Homework 6
Infer Your Types

98-317: Hype for Types

Due: 28 Februrary 2018 at 6:30 PM

Introduction

This week we discussed type inference and how it differs from type checking and type synthesis. We discovered that type inference naturally relies on the problem of unification, then looked more closely at examples of unification. Finally, we went over an algorithm for performing unification and discussed tips on how to use unification to perform type inference. The lecture slides are linked on the course website, and it may be useful to refer back to them.

This homework is divided into three parts: Required, Useful, and Fun. You will receive credit for this homework if you turn in something (not necessarily something working) for the "required" portion.

Code solutions to the problems in this assignment will be released next week.

Turning in the Homework Run the command

```
tar cf handin.tar Inference.sml ReverseInference.sml
```

inside your hw6 directory, then submit `handin.tar` to Autolab.

Lambda++

In this assignment we will use a simple programming language with functions, sums, and products, which we will call Lambda++. It differs from the language called Lambda++ in last week's homework in that it no longer requires (or even supports) type annotations. Here is its syntax:

Type	τ	$::=$	α	type variable
			$\tau \to \tau$	arrow type
			$\tau + \tau$	sum type
			$\tau \times \tau$	product type
Expression	e	$::=$	x	variable
			`fn` $x \Rightarrow e$	lambda
			$e\ e$	application
			#1 e	left projection
			#2 e	right projection
			`INL` e	left injection
			`INR` e	right injection
			`case` e `of INL` $x \Rightarrow e$ `|` `INR` $x \Rightarrow e$	case analysis

We implement this syntax with the following SML datatypes:

```
datatype typ =
  TypeVariable of int
| Arrow of typ * typ
| Plus of typ * typ
| Times of typ * typ

datatype exp =
  Variable of string
| Lambda of string * exp
| Apply of exp * exp
| Tuple of exp * exp
| First of exp
| Second of exp
| Left of exp
| Right of exp
| Case of exp * (string * exp) * (string * exp)
```

Note that we use integers to represent type variables, while we use strings to represent expression variables. This is pretty arbitrary and there's nothing interesting going on here; for example you can think of an SML value `Arrow (TypeVariable 98, TypeVariable 317)` as representing a type of the form $\alpha \to \beta$.

Here are the rules defining the statics of Lambda++:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fn}\ x \Rightarrow e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \to \tau_2 \quad \Gamma \vdash e' : \tau_1}{\Gamma \vdash e\ e' : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \#1\ e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \#2\ e : \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \texttt{INL}\ e : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \texttt{INR}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 \quad \Gamma, x_2 : \tau_2 \vdash e_2}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \texttt{INL}\ x_1 \Rightarrow e_1\ \texttt{|}\ \texttt{INR}\ x_2 \Rightarrow e_2}$$

Required

Required Task In `Inference.sml`, implement the function

```
unify : typ * typ -> (int * typ) list
```

which performs unification. If the types `t1` and `t2` can be unified, then `unify (t1, t2)` should evaluate to a list of variable mappings that, when applied (from left to right) to `t1` and `t2`, result in the same type. If `t1` and `t2` can't be unified, then `unify (t1, t2)` should raise `TypeError msg` where `msg` is an error message.

We've provided a few helper functions:

- `occurs : int -> typ -> bool` checks whether a type variable appears anywhere in a type. `occurs a t` returns whether or not the type variable `a` occurs in the type `t`.
- `subst : (int * typ) * typ -> t` substitutes type variables in a type. `subst ((a, t), tt)` replaces all occurrences of the type variable `a` in the type `tt` with the type `t`. This function is uncurried to make it easy to use with `foldl` to apply a list of variable mappings to a type.

Useful

Useful Task Also in `Inference.sml`, implement the function

`infer : exp -> typ`

which performs type inference. If the expression `e` can have a type per the statics rules of Lambda++, then `infer e` should evaluate to the most general such type, up to renaming of type variables. If `e` can not have a type, then `infer e` should raise `TypeError` with some error message.

You can use the `Top.infer` to test your `Inference.infer` function. It parses a given expression, calls your `infer` function, then prints out the inferred type (or the error message if `TypeError` is raised). Here are a few examples:

```
$ sml -m sources.cm
...
[New bindings added.]
- Top.infer "fn x => x";
(a -> a)
val it = () : unit
- Top.infer "fn x => y";
Type error: unbound variable: y
val it = () : unit
- Top.infer "fn x => fn y => x";
(a -> (b -> a))
val it = () : unit
- Top.infer "fn x => fn y => (x, INL y)";
(a -> (b -> (a * (b + c))))
val it = () : unit
```

Hints:

- The TC and EC structures (short for "type context" and "expression context") are dictionary data structures which you may want to use. The interface for these dictionaries is provided at the end of this document.
- Review the last few lecture slides, where we provide tips on implementing type inference.
- You'll probably want to implement a recursive helper function which passes around at least one dictionary.

Fun

Now that we've written a function from expressions to types, I wonder if we can write a function from types to expressions? That would be like... reverse type inference?

It turns out "reverse type inference" is decidable (for the Lambda++ type system, at least)[1][2]. But it's still pretty hard.

Fun Task In `ReverseInference.sml` implement the function

```
reverse_infer : typ -> exp option
```

If the type `t` is inhabited by a Lambda++ expression, then `reverse_infer t` should evaluate to `SOME e` where `e` is an expression of type `t`. If `t` is not inhabited by any Lambda++ expressions, then `reverse_infer t` should evaluate to `NONE`.

You can use `Top.reverse_infer` to test your `ReverseInference.reverse_infer` function. Here are a few examples:

```
- Top.reverse_infer "a -> a";
fn a => a
val it = () : unit
- Top.reverse_infer "a -> b";
Uninhabited type.
val it = () : unit
- Top.reverse_infer "(a -> b) -> a -> b";
fn c => fn d => (c d)
val it = () : unit
- Top.reverse_infer "(a + b -> c) -> (a -> c) * (b -> c)";
fn e => (fn f => (fn g => (e INL g) f), fn i => (fn k => (e INR k) i))
val it = () : unit
```

[1]By the Curry-Howard correspondence, reverse type inference is more commonly referred to as automated theorem proving

[2]I made up the term "reverse type inference".

Appendix: Dictionary Interface

```
signature DICT =
   sig

      type key
      type 'a dict

      exception Absent

      val empty : 'a dict
      val singleton : key -> 'a -> 'a dict
      val insert : 'a dict -> key -> 'a -> 'a dict
      val remove : 'a dict -> key -> 'a dict
      val find : 'a dict -> key -> 'a option
      val lookup : 'a dict -> key -> 'a
      val union : 'a dict -> 'a dict -> (key * 'a * 'a -> 'a) -> 'a dict

      val operate :
        'a dict -> key -> (unit -> 'a) -> ('a -> 'a) -> 'a option * 'a * 'a dict
      val insertMerge : 'a dict -> key -> 'a -> ('a -> 'a) -> 'a dict

      val isEmpty : 'a dict -> bool
      val member : 'a dict -> key -> bool
      val size : 'a dict -> int

      val toList : 'a dict -> (key * 'a) list
      val domain : 'a dict -> key list
      val map : ('a -> 'b) -> 'a dict -> 'b dict
      val foldl : (key * 'a * 'b -> 'b) -> 'b -> 'a dict -> 'b
      val foldr : (key * 'a * 'b -> 'b) -> 'b -> 'a dict -> 'b
      val app : (key * 'a -> unit) -> 'a dict -> unit

   end
```