1 Functional Programming and ML [part 3]

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

1.1 Types and Type system (revisited)

1.1.1 Type

- Documentation
- Prevent errors
- Support optimization

1.1.2 Subtyping

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

1.1.3 Type safety

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

1.2 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

We will look at each one of them next

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
```

If both `Int` and `Double` are subtypes of `Number`, then `max` can be applied to either subtype.

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

1.2.1 Parametric Polymorphism: ML vs. C++

- C++ function template
  - Declaration gives type of function arguments and result
  - Place declaration inside a template to define type variables
  - Function application: type checker does instantiation automatically
- ML polymorphic function
  - Declaration has no type information
  - Type inference algorithm
    - Produce type expression with variables
    - Substitute for variables as needed

Implementation

- C++

  - Templates are instantiated at program link time
  - Swap template may be stored in one file and the program(s) calling swap in another
  - Linker duplicates code for each type of use
- ML
  - Swap is compiled into one function (no need for different copies!)
  - Type-checker determines how function can be used

Comparison

- C++: more effort at link time and bigger code
- ML: runs more slowly, but gives smaller code and avoids linking problems
- Global link time errors can be more difficult to find out than local compile errors

1.3 Type checking × 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

1.4 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; };
```

`f1` takes an integer and returns an integer. On the other hand, it is safe to apply `f2` to any datatype that supports the `+` operator. In that sense, `f2` is more general.

Example

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

h must have the type: `int` $\rightarrow a$ because 0 is of type `int`

this implies that g must have the type: $a \to b$

Then f must have the type: $(a \to b) * (\texttt{int} \to a) \to b$

1.4.1 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

1.4.2 Hindley-Milner

1958 Type inference algorithm given by H.B. Curry and Robert Feys for the typed lambda calculus

1969 Roger Hindley extended the algorithm and proved that it gives the most general type

1978 Robin Milner -independently of Hindley- provided an equivalent algorithm (for ML)

1985 Luis Damas proved its completeness and extended it with polymorphism

1.4.3 Type inference algorithm

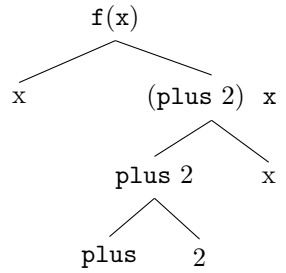
1. Assign types to leaves of syntax tree
2. Generate constraints as we go up the tree
3. Solve constraints by unification

Assuming that there exists a function `plus` declared as follows:

```
- fun plus x y = x + y;
```

Let's look at how type inference works on the function `f` below:

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



Starting at the leaves:

- Assign a type, say T_1 to `x`
- `plus` has two possible types: `int` → `int` → `int` and `real` → `real` → `real`
- 2 has type `int`

Propagate type information up the tree, resolving conflicts:

- From the two possible types of `plus` and the type of 2, we infer the type of `plus` 2 as `int` → `int` → `int`
- From the type of `plus` 2 and `x` we infer that `x` as `int`, in other words: $T_1 \mapsto$ `int`

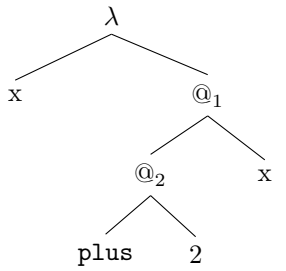
We get to the root, at which point we infer that `f` has type `int` → `int`

Note that the Mitchell book (*Concepts in programming languages*) draws the picture a bit different. They frame the problem in term of lambda-calculus. In lambda calculus, there is *function abstraction* and *function application*. Abstraction sets up a function, and application calls it with arguments.

Let `e` stand for the expression corresponding to the body of the function, meaning `((plus 2) x)`. In lambda-calculus, the declaration `fun f x = e` is written as $\lambda x.e$.

The expression `plus 2` means the application of the function `plus` to argument `2`. And the expression `(plus 2) 5` means the application of the function `(plus 2)` to the argument `5`.

In the book, the internal nodes of the "tree" are either abstractions (λ) nodes or application (@) nodes. To make it easier to refer to the different applications in the graph, I've labeled with a subscript as in $@_i$.



