Continuations

Almost all of our languages have relied on recursion to implement iterative behavior. See, for example, the implementation of `while` in the material on call-by-name. In our implementations, any `eval` call requires setting up a Java stack frame to hold the arguments to `eval`. If evaluating the arguments requires additional calls to `eval`, additional stack frames are required. So if a procedure calls itself recursively, it is possible that stack frames can build up to exhaust available stack memory. Even relatively small programs can result in stack overflow.

An `eval` method call is intended to carry out a (partial) computation – one that may be used, for example, to evaluate an actual parameter expression or to evaluate a test expression in an `if` expression. A stack frame gets created implicitly by the Java Runtime Environment (JRE) upon each `eval` call. This stack frame consists of information including the method arguments, where to find non-local variables (i.e., an environment), and a “return address” that indicates where the JRE should execute next when the method finishes. This information can be considered as an “execution context”. Once the method finishes, this context is discarded (popped off the stack) and the execution context of the caller takes over.

One way to avoid stack overflow is to maintain execution context explicitly. Instead of using the stack to hold this information, we pass along an execution context to the `eval` method that is used by the method to determine what should be executed next when the method finishes. Such an execution context is called a `continuation`. The idea is that a continuation determines how the computation should continue once the current (partial) computation is finished. This means that the method performs a partial computation and then invokes the continuation to carry out further action.
Continuations (continued)

We start with language \texttt{REF}, a language with set and call-by-reference parameter passing. In this language, the \texttt{eval} method for expressions has a single \texttt{env} parameter that gives the environment in which the expression is evaluated. In the \texttt{REFCONT} language, adding explicit execution context requires passing another parameter to the \texttt{eval} method, namely a continuation. The purpose of the continuation is to receive the value of the expression and to determine what to do next.

We implement continuations as members of the \texttt{Cont} class. A continuation may have an \texttt{apply} method that takes no parameters, or it may have an \texttt{apply(Val val)} method that takes a \texttt{Val} parameter. Here are the two principal \texttt{apply} method signatures:

\begin{verbatim}
public Cont apply(Val val);
public Cont apply();
\end{verbatim}

If a continuation’s \texttt{apply} method takes a \texttt{Val} parameter, the method receives a passed value and carries out some action based on the value. If a continuation’s \texttt{apply} method takes no parameters, it simply carries out some action. In both cases, the continuation’s \texttt{apply} method returns another continuation to determine what to do next. Some continuations implement \texttt{apply} methods with both signatures.

The parameterless \texttt{apply} method is the fundamental action that starts off evaluation. This method carries out some action and returns another continuation whose \texttt{apply} method is invoked, and so on. This proceeds until an \texttt{apply} method stops the evaluation by throwing an exception – either a runtime exception indicating an error, or a special \texttt{ContException} indicating that the expression evaluation has finished.

In the absence of an exception, applying a continuation involves creating a new execution context that continues the expression evaluation.

Here is the signature of the abstract \texttt{eval} method in our continuation-passing language, declared in the \texttt{Exp} class:

\begin{verbatim}
public abstract Cont eval(Env env, Cont cont);
\end{verbatim}
Continuations (continued)

It may appear that we have simply added recursive calls to `apply` to the already recursive calls to `eval`. The difference is that once we call `apply`, we do not need to preserve any prior execution context. In implementation languages such as Scheme, we would let the Scheme interpreter manage this through tail call elimination. In Java it’s not as simple – Java does not do tail call elimination – but we accomplish essentially the same thing through a technique called trampolining. This technique de-couples the recursive calls to `apply` by looping instead. Here’s essentially how it works:

```java
public abstract class Cont {

    public Cont apply(Val val) {
        // defaults to throwing an exception
    }

    public Cont apply() {
        // defaults to throwing an exception
    }

    public Val trampoline() {
        Cont cont = this;
        while (true)
            try {
                cont = cont.apply();
            } catch (ContException e) {
                return e.val;
            }
    }
}
```
Continuations (continued)

Things start off by creating an initial expression evaluation continuation `cont` and then jumping onto the trampoline:

```java
cont.trampoline();
```

