15-312 Foundations of Programming Languages
Recitation 11: Java and EML

Daniel Spoonhower
spoons+@cs

November 12, 2003

1 Comparing Java and EML

Today, in order to understand the similarities and differences of Java and EML, we'll translate a ordinary piece of Java code into EML. Our example will be an implementation of a MinML interpreter, so the purpose of the following code should be clear to you. We will begin by considering the implementation of de Bruijn translation.

```
abstract class Exp
{
  abstract Exp translate (Ctx ctx);
}

interface Ctx
{
  void bind (String name);
  int lookup (String name);
  void unbind ();
}

class NamedVar extends Exp
{
  String name;
  Exp translate (Ctx ctx)
  {
    return new DBVar (ctx.lookup(name));
  }
}
```

```
class DBVar extends Exp
{
  int index;
  DBVar (int i) { index = i; }
  Exp translate (Ctx ctx) { return this; }
}

// fn(t, x.e)
class Fn extends Exp
{
  Typ typ;
  String var;
  Exp body;

  Exp translate (Ctx ctx)
  {
    ctx.bind(var);
    Exp b = body.translate (ctx);
    ctx.unbind();
    return new Fn (typ, var, b);
  }
}
```

You might claim that this definition of `Fn.translate` is not "reasonable" for a Java programmer: why should we create a whole new `Fn` object when we already have a perfectly good one (i.e. `this`)? For now, take the example as it stands; we'll make a bit more realistic in a few minutes. Here's my EML translation:

```
abstract class Exp of {};
abstract class Typ of {};

class NamedVar extends Exp of { name:string };
class DBVar extends Exp of { index:int };
class Fn extends Exp of { typ:Typ, var:string, body:Exp };

abstract class Ctx of {};
fun lookup : (Ctx * string) -> int;
fun bind : (Ctx * string) -> unit;
fun unbind : (Ctx) -> unit;

fun translate : (Exp * Ctx) -> Exp;

extend fun translate (e as NamedVar {name=n}, ctx) =
  DBVar (lookup (ctx, n));
extend fun translate (e as DBVar, ctx) = e;
```

```
extend fun translate (e as Fn {typ=t, var=v, body=b}, ctx) =
  (bind (ctx, v);
   let b' = new Fn(v, translate (b, ctx)) in
     unbind ctx;
     Fn (t, v, b')
   end);
```

Now let's reconsider `Fn.translate`. As a Java programmer, I would probably write the following implementation.

```
Exp translate (Ctx ctx)
{
  ctx.bind(var);
  body = body.translate (ctx);
  ctx.unbind();
}
```

How would we translate this code? We might have been mistaken in our translation of the `Fn` class itself; perhaps this definition will suit us better.

```
class Fn extends Exp of { typ:Typ, var:string, body:Exp ref };
```

Now what does our translation of `Fn.translate` look like? Generally speaking, when should we use τ `ref` as the type of a field rather than τ?

Now let's say that, having completed our de Bruijn translation, we would like to implement a typechecker, and since we will probably use types later in our implementation,[1] we would like to keep them around. Consider the following changes to our interpreter.

```
abstract class Exp
{
  Typ typ;
  abstract Exp translate (Ctx ctx);
  // For some type environment env, set the type of this
  // expression.
  void typecheck (Env env);
}
```

How must our definition of `Fn` change? In particular, think about field shadowing. Now take this implementation of typechecking for functions.

[1] Think about when types will be useful.

```
class Fn extends Exp
{
...
  void typecheck (Env env)
  {
    // Assumes variables have been converted to de Bruijn indices
    env.extend (typ);
    body.typecheck (env);
    env.retract();
    super.typ = new Arrow (typ, body.typ);
  }
}
```

How can we translate this into EML? What's different about `Exp.typ` (as compared to `DBVar.index`)? Hint: think about how we'd represent the Java construct `null`.