Types for Module Systems

Jonathan Schuster

In this talk, I will be discussing type systems for ML modules. Specifically, we'll look at some of the complications that arise when trying to extend the typical features of the ML module system to give it more practical value, and then we'll talk about one specific formulation of a type system for ML modules encoded in System F_ω. The information in this talk primarily comes from Derek Dreyer's PhD thesis [1] and Russo et al.'s paper, *F-ing Modules* [3].

1 ML Module System - The Basics

First, let's review the general ideas behind ML modules. The syntax I use here is slightly different from what is seen in something like Standard ML, but it's only for the purpose of presentation—the concepts remain the same.

Let's take a look at what a module for sets of integers might look like:

```
IntSet = {
  type set = int list;
  val empty = nil;
  val insert = fn (x : int, s : set) : set => ...
  val isMember = fn (x : int, s : set) : bool => ...
}
```

In this example, we have a module `IntSet` that has one type declaration and three value declarations. Value declarations work similarly to fields in records: you bind them to a variable, and you can project them from the record with the usual dot syntax (e.g. `IntSet.empty`). The type declaration `type set` essentially just renames the type `int list`—with this module, whenever you see `set`, you can just mentally replace it with `int list`. You can also declare submodules within a module with something like `module X = { ... }`.

This module is fine as far as it goes, but right now, every consumer of this module knows that sets are represented as lists and can exploit that fact. We'd like to hide our representation type from the consumer, and this is where signatures and sealing come in.

Think of a signature as the type of a module—it tells the consumer what it can do with a module. For our `IntSet` module, the signature might look something like this:

```
INT_SET_SIG = {
  type set;
  val empty : set;
  val insert : (int * set) -> set
  val isMember : (int * set) -> bool
}
```

Notice the type declaration in the signature: the signature doesn't say anything about it other than that it's a type. We've effectively specified an abstract type, and we call this an *opaque* type declaration. If we had specified the actual type, with syntax like `type X = some_type`, we would call it a *transparent* type declaration. Note that just as values can have a principal type, a module can have a *principal signature* that gives the most specific information about that module.

Finally, to hide the type information, we have to apply the signature through an operation called *sealing*:

```
SealedIntSet = IntSet :> INT_SET_SIG
```

Now consumers of `SealedIntSet` cannot exploit its internal representation type.

Great, we have a modular implementation of sets with information hiding. But what if we want to generalize this to sets of any type of element? This is where functors come in. A functor is essentially a function at the module level: it takes one or more modules as input and returns a module as output.

To use this on our set example, though, we need one more thing, which is the idea of translucent signatures. If after defining a signature, you want to give a little more information about one of its types, you can do so with syntax like the following:

```
SOME_SIG where type t = some_type
```

Now, let's generalize our `IntSet` module as a functor:

```
ITEM_SIG = {
  type item;
  val equals : item * item -> bool;
}

SetFunctor = fun (Item : ITEM_SIG) => {
  type set = Item.item list;
  type item = Item.item;
  val empty = nil;
  val insert = fn (x : item, s : set) : set => ...
  val isMember = fn (x : item, s : set) : bool => ...
} :> SET where type item = Item.item
```

We now have a set functor that can be used for any kind of set. For example, if we wanted a module for a set of integers and another for a set of strings and if we had the appropriate `Integer` and `String` modules that were sealed with `ITEM_SIG`, we could do the following:

```
IntSet = SetFunctor(Integer)
StringSet = SetFunctor(String)
```

2 Complications

2.1 Higher Order Functors

Now that we've seen the basics of the ML module system, let's see what happens when we try to extend it with features programmers might want. First, what happens if we make functors higher order? That is, what if functors can consume or produce other functors as input/output, and not just modules? Consider the following:

```
SIG = { type t; }

Apply = fun (F : SIG -> SIG, X : SIG) => F(X)
Ident = fun (X : SIG) => X

Arg = { ... type t = ... } :> SIG
M1 = Ident(Arg)
M2 = Apply(Ident, Arg)
```

Looking at this code, you can see that `Apply` just applies its first argument (a functor) to its second, and that `Ident` is simply the identity functor for modules. `SIG` is just a signature that declares a single abstract type.

Now, consider `M1` and `M2`. One would think they would behave similarly, and they mostly do, except for one point: The typechecker recognizes `M1.t` to be the same as `Arg.t`, but cannot see the same equivalence for `M2.t`. Why is this?