In this code, the trampoline loops until one of the calls to `apply` throws a `ContException`. In order for expression evaluation to terminate, some continuation must therefore throw a `ContException` that jumps off the trampoline. We choose a special “halt continuation” `HaltCont` to do this. The `apply` method in this class – the one that actually jumps off the trampoline – is now trivial.

```java
public class HaltCont extends Cont {

    public Cont apply(Val val) {
        throw new ContException(val);
    }
}
```

This continuation is created once, at the top level of expression evaluation.
Continuations (continued)

One useful continuation, EvalCont, has fields for an expression, an environment, and another continuation. When an EvalCont continuation is applied, the expression is evaluated in the given environment and its value is passed to the saved continuation.

```java
public class EvalCont extends Cont {
    public Exp exp;
    public Env env;
    public Cont cont;

    public EvalCont(Exp exp, Env env, Cont cont) {
        this.exp = exp;
        this.env = env;
        this.cont = cont;
    }

    public Cont apply() {
        return exp.eval(env, cont);
    }
}
```

We assume that the saved continuation has a apply (Val) method that can receive the expression value.
Continuations (continued)

In the Exp class, we start things out by defining a simple top-level eval method as follows:

```java
public Val eval(Env env) {
    Cont cont = new EvalCont(this, env, new HaltCont());
    Val val = cont.trampoline();
    return val;
}
```

As described on slide 7.4, The HaltCont continuation created here has the default behavior of jumping off the trampoline by throwing a ContException, so once the top-level evaluation is complete and its value is passed to the apply method in the HaltCont object, the trampoline loop stops and this value is returned.

The environment env in this code starts out to be the top-level (initial) environment in the interpreter, which may have some bindings to variables as a result of define expressions.
Certain expression evaluations (such as LitExp and VarExp) involve passing a value directly to a continuation, like this:

```java
    return cont.apply(...)
```

This results in building a stack frame to call the apply method, contrary to our objective of using continuations to avoid building stack frames. In most expressions, a call to apply with a Val parameter immediately returns another continuation that the trampoline will apply with no parameters, but in some continuations, a call to an apply method with a value parameter may need to invoke another apply method with a value parameter; in these cases, the call stack can grow without bounds. To get around this, we define a special ValCont continuation that side-steps the direct call to apply with a value parameter and that passes the responsibility to the non-parameter apply method in the ValCont class:

```java
public class ValCont extends Cont {

    public Val val;
    public Cont cont;

    public ValCont(Val val, Cont cont) {
        this.val = val;
        this.cont = cont;
    }

    public Cont apply() {
        return cont.apply(val);
    }
}
```
Continuations (continued)

As noted before, the continuation-based eval method in the Exp class has the following signature:

```java
public abstract Cont eval(Env env, Cont cont);
```

Let’s consider the easiest subclasses of Exp, namely LitExp and VarExp. The eval code for the LitExp class is simple: convert the literal to an IntVal and apply this value to the continuation. Similarly, for the VarExp class, look up the variable in the given environment and apply this value to the continuation. Here is the code for both:

```java
LitExp
%%{
   public Cont eval(Env env, Cont cont) {
      return cont.apply(new IntVal(lit.toString()));
   }
}
    
VarExp
%%{
   public Cont eval(Env env, Cont cont) {
      return cont.apply(env.applyEnv(var.toString()));
   }
}
```
Continuations (continued)

A `letrec` simply creates a new environment with bindings of identifiers to `ProcVals`. Moreover, evaluating a `proc` (don’t confuse this with *applying* a `proc`) requires no more than gathering together the formal parameter list, the procedure body expression, and the captured environment. Thus the environment in which we evaluate the `letrec` body is obtained by calling the `addBindings` method in the `LetrecDecls` class, unchanged from the `REF` language. The result of this evaluation can then be applied to the same continuation, but with the extended environment. Here is the code for `eval` in the `LetrecExp` class:

```java
LetrecExp
%%{
    public Cont eval(Env env, Cont cont) {
        Env nenv = letrecDecls.addBindings(env);
        return new EvalCont(exp, nenv, cont);
    }
}
```

Notice that we don’t call the continuation’s `apply` method directly here, since we expect that the `EvalCont` continuation will ultimately do so by jumping onto the trampoline.

