Lecture Overview

Implementing TypeVariables
A lextant switcheroo
Type Variable Implementation

First off, we want to put type variables into signatures, so we must make TypeVariable a subclass of Type.

```java
package semanticAnalyzer.types;

import asmCodeGenerator.codeStorage.ASMOpcode;

public class TypeVariable implements Type {
    private String name;
    private Type typeConstraint;

    public TypeVariable(String name) {
        this.name = name;
        this.typeConstraint = PrimitiveType.NO_TYPE;
    }
}
```

The `name` field is a symbolic name mainly for use in debugging. The `typeConstraint` field is the type that this variable is set to.
Add a getter for the name, and a getter and setter for the typeConstraint. I call the latter getType() and setType(). Make the setter private.

To finish up for now, we must implement the methods of the interface Type. These are
  • getSize(), which can return anything because TypeVariables will never appear on the ASM;
  • infoString(), which can return toString();
  • pikaNativeString(), which can return toString() as well, because there is no native Pika syntax for specifying a type variable.

It’d also be a good idea to implement toString(), which I do by printing out the name field in angle brackets.

```java
public String toString() {
    return "<" + getName() + ">";
}
```

We now have a minimal working TypeVariable. Let’s go write a FunctionSignatures using it. We’ll write the
FunctionSignatures for array indexing. With a TypeVariable $S$, this should have the single signature: $a[S] \times i \rightarrow S$

```java
TypeVariable $S$ = new TypeVariable("S");
new FunctionSignatures(Punctuator.ARRAY_INDEXING,
    new FunctionSignature(
        new ArrayIndexingCodeGenerator(),
        new Array($S$), INTEGER, $S$
    )
);
```

You’ll need to have your Array class implemented for this to not give you errors. Array is another subclass of Type, and it takes a type (the array subtype) as a constructor argument.

Now let’s head over to FunctionSignature to look at the matching algorithm.
```java
public boolean accepts(List<Type> types) {
    if(types.size() != paramTypes.length) {
        return false;
    }

    for(int i=0; i<paramTypes.length; i++) {
        if(!assignableTo(paramTypes[i], types.get(i))) {
            return false;
        }
    }

    return true;
}

That was straightforward. Each parameter is being compared to each argument by assignableTo(). This is good nomenclature for when FunctionSignature will be used for functions, because we literally assign the argument values to the parameter values.

```
assignableTo() uses <type>.equals() to compare types. We need to generalize this so that TypeVariables can be “equal” to anything on their first occurrence in a match, but only “equal” to that same thing on their later occurrences in a match.

equals() is a java standard routine with a specific contract required by its collection classes, so we don’t want to mess with equals(). We’ll write something similar, but call it equivalent().

```java
private boolean assignableTo(Type variableType,
                             Type valueType) {
    if (valueType == PrimitiveType.ERROR &&
        ALL_TYPES_ACCEPT_ERROR_TYPES) {
        return true;
    }
    return variableType.equivalent(valueType);
}
```

Adding this code creates an error, because equivalent() is not yet a method on Type. So add

```java
public boolean equivalent(Type type);
```

to the interface Type. In doing so, you will get errors showing up in your implementors of type. Let’s address them one-by-one.
In PrimitiveType, we just want to check if the given type is the same constant as ‘this’:

```java
@Override
public boolean equivalent(Type type) {
    return this == type;
}
```

In Array (or ArrayType, whichever you call yours), we need to check that the given type is an array, and that it has the same subclass as ‘this’:

```java
@Override
public boolean equivalent(Type otherType) {
    if (otherType instanceof Array) {
        Array otherArray = (Array)otherType;
        return subtype.equivalent(otherArray.getSubtype());
    }
    return false;
}
```

If you are using TypeLiterals, then you should do a similar thing in that class.

Now we come to the equivalence function for TypeVariable:
public boolean equivalent(Type otherType) {
    if (otherType instanceof TypeVariable) {
        throw new RuntimeException(
            "equivalent attempted on two types containing type variables."
        );
    }
    if (this.getType() == PrimitiveType.NO_TYPE) {
        setType(otherType);
        return true;
    }    
    return this.getType().equivalent(otherType);
}

If the variable is set to NO_TYPE, then it sets itself to the other type, and returns that it matches. If the variable is not set to NO_TYPE, then the type that it is set to must be equivalent to the other type.

But there’s a subtle flaw here…the TypeVariable gets set to NO_TYPE in the constructor, but the same TypeVariable is used for many matches! It won’t get reset between attempted matches.