The reason is that the principal signature of `Ident` is `(X : SIG) -> SIG where type t = X.t`, so the typechecker can link the type of `Ident`'s argument and its result. However, with our current language, we can't express a signature for `Apply` that gives us enough information to do something like this.

One solution, proposed by MacQueen and Tofte, is to re-typecheck `Apply`'s body whenver it is used and gain extra type information that way. This will allow programs like the one above to pass, but it has two main issues:

1. This solution only works when the compiler has access to the body of `Apply`. For instance, this won't work if `Apply` was compiled separately.
2. The signature we give the module no longer expresses the type we want it to have—in a sense, the typechecker has more information than is encoded in the signature. We'd rather have some more expressive signature language that can express the equivalence relation we want.

A better solution is the idea of an *applicative* semantics for functors. So far, the functors we have been dealing with are *generative*, in that they generate new abstract types every time they are called. With applicative functors, the idea is that applying the same functor to the same arguments should always give you a module with the same abstract types, no matter how many times you apply it. Along with this, the syntax is extended to allow for projection of types from functor applications as well. So, in an applicative setting, `Apply` would have the following principal signature:

```
(F : SIG -> SIG, X : SIG) -> SIG where type t = F(X).t
```

The above signature allows us to express the typing relationship we want and fixes the problem with `Apply`.

That's not to say that applicative functors are always the right answer, though. For example, you might want to create a module for a lookup table that generates a new key for each value you insert. Generative functors would allow you to ensure that a key can only be used with the module that generated it.

2.2 Recursive Modules

Another feature we'd like to have in our module system is mutually recursive modules. For example, we'd like to write something like this:

```
A = {
  val m = ref 0;
  val f = fn (x : int) => ... B.g(y) ...
}

B = {
  val n = ref 1;
  val g = fn (z : int) => ... A.f(w) ...
}
```

Unfortunately, ML doesn't allow this: a module is only in scope after its own declaration. There are a few workarounds to get around this, however. One could just move the needed declarations into the same module, but then we lose the modular structure we were trying to create. We could also create a functor that's parameterized over the defintion of one of the modules and use it as a sort of forward reference, but that gets a bit ugly when more modules come into play. A third option is to use something like Scheme's backpatching system, setting one of the modules to a memory reference that only gets its value after the module is created.

However, none of these workarounds are satisfactory: they don't scale well as the number of mutually recursive modules grows, and some structures can only be created with truly recursive modules. In the end, we really want some sort of form for recursive modules, like this:

```
rec(X:S.M)
```

The idea is that this form defines a module `M`, where `X` is bound in its scope to the value of the module itself. The module must match the signature `S`.

Using this structure, we could define our example above as a module with two mutually recursive submodules:

```
M = rec(X:SIG.{
  A = {
    val m = ref 0;
    val f = fn (x : int) => ... X.B.g(y) ...
  }

  B = {
    val n = ref 1;
    val g = fn (z : int) => ... X.A.f(w) ...
  }
})
```

That's nice, but we haven't defined the actual semantics of our `rec` form. Let's just do the naive thing and say that it's equivalent to its unfolding, `M[rec(X:S.M)/X]`. Note that this means the module `M` will get re-evaluated every time it references `X`.

That works fine in a purely functional system, but what if our modules also have values that can have side effects (such as `m` and `n` above)? In this case, every time we recursively reference the module, we effectively reset any state associated with the original version. Instead of re-evaluating the module, what we really want to do is use the *same* module for every unfolding. We can accomplish this by using the Scheme-like backpatching trick mentioned above as a workaround, but we instead do it internally so that it's invisible to the programmer.

This leaves just one question: How do we ensure that `X` is not referenced before it's bound to the "value" of the module? This can be done either statically or dynamically, but in the interest of time, I won't go into those methods here.

