Continuations

All of our non-side-effecting languages (`V1` through `V6`) have relied on recursion to implement iterative behavior. See, for example, the procedure `updown` used to define the `pos?` procedure in one of our assignments. In our Java language 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, its underlying Java `eval` methods call themselves recursively, and it is possible that stack frames can build up to exhaust available stack memory. Even relatively small programs can result in stack overflow.

Continuations

An `eval` method call is intended to carry out some computation – one that may be used, for example, to evaluate an actual parameter expression in a procedure application or a test expression in an `if` expression. A stack frame gets created implicitly by the Java Runtime Environment (JRE) upon each `eval` method 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 overall compution should continue once the current computation is finished.

Continuations (continued)

We start with language `REF`, a language with `set` and call-by-reference parameter passing. In this language, the `eval` method for expressions has a single `env` parameter that gives the environment in which the expression is evaluated. In the `REFCONT` language, adding explicit execution context requires passing another parameter to the `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 `Cont` class, with three subclasses: `ACont`, `VCont`, and `RCont`.

Continuations (continued)

An `ACont` continuation has an `apply` method that takes no parameters. A `VCont` continuation has an `apply` method that takes a single `Val` parameter, and an `RCont` continuation has an `apply` method that takes a single `Ref` parameter. All of these continuations return an instance of `ACont`. Here are the method signatures for each:

```
ACont: public ACont apply();
VCont: public ACont apply(Val val);
RCont: public ACont apply(Ref val);
```

The essential purpose of the `apply` method for an `ACont` continuation is to carry out some action that represents "what to do now". Its return value, an instance of `ACont`, says "what to do next".

For a `VCont` continuation, the purpose of the `apply` method is to carry out some action that represents "what to do now with the value `val`". Its return value, again an instance of `ACont`, says "what to do next". Similar remarks apply to a `RCont` continuation. You will see that the `RCont` continuations are used only to implement call-by-reference parameter passing.

The `apply` method in the `ACont` class is the fundamental action that starts off evaluation. This method carries out some action and returns another `ACont` continuation whose `apply` method is invoked, and so on. This proceeds until an `apply` method stops the evaluation by throwing an exception – either a runtime exception indicating an error, or a special `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 `eval` method in our continuation-passing language, declared in the `Exp` class:

```
public abstract ACont eval(Env env, VCont vcont);
```

This method is intended to work as follows:

- evaluate the expression in the environment `env`, yielding a value which we call `val`
- call `vcont.apply(val)`, yielding an `ACont` instance which we call `acont`
- call `acont.apply()` to continue evaluation

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 the current execution context. Languages that implement tail call elimination (look this up!) do this explicitly: see Lisp, Scheme, Haskell, and Scala.

If our implementation (defining) language supported tail call elimination, we wouldn't need to worry about this. In Java it's not as simple, since Java does not do tail call elimination – but we can accomplish essentially the same thing through a technique called *trampolining*. This technique de-couples the recursive calls to `apply` by looping instead. The next slide shows how this works:

Continuations (continued)

```
public abstract class ACont {

    public Val trampoline() {
        ACont acont = this;
        while(true)
            try {
                acont = acont.apply();
            } catch (ContException e) {
                return e.val;
            }
    }

}
```

Continuations (continued)

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

```
acont.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. Since the purpose of expression evaluation is to produce a value, this exception needs to have a `Val` field that can be used to return a value to the top-level REP loop.

We choose a special "halt continuation" `HaltCont` to do this. `HaltCont` extends the `VCont` class, so its `apply` method takes a `Val` parameter. The `apply` method in this class – the one that actually jumps off the trampoline – is now trivial.

```
public class HaltCont extends VCont {

    public ACont 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 a value continuation. When an `EvalCont` continuation is applied, the expression is evaluated in the given environment and its value is passed to the saved value continuation.

```
public class EvalCont extends ACont {

    public Exp exp;
    public Env env;
    public VCont cont;

    public EvalCont(Exp exp, Env env, VCont vcont) {
        this.exp = exp;
        this.env = env;
        this.vcont = vcont;
    }

    public ACont apply() {
        return exp.eval(env, vcont);
    }
}
```

The continuation field `vcont` has an `apply(Val)` method that receives the expression value and that returns an `ACont` continuation that dictates "what to do next".