This makes application and abstraction obvious. In the case of application, consider the internal @ node which has `plus` and `2` as children. This represents the application of the function on the left-hand side (`plus`) to the argument `2` on the right-hand side. Every application node is of the same form: left-hand side is a function being applied to the argument on the right-hand side. Function abstraction, the λ nodes, have the parameter (variable) name on the left-hand size and the function body on the right-hand side. For example, the root of the previous tree has a λ node with `x` in the LHS and the function body in the RHS. To make it obvious that the `x` on the LHS is the same `x` as the one used in the function body, the book draws an edge between them. Thus, technically, the book does not draw a tree, but a more general graph instead.

The typing rule for @ node is: the LHS is a function, thus it has type $T_a \to T_b$. The RHS is the argument to the function, thus it must have type T_a.

The typing rule for λ nodes is: If the argument of the function (which is the LHS of the λ node) has type T_a and the body of the function has type T_b, then the λ node has type $T_a \to T_b$.

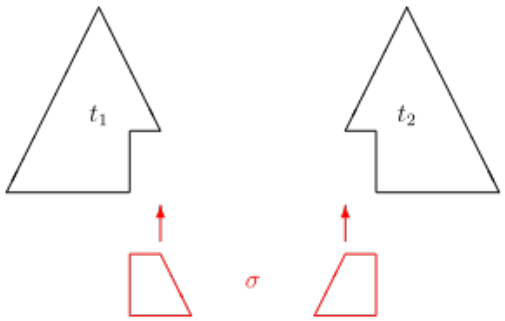
Starting from the leaves `plus` and `2`, we know `plus` has type `int -> int -> int` and `2` has type `int`. We then go up from these leaves and applying the typing rule for @ next. Since the LHS of an @ node must be a function, we know that the LHS must have type $T_a \to T_b$ where T_a must be the type of the RHS (the argument). In this case then T_a is `int` and T_b must be `int -> int`.

In the previous paragraph we concluded that `int -> int` is the type resulting from the application which has `plus` and `2` as leaves. Thus, the $@_2$ node has type `int -> int`. The parent to $@_2$ is also an application. The application $@_1$ applies a function of type `int -> int` (the type of $@_2$) to an argument `x`. At this point, we can infer that `x` must have type `int`, and that the type of the application $@_1$ is also `int`.

Finally, since we inferred that `x` has type `int` and $@_1$ has type `int`, now we can use lambda's typing rule to infer that the type of the λ node is `int -> int`.

1.4.3.1 Unification

$$x \to \texttt{int} \qquad\qquad \texttt{bool} \to y$$



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

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

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

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

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

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?

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

The terms t_1 and t_2 above cannot be unified. In other words, there does not exist a substitution σ such that $\sigma(t_1) = \sigma(t_2)$. This situation happens when a program is not properly typed. It means there is a typing error in the program. The compiler then generates an error message.

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

1.4.3.2 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++?

The purpose of the question is to show that parametric polymorphism can be a powerful part of a language. The identity function is so easily implemented in ML, `fun id x = x;`, but it is an issue for C. You could try enumerating the different types, and using function overloading like this:

```
char id(char x){ return x; }
int id(int x){ return x; }
float id(float x){ return x; }
// ...
```

The problem is, you cannot anticipate all types, specially given enumerated types and derived types: we can always create a type that is not already captured by the overloaded `id` function.

Another alternative is to threat `x` as a pointer, and return the address:

```
void* id(void* x){ return x; }
```

The problem here is that we lose type information by running something through `id`. We must convert something of type `T` to `void*` and we get back a `void*`; we lost the type information `T` associated with that something.

Again, parametric polymorphism is something powerful and not easily replicated without language support.

1.5 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 ?
```

In a nominal type system, the assignment of Bar object to foo generates an error because `Foo` and `Bar` are different names. In a structural type system, the assignment is okay because `Foo` and `Bar` have same method signatures (i.e. the same structure).

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

Note to confuse:

equality on types $\times$ equality on expressions

Equality on types

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

Question. Are `nat` and `n` equal? Nominally equal? Structurally equal?

The question of whether `nat` and `n` are equal depends on whether the language is nominally or structurally typed.

Equality on expressions

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

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

In other words, if e_1 and e_2 are expressions of an equality type, then we can check whether $e_1 = e_2$.

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?

Functions are sometimes considered *computational types*. It is easy to compare whether two functions are syntactically the same. Comparing functions semantically is much harder. In general, semantical comparison of functions impossible.