The Module Language

Zachary Kieda (zkieda@andrew.cmu.edu)

July 9, 2015

Abstract

At some point in time any project gets large enough that the developer[s] decide to design their own language that will aide in accomplishing their various routine or evil goals.

Here, we present a language used for own sinister goals – creating a easier way to construct Java modules, especially with respect to our CrowdSimulation project. In our project, a module belongs in a hierarchical tree rooted at a player. Modules present an easy way to make additions to the player's functionality, as well as provide an easy way to construct a tree of dependent modules for testing. Ultimately, we will use the Module Language (ModLang) to construct and perform tests, as well as run the code for our crowd simulation project.

At the present moment, we have the lexer, parser, and dynamic semantics working for ModLang 1.0. We will be working towards ModLang 2.0 to provide additional functionality (like importing modules from other files, and a unification of the type system).

Keywords. Programming Languages, Modules

1 Module System (Dynamic Semantics)

1.1 Module Specification

We construct modules in a tree-like system, so modules have two parameterized types, Node and Leaf. We use this specification to define the dynamic semantics of the language.

The signature for the Node module is specified below.

signature for T Node =
  add : T Node → (T Node) Module → T
  init : T Node → ()
  new : () → T Node

We define T Module such that T Node `instanceof` T Module and T Leaf `instanceof` T Module. This means that the add function accepts both Node types and Leaf types. For object-oriented programming languages, we assume that add and init are methods and the current object is the first argument for these functions.

The signature for the Leaf module is specified below.

signature for T Leaf =
  init : T Leaf → ()
  new : () → T Leaf

Signature Contracts.

We assert several contracts associated with these signatures that ensure that the modules work correctly.

For a module of type T Node, we list the method type, the contract type (requires/ensures), and the contract itself

m.init()	requires	$\forall m'$ where m' is ancestor of m, m'.init() has been called.
	ensures	$\forall m'$ where m' is a descendent of m, m'.init() has completed.
m.add(t)	requires	t `instanceof` T Module
	ensures	t is a child of m

A module of type T Leaf has the same requires contract as T Node.init. How-

ever, the ensures contract only ensures that init method has been called on this leaf.

Operations.

Additional operations can be provided in the system. For example, tree traversals that allow a module to communicate with its children/parent in either DFS or BFS ordering.

1.2 Java Implementation

In our java implementation, the types

$$\mathsf{T\ Leaf} = \texttt{SubModule<T>}$$
$$\mathsf{T\ Module} = \texttt{MultiModule<T>}$$

The new method is equivalent to a constructor that takes no args. This means that all modules used in our language have a no-arg constructor.

We provide a utility annotation `@AutoWired` that allows modules to easily access each other without setup. The annotation `@AutoWired` can only be applied to fields of type `Module`.

The code for this annotation is seen below

```
@Retention(RUNTIME)
@Target(FIELD)
public @interface AutoWired {
  public String value() default "";
}
```

All fields of type τ : *Module* annotated with `@AutoWired` will automatically find a class of type τ that is connected to this module tree and inject it into the annotated field. These fields are injected during the *linking* phase.

During the linking phase, we define a boolean variable `noThrow`. If this variable is `false`, we throw a Linking Exception if a module that extends τ could not be found at runtime.

The `value` field allows the user to specify a specific location for the linker to look for an injected class. If `value` = `""`, then the linker will search all

connected modules, and take the nearest child.

When the parameter `value` is non-empty it either defines a relative path or an absolute path to the module we're looking for with respect to the module tree. Java classes in `value` are separated by forward slashes. When `value` begins with a forward slash, it is an absolute path and we search from the root module. Otherwise, we search relative to the current module. We can use `"../"` to access the parent of this module.

Here's an example,

```
@AutoWired("../PerceptionModule/FieldOfVisionModule")
private FieldOfVisionModule fov;
```

Suppose we link with `noThrow` set to `false`. This code will search the parent module for a `PerceptionModule`, then search that module for a `FieldOfVision` module. If we can't find this module, then we throw a linking exception. Otherwise, the module we found is injected into field `fov`.

