The Module Language

Zachary Kieda (zkieda@andrew.cmu.edu)

June 12, 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 $\to$ (T Node) Module $\to T$
init	:	T Node $\to$ ()
new	:	() $\to$ 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 $\to$ ()
new	:	() $\to$ 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. However, 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 "";
  public boolean noThrow() default true;
}
```

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.

If `noThrow` 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",
  noThrow=false)
private FieldOfVisionModule fov;
```

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 1.0

1 Syntax

1.1 Tokens

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 lecture should produce the longest valid token sequence possible.

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`

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⟩	::=	⟨imports⟩ ⟨module⟩
	|	⟨module⟩
⟨javaRef⟩	::=	**ident dot** ⟨javaRef⟩
	|	**ident**
⟨module⟩	::=	⟨javaRef⟩
	|	⟨javaRef⟩ **colon has** ⟨moduleList⟩ **end**
	|	⟨javaRef⟩ **colon** ⟨module⟩
⟨moduleList⟩	::=	⟨module⟩ ⟨moduleList⟩
	|	ϵ
⟨moduleImport⟩	::=	**ident dot star**
	|	**ident**
	|	**ident dot** ⟨moduleImport⟩
⟨imports⟩	::=	**import** ⟨importList⟩ **end**
⟨importList⟩	::=	⟨moduleImport⟩ ⟨importList⟩
	|	ϵ

The parsed AST is returned as a ⟨program⟩. The ⟨imports⟩ section is used to import existing java modules from the project given by their absolute java path.

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
  edu.cmu.cs464.p3.ai.core.*
  edu.cmu.cs464.p3.ai.perception.*
end

Player :
has
  GameModule
  PerceptionModule : FieldOfVisionModule
end
```

In the first four lines we declare our imports. There is at most one import block, which must be declared before a module. 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 inference rules that describe how to elaborate the grammar into the abstract syntax tree.

$$\frac{}{\epsilon \rightsquigarrow \texttt{[]}} \qquad \frac{\langle\text{imports}\rangle \rightsquigarrow imp \qquad \langle\text{module}\rangle \rightsquigarrow m}{\langle\text{imports}\rangle\ \langle\text{module}\rangle \rightsquigarrow \mathsf{declare}(imp, m)}$$

$$\frac{\langle\text{module}\rangle \rightsquigarrow m \qquad \langle\text{moduleList}\rangle \rightsquigarrow M}{\langle\text{module}\rangle\ \langle\text{moduleList}\rangle \rightsquigarrow m :: M}$$

$$\frac{\langle\text{javaRef}\rangle \rightsquigarrow j \qquad \langle\text{module}\rangle \rightsquigarrow m}{\langle\text{javaRef}\rangle\ \textbf{colon}\ \langle\text{module}\rangle \rightsquigarrow \mathsf{parent}(\mathit{ref}(j), [m])}$$

$$\frac{\langle\text{module}\rangle \rightsquigarrow m \qquad \langle\text{moduleList}\rangle \rightsquigarrow M}{\langle\text{module}\rangle\ \textbf{colon has}\ \langle\text{moduleList}\rangle\ \textbf{end} \rightsquigarrow \mathsf{parent}(m, M)}$$

$$\frac{\langle\text{importList}\rangle \rightsquigarrow L}{\textbf{import}\ \langle\text{importList}\rangle\ \textbf{end} \rightsquigarrow L}$$

$$\frac{\langle\text{moduleImport}\rangle \rightsquigarrow l \qquad \langle\text{importList}\rangle \rightsquigarrow L}{\langle\text{moduleImport}\rangle\ \langle\text{importList}\rangle \rightsquigarrow l::L}$$

3 Static Semantics

We write $\Gamma \vdash x : \tau$ to express an expression x is type checked under the context Γ which keeps track of all declarations and their types. The following inference rules demonstrate the function of the context

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

We have several types used for τ. The first is $\mathsf{JavaPath}$, which is an abstract pathname produced by the nonterminal ⟨moduleImport⟩ and ⟨javaRef⟩. The partial function $ref : \mathsf{JavaPath} \rightharpoonup \mathsf{Module}$ defines the referenced java module.

Module Language 2.0

This is the section where the developer goes completely insane and attempts to make her/his homebrew language as meta as possible. ModLang 2.0 defines a new, unified system of importing and using modules. This system allows the user to import and use files of the ModLang as modules themselves, as well as using java packages 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.

1 Syntax

1.1 Code Example

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


CoreModules = edu.cmu.cs464.p3.ai.core
core.GameModule
```