Hopac Library Reference

Synopsis

```
namespace Hopac


type Void


type IAsyncDisposable =
  abstract DisposeAsync: unit -> Job<unit>


type Job<'x>
module Job =
  // Spawning jobs
  val queue:       Job<unit> -> Job<unit>
  val queueIgnore: Job<_>    -> Job<unit>
  val server:      Job<Void> -> Job<unit>
  val start:       Job<unit> -> Job<unit>
  val startIgnore: Job<_>    -> Job<unit>
  // Basic combinators
  val result: 'x   -> Job<'x>
  val   unit: unit -> Job<unit>
  val bind:      ('x   -> #Job<'y>) -> Job<'x> -> Job<'y>
  val delayWith: ('x   -> #Job<'y>) ->     'x  -> Job<'y>
  val map:       ('x   ->      'y)  -> Job<'x> -> Job<'y>
  val lift:      ('x   ->      'y)  ->     'x  -> Job<'y>
  val delay:     (unit -> #Job<'y>)            -> Job<'y>
  val thunk:     (unit ->      'y)             -> Job<'y>
  val apply: Job<'x> -> Job<'x -> 'y> -> Job<'y>
  val join: Job<#Job<'x>> -> Job<'x>
  val abort: unit -> Job<_>
  val Ignore: Job<_> -> Job<unit>
  // Exception handling
  val raises: exn -> Job<_>
  val tryIn:                Job<'x>  -> ('x -> #Job<'y>) -> (exn -> #Job<'y>) -> Job<'y>
  val tryInDelay: (unit -> #Job<'x>) -> ('x -> #Job<'y>) -> (exn -> #Job<'y>) -> Job<'y>
  val tryWith:                Job<'x>  -> (exn -> #Job<'x>) -> Job<'x>
  val tryWithDelay: (unit -> #Job<'x>) -> (exn -> #Job<'x>) -> Job<'x>
  val tryFinallyJob:                Job<'x>  ->      Job<unit> -> Job<'x>
  val tryFinallyJobDelay: (unit -> #Job<'x>) ->      Job<unit> -> Job<'x>
  val tryFinallyFun:                Job<'x>  -> (unit -> unit) -> Job<'x>
  val tryFinallyFunDelay: (unit -> #Job<'x>) -> (unit -> unit) -> Job<'x>
  val catch: Job<'x> -> Job<Choice<'x, exn>>
  // Finalization
  val useIn:      ('x -> #Job<'y>) -> 'x -> Job<'y> when 'x :> IDisposable
  val using:      'x -> ('x -> #Job<'y>) -> Job<'y> when 'x :> IDisposable
  val usingAsync: 'x -> ('x -> #Job<'y>) -> Job<'y> when 'x :> IAsyncDisposable
  // Repeating an operation
  val forN:       int -> Job<unit> -> Job<unit>
  val forNIgnore: int -> Job<_>    -> Job<unit>
  // Iterating over a range
  val forUpTo:         int -> int -> (int -> #Job<unit>) -> Job<unit>
  val forUpToIgnore:   int -> int -> (int -> #Job<_>)    -> Job<unit>
  val forDownTo:       int -> int -> (int -> #Job<unit>) -> Job<unit>
  val forDownToIgnore: int -> int -> (int -> #Job<_>)    -> Job<unit>
  // Iterating conditionally
  val whileDo:       (unit -> bool) ->           Job<unit> -> Job<unit>
  val whileDoDelay:  (unit -> bool) -> (unit -> #Job<_>)   -> Job<unit>
  val whileDoIgnore: (unit -> bool) ->           Job<_>    -> Job<unit>
  // Conditional
  val whenDo: bool -> Job<unit> -> Job<unit>
  // Server loops
  val forever:       Job<unit> -> Job<_>
  val foreverIgnore: Job<_>    -> Job<_>
  val iterate: 'x -> ('x -> #Job<'x>) -> Job<_>
  // Spawning server loops
  val foreverServer: Job<unit> -> Job<unit>
  // Sequences of jobs

  val seqCollect: seq<#Job<'x>> -> Job<ResizeArray<'x>>
  val conCollect: seq<#Job<'x>> -> Job<ResizeArray<'x>>
  val seqIgnore: seq<#Job<_>> -> Job<unit>
  val conIgnore: seq<#Job<_>> -> Job<unit>

  // Interop

  val fromBeginEnd: doBegin: (AsyncCallback * obj -> IAsyncResult)
                 -> doEnd: (IAsyncResult -> 'x)
                 -> Job<'x>
  val fromEndBegin: doEnd: (IAsyncResult -> 'x)
                 -> doBegin: (AsyncCallback * obj -> IAsyncResult)
                 -> Job<'x>
  val fromContinuations: (('x -> unit) -> (exn -> unit) -> unit) -> Job<'x>
  val onThreadPool: (unit -> 'x) -> Job<'x>
  val fromAsync: Async<'x> -> Job<'x>
  val   toAsync: Job<'x> -> Async<'x>
  val bindAsync: ('x -> #Job<'y>) -> Async<'x> -> Job<'y>
  val fromTask:     (unit -> Task<'x>) -> Job<'x>
  val fromUnitTask: (unit -> Task)     -> Job<unit>
  val liftTask:     ('x -> Task<'y>) -> 'x -> Job<'y>
  val liftUnitTask: ('x -> Task)     -> 'x -> Job<unit>
  val awaitTask:     Task<'x> -> Job<'x>
  val awaitUnitTask: Task     -> Job<unit>
  val bindTask:     ('x   -> #Job<'y>) -> Task<'x> -> Job<'y>
  val bindUnitTask: (unit -> #Job<'y>) -> Task     -> Job<'y>

  // Debugging

  val paranoid: Job<'x> -> Job<'x>

  module Scheduler =
    val bind: (Scheduler -> #Job<'x>) -> Job<'x>
    val get: unit -> Job<Scheduler>
    val switchToWorker: unit -> Job<unit>
    val isolate: (unit -> 'x) -> Job<'x>

  module Random =
    val bind: (uint64 -> #Job<'x>) -> Job<'x>
    val  map: (uint64 ->      'x)  -> Job<'x>
    val get: unit -> Job<uint64>


type Alt<'x> =
  inherit Job<'x>

module Alt =

  // Basic alternatives

  val always: 'x   -> Alt<'x>
  val   unit: unit -> Alt<unit>
  val once: 'x -> Alt<'x>
  val never: unit -> Alt<'x>
  val  zero: unit -> Alt<unit>
  val Ignore: Alt<_> -> Alt<unit>

  // Before actions

  val prepareJob: (unit -> #Job<#Alt<'x>>) -> Alt<'x>
  val prepare:             Job<#Alt<'x>>  -> Alt<'x>
  val prepareFun: (unit ->      #Alt<'x>)  -> Alt<'x>

  // Negative acknowledgments

  val withNackJob: (Promise<unit> -> #Job<#Alt<'x>>) -> Alt<'x>
  val withNackFun: (Promise<unit> ->      #Alt<'x>)  -> Alt<'x>
  val wrapAbortJob:      Job<unit> -> Alt<'x> -> Alt<'x>
  val wrapAbortFun: (unit -> unit) -> Alt<'x> -> Alt<'x>

  // Choices

  val choose:   seq<#Alt<'x>> -> Alt<'x>
  val choosy: array<#Alt<'x>> -> Alt<'x>

  // Randomization

  val random: (uint64 -> #Alt<'x>) -> Alt<'x>
  val chooser:  seq<#Alt<'x>> -> Alt<'x>

  // After actions

  val afterJob: ('x -> #Job<'y>) -> Alt<'x> -> Alt<'y>
  val afterFun: ('x ->      'y)  -> Alt<'x> -> Alt<'y>

  // Exception handling

  val raises: exn -> Alt<_>
  val tryIn: Alt<'x> -> ('x -> #Job<'y>) -> (exn -> #Job<'y>) -> Alt<'y>
  val tryFinallyJob: Alt<'x> ->      Job<unit> -> Alt<'x>
  val tryFinallyFun: Alt<'x> -> (unit -> unit) -> Alt<'x>
  val fromBeginEnd: doBegin: (AsyncCallback * obj -> IAsyncResult)
                 -> doEnd: (IAsyncResult -> 'x)
                 -> doCancel: (IAsyncResult -> unit)
                 -> Alt<'x>
  val fromAsync: Async<'x> -> Alt<'x>
  val   toAsync: Alt<'x> -> Async<'x>
  val fromTask:     (CancellationToken -> Task<'x>) -> Alt<'x>
  val fromUnitTask: (CancellationToken -> Task)     -> Alt<unit>

  // Debugging

  val paranoid: Alt<'x> -> Alt<'x>


type Promise<'x> =
  inherit Alt<'x>
  new: unit    -> Promise<'x>
  new: Job<'x> -> Promise<'x>
  new:     'x  -> Promise<'x>
  new: exn     -> Promise<'x>

module Promise =
  val queue: Job<'x> -> Job<Promise<'x>>
  val start: Job<'x> -> Job<Promise<'x>>
  val read: Promise<'x> -> Alt<'x>

  module Now =
    val isFulfilled: Promise<'x> -> bool
    val get: Promise<'x> -> 'x


type Ch<'x> =
  inherit Alt<'x>
  new: unit -> Ch<'x>

module Ch =
  val give: Ch<'x> -> 'x -> Alt<unit>
  val send: Ch<'x> -> 'x -> Job<unit>
  val take: Ch<'x> -> Alt<'x>

  module Try =
    val give: Ch<'x> -> 'x -> Job<bool>
    val take: Ch<'x> -> Job<option<'x>>

  module Now =
    val send: Ch<'x> -> 'x -> unit


type IVar<'x> =
  inherit Promise<'x>
  new: unit -> IVar<'x>
  new: 'x   -> IVar<'x>
  new: exn  -> IVar<'x>

module IVar =
  val    fill: IVar<'x> -> 'x -> Job<unit>
  val tryFill: IVar<'x> -> 'x -> Job<unit>
  val    fillFailure: IVar<'x> -> exn -> Job<unit>
  val tryFillFailure: IVar<'x> -> exn -> Job<unit>
  val read: IVar<'x> -> Alt<'x>

  module Now =
    val isFull: IVar<'x> -> bool
    val get: IVar<'x> -> 'x


type Latch =
  inherit Alt<unit>
  new: int -> Latch

module Latch =

  // Await

  val await: Latch -> Alt<unit>

  // Higher-order interface

  val within: (Latch -> #Job<'x>) -> Job<'x>
  val holding: Latch -> Job<'x> -> Job<'x>
  val queue: Latch -> Job<unit> -> Job<unit>
  val queueAsPromise: Latch -> Job<'x> -> Job<Promise<'x>>

  // First-order interface

  val decrement: Latch -> Job<unit>

  module Now =
    val increment: Latch -> unit
  inherit Alt<'x>
  new: unit -> MVar<'x>
  new: 'x   -> MVar<'x>

module MVar =

  // Primitives

  val fill: MVar<'x> -> 'x -> Job<unit>
  val take: MVar<'x> -> Alt<'x>

  // Read

  val read: MVar<'x> -> Alt<'x>

  // Mutate

  val    mutateJob: ('x -> #Job<'x>) -> MVar<'x> -> Alt<unit>
  val tryMutateJob: ('x -> #Job<'x>) -> MVar<'x> -> Alt<unit>
  val    mutateFun: ('x ->      'x)  -> MVar<'x> -> Alt<unit>
  val tryMutateFun: ('x ->      'x)  -> MVar<'x> -> Alt<unit>

  // Modify

  val    modifyJob: ('x -> #Job<'x * 'y>) -> MVar<'x> -> Alt<'y>
  val tryModifyJob: ('x -> #Job<'x * 'y>) -> MVar<'x> -> Alt<'y>
  val    modifyFun: ('x ->      'x * 'y)  -> MVar<'x> -> Alt<'y>
  val tryModifyFun: ('x ->      'x * 'y)  -> MVar<'x> -> Alt<'y>


type BoundedMb<'x> =
  new: int -> BoundedMb<'x>

module BoundedMb =
  val put: BoundedMb<'x> -> 'x -> Alt<unit>
  val take: BoundedMb<'x> -> Alt<'x>


type Mailbox<'x> =
  inherit Alt<'x>
  new: unit -> Mailbox<'x>

module Mailbox =
  val send: Mailbox<'x> -> 'x -> Job<unit>
  val take: Mailbox<'x> -> Alt<'x>

  module Now =
    val send: Mailbox<'x> -> 'x -> unit


type Lock =
  new: unit -> Lock

module Lock =
  val duringJob: Lock ->      Job<'x> -> Job<'x>
  val duringFun: Lock -> (unit -> 'x) -> Job<'x>


module Extensions =

  module Array =
    val mapJob: ('x -> #Job<'y>) -> array<'x> -> Job<array<'y>>
    val iterJob:       ('x -> #Job<unit>) -> array<'x> -> Job<unit>
    val iterJobIgnore: ('x -> #Job<_>)    -> array<'x> -> Job<unit>

  module Seq =
    val foldJob:     ('x -> 'y -> #Job<'x>) -> 'x -> seq<'y> -> Job<'x>
    val foldFromJob: 'x -> ('x -> 'y -> #Job<'x>) -> seq<'y> -> Job<'x>
