IIT CS440: Programming Languages and Translators

Homework 5: Types and Unification

Prof. Stefan Muller
TA: Xincheng Yang

Out: Thursday, Apr. 8
Due: Tuesday, Apr. 20 11:59pm CDT

Updated Apr. 16

This assignment contains 5 written tasks and 2 programming tasks, for a total of 55 points, in addition to a maximum of 0 bonus points.

0 Logistics and Submission - Important

The same rules as on HW2-4 apply. In particular:

1. Make sure you read and understand the updated/clarified collaboration policy on the course website.
2. The complicated skeleton code of this assignment will make testing in the `ocaml` toplevel or on Try-OCaml very difficult. Instructions for testing your code are provided in the writeups of the programming problems.
3. Your answers to the programming problems will go in the files `unify.ml` and `typecheck.ml`. You only need to submit this file and your written pdf. Do not rename the .ml files.

Bonus: Walkthrough videos. We've included two videos on Blackboard with the assignment. One walks through the skeleton code distributed with the assignment and the other demonstrates testing for this assignment. They're not strictly necessary to complete the assignment (the testing instructions are in the writeup as well), but if you're confused or curious about how all the code fits together, we encourage you to watch them.

1 STLC

In this section, refer to the syntax and typing rules for the Simply-typed λ calculus, given in lecture.
Task 1.1 (Written, 12 points).

Give the STLC types of the following expressions:

(a) $\lambda x : \mathsf{unit}.x$

(b) $((), \lambda x : \mathsf{unit}.x)$

(c) $\lambda x : \mathsf{unit}.(x, x)$

(d) $\mathsf{snd}\ ((), ())$

Task 1.2 (Written, 9 points).

Give a typing derivation for $\bullet \vdash \lambda f : \mathsf{unit} \rightarrow (\mathsf{unit} \times \mathsf{unit}).\mathsf{fst}\ (f\ ()) : (\mathsf{unit} \rightarrow (\mathsf{unit} \times \mathsf{unit})) \rightarrow \mathsf{unit}$.
Updated Apr. 16: Resulting type was fixed

2 MiniCaml

Image Credit: Petr Kratochvil

In this section, we will be working with MiniCaml, a language with more features than MicrOCaml from Homeworks 3 and 4, but still not quite all the features of OCaml.

The grammar of MiniCaml is shown below.

$$
\begin{array}{lcl}
\mathit{op} & \rightarrow & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid < \mid \leq \mid > \mid \geq \mid \texttt{=} \mid \texttt{<>} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{\^{}} \\
\tau & \rightarrow & \texttt{int} \mid \texttt{string} \mid \texttt{bool} \mid \texttt{unit} \mid \tau\ \texttt{list} \mid \tau\ \texttt{->}\ \tau \mid \tau\ \texttt{*}\ \tau \\
e & \rightarrow & \mathit{var} \mid \mathit{num} \mid \mathit{string} \mid \texttt{true} \mid \texttt{false} \mid \texttt{()} \mid \texttt{[]} \mid e\ \mathit{op}\ e \mid \texttt{fun}\ \mathit{var}\ \texttt{->}\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
 & \mid & \texttt{let}\ \mathit{pat}\ \mathit{optannot}\ =\ e\ \texttt{in}\ e \mid \texttt{let}\ \mathit{var}\ \mathit{pat}\ \mathit{optannot}\ =\ e\ \texttt{in}\ e \\
 & \mid & \texttt{let rec}\ \mathit{var}\ \mathit{pat}\ \mathit{optannot}\ =\ e\ \texttt{in}\ e \mid \texttt{let}\ (\mathit{var}, \mathit{var}) = e\ \texttt{in}\ e \mid e\ e \\
 & \mid & \texttt{match}\ e\ \texttt{with []}\ \texttt{->}\ e\ \texttt{|}\ \mathit{var}\texttt{::}\mathit{var}\ \texttt{->}\ e \mid e, e \mid e\texttt{::}e \mid e\ \texttt{:}\ e \\
\mathit{pat} & \rightarrow & \mathit{var} \mid (\mathit{var}\ \texttt{:}\ \tau) \\
\mathit{optannot} & \rightarrow & \epsilon \mid \texttt{:}\ \tau \\
\mathit{decl} & \rightarrow & \texttt{let}\ \mathit{pat}\ \mathit{optannot}\ =\ e\texttt{;;} \mid \texttt{let}\ \mathit{var}\ \mathit{pat}\ \mathit{optannot}\ =\ e\texttt{;;} \mid \texttt{let rec}\ \mathit{var}\ \mathit{pat}\ \mathit{optannot}\ =\ e\texttt{;;} \mid e\texttt{;;} \\
\mathit{prog} & \rightarrow & \mathit{decl} \mid \mathit{decl}\ \mathit{prog}
\end{array}
$$

