Functional Programming and ML [part 3]

In part based on slides from Gerardo Schneider, which where in turn based on John C. Mitchell's

Types and Type system (revisited)

Type

- Documentation
- Prevent errors
- Support optimization

Subtyping

- Substitutivity, aka the *Liskov substitution principle*
- No subtyping in ML

Type safety

- Progress and preservation
  - preservation is sometimes called *subject reduction*
- Soundness and Completeness
- Static versus dynamic/runtime checks

Polymorphism

Question. What does `poly` mean? And `morphous`?

What does `polymorphism` mean?

Three main flavors of polymorphism

1. Parametric polymorphism
2. Ad hoc polymorphism
3. Subtype polymorphism

1. Parametric polymorphism

- Single function may be given many types
- The type expression involves **type variables**

```
- map;
val it = fn : ('a -> 'b) -> 'a list -> 'b list
```

Question. Can you think of other (parametrically) polymorphic functions?

2. Ad hoc polymorphism

Also known as function overloading

- When a function has more than one definition
- Each definition having a different signature
  - different types for its arguments
- Overloading is resolved at compile time,
  - based on the function usage and context

```
- 3 + 1;
- 3.14 + 1.0;
```

3. Subtype polymorphism

- We write `S <: T` to express that `S` is a subtype of `T`
- If `S <: T`, then any expression of type `S` can be safely used in a context where a expression of type `T` is expected

```
function max (x as Number, y as Number) is
  ...
end
```

The example above is not ML syntax. ML does not have subtyping.

Type checking $\times$ Type inference

Type checking

- Check whether the programmer is mixing types in an unsafe way

Type inference

- Determines the type of an expression based on its sub-expressions
- Allows for type declarations to omitted

Type inference

- Type inference naturally leads to polymorphism
- Inference uses **type variables** and some of these might not be resolved

Question. What are the requirements on the argument passed to `f1`? How about `f2`?

```
int f1(int x) { return x+1; };

f2(x) { return x+1; };
```

Example

```
fun f(g,h) = g(h(0));
```

Different flavors of parametric polymorphism

System F

- a powerful parametrically polymorphic type system,
- however, type inference is not decidable [Wells'94]
- recently gaining popularity in practice because
  - limitations of HM have become apparent
  - extensions of System F address initial drawbacks

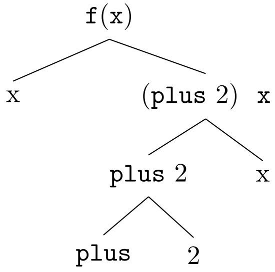
Hindley-Milner (HM) type system

- a restriction on System F
- type inference is decidable
- implemented in ML

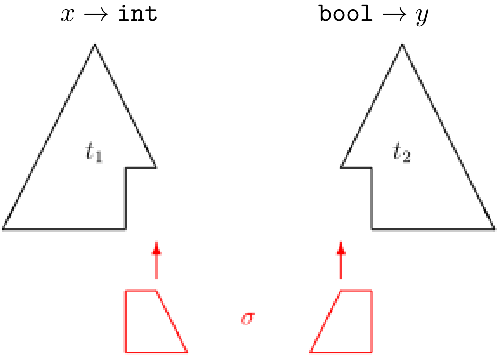
Type inference algorithm

1. Assign types to leaves of syntax tree
2. Generate constraints as we go up the tree
3. Solve constraints by unification

```
- fun f x = ((plus 2) x);
```



Unification



$\sigma(t_1) = x \mapsto \texttt{bool}$ $\sigma(t_2) = y \mapsto \texttt{int}$

Algorithm terminates and finds the **most general unifier** (if there exists one)

$$t_1 = x \to \texttt{int} \qquad\qquad t_2 = y \to z$$

Algorithm terminates and finds the **most general unifier** (if there exists one)

$$t_1 = x \to \texttt{int} \qquad t_2 = y \to z$$

$$\sigma(t_1) = x \mapsto y \qquad \sigma(t_2) = z \mapsto \texttt{int}$$

$$\sigma'(t_1) = y \mapsto x \qquad \sigma'(t_2) = z \mapsto \texttt{int}$$

The most general unifier is unique up to renaming: $\sigma \cong \sigma'$

Question.

- What happens when trying to unify t_1 and t_2 below?
- What situation can lead to this?
- What does it mean for a programmer?

$$t_1 = x \rightarrow \texttt{int} \qquad\qquad t_2 = y \rightarrow \texttt{bool}$$

Unification has applications besides type inference, for example in *logic programming*, as we will see with Prolog

Type inference, conclusion

- Eliminates or reduces the need for variable type declarations
- Finds the *most general type* by solving constraints via *unification*
- Leads to a flavor of parametric polymorphism

```
- fun id x = x;
val id = fn : 'a -> 'a
```

Question. How would you implement `id` in C++?

Type equality

- How to determine whether two types are equal
- Nominal × Structural type system

```
class Foo {
  method(input: string): number { ... }
}

class Bar {
  method(input: string): number { ... }
}
```

```
let foo: Foo = new Bar(); // Error OR Okay ?
```

https://medium.com/@thejameskyle/type-systems-structural-vs-nominal-typing-explained-56511dd969f4

Note to confuse:

equality on types × equality on expressions

Equality on types

```
let foo: Foo = new Bar(); // Error OR Okay ?
```

Equality on expressions

```
1 = 1;
1 = 2;
```

Types whose expressions can be checked for equality are called **equality types**.

In (S)ML we have:

Equality types	Depends	Not equality types
int	tuples	reals
bool	records	functions
char	data-types	abstract data types
string	lists	

Tuples, records, data-types, and lists are equality types if their *subparts* are equality types.

Question. Functions are generally not considered equality types. Why? What is difficult in comparing two functions?