The `eval` method in the `ProcExp` class is even simpler, since creating a closure does not require any further evaluation.

```java
ProcExp
%%{
    public Cont eval(Env env, Cont cont) {
        return cont.apply(proc.makeClosure(env));
    }
}
```
Continuations (continued)

An if expression requires that the test expression be evaluated before evaluating exactly one of trueExp or falseExp. So after evaluating the test expression, an IfCont continuation uses the result of the test to determine which of these two expressions must be evaluated next. The resulting value is then passed on to the original continuation. The IfCont class appears as follows:

```java
public class IfCont extends Cont {
    public Exp testExp;
    public Exp trueExp;
    public Exp falseExp;
    public Env env;
    public Cont cont;
    public Cont evalCont;

    public IfCont(Exp testExp, Exp trueExp, Exp falseExp, Env env, Cont cont) {
        this.testExp = testExp;
        this.trueExp = trueExp;
        this.falseExp = falseExp;
        this.env = env;
        this.cont = cont;
    }

    // continued on next slide ...
```
Continuations (continued)

// ... continued from previous slide

public Cont apply(Val val) {
    if (val.isTrue())
        return new EvalCont(trueExp, env, cont);
    else
        return new EvalCont(falseExp, env, cont);
}

public Cont apply() {
    return testExp.eval(env, this);
}

In the IfExp class, the eval method simply creates the appropriate IfCont continuation and passes control to it.

IfExp
%%{
    public Cont eval(Env env, Cont cont) {
        return new IfCont(testExp, trueExp, falseExp, env, cont);
    }
}
Continuations (continued)

To evaluate a `SeqExp`, we must evaluate each of the expressions in the sequence in turn, keeping only the value of the last expression and passing it on to the continuation. We create a `SequenceCont` continuation that has fields for an iterator that produces the next expression in the sequence if there is one, the environment in which the expressions are evaluated, and the original continuation to which we send the final expression value.

    public class SequenceCont extends Cont {
        public Iterator<Exp> expIter; // a list of additional expressions
        public Env env;               // environment for expression evaluation
        public Cont cont;             // how to continue the computation

        public SequenceCont (List<Exp> expList, Env env, Cont cont) {
            this.expIter = expList.iterator(); // create an iterator on expList
            this.env = env;
            this.cont = cont;
        }

        public Cont apply(Val val) {
            if (expIter.hasNext()) {
                // discard the current value and evaluate the next expression
                Exp exp = expIter.next();
                return new EvalCont(exp, env, this); // calls apply again!
            }
            return new ValCont(val, cont); // continue with the last Val
        }
    }

The `eval` method in the `SeqExp` class creates a `SequenceCont` object and evaluates the first expression in the sequence using this continuation. As an optimization, if the `expList` is empty, we simply side-step the creation of the `SequenceCont` object and directly arrange to evaluate the expression `exp` with the original continuation.

```java
SeqExp
%
public Cont eval(Env env, Cont cont) {
    List<Exp> expList = seqExps.expList;
    if (expList.size() > 0)
        cont = new SequenceCont(expList, env, cont);
    return new EvalCont(exp, env, cont);
}
%
```
Continuations (continued)

To evaluate a primitive application in the PrimappExp class, we must evaluate the operand expressions and then pass the arguments to the primitive to evaluate, applying the resulting value to the continuation.

We carry out the evaluation of the operand expressions in a manner similar to a sequence expression, except instead of discarding all but the last values, we accumulate them in a List. Once all of the operand expressions have been evaluated, we send them to the primitive for evaluation and apply the result to the original continuation.

The PrimappCont continuation, given on the next slide, has fields for the primitive to apply, an iterator on the operand expression list, the environment in which the operand expressions are evaluated, a list of operand expression values, and the continuation to apply the final primitive value. The apply method in this class determines if there are more operand expressions to evaluate. If so, the expression is evaluated and its value is accumulated by this PrimappCont continuation into the list of operand expression values. If not, the primitive is applied to its arguments and the resulting value is applied to the original continuation.