You may notice that you can actually write a pretty large subset of OCaml in MiniCaml without making any changes. In particular, we've gotten rid of MicrOCaml's odd `app` e `to` e syntax and replaced it with normal OCaml application. One nice result of this is that, while you can't necessarily take any OCaml program and run it in MiniCaml, you *can* run any MiniCaml program through OCaml (`ocaml` or TryOCaml) to figure out what types it should have or what the result should be. A couple non-obvious restrictions present in MiniCaml:

1. Pattern matching is limited to using `let` to break apart pairs and using `match` to match on a list (note that we haven't defined `fst` or `snd`: you have to break apart pairs with pattern matching; you can define them yourself though).

2. Functions (both lambdas and let-defined functions) can only take one argument. You can get around this with currying (though that doesn't work well for recursive functions) or having functions take pairs (see `map` in `examples/rec.ml`).

3. As in OCaml, a program consists of one or more top-level declarations, where a declaration can be a `let` declaration or an expression by itself. Unlike in OCaml, these declarations **must** be followed by two semicolons (see the midterm for why this makes our lives easier writing the parser).

4. You can't use type variables in annotations (e.g. `let f (x: 'a) : 'a = x`. MiniCaml can still infer polymorphic types with type variables though.

The type definitions for MiniCaml are given in `types.ml` and described in the walkthrough video on Blackboard.

The file `unify.ml` contains code to unify two types, which is used in type checking. Unification in MiniCaml works very similarly to unification in STLC, with a couple of extensions. Recall that unification takes two types τ_1 and τ_2, potentially with unification variables like $?_1$. These are "holes" that can be unified with anything. Each instance of a particular variable must be filled with the same thing though (for example, if $?_1$ appears in both types, you must replace $?_1$ with the same type in both). In class, we considered an imperative version of unification that "magically" updates unification variables. On this homework, we'll consider a functional unification algorithm that instead returns a *substitution*, which is a list of pairs $(?_i, \tau_i)$ meaning "replace $?_i$ with τ_i" (*Updated 4/14*: Typo fix). As with substituting values for variables, we can write $[\tau_i/?_i]\tau$ to mean "τ with all instances of $?_i$ replaced by τ_i." For a substitution σ, we'll write $[\sigma]\tau$ to mean "τ with all the replacements in σ". So, for example,

$$[[(?_0, \texttt{int}); (?_1, \texttt{string})]](?_0 \texttt{ -> } ?_1) = \texttt{int -> string}$$

and

$$[[(?_0, ?_1\ \texttt{list}); (?_1, \texttt{int})]]?_0 = \texttt{int list}$$

The OCaml definition for substitutions is given in `unify.ml`:

`type substitution = (int * typ) list`

In that definition, ?0 is represented as just the integer 0. We also include a function `sub_all : substitution -> typ -> typ`,where `sub_all s t` computes $[s]t$.

Here's the (recursive) unification algorithm:

Algorithm: Unify(τ_1, τ_2)
Returns a substitution or an error.

1. If τ_1 and τ_2 are both the same base type (`int`, `string`, `bool` or `unit`), return the empty list `[]`.

2. If $\tau_1 = \tau_2 = ?_i$ (i.e., they are the same unification variable), return `[]`.

3. If $\tau_1 = ?_i$, then:
   (a) If $?_i$ occurs in τ_2, then error.
   (b) Otherwise, return $[(?_i, \tau_2)]$.

4. If $\tau_2 = ?_i$, then:
   (a) If $?_i$ occurs in τ_1, then error.
   (b) Otherwise, return $[(?_i, \tau_1)]$.

   Updated 4/13: the return values of these two cases were swapped

5. If $\tau_1 = \tau_1'$ `list` and $\tau_2 = \tau_2'$ `list`, then return Unify(τ_1', τ_2')

6. If $\tau_1 = \tau_1'$ `->` τ_1'' and $\tau_2 = \tau_2'$ `->` τ_2'' then let σ = Unify(τ_1', τ_2') and return σ concatenated with Unify($[\sigma]\tau_1''$, $[\sigma]\tau_2''$).