Continuations (continued)

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

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

As described on slide 7.7, 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.

Continuations (continued)

For certain expressions (such as `LitExp` and `VarExp`), the expression value could be determined with little additional work, so this value could be sent directly to its `VCont` continuation, like this:

```
return vcont.apply(...)
```

However, this results in building a stack frame to call the `apply` method, contrary to our objective of using continuations to avoid building stack frames.

Continuations (continued)

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 int the `ValCont` class:

```
public class ValCont extends ACont {

    public Val val;
    public VCont vcont;

    public ValCont(Val val, VCont vcont) {
        this.val = val;
        this.vcont = vcont;
    }
    public ACont apply() {
        return vcont.apply(val);
    }
}
```

It may appear that this `apply` method just postpones the recursive call to `vcont.apply(val)`, but since the `apply` method in the `ValCont` instance is being handled by the trampoline, this de-couples the direct recursion, replacing it with iteration.

Continuations (continued)

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

```
public abstract ACont eval(Env env, VCont vcont);
```

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` value and return a `ValCont` continuation that passes the value to the `vcont` continuation. Similarly, for the `VarExp` class, look up the variable in the given environment to get the value it is bound to, and return a `ValCont` continuation that passes this value to the `vcont` continuation. Here is the code for both:

```
LitExp
%%%
    public ACont eval(Env env, VCont vcont) {
        return new ValCont(new IntVal(lit.toString()), vcont);
    }
%%%

VarExp
%%%
    public ACont eval(Env env, VCont vcont) {
        return new ValCont(env.applyEnv(var.toString()), vcont);
    }
%%%
```

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. Recall that `addBindings` returns an environment that extends the given environment by binding the LHS variables to (references to) their corresponding RHS `ProcVals`. The `eval` method then returns a `EvalCont` continuation that evaluates the body of the `letrec` in this extended environment and passes its value to the `vcont` continuation. Here is the code for `eval` in the `LetrecExp` class:

```
LetrecExp
%%%
    public ACont eval(Env env, VCont vcont) {
        Env nenv = letrecDecls.addBindings(env);
        return new EvalCont(exp, nenv, vcont);
    }
%%%
```

Notice that we don't call the `vcont apply` method directly here, since we expect that the `EvalCont` continuation will ultimately do so when it jumps onto the trampoline.

Continuations (continued)

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

```
ProcExp
%%%
    public ACont eval(Env env, VCont vcont) {
        return new ValCont(proc.makeClosure(env), vcont);
    }
%%%
```

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 (it's a `VCont`!) to determine which of these two expressions must be evaluated next. The resulting value is then passed on to the original value continuation. The `IfCont` class appears as follows:

```
public class IfCont extends VCont {

    public Exp trueExp;
    public Exp falseExp;
    public Env env;
    public VCont vcont;

    public IfCont(Exp trueExp, Exp falseExp, Env env, VCont vcont) {
        this.trueExp = trueExp;
        this.falseExp = falseExp;
        this.env = env;
        this.vcont = vcont;
    }

// continued on next slide ...
```

Continuations (continued)


```
// ... continued from previous slide

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

In the `IfExp` class, the `eval` method creates an `EvalCont` method that evaluates the test expression and passes its value to a suitably constructed `IfCont` value continuation.

```
IfExp
%%%
     public ACont eval(Env env, VCont vcont) {
          return new EvalCont(testExp,
                              env,
                              new IfCont(trueExp,falseExp,env,vcont));
     }
%%%
```

Continuations (continued)

To evaluate a `SeqExp`, we evaluate the first expression and save its value. If there are more expressions in the list, we evaluate each of them in turn, keeping only the value of the last expression and passing it on to the saved continuation. We create a `SequenceCont` continuation that has fields for the initial expression, 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.

The `SequenceCont` class is on the next slide.

