40TH ANNIVERSARY SPECIAL EDITION

Theodore Norvell NDEV

Take Back Control

Or How I Learned to Stop Worrying and Love Event Driven Code

DVD

What other people are saying about this talk:

- “[...] our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.” --- E.W.Dijkstra

The Problem

- Problem: How to write event driven code in a structured fashion.

Outline

- Event driven programs
- Structured vs unstructured control structures
- Some ideas I had
- TBC: a monad-based library to solve the problem

McMillan's warning

- When asked what sort of thing is most likely to blow a program off course, Harald McMillan would have said

Events, dear boy, events!

Events

- In a GUI: User actions such as keypresses, mouse actions, button clicks, focus, blur, etc.
- In a server: Requests from clients. Responses from DBMSs.
- In a client: Responses from servers.
- In a real-time system: The passage of time
- In a concurrent or distributed program: Receiving messages.
- In a concurrent program: Semaphore changes state.

A use case

Use case: Greet the user forever

0 The following sequence is repeated forever

0.0 System: Prompts for name
0.1 User: Types in a name and presses “enter”
0.2 System: Greets the user by name

The Use case tells a story.

A non-event-driven program

```
proc main()
   loop
      print "What is your name?"
      var name := readLine
      print "Hello, " name "."
```

The code tells a story. The structure of the story is reflected in the structure of the code.

An event-driven program

```
var nameBox := new TextField()
var question := new Label( “What is your name” )
var reply := new Label()

proc main()
   nameBox.on(enter, nameBoxHandler)
   show question
   show nameBox
   show reply

proc nameBoxHandler()
   var name := nameBox.contents()
   reply.text := “Hello, ” name “.”
```

Where did the control structure go?

Narrative structure

The structure of the non-event-driven code follows the narrative of the use case

Use case: Greet the user forever

0 The following sequence is repeated forever

- 0.0 System: Prompts for name
- 0.1 User: Types in a name and presses "enter"
- 0.2 System: Greets the user by name

```
proc main()
    loop
        print "What is your
                              name?"
        var name := readLine
        print "Hello, " name "."
```

Unstructured

Use case: Greet ...

0 The following sequence is repeated forever

- 0.0 System: Prompts for name
- 0.1 User: Types in a name and presses "enter"
- 0.2 System: Greets the user by name

```
var nameBox := new TextField
var question := new Label( "W
var reply := new Label()

proc main()
    nameBox.on(enter,nameBo
    show question
    show nameBox
    show reply

proc nameBoxHandler()
    var name := nameBox.conte
    reply.text := "Hello, " name "."
```

Changing requirements

Use case: Greet the user forever

0 The following sequence is repeated forever

- 0.0 System: Prompts for name
- 0.1 User: Types in a name and presses "enter"
- 0.2 System: Greets the user by name
- 0.3 Wait 1 second

```
proc main()
    loop
        print "What is your
                        name?"
        var name := readLine
        print "Hello, " name "."

        pause 1000
```

Changing requirements

```
var nameBox := new TextField()
var question := new Label( “What's your name” )
var reply := new Label()
var timer := new Timer()

proc main()
    nameBox.on(enter,nameBoxHander)
    timer.on( done, timeHander )
    show question ; show nameBox ; show reply

proc nameBoxHandler()
    var name := nameBox.contents
    reply.text := “Hello, ” name “.”
    hide question ; hide nameBox ;
    start timer

proc timeHandler()
    stop timer ; show question ; clear nameBox ; show nameBox
```

Protocol

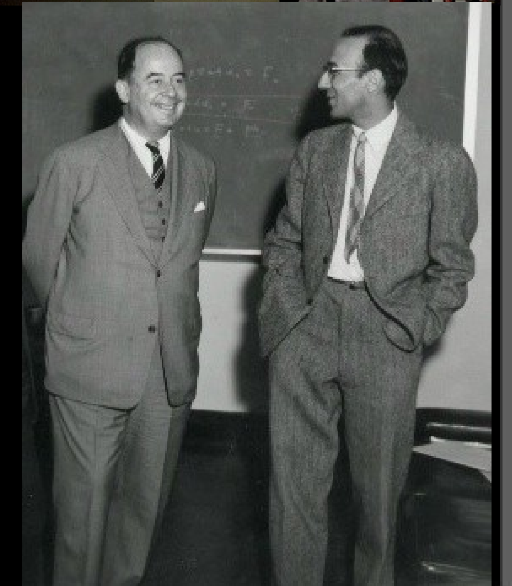
- A prefix-closed set of sequences of events.
- What formalisms do we have for describing sets of sequences?
  - State machines (Σ, Q, q_0, T)
  - Regular Expressions
  - Context Free Grammars (BNF)
  - Extended Context Free Grammar (EBNF)
    ECFG = CFG + RE