In the PrimappExp class, the eval method creates an instance of the PrimappCont continuation which then jumps onto the trampoline.

```java
PrimappExp
%
    public Cont eval(Env env, Cont cont) {
        return new PrimappCont(prim, rands.expList, env, cont);
    }
%
```
Continuations (continued)

```java
public class PrimappCont extends Cont {

    public Prim prim; // the primitive to apply
    public Iterator<Exp> expIter; // actual parameter expressions
    public Env env; // Env in which actual params are evaluated
    public Cont cont; // the continuation

    public List<Val> valList; // list of arguments (built here)

    public PrimappCont(Prim prim, List<Exp> expList, Env env, Cont cont) {
        this.prim = prim; this.expIter = expList.iterator();
        this.env = env; this.cont = cont;
        this.valList = new ArrayList<Val>();
    }

    public Cont apply(Val val) {
        valList.add(val);
        return this;
    }

    public Cont apply() {
        if (expIter.hasNext()) {
            Exp exp = expIter.next();
            return exp.eval(env, this);
        }
        Val val = prim.apply(Val.toArray(valList));
        return cont.apply(val);
    }
}
```
Continuations (continued)

Two more expressions require our attention: let expressions and procedure applications. As we have shown, a let expression can (mostly) be converted into a procedure application, so code for both of these should be much the same. The caution here is that call-by-reference parameter passing semantics for procedure application behaves differently from the value semantics for the RHS expressions in a let.

For a let expression, we first evaluate the RHS expressions in the given environment, which produces a list of references to these values. We then bind these references to their corresponding LHS variables, extend the environment with these bindings, and finally evaluate the body expression in this extended environment. This value is then passed on to the continuation in which the let expression appears.

The LetCont continuation, given on the next two slides, does the trick.
public class LetCont extends Cont {

    public List<Token> varList; // LHS identifiers
    public Iterator<Exp> expIter; // RHS expressions
    public Exp exp; // the let body expression
    public Env env; // evaluate expressions in this env
    public Cont cont;

    public List<Ref> refList; // resulting references (initially empty)

    public LetCont(List<Token> varList, List<Exp> expList, Exp exp, Env env, Cont cont) {
        this.varList = varList;
        this.expIter = expList.iterator();
        this.exp = exp;
        this.env = env;
        this.cont = cont;
        this.refList = new ArrayList<Ref>();
    }

    // continued on next slide ...
Continuations (continued)

    // ... continued from previous slide

    public Cont apply(Val val) {
        refList.add(new ValRef(val));
        return this;
    }

    public Cont apply() {
        if (expIter.hasNext()) {
            Exp exp = expIter.next();
            return exp.eval(env, this);
        }
        Bindings bindings = new Bindings(varList, refList);
        Env nenv = env.extendEnvRef(bindings);
        return exp.eval(nenv, cont);
    }
Continuations (continued)

In the LetExp class, its eval method asks its letDecls object to create the appropriate LetCont continuation object from the LetDecls list of identifiers and operand expressions. This continuation, when applied, binds the LHS identifiers to the RHS expression values and uses these bindings to evaluate the let expression body.

```java
public Cont eval(Env env, Cont cont) {
    return letDecls.makeLetCont(exp, env, cont);
}
```

The makeLetCont method in the LetDecls class simply creates the appropriate continuation:

```java
public Cont makeLetCont(Exp exp, Env env, Cont cont) {
    if (varList.size() > 0)
        return new LetCont(varList, expList, exp, env, cont);
    else
        return new EvalCont(exp, env, cont);
}
```
Continuations (continued)

To evaluate a procedure application, we evaluate the procedure expression and pass this on to an AppCont continuation that takes the resulting procedure value (a closure) and applies it to the actual parameter expressions. The eval method in the AppExp class is given here:

```java
AppExp

{public Cont eval(Env env, Cont cont) {
    cont = new AppCont(rands.expList, env, cont);
    return new EvalCont(exp, env, cont);
}
}
```

The AppCont continuation, shown on the next two slides, gets the procVal to evaluate from the above call to eval, evaluates the reference parameters, and passes the reference parameters to the procVal to evaluate the procedure body.
Continuations (continued)

```java
public class AppCont extends Cont {

    public Iterator<Exp> expIter;  // delivers the actual parameter expressions
    public Env env;  // evaluate the params in this environment
    public Cont cont;  // who gets the result

    public Val procVal;  // should evaluate to a ProcVal
    public List<Ref> refList;  // list of references to pass to the procVal

    public AppCont(List<Exp> expList, Env env, Cont cont) {
        this.expIter = expList.iterator();
        this.env = env;
        this.cont = cont;
        refList = new ArrayList<Ref>();
    }

    public Cont apply(Ref ref) {
        refList.add(ref);
        return this; // evaluate more reference parameters (if any)
    }

    public Cont apply(Val val) {
        procVal = val;
        return this; // now start evaluating the reference parameters
    }

    // continued on next slide ...
```
Continuations (continued)

