Continuations

All of our non-side-effecting languages (`V1` through `V6`) have relie
to implement iterative behavior. See, for example, the mutually
cedures `even?` and `odd?` shown in Slides 3.89 and 3.93. In our
implementations, any `eval` call requires setting up a Java stack fra
arguments to `eval`. If evaluating the arguments requires additional
additional stack frames are required. So if a procedure calls itself
underlying Java `eval` methods call themselves recursively, and it
stack frames can build up to exhaust available stack memory. Even
programs can result in stack overflow.

Continuations

An `eval` method call is intended to carry out some computation -
be used, for example, to evaluate an actual parameter expression
application or a test expression in an `if` expression. A stack fran
implicitly by the Java Runtime Environment (JRE) upon each `eva`
This stack frame consists of information including the method argur
find non-local variables (*i.e.*, an environment), and a "return address
where the JRE should execute next when the method finishes. This i
be considered as an "execution context". Once the method finishes,
discarded (popped off the stack) and the execution context of the ca

One way to avoid stack overflow is to maintain execution context
stead of using the JRE stack to hold this information, we pass alon
context to the `eval` method that is used by the method to determi
be executed next when the method finishes. Such an execution con
continuation. The idea is that a continuation determines how the ove
should continue once the current computation is finished.

Continuations (continued)

We start with language `REF`, a language with `set` and call-by-refe
ter passing. In this language, the `eval` method for expressions ha
parameter that gives the environment in which the expression is ev
`REFCONT` language, adding explicit execution context requires pass
rameter to the `eval` method, namely a continuation. The purpose o
tion is to receive the value of the expression and to determine what t

We implement continuations as members of the `Cont` class, with th
`ACont`, `VCont`, and `RCont`.

Continuations (continued)

An `ACont` continuation has an `apply` method that takes no parame continuation has an `apply` method that takes a single `Val` para `RCont` continuation has an `apply` method that takes a single `Ref` of these continuations return an instance of `ACont`. Here are the me 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` continua out some action that represents "what to do now". Its return value, `ACont`, says "what to do next".

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

Continuations (continued)

The `apply` method in the `ACont` class is the fundamental actio evaluation. This method carries out some action and returns another uation whose `apply` method is invoked, and so on. This proceeds u method stops the evaluation by throwing an exception – either a run indicating an error, or a special `ContException` indicating that evaluation has finished.

In the absence of an exception, applying a continuation involves cre ecution context that continues the expression evaluation.

Here is the signature of the abstract `eval` method in our continuatio guage, declared in the `Exp` class:

```
public abstract ACont eval(Env env, VCont v
```

This method is intended to work as follows:

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

Continuations (continued)

It may appear that we have simply added recursive calls to `apply` recursive calls to `eval`. The difference is that once we call `app` need to preserve the current execution context. Languages that imp elimination (look this up!) do this explicitly: see Lisp, Scheme, Has

If our implementation language supported tail call elimination, we to worry about this. In Java it's not as simple, since Java does r elimination – but we can accomplish essentially the same thing throu called *trampolining*. This technique de-couples the recursive calls 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 contir
and then jumping onto the trampoline:

```
acont.trampoline();
```

In this code, the trampoline loops until one of the calls to ap
ContException. In order for expression evaluation to termin
tinuation must therefore throw a ContException that jumps off
Since the purpose of expression evaluation is to produce a value,
needs to have a Val field that can be used to return a value to the
loop.