Some broad opinions

- State Machines
  - Unstructured
    (Not compositional)
  - No abstraction
  - No recursion
- Regular expressions
  - Structured: loop, choice, sequence
  - No abstraction. No recursion.
- CFGs
  - Limited structure
  - Abstraction (naming).
  - Recursion
- Extended CFGs
  - Structuring
  - Abstraction
  - Recursion

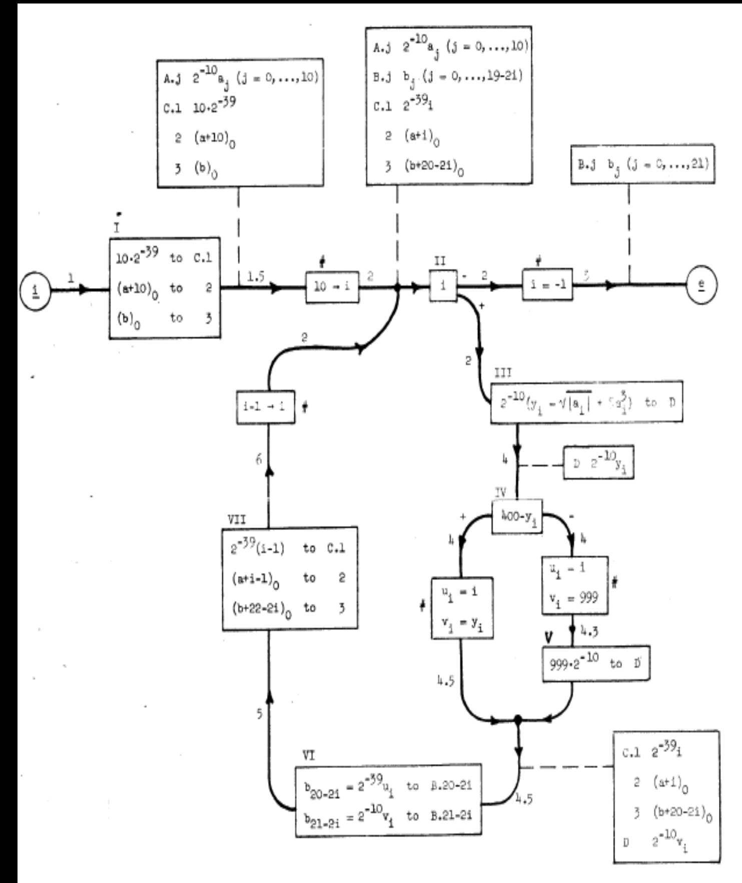
Digression on unstructured and structured programming

- 1948: Konrad Zuse publishes a paper on Plankalkül, a structured programming language
  - No one reads it.
- 1948: Von Neumann and Goldstein introduce “flow diagrams” in *Planning and Coding of Problems for an Electronic Computing instrument*
  - Everyone reads it.
- 1948: Coding is in machine language with branch instructions or conditional branch instructions (at best!).



