~~Phantom Types~~

Hype for Types: Lecture 3

Spring 2019

Password: Spooky

Polymorphism

Hype for Types: Lecture 3

Spring 2019

Wacky applications of

Polymorphism

Hype for Types: Lecture 3

Spring 2019

What is polymorphism?

- A piece of code is *polymorphic* if it satisfies two properties
  1. It works with values of any type.
  2. It treats all types the same.
- Examples:

```
fn x => x


fn (x, y) => x


fun map f [] = []
  | map f (x::xs) = (f x)::(map f xs)
```

Works with values of any type

- The types of polymorphic values are given by *type variables.*
- Type variables can be substituted with any type

```
(fn (x : ‘a) => x) : ‘a -> ‘a


(fn (x : ‘a, y : ‘b) => x) : ‘a * ‘b -> ‘a


fun map (f : ‘a -> ‘b) ([] : ‘a list) : ‘b list = []
  | map f (x::xs) = (f x)::(map f xs)
```

Should treat all types the same

If I have

- A relation R between two types τ_1 and τ_2
- A polymorphic function `p : ‘a -> anything`
- Some values $v_1 : \tau_1$ and $v_2 : \tau_2$ such that $R(v_1, v_2)$

then

$$\texttt{p}(\texttt{v}_1) \cong \texttt{p}(\texttt{v}_2)$$

(if you use R to check equality anywhere values of type τ_1 and τ_2 are compared)

Example

- τ_1 = bool and τ_2 = int
- R (x, y) = (x is false and y is 0) or (x is true and y is 1)
- `val p = fn x => x`

Does p(true) $\cong_R$ p(1)? ($\cong_R$ means $\cong$ but using R to compare ints and bools)

`(fn x => x) true`

$\cong_R$ `true` [stepping]

$\cong_R$ `1` [R(true, 1)]

$\cong_R$ `(fn x => x) 1` [backwards stepping]

It works! But why?

Let’s see how it could *not* be the case

1. It works with values of any type. Other languages have a form of polymorphism that involves overloading.

```
fun foo (x : int) : int = x + 1
fun foo (y : bool) : bool = not y

val 10 = foo 9
val false = foo true
```

2. It treats all types the same. Casing on a type, for example.

```
fun foo (x : ‘a) : int = if ‘a = int then 0 else 1
```

How is it typechecked?

Polymorphic functions are compiled to *functions from types to functions* (1) that *cannot case on their input* (2).

1. Functions on values have to accept any value of that type.
   Functions on types have to accept any type.

2. If the function can’t case on its input, it can’t treat different types differently.

Type functions

```
val id : forall ‘a.‘a -> ‘a = tfn ‘a => fn (x : ‘a) => x

val int_id : int -> int  = id[int]      [fn (x : int) => x / int_id]

val true = id[bool](true)
```

Datatypes?

They can be polymorphic too!

```
datatype 'a tree = Empty
                 | Node of 'a tree * 'a * 'a tree
```

- `tfn ‘a => fn (x : ‘a) => x` is a function from type to value
- `tree` is a function from type to type.

Who cares?

Why do we want polymorphism?

Why should we care that polymorphic functions

1. Work with values of any type
2. Treat all types the same

Wacky Applications of Polymorphism

Phantom types, free theorems, and typesafe refs

Phantom Types

Useless type parameters!

Phantom Types

Type declarations where the type parameter *is never used*

```
datatype ‘a index = Idx of int
                  | OutOfBounds of int


type ‘a integer = int
```

Why would you

ever

want that?

For lots of things!