We choose a special "halt continuation" HaltCont to do this. H
tends the VCont class, so its apply method takes a Val paramete
method in this class – the one that actually jumps off the trampoline

```
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 eval

Continuations (continued)

One useful continuation, EvalCont, has fields for an expression, an environ
continuation. When an EvalCont continuation is applied, the expression is eval
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 exp
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
as follows:

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

As described on Slide 8.7, The HaltCont continuation created here
behavior of jumping off the trampoline by throwing a ContExcep
the top-level evaluation is complete and its value is passed to the ap
the HaltCont object, the trampoline loop stops and this value is re

Continuations (continued)

For certain expressions (such as LitExp and VarExp) whose val
termined directly, the value could be sent to its VCont continuation

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

However, this would result in building a stack frame to call the a
contrary to our objective of using continuations (and the trampoline)
ing stack frames.

Continuations (continued)

To get around this, we define a special `ValCont` continuation tha
direct call to `apply` with a value parameter and that passes the respo
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 rec
`vcont.apply(val)`, but since the `apply` method in the `Val`
is being handled by the trampoline, this de-couples the direct recur
it with iteration.

Continuations (continued)

As noted before, the continuation-based `eval` method in the `Ex`
following signature:

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

Let's consider the easiest subclasses of `Exp`, namely `LitExp` and
`eval` code for the `LitExp` class is simple: convert the literal to an
and return a `ValCont` continuation that passes the value to the `vc`
tion. Similarly, for the `VarExp` class, look up the variable in the give
to get the value it is bound to, and return a `ValCont` continuation
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()
    }
%%%

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

Continuations (continued)

A `letrec` simply creates a new environment with bindings o `ProcVals`. Moreover, evaluating a `proc` (don't confuse this w `proc`) requires no more than gathering together the formal paramet cedure body expression, and the captured environment. Thus the which we evaluate the `letrec` body is obtained by calling the `a` method in the `LetrecDecls` class, unchanged from the `REF` lan that `addBindings` returns an environment that extends the given e binding the LHS variables to (references to) their corresponding RH The `eval` method then returns a `EvalCont` continuation that eva of the `letrec` in this extended environment and passes its value 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 pect that the `EvalCont` continuation will ultimately do so when it trampoline.

Continuations (continued)

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

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

Continuations (continued)

An `if` expression requires that the test expression be evaluated be
exactly one of `trueExp` or `falseExp`. So after evaluating the test
`IfCont` continuation uses the result of the test (it's a `VCont`!) to de
of these two expressions must be evaluated next. The resulting value
on to the original value continuation. The `IfCont` class appears as

```
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
        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` metho
the test expression and passes its value to a suitably constructed
continuation.

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

Continuations (continued)

To evaluate a `SeqExp`, we evaluate the first expression and save its
are more expressions in the list, we evaluate each of them in turn, k
value of the last expression and passing it on to the saved continuati
`SequenceCont` continuation that has fields for the initial express
that produces the next expression in the sequence if there is one, t
in which the expressions are evaluated, and the original continuati
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 se

    public SequenceCont (List<Exp> expList, Env env, VCont vcc
        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 t
    }

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

Continuations (continued)

The `eval` method in the `SeqExp` class creates a continuation to ev expression in the sequence, which passes this value to a `Seque` determines if more expressions need to be evaluated. As an optir `expList` is empty, we simply side-step the creation of the `Seque` ject and directly arrange to evaluate the first expression `exp` usi 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(exp
        // 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 the operand expressions and then send the arguments to the primit passing the resulting value to the saved continuation.

We can use the `evalRands` method defined in the `Rands` class to evaluating each of the operand expressions. This method returns a `L` Since the `apply` method for each primitive takes an array of valu 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[s
        Val val = prim.apply(valArray);
        return new ValCont(val, vcont);
    }
%%%
```

Continuations (continued)

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

In the LetExp class, its eval method asks its letDecls object t vironment by binding all of the LHS variables to (references to) the This extended environment is used to return an EvalCont object th body of the let in this extended environment and pass its value or continuation.

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

Continuations (continued)

The addBindings method in the LetDecls class evaluates th sions using the evalRands method in a suitably constructed Ranc method returns a List of Vals. Since our environments bind string we must convert this list of values to a list of references using the method in the Ref class. These values are then bound to the variable obtaining the environment in which the body of the let is to be ev optimization, if there are no variable bindings in the let, addBin 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.valsToRef
            env = env.extendEnvRef(bindings);
        }
        return env;
    }
%%%
```

Continuations (continued)

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

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

The `AppCont` continuation, shown on the next slide, gets an expre
evaluate to a `ProcVal`, evaluates the reference parameters, and pa
ence parameters along with the `vcont` continuation to the `ProcV`
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` mu `ProcVal`. Notice that the `evalRandsRef` method produces a lis not values. Once the list of references to actual parameters is built the `apply` method of the `procVal` object, along with the `vcor` This `apply` method binds the references to the procedure's forn evaluates the procedure body in the appropriate extended environm 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 vcor
    Env env = this.env; // local copy of the cap
    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
        env = env.extendEnvRef(bindings);
    }
    return new EvalCont(body, env, vcont);
}
```

Continuations (continued)

Finally we handle `set` expressions. A `SetCont` object captures t
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 tl
        return new ValCont(val, vcont); // pass the
    }
}
```