    val iterJob:       ('x -> #Job<unit>) -> seq<'x> -> Job<unit>
    val iterJobIgnore: ('x -> #Job<_>)    -> seq<'x> -> Job<unit>
    val mapJob: ('x -> #Job<'y>) -> seq<'x> -> Job<ResizeArray<'y>>

    module Con =
      val iterJob:       ('x -> #Job<unit>) -> seq<'x> -> Job<unit>
      val iterJobIgnore: ('x -> #Job<_>)    -> seq<'x> -> Job<unit>
      val mapJob: ('x -> #Job<'y>) -> seq<'x> -> Job<ResizeArray<'y>>

  module Async =

    [<AbstractClass>]
    type OnWithSchedulerBuilder =
      new: unit -> OnWithSchedulerBuilder
      abstract Scheduler: Scheduler
      abstract Context: SynchronizationContext
      member Bind:  Task<'x> * ('x -> Async<'y>) -> Async<'y>
      member Bind:   Job<'x> * ('x -> Async<'y>) -> Async<'y>
      member Bind: Async<'x> * ('x -> Async<'y>) -> Async<'y>
      member Combine: Async<unit> * Async<'x> -> Async<'x>
      member Delay: (unit -> Async<'x>) -> Async<'x>
      member For: seq<'x> * ('x -> Async<unit>) -> Async<unit>
      member Return: 'x -> Async<'x>
      member ReturnFrom:  Task<'x> -> Async<'x>
      member ReturnFrom: Async<'x> -> Async<'x>
      member TryFinally: Async<'x> * (unit -> unit) -> Async<'x>
      member TryWith: Async<'x> * (exn -> Async<'x>) -> Async<'x>
      member Using: 'x * ('x -> Async<'y>) -> Async<'y> when 'x :> IDisposable
      member While: (unit -> bool) * Async<unit> -> Async<unit>
      member Zero: unit -> Async<unit>
      member Run: Async<'x> -> Job<'x>

    module Global =
      val onMain: unit -> OnWithSchedulerBuilder

    val setMain: SynchronizationContext -> unit
    val getMain: unit -> SynchronizationContext

  val asyncOn: SynchronizationContext -> Scheduler -> Async.OnWithSchedulerBuilder

  type Task with
    static member startJob: Job<'x> -> Job<Task<'x>>

  exception OnCompleted

  type IObservable<'x> with
    member onceAltOn: SynchronizationContext -> Alt<'x>
    member onceAltOnMain: Alt<'x>
    member onceAlt: Alt<'x>


module Infixes =

  // Query-Reply

  val ( *<+->= ): Ch<'q> ->   (Ch<'r> -> Promise<unit> -> #Job<'q>) -> Alt<'r>
  val ( *<+->- ): Ch<'q> ->   (Ch<'r> -> Promise<unit> ->      'q)  -> Alt<'r>
  val ( *<-=>= ): Ch<'q> -> (IVar<'r>                  -> #Job<'q>) -> Alt<'r>
  val ( *<-=>- ): Ch<'q> -> (IVar<'r>                  ->      'q)  -> Alt<'r>
  val ( *<+=>= ): Ch<'q> -> (IVar<'r>                  -> #Job<'q>) -> Alt<'r>
  val ( *<+=>- ): Ch<'q> -> (IVar<'r>                  ->      'q)  -> Alt<'r>

  // Message passing

  val ( *<-  ):      Ch<'x> -> 'x  -> Alt<unit>
  val ( *<+  ):      Ch<'x> -> 'x  -> Job<unit>
  val ( *<=  ):    IVar<'x> -> 'x  -> Job<unit>
  val ( *<=! ):    IVar<'x> -> exn -> Job<unit>
  val ( *<<= ):    MVar<'x> -> 'x  -> Job<unit>
  val ( *<<+ ): Mailbox<'x> -> 'x  -> Job<unit>

  // After actions

  val ( ^=>  ): Alt<'x> -> ('x -> #Job<'y>) -> Alt<'y>
  val ( ^->  ): Alt<'x> -> ('x ->       'y)  -> Alt<'y>
  val ( ^=>. ): Alt<_>  ->         Job<'y>  -> Alt<'y>
  val ( ^->. ): Alt<_>  ->             'y   -> Alt<'y>
  val ( ^->! ): Alt<_>  ->             exn  -> Alt<_>

  // Choices

  val ( <|>  ): Alt<'x> -> Alt<'x> ->     Alt<'x>
  val ( <|>* ): Alt<'x> -> Alt<'x> -> Promise<'x>
  val ( <~>  ): Alt<'x> -> Alt<'x> ->     Alt<'x>
  val ( <~>* ): Alt<'x> -> Alt<'x> -> Promise<'x>

  // Sequencing

  val ( >>=   ): Job<'x> -> ('x -> #Job<'y>) ->     Job<'y>
  val ( >>=*  ): Job<'x> -> ('x -> #Job<'y>) -> Promise<'y>
  val ( >>-   ): Job<'x> -> ('x ->       'y)  ->     Job<'y>
  val ( >>-*  ): Job<'x> -> ('x ->       'y)  -> Promise<'y>
  val ( >>=.  ): Job<_>  ->         Job<'y>  ->     Job<'y>
  val ( >>=*. ): Job<_>  ->         Job<'y>  -> Promise<'y>
  val ( >>-.  ): Job<_>  ->             'y   ->     Job<'y>
  val ( >>-*. ): Job<_>  ->             'y   -> Promise<'y>
  val ( >>-!  ): Job<_>  ->             exn  ->     Job<_>
  val ( >>-*! ): Job<_>  ->             exn  -> Promise<_>

  // Composition

  val ( >=>   ): ('x -> #Job<'y>) -> ('y -> #Job<'z>) -> 'x ->     Job<'z>
  val ( >=>*  ): ('x -> #Job<'y>) -> ('y -> #Job<'z>) -> 'x -> Promise<'z>
  val ( >->   ): ('x -> #Job<'y>) -> ('y ->       'z)  -> 'x ->     Job<'z>
  val ( >->*  ): ('x -> #Job<'y>) -> ('y ->       'z)  -> 'x -> Promise<'z>
  val ( >=>.  ): ('x -> #Job<_>)  ->         Job<'z>  -> 'x ->     Job<'z>
  val ( >=>*. ): ('x -> #Job<_>)  ->         Job<'z>  -> 'x -> Promise<'z>
  val ( >->.  ): ('x -> #Job<_>)  ->             'z   -> 'x ->     Job<'z>
  val ( >->*. ): ('x -> #Job<_>)  ->             'z   -> 'x -> Promise<'z>
  val ( >->!  ): ('x -> #Job<_>)  ->             exn  -> 'x ->     Job<_>
  val ( >->*! ): ('x -> #Job<_>)  ->             exn  -> 'x -> Promise<_>

  // Pairing

  val ( <&> ): Job<'x> -> Job<'y> -> Job<'x * 'y>
  val ( <*> ): Job<'x> -> Job<'y> -> Job<'x * 'y>
  val ( <+> ): Alt<'x> -> Alt<'y> -> Alt<'x * 'y>
  inherit Alt<unit>

module Proc =

  // Spawning jobs with handles

  val queue:       Job<unit> -> Job<Proc>
  val queueIgnore: Job<_>    -> Job<Proc>
  val start:       Job<unit> -> Job<Proc>
  val startIgnore: Job<_>    -> Job<Proc>

  // Access to handle

  val bind: (Proc -> #Job<'x>) -> Job<'x>
  val  map: (Proc ->      'x)  -> Job<'x>
  val self: unit -> Job<Proc>

  // Joining

  val join: Proc -> Alt<unit>


type JobBuilder =
  new: unit -> JobBuilder
  member Bind: IObservable<'x> * ('x -> Job<'y>) -> Job<'y>
  member Bind:       Async<'x> * ('x -> Job<'y>) -> Job<'y>
  member Bind:        Task<'x> * ('x -> Job<'y>) -> Job<'y>
  member Bind:         Job<'x> * ('x -> Job<'y>) -> Job<'y>
  member Combine: Job<unit> * (unit -> Job<'x>) -> Job<'x>
  member Delay: (unit -> Job<'x>) -> (unit -> Job<'x>)
  member For: seq<'x> * ('x -> Job<unit>) -> Job<unit>
  member Return: 'x -> Job<'x>
  member ReturnFrom: IObservable<'x> -> Job<'x>
  member ReturnFrom:       Async<'x> -> Job<'x>
  member ReturnFrom:        Task<'x> -> Job<'x>
  member ReturnFrom:         Job<'x> -> Job<'x>
  member Run: (unit -> Job<'x>) -> Job<'x>
  member TryFinally: (unit -> Job<'x>) * (unit -> unit) -> Job<'x>
  member TryWith: (unit -> Job<'x>) * (exn -> Job<'x>) -> Job<'x>
  member Using: 'x * ('x -> Job<'y>) -> Job<'y> when 'x :> IDisposable
  member While: (unit -> bool) * (unit -> Job<unit>) -> Job<unit>
  member Zero: unit -> Job<unit>


type EmbeddedJob<'x> = struct
    val Job: Job<'x>
    new: Job<'x> -> EmbeddedJob<'x>
  end

type EmbeddedJobBuilder =
  inherit JobBuilder
  new: unit -> EmbeddedJobBuilder
  member Run: (unit -> Job<'x>) -> EmbeddedJob<'x>


type Scheduler

module Scheduler =

  type Create =
    {
      Foreground: option<bool>
      IdleHandler: option<Job<int>>
      MaxStackSize: option<int>
      NumWorkers: option<int>
      TopLevelHandler: option<exn -> Job<unit>>
    }
    static member Def: Create

  module Global =
    val setCreate: Create -> unit

  val create: Create -> Scheduler
  val queue:       Scheduler -> Job<unit> -> unit
  val queueIgnore: Scheduler -> Job<_>    -> unit
  val server:      Scheduler -> Job<Void> -> unit
  val start:       Scheduler -> Job<unit> -> unit
  val startIgnore: Scheduler -> Job<_>    -> unit
  val startWithActions: Scheduler -> (exn -> unit) -> ('x -> unit) -> Job<'x> -> unit
  val run: Scheduler -> Job<'x> -> 'x
  val wait: Scheduler -> unit
  val kill: Scheduler -> unit


module Stream =

  // Stream representation
    | Cons of Value: 'x * Next: Promise<Cons<'x>>
    | Nil
  type Stream<'x> = Promise<Cons<'x>>

  // Stream sources and variables

  type Src<'x>
  module Src =
    val create: unit -> Src<'x>
    val value: Src<'x> -> 'x  -> Job<unit>
    val error: Src<'x> -> exn -> Job<unit>
    val close: Src<'x>        -> Job<unit>
    val tap: Src<'x> -> Stream<'x>
  type Var<'x>
  module Var =
    val create: 'x -> Var<'x>
    val get: Var<'x> -> 'x
    val set: Var<'x> -> 'x -> Job<unit>
    val tap: Var<'x> -> Stream<'x>
  type MVar<'x>
  module MVar =
    val create: 'x -> MVar<'x>
    val get: MVar<'x> -> Job<'x>
    val set: MVar<'x> -> 'x -> Job<unit>
    val updateFun: MVar<'x> -> ('x ->      'x)  -> Job<unit>
    val updateJob: MVar<'x> -> ('x -> #Job<'x>) -> Job<unit>
    val tap: MVar<'x> -> Stream<'x>
  type Property<'x> =
    interface INotifyPropertyChanged
    new: 'x -> Property<'x>
    member Value: 'x with get, set
    member Tap: unit -> Stream<'x>

  // Constructing streams

  val   nil<'x> : Stream<'x>
  val never<'x> : Stream<'x>
  val cons: 'x -> Stream<'x> -> Stream<'x>
  val delay: (unit -> #Job<Cons<'x>>) -> Stream<'x>
  val  error: exn -> Stream<'x>
  val    one: 'x  -> Stream<'x>
  val repeat: 'x  -> Stream<'x>
  val ofSeq: seq<'x> -> Stream<'x>
  val once: Job<'x> -> Stream<'x>

  // Generating lazy streams

  val indefinitely: Job<'x> -> Stream<'x>
  val unfoldJob: ('s -> #Job<option<'x * 's>>) -> ('s -> Stream<'x>)
  val unfoldFun: ('s ->      option<'x * 's>)  -> ('s -> Stream<'x>)
  [<AbstractClass>]
  type GenerateFuns<'s, 'x> =
    new: unit -> GenerateFuns<'s, 'x>
    abstract  While: 's -> bool
    abstract   Next: 's -> 's
    abstract Select: 's -> 'x
  val generateFuns: 's -> GenerateFuns<'s, 'x> -> Stream<'x>
  val generateFun: initial: 's
                -> doWhile: ('s -> bool)
                -> doNext: ('s -> 's)
                -> doSelect: ('s -> 'x)
                -> Stream<'x>
  val iterateJob: ('x -> #Job<'x>) -> 'x -> Stream<'x>
  val iterateFun: ('x ->      'x)  -> 'x -> Stream<'x>