```
SequenceCont
%%%
import java.util.*;

// used with SeqExps
public class SequenceCont extends VCont {

    public Iterator<Exp> expIter; // iterate over the expList
    public Env env;               // the environment
    public VCont vcont;           // apply this to the last sequence value

    public SequenceCont (List<Exp> expList, Env env, VCont vcont) {
        this.env = env;
        this.vcont = vcont;
        this.expIter = expList.iterator();
    }

    public ACont apply(Val val) {
        if (expIter.hasNext()) {
            Exp exp = expIter.next();
            return new EvalCont(exp, env, this);
        }
        return new ValCont(val, vcont); // pass the last Val to vcont
    }

    public String toString() {
        return "SequenceCont";
    }
}
%%%
```

Continuations (continued)

The `eval` method in the `SeqExp` class creates a continuation to evaluate the first expression in the sequence, which passes this value to a `SequenceCont` that determines if more expressions need to be evaluated. As an optimization, if the `expList` is empty, we simply side-step the creation of the `SequenceCont` object and directly arrange to evaluate the first expression `exp` using the original continuation.

```
SeqExp
%%%
    public ACont eval(Env env, VCont vcont) {
        List<Exp> expList = seqExps.expList;
        if (expList.size() > 0)
            return new EvalCont(exp,
                                env,
                                new SequenceCont(expList,env,vcont));
        // if only one expression, just evaluate it
        return new EvalCont(exp, env, vcont);
    }
%%%
```

Continuations (continued)

To evaluate a primitive application in the `PrimappExp` class, we must evaluate the operand expressions and then send the arguments to the primitive to evaluate, passing the resulting value to the saved continuation.

We can use the `evalRands` method defined in the `Rands` class to do the work of evaluating each of the operand expressions. This method returns a `List` of values. Since the `apply` method for each primitive takes an array of values, we need to convert the `List` into an array before calling `apply`.

```
PrimappExp
%%%
    public ACont eval(Env env, VCont vcont) {
        List<Val> valList = rands.evalRands(env);
        int size = valList.size();
        Val [] valArray = valList.toArray(new Val[size]);
        Val val = prim.apply(valArray);
        return new ValCont(val, vcont);
    }
%%%
```

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`.

In the `LetExp` class, its `eval` method asks its `letDecls` object to extend its environment by binding all of the LHS variables to (references to) their RHS values. This extended environment is used to return an `EvalCont` object that evaluates the body of the `let` in this extended environment and pass its value on to the `vcont` continuation.

```
LetExp
%%%
    public ACont eval(Env env, VCont vcont) {
        env = letDecls.addBindings(env);
        return new EvalCont(exp, env, vcont);
    }
%%%
```

The addBindings method in the LetDecls class evaluates the RHS expressions using the evalRands method in a suitably constructed Rands object. This method returns a List of Vals. Since our environments bind strings to references, we must convert this list of values to a list of references using the valsToRefs method in the Ref class. These values are then bound to the variables in varList, obtaining the environment in which the body of the let is to be evaluated. As an optimization, if there are no variable bindings in the let, addBindings simply returns the original environment.

```
LetDecls
%%%
    public Env addBindings(Env env) {
        if (varList.size() > 0) {
            Rands rands = new Rands(expList);
            List<Val> valList = rands.evalRands(env);
            Bindings bindings =
                new Bindings(varList, Ref.valsToRefs(valList));
            env = env.extendEnvRef(bindings);
        }
        return env;
    }
%%%
```

Continuations (continued)

To evaluate a procedure application, we need to evaluate the procedure expression (it must evaluate to a `ProcVal`), the actual parameter expressions using reference semantics, bind the actual parameter references to their formal parameter names, use these bindings to extend the environment captured by the procedure, and finally evaluate the body of the procedure in this extended environment. The `eval` method in the `AppExp` class is given here:

```
AppExp
%%%
public Cont eval(Env env, Cont cont) {
    return new EvalCont(exp,
                        env,
                        new AppCont(rands, env, vcont));
}
%%%
```

The `AppCont` continuation, shown on the next slide, gets an expression that must evaluate to a `ProcVal`, evaluates the reference parameters, and passes the reference parameters along with the `vcont` continuation to the `ProcVal` to evaluate the procedure body and pass its value along for further processing.

Continuations (continued)


