Modules for Standard ML

I Basic Principles and Terminology

In its simplest form, a *module* is just a collection of declarations, or in fact a single compound declaration, that is thought of as a meaningful program unit and given an identity, allowing it to be named.

The evaluation of a declaration produces an environment. However, a declaration will, in general, contain free identifiers, which we will call its *prerequisites*, and its evaluation requires an environment which binds these prerequisites. Thus the meaning of a declaration *dec* is a function ⟦dec⟧ : Env → Env.

An environment E supplying context for the evaluation of a declaration *dec* must satisfy certain constraints. First, it must bind all the prerequisites of *dec* with the appropriate kinds of bindings (i.e. identifiers used as types in *dec* must be bound to types, etc.), and second, the types assigned to value, constructor, and exception ids by E must permit a correct typing of *dec*. (It must be self-consistent as well; e.g. the proper arities must be assigned to type constructor names, and type expressions must be well-formed.)

Modules for SML

If a declaration (i.e. module) is to be sufficiently self-contained to permit it to be separately compiled (i.e. compiled outside the context of any particular program), then it must be augmented with a full specification of its prerequisites and their required types. This specification takes the form of an environment signature and will be called the inward interface of the declaration. The specification directives of [SML, §4.3] are just such an inward interface. {Note: Poly spec}.

The meaning of such an augmented declaration, ⟨I-spec, dec⟩ is a function

$$[\![\langle \text{I-spec}, \text{dec} \rangle]\!] : \text{Env}_{\text{I-spec}} \to \text{Env}$$

where $\text{Env}_{\text{I-spec}}$ is the "type" of those environments that satisfy the inward specification I-spec.

The result of this evaluation is an environment that satisfies an outward interface, which will be defined as the environment signature obtained by type-checking the declaration relative to the inward specification. Thus a more precise typing for the declaration would be

$$[\![\langle \text{I-spec}, \text{dec} \rangle]\!] : \text{Env}_{\text{I-spec}} \to \text{Env}_{\text{O-spec}}$$

Modules for STL

where O-spec is the outward interface specification, derived by type-checking dec relative to I-spec. The O-spec specifies the identifiers bound by dec.

The result of evaluating a declaration dec in an environment E_1 is a new environment $[\![dec]\!](E_1)$ which is typically used to enrich or augment another environment E_2 by concatenation, yielding the environment

$$E_2 ; [\![dec]\!](E_1).$$

Note that there is no necessary connection between the environment E_1 used to evaluate dec and E_2, the environment augmented by the resultant environment. {Note: Spec Proposal}

To summarize, a minimal notion of module consists of a declaration together with an environment signature specifying its prerequisites (the inward interface). These together determine the module's outward interface, which is an environment signature specifying the bindings created by the declaration. {Note: Spec directives} A module is actually an environment generator, i.e. a function which, when applied to an

Modules for SML

appropriate prerequisite environment, produces an environment satisfying the outward interface. This resultant environment will be called an instance of the module. The distinction between a module as an environment generator and the environments or instances it generates is important to keep in mind, since it is common to suppress this distinction. {Note: Generator vs Instance}.

We can make a module more self-contained by specifying exactly where its prerequisite bindings come from. A natural way to do this is to indicate that they are bound in some given module instances. If $C = \langle I\text{-spec}, dec \rangle$ and A and B are module instances, then we can create an instance of C by

$$[\![C]\!](A;B)$$

assuming that the environment A;B satisfies the inward interface I-spec. This will be the case if $O\text{-spec}_A ; O\text{-spec}_B$ matches (or includes) I-spec.

For the purpose of separate compilation, we can replace the inward interface with a concatenation of

Modules for SML

outward interfaces of modules that includes, or "covers" the inward interface (i.e. which specifies all the prerequisite identifiers appropriately). For compilation, we do not need the module instances A and B, but just their outward interfaces. Any other instances satisfying these outward interfaces would do to create an instance of C. Thus $O\text{-}spec_A$ and $O\text{-}spec_B$ are a kind of parameter "type" specification for C, which could be reformulated as the following function

$$C = \lambda A : O\text{-}spec_A, B : O\text{-}spec_B \,.\, [\![dec\, B(A;B)$$

{Note: Interfaces}.

Modules for SML

Dependence and Inheritance

If we define a module instance

$$C = C(A,B)$$

by evaluating the declaration of C in the environment A;B, then clearly C can be said to depend on A and B, because bindings from A and B are used to help define the types, values, and exceptions bound in C (by giving meanings to the prerequisite identifiers of C).

There are two forms of dependence: visible and invisible. A binding $x = v : \sigma$ in C depends <u>visibly</u> on type bindings in A or B whose names occur in σ. A hidden dependency occurs when the declaration binding x refers to prerequisites bound in the context A;B, but the type of x does not involve type constructors from A or B. C is visibly dependent on its context A;B if any of its bindings are, i.e. if the outward interface $O\text{-spec}_C$ involves types from $O\text{-spec}_A$; $O\text{-spec}_B$. Otherwise the dependence is invisible. Note that dependence on values and exceptions can

Modules for SML

only be invisible, though value dependencies often give rise to visible type dependencies. Also visible dependencies are statically detectible.

Dependence relationships are most naturally defined at the level of bindings, but the terminology can be extended to relate modules (or module signatures).

If C visibly depends on a parameter module A (meaning $spec_C$ mentions types from $spec_A$), then C (and its instances) are not truly self-contained. The types of certain names bound in C will contain free or uninterpreted type constructor names which should be bound by the instance A on which C depends.

The completeness of C can be restored by causing instances of C to inherit bindings from the parameter A. A straightforward way to achieve this is to redefine C as

$$C = \lambda A : spec_A . \; A; [\![dec]\!](A)$$

with $spec_C = spec_A ; spec(dec_C, spec_A)$. Of course, this causes $spec_C$ to contain all of binding specs

in $spec_a$ which are not masked by rebindings in $SPEC(dec_c, spec_a)$, and not just the minimal set of binding signatures needed to close $SPEC(dec_c, spec_a)$. However, from a pragmatic point of view this seems justified because an environment which binds a type without supplying any operations relating to that type leaves the type binding fairly useless (unless operations relating to that particular type can be supplied by an outside source).