14 - Haskell

Haskell is a programming language designed to be elegant, concise, and expressive. It was named after the logician Haskell Curry and was first released in 1990.

Characteristics:

- Purely functional: avoids side effects/mutable states;
- Lazy evaluation;
- Static typing;
- Type Inference;
- Higher-order functions;
- Monads: used to encapsulate effects in a purely functional way;
- Lazy garbage collection: only runs when memory is needed, rather than at fixed intervals.

The following will only cover the most important/theoretical topics. Please refer to the lectures for everything else.

Laziness

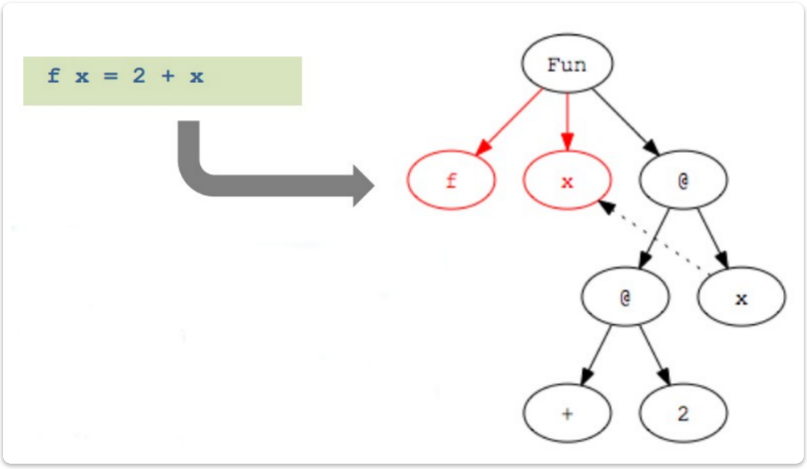
In Haskell, an expression yet to be evaluated is represented by a thunk. Thunks are created whenever an expression is bound to a variable or passed as an argument to a function, and they are evaluated automatically when their values are needed.

When a thunk is evaluated, its value is usually saved (or "memoized") so that it doesn't need to be re-evaluated again later. This can be useful for improving performance, because it can avoid the cost of recomputing the same value multiple times.

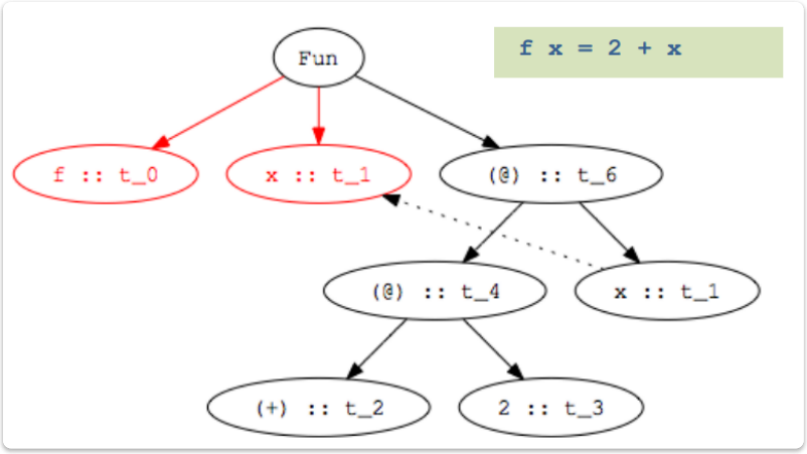
Type Inference

Steps:

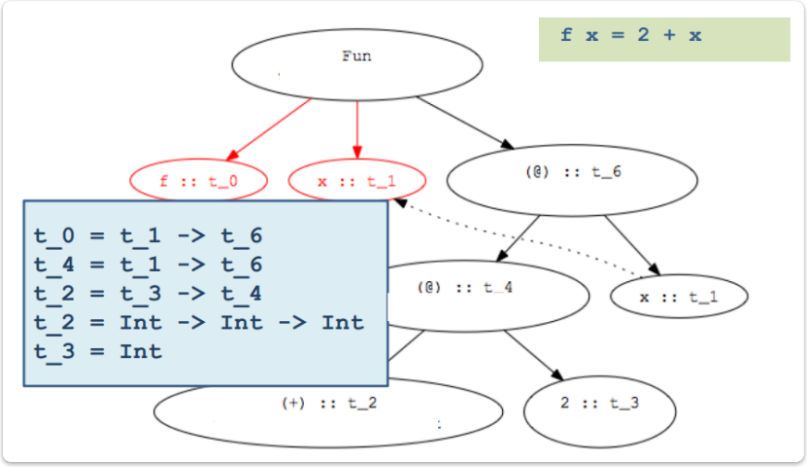
1. Parsing: the program is parsed to construct a parse tree (note that @ is for function application).