3 Encoding ML Modules in System F_ω (F-ing Modules)

3.1 Motivation

In 1986, David MacQueen wrote a paper arguing that existential types were too cumbersome to express a type system for every day modular programming (such as in ML) [2]. In MacQueen's opinion, restricting operations on existential types to the scope of an unpack form was too limiting and inevitably led to expanding the unpack's scope to include large portions of the program. As an alternative, he proposed using *strong existentials*, which adds the type $e.\alpha$ and a type rule like the following:

$$\frac{\Delta;\Gamma \vdash e : \exists\alpha.\sigma \qquad \Delta,\alpha;\Gamma,x:\sigma \vdash e' : \tau' \qquad \Delta \vdash \tau' \qquad \alpha \notin \Delta \qquad \mathit{pure}(e)}{\Delta;\Gamma \vdash \mathsf{unpack}\langle\alpha,x\rangle = e \ \mathsf{in}\ e' : \tau'[e.\alpha/\alpha]}$$

Note that this essentially allows existentially bound types to escape their unpack by being bound to their original pack expression.

However, allowing terms into the type language created a dependent type system, which of course has undecidable type-checking in general. Over the next two decades, researchers created more and more refined type systems to resolve this issue, but none were satisfactory, and they gave the ML module system the reputation of being overly complex. What was needed was a simple to understand static and dynamic semantics for the ML module system.

Finally, in 2010, Rossberg et al. published F-ing Modules [3], contradicting MacQueen's opinion of existential types and arguing that one could indeed simply express the semantics of ML modules using nothing but the type system of System F_ω. This led to the paper's slogan, "ML modules are just a particular mode of use of System F_ω." The rest of this talk describes their solution.

3.2 A Description of Elaboration

The key insight of the F-ing Modules approach was that, instead of giving a direct semantics for ML, one should define an *elaboration semantics* that defines the language in terms of some other target language (in this case, System F_ω). Thus, ML terms elaborate to F_ω terms, ML types elaborate to F_ω types, etc.

The main question is how to elaborate ML modules and signatures, especially their abstract types. To address this, the paper presents the idea of *semantic signatures*. A semantic signature is an F_ω type that represents the *semantic* interpretation of a *syntactic* signature in ML. Specifically, a semantic signature is an existential type that binds all of the abstract types in the ML signature. We represent a semantic signature as Ξ, which has the form $\exists\overline{\alpha}.\Sigma$. We call this an *abstract signature*, since it binds abstract types. Σ, on the other hand, is a *concrete signature*: an F_ω record type that gives the types for the various declarations within the module/signature.

Let's look at a concrete example. Recall the signature of our original Set module, `INT_SET_SIG`:

```
INT_SET_SIG = {
  type set : *;
  val empty : set;
  val insert : (int * set) -> set;
  val isMember : (int * set) -> bool;
}
```

I've added the kind annotation to the type here, since we'll now be dealing with higher-kinded types. By elaborating this syntactic signature into a semantic signature, we get the following:

$$
\begin{aligned}
\exists\alpha.\{&l_{set} : \{\mathsf{typ} : \forall\beta : (\Omega \to \Omega).\beta\alpha \to \beta\alpha\},\\
&l_{empty} : \{\mathsf{val} : \alpha\},\\
&l_{insert} : \{\mathsf{val} : \mathit{Int} \times \alpha \to \alpha\},\\
&l_{isMember} : \{\mathsf{val} : \mathit{Int} \times \alpha \to \mathit{Bool}\}\\
&\}
\end{aligned}
$$

This looks complicated, but let's break it down. First, notice that the type declaration has been turned into the existentially bound type α, and its scope is the entire signature. Any abstract types defined in a module or a submodule are always hoisted to the top-level of the signature.

Ignoring the type declaration l_{set} for a moment, we have three other labels in our concrete signature: l_{empty}, l_{insert}, and $l_{isMember}$. We assume that we can inject any module field name `X` into a record label l_X, and that these labels will not conflict with other labels used in the target language. Every declared field in a module will become one of the fields in an F_ω record type, but since we need to distinguish between value and type declarations, we wrap each one in a single-field record labeled as either "typ" or "val". Within that record type, you can see that each of the functions has the expected type.

Finally, examine the field for the type declaration, l_{set}. If you look closely, you can see that this is essentially an identity function. The reason for this is that we don't really care what the value of this declaration is - we only need it for typechecking, and then we can ignore the actual value at runtime.

