Theodore Norvell
Dept. ECE, Memorial University of Newfoundland
CSER 2021

Upending Inversion of Control

Theodore Norvell
Dept. ECE, Memorial University of Newfoundland
CSER 2021

Upending Inversion of Control

Do you practice structured programming?

- Of course you do.
- We were taught to use structured control constructs such as
  - loops
  - ifs
  - sequential composition (one damn thing after another)
- Goto statements are bad
- Subroutines are good
- Global state is bad.

But,

do you practice structured programming
when you are handling events?

Events

- An event is anything a program may need to wait for
  - In a GUI:
    - user actions such as keypresses, mouse actions, button clicks, etc.
  - In distributed or system:
    - incoming requests and responses from clients, servers, and peers.
  - In a concurrent program
    - changes of state
    - messages on internal channels

Example: A use case

A Use case tells a story.

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

Example: A "console" 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.

Narrative structure

The structure of the console program follows the narrative of the use case

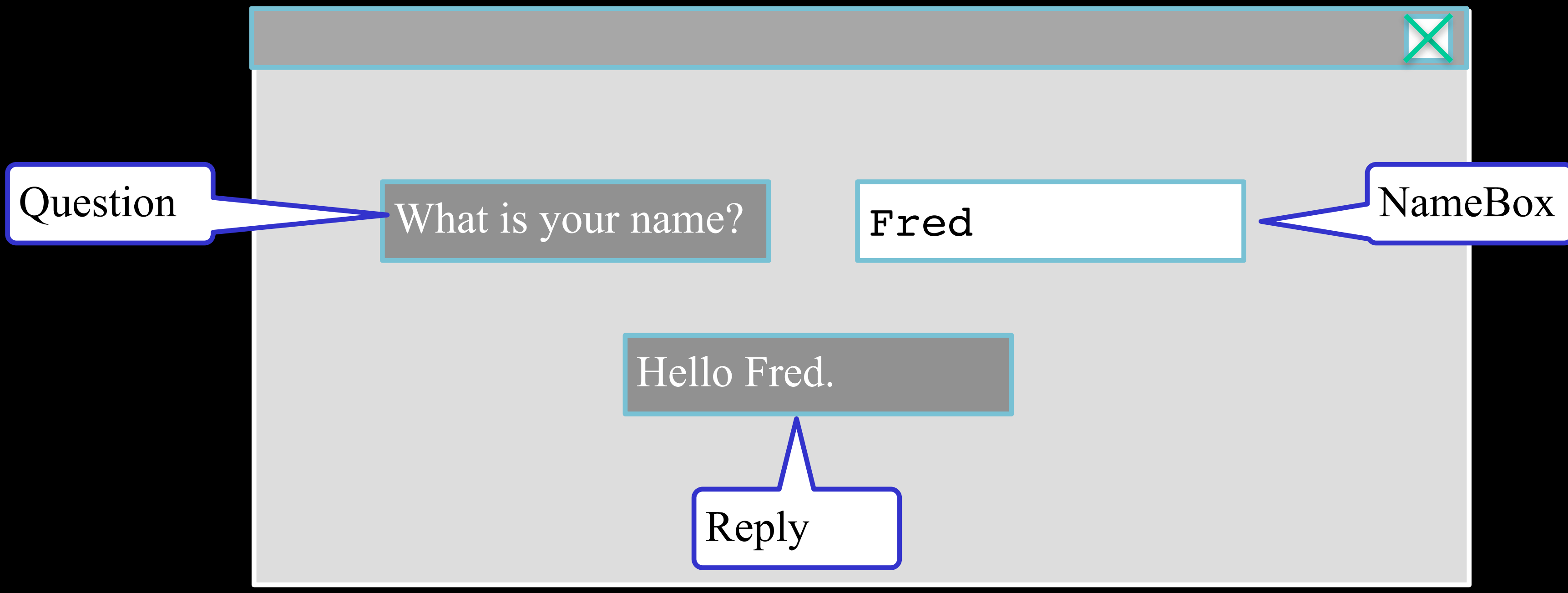
Use case: Greet the user forever

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 "."
```

But we want a GUI!



The GUI 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 “.”
```

Event handler. means inversion of control.