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(re
}
```

Continuations (continued)

Recall that all of these continuations end up jumping on the tramp
out the computations iteratively instead of recursively. In particul
that makes a tail call (*i.e.*, the return value of the procedure is the v
procedure application) discards its own execution context by passi
value to the current continuation instead of saving its current exe
while evaluating the tail call. Remember that continuations represe
happen now and in the future, not *what has happened in the past*.

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

The even/odd mutual recursion example clearly shows that, without
relatively small arguments to `even?` result in stack overflow. Using
an application such as `.even?(100000000)` terminates normall
the mutually recursive calls in the even/odd example are all in tail p

Exception Handling

Because a continuation holds an execution context, it is possible
ecution context of an early part of a computation and to return to
case something unusual happens later. This gives us the opportunit
exception handling: that is, the ability to stop the evaluation of an
turning instead to a saved execution context.

We implement exception handling by allowing for named *exceptio*
save the current continuation and that otherwise behave like proce
ception handlers are installed in a special *exception environment*
from the normal evaluation environment. When a named exception
we describe shortly – the most recent exception handler having that
up in the exception environment, the handler is applied (just like a p
the resulting value is passed to the handler's saved continuation.

Since the saved continuation jumps onto the trampoline by callin
ation's `apply` method, the program execution continues at the p
exception handler was installed rather than at the point where the
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<Har
```

The `CATCH`, `THROW`, and `HANDLER` tokens are defined in the obvic

One difference between exception handlers and ordinary procedur an exception is thrown, the exception handler is found and evaluate exception environment rather than in the current evaluation envi example, the body of a top-level procedure can throw an exception is not visible at the top level but which is defined and invoked in a n environment when the top-level procedure is applied. To throw all that is required is that the handler must be visible in the chai 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 exc for the identifier `eee`. This because the procedure `p` captures the t tion environment (which is empty) and there is no binding for `eee` i environment.

The `catch` expression, on the other hand, evaluates to 6. Within pression, the exception handler identifier `eee` is bound to a handler plus its actual parameter value. This handler is added to the exceptic of the `catch` expression, so when the `p` procedure is called in this sion, the `throw eee(5)` can see the binding for the identifier ee apply the handler with an actual parameter value of 5. A `throw` ider up using dyanmic scope rules instead of static scope rules; this beh identical to macro invocation as compared to procedure invocation.

Exception Handling (continued)

Since we want to maintain static scope rules in ordinary expression
allow dynamic scope rules in exception handling, we maintain two
a static environment for expression evaluation – usually called `er`
namic environment for exception handling – usually called `xenv`.
`xenv` are passed to `eval` methods, but the `xenv` environment is u
installing handlers (in a `catch` expression) and when throwing exc

Here is the code for a `CatchExp`. The code for `HandlerDecls`
page.

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

Exception Handling (continued)

```
HandlerDecls
%%%
    public Env addBindings(Env env, Env xenv, VCont
        List<String> idList = new ArrayList<String>
        List<Val> valList = new ArrayList<Val>();
        for (Handler h : handlerList)
            valList.add(h.makeHandler(env, xenv, vco
        Bindings bindings =
            new Bindings(varList, Ref.valsToRefs(val
        return xenv.extendEnvRef(bindings);
    }
%%%
```

A `HandlerVal` behaves much like a `ProcVal`, except that a `Han`
captures the continuation and the exception environment in which
created. When the handler is applied, the handler body is evaluated u
exception environment, with the result passed on to the saved con
can therefore define the `HandlerVal` class as a subclass of the `P`
with two pieces of additional information: the saved exception envir
saved continuation.