```
public class AppCont extends VCont {

    public Rands rands; // the actual parameter expressions
    public Env env;     // evaluate the params in this env
    public VCont vcont; // who gets the result

    public AppCont(Rands rands, Env env, VCont vcont) {
        this.rands = rands;
        this.env = env;
        this.vcont = vcont;
    }

    public ACont apply(Val val) {
        ProcVal procVal = val.procVal();
        List<Ref> refList = rands.evalRandsRef(env);
        return procVal.apply(refList, vcont);
    }

    public String toString() {
        return "AppCont";
    }

}
```

Continuations (continued)

In the `apply(Val val)` method in the AppCont class, `val` must evaluate to a `ProcVal`. Notice that the `evalRandsRef` method produces a list of references, not values. Once the list of references to actual parameters is built, it is passed to the `apply` method of the `procVal` object, along with the `vcont` continution. This `apply` method binds the references to the procedure's formal parameters, evaluates the procedure body in the appropriate extended environment, and passes this value to the `vcont` continuation for further disposition.

Continuations (continued)

The code for this `apply` method is in the `ProcVal` class:

```
public ACont apply(List<Ref> refList, VCont vcont) {
    Env env = this.env; // local copy of the captured env
    List<Token> varList = formals.varList;
    if (refList.size() != varList.size())
        throw new RuntimeException(
            "formal/actual parameter mismatch"
        );
    if (varList.size() > 0) {
        Bindings bindings = new Bindings(varList, refList);
        env = env.extendEnvRef(bindings);
    }
    return new EvalCont(body, env, vcont);
}
```

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`:

```
public class SetCont extends VCont {

    public Ref ref;
    public VCont vcont;

    public SetCont(Ref ref, VCont vcont) {
        this.ref = ref;
        this.vcont = vcont;
    }

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

Continuations (continued)

The `eval` method in the `SetExp` class follows:

```
public ACont eval(Env env, VCont vcont) {
    // don't actually modify the binding yet
    Ref ref = env.applyEnvRef(var.toString());
    return new EvalCont(exp, env, new SetCont(ref, vcont));
}
```

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 own execution context by passing the tail call value to the current continuation instead of saving its current execution context while evaluating the tail call. Remember that continuations represent *what should happen now and in the future*, not *what has happened in the past*.

For non-tail calls – for example, the naive recursive 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:

```
<exp>:CatchExp  ::= CATCH <handlerDecls> IN <exp>
                 CatchExp(HandlerDecls handlerDecls, Exp exp)
<exp>:ThrowExp  ::= THROW <VAR> LPAREN <rands> RPAREN
                 ThrowExp(Token var, Rands rands)
<handler>       ::= HANDLER LPAREN <formals> RPAREN <exp>
                 Handler(Formals formals, Exp exp)
<handlerDecls>  **= <VAR> EQUALS <handler>
                 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.

Exception Handling (continued)

Consider the following example:

```
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 dyanmic 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`. The code for `HandlerDecls` is on the next page.

```
CatchExp
%%%
    public ACont eval(Env env, Env xenv, VCont vcont) {
        xenv = handlerDecls.addBindings(env, xenv, vcont);
        return new EvalCont(exp, env, xenv, vcont);
    }
%%%
```

