Take Back Control

Presented at NECEC 2015, St. John's, NL

Theodore S. Norvell

Computer Engineering Research Labs, Dept. ECE, MUN

2015 November 5

The research question.

Can we write pull code in a pushy world?

Asynchrony

Events happen when they want to, not when our program wants them to.

- Users provide input.
- Other computers send data (or fail).
- Parallel computations complete or indicate progress or fail

The world pushes data at our programs.

The usual solution

The usual solution

- Inversion of control.
- Program responds to events.
- The story line of the program is not reflected in the code.

What do I mean by that?

How I learned to program

The code **pulls** the data it needs from the user, from files, from the net.

```
proc main()
    loop
        print "What is your name"
        var name := read
        print "hello " name
```

Can you see the story told by the code?
The structure of interaction is reflected in the structure of the code.

How we write code in an asynchronous world

The world **pushes** data at our program.

```
var nameBox := new TextBox()
var question := new Label( "What is your name" )
var reply := new Label()
proc main()
    attach handler nameBox nameBoxHandler
    show question
    show nameBox
proc nameBoxHander( event )
    var name := nameBox.contents()
    reply.text := "hello " name
    show reply
```

Where did the structure go?

Inversion of control

This style of programming is named *Inversion of Control*
The UI framework calls our code instead of our code calling the library.
Having a fancy name doesn't mean it's a good idea.

Inversion of control

This style of programming is named *Inversion of Control*

The UI framework calls our code instead of our code calling the library.

Having a fancy name doesn't mean it's a good idea.

Use cases

Use cases are stories. They are often used in requirements gathering.
A use case is like a script that describes a (part of) the interactions various parties.

Juliet Go ask his name: if he be married.
My grave is like to be my wedding bed.

Nurse His name is Romeo, and a Montague;
The only son of your great enemy.

Juliet My only love sprung from my only hate!
Too early seen unknown, and known too late!
Prodigious birth of love it is to me,
That I must love a loathed enemy.

Use cases

Use case: Greet user by name forever.

0. **System:** Prompts for name.
1. **User:** Types in a name and presses “enter”.
2. **System:** Greets the user by name
3. Back to 0.

Functional requirements can be captured by a set of use cases.

Use cases

Note how the "pull code" follows the structure of the use case.

0. **System:** Prompts for name.
1. **User:** Types in name and enter.
2. **System:** Greets the user by name
3. Back to 0.

loop

    print "What is your name"
    var name := **read**
    print "hello " name

Lack of structure with inversion of control

```
var nameBox := new TextBox()
var question := new Label( "What is your name" )
var reply := new Label()
proc main()
    attach handler nameBox nameBoxHandler
    show question
    show nameBox
proc nameBoxHander( event )
    var name := nameBox.contents()
    reply.text := "hello "  name
    show reply
```

Where did the structure go?

Add a requirement

We want that each name entered is greeted for a minimum of 1 second.

0. **System:** Prompts for name.
1. **User:** Types in name and enter.
2. **System:** Greets the user by name
3. **System:** Waits one second
4. Back to 0.

Add a requirement to “pull code”

```
proc main()
    loop
        print "What is your name"
        var name := read
        print "hello " name
        pause 1.0
```

Add a requirement to inversion of control version

... **var** timer := **new** Timer(1.0)
proc main()
    attach handler nameBox nameBoxHandler
    attach handler timer timeHandler
    show question ; **show** nameBox
proc timeHandler()
    attach handler nameBox nameBoxHandler
    stop timer ; **show** question ; **show** nameBox
proc nameBoxHander(event)
    var name := nameBox.contents()
    reply.text := "hello " name ; **show** reply
    hide question ; **hide** nameBox
    detach handler nameBox nameBoxHandler
    start timer

Add a requirement

Code inflation:

Pull (console app)	Push (IoC)
20%	100%

But the code inflation is not the main point.

It is that, with inversion of control, changes are all over the place.

- No structure.
- No procedural abstraction.
- New global state.
- New global invariants.

Add a requirement

Code inflation:

Pull (console app)	Push (IoC)
20%	100%

But the code inflation is not the main point.

It is that, with inversion of control, changes are all over the place.

- No structure.
- No procedural abstraction.
- New global state.
- New global invariants.

Callback hell

Programmers have a name for programming with callbacks:

Callback hell

Why?

- Global state
- Often implicit state buried in objects like 'timer'
- Typically undocumented global invariants
  - "if Timer 'timer' is running, Label 'question' is not showing"
- It is unstructured.
- It defies procedural abstraction.
- All stories (use cases) are put into a blender.
- Lack of modularity, tracabilty, malleability, maintainability.

Callback hell

Programmers have a name for programming with callbacks:
Callback hell

Why?

