

# Type Systems for Programming Languages (15-814)

## Lecture Notes, Fall 2006, Week 12

Kanat Tangwongsan

December 5 and 7, 2006

### 1 Review

We developed the following important notions in the past two lectures:

- “Dot notation” (direct access). We know that

$$\frac{\cdot \vdash e : \text{sig } t.\tau}{\cdot \vdash \text{rpn}(e) \text{ type}} \qquad \frac{\cdot \vdash e : \text{sig } t.\tau}{\cdot \vdash \text{ops}(e) : [\text{rpn}(e)/t]\tau}$$

- Sealing. Given an expression  $e$  and a type  $\sigma$ , we can impose an abstraction on  $e$  by *sealing* it with  $\sigma$ ; we write  $e :> \sigma$ .
- Determinate Type. An expression  $e$  has a determinate type if the type of  $e$  can be determined at compile time. Recall from previous lectures that, if the type of  $e$  depends on external information (e.g. phase of the moon),  $e$  may not be determinate.

### 2 First-Class and Second-Class Modules

Are signatures types? Are modules expressions? So far, the answers to both questions are yes, because we have a first-class module system. In first-class modules (FCM), we treat signatures as types and modules as expressions.

$$\begin{aligned} \tau &::= \dots \mid \text{sig } t.\tau \mid \text{rpn}(e) && \text{(types)} \\ e &::= \dots \mid \text{str}(\tau, e) \mid \text{ops}(e) && \text{(exprs)} \end{aligned}$$

By contrast, second-class modules (SCM) segregate the notions of signatures and modules from the standard notions of types and expressions. In second-class modules, we have the following grammar:

$$\begin{aligned} \sigma &::= \text{sig } t.\tau[ \dots ] && \text{(signatures)} \\ m &::= u \mid \text{let } u : \sigma \text{ be } m_1 \text{ in } m_2 \mid \text{str}(\tau, e) && \text{(modules)} \\ \tau &::= \dots \mid \text{rpn}(m) && \text{(types)} \\ e &::= \dots \mid \text{ops}(m) && \text{(exprs)} \end{aligned}$$

### 3 Second-Class Modules

The focus of this lecture is on second-class modules. Even though the name second-class modules may sound pejorative, there are many reasons that we should favor them more than the first-class modules.

- In comparison to the first-class scheme, the second-class scheme has an expanded class of determinate modules—modules are harder to fail to be determinate.
- With SCM, we have both FCM and SCM. The idea is to regard FCM as an existential type. The grammar will look like this:

$$\begin{aligned}\tau &::= \dots \mid \exists t. \sigma \\ e &::= \dots \mid \text{pack}(\tau, e) \mid \text{open } e\end{aligned}$$

- First-class modules are undecidable.
- The design of second-class modules does not “infect” the core language with subtyping.

### 3.1 Rules for SCM

We now proceed to develop a scheme of second-class modules by introducing the following judgment forms, which describe our scheme of second-class modules. In this system, the hypotheticals ( $\Gamma$ ) are entries of the forms:  $t$  type,  $x:\tau$ , and  $u \downarrow \sigma$ .

$$\begin{array}{ccc} e \text{ type} & \sigma \text{ sig} & \sigma_1 <: \sigma_2 \\ e : \tau & \sigma_1 \equiv \sigma_2 & \tau_1 \equiv \tau_2 \\ m \text{ det} & m : \sigma & m \downarrow \sigma \end{array}$$

Note that  $m \downarrow \sigma$  is not an actual judgment: the shorthand  $m \downarrow \sigma$  combines together  $m \text{ det}$  and  $\cdot \vdash m : \sigma$ . That is, we write  $m \downarrow \sigma$  to mean “ $m$  is determinate and has type  $\sigma$ ”. The following rules, categorized by their judgment forms, define our system:

$$\boxed{\Gamma \vdash \sigma \text{ sig}}$$

$$\frac{\Gamma, t \text{ type} \vdash \tau \text{ type}}{\Gamma \vdash \text{sig } t. \tau \text{ sig}}$$