Where did the control structure go?

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

```
proc main()
    loop
        print “What is your
                        name?”
        var name := readLine
        print “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 ms
```

Changing requirements

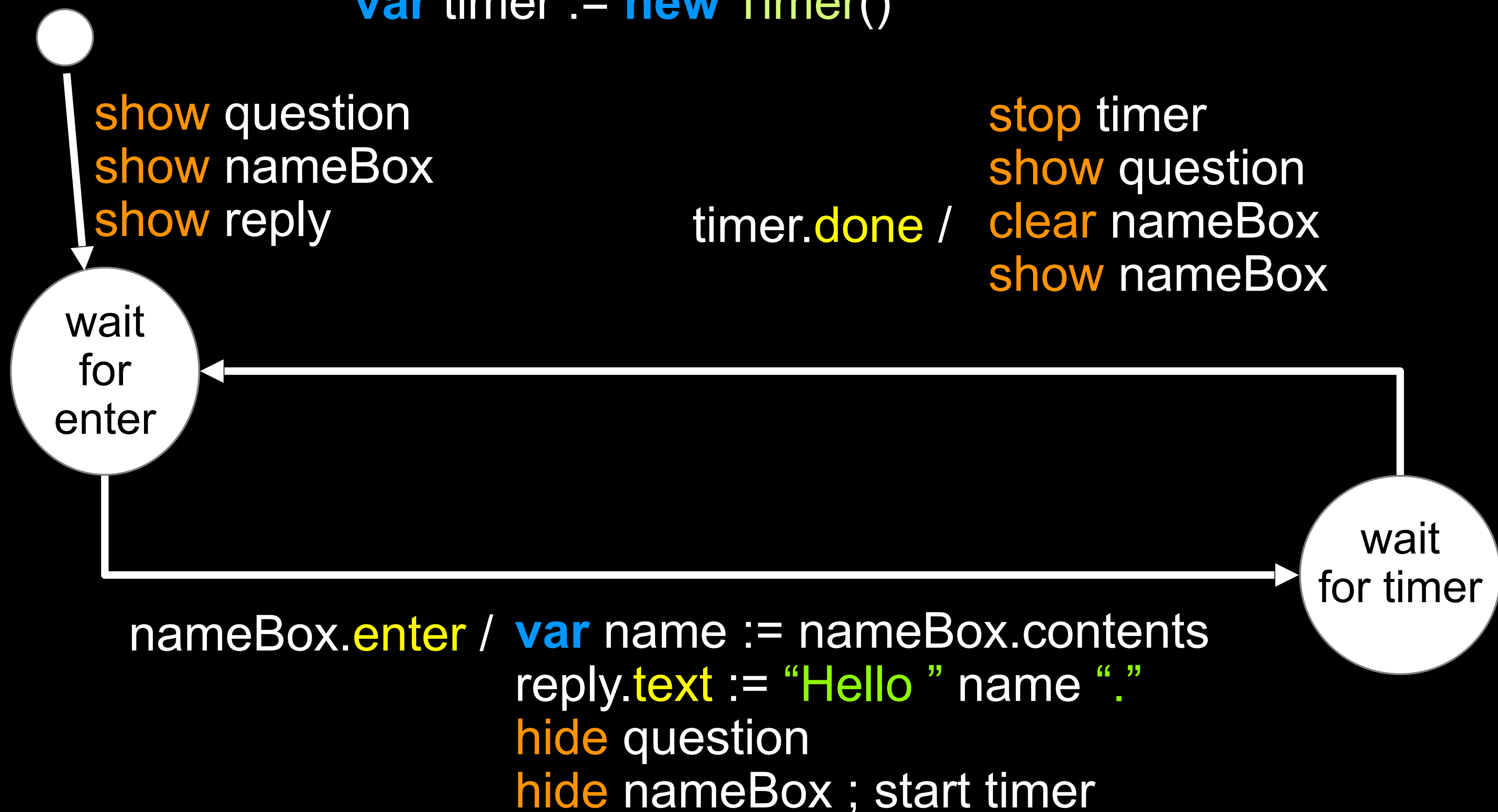
```
var nameBox := new TextField()
var question := new Label( “What is your name” )
var reply := new Label()
var timer := new Timer(1000 ms)

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
```

State machines

- Inversion of control programs are state machines
- Unstructured programming all over again
  - But worse
- Where are the states?
  - Our program has two states
  - Where are they in the code?