  // Generating streams based on time

  val afterDateTimeOffsets: Stream<DateTimeOffset> -> Stream<DateTimeOffset>
  val afterDateTimeOffset: DateTimeOffset -> Stream<DateTimeOffset>
  val afterTimeSpan: TimeSpan -> Stream<unit>

  // Sampling live streams

  val shift: timeout: Job<_> -> Stream<'x> -> Stream<'x>
  val combineLatest: Stream<'x> -> Stream<'y> -> Stream<'x * 'y>
  val debounce: timeout: Alt<_> -> Stream<'x> -> Stream<'x>
  val ignoreUntil: timeout: Job<_> -> Stream<'x> -> Stream<'x>
  val ignoreWhile: timeout: Job<_> -> Stream<'x> -> Stream<'x>
  val samplesBefore: ticks: Stream<_> -> Stream<'x> -> Stream<'x>
  val samplesAfter:  ticks: Stream<_> -> Stream<'x> -> Stream<'x>
  val skipUntil: Alt<_> -> Stream<'x> -> Stream<'x>
  val takeUntil: Alt<_> -> Stream<'x> -> Stream<'x>
  val takeAndSkipUntil: Alt<_> -> Stream<'x> -> Stream<'x> * Stream<'x>
```

```
[<AbstractClass>]
type KeepPrecedingFuns<'x, 'y> =
  new: unit -> KeepPrecedingFuns<'x, 'y>
  abstract First: 'x -> 'y
  abstract Next: 'y * 'x -> 'y

val keepPrecedingFuns: KeepPrecedingFuns<'x, 'y> -> Stream<'x> -> Stream<'y>
val keepPreceding: maxCount: int -> Stream<'x> -> Stream<Queue<'x>>
val keepPreceding1: Stream<'x> -> Stream<'x>
val keepFollowing1: Stream<'x> -> Stream<'x>

// Polling lazy streams

val  afterEach: timeout: Job<_> -> Stream<'x> -> Stream<'x>
val beforeEach: timeout: Job<_> -> Stream<'x> -> Stream<'x>
val  delayEach: timeout: Job<_> -> Stream<'x> -> Stream<'x>
val duringEach: timeout: Job<_> -> Stream<'x> -> Stream<'x>
val pullOn: ticks: Stream<_> -> Stream<'x> -> Stream<'x>
val zip: Stream<'x> -> Stream<'y> -> Stream<'x * 'y>
val zipWithFun: ('x -> 'y -> 'z) -> Stream<'x> -> Stream<'y> -> Stream<'z>

// Interop with observables

val ofObservableOn: subscribeOn: SynchronizationContext -> IObservable<'x> -> Stream<'x>
val ofObservableOnMain: IObservable<'x> -> Stream<'x>
val ofObservable: IObservable<'x> -> Stream<'x>
val toObservable: Stream<'x> -> IObservable<'x>

// Mapping values

val mapJob:                         ('x -> #Job<'y>) -> Stream<'x> -> Stream<'y>
val mapFun:                         ('x ->      'y)  -> Stream<'x> -> Stream<'y>
val mapPipelinedJob: degree: int -> ('x -> #Job<'y>) -> Stream<'x> -> Stream<'y>
val mapPipelinedFun: degree: int -> ('x ->      'y)  -> Stream<'x> -> Stream<'y>
val mapConst:                                   'y   -> Stream<'x> -> Stream<'y>
val mapIgnore:                                          Stream<'x> -> Stream<unit>

// Filtering by value

val chooseJob: ('x -> #Job<option<'y>>) -> Stream<'x> -> Stream<'y>
val chooseFun: ('x ->      option<'y>)  -> Stream<'x> -> Stream<'y>
val choose: Stream<option<'x>> -> Stream<'x>
val filterJob: ('x -> #Job<bool>) -> Stream<'x> -> Stream<'x>
val filterFun: ('x ->      bool)  -> Stream<'x> -> Stream<'x>

// Grouping by value

val groupByJob: ('k -> Job<unit> -> Stream<'x> -> #Job<'y>) -> ('x -> #Job<'k>) -> Stream<'x> -> Stream<'y> when 'k: equality
val groupByFun: ('k -> Job<unit> -> Stream<'x> ->      'y)  -> ('x ->      'k)  -> Stream<'x> -> Stream<'y> when 'k: equality

// Buffering values

val buffer: int -> Stream<'x> -> Stream<array<'x>>

// Accumulating state

val     scanJob: ('s -> 'x -> #Job<'s>) -> 's -> Stream<'x> -> Stream<'s>
val     scanFun: ('s -> 'x ->      's)  -> 's -> Stream<'x> -> Stream<'s>
val scanFromJob: 's -> ('s -> 'x -> #Job<'s>) -> Stream<'x> -> Stream<'s>
val scanFromFun: 's -> ('s -> 'x ->      's)  -> Stream<'x> -> Stream<'s>

// Skipping duplicates

val distinctByJob: ('x -> #Job<'k>) -> Stream<'x> -> Stream<'x> when 'k: equality
val distinctByFun: ('x ->      'k)  -> Stream<'x> -> Stream<'x> when 'k: equality
val distinctUntilChangedWithJob: ('x -> 'x -> #Job<bool>) -> Stream<'x> -> Stream<'x>
val distinctUntilChangedWithFun: ('x -> 'x ->      bool)  -> Stream<'x> -> Stream<'x>
val distinctUntilChangedByJob: ('x -> #Job<'k>) -> Stream<'x> -> Stream<'x> when 'k: equality
val distinctUntilChangedByFun: ('x ->      'k)  -> Stream<'x> -> Stream<'x> when 'k: equality
val distinctUntilChanged:                          Stream<'x> -> Stream<'x> when 'x: equality

// Skipping and taking by value or count

val skip: int64 -> Stream<'x> -> Stream<'x>
val take: int64 -> Stream<'x> -> Stream<'x>
val skipWhileJob: ('x -> #Job<bool>) -> Stream<'x> -> Stream<'x>
val skipWhileFun: ('x ->      bool)  -> Stream<'x> -> Stream<'x>
val takeWhileJob: ('x -> #Job<bool>) -> Stream<'x> -> Stream<'x>
val takeWhileFun: ('x ->      bool)  -> Stream<'x> -> Stream<'x>

// Exception handling

val catch: (exn -> #Stream<'x>) -> Stream<'x> -> Stream<'x>

// Finalization

val    onCloseJob:      Job<unit> -> Stream<'x> -> Stream<'x>
val    onCloseFun: (unit -> unit) -> Stream<'x> -> Stream<'x>
val doFinalizeJob:      Job<unit> -> Stream<'x> -> Stream<'x>
val doFinalizeFun: (unit -> unit) -> Stream<'x> -> Stream<'x>

// Reducing stream to a value

val count: Stream<'x> -> Job<int64>
val     foldJob: ('s -> 'x -> #Job<'s>) -> 's -> Stream<'x> -> Job<'s>
```

```
  val foldFromJob: 's -> ('s -> 'x -> #Job<'s>) -> Stream<'x> -> Job<'s>
  val foldFromFun: 's -> ('s -> 'x ->      's)  -> Stream<'x> -> Job<'s>
  val     foldBack: ('x -> Promise<'s> -> 'sJ) -> Stream<'x> -> 'sJ -> Promise<'s> when 'sJ :> Job<'s>
  val foldFromBack: 'sJ -> (Promise<'s> -> 'x -> 'sJ) -> Stream<'x> -> Promise<'s> when 'sJ :> Job<'s>
  val tryPickJob: ('x -> #Job<option<'y>>) -> Stream<'x> -> Job<option<'y>>
  val tryPickFun: ('x ->      option<'y>)  -> Stream<'x> -> Job<option<'y>>

  // Iterating over streams

  val iterJob: ('x -> #Job<unit>) -> Stream<'x> -> Job<unit>
  val iterFun: ('x ->      unit)  -> Stream<'x> -> Job<unit>
  val iter: Stream<'x> -> Job<unit>
  val consumeJob: ('x -> #Job<unit>) -> Stream<'x> -> unit
  val consumeFun: ('x ->      unit)  -> Stream<'x> -> unit
  val consume: Stream<'x> -> unit

  // Prefixes and suffixes

  val head: Stream<'x> -> Stream<'x>
  val tail: Stream<'x> -> Stream<'x>
  val init: Stream<'x> -> Stream<'x>
  val last: Stream<'x> -> Stream<'x>
  val inits: Stream<'x> -> Stream<Stream<'x>>
  val tails: Stream<'x> -> Stream<Stream<'x>>
  val initsMapFun: (Stream<'x> -> 'y) -> Stream<'x> -> Stream<'y>
  val tailsMapFun: (Stream<'x> -> 'y) -> Stream<'x> -> Stream<'y>

  // Joining lazy streams

  val append: Stream<'x> -> Stream<'x> -> Stream<'x>
  val appendAll: Stream<#Stream<'x>> -> Stream<'x>
  val appendMap: ('x -> #Stream<'y>) -> Stream<'x> -> Stream<'y>

  // Joining live streams

  val    amb: Stream<'x> -> Stream<'x> -> Stream<'x>
  val  merge: Stream<'x> -> Stream<'x> -> Stream<'x>
  val switch: Stream<'x> -> Stream<'x> -> Stream<'x>
  val switchTo: Stream<'x> -> Stream<'x> -> Stream<'x>
  val    ambAll: Stream<#Stream<'x>> -> Stream<'x>
  val  mergeAll: Stream<#Stream<'x>> -> Stream<'x>
  val switchAll: Stream<#Stream<'x>> -> Stream<'x>
  val    ambMap: ('x -> #Stream<'y>) -> Stream<'x> -> Stream<'y>
  val  mergeMap: ('x -> #Stream<'y>) -> Stream<'x> -> Stream<'y>
  val switchMap: ('x -> #Stream<'y>) -> Stream<'x> -> Stream<'y>

  // Stream computation expression builders

  [<AbstractClass>]
  type Builder =
    new: unit -> Builder
    abstract Combine': Alt<Cons<'x>> * Alt<Cons<'x>> -> Alt<Cons<'x>>
    abstract Zero: unit -> Stream<'x>
    member Combine: Stream<'x> * Stream<'x> -> Stream<'x>
    member Bind: Stream<'x> * ('x -> Stream<'y>) -> Stream<'y>
    member Delay: (unit -> Stream<'x>) -> Stream<'x>
    member For: seq<'x> * ('x -> Stream<'y>) -> Stream<'y>
    member TryWith: Stream<'x> * (exn -> Stream<'x>) -> Stream<'x>
    member While: (unit -> bool) * Stream<'x> -> Stream<'x>
    member Yield: 'x -> Stream<'x>
    member YieldFrom: Stream<'x> -> Stream<'x>

  val    ambed: Builder
  val appended: Builder
  val   merged: Builder
  val switched: Builder

  // Misc

  val cycle: Stream<'x> -> Stream<'x>
  val values: Stream<'x> -> Alt<'x>

  // Primitive combinators

  val    amb': Alt<Cons<'x>> -> Alt<Cons<'x>> -> Alt<Cons<'x>>
  val append': Alt<Cons<'x>> -> Alt<Cons<'x>> -> Alt<Cons<'x>>
  val  merge': Alt<Cons<'x>> -> Alt<Cons<'x>> -> Alt<Cons<'x>>
  val switch': Alt<Cons<'x>> -> Alt<Cons<'x>> -> Alt<Cons<'x>>
  val joinWith: ('y -> Alt<Cons<'z>> -> #Job<Cons<'z>>)                -> Stream<'y> -> Stream<'z>
  val  mapJoin: ('y -> Alt<Cons<'z>> -> #Job<Cons<'z>>) -> ('x -> 'y) -> Stream<'x> -> Stream<'z>

  // Testing

  val toSeq: Stream<'x> -> Job<ResizeArray<'x>>


[<AutoOpen>]
module Hopac =

  type Stream<'x> = Stream.Stream<'x>

  // Computation expression builders
```

```
val onMain: Extensions.Async.OnWithSchedulerBuilder

// Spawning jobs

val queue:                 Job<unit> -> unit
val queueIgnore:           Job<_>    -> unit
val queueDelay: (unit -> #Job<_>)    -> unit
val server:                Job<Void> -> unit
val start:                 Job<unit> -> unit
val startIgnore:           Job<_>    -> unit
val startDelay: (unit -> #Job<_>)    -> unit
val run:                Job<'x>  -> 'x
val runDelay: (unit -> #Job<'x>) -> 'x

// Interop

val queueAsTask: Job<'x> -> Task<'x>
val startAsTask: Job<'x> -> Task<'x>
val startWithActions: (exn -> unit) -> ('x -> unit) -> Job<'x> -> unit

// Timeouts

val timeOut:       TimeSpan -> Alt<unit>
val timeOutMillis: int      -> Alt<unit>
val idle:                      Alt<unit>

// Promises

val memo: Job<'x> -> Promise<'x>

// Type ascription helpers

val asAlt: Alt<'x> -> Alt<'x>
val asJob: Job<'x> -> Job<'x>
```