$$\boxed{\Gamma \vdash \sigma_1 \equiv \sigma_2}$$

$$\frac{\Gamma, t \text{ type} \vdash \tau \equiv \tau'}{\Gamma \vdash \text{sig } t. \tau \equiv \text{sig } t. \tau'}$$

$$\boxed{\Gamma \vdash m : \sigma}$$

$$\frac{\Gamma \vdash m : \sigma \quad \Gamma \vdash \sigma \equiv \sigma'}{\Gamma \vdash m : \sigma'}$$

$$\frac{\Gamma \vdash m : \sigma}{\Gamma \vdash \underbrace{m : > \sigma}_{\text{impose } \sigma \text{ on } m} : \sigma}$$

$$\frac{\Gamma \vdash \rho \text{ type} \quad \Gamma \vdash e : [\rho/t] \tau}{\Gamma \vdash \text{str}(\rho, e) : \text{sig } t. \tau}$$

$$\frac{\Gamma \vdash \sigma_1 \text{ sig} \quad \Gamma \vdash m_1 \text{ det} \quad \Gamma \vdash \sigma_2 \text{ sig} \quad \Gamma, u : \sigma_1 \vdash m_2 : \sigma_2}{\Gamma \vdash \text{let } u : \sigma_1 \text{ be } m_1 \text{ in } m_2 : \sigma_2}$$

$$\boxed{\Gamma \vdash m \text{ det}}$$

$$\overline{\Gamma, u \text{ det} \vdash u \text{ det}}$$

$$\overline{\Gamma \vdash \text{str}(\rho, e) \text{ det}}$$

Under this set of rules, it is legitimate for  $e$  to depend on outside information (e.g., consult the phase of the moon). However, we note that we cannot have the rule  $\Gamma \vdash e :> \sigma$  *det* in this system.

$$\boxed{\Gamma \vdash \tau \text{ type}}$$

$$\frac{}{\Gamma, t \text{ type} \vdash t \text{ type}} \qquad \frac{\Gamma \vdash m \downarrow \text{sig } t.\tau}{\Gamma \vdash \text{rpn}(m) \text{ type}}$$

$$\boxed{\Gamma \vdash \tau_1 \equiv \tau_2}$$

$$\frac{}{\Gamma \vdash \text{rpn}(\text{str}(\rho, e)) \equiv \rho}$$

Note that, since  $m :> \sigma$  is possibly indeterminate, the type-expression  $\text{rpn}(m :> \sigma)$  is ill-formed.

$$\boxed{\Gamma \vdash e : \tau}$$

The rules for typing expressions are the standard typing rules, together with the following rule:

$$\frac{\Gamma \vdash m \downarrow \text{sig } t.\tau}{\Gamma \vdash \text{ops}(m) : [\text{rpn}(m)/t] \tau}$$

Consider the following example to contrast our scheme of second-class modules with our previous attempts. We recall that, previously, every time we **open** a package, we obtain a new type. This is necessary in our previous schemes, because each time we **open** a package, we obtain a completely independent thing (say, the package depends on the age of the universe).

```

let  $u$  be  $m :> \sigma$  in
  :
  ... Expressions involving  $\text{rpn}(u)$ 
  :
  ... Expressions involving  $\text{ops}(u)$ 
  :

```

In the fragment of code above, we bind the module  $m :> \sigma$  to  $u$ . This is the key to make  $u$  determinate: after this binding, when we refer to  $\text{rpn}(u)$  or  $\text{ops}(u)$ , it refers to the same  $u$ . We never look at the module  $m$  again after binding it to a variable. Therefore, even if  $m$  changes every time someone looks at it, our copy of  $u$  remains the same. We remark that deciding if two expressions are equivalent in general is undecidable; however, in this framework, type equivalence is independent of expression equivalence, allowing us to compare types.