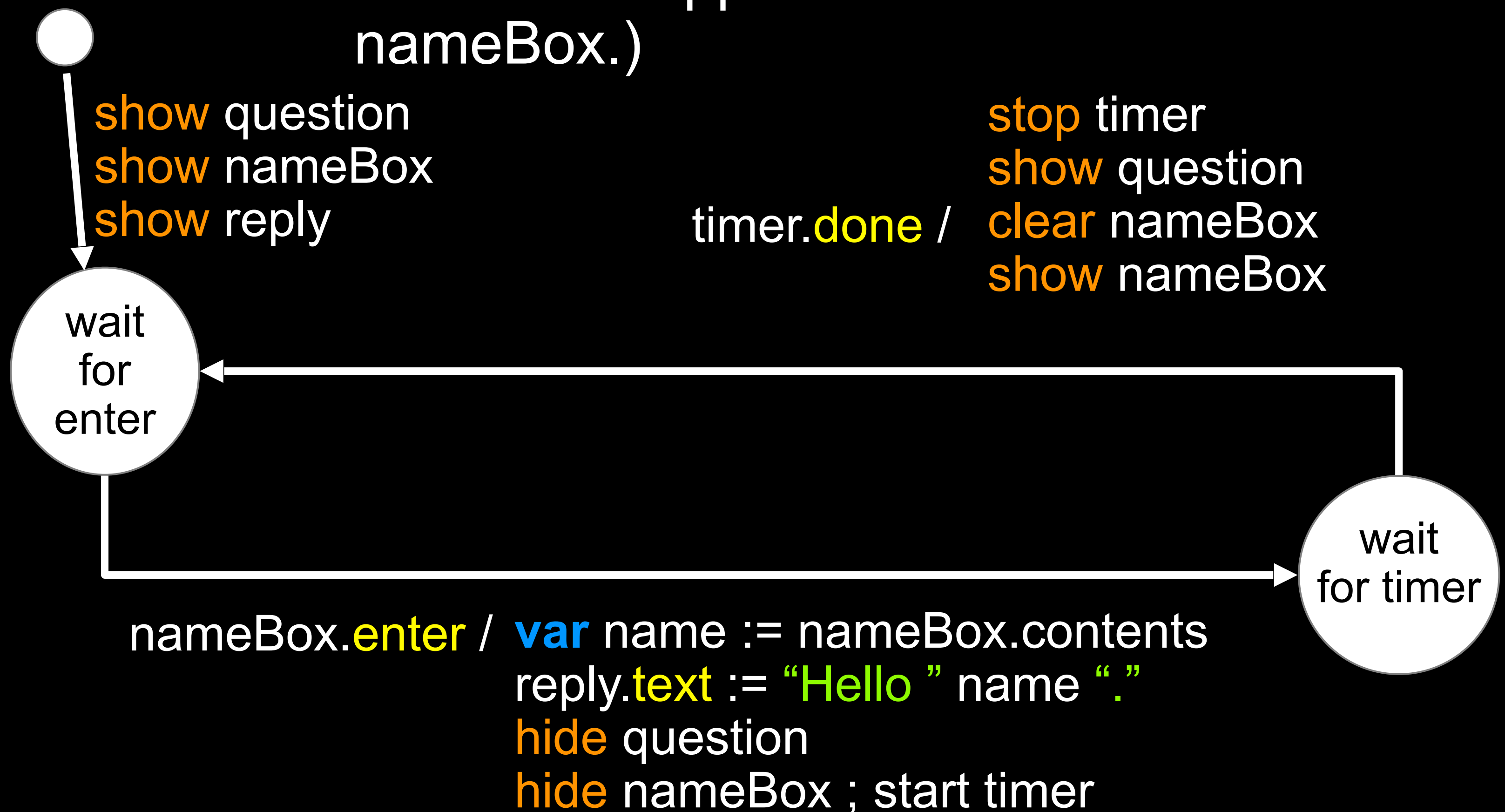
Inv. of control ≃ State machine

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



Inv. of control ≃ State machine

NEAR BUG! What if there is an enter event while the timer is running? (The only reason this won't happen is that I hide the nameBox.)