Description

```
namespace Hopac
```

Hopac is a library for F# with the aim of making it easier to write correct, modular and efficient parallel, asynchronous, concurrent and reactive programs. The design of Hopac draws inspiration from languages such as Concurrent ML and Cilk. Similar to Concurrent ML, Hopac provides message passing primitives and supports the construction of first-class synchronous abstractions, see `Alt<_>`. Parallel jobs (lightweight threads) in Hopac are created using techniques similar to the F# Async framework, see `Job<_>`. Hopac runs parallel jobs using a work distributing scheduler in a non-preemptive fashion. Hopac also includes an implementation of choice streams, see `Stream.Stream<_>`, that offers a simple approach to reactive programming.

Before you begin using Hopac, make sure that you have configured your F# interactive and your application to use server garbage collection. By default, .Net uses single-threaded workstation garbage collection, which makes it impossible for parallel programs to scale.

The documentation of many of the primitives contains a reference implementation. In most cases, actual implementations are optimized by taking advantage of internal implementation details and may be significantly faster than the reference implementation. The reference implementations are given for a number of reasons. First of all, they hopefully help to better understand the semantics of the primitives. In some cases, the reference implementations also demonstrate how you can interface Hopac with other systems without the need to extend the primitives of Hopac. The reference implementations can also be seen as examples of how various primitives can be used to implement more complex operations.

Here is a diagram of some of Hopac's types:

```
                         Job<_>
                           |
                         Alt<_>
                           |
   +-------+--------+----+-----+--------+--------+
   |       |        |          |        |        |
  Ch<_>  Latch   Mailbox<_>  MVar<_>  Proc   Promise<_>
                                                 |
                                               IVar<_>
```

```
type Void
```

A type that has no public constructors to indicate that a job or function does not return normally.

```
type IAsyncDisposable =
```

An experimental interface for asynchronously disposable resources. See also: `usingAsync`.

The point of this interface is that resources using jobs may not be easily synchronously disposable. Expressing the dispose operation as a job allows the operation to wait for parallel, asynchronous and concurrent operations to complete.