### 3.2 Issues

Despite various advantages of our new design, we wish to critique and improve on certain aspects of this design:

#1 No unicity of types. For example, the expression  $\text{str}(\text{nat}, \lambda x:\text{nat}.x)$  can assume any of the following types, which all differ.

$$\begin{aligned} \text{str}(\text{nat}, \lambda x:\text{nat}.x) &: \text{sig } t.(t \rightarrow t) \\ &: \text{sig } t.(t \rightarrow \text{nat}) \\ &: \text{sig } t.(\text{nat} \rightarrow t) \\ &: \text{sig } t.(\text{nat} \rightarrow \text{nat}) \end{aligned}$$

#2 Conservative with respect to type identity. In the following fragment of code, even though we bind  $v$  to  $u$ , the system cannot conclude that  $\text{rpn}(u)$  is the same as  $\text{rpn}(v)$ .

```

let  $u$  be  $m$  in
  ⋮
  let  $v$  be  $u$  in
    ⋮
     $\text{rpn}(u) \neq \text{rpn}(v)$ 

```

## 4 Translucency

The signature  $\text{sig } t.\tau$  is called *opaque* as it hides the identity of  $t$  from the outside, whereas the signature  $\text{sig } t = \rho.\tau$  is called *transparent* because it reveals that the occurrence of  $t$  in  $\tau$  means  $\rho$ . We informally define the notion of a *principal type* by an example.

$$\text{sig } t = \text{nat}.(t \rightarrow t) \equiv \text{sig } t = \text{nat}.(t \rightarrow \text{nat}) \equiv \text{sig } t = \text{nat}.(t \rightarrow \text{nat}) \equiv \text{sig } t = \text{nat}.(t \rightarrow \text{nat})$$

By revealing the type of  $t$ , everything can have a unique type, called the *principal type*, which provides the unicity in types. The following rules achieve this goal:

$$\frac{\Gamma \vdash \sigma_1 \equiv \sigma_2}{\Gamma \vdash \sigma_1 <: \sigma_2} \text{ (REFL)} \qquad \frac{}{\Gamma \vdash \text{sig } t = \rho.\tau <: \text{sig } t.\tau} \text{ ("Forget")} \qquad \boxed{\Gamma \vdash \sigma_1 <: \sigma_2}$$

$$\frac{}{\Gamma \vdash \text{sig } t = \rho.\tau \equiv \text{sig } t = \rho.[\rho/t]\tau} \text{ (Propagation)} \qquad \boxed{\Gamma \vdash \sigma_1 \equiv \sigma_2}$$

$$\frac{\Gamma \vdash m : \sigma \quad \Sigma \vdash \sigma <: \sigma'}{\Gamma \vdash m : \sigma'} \text{ (SUB)} \qquad \frac{\Gamma \vdash \rho \text{ type} \quad \Gamma \vdash e : [\rho/t]\tau}{\Gamma \vdash \text{str}(\rho, e) : \text{sig } t = \rho.\tau} \qquad \boxed{\Gamma \vdash m : \sigma}$$

With this set of rules, the structure  $\text{str}(\text{nat}, \lambda x:\text{nat}.x)$  resolves to the type  $\text{sig } t.\text{nat} \rightarrow \text{nat}$  through  $\text{sig } t = \text{nat}.t \rightarrow t \equiv \text{sig } t = \text{nat}.\text{nat} \rightarrow \text{nat} <: \text{sig } t.\text{nat} \rightarrow \text{nat}$ .

With a little more work, we can extend our framework to make it less conservative with respect to type identity, fixing the second issue.

$$\frac{\Gamma \vdash m \downarrow \text{sig } t.\tau}{\Gamma \vdash m : \text{sig } t = \text{rpn}(m).\tau} \text{ ("Self")}$$

The “self” rule enables the typing  $\Gamma, x \downarrow \text{sig } t.\tau \vdash x : \text{sig } t = \text{rpn}(x).\tau$ . We now revisit the example we had before.

```

let  $u$  be  $m$  in //  $u$  now has type  $\text{sig } t.\tau$ 
  ⋮
  let  $v$  be  $u$  in // the new rules give that  $u : \text{sig } t = \text{rpn}(u).\tau$ 
    ⋮
     $\text{rpn}(u) = \text{rpn}(v)$  // Now we can conclude this.

```

## 5 Recap: Key Ideas in Modules

Before delving any further, we recall key ideas in our development thus far.