7. Similar to above for if $\tau_1 = \tau_1' \mathtt{*} \tau_1''$ and $\tau_2 = \tau_2' \mathtt{*} \tau_2''$

8. Otherwise, error.

Case 6 above is worth discussing in more detail. When we unify τ_1' and τ_2', we get a substitution σ, which gives us several substitutions we need to perform. We need to perform those substitutions immediately on τ_1'' and τ_2'' before unifying them, because σ might tell us something like "replace $?_0$ with `unit`", and τ_1'' might have $?_0$ in it. To make this clearer, suppose $\tau_1 = \texttt{int -> string}$ and $\tau_2 = ?_0 \texttt{ -> } ?_0$. Then, we have $\tau_1' = \texttt{int}, \tau_2'' = ?_0, \tau_1'' = \texttt{string}$ and $\tau_2'' = ?_0$. When we unify `int` and $?_0$, we get the substitution $[(?_0, \texttt{int})]$. If we then go ahead and unify `string` and $?_0$, we'd get the substitution $[(?_0, \texttt{string})]$. Now, we have to somehow reconcile these two substitutions, which of course isn't possible because $?_0$ can't be both `int` and `string`. Instead, we unify $[[(?_0, \texttt{int})]]\texttt{string} = \texttt{string}$ and $[[(?_0, \texttt{int})]]?_0 = \texttt{int}$, which gives us an error (which is what we want because these two types can't be unified.)

Task 2.1 (Written, 14 points).

For each of the following pairs of types, say whether or not the two types can be unified. If they can, give the substitution that results. If not, briefly (in one sentence or so) describe why not.

(a)	`int *` $?_1$	$?_2$
(b)	`int *` $?_1$	$?_2$ `* string`
(c)	`int *` $?_1$	$?_1$ `* string`
(d)	`int ->` $?_1$	`string ->` $?_2$
(e)	$?_1$ `->` $?_2$	$?_3$ `*` $?_4$
(f)	$?_1$ `list`	$?_2$ `list list`
(g)	$?_1$ `list`	$?_1$

Task 2.2 (Programming, 15 points).

Implement the function `unify : typ -> typ -> typ` in `unify.ml`, following the algorithm above.

The file `unify.ml` also contains one other function you might want: `new_type : unit -> typ` generates a new type consisting of just a unification variable $?_n$ where $?_n$ hasn't been used in the program before. This is useful for when we need to "guess" types to unify later.

The file `typecheck.ml` includes code to typecheck MiniCaml programs. It uses the unification function you wrote above. We did most of the work, you just need to implement one small function that infers the types of constants.

Task 2.3 (Programming, 5 points).

Implement the function `type_of_const : const -> typ` in `typecheck.ml`. The function should return the type of the given constant. You shouldn't need to do any unification. In the case of `Nil`, which represents the empty list `[]`, the type is of course τ `list` for some τ. What's τ? We have no way of knowing at this point, so you'll just have to take a guess...

You can use `new_type`, as well as any other functions from `unify.ml`.

2.1 Testing

You can write tests using `assert` in the files you edited, as usual. When you're done, you can also use

```
make
```

to compile all of the code we gave you, which together makes a typechecker and interpreter for MiniCaml. You can run it using

```
./miniml <file>
```

Where `<file>` is a MiniCaml file. We've given you several examples in `examples`, as well as several examples that *should not typecheck* in `examples/error`. MiniCaml will read in the file, typecheck all of the declarations, print out their types, and then evaluate them and print out their values (as in the OCaml toplevel, functions will not be printed). For example:

```
$./miniml examples/rec.ml
sum: int list -> int
length: 'a list -> int
map: ('d -> 'c * 'd list) -> 'c list
int_to_string: int -> string
onetwothree: string list

sum = <fun>
length = <fun>
map = <fun>
int_to_string = <fun>
onetwothree = (one)::((two)::((three)::([])))
$
```

You can also use MiniCaml like the OCaml toplevel by giving no arguments:

```
$ ./miniml
Welcome to MiniCaml. Type #quit;; to exit.
# 3;;
-: int
- = 3
# (3, "Hi!");;
-: int * string
- = (3, "Hi!")
# fun x -> x;;
-: 'b -> 'b
- = <fun>
# #quit;;
```

3 Standard Written Questions

Task 3.1 (Written, 0 points).

How long (approximately, in hours/minutes of actual working time) did you spend on this homework, total? Your honest feedback will help us with future homeworks.

Task 3.2 (Written, 0 points).

Who, if anyone, did you collaborate with (and in what way), and what outside sources, if any, did you consult in working on this homework?