Plankalkül vs Flow Diagrams

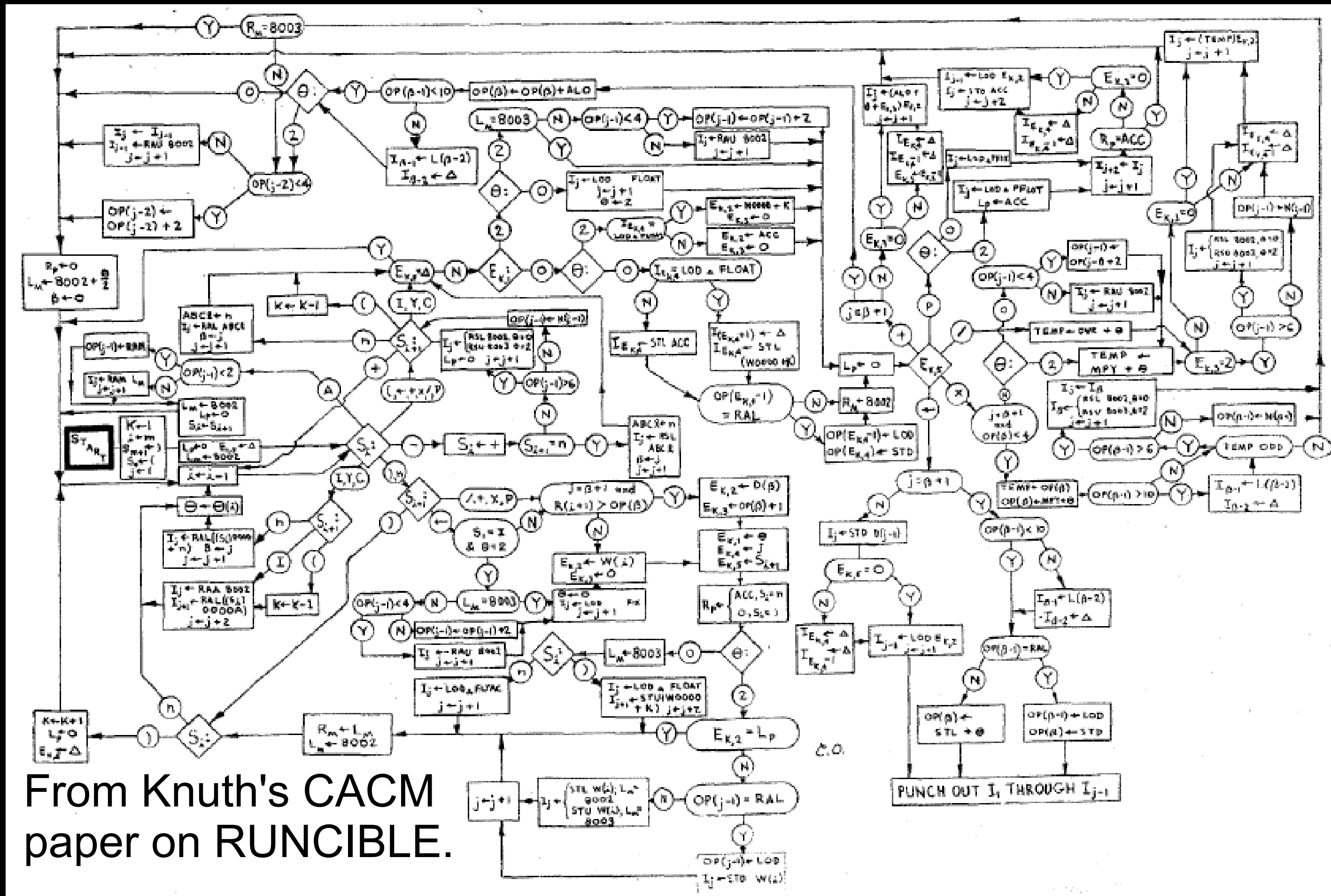
```
A2 = (A9,AΔ1)
P1    |R(V)  ⇒  R
     V|  0       0
     A|  Δ1      Δ1
      | √|V| + 5 x V³  ⇒  R
     V|  0          0      0
     A|  Δ1         Δ1     Δ1
P2    |  R(V)  ⇒  R
     V|    0       0
     A|  11 x Δ1   11 x 2
      |  W2(11) ⎡ R1(V)  ⇒  Z
     V|         ⎢ 0  0      0
     K|         ⎢    i
     A|         ⎢    Δ1     Δ1
      |         ⎢ Z > 400   ⇀   (i,+∞)   ⇒  R ┌ (10-i)
     V|         ⎢ 0                          0 ┘
     K|         ⎢
     A|         ⎢ Δ1             9            2     9
      |         ⎢ ‾‾‾‾‾‾‾
      |         ⎢ Z > 400   ⇀   (i,Z)    ⇒  R ┌ (10-i)
     V|         ⎢ 0               0          0 ┘
     K|         ⎢
     A|         ⎣ Δ1             9  Δ1        2     9
```



Digression on unstructured and structured programming

- 1953: Wheeler (re)invents the subroutine.
- 1958: Flow chart are the state of the art.
  - Coding is in assembly or Fortran II (computed go to)
- 1958--1960: Algol 58, Algol 60, and LISP provide structured control constructs if-then-else, while-do, for-do
- 1968: ACM Curriculum Committee recommends flowcharts in the first course of computer science programs.
  - Fortran IV and COBOL are dominant languages

