This topic briefly covers one aspect of the implementation of programming languages: how type inference is performed.
Type inference

- Many modern functional languages like F# are statically typed, but support type inference.
- Other languages are increasingly including type inference (e.g. C#)
- The key technique in type inference is to use a type variable for the type of each sub-expression/sub-pattern in a definition.
  - Then equations between these variables are generated based on the types needed by the constructs in the program.
  - Solving these equations by unification then infers the type for the definition, and ensures that the types of the sub-expressions/sub-patterns are consistent with each other.
Types

A type is either

- a type variable, e.g. 'a, 'b, 'r3, etc., or
- a type constructor applied to the right number of arguments
  - also called a constructed type
  - e.g. int, 'a list, 'a -> 'a, Int*a are constructed types

A type is polymorphic if it contains any type variables, otherwise it is monomorphic

A type equation is an assignment of a type to a type variable – we will use these to describe our type inference algorithm

- e.g. ‘r4 = int
- e.g. ‘r2 = bool list
- e.g. ‘r5 = ‘r1 list
- e.g. ‘r3 = r2 * char list -> int -> ‘r4
- e.g. ‘r1 = ‘r13
Substitution and generality of types

We want to infer the most general type of each definition

- A type $T_1$ is an instantiation of a type $T_1$ if $T_1$ can be obtained by consistently substituting types for type variables in $T_1$
  - if $T_1$ is an instantiation of $T_2$ $T_2$ is more general than $T_1$ and $T_1$ is more specific than $T_2$
  - e.g. $\text{bool} \rightarrow \text{bool}$ is an instantiation of $a \rightarrow a$
  - e.g. $\text{char} \ast [\text{char}]$ is an instantiation of $c \ast d$
  - e.g. $\text{char} \ast d$ is an instantiation of $c \ast d$

- The most general unifier of two types $T_1$ and $T_2$ is the most general type $T$ that is an instantiation of both $T_1$ and $T_2$
  - e.g. the mgu of $c \ast a \ast \text{int}$ and $d \ast \text{char} \ast \text{char}$ is $c \ast \text{char} \ast \text{int}$
  - e.g. the mgu of $a \ast a$ and $\text{char} \ast \text{char}$ is $\text{char} \ast \text{char}$
  - e.g. the types $a \ast \text{int}$ and $\text{char} \ast \text{bool}$ have no unifier

The most general type $T$ of a function $f$ can be instantiated to any other type for $f$.

- the mgt of a function $f$ must be
  - general enough to allow every type that works for $f$
  - specific enough to exclude every type that doesn't work for $f$
Building the set of type equations

Assign a type variable to each expression and pattern in a program.

- This includes sub-expressions within larger expressions, and also sub-patterns.
- We can just assign the type variables sequentially: 'r1, 'r2, 'r3, etc.
- Each bound program variable v should consistently be assigned the same 'rv.

Then, for each expression/pattern construct a type equation which specifies the relationship between types that is required by that expression/pattern.

- Generally these equations relate the type of the whole expression/pattern to the types of any sub-expressions/sub-patterns immediately within it.
- e.g. for each function application \( e1 \ e2 \) we have the equation

  \[ 'r1 = 'r2 \rightarrow 'r3 \]

  where 'r1 is the type variable for the function: \( e1 \)
  'r2 is the type variable for the argument: \( e2 \)
  'r3 is the type variable for the application: \( e1 \ e2 \)

Then, solve these equations by unification (similar to the project).

- Report an error if unification fails, otherwise report the type obtained for the type variable for each top-level variable.
Polymorphism in type inference

Parametric polymorphism can be included in this algorithm roughly by:

- Solving the equations for each let declaration separately before continuing to the following parts of the program.
- When the inferred type $T$ for a let-bound variable includes type variables $'r_1, 'r_2, \ldots, 'r_N$, that don’t depend on other parts of the program, then its type is generalised to a type scheme:

  $$\forall ('r_1, 'r_2, \ldots, 'r_N) T$$

- E.g., the declaration:

  ```
  let id x = x
  ```

  Would be assigned the type scheme $\forall ('a) 'a$

  And the map function would be assigned the type scheme:

  $$\forall ('a,'b) ('a->'b) -> list<'a> -> list<'b>$$

- Then, each time the let-bound variable is used, a fresh set of type variables is used in place of the generalised variables.
- This allows variables to be used with many different types.
Value Restriction

In call-by-value languages like F#, it is necessary to restrict polymorphism to avoid making the type system unsound. Consider:

```fsharp
let r = ref []  // r : list<'a> ref
r:=[1]         // Seems fine - choose 'a=int
List.map not !r // Seems fine - choose 'a=bool  But!!!
```

• This program will try to apply "not" to the integer 1, which is exactly what the type system is supposed to prevent.

• The issue is that if we substituted `ref []` for `r` throughout it would change the meaning of the program.

• The usual solution (adopted in F#) is to only allow values to be polymorphic.
  ◦ Substituting values never changes the meaning of a program.

• Since functions are values, this restriction is not much of an issue in practice.
Overloading and subtyping

Overloading allows some variables (or methods) to refer to different things depending on the type.

Unfortunately this complicates type inference.

- Where there is overloading each of the possibilities must be tried.
- When many things are overloaded, there can be a lot of possibilities.
- Sometimes more than one possibility will succeed – causing ambiguity.

F# deals with these issues roughly by requiring that the types be determined sufficiently by the earlier parts of the program.

- This means sometimes it's necessary to use |> or type constraints.
- This is a pragmatic solution that works well enough in practice.
- It also handles similar issues with subtyping.

More power approaches are possible however.

- Haskell uses a sophisticated system of "type classes" that in the extreme is like compile-time logic programs that build programs.