Modularisation

Programming Language Design 2024

Hans Hüttel
based on original slides by Torben Ægidius Mogensen

4 March 2024

Part I

The underlying idea

Modularization can help structure programs

If we can structure our program as a collection of linked modules, we can

- Structure large programs into smaller pieces with clear interfaces.
- More easily replace of parts of a program
- Abstract away implementation details from the rest of program.
- Reuse code.

Modularization eases compilation and execution

. . . and once we have our program, we can

- Reduce compilation time by only recompiling changed modules.
- Share code among several programs at runtime.

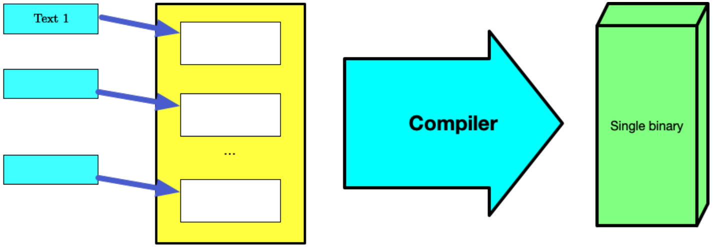
Learning goals

- To be able to understand and argue for the reasons for modularizing programs
- To be able to understand and explain the pros and cons of the main approaches to modularization at implementation level: text inclusion, linking and shared modules
- To be able to give an overview of the main ideas for supporting modularization at the language level, including the notions of signatures and headers

Part II

Modules at implementation level

An early approach: Text import



Early languages did not have modules, so reuse was by explicit copying of program text with no enforced indication of from where.

The copied code does not benefit from changes made to the original, or vice versa.

An early approach: Text import

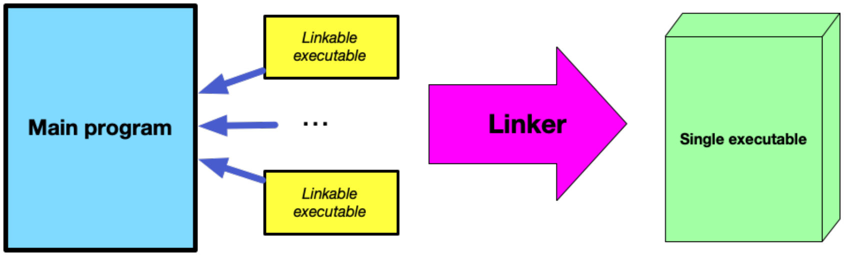
With the advent of permanent disk storage, code could be included by text import commands.

- Only one shared copy of text on disk.
- Changes to the text can be propagated to "users" by recompilation.

On the other hand the copied code gets recompiled every time the program using it is recompiled. If a program consists mostly of imported code, this can be significant.

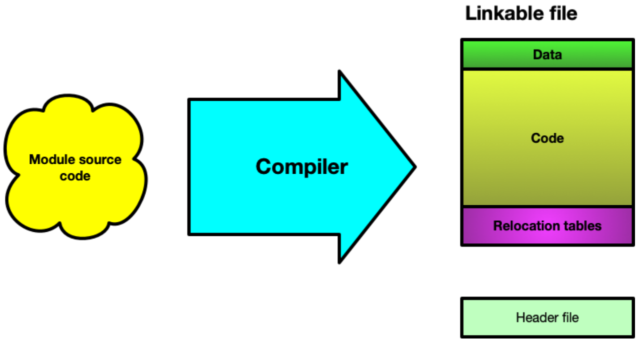
Separate compilation

Instead, let us compile modules separately and combine them to a single executable! We could link before execution.



Linking combines multiple linkable files to an executable.

