# Allocatio-Free Async Workflows

## Motivation

Consider the following async workflow:

let x = async {

for i in 1..1000 do

let! j = foo(i)

doSomething (i + j)

}

Under the current translation, the above becomes, approximately:

async.For(seq {1..1000},

fun i ->

async.Bind(foo(i),

fun j -> doSomething(i+j); async.Zero())

)

Or, giving names to anonymous lambdas:

let forBody =

fun i ->

let letBang = fun j -> doSomething(i+j); async.Zero()

async.Bind(foo(i), letBang)

async.For(seq {1..1000}, forBody)

We can see, in forBody function, that every “for” iteration allocates a fresh closure, letBang. The reason for this is that letBang captures the value of i, and therefore apparently cannot be reused between “for” iterations.

However, since letBang represents a continuation for “foo(i)” we know that letBang will never be called more than once. Moreover, at any given time, for all values of i, only one letBang will be executed. Therefore we can reuse the letBang closure between different loop iteration if only we replace i with a mutable:

let i = ref Unchecked.defaultof<int>

let letBang = fun j-> doSomething(!i + j); async.Zero()

let forBody =

fun i' ->

i := i'

async.Bind(foo(!i), letBang)

async.For(seq {1..1000}, forBody)

Applying these ideas to async workflows in a systematic way leads us to implementation of async workflow as a state machine.

Additional motivation for this feature is debugging experience improvement (variables that appear to be in scope are all accessible in Watch window)

## Requirements

* FSharp.Core 4.5 is an in-place update for FSharp.Core 4.0, hence the state machine generated MUST be compatible with async workflows implementation in F# 2.0
* Execution of while/for loops where all inner let! has constants on their rhs MUST do constant (in the number of iterations) allocations:

async {

for i = 1 to n do

let! j = foo

doSomething (i+j)

}

* Executions of tail-recursive asyncs where all inner let! has constants on their rhs MUST do do constant (in the number of recursive calls) allocations:

let rec loop i =

async {

if i > 0 then

return ()

else

return! loop (i-1)

}

## The Proposal

### Example code generations

#### Simple Bind

For the code fragment

let simpleBind =

async {

let a = calcA()

let! b = getB a

return getResult a b

}

The first approximation of a generated state machine might look like this (Fig 1 contains the definitions of data types involved):

type ``simpleBind@generatedStates`` =

| INIT

| AFTER\_B

type ``simpleBind@generated`` (asyncParams : AsyncParams<unit>) =

let mutable state = INIT

let mutable a = Unchecked.defaultof<\_>

let mutable b = Unchecked.defaultof<\_>

let stateMachine=

fun () ->

match state with

| INIT ->

a <- calcA()

let (P getB\_Body) = getB a

tailcall getB\_Body b\_ContinuationRecord

| AFTER\_B ->

let ``$result`` = getResult a b

tailcall asyncParams.cont ``$result``

and b\_ContinuationRecord =

{ asyncParam with

cont = fun bResult ->

b <- bResult

state <- AFTER\_B

tailcall stateMachine () }

member this.Run() = tailcall stateMachine ()

let simpleBind : Async<\_> =

P (fun asyncParams ->

tailcall (new ``simpleBind@generated``(asyncParams)).Run ())

Every top-level let-bound variable becomes a mutable field in generated class. Every bind point (let!) begets a state and a preallocated “continuation record”. tailcall pseudo-keyword indicates tail calls. Code generation should replace direct tail calls with indirect ones using the trampoline.

The above example does not treat cancellation and exception handling. The full exception handling and cancellation checking code generation for this example is as follows:

type FakeUnitValue = | FakeUnit

type cont<'T> = ('T -> FakeUnitValue)

type econt = (exn -> FakeUnitValue)

type ccont = (OperationCanceledException -> FakeUnitValue)

type TrampolineHolder = class (\* ... \*) end

type AsyncParamsAux =

{ token : CancellationToken;

econt : econt;

ccont : ccont;

trampolineHolder : TrampolineHolder

}

type AsyncParams<'T> =

{ cont : cont<'T>

aux : AsyncParamsAux

}

type Async<'T> = P of (AsyncParams<'T> -> FakeUnitValue)