Digression on unstructured and structured programming



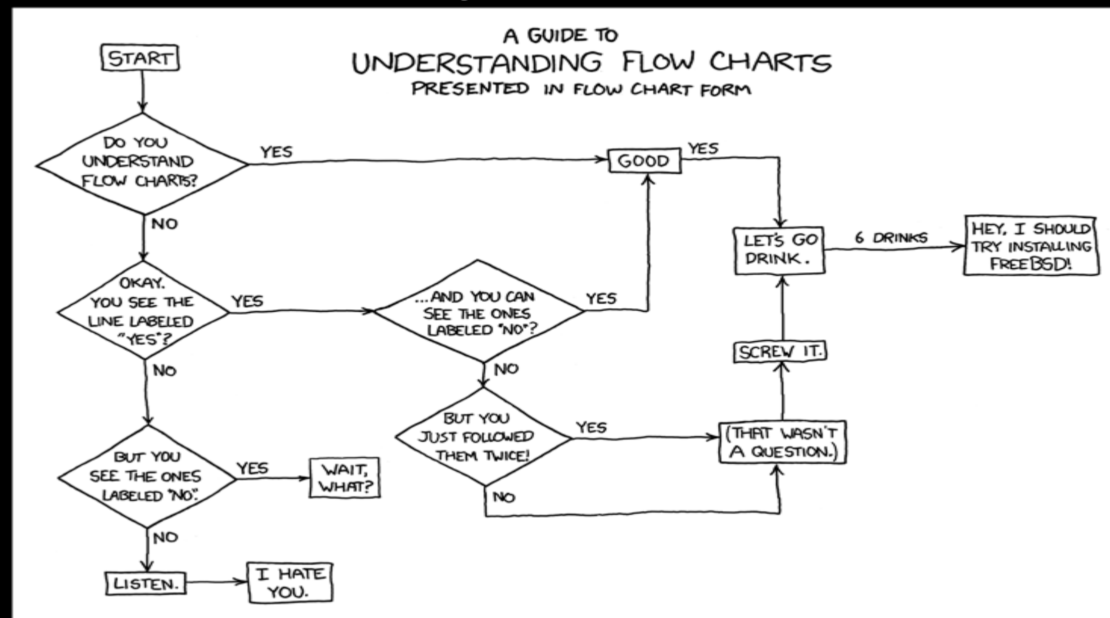
From Knuth's CACM paper on RUNCIBLE.

Digression on unstructured and structured programming

- 1967: Floyd's paper on verification of flowcharts
- 1968: Dijkstra pens his “GO TO statement considered harmful”, sparking in the Structured Programming Revolution.
  - “The go to statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program.”
- 1969: Hoare's paper on verification of structured programs
- 1978: The revolution is mostly over. Most programmers are taught to only use GOTOs to emulate “structured code”.
  - Popular languages include Fortran IV and Pascal
- 1988: Flowcharts and unstructured programming are dead.

Digression on unstructured and structured programming

- 1998-2018 Flowcharts are popular in comic strips, but not in software engineering.
  - Unstructured code is not well regarded
  - Most popular languages (JavaScript, Java, Python) don't even have a go to statement.