The best place to fix this problem is in FunctionSignatures.accepts(), which looks at one attempted match of the signature.
public boolean accepts(List<Type> types) {
    resetTypeVariables();

    if(types.size() != paramTypes.length) {
        ...
    }

    That certainly expresses the intent. But how can the FunctionSignature implement resetTypeVariables? There are (at least!) two ways to do this.

    1. Add a method resetTypeVariables on Type. This method would have to be recursive (like equivalence was), and TypeVariables would implement it by resetting themselves. The FunctionSignature would simply call resetTypeVariables on its argument and result types.

    2. Pass a list of type variables to FunctionSignature, so it knows which TypeVariables to reset.

    I’ll do the second option. This means adding a constructor argument to FunctionSignature. Since I don’t want to mess up any existing signatures, I’ll make a new constructor:
public FunctionSignature(Object whichVariant, List<TypeVariable> typeVariables, Type ...types) {

    assert(types.length >= 1);
    this.typeVariables = typeVariables;
    storeParamTypes(types);
    resultType = types[types.length-1];
    this.whichVariant = whichVariant;
}

To do this, I have to add a list of TypeVariables called typeVariables to the fields of FunctionSignature:

    private List<TypeVariable> typeVariables;

Now I rewrite the old constructor to make sure that this field is always initialized:

    public FunctionSignature(Object whichVariant, Type ...types) {
        this(whichVariant,
            new ArrayList<TypeVariable>(),
            types);
    }

    That constructor simply calls the other one (the one at the top of the page).
Now, `resetTypeVariables()` is easy to implement:

```java
private void resetTypeVariables() {
    for (TypeVariable T : typeVariables) {
        T.reset();
    }
}
```

All that remains is to modify array indexing’s `FunctionSignatures` to pass along the list of `TypeVariables` to `FunctionSignature`.

```java
TypeVariable S = new TypeVariable("S");
List<TypeVariable> setS = Arrays.asList(S);

new FunctionSignatures(Punctuator.ARRAY_INDEXING,
   new FunctionSignature(
       new ArrayIndexingCodeGenerator(),
       setS,
       new Array(S), INTEGER, S)
);
```

This works to recognize valid array indexing expressions, but it fails when giving the result type. We haven’t provided for returning a result type when there is a type variable in that type.
To do so, we’ll add another method on Type that will return the type with the type variables substituted out. We’ll call it getConcreteType(). In Type.java:

```java
public Type getConcreteType();
```

This again causes errors in all of Type’s implementors. For a PrimitiveType, the concrete type is just itself:

```java
@Override
public Type getConcreteType() {
    return this;
}
```

For arrays, we must return the type that is an array of the concrete type of the subclass.

```java
@Override
public Type getConcreteType() {
    Type concreteSubtype =
        subtype.getConcreteType();

    return new Array(concreteSubtype);
}
```
And for type variables, we must return the concrete type of the type that the type variable is bound to. Hopefully the subtype is already concrete, but I like to be sure:

```java
@Override
public Type getConcreteType() {
    return getType().getConcreteType();
}
```

Recall that I called the getter for the typeConstraint “getType()”. You can use “getTypeConstraint” if that makes it clear to you.

We’ll also need to change FunctionSignature to get a concrete type as the result type:

```java
public Type resultType() {
    return resultType.getConcreteType();
}
```

Phew! TypeVariables work now. Clearly, they were just a little feature.
If you’re wondering how I got a Punctuator.ARRAY_INDEXING rather than a Punctuator.LEFT_BRACKET, I did a sneaky thing.

First, I defined Punctuator.ARRAY_INDEXING with an empty string as the lexeme. This means the punctuator scanner will never create this punctuator.

```markdown
LEFT_BRACKET("[")
...
ARRAY_INDEXING(""")
```

So the lexical analyzer will never create a LextantToken with the Punctuator ARRAY_INDEXING.

In the parser, where I parse array indexing, I look for the LEFT_BRACKET, but create an artificial LextantToken with the ARRAY_INDEXING punctuator. I use that to construct the [Binary]OperatorNode rather than the real token.
ParseNode base = parseAtomicExpression();
while(nowReading.isLextant(Punctuator.LEFT_BRACKET)) {
    Token realToken = nowReading;
    Token indexToken = LextantToken.artificial(realToken,
        Punctuator.ARRAY_INDEXING);
    readToken();
    ParseNode index = parseExpression();
    expect(Punctuator.CLOSE_BRACKET);

    base = OperatorNode.withChildren(indexToken,
        base, index);
}
return base;

Where LextantToken.artificial looks like:

public static LextantToken artificial(  
        Locator locator, Lextant lextant) {

    String lexeme = lextant.getLexeme();
    if(lexeme.equals("")) {
        lexeme = lextant.toString();
    }
    return new LextantToken(locator.getLocation(),
        lexeme, lextant);
}