- Dot Notation. This was our motivation to provide a flexible abstraction mechanism for decomposing programs.
- Determinacy. We want modules to have a well-determinate type. Sealed modules do not have this property.
- Translucency. This gives us controlled exposure of the representation information. The literature sometimes refers to it as *type sharing*.

In dealing with the two issues previously discussed, several additions are made to our vanilla SCM. We make certain remarks below.

- In terms of signatures, we add the “forget” rule (sometimes called “forgetting sharing”). The rules for  $\Gamma \vdash \sigma \text{ sig}$  and the “forget” rule are purely static. They have no runtime effects and cannot be simulated with a coercion. This is different from the subtyping relations we previously studied, where one can think that the compiler inserts coercions to the code at compile time to allow subtyping.
- In terms of signature equality ( $\sigma_1 \equiv \sigma_2$ ), we add the propagation rule, as its name suggested, allows for the type information into the signature:  $\text{sig } t = \rho.\tau \equiv \text{sig } t = \rho.[\rho/t]\tau$ .
- In terms of module typing ( $m : \sigma$ ), we introduce rules for (1) subsumption, (2) selfification, (3) initial transparency, and (4) sealing.

## 5.1 Kinds

Both types of signatures—opaque and transparent—are captured by a more general object known as *kind*, written  $\text{sig } t :: K.\tau$ . The notion of kind gives rise to the following new grammar:

$$\begin{aligned}
 \tau &::= \dots \\
 e &::= \dots \\
 K &::= \text{Type} \mid \dots && \text{(kinds)} \\
 C &::= \tau \mid \dots && \text{(constructors)} \\
 \sigma &::= \text{sig } t.\tau \mid \text{sig } t = \rho.\tau \\
 m &::= \dots
 \end{aligned}$$

where  $\tau$  and  $e$  are considered dynamic;  $K$  (kinds) and  $C$  (constructors) static; and  $\sigma$  and  $m$  hybrid. The full treatment of this theory is beyond the scope of this class.

## 5.2 Connections to ML

One may wonder how our module system relates to the module system of ML, especially since our notation for signature ( $\text{sig}$ ) and that of ML are very similar. We point out the following relationships:

$$\text{sig } t.\tau \sim \boxed{\begin{array}{l} \text{sig type } t \\ \text{val } x:\tau \\ \text{end} \end{array}} \qquad \text{sig } t = \rho.\tau \sim \boxed{\begin{array}{l} \text{sig type } t=\rho \\ \text{val } x:\tau \\ \text{end} \end{array}}$$

The corresponding structure is

```

struct
  type t = ρ
  val x:τ = ...
end

```

## 6 Hierarchy (Substructure)

We start with an example that motivates the need for hierarchy. In general, we can imagine declaring a signature which refers to another structure inside of it:

```
sig
  structure S : SIG_1
  type t
  :
end
```

This basic structure forms a tree of dependences. In certain cases, we need a stronger sense of dependence, in which case the dependence structure may form a DAG. For example,

<pre>D = sig   structure S_1 : K_1   structure S_2 : K_2   sharing K_1.t = K_2.t   type t   : end</pre>	<pre>K_1 = sig   type t   : end K_2 = sig   type t   : end</pre>
---	--

This example illustrates a typical situation in a dictionary structure, where possibly  $K_1$  provides an equality test and  $K_2$  provides a hash function, both of which operate on the same key type  $t$ . In such a situation, it is important for the signature  $D$  to capture the idea that  $K_1.t = K_2.t$ . Note that if we can rewrite  $K_2$ , we can change the line `type t` to `type t = K_1.t` to enforce this constraint.

We investigate a systematic solution of this issue below.

