
3. *Insert* it to get extended `[b := int, a := int]`

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`	`[]`
`a`	`Int`	`int`	`a := int`
`a`	`b`	`b`	`a := b`
`-> (a) b`	`-> (a) d`	`-> (a) d`	`b := d`

T1	T2	Unified	Substitution
`-> (a) int`	`-> (bool) b`	`-> (bool) int`	`a := bool, b := int`
`Int`	`Bool`	*Error*	*Error*
`Int`	`a -> b`	*Error*	*Error*
`a`	`a -> Int`	*Error*	*Error*

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

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?

Plan

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

Generalize and Instantiate

Recall the example:

```
(defn (id x) x)

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

For `(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 type variables,
2. Remove those constrained by *other* in-scope variables.

We represent the types of **in scope** program variables as a **type environment**: a map from program variables to (inferred) *polymorphic* type.

```
<env> := [<id_1> := <poly_1>, ..., <idn> := <poly_n>]
```

Generalize

We can now implement `generalize(env, ty)` as:

1. Compute (free) type variables of `ty`
2. Compute (free) type variables of `env`
3. Unconstrained = (1) - (2)
4. Slap a `forall` on the unconstrained variables.

QUIZ: Free Variables of a Type

Lets fill in the implementation of `free_vars` which computes the set of variables that appear inside a (poly) type.

```
/* Free Variables of a Type */
fn free_vars(ty: &Ty) -> HashSet<TyVar> {
  match ty {
    int | bool =>

    a =>

    (-> (t0...) t) =>

    (forall (a0...an) t) =>

  }
}

/* Free Variables of a Env */
fn free_vars_env(env: &Env) -> HashSet<TyVar> {
  let mut res = HashSet::new();



  res
}
```

Instantiate

To **instantiate** a poly-type `(forall (a1...an) ty)` we:

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

For example, to instantiate

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

we:

1. Generate a fresh variable e.g. `a66`,
2. Substitute `[a := a66]` in the body

to get

```
(-> (a66) a66)
```

Plan

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

Inference

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

```
fn infer(env: &TypeEnv, subst: &mut Subst, e: &Expr)
   -> 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.

```
n     => int,
true  => bool,
false => bool,
input => int,
```

QUIZ: Inference: Variables

For identifiers, `x` we just **lookup** their type in the `env`.

`env`	`[ten := int]`
`subst`	`[]`
`expr`	`ten`
`infer(...)`	

Inference: Function Calls

To infer the type of a function call `(f e1...en)`

1. **Instantiate** poly-type of `f` in `env` as `(-> (s1...sn) out)`
2. **Infer** the *actual* types of the `e1...en` as `t1...tn`
3. **Unify** `(-> (s1...sn) out)` with `(-> (t1...tn) out)`
4. **Return** the (substituted) `out` as the inferred type.

QUIZ: Infer Call
Fill in the blanks with the inferred type

`env`	`[incr := (-> (int) int)]`
`subst`	`[]`
`expr`	`(incr 99)`
`infer(...)`	

`env`	`[id := (forall (a) (-> (a) a))]`
`subst`	`[]`
`expr`	`(id 10)`
`infer(...)`	

QUIZ: Why do we *instantiate*?

Inference: Function Definitions

For function definitions `(defn (f x1...xn) body)` we

1. **Generate** a *function type* with fresh `t1..tn` and `out_ty`,
2. **Extend** `env` so `x1..xn` have (fresh) `t1..tn`,
3. **Infer** 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.

QUIZ: Inference for Definitions
Fill in the blanks with the inferred type

`env`	`[incr := (-> (int) int)]`
`subst`	`[]`
`expr`	`(fn (x) (incr x))`
`infer(...)`	

`env`	`[incr := (-> (int) int)]`
`subst`	`[]`
`expr`	`(fn (y) y)`
`infer(...)`	

Inference: Let-bindings

For let-bindings `(let (x e1) e2)` we

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`,
4. **Extend** the `env` to map `x` to `s1` and,
5. **Infer** the type of `e2` in the extended environment.

QUIZ: Inference for Let-Bindings

`env`	`[incr := (-> (int) int)]`
`subst`	`[]`
`expr`	`(let (foo (fn (x) (incr x)))` `  (foo 5))`
`infer(...)`	

`env`	`[incr := (-> (int) int)]`
`subst`	`[]`
`expr`	`(let (id (fn (y) y))` `  (id 9))`
`infer(...)`	

Extensions

The above gives you the basic idea, now you can 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:

```
(-> (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.

- 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 (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 (length l)
  (if (= l nil)
      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.

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 ...)))
   │                    ^^^^^^^^^^^^^^^^
```

Checking Specifications

Sometimes we may want to restrict a function to 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)
```