Note that simply calling `run (x.DisposeAsync ())` defeats the purpose of this interface, unless it is known that the call is not made from a Hopac worker thread and no communication is needed between that thread and the dispose job.

```
abstract DisposeAsync: unit -> Job<unit>
```

Returns a job that needs to be executed to dispose the resource. The returned job should wait until the resource is properly disposed.

```
override this.DisposeAsync () = this.DisposeFlag <- true ; Job.unit ()
```

A typical correct disposal pattern could look something like this:

```
override this.DisposeAsync () =
  IVar.tryFill requestDisposeIVar () >>=.
  completedDisposeIVar
```

The above first signals the server to dispose by filling a synchronous variable. This is non-blocking and does not leak resources. Then the above waits until the server has signaled that disposal is complete. If disposal has already been requested, the first operation does nothing. Note that two separate variables are used.

The server loop corresponding to the above could look like this:

```
let rec serverLoop ... =
  ...
  let disposeAlt () =
    requestDisposeIVar ^=> fun () ->
    ...
    completedDisposeIVar *<= ()
  ...
  ... <|> disposeAlt () <|> ...
```

In some cases it may be preferable to have the server loop take requests mainly from a single channel (or mailbox):

```
let rec serverLoop ... =
  requestCh >>= function
    ...
    | RequestDispose ->
      ...
      completedDisposeIVar *<= ()
    ...
```

In such a case, one can still use the above two variable disposal pattern by spawning a job that forwards the disposal request to the server request channel before the server loop is started:

```
requestDisposeIVar >>=. requestCh *<- RequestDispose |> start
```

Alternatively, it is usually acceptable to simply send an asynchronous dispose request to the server:

```
override this.DisposeAsync () =
  requestCh *<+ RequestDispose >>=.
  completedDisposeIVar
```

```
type Job<'x>
```

Represents a lightweight thread of execution.

Jobs are defined using expression builders like the `JobBuilder`, accessible via the `Hopac.job` binding, or using monadic combinators and can then be executed using e.g. `Hopac.run`.

For example, here is a function that creates a job that computes Fibonacci numbers:

```
let rec fib n = job {
  if n < 2L then
    return n
  else
    let! (x, y) = fib (n-2L) <*> fib (n-1L)
    return x+y
}
```

It can be run, for example, by using the global scheduler:

```
> fib 30L |> run ;;
val it : int = 832040L
```

If you ran the above above examples, you just did the equivalent of running roughly your first million parallel jobs using Hopac.

```
module Job =
```

Operations on jobs.

```
val queue: Job<unit> -> Job<unit>
```

Creates a job that schedules the given job to be run as a separate concurrent job. Use `Promise.queue` if you need to be able to get the result. See also: `Proc.queue`.

current job, while `queue` queues the new job. `start` has slightly lower overhead than `queue` and is likely to be faster in cases where the new job blocks immediately.

For best performance the choice of which operation to use, `start` or `queue`, depends on the critical path of your system. It is generally preferable to keep control in the job that is on the critical path.

```
val queueIgnore: Job<_> -> Job<unit>
```

Creates a job that schedules the given job to be run as a separate concurrent job. `queueIgnore xJ` is equivalent to `Job.Ignore xJ |> queue`.

```
val server: Job<Void> -> Job<unit>
```

Creates a job that immediately starts running the given job as a separate concurrent job like `start`, but the given job is known never to return normally, so the job can be spawned in an even more lightweight manner.

```
val start: Job<unit> -> Job<unit>
```

Creates a job that immediately starts running the given job as a separate concurrent job. Use `Promise.start` if you need to be able to get the result. Use `Job.server` if the job never returns normally. See also: `Job.queue`, `Proc.start`.

```
val startIgnore: Job<_> -> Job<unit>
```

Creates a job that immediately starts running the given job as a separate concurrent job. `startIgnore xJ` is equivalent to `Job.Ignore xJ |> start`.

```
val result: 'x -> Job<'x>
```

Creates a job with the given result. See also: `lift`, `thunk`, `unit`.

```
val unit: unit -> Job<unit>
```

Returns a job that does nothing and returns `()`. `unit ()` is an optimized version of `result ()`.

```
val bind: ('x -> #Job<'y>) -> Job<'x> -> Job<'y>
```

Creates a job that first runs the given job and then passes the result of that job to the given function to build another job which will then be run. This is the same as `>>=` with the arguments flipped.

```
val delayWith: ('x -> #Job<'y>) -> 'x -> Job<'y>
```

Creates a job that calls the given function with the given value to build a job that will then be run. `delayWith x2yJ x` is equivalent to `result x >>= x2yJ`.

```
val map: ('x -> 'y) -> Job<'x> -> Job<'y>
```

Creates a job that runs the given job and maps the result of the job with the given function. This is the same as `>>-` with the arguments flipped.

```
val lift: ('x -> 'y) -> 'x -> Job<'y>
```

Creates a job that calls the given function with the given value to compute the result of the job. `lift x2y x` is equivalent to `result x >>- x2y`. Note that `x2y x |> result` is not the same.

```
val delay: (unit -> #Job<'y>) -> Job<'y>
```

Creates a job that calls the given function to build a job that will then be run. `delay u2xJ` is equivalent to `result () >>= u2xJ`.

Use of `delay` is often essential for making sure that a job constructed with user-defined code properly captures side-effects performed in the user-defined code or that a job is not constructed too eagerly (e.g. traversing an entire data structure to build a very large job object). However, it is also the case that there is no need to wrap every constructed job with `delay` and avoiding unnecessary `delay` operations can improve performance.

```
val thunk: (unit -> 'y) -> Job<'y>
```

Creates a job that invokes the given thunk to compute the result of the job. `thunk u2x` is equivalent to `result () >>- u2x`.

```
val apply: Job<'x> -> Job<'x -> 'y> -> Job<'y>
```

`x2yJ |> apply xJ` is equivalent to `x2yJ >>= fun x2y -> xJ >>- x2y`.

```
val join: Job<#Job<'x>> -> Job<'x>
```

`join xJJ` is equivalent to `bind id xJJ`.

```
val abort: unit -> Job<_>
```

Note that when a job aborts, it is considered to be equivalent to having the job block indefinitely and the job will be garbage collected. This also means that the job neither returns succesfully nor fails with an exception. This can sometimes be just what you want. However, in order to execute clean-up operations implemented with `using` or `tryFinallyFun` or `tryFinallyJob`, the job must either return normally or raise an exception. In other words, do not use `abort` in such a case.

```
val Ignore: Job<_> -> Job<unit>
```

Creates a job like the given job except that the result of the job will be `()`. `Ignore xJ` is equivalent to `xJ >>- ignore`.

```
val raises: exn -> Job<_>
```

Creates a job that has the effect of raising the specified exception. `raises e` is equivalent to `Job.delayWith raise e`.

```
val tryIn: Job<'x> -> ('x -> #Job<'y>) -> (exn -> #Job<'y>) -> Job<'y>
```

Implements the `try-in-unless` exception handling construct for jobs. Both of the continuation jobs `'x -> Job<'y>`, for success, and `exn -> Job<'y>`, for failure, are invoked from a tail position. See also: `tryInDelay`.

Note that the workflow notation of F# does not support this operation. It only supports the `Job.tryWith` operation. `Job.tryIn` makes it easier to write exception handling code that has the desired tail-call properties.

```
val tryInDelay: (unit -> #Job<'x>) -> ('x -> #Job<'y>) -> (exn -> #Job<'y>) -> Job<'y>
```

Implements the `try-in-unless` exception handling construct for jobs. Both of the continuation jobs `'x -> Job<'y>`, for success, and `exn -> Job<'y>`, for failure, are invoked from a tail position. `tryInDelay u2xJ x2yJ e2yJ` is equivalent to `tryIn (delay u2xJ) x2yJ e2yJ`.

```
val tryWith: Job<'x> -> (exn -> #Job<'x>) -> Job<'x>
```

Implements the try-with exception handling construct for jobs.

Reference implementation:

```
let tryWith xJ e2xJ = tryIn xJ result e2xJ
```

```
val tryWithDelay: (unit -> #Job<'x>) -> (exn -> #Job<'x>) -> Job<'x>
```

Implements the try-with exception handling construct for jobs.

```
val tryFinallyJob: Job<'x> -> Job<unit> -> Job<'x>
```

Implements a variation of the `try-finally` exception handling construct for jobs. The given action, specified as a job, is executed after the job has been run, whether it fails or completes successfully.

Note that the workflow notation of F# does not support this operation. It only supports the weaker `tryFinallyFun` operation.

Reference implementation:

```
let tryFinallyJob xJ uJ =
  tryIn xJ
   <| fun x -> uJ >>-. x
   <| fun e -> uJ >>-! e
```

```
val tryFinallyJobDelay: (unit -> #Job<'x>) -> Job<unit> -> Job<'x>
```

Implements a variation of the `try-finally` exception handling construct for jobs. The given action, specified as a job, is executed after the job has been run, whether it fails or completes successfully.

```
val tryFinallyFun: Job<'x> -> (unit -> unit) -> Job<'x>
```

Implements a variation of the `try-finally` exception handling construct for jobs. The given action, specified as a function, is executed after the job has been run, whether it fails or completes successfully.

Reference implementation:

```
let tryFinallyFun xJ u2u =
  tryFinallyJob xJ
   <| thunk u2u
```

```
val tryFinallyFunDelay: (unit -> #Job<'x>) -> (unit -> unit) -> Job<'x>
```

Implements a variation of the `try-finally` exception handling construct for jobs. The given action, specified as a function, is executed after the job has been run, whether it fails or completes successfully.

```
val catch: Job<'x> -> Job<Choice<'x, exn>>
```

Reference implementation:

```
let catch xJ =
  tryIn xJ
   <| lift Choice1Of2
   <| lift Choice2Of2
```

```
val useIn: ('x -> #Job<'y>) -> 'x -> Job<'y> when 'x :> IDisposable
```

`useIn x2yJ x` is equivalent to `using x x2yJ` and can be more convenient to use in pipelines (i.e. `x |> useIn x2yJ`).

```
val using: 'x -> ('x -> #Job<'y>) -> Job<'y> when 'x :> IDisposable
```

Implements the `use` construct for jobs. The `Dispose` method of the given disposable object is called after running the job constructed with the disposable object. See also: `abort`, `usingAsync`.