Exception Handling (continued)

When an exception is thrown – always by name, and never anonymou
is looked up in the exception environment and the handler is applie
value to its saved continuation instead of the continuation in which t
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
throwing additional exceptions, which can result in a cascade of
dling, as shown on the following slide. Notice, too, that when the h
evaluated, its exception environment is the one in which the catc
evaluated. This means if an exception is thrown when evaluating a
its handler is searched for outside of the catch expression handler
expression. If evaluating an expression results in a throw that refe
name that is not in the current exception environment, the value of
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

%% throwing an exception in a handler's body express
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 bo
catch
    h = handler () throw h()
in
    throw h() % no binding for h (in the handler's
```

Exception Handling (continued)

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

- The top-level exception environment is always empty.
- Handlers defined in a `catch` expression are added to the exceptic
in which the `catch` expression appears, and this extended exc
ment becomes the exception environment in which the `catch` e
is evaluated.
- A handler defined in a `catch` expression saves the evaluation
environments and the execution continuation in which the `cat`
appears, in addition to the handler's formal parameters and body
- When evaluating a `throw` expression, the appropriate handl
searching the current exception environment in which the `thr`
appears.

continued on next slide ...

Exception Handling (continued)

... continued from previous slide

- The exception environment in which the actual parameters of a
are evaluated is the same as the exception environment in whic
is thrown.
- The exception environment in which the actual parameters of
primitive application are evaluated is the same as the exception
which the procedure or primitive is applied.
- The exception environment when evaluating a procedure body (v
dure is applied) is the same as the exception environment in whic
is applied.
- When evaluating the thrown handler exception body, the exceptic
and continuation are those saved by the handler when the handl
in the `catch` expression.

Concurrency

Using trampolining, we repeatedly apply continuations until a valu
the `HaltCont` continuation, which stops the trampolining and r
Since each continuation contains *complete information* about how
evaluation is to proceed, it is possible to have multiple threads of
pression evaluation by associating each thread with a continuation
the thread's current execution context.

Observe that we have used a `Rands` object to evaluate a collection
returning a list of values or references. Such lists are used to pas
primitive operators or procedures or to bind values to the LHS va
expressions.

Assuming that the order of evaluation of expressions in a `Rands`
matter, we can carry out these evaluations *concurrently* (or *in paral*
able hardware to support real concurrency, this can result in run-tim

Concurrency

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

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

To maintain the value order, we create an array of value slots, wi
expression. For each expression in the expression list, we create
that knows about the expression, its evaluation environment, and the
expression value should go. We can then dispatch all of these co
mechanism that evaluates them in parallel (or at least simulate paral
once each expression evaluation completes, its continuation deposi
value in the corresponding value slot.

Concurrency (continued)

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

This approach does not achieve true parallelism, since we are still ap tinuation steps one at a time (using the trampoline), but in the prese hardware, it would not be difficult to dispatch the application of the uations 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 deposited in the proper place in the array of values. The followir represents what to do with the expression value once the evaluation

```
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
    }

}
```

Concurrency (continued)

Before creating a continuation to carry out concurrent evaluation o pressions in a `Rands` object, we need to create an array that holc values. The result of evaluating concurrent expressions must affec shared environment, possibly the top-level environmment or in `let` ings. 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 paral
      _ = .d(1000)
      _ = .d(10000)
      _ = .d(100)
    in
      count
```

In this case, the value of count ends up being 10000 and not 1110(expect. (Note the use of '_' as a dummy variable place-holder.)

Concurrency (continued)

The problem here is an example of the "simultaneous update proble
a "race condition". Evaluating an expression like `set count=a`
can result in the creation of multiple continuations – several, for e
evaluate `add1(count)` – and when `count` is in the process of
in one thread, there may be other threads that are in the process o
modify it as well, with unpredictable results.

When concurrent expressions use side-effects to do their work, we
guard against race conditions. We do this through an `atomic` ex
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 evaluatio
expression by evaluating the expression directly – using a non-threa
Once the value has been determined, we pass it on to the pendin
so the threading can continue. Observe that during the evaluation
expression, the threaded trampoline stops processing queued contin

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

It's harmless to evaluate an `atomic` expression in a non-threade
However, an `atomic` expression must complete before its value
to the next continuation, so deeply nested `atomic` expressions ca
overflow. Using `atomic` expressions should be done sparingly.

Concurrency (continued)

The race condition in the previous example can now be solved by m
ification 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 memo
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
      else count
  in
    .d(16) %% => 65536
```

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