// ... continued from previous slide

public Cont apply() {
    if (expIter.hasNext()) {
        Exp exp = expIter.next();
        return exp.evalRef(env, this); // evaluate the reference
    }
    return procVal.apply(refList, cont); // let the procVal do the rest
}

Notice that the evalRef method produces a reference, not a value, so this continuation’s apply(Ref) method is used to build the list of references.

Once the list of references to actual parameters is built, it is passed to the apply method of the procVal object that binds the references to the formal parameters and evaluates the procedure body in the appropriate extended environment. The code for this apply method is in the ProcVal class:

    public Cont apply(List<Ref> refList, Cont cont) {
        Env env = bodyEnv; // the captured environment
        if (refList.size() > 0) {
            Bindings bindings = new Bindings(formals.varList, refList);
            env = env.extendEnvRef(bindings);
        }
        return new EvalCont(body, env, cont);
    }
Continuations (continued)

In order for everything to work, the `evalRef` method defined in all of the `Exp` classes must return a reference, not a value. Except for a `VarExp`, these references are obtained by evaluating the expression values and turning them into references. This is accomplished by the `EvalRefCont` class:

```java
public class EvalRefCont extends Cont {

    public Exp exp;
    public Env env;
    public Cont cont;

    public EvalRefCont(Exp exp, Env env, Cont cont) {
        this.exp = exp;
        this.env = env;
        this.cont = cont;
    }

    public Cont apply(Val val) {
        return cont.apply(new ValRef(val));
    }

    public Cont apply() {
        return exp.eval(env, this); // this will convert the val to a ref
    }
}
```
Continuations (continued)

The default `evalRef` method, defined in the `Exp` class – and therefore adopted by all of its sub-classes (except when overridden), uses an `EvalRefCont` to turn the value into a reference and to send this reference to the pending `AppCont` continuation:

```java
public Cont evalRef(Env env, Cont cont) {
    return new EvalRefCont(this, env, cont);
}
```

In the `VarExp` class, its `evalRef` method overrides the default `evalRef` method in the `Exp` class. This method looks up the variable in the current environment and passes its corresponding reference directly to the pending `AppCont` continuation:

```java
public Cont evalRef(Env env, Cont cont) {
    return cont.apply(env.applyEnvRef(var.str));
}
```

Remember that the purpose of the `evalRef` method is to implement reference semantics for actual parameters in procedure application. The only place where reference semantics differs from value semantics is when the actual parameter expression is a `VarExp`. 
Continuations (continued)

Finally we handle set expressions. A SetCont object captures the information necessary to modify the value of the LHS variable of a set:

```java
public class SetCont extends Cont {

    public Ref ref;
    public Cont cont;

    public SetCont(Ref ref, Cont cont) {
        this.ref = ref;
        this.cont = cont;
    }

    public Cont apply(Val val) {
        ref.setRef(val); // modify the binding
        return new ValCont(val, cont); // pass the value on
    }
}
```

The eval method in the SetExp class follows:

```java
SetExp
%
public Cont eval(Env env, Cont cont) {
    Ref ref = env.applyEnvRef(var.toString());
    // the SetCont continuation modifies the binding
    cont = new SetCont(ref, cont);
    return new EvalCont(exp, env, cont);
%
}
```
Continuations (continued)

Recall that all of these continuations end up jumping on the trampoline, carrying out the computations iteratively instead of recursively. In particular, a procedure that makes a tail call (i.e., the return value of the procedure is the value of another procedure application) discards its execution context by passing the tail call value to the current continuation instead of keeping its current execution context while evaluating the tail call.

For non-tail calls – for example, the naive implementation of the factorial function – there is no way to avoid building nested execution contexts, since the recursive calls are not in tail position. The basic principle here is that evaluating actual parameters requires creating a nested execution context, but that calling procedures does not.