Compiling a module



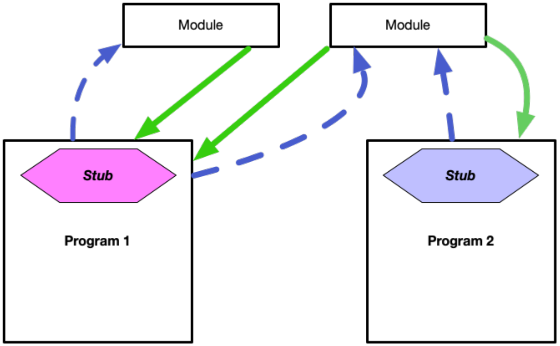
A module is compiled to a *linkable file* that contains code segments, data segments, and *relocation tables*.

A *header file* (or *signature*) contains information that allows a program that uses the module to be type-checked and compiled independently of the module.

Examples

- In C, a module file is `file.c`, a header file is `file.h`, and a linkable file is `file.o`.
- In F#, a module file is `file.fs`, a signature file is `file.fsi`, and a linkable file is `file.dll`

Shared modules



Instead of linking ahead of time to produce an executable, we could also link modules at load-time, allowing several programs to share the same module code in memory.

Shared modules

This saves memory and speeds up loading!

- Linkable file is loaded into memory, and local addresses are resolved.
- Programs using the module are compiled/linked to include a *stub* that at will runtime load the module (if it is not already loaded) and build a table of addresses of module entities, and this table is used by the program to access these module entities (by indirection).
- Module code has to be re-entrant: Multiple invocations of the module must be able to safely run concurrently .
- Reference counting is used to determine when memory for a module can be freed.

Part III

Language support for modularization

Modules with abstraction

Header files / signatures need not specify types for all entities in a module. This allows *abstraction*.

A signature relates to a module in the same way that a type relates to a value: It specifies the exposed capabilities of a module. Options are

Hand-written signature: Allows precise control over what information is made public, and can allow multiple different modules to implement the same signature.

Annotations in module: Explicit declaration of exported and private entities. Signature can be generated from module.

Abstract types: Signature specifies only the name of a type (and, possibly, some properties thereof) instead of full type specification.

Name spaces

To avoid name clashes, a module can have its own *name space*: A scope for the entities declared in the module.

- Names in a module are accessed by prefixing with the module name, e.g., `module.name` or `module::name`.
- A module can be *opened* to merge its name space with the current scope. Avoids the need for prefixes, but can give name clashes. Generally, names in opened module shadows names already in scope. Some module system allows renaming when opening a module.
- Nested modules / name spaces are possible. Names are accessed by using a path (e.g., `System.IO.File.Open`).

Modules versus classes

- A module is similar to a *static class*, but allows multiple types to be declared. A signature corresponds to an interface.
- Dynamic classes are different: A class is a type of an object, and an object is a value.
- Mixins in object-oriented languages are similar to module import: Mixins import fields and methods from one or more classes to a new class, but the new class does not inherit from these.

Modules as parameters / values

As mentioned, a signature can be seen as the type of a module. This can allow a program or module to be parameterised by *module parameters* constrained by signatures.

Some ML variants (Moscow ML, MixML) allow modules to be passed as values at run time. This blurs the concept of modules, making them more like objects (but without the inheritance).

Example of ML signatures

```
signature symbolTable =
     sig
       type (''key, 'value) table
       val empty :  (''key, 'value) table
       val insert :  (''key, 'value) table ->
          (''key * 'value)
                    -> (''key, 'value) table
       val lookup :  (''key, 'value) table -> ''
          key -> 'value option
     end
```

Example of ML structures

Different structures can implement the same signature. Here are two different ways of implementing a symbol table.

```
structure simpleTable :> symbolTable =
struct
    Implementation as linked list
    end

structure hashTable :> symbolTable =
struct
    Implementation as hash table
    end
```

Example of ML functors

A functor in ML is a parameterized structure that takes one or more structures as arguments.

All application happens at compile time.

```
functor compiler (table :  symbolTable) =
struct
...
end

structure comp1 = compiler(simpleTable)
structure comp2 = compiler(hashTable)
```