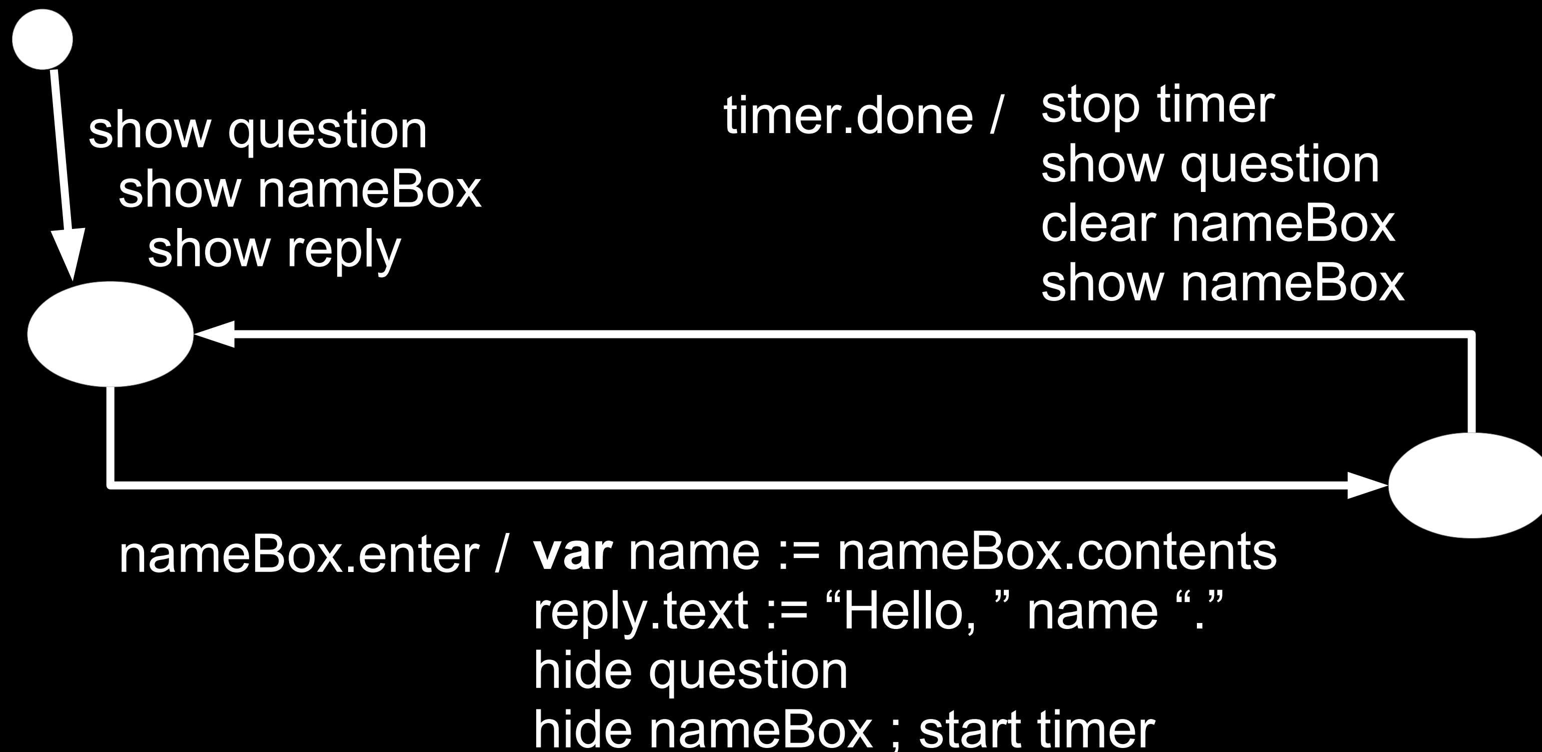


Back to Events

- Inversion of control programs are state machines
- and state machines are just unstructured programming all over again.
- Worse than that, the representation of state is often entirely implicit.
    - Consider the program we just saw.
    - It has two states. How are these states represented?
    - Only the transitions are explicitly represented.

Inv. of control ~= State machine

var nameBox := **new** TextField()
var question := **new** Label(“What's your name”)
var reply := **new** Label()
var timer := **new** Timer()



Non-inverted control ~= ECFGs

```
var nameBox := new TextField()
var question := new Label( "What's your name" )
var reply := new Label()

<main> ::= show nameBox ; show question; show reply
           <main loop>

<main loop> ::=  ( clear nameBox
                   show question
                   show nameBox
                   <get and show reply>
                   hide question
                   hide nameBox
                   pause 1000  )*

<get and show reply> ::= await nameBox.enter
                         var name := nameBox.contents
                         reply.text := "Hello, " name "."
```

An early idea

- Background: A “parser generator” translates grammars (annotated with actions) to non-event-driven driven code in an implementation language. (E.g., yacc, ANTLR, JavaCC, etc.)
- Problem: How to write UI code cleanly
- Solution 0:
  - Design user interfaces using grammars annotated with actions
  - We need a “UI generator” to generate UI code from annotated grammars.

Parser combinators

- An alternative to parser generators is “parser combinators”.
- A parser combinator is a function that takes one or more parsers and produces a parser.
- Parser combinator libraries allow us to write recursive descent parsers directly in the implementation language but which look like the EBNF grammar.
- EDSL (Embedded Domain Specific Language)

Parser combinators

- Example:
  - <A> → b <C> | d <E>   is coded as

```
function A() : Parser {
    return  alt( tok("b").seq(C),
                 tok("d").seq(E) ) ; }
```

- Here are the key library routines:
  - tok makes a parser from a terminal
  - alt(p,q) is a parser where p and q are parsers
  - p.seq(f) is a parser where p is a parser and f returns a parser.

A better idea

- Problem: How can we write UI code cleanly
- Solution 1:
  - Use a library of combinators
  - to write code directly in the implementation language
  - but which looks a lot like a grammar.

A personal note

- In 1992 I went to a summer school in Germany
- Phil Wadler was giving lectures on *Monads*
- I asked him about a problem I was having structuring a parser in a pure functional language.
- His advice:

Use a monad

- This has often proved good advice.

Can we use a Monad?

- Yes!
- Monadic Parser Combinators are a well known style of Parser Combinator.
- Monads work well with parsers because they help move information around both implicitly and explicitly.
- But what is a monad?

What is a Monad?

- Each monad M is a generic type with two operations
  - unit is a function that maps each A to an M<A>
  - p.bind(f) is an M<A> when p is an M<B> and f is a function that maps each B to an M<A>
  - With some laws similar to the monoid laws.
    1. p.bind(unit) = p
    2. unit(a).bind(f) = f(a)
    3. (p.bind(f)).bind(g) = p.bind(λx·f(x).bind(g))

Monads are useful

- Monads are used for implicit state, exceptions, I/O, collections, nondeterminacy, parsing, STM, etc.
- They are a core concept in Haskell – a functional language
- M<A> is a type representing some set of things that can produce values of type A
- unit injects a value into the monad.
- bind usually represents some kind of sequencing or composition.