Elaborating the actual module `SET` gives the expected corresponding value of the above type:

$$
\begin{aligned}
\mathsf{pack}\langle \mathit{IntList}, \{&l_{set} = \{\mathsf{typ} = \Lambda\alpha : (\Omega \to \Omega).\lambda x : \alpha\ \mathit{IntList}.x\},\\
&l_{empty} = \{\mathsf{val} = (\ldots)\},\\
&l_{insert} = \{\mathsf{val} = (\ldots)\},\\
&l_{isMemmber} = \{\mathsf{val} = (\ldots)\}\}\rangle
\end{aligned}
$$

I'm not showing the translations of the functions here since this presentation assumes some elaboration for the terms of the core language (lambdas, applications, etc.).

Writing out the records that wrap values as typ and value gets messy, so we introduce the following shorthand:

(value declaration type)	$[\tau]$	$:=$	$\{\mathsf{val} : \tau\}$
(type declaration type)	$[= \tau : \kappa]$	$:=$	$\{\mathsf{typ} : \forall \alpha : (\kappa \to \Omega).\alpha\tau \to \alpha\tau\}$
(value binding)	$[e]$	$:=$	$\{\mathsf{val} = e\}$
(type binding)	$[\tau : \kappa]$	$:=$	$\{\mathsf{typ} = \Lambda\alpha : (\kappa \to \Omega).\lambda x : \alpha\tau.x\}$

Using this shorthand, our examples become the following:

$$\begin{aligned}\exists \alpha.\{&l_{set} : [= \alpha : \Omega]\},\\ &l_{empty} : [Int],\\ &l_{insert} : [Int \times \alpha \to \alpha],\\ &l_{isMember} : [Int \times \alpha \to Bool]\}\end{aligned}$$

$$\begin{aligned}\mathsf{pack}\langle IntList, \{&l_t = [IntList : \Omega],\\ &l_{empty} = [(\ldots)],\\ &l_{insert} = [(\ldots)],\\ &l_{isMemmber} = [(\ldots)]\}\rangle\end{aligned}$$

Projecting a value out of the module does the appropriate packs and unpacks. For example, the expression `IntSet.empty` would elaborate to this:

$$(\mathsf{unpack}\langle\overline{\alpha}, y\rangle = (\mathsf{unpack}\langle\overline{\alpha}, x\rangle = IntSet \text{ in } \mathsf{pack}\ \langle\overline{\alpha}, x.l_{IntSet}\rangle) \text{ in } y).val$$

3.3 A Peek at the Elaboration Rules

Now that you have an intuition for how the system works, let's look at the actual formalisms. The elaboration is done through a series of syntax-directed elaboration judgments, the majority of which are translation judgments that specify how to convert a term from ML to its corresponding syntax in F$_\omega$. For example, let's look at the rule for value declarations in a signature:

$$\frac{\Gamma \vdash T : \Omega \rightsquigarrow \tau}{\Gamma \vdash \mathsf{val}\ X : T \rightsquigarrow \{l_X : [\tau]\}}$$

The conclusion of this rule can be read as "Gamma proves the value declaration elaborates to the F$_\omega$ declaration $\{l_X : [\tau]\}$". The premise checks to make sure that the given type is a proper type, and it produces its corresponding F$_\omega$ type τ, which is used in the conclusion.

Similarly, let's look at the rule for a value binding in a declaration:

$$\frac{\Gamma \vdash E : \tau \rightsquigarrow e}{\Gamma \vdash \mathsf{val}\ X = E : \{l_X : [\tau]\} \rightsquigarrow \{l_X = [e]\}}$$

The conclusion of this rule is similar to the earlier rule, except that along with producing the F_ω type, we also produce the F_ω value, e. The premise says that to translate this value binding, we have to have that the ML expression `E` translates to the F_ω expression e with type τ.

Next, let's look at the rule for sealing a module:

$$\frac{\Gamma(X) = \Sigma \qquad \Gamma \vdash S \rightsquigarrow \Xi \qquad \Gamma \vdash \Sigma \leq \Xi \uparrow \overline{\tau} \rightsquigarrow f}{\Gamma \vdash X :> S : \Xi \rightsquigarrow \mathsf{pack}\langle \overline{\tau}, f X\rangle}$$

There are two important things to note here. The first is that, since we're sealing a module, we introduce a pack form to seal away our abstract types. Second, we've introduced a new type of judgment in our last premise. This judgment is the signature matching judgment. It says that the concrete signature Σ matches the abstract signature Ξ if you replace all instances of α in Ξ with τ, and it returns a function f that converts an F_ω module expression from the concrete type to the abstract type.