Reference implementation:

```
let using (x: 'x when 'x :> IDisposable) x2yJ =
  tryFinallyFun (delayWith x2yJ x) (x :> IDisposable).Dispose
```

Note that the `Dispose` method is not called if the job aborts before returning from the scope of the `using` job. This is not a serious problem, because scoped disposal of managed resources is usually an optimization and unmanaged resources should already be cleaned up by finalizers. In cases where you need to ensure scoped disposal, make sure that the job does not abort before returning.

```
val usingAsync: 'x -> ('x -> #Job<'y>) -> Job<'y> when 'x :> IAsyncDisposable
```

Implements an experimental `use` like construct for asynchronously disposable resources. The `DisposeAsync` method of the asynchronously disposable resource is called to construct a job that is later used to dispose the resource after the constructed job returns. See also: `abort`, `using`.

Reference implementation:

```
let usingAsync (x: 'x when 'x :> IAsyncDisposable) x2yJ =
  tryFinallyJob <| delayWith x2yJ x
   <| x.DisposeAsync ()
```

```
val forN: int -> Job<unit> -> Job<unit>
```

Creates a job that runs the given job sequentially the given number of times.

Reference implementation:

```
let rec forN n uJ =
  if n > 0 then
    uJ >>= fun () -> forN (n - 1) uJ
  else
    Job.unit ()
```

```
val forNIgnore: int -> Job<_> -> Job<unit>
```

`forNIgnore n xJ` is equivalent to `Job.Ignore xJ |> forN n`.

```
val forUpTo: int -> int -> (int -> #Job<unit>) -> Job<unit>
```

`forUpTo lo hi i2uJ` creates a job that sequentially iterates from `lo` to `hi` (inclusive) and calls the given function to construct jobs that will be executed.

Reference implementation:

```
let rec forUpTo lo hi i2uJ =
  if lo <= hi then
    i2uJ lo >>= fun () -> forUpTo (lo + 1) hi i2uJ
  else
    Job.unit ()
```

Rationale: The reason for iterating over an inclusive range is to make this construct work like a `for ... to ... do ...` loop of the base F# language.

```
val forUpToIgnore: int -> int -> (int -> #Job<_>) -> Job<unit>
```

`forUpToIgnore lo hi i2xJ` is equivalent to `forUpTo lo hi (i2xJ >> Job.Ignore)`.

```
val forDownTo: int -> int -> (int -> #Job<unit>) -> Job<unit>
```

`forDownTo hi lo i2uJ` creates a job that sequentially iterates from `hi` to `lo` (inclusive) and calls the given function to construct jobs

Reference implementation:

```
let rec forDownTo hi lo i2uJ =
  if hi >= lo then
    i2uJ hi >>= fun () -> forDownTo (hi - 1) lo i2uJ
  else
    Job.unit ()
```

Rationale: The reason for iterating over an inclusive range is to make this construct work like a `for ... downto ... do ...` loop of the base F# language.

`val forDownToIgnore: int -> int -> (int -> #Job<_>) -> Job<unit>`

`forDownToIgnore hi lo i2xJ` is equivalent to `forDownTo hi lo (i2xJ >> Job.Ignore)`.

`val whileDo: (unit -> bool) -> Job<unit> -> Job<unit>`

`whileDo u2b uJ` creates a job that sequentially executes the `uJ` job as long as `u2b ()` returns `true`. See also: `whileDoDelay`.

Reference implementation:

```
let whileDo u2b uJ = Job.delay <| fun () ->
  let rec loop () =
    if u2b () then
      uJ >>= loop
    else
      Job.unit ()
  loop ()
```

`val whileDoDelay: (unit -> bool) -> (unit -> #Job<_>) -> Job<unit>`

`whileDoDelay u2b u2xJ` creates a job that sequentially constructs a job with `u2xJ` and executes it as long as `u2b ()` returns `true`.

`val whileDoIgnore: (unit -> bool) -> Job<_> -> Job<unit>`

`whileDoIgnore u2b xJ` creates a job that sequentially executes the `xJ` job as long as `u2b ()` returns `true`. `whileDoIgnore u2b xJ` is equivalent to `Job.Ignore xJ |> whileDo u2b`.

`val whenDo: bool -> Job<unit> -> Job<unit>`

`whenDo b uJ` is equivalent to `if b then uJ else Job.unit ()`.

`val forever: Job<unit> -> Job<_>`

Creates a job that repeats the given job indefinitely. See also: `foreverServer`, `iterate`.

It is a common programming pattern to use server jobs that loop indefinitely and communicate with clients via channels. When a job is blocked waiting for communication on one or more channels and the channels become garbage (no longer reachable by any other job) the job can be garbage collected as well.

Reference implementation:

```
let rec forever uJ = uJ >>= fun () -> forever uJ
```

`val foreverIgnore: Job<_> -> Job<_>`

`foreverIgnore xJ` is equivalent to `Job.Ignore xJ |> forever`.

`val iterate: 'x -> ('x -> #Job<'x>) -> Job<_>`

Creates a job that indefinitely iterates the given job constructor starting with the given value. See also: `iterateServer`, `forever`.

It is a common programming pattern to use server jobs that loop indefinitely and communicate with clients via channels. When a job is blocked waiting for communication on one or more channels and the channels become garbage (no longer reachable by any other job) the job can be garbage collected as well.

Reference implementation:

```
let rec iterate x x2xJ =
  x2xJ x >>= fun x -> iterate x x2xJ
```

`val foreverServer: Job<unit> -> Job<unit>`

Creates a job that starts a separate server job that repeats the given job indefinitely. `foreverServer xJ` is equivalent to `forever xJ |> server`.

Creates a job that starts a separate server job that indefinitely iterates the given job constructor starting with the given value. `iterateServer x x2xJ` is equivalent to `iterate x x2xJ |> server`.

```
val seqCollect: seq<#Job<'x>> -> Job<ResizeArray<'x>>
```

Creates a job that runs all of the jobs in sequence and returns a list of the results. See also: `seqIgnore`, `conCollect`, `Seq.mapJob`.

Reference implementation:

```
let seqCollect (xJs: seq<Job<'x>>) = Job.delay <| fun () ->
  let xs = ResizeArray<_> ()
  Job.using <| xJs.GetEnumerator () <| fun xJs ->
  Job.whileDoDelay xJs.MoveNext <| fun () ->
        xJs.Current >>- xs.Add
  >>-. xs
```

```
val conCollect: seq<#Job<'x>> -> Job<ResizeArray<'x>>
```

Creates a job that runs all of the jobs as separate concurrent jobs and returns a list of the results. See also: `conIgnore`, `seqCollect`, `Seq.Con.mapJob`.

Note that when multiple jobs raise exceptions, then the created job raises an `AggregateException`.

Note that this is not optimal for fine-grained parallel execution.

```
val seqIgnore: seq<#Job<_>> -> Job<unit>
```

Creates a job that runs all of the jobs in sequence. The results of the jobs are ignored. See also: `seqCollect`, `conIgnore`, `Seq.iterJob`.

Reference implementation:

```
let seqIgnore (uJs: seq<#Job<unit>>) = Job.delay <| fun () ->
  Job.using <| uJs.GetEnumerator () <| fun uJs ->
  Job.whileDoDelay uJs.MoveNext <| fun () ->
    uJs.Current
```

```
val conIgnore: seq<#Job<_>> -> Job<unit>
```

Creates a job that runs all of the jobs as separate concurrent jobs and then waits for all of the jobs to finish. The results of the jobs are ignored. See also: `conCollect`, `seqIgnore`, `Seq.Con.iterJob`.

Note that when multiple jobs raise exceptions, then the created job raises an `AggregateException`.

Note that this is not optimal for fine-grained parallel execution.

```
val fromBeginEnd: doBegin: (AsyncCallback * obj -> IAsyncResult)
               -> doEnd: (IAsyncResult -> 'x)
               -> Job<'x>
```

Creates a job that performs the asynchronous operation defined by the given pair of `doBegin` and `doEnd` operations. See also: `Alt.fromBeginEnd`.

Reference implementation:

```
let fromBeginEnd doBegin doEnd =
  Job.Scheduler.bind <| fun sr ->
  let xI = IVar ()
  doBegin <| AsyncCallback (fun ar ->
    Scheduler.start sr (try xI *<= doEnd ar with e -> xI *<=! e))
  |> ignore
  xI
```

```
val fromEndBegin: doEnd: (IAsyncResult -> 'x)
               -> doBegin: (AsyncCallback * obj -> IAsyncResult)
               -> Job<'x>
```

`fromEndBegin doEnd doBegin` is equivalent to `fromBeginEnd doBegin doEnd`.

```
val fromContinuations: (('x -> unit) -> (exn -> unit) -> unit) -> Job<'x>
```

Creates a job that starts an asynchronous operation by calling the given function with success and failure continuations of which exactly one must be called once.

```
val onThreadPool: (unit -> 'x) -> Job<'x>
```

Creates a job that queues the given thunk to execute on the system `ThreadPool` and then waits for the result of the thunk.

```
val fromAsync: Async<'x> -> Job<'x>
```

Creates a job that starts the given async operation and waits for it to complete. See also: `Alt.fromAsync`.

Note that the async operation is started on whichever thread (and synchronization context) the job happens to be executed on. Transfer the async operation explicitly, e.g. by using `Async.SwitchToContext`, to the desired context when necessary.

```
val toAsync: Job<'x> -> Async<'x>
```

Creates an async operation that starts the given job and waits for it to complete.

```
val bindAsync: ('x -> #Job<'y>) -> Async<'x> -> Job<'y>
```

`bindAsync x2yJ xA` is equivalent to `fromAsync xA >>= x2yJ`.

```
val fromTask: (unit -> Task<'x>) -> Job<'x>
```

Creates a job that calls the given function to start a task and waits for it to complete. See also: `Alt.fromTask`.

```
val fromUnitTask: (unit -> Task) -> Job<unit>
```

Creates a job that calls the given function to start a task and waits for it to complete. See also: `Alt.fromUnitTask`.

```
val liftTask: ('x -> Task<'y>) -> 'x -> Job<'y>
```

`liftTask x2yT` is equivalent to `fun x -> fromTask <| fun () -> x2yT x`.

```
val liftUnitTask: ('x -> Task) -> 'x -> Job<unit>
```

`liftUnitTask x2uT` is equivalent to `fun x -> fromUnitTask <| fun () -> x2uT x`.

```
val awaitTask: Task<'x> -> Job<'x>
```

Creates a job that waits for the given task to finish and then returns the result of the task. Note that this does not start the task. Make sure that the task is started correctly. See also: `fromTask`.