Monads in Imperative Programming

- In functional programming monads are largely a solution for “problems” imperative programmers don't have:
  - No implicit state / Haskell's StateMonad
  - No synchronous IO / Haskell's IO monad
  - No exceptions / Haskell's Except transformer
- But, there are problems that imperative programmers do have where monads can help
  - Poor Asynchronous IO
  - No concurrent programming (in some environments)
  - Collections

Take Back Control

- Take Back Control (TBC) is a library I've designed for dealing with
  - Asynchronous I/O
  - Cooperative multithreading
  - Event driven programming in general
- Written in the Haxe language
- Haxe transpiles to JavaScript, Python, and other languages
- TBC can be used from Haxe, Python, JavaScript, Typescript, CoffeeScript, PHP, etc.

Take Back Control

Sneak peek: This is code written in Haxe using TBC.

```
static function mainLoop() : Process<Triv> { return
    loop  ( clearText( nameBox ) >
            show( nameBox )  >
            show( question ) >
            getAndDisplayAnswer() >
            hide( question ) >
            hide( nameBox ) >
            pause( 1000 ) ) ; }

static function getAndDisplayAnswer()
: Process<Triv> { return
      await( enter( nameBox ) && getValue( nameBox ) ) >=
      (name:String) -> hello(name) ; }
```

The Process Monad

- A central type provided by TBC is
  The Process Monad.
- For each type `A`, there is a type `Process<A>` that represents processes that deal with events and then produce values of type `A`.
- `unit(x)` is a process that deals with no events and produces the value `x` right away.
- `p.bind(f)` is a process that first behaves like process `p` and then like `f(b)` where `b` is the value produced by process `p`.

Types

- For any types `A` and `B`

  ```
  unit(a) : Process<A>
  ```

  and

- ```
  p.bind(f) : Process<A>
  ```

- if

  ```
  a : A
  p : Process<B>
  f : f -> Process<A>
  ```

  (i.e. `f` is a function from `B` to `Process<A>`)
  ```

Digression on anonymous functions: a.k.a. λ expressions

In Haxe

```
function f( i : Int ) : Bool { return i==x ; }
... a.map( f ) ...
```

is the same as

```
final f = function( i : Int ) : Bool { return i==x ; } ;
... a.map( f ) ...
```

Digression on anonymous functions: a.k.a. λ expressions

And

```
final f = function( i:Int ) : Bool { return i==x ; } ;
... a.map( f ) ...
```

is the same (if `f` is only used once) as

```
... a.map( function( i:Int ) : Bool { return i==x ; } ) ...
```

which is the same (in Haxe 4) as

```
... a.map( (i:Int) -> i==x  ) ...
```

which is the same as

```
... a.map( i -> i==x ) …
```

In JavaScript: `a.map( i => i==x )`

An example: unit and bind

- `pause(t)` is a process that waits until `t` milliseconds have passed. It produces `null`.
- `pause(1000).bind( x -> unit(42) )` produces a result of 42 after a 1 second delay.

Side effects: exec

- `exec(f)` is a process that waits for no events and immediately calls `f`; it produces the result of `f()`.
- E.g.

```
    pause(1000).bind( x -> print(42) )
```

  prints 42 after a pause of 1 second if we define

```
    function print(n : Int):Process<Triv> {
        return exec( ()->{trace(n);null} );}
```

Operators

- In Haxe we can use operator overloading

  `p >= f` abbreviates `p.bind( f )`

  `p > q` abbreviates (approximately) `p.bind( x -> q )`

- For example
  - `pause(1000) > print(42)`
    prints 42 after 1s

- Fine print: `pause(1000) > print(42)` calls print immediately, but `pause(1000) >= x -> print(42)` calls print 1 second after the process starts.
  - As `print` is a pure function, it doesn't matter when it's called.

Reading Monadic Code

JavaScript / TypeScript

```
p.bind( x =>
E.bind( y =>
F.bind( z =>
G ) ) )
```

The result of p is *bound* to x etc.

Haxe

```
p >=  x ->
E >=  y ->
F >=  z ->
G
```

That's why it's called “bind”.

Note that x can be used in expressions E, F, and G.

A little Haxe Hack for Haskellers

I wrote a macro called `seq` so that

```
p >=  x ->
E >=  y ->
F >=  z ->
G
```

can be written as

```
seq( (var x := p),
     (var y := E),
     (var z := F),
     G )
```

A strange coincidence

In Haskel, bind is written as >>=.

But Haxe doesn't allow new operators, so I lopped off one > to get >=.

By coincidence, Zuse's 1948 paper on Plankalkül used almost the same symbol and direction for assignment.

Implementing the Process Monad

- Each process `p : Process<A>` has a method
  `p.go( k : A -> void )`
- The `go` method initiates the process.
- Its argument specifies what is to be done with the result. `k` is for *k*ontinuation.
- `unit(a).go( k )` means `k(a)`
- `p.bind( f ).go( k )` means
  `p.go( b -> f(b).go( k ) )`

Implementing the Process Monad

- `exec(f).go( k )` means `k( f() )`
- `pause( t ).go( k )` means

```
var timer = new Timer( t ) ;
timer.run = () -> k( null ) ;
timer.start() ;
```

- E.g. `pause(1000).bind( x->print(42) ).go(k)`

  ≡ (approx.)

