CSE 341: Programming Languages

Autumn 2005

Lecture 12 — Modules and Abstract Types

Modules

Large programs benefit from more structure than a list of bindings.

Breaking into parts allows separate reasoning:

- Application-level: in terms of module (in ML, structure) invariants
- Type-checking level: in terms of module types
- Implementation level: in terms of module code-generation

By providing a *restricted* interface (in ML, a signature), there are *more* equivalent implementations in terms of the interface.

Key restrictions:

- Make bindings inaccessible
- Make types abstract (know type exists, but not its definition)

SML has a much fancier module system, but we'll stick with the basics.

Abstract types are a "top-5" feature of modern languages.

Structure basics

Syntax: `structure Name = struct` *bindings* `end`

If `x` is a variable, exception, type, constructor, etc. defined in `Name`, the rest of the program refers to it via `Name.x`

(You can also do `open Name`, which is often bad style, but convenient when testing.)

So far, this is just *namespace management*, which is important for large programs, but not very interesting.

Signature basics

(For those interested in learning more, we're doing only *opaque signatures* on structure definitions.)

A signature `signature SQUID = sig ... end` is like a type for a structure.

- Describes what types a structure provides.
- Describes what values a structure provides (and their types).

Writing `structure Name :> SQUID = struct` *bindings* `end`:

- Ensures `Name` is a legal implementation of `SQUID`.
- Ensures code outside of `Name` assumes nothing more than what `SQUID` provides.

Hence signatures are what really enable separate reasoning.

Signature matching

Is `Name` a legal implementation of `SQUID`.

- Clearly it must define everything in `SQUID`.
- It can define more (unavailable outside of `Name`).
- `SQUID` can restrict the type of polymorphic functions.
- `SQUID` can make types abstract.

In particular, making a datatype abstract hides the constructors, so clients have no (direct) way to create or access-parts-of values of the type.

That's often a good thing.

Remember

Key tools for modularity/information hiding in ML: structures and signatures (and functors, which we're skipping).

A signature that "hides more" makes it easier to:

- Replace the structure implementation without breaking clients.
- Reason about how clients use the structure.

Note: See the extended example code for this lecture for more details...