To statically…

- Keep track of where data came from or where it’s going
- Enforce privacy or data sanitization policies
- Keep track of data structure properties
- Control access to operations on data structures

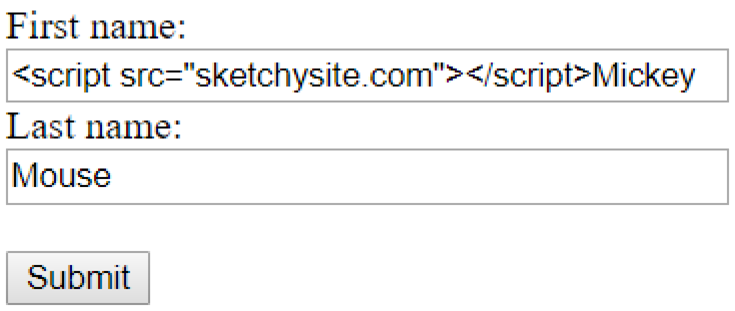
Example: Enforcing Sanitization

```
{ firstName: "<script src=\"sketchysite.com\"></script>Mickey",
  lastName: "Mouse" }

➔

 { firstName: "Mickey",
  lastName: "Mouse" }
```

User Input → Sanitize → Display on Website

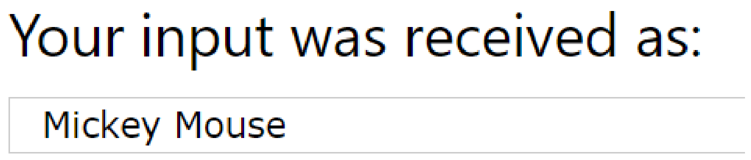


First name:

<script src="sketchysite.com"></script>Mickey

Last name:

Mouse



Your input was received as:

Mickey Mouse

```
signature INPUT = sig
  type unsanitized
  type sanitized
  type 'a input

  val getInput : unit -> unsanitized input
  val sanitize : (string -> string) -> 'a input -> sanitized input
  val print_to_user : sanitized input -> unit
end

structure Input :> INPUT = struct
  datatype unsanitized = U
  datatype sanitized = S
  type 'a input = string

  fun getInput () : unsanitized input =
      case TextIO.inputLine (TextIO.stdIn) of
          NONE => raise Fail "No user input at this time.\n"
        | SOME s  => s

  fun sanitize (sanitizer : string -> string) (s : 'a input) : sanitized input =
      sanitizer s

  fun print_to_user (s : sanitized input) : unit = print s
end
```

How does polymorphism help?

- Type parameter allows multiple input types without needing a function to convert between them
- Polymorphic functions must accept both sanitized and unsanitized inputs (Fact 1 of Polymorphism)
- The functions can’t case on whether their input is sanitized or unsanitized (Fact 2 of Polymorphism)

This makes the type parameter *invisible* at runtime. It is checked statically, and then it disappears. Efficient!