```
var timer = new Timer(1000) ;
timer.run = () -> (x ->
                     exec( ()->
                            {trace(42);null} )
                   )(null).go(k) ;
timer.start() ;
```

  ≡

```
var timer = new Timer(1000) ;
timer.run = () -> k({trace(42);null}) ;
timer.start() ;
```

Extending the framework

- You can easily extend the framework by creating your own classes that implement the `Process` interface.
- You just extend class `ProcessA<A>` while overriding method

```
public function go( k : A -> Void ) { … }
```

Loops

Define

```
public static function loop<A>( p : Process<A> )
                                        : Process<Triv> {
     return p >= (a -> loop(p)) ; }
```

- [N.B. It looks like an infinite recursion, but it is not! Bind does not call `a -> loop(p)` . It just stores the function in the `Process` object that gets returned. The following definition would not work

```
public static function loop<A>( p : Process<A> )
                                        : Process<Triv> {
     return p > loop(p) ; }
```

This *is* an infinite recursion.]

Now you can understand

our example

```
static function mainLoop() : Process<Triv> { return
        loop( clearText( nameBox ) >
              show( nameBox )  >
              show( question ) >
              getAndDisplayAnswer() >
              hide( question ) >
              hide( nameBox ) >
              pause( 1000 ) ) ;
```

We define clearText as:

```
static function clearText( el : InputElement ) :
                                  Process<Triv> { return
        exec( () -> {el.value = ""; null;}  ) ; }
```

And similarly for hide and show.

Guards

- But what about

```
static function getAndDisplayAnswer() : Process<Triv> {
    return await( enter( nameBox ) && getValue( nameBox ) )
           >= (name:String) -> hello(name) ; }
```

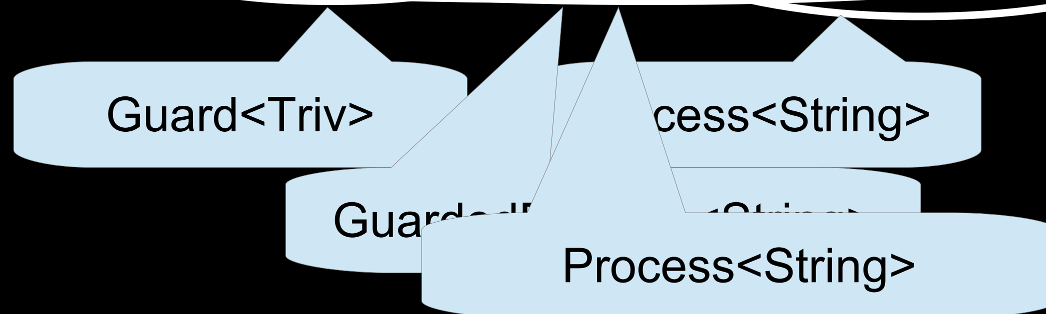
- What's happening there?
- Define a type `Guard<E>`. Objects of type `Guard<E>` represent the act of waiting for one of some set of events.
- Above `enter(nameBox)` is a `Guard` object representing the act of waiting for the enter key to be pressed in the name box.

Guarded processes

- If e is a Guard<E> and p is a Process<A>, then
  e && p   is a GuardedProcess<A>
- It represents the act of waiting for an event and then behaving like p.
- If gp is a GuardedProcess<A>, then
  await(gp)   is a Process<A>

So in the example we have

```
await( enter(nameBox) && getValue(nameBox) )
```



Guarded Processes

- We have

```
static function getAndDisplayAnswer() : Process<Triv> {
  return await( enter( nameBox ) && getValue( nameBox ) )
         >= (name:String) -> hello(name) ; }

static function hello( name : String ): Process<Triv> {
  return putText( reply, "Hello "+name ) ; }
```

- So the result of the `await` process is piped into the `hello` function which produces a process to put the string into the reply box.

Event values

- The `&&` operator throws away the underlying event data.
- We can also pipe information from the event to a process.
- If `e` is an `Guard<E>` and `f` is a function in `E -> Process<A>`, then