Reference implementation:

```
let awaitTask (xT: Task<'x>) =
  Job.Scheduler.bind <| fun sr ->
  let xI = IVar ()
  xT.ContinueWith (Action<Threading.Tasks.Task>(fun _ ->
    Scheduler.start sr (try xI *<= xT.Result with e -> xI *<=! e)))
  |> ignore
  xI
```

```
val awaitUnitTask: Task -> Job<unit>
```

Creates a job that waits until the given task finishes. Note that this does not start the task. Make sure that the task is started correctly. See also: `fromUnitTask`.

```
val bindTask: ('x -> #Job<'y>) -> Task<'x> -> Job<'y>
```

`bindTask x2yJ xT` is equivalent to `awaitTask xT >>= x2yJ`.

```
val bindUnitTask: (unit -> #Job<'y>) -> Task -> Job<'y>
```

`bindUnitTask u2xJ uT` is equivalent to `awaitUnitTask uT >>= u2xJ`.

```
val paranoid: Job<'x> -> Job<'x>
```

Given a job, creates a new job that behaves exactly like the given job, except that the new job obviously cannot be directly downcast to the underlying type of the given job. This operation is provided for debugging purposes. You can always break abstractions using reflection. See also: `Alt.paranoid`.

```
module Scheduler =
```

Operations for dealing with the scheduler.

```
val bind: (Scheduler -> #Job<'x>) -> Job<'x>
```

`bind s2xJ` creates a job that calls the given job constructor with the scheduler under which the job is being executed. `bind` allows interfacing Hopac with existing asynchronous operations that do not fall into a pattern that is already supported explicitly.

Hopac jobs are executed under a scheduler. In almost all cases the scheduler is the global scheduler, but Hopac also allows local schedulers to be created for special purposes. A job that is suspended for the duration of an external asynchronous operation

Suppose, for example, that some system provides an asynchronous operation with the following signature:

```
val opWithCallback: Input
                 -> onSuccess: (Output -> unit)
                 -> onFailure: (exn -> unit)
                 -> unit
```

We would like to wrap the asynchronous operation as a job with following signature:

```
val opAsJob: Input -> Job<Output>
```

This can be done by using a write once variable, which will be filled with the result of the operation, and using `bind` to capture the current scheduler:

```
let opAsJob input = Job.Scheduler.bind <| fun scheduler ->
  let resultIVar = IVar ()
  let handleWith fill result =
    fill resultIVar result |> Scheduler.start scheduler
  ioWithCallback input
   <| handleWith IVar.fill
   <| handleWith IVar.fillFailure
  resultIVar
```

Note that the `Scheduler.start` operation is used to explicitly start the fill operation on the captured scheduler.

There are other similar examples as reference implementations of various Hopac primitives. See, for example, the reference implementations of `fromBeginEnd` and `Job.awaitTask`.

```
val get: unit -> Job<Scheduler>
```

Returns a job that returns the scheduler under which the job is being run. `get ()` is equivalent to `bind result`.

```
val switchToWorker: unit -> Job<unit>
```

Returns a job that ensures that the immediately following operation will be executed on a Hopac worker thread.

```
val isolate: (unit -> 'x) -> Job<'x>
```

`isolate u2x` is like `thunk u2x`, but it is ensured that the blocking invocation of `u2x` does not prevent scheduling of other work.

```
module Random =
```

Operations on the built-in pseudo random number generator (PRNG) of Hopac.

Note that every actual Hopac worker thread has its own PRNG state and is initialized with a distinct seed. However, when you `Hopac.start` or `Hopac.run` jobs from some non worker thread, it is possible that successive executions generate the same sequence of numbers. In the extremely rare case that could be a problem, use `Hopac.queue` or `switchToWorker`.

```
val bind: (uint64 -> #Job<'x>) -> Job<'x>
```

`bind r2xJ` creates a job that calls the given job constructor with a pseudo random 64-bit unsigned integer.

```
val map: (uint64 -> 'x) -> Job<'x>
```

`map r2x` is equivalent to `bind (r2x >> result)`.

```
val get: unit -> Job<uint64>
```

Returns a job that generates a pseudo random 64-bit unsigned integer. `get ()` is equivalent to `bind result`.

```
type Alt<'x> =
```

Represents a first-class selective synchronous operation.

The inspiration for alternatives comes from the events of Concurrent ML. The term "alternative" was chosen, because the term "event" is already widely used in .Net.

Simpler forms of selective synchronization exists in various languages. For example, the occam language has an `alt` statement, the Go language has a `select` statement and Clojure's core.async has an `alt` function. In Hopac and Concurrent ML, selective synchronous operations are not limited to primitive message passing operations (see `Ch.give` and `Ch.take`), but are instead first-class values (see `choose`) and can be extended with user-defined code (see `afterJob` and `withNackJob`) allowing the encapsulation of concurrent protocols as selective synchronous operations.

The idea of alternatives is to allow one to introduce new selective synchronous operations to be used with non-determinic choice aka `choose`. Obviously, when you have a concurrent server that responds to some protocol, you don't have to perform the protocol as a selective synchronous operation. However, if you do encapsulate the protocol as a selective synchronous operation, you can then combine the operation with other selective synchronous operations. That is the essence of Hopac and CML.

may take advantage of idempotency, rendezvous and negative acknowledgments. Here are few rules of thumb:

- If you don't need to send arguments to the server, you can synchronize using a `take` operation on the server's reply channel. E.g. an operation to take an element from a concurrent buffer.

- If you don't need a result from the server, aside from acknowledgment that the operation has been performed, you can synchronize using a `give` operation on the server's request channel. E.g. an operation to remove a specified element from a concurrent bag.

- If you have an idempotent operation, you can use `prepareJob` to send the arguments and a write once variable to the server and then synchronize using a `read` operation on the write once variable for the reply. E.g. request to receive a timeout event.

- If you have a non-idempotent operation, you can use `withNackJob` to send the arguments, negative acknowledgment token and a channel to the server and then synchronize using a `take` operation on the channel for the reply. See `withNackJob` for an illustrative toy example.

```
inherit Job<'x>
```

`Alt<'x>` is a subtype of `Job<'x>`. You can use an alternative in any context that requires a job.

```
module Alt =
```

Operations on first-class synchronous operations or alternatives.

```
val always: 'x -> Alt<'x>
```

Creates an alternative that is always available and results in the given value.

Note that when there are alternatives immediately available in a choice, the first such alternative will be committed to.

```
val unit: unit -> Alt<unit>
```

Returns an alternative that is always available and results in the unit value. `unit ()` is an optimized version of `always ()`.

```
val once: 'x -> Alt<'x>
```

Returns an alternative that can be committed to once and that produces the given value.

`once` is basically an optimized version of

```
let once x =
  let xCh = Ch ()
  xCh *<+ x |> run
  paranoid xCh
```

```
val never: unit -> Alt<'x>
```

Creates an alternative that is never available.

Note that synchronizing on `never ()`, without other alternatives, is equivalent to performing `abort ()`.

```
val zero: unit -> Alt<unit>
```

Returns an alternative that is never available. `zero ()` is an optimized version of `never ()`.

```
val Ignore: Alt<_> -> Alt<unit>
```

`Ignore xA` is equivalent to `xA ^-> fun _ -> ()`.

```
val prepareJob: (unit -> #Job<#Alt<'x>>) -> Alt<'x>
```

Creates an alternative that is computed at instantiation time with the given job. See also: `*<-=>-`, `prepareFun`, `withNackJob`.

`prepareJob` allows client-server protocols that do not require the server to be notified when the client aborts the transaction to be encapsulated as selective operations. For example, the given job may create and send a request to a server and then return an alternative that waits for the server's reply.

Reference implementation:

```
let prepareJob u2xAJ = withNackJob (ignore >> u2xAJ)
```

Note that, like with `withNackJob`, it is essential to avoid blocking inside `prepareJob`.

```
val prepare: Job<#Alt<'x>> -> Alt<'x>
```

Creates an alternative that is computed at instantiation time with the given job. `prepare xAJ` is equivalent to `prepareJob <| fun () -> xAJ`.

```
val prepareFun: (unit -> #Alt<'x>) -> Alt<'x>
```

prepareFun is an optimized weaker form of prepareJob that can be used when no concurrent operations beyond the returned alternative are required by the encapsulated request protocol.

Reference implementation:

```
let prepareFun u2xA = prepareJob (u2xA >> result)
```

```
val withNackJob: (Promise<unit> -> #Job<#Alt<'x>>) -> Alt<'x>
```

Creates an alternative that is computed at instantiation time with the given job constructed with a negative acknowledgment alternative. See also: `*<+->-`, `withNackFun`, `prepareJob`.

`withNackJob` allows client-server protocols that do require the server to be notified when the client aborts the transaction to be encapsulated as selective operations. The negative acknowledgment alternative will be available in case some other instantiated alternative involved in the choice is committed to instead.

Like `prepare`, `withNackJob` is typically used to encapsulate the client side operation of a concurrent protocol. The client side operation typically constructs a request, containing the negative acknowledgment alternative, sends it to a server and then returns an alternative that waits for a rendezvous with the server. In case the client later commits to some other alternative, the negative acknowledgment token becomes available and the server can also abort the operation.

Here is a simple example of an operation encapsulated using `withNackJob`. The idea is that we have a server that maintains a counter. Clients can request the server to increment the counter by a specific amount and return the incremented counter value. We further want to make it so that in case the client does not commit to the operation, the counter in the server is not updated.

Here is the server communication channel and the server loop:

```
let counterServer : Ch<int * Promise<unit> * Ch<int>> =
  let reqCh = Ch ()
  server << Job.iterate 0 <| fun oldCounter ->
    reqCh >>= fun (n, nack, replyCh) ->
    let newCounter = oldCounter + n
    replyCh *<- newCounter ^->. newCounter <|>
    nack                   ^->. oldCounter
  reqCh
```

Note how the server tries to synchronize on either giving the new counter value to the client or the negative acknowledgment.

Here is the encapsulated client side operation:

```
let incrementBy n : Alt<int> = Alt.withNackJob <| fun nack ->
  let replyCh = Ch ()
  counterServer *<+ (n, nack, replyCh) >>-.
  replyCh
```

Note that the above can be expressed more concisely using `*<+->-`.