Module Language 2.0

1 Syntax

Whitespace and Token Delimiting

Whitespace is either a space, a horizontal tab (`\t`), vertical tab (`\v`), linefeed (`\n`), carriage return (`\r`), or form feed (`\f`) in ASCII encoding. Whitespace is ignored, except that it delimits tokens. Note that whitespace is not a *requirement* to terminate a token. Whitespace can disambiguate two tokens when one of them is present as a prefix in another. The lexer should produce the longest valid token sequence possible.

Comments

ModLang source programs may contain SML-style comments of the form `(* ... *)` for multi-line comments. Multi-line comments may be nested (and of course the delimiters must be balanced).

1.1 Tokens

The tokens and lexing is presented below in BNF,

ident	::=	`[a-zA-Z$_][a-zA-Z0-9$_]*`
colon	::=	`:`
dot	::=	`.`
star	::=	`*`
import	::=	`import`
has	::=	`has`
end	::=	`end`
langle	::=	`<`
rangle	::=	`>`
in	::=	`in`
open	::=	`open`
equals	::=	`=`
filePath	::=	`(?:[^/]*/)*?([^/.]*)(?:\\..*)`

Terminal referenced in the grammar are in **bold**. **ident** is described using regular expressions.

1.2 Grammar

The context free grammar is presented below,

⟨program⟩	::=	**import** ⟨moduleList⟩ **in** ⟨module⟩ **end**
	|	⟨module⟩
⟨moduleList⟩	::=	**open** ⟨module⟩ ⟨moduleList⟩
	|	⟨module⟩ ⟨moduleList⟩
	|	**import** ⟨moduleList⟩ **in** ⟨moduleList⟩ **end** ⟨moduleList⟩
	|	ϵ
⟨module⟩	::=	⟨prefix⟩ **colon has** ⟨moduleList⟩ **end** ⟨spec⟩
	|	⟨prefix⟩ **colon** ⟨module⟩
	|	**ident equals** ⟨module⟩
	|	⟨prefix⟩
⟨lvalue⟩	::=	**langle filePath rangle dot ident** ⟨spec⟩
	|	**ident** ⟨spec⟩
⟨prefix⟩	::=	**langle filePath rangle** ⟨spec⟩
	|	**ident** ⟨spec⟩
⟨spec⟩	::=	**dot ident** ⟨spec⟩ | ϵ

The parsed AST is returned as a ⟨program⟩. The ⟨imports⟩ section is used to import or define modules from the project into the current context without adding them to the current module.

1.3 Code Example

We present a code example of ModLang that generates a player with the modules `GameModule`, `PerceptionModule`. The `VieldOfVisionModule` is a child of the `PerceptionModule`.

```
import open edu.cmu.cs464.p3.ai.core
       open edu.cmu.cs464.p3.ai.perception
in
   Player :
   has GameModule
       PerceptionModule : FieldOfVisionModule
   end
end
```

In the first and second lines we declare our imports. The compiled script should return a single, fully initialized and linked module. It's important to note that this language is just markup and is not turing complete. This language is supposed to easily generate a hierarchy of modules to produce a module that will be used in a turing complete language.

2 Elaboration

As the name is intended to suggest, *elaboration* is the process of transforming the literal parse tree into to one that is simpler and more well behaved – the abstract syntax tree. Much of this may be accomplished directly in the semantic actions that accompany the grammar rules.

We propose the following tree structure as the abstract syntax tree

$$
\begin{array}{rcl}
mod & ::= & \mathsf{file}(\mathbf{filePath}) \mid \mathsf{parent}(mod, modList) \\
 & \mid & \mathsf{scope}(mod, \mathbf{ident}) \mid \mathbf{ident} \\
 & \mid & \mathsf{decl}(\mathbf{ident}, mod) \\
modList & ::= & [] \mid mod :: modList \\
 & \mid & \mathsf{context}(modList, modList) :: modList \\
 & \mid & \mathsf{open}(mod) :: modList
\end{array}
$$

The decl structure is used to declare a name in a given scope. The left hand argument refers to the module that will be bound to a name, and the right hand side refers to a scope and name that the left hand argument will be bound to. Note that a decl will only have the structure $\mathsf{decl}(\mathit{mod}, \textbf{ident})$ or $\mathsf{decl}(\mathit{mod}, \mathsf{scope}(\mathit{mod}, \textbf{ident}))$ due to the structure of ⟨lvalue⟩.

