lab29.txt

```
+--------------------------+
| CS 4110 - 11/07/2025     |
| Lecture 28 - Polymorphism |
+--------------------------+
```

We've explored types that allow us to model many of the features found in high-level prog
ramming languages. But one important feature is missing: recursive types. Consider the fo
llowing OCaml type:

```
type tree =
  | Leaf of unit
  | Node of tree * int * tree
```

We can model the top-level constructors using a sum, writing inl and
inr instead of Leaf and Node, and we can model the data carried in a
Node as a product. This leads to an equation like:

```
 tree = unit + tree * int * tree
```

But how do we make this a *definition*? To achieve this, we can add
recursive types:

```
tau ::= ...
      | alpha
      | mu alpha. tau
```

The type mu alpha. tau can be transformed into its unfolding and
back. That is:

```
mu alpha. tau <-> tau [ mu alpha. tau / alpha]
```

In the literature, there are two main ways to handle the "<->"
above.

One approach is by treating the arrow as meaning a literal equality,
also known as "equi-recursive" types. Here we treat the "<->" as an
equality, and we add typing rules to allow the two forms to be used
interchangeably:

```
G |- e : tau[mu alpha. tau / alpha]
-----------------------------------
G |- e : mu alpha. tau


G |- e : tau[mu alpha. tau / alpha]
-----------------------------------
G |- e : mu alpha. tau
```

This approach is attractive, but it has some negative aspects. In
particular, the type system is no-longer "syntax-directed" which
complicates implementations and metatheory.

Thus, a second approach is to treat the arrow as meaning an
isomorphism, also known as "iso-recursive" types. Here we add a new
expression to the language

```
e ::= fold_{mu alpha. tau} e
    | unfold_{mu alpha. tau} e
```

We also add an operational semantics rule to handle these new forms.

```
---------------------
unfold (fold e) -> e
```

Now the typing rules become:

```
G |- e : tau[mu alpha. tau / alpha]
-----------------------------------------
G |- fold_{mu alpha. tau} e : mu alpha. tau
```

```
  G |- e : tau[mu alpha. tau / alpha]
  ------------------------------------------
  G |- unfold_{mu alpha. tau} e : mu alpha. tau
```

This approach is simpler in implementations and metatheory, but of course it makes programs more complicate. In practice, interpreters and compilers can insert the extra expressions automatically. For example, in OCaml, after using a constructor to build an element of a data type, we can fold it into a recursive type; and before pattern matching, we can unfold the recursive type so its structure can be examined.

To keep the notation light, we will often omit the type annotations on fold/unfold when they are clear from context.

As an example, let's see how we can encode the *untyped* lambda calculus, including non-terminating expressions, using recursive types.

Let U = mu alpha. alpha -> alpha. Note that the unfolding of the type, (alpha -> alpha)[mu alpha.alpha -> alpha / alpha] is (mu alpha.alpha -> alpha) -> (mu alpha.alpha -> alpha), or just U -> U.

Now we can define a translation [[.]] : Exp -> U

```
[[ x ]] = x
[[ \x. e]] = [[ fold (\x : U. [[e]]) ]]
[[ e1 e2 ]] = [[ (unfold e1) e2 ]]
```

Now let's turn to a different use of advanced types, namely polymorphism. This idea is based on a result of John C. Reynolds, as explained by Philip Wadler in his paper "Theorems for Free!"

Suppose I tell you that I might have System F terms of the following types?

* forall alpha. alpha -> alpha
* forall alpha. alpha -> alpha -> alpha
* forall alpha. alpha

For the first, it should be clear that essentially the term must behave like the identity function. Why? Well, it takes a value of type alpha, where alpha is a type parameter, and returns a value of type alpha. Besides returning its argument, how can it conjure an alpha? It can't! It can do silly things like passing the top-level argument to an inner identity function, but at the end of the day, it must return it.

By a similar argument, for the second, it should be clear that the function can either return its first or second argument -- i.e., it behaves like the Church booleans, either TRUE or FALSE.

Finally, the last type is uninhabited. There is no way to build an alpha out of nothing.

Note that this informal reasoning would be more complicated in a language with non-termination. But in System F, all programs terminate.

Now, how do we actually make this reasoning systematic. The key insight, due to Reynolds, is that we can define a kind of semantics for types based on a relational interpretation. And we can show that for polymorphic types, those relations are respected. The details are a bit technical (see Wadler's paper for an accessible presentation with lots of examples), but using this property, we can derive behavioral properties of programs just from their types, which is pretty amazing!

For instance, if we have list types, then if we have a function

```
  r : forall alpha. list alpha -> list alpha
```

and we have a function

```
  f : tau1 -> tau2
```

and we have the usual map function on lists:

```
  map : forall alpha, beta. (alpha -> beta) -> alpha list -> beta list
```

Then by relational reasoning we can show that

```
 (\l. r[tau2] (map f l)) = (\l. map f (r[tau 1] l))
```

Intuitively, r can modify the structure of the list, but it can't really do anything with its elements, besides "shuffling the deck chairs."

Amazingly, we can derive this theorem, and many others, and it works for any types! Check out Wadler's paper for the details!