Exception Handling (continued)


```
HandlerDecls
%%%
    public Env addBindings(Env env, Env xenv, VCont vcont) {
        List<String> idList = new ArrayList<String>();
        List<Val> valList = new ArrayList<Val>();
        for (Handler h : handlerList)
            valList.add(h.makeHandler(env, xenv, vcont));
        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 using 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.

Exception Handling (continued)

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:

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

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.

continued on next slide ...

Exception Handling (continued)

... continued from previous slide

- 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.

Observe that we have used a `Rands` object to evaluate a collection of expressions, returning a list of values or references. Such lists are used to pass arguments to primitive operators or procedures or to bind values to the LHS variables in `let` expressions.

Assuming that the order of evaluation of expressions in a `Rands` object does not matter, we can carry out these evaluations *concurrently* (or *in parallel*). With suitable hardware to support real concurrency, this can result in run-time improvement.

Concurrency

Up to now, we have evaluated the expressions in a `Rands` object in the order in which they appear in the expression list, and we have built the corresponding list of values (or references) to appear in the same order.

In the presence of concurrency, the order in which the expression evaluations complete is almost certainly not the same as the order in which the expressions appear in the expression list.

To maintain the value order, we create an array of value slots, with one slot per expression. For each expression in the expression list, we create a continuation that knows about the expression, its evaluation environment, and the slot where the expression value should go. We can then dispatch all of these continuations to a mechanism that evaluates them in parallel (or at least simulate parallel evaluation); once each expression evaluation completes, its continuation deposits the resulting value in the corresponding value slot.

Concurrency (continued)

In general, we want to define a mechanism that can carry out a simulation of parallel execution of multiple continuations. First, we build a queue that holds all of the continuations to be executed in parallel. Then we create a "wrapper" continuation that takes a single continuation from the queue, calls the dequeued continuation's `apply()` method, and puts the result back on the end of the queue. Once all of the continuations have completed, the wrapper continuation returns the next continuation step in the evaluation (such as primitive application, procedure application, or `let` body evaluation). The wrapper continuation uses the trampoline to carry out each of its dequque steps. A `ConcurrentCont` class, shown on the next slide, serves this purpose.

This approach does not achieve true parallelism, since we are still applying the continuation steps one at a time (using the trampoline), but in the presence of suitable hardware, it would not be difficult to dispatch the application of the queued continuations to separate threads.

Concurrency (continued)


```
ConcurrentCont
%%%
import java.util.*;

public class ConcurrentCont extends ACont {

    public Queue<ACont> queue; // apply these in parallel
    public ACont acont; // what to do when the queue is empty

    public ConcurrentCont(Queue<ACont> queue, ACont acont) {
        this.queue = queue;
        this.acont = acont;
    }

    public ACont apply() {
        ACont thread = queue.poll();
        if (thread == null)
            return acont;
        try {
            thread = thread.apply();
            queue.add(thread);
        } catch (NullContException) {
        }
        return this; // bounce me!
    }
}
%%%
```

Concurrency (continued)

When a queued expression evaluation continuation completes, its value must be deposited in the proper place in the array of values. The following continuation represents what to do with the expression value once the evaluation completes:

```
public class ValIndexCont extends VCont {

    Val [] valArray; // an array of values
    int index;       // where to put the result

    public ValIndexCont(Val [] valArray, int index) {
        this.valArray = valArray;
        this.index = index;
    }

    public ACont apply(Val val) {
        valArray[index] = val;
        throw new NullContException(); // all done
    }

}
```

Before creating a continuation to carry out concurrent evaluation of the list of expressions in a `Rands` object, we need to create an array that holds the resulting values. The result of evaluating concurrent expressions must affect some sort of shared environment, possibly the top-level environmment or in `let` variable bindings. Here's an example of such a situation:

```
let
  count = 0
in
  letrec
    d = proc(t)
      if t
      then {set count=add1(count) ; .d(sub1(t))}
      else 0
  in
    let % the RHS expressions are evaluated in parallel
      _ = .d(1000)
      _ = .d(10000)
      _ = .d(100)
    in
      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)
```

Concurrency (continued)

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.

Concurrency (continued)

The race condition in the previous example can now be solved by making the modification of `count` atomic:

```
let
  count = 0
in
  letrec
    d = proc(t)
      if t
      then {atomic set count=add1(count) ; .d(sub1(t))}
      else 0
  in
    let
      _ = .d(1000)
      _ = .d(10000)
      _ = .d(100)
    in
      count
```

This expression evaluates to 11100, as expected.

Concurrency (continued)

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

```
let
  count = 1
in
  letrec
    par = proc(f, g) atomic set count=add1(count)
    d = proc(t)
      if t
      then
        let
          t1 = sub1(t)
        in
          .par(.d(t1), .d(t1)) % evaluate actuals in parallel
      else count
  in
    .d(16) %% => 65536
```

See what happens if you omit the `atomic` modifier in the definition of `par`.