The even/odd mutual recursion example clearly shows that without continuations, even relative small arguments to even? result in stack overflow. Using continuations, an application such as .even?(100000000) terminates normally. Observe that the mutually recursive calls in the even/odd example are all in tail position.
Exception Handling

Because a continuation holds an execution context, it is possible to save the execution context of an early part of a computation and to return to this context in case something unusual happens later. This gives us the opportunity to implement exception handling: that is, the ability to stop the evaluation of an expression, returning instead to a saved execution context.

We implement exception handling by allowing for named exception handlers that save the current continuation and that otherwise behave like procedures. The exception handlers are installed in a special exception environment that is separate from the normal evaluation environment. When a named exception is thrown – as we describe shortly – the most recent exception handler having that name is looked up in the exception environment, the handler is applied (just like a procedure), and the resulting value is passed to the handler’s saved continuation.

Since the saved continuation jumps onto the trampoline by calling the continuation’s apply method, the program execution continues at the point where the exception handler was installed rather than at the point where the exception was thrown.
Exception Handling (continued)

Here are the new grammar rules that support our exception handling:

\[
\begin{align*}
\text{<exp>:CatchExp} & ::= \text{CATCH } \text{<handlerDecls>} \text{ IN } \text{<exp>}
\end{align*}
\]
\[
\text{CatchExp(HandlerDecls handlerDecls, Exp exp)}
\]

\[
\begin{align*}
\text{<exp>:ThrowExp} & ::= \text{THROW } \text{<VAR>} \text{ LPAREN } \text{<rands>} \text{ RPAREN}
\end{align*}
\]
\[
\text{ThrowExp(Token var, Rands rands)}
\]

\[
\begin{align*}
\text{<handler>} & ::= \text{HANDLER LPAREN } \text{<formals>} \text{ RPAREN } \text{<exp>}
\end{align*}
\]
\[
\text{Handler(Formals formals, Exp exp)}
\]

\[
\begin{align*}
\text{<handlerDecls>} & ::= \text{<VAR>} \text{ EQUALS } \text{<handler>}
\end{align*}
\]
\[
\text{HandlerDecls(List<Token> varList prim, List<Handler> handlerList)}
\]

The CATCH, THROW, and HANDLER tokens are defined in the obvious way.

One difference between exception handlers and ordinary procedures is that when an exception is thrown, the exception handler is found and evaluated in the current exception environment rather than in the current evaluation environment. For example, the body of a top-level procedure can throw an exception whose handler is not visible at the top level but which is defined and invoked in a nested exception environment when the top-level procedure is applied. To throw an exception, all that is required is that the handler must be visible in the chain of exception environments when the exception is thrown. An example of this is on the following slide.
define p = proc() throw eee(5)  
.p() % no binding for eee
catch
  eee = handler(x) add1(x) 
in
  .p() % evaluates to 6
  .p() % still no binding for eee

When .p() is evaluated just after the definition of p, there is no exception binding for the identifier eee. This because the procedure p captures the top-level exception environment (which is empty) and there is no binding for eee in this exception environment.

The catch expression, on the other hand, evaluates to 6. Within the catch expression, the exception handler identifier eee is bound to a handler that returns one plus its actual parameter value. This handler is added to the exception environment of the catch expression, so when the p procedure is called in this catch expression, the throw eee(5) can see the binding for the identifier eee and can thus apply the handler with an actual parameter value of 5. A throw identifier is looked up using dynamic scope rules instead of static scope rules; this behavior is almost identical to macro invocation as compared to procedure invocation.
Exception Handling (continued)

Since we want to maintain static scope rules in ordinary expression evaluation, but allow dynamic scope rules in exception handling, we maintain two environments: a static environment for expression evaluation – usually called $env$ – and a dynamic environment for exception handling – usually called $xenv$. Both $env$ and $xenv$ are passed to eval methods, but the $xenv$ environment is used only when installing handlers (in a catch expression) and when throwing exceptions.