What I would like

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

proc main()

    show reply

    loop
        clear nameBox
        show nameBox
        show question

        getAndDisplayAnswer
        hide question
        hide nameBox

        pause 1000 ms
```

```
proc getAndDisplayAnswer
        wait for enter on nameBox
        var name := nameBox.contents
        reply.text := “Hello ” name “.”
```

What I would like

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

proc main()

    show reply

    loop
        clear nameBox
        show nameBox
        show question

        getAndDisplayAnswer
        hide question
        hide nameBox

        pause 1000 ms
```

```
proc getAndDisplayAnswer
    wait for enter on nameBox
    var name := nameBox.contents
    reply.text := “Hello ” name “.”
```

state

state

The Problem

How can we write event-driven code in a structured fashion?

The idea

Make a library based on process algebra

- Process algebras
  - extend context-free grammars with
    - interactivity
    - concurrency
    - communication
    - internal and external choice
  - Examples: CSP, CCP, π-calculus, ACP

Take Back Control*

Take Back Control (TBC)

- A library for
  - Asynchronous I/O
  - Cooperative multithreading
  - Event-driven programming in general
- Written in the Haxe language
  - Haxe transpiles to JavaScript, Python, and other languages
- Abstracts away from inversion of control

* I had this name *before* the Brexit "Leave" campaign.

Example

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 ) )
      >= hello ; }

static function main() {
    ... create the GUI ...
    mainLoop().run( ) ; }
```

`Process`es

A generic type

`Process<A>`

Each object of type `Process<A>`

- is immutable
- represents a *specification of* behaviour
- has a result type `A`

Processes

When p is a Process<A>

- p.run() starts running the process & returns immediately
- p.go(f, g) similar with continuations
  - f(a) if the run succeeds and
  - g(e) if it fails