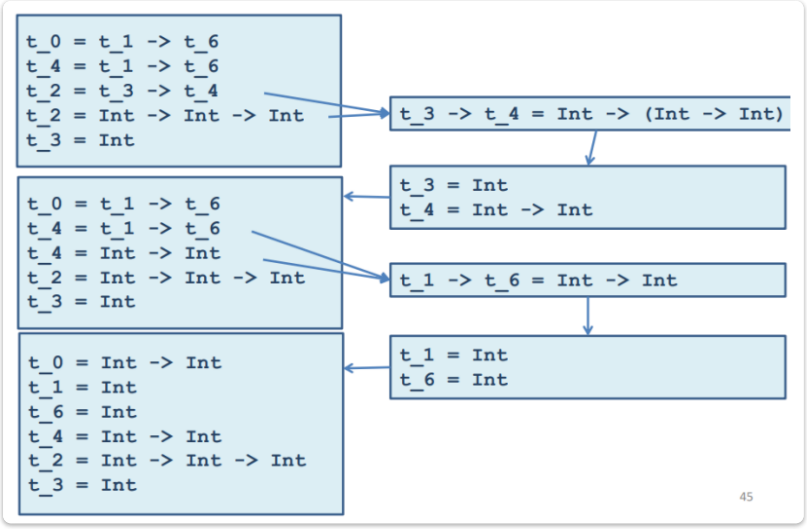
2. Assign type variables to nodes.



3. Generate constraints.



4. Solve constraints by executing the unification algorithm.



5. Determine type of top-level declaration. In the example, at the end we where interested in `t_0`, which is our result.

Note that type inference always produces the principal type (most general type), while there could be many less general types.

Also, type inference can be used for polymorphism. In that case, type specialization happens, which involves generating specialized versions of the function for each concrete type that it is used with.

Kind Inference

in Haskell, types themselves have types, which are called kinds.

For example, consider the `Maybe` type constructor, which has kind `* -> *`. This means that `Maybe` takes a type of kind `*` (i.e., a concrete type, such as `Int` or `String`) and returns a new type of kind `*` (i.e., another concrete type). The kind of `Maybe` is inferred by the compiler based on how it is used in the code.

Data types

Data types are used to define and represent the different kinds of data. Haskell provides a rich set of built-in data types, like Integers, Strings, Tuples...

Moreover, it is possible to define custom data types, allowing to represent more complex data structures.

For example:

```haskell
data Shape = Nil | Circle Float Float Float | Rectangle Float Float Float Float
```

Nil is a clause defining a possible value, while Circle and Rectangle are constructors taking respectively 3/4 Floats.

Type Classes

A Type Class allows to define a set of methods that can be performed on a given type. It is similar to an interface in object-oriented programming, but with a more powerful and flexible type system. They allow ad hoc polymorphism.

Any type can be made an instance of a type class by providing implementations for its methods. This allows functions that are polymorphic over a type class to be used with any type that is an instance of that class.

Beside user-defined Type Classes, there are built-in type classes, such as Eq, Ord, Show, Bounded...

Differently from interfaces, they can be defined for existing types, not just for types that are explicitly designed to implement them. This means that any type can be made an instance of a type class simply by providing implementations for its methods, without having to modify the type itself.

Design

There are 3 parts concerning the design of type classes:

1. Type Class declaration: define a set of methods with a name and one or more type variable.

```haskell
class Num a where
        (+) :: a -> a -> a -- NOTE: Haskell allows overloading of primitive operators
        (*) :: a -> a -> a
        negate :: a -> a
```

2. Type Class instance declaration: specify the implementations for a particular type.

```haskell
instance Num Int where
        a + b = intPlus a b
        a * b = intTimes a b
        negate a = intNeg a
```

3. Qualified types (or Type Constraints): concisely express the operations required on otherwise polymorphic type.

```haskell
square :: Num n => n -> n
square x = x*x
```

Compilation

Type classes are implemented by generating code at compile-time using a process called type class specialization.