We propose the following inference rules that describe how to elaborate the grammar into the abstract syntax tree.

$$\frac{}{\epsilon \rightsquigarrow \texttt{[]}} \qquad \frac{\langle \text{module} \rangle \rightsquigarrow m \qquad \langle \text{moduleList} \rangle \rightsquigarrow M}{\begin{array}{c}\textbf{open}\ \langle \text{module} \rangle\ \langle \text{moduleList} \rangle \rightsquigarrow \mathsf{open}(m)::M, \\ \langle \text{module} \rangle\ \langle \text{moduleList} \rangle \rightsquigarrow m::M\end{array}}$$

$$\frac{\langle \text{moduleList} \rangle \rightsquigarrow M_1 \qquad \langle \text{moduleList} \rangle \rightsquigarrow M_2 \qquad \langle \text{moduleList} \rangle \rightsquigarrow M_3}{\textbf{import}\ \langle \text{moduleList} \rangle\ \textbf{in}\ \langle \text{moduleList} \rangle\ \textbf{end}\ \langle \text{moduleList} \rangle \rightsquigarrow \mathsf{context}(M_1, M_2)::M_3}$$

$$\frac{\langle \text{prefix} \rangle \rightsquigarrow p \qquad \langle \text{modList} \rangle \rightsquigarrow M}{\langle \text{prefix} \rangle\ \textbf{colon has}\ \langle \text{modList} \rangle\ \textbf{end}\ \epsilon \rightsquigarrow \mathsf{parent}(p, M)}$$

$$\frac{\langle \text{prefix} \rangle\ \textbf{colon has}\ \langle \text{modList} \rangle\ \textbf{end} \langle \text{spec} \rangle \rightsquigarrow M \qquad \textbf{ident} \rightsquigarrow id}{\langle \text{prefix} \rangle\ \textbf{colon has}\ \langle \text{modList} \rangle\ \textbf{end}\ \langle \text{spec} \rangle\ \textbf{dot ident} \rightsquigarrow \mathsf{scope}(M, id)}$$

$$\frac{\langle \text{prefix} \rangle \rightsquigarrow p \qquad \langle \text{module} \rangle \rightsquigarrow m}{\langle \text{prefix} \rangle\ \textbf{colon}\ \langle \text{module} \rangle \rightsquigarrow \mathsf{parent}(p, \texttt{[}m\texttt{]})}$$

$$\frac{\textbf{ident} \rightsquigarrow id \qquad \langle \text{module} \rangle \rightsquigarrow m}{\textbf{ident equals}\ \langle \text{module} \rangle \rightsquigarrow \mathsf{decl}(id, m)}$$

$$\frac{\textbf{filePath} \rightsquigarrow f}{\textbf{langle filePath rangle}\ \epsilon \rightsquigarrow \mathsf{file}(f)}$$

$$\frac{\textbf{langle filePath rangle}\ \langle \text{spec} \rangle \rightsquigarrow M \qquad \textbf{ident} \rightsquigarrow id}{\textbf{langle filePath rangle}\ \langle \text{spec} \rangle\ \textbf{dot ident} \rightsquigarrow \mathsf{scope}(M, id)}$$

Here's an example translation from code to AST,

```
(* example syntax *)
<hello/World.mod>.asdf:g = <hello/Player.mod>:has
 open modules.player
end
```

And here's the resulting AST from using the inference rules,

```
scope(file("hello/World.mod"), "asdf"),
  parent(file("hello/Player.mod"),
    [open(scope("modules", "player"))]))
```

3 Static Semantics

> This is the section where the developer goes completely insane and makes her/his homebrew language as meta as possible. We define unified system of importing and using modules. This system allows the user to import and use ModLang files and java packages as as modules. We also formally define and use scope of modules to select submodules. This turns the language into more of a build system but whatever.

3.1 File Imports

Notice that any module program is represented by a single module. This allows a ModLang file to represent a module directly, much like how a java file must can only represent a single public java class.

The first capture group of a **filePath** represents the name an imported module is bound to. This capture group retrieves the file name without the file's extension. Thus, a file f with file name n : **ident** is represented in the AST as $(n, f) : mod$.

We permit escape sequences in a **filePath** to represent whitespace and other characters that may be represented. We permit all java escape sequences, along with a few more. We add in the `\s` escape sequence to represent a space, and the `\>` escape sequence to represent a `>` character. Actual whitespace in a **filePath** can be used to delimit tokens and is ignored. A token in a **filePath** is a file separator (`/`), or a sequence of non-whitespace tokens as a file name (**ident**).