- Global state
- Often implicit state buried in objects like ‘timer’
- Typically undocumented global invariants
  - “if Timer ‘timer’ is running, Label ‘question’ is not showing”
- It is unstructured.
- It defies procedural abstraction.
- All stories (use cases) are put into a blender.
- Lack of modularity, tracabilty, malleability, maintainability.

Callback hell

Programmers have a name for programming with callbacks:
Callback hell
Why?

- Global state
- Often implicit state buried in objects like ‘timer’
- Typically undocumented global invariants
  - “if Timer ‘timer’ is running, Label ‘question’ is not showing”
- It is unstructured.
- It defies procedural abstraction.
- All stories (use cases) are put into a blender.
- Lack of modularity, tracabilty, malleability, maintainability.

Writing pull code in a pushy world.

"The old dog barks backwards" – Robert Frost

I'm nostalgic.

I miss writing code that I can understand the next day.

The question:

Can we write pull code in a pushy world?

Writing pull code in a pushy world.

"The old dog barks backwards" – Robert Frost

I'm nostalgic.

I miss writing code that I can understand the next day.

The question:

Can we write pull code in a pushy world?

Monads to the rescue

A *monad* is a set of objects together with a sequencing operation.
Like a *monoid* but with a twist.
But I don't have time to explain all about monads.
We'll just look at the one that solves our problem.

Monads to the rescue

A *monad* is a set of objects together with a sequencing operation.
Like a *monoid* but with a twist.
But I don't have time to explain all about monads.
We'll just look at the one that solves our problem.

Take Back Control

The rest of the talk presents
a library called *Take Back Control*.
TBC is written in Haxe.
Haxe a Java-like language that can be translated into

- JavaScript
- Java
- C++
- C#
- Python

So the library can be used on a variety of platforms and from a variety of languages.

Objects that represent actions

Objects of class `Process` can represent individual actions.

- `exec(fred)` is a `Process` that represents the action of calling function `fred`.

Sequencing

Objects of class `Process` can also represent sequences of actions.

- `exec(fred) > exec(ginger)` is a `Process` that represents the sequence of actions
  - calling function `fred`
  - and later calling function `ginger`

`>` is sequential composition.

In general a `Process` represents a set of sequences of individual actions and events.

Sequencing

Objects of class `Process` can also represent sequences of actions.

- `exec(fred) > exec(ginger)` is a `Process` that represents the sequence of actions
  - calling function `fred`
  - and later calling function `ginger`

`>` is sequential composition.

In general a `Process` represents a set of sequences of individual actions and events.

Sequencing

Objects of class `Process` can also represent sequences of actions.

- `exec(fred) > exec(ginger)` is a `Process` that represents the sequence of actions
  - calling function `fred`
  - and later calling function `ginger`

`>` is sequential composition.

In general a `Process` represents a set of sequences of individual actions and events.

Processes produce results

In fact `Process` is parameterized by a type.

A `Process<Int>` object is a process that, when executed, outputs an `Int`

Examples

- `unit(42)` is a `Process<Int>` that, when executed, produces 42
- If `fred : Void -> Int` is a function that, when called, produces an `Int` then
  - `exec(fred)` is a `Process<Int>` that, when executed, produces an `Int` by calling `fred`
- `exec(fred) > exec(ginger)`, when executed, produces the result of calling `ginger` after calling `fred`.

Processes produce results

In fact `Process` is parameterized by a type.

A `Process<Int>` object is a process that, when executed, outputs an `Int`

Examples

- `unit(42)` is a `Process<Int>` that, when executed, produces 42
- If `fred : Void -> Int` is a function that, when called, produces an `Int` then
  - `exec(fred)` is a `Process<Int>` that, when executed, produces an `Int` by calling `fred`
- `exec(fred) > exec(ginger)`, when executed, produces the result of calling `ginger` after calling `fred`.

Processes produce results

In fact `Process` is parameterized by a type.

A `Process<Int>` object is a process that, when executed, outputs an `Int`

Examples

- `unit(42)` is a `Process<Int>` that, when executed, produces 42
- If `fred : Void -> Int` is a function that, when called, produces an `Int` then
  - `exec(fred)` is a `Process<Int>` that, when executed, produces an `Int` by calling `fred`
- `exec(fred) > exec(ginger)`, when executed, produces the result of calling `ginger` after calling `fred`.

Processes produce results

In fact `Process` is parameterized by a type.

A `Process<Int>` object is a process that, when executed, outputs an `Int`

Examples

- `unit(42)` is a `Process<Int>` that, when executed, produces 42
- If `fred : Void -> Int` is a function that, when called, produces an `Int` then
  - `exec(fred)` is a `Process<Int>` that, when executed, produces an `Int` by calling `fred`
- `exec(fred) > exec(ginger)`, when executed, produces the result of calling `ginger` after calling `fred`.

Processes may take time

- `pause(1000)` is a `Process<Triv>` that takes 1000 milliseconds to complete execution.
  (`Triv` is a type that has only one value: `null`)
- `exec(fred) > pause(1000) > exec(ginger)`, when executed, produces the result of calling `ginger` 1 second after calling `fred`.