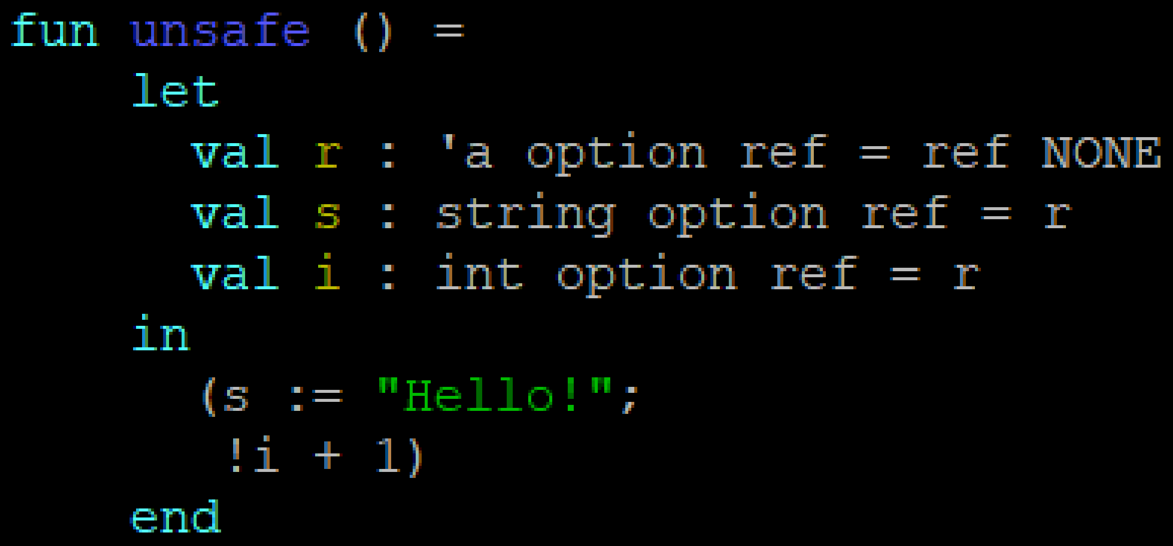
Free Theorems

Free Theorems

Theorems that you get about your code just that come from points (1) and (2) about polymorphism

- There is only one function of type `'a -> 'a` (up to equivalence)
- There are very few ways to write the map function that typecheck
  - map : ('a -> 'b) -> 'a list -> 'b list
- How many ways are there to write a function of type `'a * 'a -> 'a`?
  - fn (a, b) => a
  - fn (a, b) => b

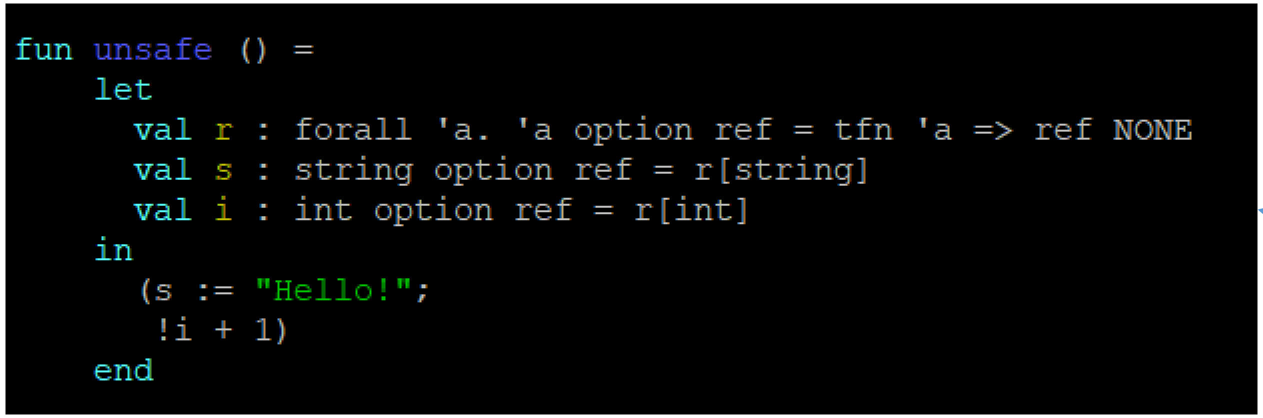
Typesafe refs



```
fun unsafe () =
    let
      val r : 'a option ref = ref NONE
      val s : string option ref = r
      val i : int option ref = r
    in
      (s := "Hello!";
       !i + 1)
    end
```

`unsafe` is not unsafe!

- Rewrite it as the compiler does when it compiles the code

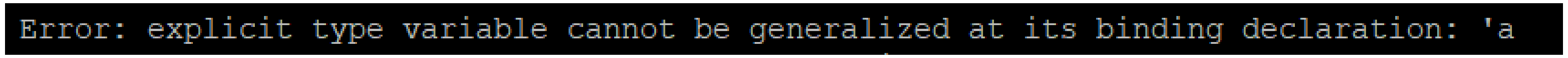


```
fun unsafe () =
    let
      val r : forall 'a. 'a option ref = tfn 'a => ref NONE
      val s : string option ref = r[string]
      val i : int option ref = r[int]
    in
      (s := "Hello!";
       !i + 1)
    end
```

Not valid SML syntax – only the compiler can write in tfn and [type]

- s and i are different refs, because they're generated by two different calls to r!

(This is very unintuitive, however, so SML institutes the *value restriction* to prevent confusion. Value Restriction: all polymorphic expressions must be values)



```
Error: explicit type variable cannot be generalized at its binding declaration: 'a
```