In order to express the static semantics of this language, we use a unholy mashup of denotational semantics notation and classic static semantics notation.

Since we're dealing with classes in java, a type might have many super types because of inheritance. We write $\tau <: \tau'$ when the class τ is a subtype of, or extends the class τ'. We define the type **Ty** to describe java classes, such that $\tau \in \mathbf{Ty}$ if and only if $\tau <: \texttt{Module}$.

We define the type $\mathbf{Ns} = \mathit{javaRef} \rightharpoonup \mathbf{Ty}$ to define our namespace. The partial function $\Gamma : \mathbf{Ns}$ to express a namespace from a java reference to a java module class. Classically $\Gamma \vdash x : \tau$ *valid* is written to express that expression x has type τ under the context Γ. Here, we will write $(x, \tau) \in \Gamma$ as an equivalent expression.

The partial function $\texttt{ClassLoader} : \mathit{javaRef} \rightharpoonup \texttt{Class<?>}$ is used to access java classes by using their abstract path name. The `ClassLoader` is a component of the JVM standard used to access classes reflectively.

We define the initial state of the program. The initial namespace, Γ_i, is

$$(x, \tau) \in \Gamma_i \iff (x, \texttt{Class<}\tau\texttt{>}) \in \texttt{ClassLoader}, \tau <: \texttt{Module}$$

When parsing, we also keep track of a *secondary* context, $\mathcal{C} : \mathbf{Ty}$, that keeps track of the current module type. This ensures that added submodules will also have a valid type with respect to their parent.

In order to formally define evaluation rules, we define the current state of the evaluation as $\mathbf{St} = (\mathbf{Ns}, \mathbf{Ty})$. For any statement s in our AST, $|\, e \,| : \mathbf{St} \rightharpoonup \mathbf{St}$ is a partial function. We say $(\sigma, \sigma') \in |\, e \,|$ if and only if the expression e is valid with the starting state σ, and will conclude with an ending state σ'.

$$\frac{}{(\sigma, \sigma') \in \texttt{[]}} \qquad \frac{(\sigma, \sigma') \in |\, \mathit{mod} \,|}{\sigma = \sigma'}$$

$$\frac{(\sigma, \sigma') \in |\, \mathsf{program}(\mathit{imports}, \mathit{mod}) \,|}{\exists \sigma''.\ (\sigma, \sigma'') \in |\, \mathit{import} \,|,\ (\sigma'', \sigma') \in |\, \mathit{mod} \,|} \qquad \frac{(\Gamma, \mathcal{C}), (\Gamma', \mathcal{C}') \in |\, \mathit{import} \,|}{\mathcal{C} = \mathcal{C}'}$$

Notice that in our evaluation rules, we allow imports to shadow previously defined imports.

4 Design Patterns

Because of the unification of the module system, many possible design patterns become possible. A few are listed below

4.1 Override a Module

```
<modules/Player.module> :
has
  PerceptionModule.FieldOfVision
    = <modules/AlternativeFov.module>
end
```

Here, we bind the submodule name `FieldOfVision` in the module `Player.PerceptionModule` to the module defined by the file `modules/AlternativeFov.module`. This will use the `AlternateFov` in place of any existing module named `FieldOfVision`. Note that due to our dynamic semanics, the module `FieldOfVision` is never instantiated or added to the resulting module.

4.2 Aggregation

Suppose we have some aggregation module `Agg` that take modules of a certain type, and returns an aggregate of all of its submodules. If we have a package that defines a series of modules that `Agg` can aggregate, we could build the aggregator as follows,

```
import
  (* com.company has the Agg
     class as well as a package
     util.aggregators *)
  com.company
in
  Agg:
  has
   open util.aggregators
  end
end
```

4.3 Union

One can union multiple modules together into a single module.

```
import
  <modules/Player.module>
  <modules/PlayerModifications0.module>
  <modules/PlayerModifications1.module>
in
  Player :
  has
   open PlayerModifications0
open PlayerModifications1
  end
end
```

In this case, the names in `PlayerModifications0` and `PlayerModifications1` will shadow names defined in `Player`. Similarly, the names in `PlayerModifications1` will shadow names in `PlayerModifications0` if there are name conflicts.