When a type class is used in a function or expression, the compiler generates a dictionary or table of function implementations for that type class, based on the specific type or types that are used as arguments. This dictionary or

table is then used to determine which implementation of the type class's methods should be used at runtime.

Indeed, when we write:

```haskell
square :: Num n => n -> n
square x = x*x
```

It is compiled to:

```haskell
square :: Num n -> n -> n
square d x = (*) d x x
```

Subclasses

Type classes can be subclassed to create a hierarchy of related classes. To define a subclass, we use the `class` keyword followed by the name of the subclass and a list of one or more parent or superclass constraints.

For example:

```haskell
class Eq a => Ord a where -- Ord is a subclass of Eq
        -- ...
```

To create an instance of a subclass, we must provide implementations for all of its methods, as well as any methods inherited from its parent or superclass type classes.

Default Methods

Type classes can define "default methods", which can be optionally overridden by instances.

For example:

```haskell
class Eq a where
        (==) :: a -> a -> Bool
        x == y = not (x /= y)
        (/=) :: a -> a -> Bool
        x /= y = not (x == y)
```

Deriving

The `deriving` keyword is used to automatically generate instances of certain type classes for a given data type. Supported type classes are: Read, Show, Bounded, Enum, Eq, and Ord.

Example:

```haskell
data Color = Red | Green | Blue deriving (Show, Read, Eq, Ord)
```

Type Constructors

Type constructors generalize the concept of type classes.

A type constructor is a function that takes one or more types and returns a new type. Thus, the type variables passed must have kind `* -> *` (from type to type). For example, the list type constructor in Haskell, written as `[]`, takes a single type as input and returns a new type that represents a list of values of that type.

Unlike regular type classes, type constructors apply to a family of related types, rather than just a single concrete type.

Functors

Intuitively: they are things that can be mapped over.

A Functor is a type constructor that allows for mapping a function over a container type. It provides a powerful abstraction for working with container types in Haskell, as it allows to apply the same functions to different container types without having to write specialized code for each one.

Definition:

```language-haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

The `Functor` constructor class is part of the standard Prelude.

Monads

Intuitively: a "wrapper" around a value or computation that allow us to perform computations in a specific order, while also providing us with mechanisms for handling side effects, errors, and other issues that may arise during computation.

A Monad is a type constructor that allows to sequence computations that involve side effects while still maintaining referential transparency. It provides a way to abstract away the details of how those computations are sequenced, and allows for more concise and modular code.

Definition:

```language-haskell
class Monad m where
        return :: a -> m a                -- Return
        (>>=)  :: m a -> (a -> m b) -> m b -- Bind
```

- The return methods takes a value of type `a` and wraps it in a monad of type `m a`.
- The >>= (bind) method takes a monadic value of type `m a` and a function that takes a value of type `a` and returns a monadic value of type `m b`. It sequences the two computations, applying the function to the value inside the monad and returning the resulting monadic value of type `m b`.

Examples:

- `Maybe` monad: represents computations that may return a value or may fail with an error;
- `IO` monad: represents computations that perform I/O operations. When we use the IO monad to perform an operation, any side effects that are produced are encapsulated within the monad, and the result of the computation is returned as a value that we can then use in further computations;

Haskell also provides syntactic sugar to make monads even more readable: the do syntax.

```
bothGrandfathers p =
      father p >>=
          (\dad -> father dad >>=
              (\gf1 -> mother p >>=
                  (\mom -> father mom >>=
                      (\gf2 -> return (gf1,gf2) ))))
```

```
bothGrandfathers p = do {
       dad <- father p;
       gf1 <- father dad;
       mom <- mother p;
       gf2 <- father mom;
       return (gf1, gf2);
     }
```

```
bothGrandfathers p = do
       dad <- father p
       gf1 <- father dad
       mom <- mother p
       gf2 <- father mom
       return (gf1, gf2)
```

Some Higher-Order Functions

- `map`
- `filter`
- reduce (`foldl`, `foldr`)