Here is the code for a CatchExp and its associated code for HandlerDecls to add the handler bindings to the exception environment:

```java
CatchExp

public Cont eval(Env env, Env xenv, Cont cont) {
    xenv = handlerDecls.addBindings(env, xenv, cont);
    return new EvalCont(exp, env, xenv, cont);
}

HandlerDecls

public Env addBindings(Env env, Env xenv, Cont cont) {
    List<String> idList = new ArrayList<String>();
    List<Val> valList = new ArrayList<Val>();
    for (Handler h : handlerList)
        valList.add(h.makeHandler(env, xenv, cont));
    Bindings bindings =
        new Bindings(varList, Ref.valsToRefs(valList));
    return xenv.extendEnvRef(bindings);
}
```
A HandlerVal behaves much like a ProcVal, except that a HandlerVal also captures the continuation and the exception environment in which the handler is created. When the handler is applied, the handler body is evaluated in the saved exception environment, with the result passed on to the saved continuation. We can therefore define the HandlerVal class as a subclass of the ProcVal class, with two pieces of additional information: the saved exception environment and the saved continuation.

When an exception is thrown – always by name, and never anonymously – the name is looked up in the exception environment and the handler is applied, returning its value to its saved continuation instead of the continuation in which the exception is thrown:

```java
ThrowExp
%
{
    public Cont eval(Env env, Env xenv, Cont cont) {
        HandlerVal handler = (HandlerVal)xenv.applyEnv(var.toString());
        return handler.apply(rands.expList, env, xenv,
            handler.xenv, handler.cont);
    }
%
}
```

Notice that evaluating the handler’s actual parameters or its body may result in throwing additional exceptions, which can result in a cascade of exception handling, as shown on the following slide. Notice, too, that when the handler body is evaluated, its exception environment is the one in which the catch expression is evaluated. This means if an exception is thrown when evaluating a handler body, its handler is searched for outside of the catch expression handlers named in the expression.

If evaluating an expression results in a throw that refers to a handler name that is not in the current exception environment, the value of the expression is undefined.
Exception Handling (continued)

%% throwing an exception while evaluating
%% a handler’s actual parameter expressions
catch
    h = handler(x) add1(x)
    k = handler(x) *(x,x)
in
    throw h(throw k(3)) % evaluates to 9 ; h is not actually thrown

%% throwing an exception in a handler’s body expression
catch
    h = handler() 5
in
    catch
        h = handler(x) {throw h() ; x}
in
        throw h(21) % evaluates to 5

%% throwing an unbound exception in the handler’s body
catch
    h = handler () throw h()
in
    throw h() % no binding for h (in the handler’s ’throw’)
**Exception Handling** (continued)

The exception environment rules when evaluating expressions, defining handlers, and throwing exceptions are:

- The top-level exception environment is always empty.
- Handlers defined in a `catch` expression are added to the exception environment in which the `catch` expression appears, and this extended exception environment becomes the exception environment in which the `catch` expression body is evaluated.
- A handler defined in a `catch` expression saves the evaluation and exception environments and the execution continuation in which the `catch` expression appears, in addition to the handler’s formal parameters and body expression.
- When evaluating a `throw` expression, the appropriate handler is found by searching the current exception environment in which the `throw` expression appears.
- The exception environment in which the actual parameters of a thrown handler are evaluated is the same as the exception environment in which the exception is thrown.
- The exception environment in which the actual parameters of a procedure or primitive application are evaluated is the same as the exception environment in which the procedure or primitive is applied.
- The exception environment when evaluating a procedure body (when the procedure is applied) is the same as the exception environment in which the procedure is applied.
- When evaluating the thrown handler exception body, the exception environment and continuation are those saved by the handler when the handler was defined in the `catch` expression.
Concurrency

Using trampolining, we repeatedly apply continuations until a value is returned to the `HaltCont` continuation, which stops the trampolining and returns a value. Since each continuation contains complete information about how the expression evaluation is to proceed, it is possible to have multiple threads of concurrent expression evaluation by associating each thread with a continuation that represents the thread’s current execution context.

We add concurrency to our language by defining an expression that is similar to a sequence expression, except that the constituent expressions are all evaluated in parallel.

\[
<\text{exp}> : \text{ConcExp} ::= \{\text{&} \} <\text{threadExps}> \{\&\}
\]

\[
\text{ConcExp}(\text{ThreadExps threadExps})
\]

\[
<\text{threadExps}> ::= <\text{exp}> +\text{&}
\]

\[
\text{ThreadExps(List<Exp> threadExps)}
\]

The `{&}` and `{&}` tokens are ‘{ & ’ and ‘ & }’, respectively. The `+&` token is ‘&’, which is meant to suggest parallel evaluation rather than sequential.