### 6.1 Rules of the Game

We augment our language as follows:

$$\sigma ::= \dots \mid \Sigma u : \sigma_1. \sigma_2$$

$$m ::= \dots \mid \langle m_1, m_2 \rangle \mid \mathbf{fst} m \mid \mathbf{snd} m$$

Note that the declaration in the example above can be expressed in this language as  $\Sigma u : \mathbf{sig} t. \tau. \mathbf{sig} t = \mathbf{rpn}(u). \tau'$ . This enforces  $K_2.t = K_1.t$  through the variable  $u$ . We describe an introduction rule for  $\langle m_1, m_2 \rangle$  and illustrate a use of it.

$$\frac{\Gamma \vdash m_1 : \sigma_1 \quad \Gamma \vdash m_2 : \sigma_2}{\Gamma \vdash \langle m_1, m_2 \rangle : \Sigma u : \sigma_1. \sigma_2}$$

This rule is analogous to the pair-introduction rule. As an example, consider that using this rule, we can type  $\langle \mathbf{str}(\mathbf{nat}, =), \mathbf{str}(\mathbf{nat}, <) \rangle < : \Sigma u : (\mathbf{sig} t. t \times t \rightarrow \mathbf{bool}). \mathbf{sig} t = \mathbf{rpn}(u). t \times t \rightarrow \mathbf{bool}$ .

For the elimination rules, we write a seemingly natural set of rules:

$$\frac{\Gamma \vdash m : \Sigma u : \sigma_1. \sigma_2}{\Gamma \vdash \mathbf{fst} m : \sigma_1} \qquad \frac{\Gamma \vdash m : \Sigma u : \sigma_1. \sigma_2}{\Gamma \vdash \mathbf{snd} m : [\mathbf{fst} m/u] \sigma_2}$$

We then argue that the rule for  $\mathbf{snd}$  will not work: The elimination rule  $\mathbf{snd}$  relies on  $\mathbf{fst} m$  having a well-defined type. This unfortunately may not be the case. We have the following remedies:

- Possibility #1. We re-define the elimination rule **snd**, requiring  $m$  is determinate in addition to having type  $\Sigma u : \sigma_1.\sigma_2$ . The rule now reads:

$$\frac{\Gamma \vdash m \downarrow \Sigma u : \sigma_1.\sigma_2}{\Gamma \vdash \mathbf{snd} m : [\mathbf{fst} m/u] \sigma_2}$$

- Possibility #2. We can alternatively extend the notion of determinacy and set up a re-typing rule so that we can break the “pointer” between  $\sigma_1$  and  $\sigma_2$ . Bob asserts that this solution essentially is equivalent to our previous solution. First, change the the elimination rule **snd** to

$$\frac{\Gamma \vdash m : \Sigma \_ : \sigma_1.\sigma_2}{\Gamma \vdash \mathbf{snd} m : \sigma_2}$$

so that the type  $\sigma_2$  no longer refers to  $\sigma_1$ . The notion of determinacy is extended by the following rules:

$$\frac{\Gamma \vdash m_1 \mathit{det} \quad \Gamma \vdash m_2 \mathit{det}}{\Gamma \vdash \langle m_1, m_2 \rangle \mathit{det}} \quad \frac{\Gamma \vdash m \mathit{det}}{\Gamma \vdash \mathbf{fst} m \mathit{det}} \quad \frac{\Gamma \vdash m \mathit{det}}{\Gamma \vdash \mathbf{snd} m \mathit{det}}$$

For re-typing, we have

$$\frac{\Gamma \vdash m \downarrow \Sigma u : \sigma_1.\sigma_2 \quad \Gamma \vdash \mathbf{fst} m \downarrow \sigma'_1 \quad \Gamma \vdash \mathbf{snd} m \downarrow \sigma'_2}{\Gamma \vdash m : \sigma'_1 \times \sigma'_2} \text{ (retyping)}$$

As an example, we consider the following:

$$\begin{aligned} m \downarrow \Sigma u : (\mathbf{sig} t.\tau).(\mathbf{sig} t' = \mathbf{rpn}(u).\tau') &\equiv m \downarrow \Sigma u : (\mathbf{sig} t.\tau).(\mathbf{sig} t' = \mathbf{rpn}(\mathbf{fst} m).\tau') \\ &\equiv m \downarrow \Sigma \_ : (\mathbf{sig} t.\tau).(\mathbf{sig} t' = \mathbf{rpn}(\mathbf{fst} m).\tau') \end{aligned}$$

On the second line, we “propagate” the dependence and remove the link  $u$ . This works because when  $m$  is determinate, we know that  $u = \mathbf{fst} m$ , and now we are allowed to propagate this information and break the link.