```
static function main() {
    ... create the GUI ...
    mainLoop().run( ) ; }
```

Build the view

Build a controller process object

Run the controller

Making `Process` Objects

Some ways to make process objects

- `pause( t )`
  - when run, it waits t milliseconds
  - the result is null
- `exec( f )` is a `Process<A>`
  - when run, it calls closure `f : () -> A`
  - the result is the value of `f()`

Making `Process` Objects

Some ways to make process objects

- `pause( t )`
  - when run, it waits t milliseconds
  - the result is null
- `exec( f )` is a `Process<A>`
  - when run, it calls closure `f : () -> A`
  - the result is the value of `f()`

Using exec

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

Using exec

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

`() -> {el.value = ""; null;}` is a Haxe lambda expression

Using `exec`

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

`() -> {el.value = ""; null;}` is a Haxe lambda expression

`show`, `hide`, `getText` and `putText` are similar

Process combinators

Some ways to combine process objects

- `p > q`
  - run `p` and then run `q`.
- `p >= f`
  - run `p` to get a result `a`
  - then run the result of `f(a)`
- `loop( p )` run `p` over and over forever

Process combinators

Some ways to combine process objects

- `p > q`
    - run `p` and then run `q`.
- `p >= f`
    - run `p` to get a result `a`
    - then run the result of `f(a)`
- `loop( p )` run `p` over and over forever

Process combinators

Some ways to combine process objects

- `p > q`
  - run `p` and then run `q`.
- `p >= f`
  - run `p` to get a result `a`
  - then run the result of `f(a)`
- `loop( p )` run `p` over and over forever

Process combinators

Some ways to combine process objects

- `p > q`
    - run `p` and then run `q`.
- `p >= f`
    - run `p` to get a result `a`
    - then run the result of `f(a)`
- `loop( p )` run `p` over and over forever

Monad inspired by parsing combinators

Our example

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

Guards represent events

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

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

- `enter( nameBox )`
  - makes a `Guard<js.html.Event>` object representing events
- `getValue( nameBox )`
  - makes a `Process<String>`
- `enter( nameBox ) && getValue( nameBox )`
  - makes a `GuardedProcess<String>` object
- `await( ... )`
  - makes a `Process` from the `GuardedProcess`.
- `await( ... ) >= hello`
  - makes a `Process` that, when run, will update the `reply` label.

Guards represent events

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

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

- `enter( nameBox )`
    - makes a `Guard<js.html.Event>` object representing events
- `getValue( nameBox )`
    - makes a `Process<String>`
- `enter( nameBox ) && getValue( nameBox )`
    - makes a `GuardedProcess<String>` object
- `await( ... )`
    - makes a `Process` from the `GuardedProcess`.
- `await( ... ) >= hello`
    - makes a `Process` that, when run, will update the `reply` label.

Guards represent events

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

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

- `enter( nameBox )`
  - makes a `Guard<js.html.Event>` object representing events
- `getValue( nameBox )`
  - makes a `Process<String>`
- `enter( nameBox ) && getValue( nameBox )`
  - makes a `GuardedProcess<String>` object
- `await( ... )`
  - makes a `Process` from the `GuardedProcess`.
- `await( ... ) >= hello`
  - makes a `Process` that, when run, will update the `reply` label.

Guards represent events

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

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

- `enter( nameBox )`
  - makes a `Guard<js.html.Event>` object representing events
- `getValue( nameBox )`
  - makes a `Process<String>`
- `enter( nameBox ) && getValue( nameBox )`
  - makes a `GuardedProcess<String>` object
- `await( ... )`
  - makes a `Process` from the `GuardedProcess`.
- `await( ... ) >= hello`
  - makes a `Process` that, when run, will update the `reply` label.

Guards represent events

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

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

- `enter( nameBox )`
  - makes a `Guard<js.html.Event>` object representing events
- `getValue( nameBox )`
  - makes a `Process<String>`
- `enter( nameBox ) && getValue( nameBox )`
  - makes a `GuardedProcess<String>` object
- `await( ... )`
  - makes a `Process` from the `GuardedProcess`.
- `await( ... ) >= hello`
  - makes a `Process` that, when run, will update the `reply` label.

Guards represent events

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

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

- `enter( nameBox )`
  - makes a `Guard<js.html.Event>` object representing events
- `getValue( nameBox )`
  - makes a `Process<String>`
- `enter( nameBox ) && getValue( nameBox )`
  - makes a `GuardedProcess<String>` object
- `await( ... )`
  - makes a `Process` from the `GuardedProcess`.
- `await( ... ) >= hello`
  - makes a `Process` that, when run, will update the `reply` label.

Event-driven choice

- Given two guarded processes `gp0` and `gp1`,
  `gp0 || gp1` is also a guarded process
- The first event to happen wins.
- Combining use cases

```
loop ( await(      saveUseCase
                || loadUseCase
                || addItemUseCase
                || deleteItemUseCase )
```

Things I don't have time to show

- Filtering events
- Or-ing events
- Parallel composition `par(p, q)` is a Process object.
  - The two processes are run on the same thread!
- Exception handling
- Loops with exits
- Communication channels – being developed

Similar things

- promises
  - provide some structure ✔
  - not immutable (Promise is not a monad!) ✘
  - handle choice poorly ✘
- async / await
  - do not handle choice ✘
  - part of the language ✘✔
- clojure.core.async
  - similar
  - relies on macros ✘
- actors
  - executions as objects ✘✔
  - only parallel composition ✘
- process algebras
  - same idea ✔
  - implementations? ✘

Conclusion

- TBC an extensible, embedded, domain-specific library supporting
  - Composition: sequential, parallel, choice, looping
  - Abstraction via subroutines and parameters
  - Recursion
- We can write code that
  - is structured and subroutineable
  - corresponds to use cases
  - is easy to understand, modify, and maintain.

Thank you

Read more at

http://sourcephile.blogspot.com/2015/05/

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 ) { … }
```

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 >= 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() ;
```

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.]

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.

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 ;
  }
```

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 )`

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.