When a concurrent expression is encountered, we build an `EvalCont` continuation for each of the expressions to be evaluated in parallel, and we add these to a queue of continuations constructed in the `ConcurrentCont` class. The `apply` method in this class checks to see if there are any continuations in the queue. If so, a continuation is de-queued and applied, putting the resulting continuation back on the queue. If applying this continuation throws an exception, this means that the corresponding expression evaluation is done, and its value is silently ignored.

The value of a continuation expression is always zero. The expressions to be evaluated in parallel serve only to create side-effects.
Concurrency (continued)

To make concurrency work at the level of our defined language, we simply rely on trampolining to apply each concurrent continuation, in whatever order these continuations appear in the queue. This is not truly parallel – we could have used Java threads for this – but it does simulate concurrency.

The `ConcurrentCont` class appears here:

```java
public class ConcurrentCont extends Cont {

    public Queue<Cont> queue; // queue of continuations to apply
    public Cont cont; // what to do at the end

    public ConcurrentCont (List<Exp> expList, Env env, Cont cont) {
        queue = new LinkedList<Cont>();
        for (Exp exp : expList)
            queue.add(new EvalCont(exp, env, HaltCont.halt));
        this.cont = cont;
    }

    continued on next slide ...
```
Concurrent evaluation of the constituent expressions is only used to carry out side-effects. Their expression values are ignored.

Observe that `thread.apply()` will throw an exception if the `thread` object is a `HaltCont`, meaning that the corresponding concurrent expression evaluation is complete. In this case, the expression value is discarded – the `ContException` object is ignored – and nothing is put back on the queue. If `thread.apply()` does not throw an exception, its result is put back on the queue for later application.
Concurrency (continued)

The result of evaluating concurrent expressions must affect some sort of shared environment, possibly the top-level environment. Here’s an example of such a situation:

```
define count = 0
define d = proc(t)
    if t
        then {set count=add1(count) ; .d(sub1(t))}
    else 0
{& .d(1000) & .d(10000) & .d(100) &} % evaluate these concurrently
count %% => ???
```

In this case, the value of count ends up being 10000 and not 11100 as one would expect.
Concurrency (continued)

The problem here is an example of the “simultaneous update problem”, also called a “race condition”. Evaluating an expression like set count = add1(count) can result in the creation of multiple continuations – several, for example, just to evaluate add1(count) – and when count is in the process of being modified in one thread, there may be other threads that are in the process of attempting to modify it as well, with unpredictable results.

Since concurrent expressions must do their work through side-effects, we need a way to guard against race conditions. We do this through an atomic expression. The concrete and abstract syntax of such an expression is given here:

<exp>:AtomicExp ::= ATOMIC <exp>

AtomicExp(Exp exp)

When evaluating an atomic expression, we circumvent the evaluation of the queued expression by evaluating the expression directly – using a non-threaded trampoline. Once the value has been determined, we pass it on to the pending continuation so the threading can continue. Observe that during the evaluation of an atomic expression, the threaded trampoline stops processing queued continuations.

AtomicExp

%%{
   public Cont eval(Env env, Cont cont) {
      Val val = exp.eval(env); // don’t thread on me
      return cont.apply(val);
   }
}

It’s harmless to evaluate an atomic expression in a non-threaded environment. However, an atomic expression must complete before its value can be applied to the next continuation, so deeply nested atomic expressions can lead to stack overflow. Using atomic expressions should be done sparingly.
The race condition in the previous example can now be solved by making the modification of `count` atomic:

```
define count = 0
define d = proc(t)
    if t
        then {atomic set count=add1(count) ; .d(sub1(t))}
        else 0
    {& .d(1000) & .d(10000) & .d(100) &}
count %% => 11100
```

Of course, threads can start other threads, limited only by the memory limits of the underlying machine.

```
define count=1
define d = proc(t)
    if t
        then
            let
                t1 = sub1(t)
            in
                { {& .d(t1) & .d(t1) &} ; atomic set count=add1(count) }
        else count
    .d(16) %% => 65536
```