**Fig 1.** Definitions of async implementation-related data types

let stateMachine=

fun () ->

match state with

| INIT ->

if asyncParams.aux.token.IsCancellationRequested then

tailcall asyncParams.aux.ccont (OperationCanceledException())

else

let mutable exn = null

let getB\_Body =

try

a <- calcA()

let (P getB\_Body) = getB a

getB\_Body

with

| e -> exn <- e; Unchecked.defaultof<\_>

match exn with

| null -> tailcall getB\_Body b\_ContinuationRecord

| \_ -> tailcall asyncParams.aux.econt

| AFTER\_B ->

if asyncParams.aux.token.IsCancellationRequested then

tailcall asyncParams.aux.ccont (OperationCanceledException())

else

let mutable exn = null

let ``$result`` =

try

getResult a b

with

| e -> exn <- e; Unchecked.defaultof<\_>

match exn with

| null -> tailcall asyncParams.cont ``$result``

| \_ -> tailcall asyncParams.aux.econt exn

#### Generation for if..then..else

Generation of “if..then..else” where at least one of branches contains an async point introduces states for “then” branch, “else” branch, and “after if” statement. The following example

let ``if`` =

async {

let x = calcX()

if condition x then

do! foo x

else

bar x

return getResult x

}

Translates into

type ``if@generatedStates`` =

| INIT

| CONDITION\_TRUE

| CONDITION\_FALSE

| AFTER\_IF

type ``if@generated``(asyncParams : AsyncParams<\_>) =

let mutable state = INIT

let mutable x = Unchecked.defaultof<\_>

let rec stateMachine () =

match state with

| INIT ->

let mutable exn = null

let conditionResult =

try

x <- calcX()

condition c

with

| e -> exn <- e; Unchecked.defaultof<bool>

match exn with

| null ->

if conditionResult then

state <- CONDITION\_TRUE

else

state <- CONDITION\_FALSE

tailcall stateMachine ()

| \_ -> tailcall asyncParams.aux.econt exn

| CONDITION\_TRUE ->

<< translation for do! with target state AFTER\_IF >>

| CONDITION\_FALSE ->

let mutable exn - null

try

bar x

with

| e -> exn <- e

match exn with

| null -> state <- AFTER\_IF; tailcall stateMachine ()

| \_ -> tailcall asyncParams.aux.econt exn

| AFTER\_IF ->

<< translation for return >>

...

Note: this example converts state transitions not involving async points into tailcalls to stateMachine. It is possible to replace it with a while loop around a match in stateMachine function, along the lines of:

let rec stateMachine()

let mutable continue = true

while true do

match state with

<< all state transtions not involving async jumps go here >>

match state with

<< all state transtions involving async jumps go here

to preserve tailcall property >>

#### Recursive loop

Simple recursive loops are easy to detect. The recursive “return!” translates into mutation of parameters, setting the state to INIT and tailcalling the stateMachine():

let rec loop i =

async {

if i > 0 then

return ()

else

return! loop (i-1)

}

becomes

type ``loop@generatedStates``() =

| INIT

| CONDITION\_TRUE

| CONDITION\_FALSE

| AFTER\_IF

type ``loop@generated``(i : int; asyncParams : AsyncParams<unit>)

let mutable i = i

let mutable state = INIT

let rec stateMachine () =

match state with

| INIT ->

if i > 0 then

state <- CONDITION\_TRUE

else

state <- CONDITION\_FALSE

tailcall stateMachine ()

| CONDITION\_TRUE ->

state <- AFTER\_IF

tailcall stateMachine()

| CONDITION\_FALSE ->

let tempI = i - 1

i <- tempI

state <- INIT

tailcall stateMachine()

| AFTER\_IF ->

tailcall asyncParams.cont ()

member this.Run () = stateMahcine()

let loop i = P (fun p -> tailcall (new ``loop@generated``(i,p)).Run) ()

Note that in INIT state the values of variables representing top-level lets are irrelevant and are guaranteed to be rewritten.