The module projection rule is what takes care of most of the automatic packing and unpacking:

$$\frac{\Gamma \vdash M : \exists \overline{\alpha}.\{l_X : \Sigma, \overline{l_{X'} : \Sigma'}\} \rightsquigarrow e}{\Gamma \vdash M.X : \exists \overline{\alpha}.\Sigma \rightsquigarrow \mathsf{unpack}\langle \overline{\alpha}, y\rangle = e \;\mathsf{in}\; \mathsf{pack}\langle \overline{\alpha}, y.l_X\rangle}$$

As you can see, this rule unpacks the given module expression, projects out the desired field, and repacks it with the same types. A later rule converts this new pack expression into a normal expression by checking that none of the abstract types are needed in the enclosed value.

Finally, the two other interesting rules are the abstraction and application rules for functors:

$$\frac{\Gamma \vdash S \rightsquigarrow \exists \overline{\alpha}.\Sigma \qquad \Gamma, \overline{\alpha}, X : \Sigma \vdash M : \Xi \rightsquigarrow e}{\Gamma \vdash \mathsf{fun}(X : S) \Rightarrow M : \forall \alpha.\Sigma \rightarrow \Xi \rightsquigarrow \Lambda \overline{\alpha}.\lambda X : \Sigma.e}$$

$$\frac{\Gamma(X_1) = \forall \overline{\alpha}.\Sigma' \rightarrow \Xi \qquad \Gamma(X_2) = \Sigma \qquad \Gamma \vdash \Sigma \leq \exists \overline{\alpha}\Sigma' \uparrow \overline{\tau} \rightsquigarrow f}{\Gamma \vdash X_1(X_2) : \Xi[\overline{\tau}/\overline{\alpha}] \rightsquigarrow (X_1 \overline{\tau})(f X_2)}$$

In a sense, you can see how these look like the typical lambda abstraction and application rules from the STLC. In the abstraction rule, we check that the body of the functor has a certain "type" (signature) when evaluated in the environment where the $\overline{\alpha}$ and X are bound. In the application rule, we check that X_1 is actually a functor, and we check that X_2 matches the type that the functor expects.

The application rule introduces the signature matching judment, $\Gamma \vdash \Sigma \leq \exists \overline{\alpha}\Sigma' \uparrow \overline{\tau} \rightsquigarrow f$. The judgment says that the concrete signature Σ *matches* the abstract signature $\exists \overline{\alpha}.\Sigma'$ if one substitutes the types $\overline{\tau}$ for $\overline{\alpha}$, and it gives the conversion function f which converts a module expression matching the concrete signature to one that matches the abstract one with the types substituted in. The rule for that judgment basically just does the substitution for the types and checks that Σ is a subtype of the resulting type.

There are several more rules like the ones I've shown, but the above are some of the most interesting. By following these rules in a syntax-directed way, a program can come up with a complete translation of an ML program into F_ω.

3.4 Soundness

As it turns out, proving soundness is rather easy in this system. Since we're elaborating ML expressions to terms in F_ω (which we know to have a sound type system), all we have to do is prove that the elaboration only produces well-typed terms in F_ω. We can prove this using a relatively simple simultaneous induction on derivations of our various elaboration judgments, but I won't go into any more detail here.

3.5 Controversy of Elaboration Semantics

Something I found interesting: in all of this, while we proved type soundness, we haven't tried to prove any sort of semantic equivalence, which seems especially important for something with a formal definition like SML. Instead, we actually *define* the language in terms of this elaboration. Perhaps future work would look at some sort of semantic equivalence.

3.6 Conclusion

In their work, Russo et al. set out to create a simple, easy-to-understand static and dynamic semantics for the module language of ML. By creating an elaboration semantics and using the idea of semantic signatures to represent modules' abstract types as existential types, they were able to show that the ML module system and its asscoiated type system can be understood in System F_ω alone, without the need for a complicated dependent type system.

References

[1] Derek Dreyer. *Understanding and Evolving the ML Module System.* PhD thesis, Carnegie Mellon University, May 2005.

[2] David B. MacQueen. Using dependent types to express modular structure. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '86, pages 277–286, New York, NY, USA, 1986. ACM.

[3] Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. F-ing modules. In *Proceedings of the 5th ACM SIGPLAN workshop on Types in language*

design and implementation, TLDI '10, pages 89–102, New York, NY, USA, 2010. ACM.