CSE 341: Programming Languages

Dan Grossman

Spring 2004

Lecture 11— Modules and Abstract Types

Finishing up equivalence

End of last class: Decided `fn x => e x` can be replaced by `e` if `e` terminates, is effect-free, and has no free occurrence of `x`.

Example:

```
fn y => (if b then fn z => z+1 else fn z => z+2) y
```

can be replaced by

```
(if b then fn z => z+1 else fn z => z+2)
```

Non-example:

```
fn y => (if b then raise E else fn z => z+2) y
```

cannot be replaced by

```
(if b then raise E else fn z => z+2)
```

When `b` is bound to true, the former evaluates to a function that raises an exception when called and the latter raises an exception.

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 BLAH = sig ... end` is like a type for a structure.

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

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

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

Hence signatures are what really enable separate reasoning.

Signature matching

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

- Clearly it must define everything in `BLAH`.
- It can define more (unavailable outside of `Name`.
- `BLAH` can restrict the type of polymorphic functions.
- `BLAH` 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

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

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

Note: The real "content" of this lecture is in the extended example.