#### For and While loops

The translation for for and while loops follows straightforwardly from the translation of bind, if and recursive loops.

#### Abnormal execution handling – try .. with, try..finally and TryCanceled

Translation of exception handling is quite straightforward since the list of exception handlers is known statically at translation time (but see below the recursive loop case). Every “with” block begets a execption continuation that is called on execption handling and passed through as a part of continuation record, and an “after try block” state in which the state machine resumes if the exception was handled. The following async workflow:

let tryWith =

async {

try

do! insideTryBlock()

with

| :? MyException -> handleMyException()

do! outsideTryBlock()

}

translates into

type ``tryWith@generatedStates`` =

| INIT

| AFTER\_INSIDE\_TRY\_BLOCK

| AFTER\_TRY\_BLOCK

| AFTER\_OUTSIDE\_TRY\_BLOCK

type ``tryWith@generated``(asyncParams : AsyncParams) =

let mutable state = INIT

let rec stateMachine = fun () ->

match state with

| INIT ->

// enter the try block immediately

let mutable exn = null

let insideTryBlockBody =

try

let (P body) = insideTryBlock()

body

with

| e -> exn <- e

match exn with

| null ->

tailcall insideTryBlockBody insideTryBlock\_ContinuationRecord

| \_ ->

tailcall insideTryBlockEcont exn

| AFTER\_TRY\_BLOCK ->

let mutable exn = null

let outsideTryBlockBody =

try

try

let (P body) = outsideTryBlock()

body

with

| e -> exn <- e

match exn with

| null ->

tailcall

outsideTryBlockBody outsideTryBlock\_ContinuationRecord

| \_ ->

tailcall asyncParams.aux.econt exn

| AFTER\_INSIDE\_TRY\_BLOCK ->

state <- AFTER\_TRY\_BLOCK; tailcall stateMachine ()

| AFTER\_OUTSIDE\_TRY\_BLOCK ->

tailcall asyncParams.cont ()

and insideTryBlockEcont = fun exn ->

match exn with

| :? MyException ->

handleMyException()

state <- AFTER\_TRY\_BLOCK

tailcall stateMachine ()

and insideTryBlock\_ContinuationRecord =

{ cont = fun () ->

state <- AFTER\_INSIDE\_TRY\_BLOCK

tailcall stateMachine ()

aux = { asyncParam.aux with econt = insideTryBlockEcont

}

and outsideTryBlock\_ContinuationRecord =

{ asyncParams with

cont = fun () ->

state <- AFTER\_OUTSIDE\_TRY\_BLOCK

tailcall stateMachine ()

}

Note how exception handling and exception continuations change inside and outside “try .. with” block.

This scheme extends easily for TryCancelled and “try..finally”. In the latter case compensation code is duplicated between normal state transition and exception continuation.

#### Recursive loops within try..with and friends

The translation as it is above would break for recursive loops inside “try..with” and friends. Recall that the “return!” transition of the below:

let rec loop i =

async {

try

if i > 0 then

return ()

else

return! loop (i-1)

with

| :? MyException -> handler()

}

looks like this:

let tempI = i - 1

i <- tempI

state <- INIT

tailcall stateMachine ()

The recursive invocation of stateMachine knows nothing about the exception handler and will continue to call asyncParams.aux.econt.

The easy solution to that is making asyncParams a mutable field of the class and updating it with a new exception handler before recursive call to stateMachine.

### Formal Translation Specification

[TBD – but follows straightforwardly from the above examples]

## Design issues

1. State machines increase lifetime of let-bound objects inside the async. Is it a problem?
2. Detection of mutually recursive loops. Shall we generate a single state machine for mutually-recursive asyncs?
3. Shall we do state machine inlining?
4. We are considering asyncSeq for dev11.
   1. If we implement asyncSeq in terms of async as designed here, will it give us acceptable performance
      1. Hunch: yes if we implement inlining, no otherwise
   2. If the perf is inadequate, shall we implement the same translation for asyncSeq (looks like an easy extension)
5. [Tough] This design only works for async workflows. How to extend it for arbitrary workflows? (litmus test: seq and asyncSeq must be implementable)