  ```
  e >> f
  ```

  is a `GuardedProcess<A>`. For example

  ```
  await( e >> unit )
  ```

  is a `Process<E>`.

Event filtering

- If `e` is a `Guard<E>` and `g` is a function in `E -> Bool`, then `e & g` is a `Guard<E>`
- `e & g` ignores events where `g` gives false. For example the `enter( nameBox )` guard is constructed as follows

```
static function enter( el : Element ) : Guard<Event> {

        function isEnterKey( ev : Event ) : Bool {
            var kev = cast(ev, KeyboardEvent) ;
            return kev.code == "Enter" ; }

        return keypress( nameBox ) & isEnterKey ;
}
```

Choices

- Given two guarded processes `gp0` and `gp1`,
  `gp0 || gp1` is also a guarded process
- The first event to happen wins.
- Here is an example

```
loop( await(
          upKey(body) >> preventDefault > exec( bigger )
        || downKey(body) >> preventDefault > exec( smaller ) ))
```

- Here is an example with a timeout.

```
await(    click( b1a ) && out("1A")
       || click( b1b ) && out("1B")
       || timeout( 2000 ) && out( "too slow" ) )
```

Extending the framework

- You can create your own class of guards just by extending class `GuardA<E>` while overriding this method
- `public function enable( k : E -> Void ) : Disabler { …`
- The `k` represents the thing to do when the event happens.
- The result is simply an object that can disable the guard.

Implementing await

- Consider
- `await( g && p || h && q ).go( k )`
- Enables, guard `g`, passing in a continuation that
  - Disables both `g` and `h` and then
  - calls `p.go( k )`
- Also enables guard `h`, passing in a continuation that
  - Disables both `g` and `h` and then
  - calls `q.go( k )`

Cooperative Multithreading

- JavaScript has one thread.
- But just as cooperative multitasking shares one CPU among many tasks, cooperative multithreading can share one thread among many executing processes.
- If `p` and `q` are both processes then `par(p, q)` is a process.
- `par(p, q).go( k )` starts both processes. `k` will be called only after both processes complete.
- Each process runs uninterrupted until it awaits an event.
- `parFor(n, f)` runs processes `f(0)`, `f(1)`, …, `f(n-1)` in parallel.

Exception Handling

- What if there is an exception?
- We can set up an exception handler

```
     attempt( p, f )
or   attempt(p, f, q)
```

  where `f` is a function from exceptions to processes and q is a process to be done regardless.
- E.g.

```
openFile >= (h:Handle) ->
attempt( doStuffWithIt( h ),
         (ex:Dynamic) -> cope(ex),
         closeFile( h ) )
```

- Implementation: I lied earlier. The `go` method actually takes two continuations: One for normal termination and one for exceptional termination.

Process Algebras

- Process Algebras are theoretical calculi intended for modeling concurrent code
- similar to how lambda calculus models sequential code.
- Communicating Sequential Processes (CSP)
- Calculus of Communicating Systems (CCS)
- Although TBC was inspired by Context Free Grammars, it ended up recreating many of the ideas of process algebras.

Other approaches

- See blog post “What color is your function?” by Bob Nystrom
  http://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/
- Node style functions. Explicit continuation passing everywhere.
- Promises. More compositional than node-style functions.
- Promises + async/await syntax from ES8. Take care of much of the boiler plate of promises.
- These cope with sequencing, but none really tackle choice.

Other approaches

- Mostly require unusual language features, multithreading, or macros.
- C#: async methods (language feature)
- Java Swing dialogs (multithreading)
- Golang: go routines (lang., multithreading)
- Closjure: core.async (relies on macros. Can run on 1 thread or more!)

Conclusion

- Inversion of control is not as cool as it sounds.
  - It's unstructured programming all over again.
- TBC gives you an *extensible* framework to write *structured* programs for
  - asynchronous event handling
  - concurrency (cooperative multithreading)

Conclusion

- TBC supports
  - Composition: sequential, parallel, choice, looping
  - Abstraction via subroutines and parameters
  - Recursion if you need it.
- TBC works in single-threaded environments and can be used from
  - Haxe
  - JavaScript / Typescript
  - Python
  - PHP

Why Strangelove?

- In Dr. Strangelove, events take the world to the brink of disaster when Colonel Jack Ripper tries to single handedly start a nuclear war.
- Ripper inverts control.
- It is up to Presidents Merkin and Kissov to Take Back Control.

O.P.E.

P.O.E.

- In 1992 Phil Wadler popularized Monads in a paper called:

  The Essence of Functional Programming

- One of his monads is essentially the Process monad.
- By *Functional Programming*, he meant *Pure* Functional Programming.
  - Essence Of Pure functional programming
  - → Essence Of Purity
  - → Purity Of Essence
  - → Parsers Of Events
  - → Processes Over Events
  - → Peace On Earth

The end