Processes may take time

- `pause(1000)` is a `Process<Triv>` that takes 1000 milliseconds to complete execution.
  (`Triv` is a type that has only one value: `null`)
- `exec(fred) > pause(1000) > exec(ginger)`, when executed, produces the result of calling `ginger` 1 second after calling `fred`.

Processes may take time

- `pause(1000)` is a `Process<Triv>` that takes 1000 milliseconds to complete execution.
  (`Triv` is a type that has only one value: `null`)
- `exec(fred) > pause(1000) > exec(ginger)`, when executed, produces the result of calling `ginger` 1 second after calling `fred`.

Loops

If `p` is a `Process<A>` then `loop(p)` is `(p>p>p>...)`

Now we can implement our example

```
function main() {
    var p =
        loop  (
            clearText( nameBox ) >
            show( nameBox ) >
            show( question ) >
            getAndDisplayAnswer() > // Implement later.
            hide( question ) >
            hide( nameBox ) >
            pause( 1000 )  ) ;
    p.go( function(x:Triv){ } ) ; // Execute p
}
```

go

So what is this `go` method?

If

- `p : Process<A>`
- `k : A -> Void`, i.e. a function with no result.

`p.go(k)` starts execution of process `p` and ensures that (when and if the execution terminates) its output will be input to function `k`.

Examples

- If `p` is `unit(42)`, then `p.go(k)` $\equiv$ `k(42)`
- If `p` is `exec(fred)`, then `p.go(k)` $\equiv$ `k(fred())`
- if `p` is `exec(fred) > pause(1000) > exec(ginger)`, then `p.go(k)` calls `fred` and ensures that 1 second later `k(ginger())` is executed.

Implementing sequential composition

Suppose

- `p : Process<A>`
- `q : Process<B>`
- `k : B -> Void`

`p > q` is an object `r` where

`r.go(k)` $\equiv$ `p.go( function(x:A){ q.go(k); } )`

That's all there is to implementing sequential composition.

Bind

A useful variation on sequential composition is called *bind*

Suppose

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

then `p >= f : Process<B>`

`p >= f` is a process that, when executed,

- executes `p` to get a result `x`
- and then executes `f(x)`

`p >= f` is an object `r` where

`r.go(k)` $\equiv$ `p.go( function(x:A){ f(x).go(k);})`

Guards and Guarded Processes

A `Guard<E>` object represents a set of events.
If

- `g : Guard<E>`
- `p : Process<A>`

then `g && p : GuardedProcess<A>`
A `GuardedProcess<A>` represents an set of sequences of actions that can be triggered by a set of events.

Guards and Guarded Processes

Example

Suppose

- `enter(nameBox)` represents the event of the enter key in `nameBox`
- `getValue(nameBox) : Process<String>`

then

`enter(nameBox) && getValue(nameBox)` is a guarded process.

Await

If

- `g : Guard<E>`
- `p : Process<A>`

then `await( g && p ) : Process<A>`
Executing `await( g && p )`

- events enabled
- event happens
- events are disabled
- `p` executes.

Finishing the example

```
function main() {
    var p = ... >
        loop (
            ... >
            getAndDisplayAnswer() >
            ... ) ;
    p.go( function(x:Triv){} )  ; }

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

function hello( name : String ) { return
    putText( reply, "Hello "+name ) ; }
```

Choices

We can make a choice between guarded processes

```
await( gp || gq )
```

Await can wait for any of a number of events:

```
await(
        enter( nameBox ) && getValue(nameBox) >= hello
    ||
        timeout(5000) && flash( question ) > invoke( top )
)
```

Fixed points

We can create loops that allow exits by using a fixed-point operator:

```
function getAndDisplayAnswer( ) : Process<Triv> {
    function f( top : Void -> Process<Triv> ) { return
      await(
        enter( nameBox ) && getValue(nameBox) >= hello
      ||
        timeout(5000) && flash(question) > invoke(top)
      ) ; }
    return fix( f ) ;
}
```

Conclusion

Question:

Can we write pull code in a pushy world?

Answer:

Yes, by using

- A suitable set of combinators
- inspired by context free grammars,
- process algebras, and
- monads
- to build structures representing
- sets of sequences of actions and events.

Thanks.

Conclusion

Question:

Can we write pull code in a pushy world?

Answer:

Yes, by using

- A suitable set of combinators
- inspired by context free grammars,
- process algebras, and
- monads
- to build structures representing
- sets of sequences of actions and events.

Thanks.

Conclusion

Question:

Can we write pull code in a pushy world?

Answer:

Yes, by using

- A suitable set of combinators
- inspired by context free grammars,
- process algebras, and
- monads
- to build structures representing
- sets of sequences of actions and events.

Thanks.