The client side operation just sends the negative acknowledgment to the server as a part of the request. It is essential that a synchronous rendezvous via a channel, rather than e.g. a write once variable, is used for the reply. It is also essential to avoid blocking inside `withNackJob`, which is why an asynchronous send is used inside the client side operation.

Note that if an alternative created with `withNackJob` is not instantiated, then no negative acknowledgment is created. For example, given an alternative of the form `always () <|> withNackJob (...)` the `withNackJob` alternative is never instantiated.

```
val withNackFun: (Promise<unit> -> #Alt<'x>) -> Alt<'x>
```

`withNackFun n2xA` is equivalent to `withNackJob (Job.lift n2xA)`.

```
val wrapAbortJob: Job<unit> -> Alt<'x> -> Alt<'x>
```

Returns a new alternative that that makes it so that the given job will be started as a separate concurrent job if the given alternative isn't the one being committed to. See also: `wrapAbortFun`, `withNackJob`.

`wrapAbortJob` and `withNackJob` have roughly equivalent expressive power and `wrapAbortJob` can be expressed in terms of `withNackJob`. Sometimes `wrapAbortJob` more directly fits the desired usage than `withNackJob` and should be preferred in those cases. In particular, consider using `wrapAbortJob`, when you have an alternative whose implementation is similar to the following reference implementation:

```
let wrapAbortJob (abortAct: Job<unit>) (evt: Alt<'x>) : Alt<'x> =
  Alt.withNackJob <| fun nack ->
  nack >>=. abortAct |> Job.start >>-.
  evt
```

Historical note: Originally Concurrent ML only provided a corresponding combinator named `wrapAbort`. Later Concurrent ML changed to provide only `withNack` as a primitive, because it is a better fit for most use cases, and `wrapAbort` could be expressed in terms of it. Racket only provides `withNack` and, under Racket's model, `withNack` cannot be expressed in terms of `wrapAbort`.

```
val wrapAbortFun: (unit -> unit) -> Alt<'x> -> Alt<'x>
```

`wrapAbortFun u2u xA` is equivalent to `wrapAbortJob (Job.thunk u2u) xA`.

Creates an alternative that is available when any one of the given alternatives is. See also: `choosy`, `<|>`.

Note that `choose []` is equivalent to `never ()`.

Reference implementation:

```
let choose xAs = Alt.prepareFun <| fun () ->
  Seq.foldBack (<|>) xAs <| never ()
```

Above, `Seq.foldBack` has the obvious meaning. Alternatively we could define `xA1 <|> xA2` to be equivalent to `choose [xA1; xA2]` and consider `choose` as primitive.

```
val choosy: array<#Alt<'x>> -> Alt<'x>
```

`choosy xAs` (read: choose array) is an optimized version of `choose xAs` when `xAs` is an array. Do not write `choosy (Seq.toArray xAs)` instead of `choose xAs` unless the resulting alternative is reused many times.

One dominating cost in .Net is memory allocations. To choose between various forms of non-determistic choice, the following low level implementation details may be of interest.

```
choosy [| ... |]
```

Creation: 1 array + 1 object. Use: 1 object. Total cost: 3 allocations.

```
xA1 <|> xA2
```

Creation: 1 object. Use: 1 object. Total cost: 2 allocations.

```
xA1 <|> xA2 <|> xA3
```

Creation: 2 objects. Use: 2 objects. Total cost: 4 allocations.

If you are choosy, then when choosing between 2 or 3 alternatives, `<|>` is likely to be fastest. When choosing between 4 or more alternatives, `choosy` is likely to be fastest.

```
val random: (uint64 -> #Alt<'x>) -> Alt<'x>
```

Creates an alternative that is computed at instantiation time with the the given function, which will be called with a pseudo random 64-bit unsigned integer. See also: `Random.bind`.

```
val chooser: seq<#Alt<'x>> -> Alt<'x>
```

`chooser xAs` is like `choose xAs` except that the order in which the alternatives from the sequence are instantiated will be determined at random each time the alternative is used. See also: `<~>`.

Note that randomization only applies to the instantiation order. It makes no difference after instantiation.

```
val afterJob: ('x -> #Job<'y>) -> Alt<'x> -> Alt<'y>
```

Creates an alternative whose result is passed to the given job constructor and processed with the resulting job after the given alternative has been committed to. This is the same as `^=>` with the arguments flipped.

Note that although this operator has a type somewhat similar to a monadic bind operation, alternatives do not form a monad (with the `always` alternative constructor). So called Transactional Events do form a monad, but require a more complex synchronization protocol.

```
val afterFun: ('x -> 'y) -> Alt<'x> -> Alt<'y>
```

`xA |> afterFun x2y` is equivalent to `xA |> afterJob (x2y >> result)`. This is the same as `^->` with the arguments flipped.

```
val raises: exn -> Alt<_>
```

Creates an alternative that has the effect of raising the specified exception. `raises e` is equivalent to `prepareFun <| fun () -> raise e`.

```
val tryIn: Alt<'x> -> ('x -> #Job<'y>) -> (exn -> #Job<'y>) -> Alt<'y>
```

Implements the `try-in-unless` exception handling construct for alternatives. Both of the continuation jobs `'x -> Job<'y>`, for success, and `exn -> Job<'y>`, for failure, are invoked from a tail position.

Exceptions from both before and after the commit point can be handled. An exception that occurs before a commit point, from the user code in a `prepareJob`, or `withNackJob`, results in treating that exception as the commit point.

Note you can also use function or job level exception handling before the commit point within the user code in a `prepareJob` or `withNackJob`.

```
val tryFinallyJob: Alt<'x> -> Job<unit> -> Alt<'x>
```

cuted after the alternative has been committed to, whether the alternative fails or completes successfully. Note that the action is not executed in case the alternative is not committed to. Use `withNackJob` to attach the action to the non-committed case.

Reference implementation:

```
let tryFinallyJob xA uJ =
  tryIn xA
   <| fun x -> uJ >>-. x
   <| fun e -> uJ >>-! e
```

```
val tryFinallyFun: Alt<'x> -> (unit -> unit) -> Alt<'x>
```

Implements a variation of the `try-finally` exception handling construct for alternatives. The given action, specified as a function, is executed after the alternative has been committed to, whether the alternative fails or completes successfully. Note that the action is not executed in case the alternative is not committed to. Use `withNackJob` to attach the action to the non-committed case.

Reference implementation:

```
let tryFinallyFun xA u2u =
  tryFinallyJob xA
   <| Job.thunk u2u
```

```
val fromBeginEnd: doBegin: (AsyncCallback * obj -> IAsyncResult)
               -> doEnd: (IAsyncResult -> 'x)
               -> doCancel: (IAsyncResult -> unit)
               -> Alt<'x>
```

Creates an alternative that, when instantiated, starts the cancellable asynchronous operation defined by the given `doBegin`, `doEnd` and `doCancel` operations and waits for it to complete, after which the alternative becomes available. If some other alternative is committed to a in a choice before the operation completes, then the operation is cancelled. See also: `Job.fromBeginEnd`.

```
val fromAsync: Async<'x> -> Alt<'x>
```

Creates an alternative that, when instantiated, starts the given cancellable async operation and waits for it to complete, after which the alternative becomes available. If some other alternative is committed to in a choice before the operation completes, then the operation is cancelled. See also: `Job.fromAsync`.

Note that the async operation is started on whichever thread (and synchronization context) the job happens to be executed on. Transfer the async operation explicitly, e.g. by using `Async.SwitchToContext`, to the desired context when necessary.

```
val toAsync: Alt<'x> -> Async<'x>
```

Creates an async operation that starts the given alternative and waits for it to be committed to. If the async operation is cancelled before the alternative is committed to, an attempt is made to also cancel the alternative by making a cancellation alternative available. Note that cancellation is not transactional and `Alt.toAsync >> Alt.fromAsync` is not the identity function. See also: `Job.toAsync`.

```
val fromTask: (CancellationToken -> Task<'x>) -> Alt<'x>
```

Creates an alternative that, when instantiated, calls the given function with a cancellation token to start a cancellable task and waits for it to complete, after which the alternative becomes available. If some other alternative is committed to in a choice before the task completes, then the token will be cancelled. See also: `Job.fromTask`.

```
val fromUnitTask: (CancellationToken -> Task) -> Alt<unit>
```

Creates an alternative that, when instantiated, calls the given function with a cancellation token to start a cancellable task and waits for it to complete, after which the alternative becomes available. If some other alternative is committed to in a choice before the task completes, then the token will be cancelled. See also: `Job.fromUnitTask`.

```
val paranoid: Alt<'x> -> Alt<'x>
```

Given an alternative, creates a new alternative that behaves exactly like the given alternative, except that the new alternative obviously cannot be directly downcast to the underlying type of the given alternative. This operation is provided for debugging purposes. You can always break abstractions using reflection. See also: `Job.paranoid`.

```
type Promise<'x> =
```

Represents a promise to produce a result at some point in the future.

Promises are used when a parallel job is started for the purpose of computing a result. When multiple parallel jobs need to be started to compute results in parallel in regular patterns, combinators such as `<*>`, `Job.conCollect` and `Seq.Con.mapJob` may be easier to use and provide improved performance.

```
inherit Alt<'x>
```

`Promise<'x>` is a subtype of `Alt<'x>` and `xPr :> Alt<'x>` is equivalent to `Promise.read xPr`.

```
new: unit -> Promise<'x>
```

`new: Job<'x> -> Promise<'x>`

Creates a promise whose value is computed lazily with the given job when an attempt is made to read the promise. Although the job is not started immediately, the effect is that the delayed job will be run as a separate job, which means it is possible to communicate with it as long the delayed job is started before trying to communicate with it. See also: `memo`.

`new: 'x -> Promise<'x>`

Creates a promise with the given value.

`new: exn -> Promise<'x>`

Creates a promise with the given failure exception.

`module Promise =`

Operations on promises.

`val queue: Job<'x> -> Job<Promise<'x>>`

Creates a job that creates a promise, whose value is computed with the given job, which is scheduled to be run as a separate concurrent job. See also: `start`, `Job.queue`.

`val start: Job<'x> -> Job<Promise<'x>>`

Creates a job that creates a promise, whose value is computed with the given job, which is immediately started to run as a separate concurrent job. See also: `queue`, `Job.queue`.

`val read: Promise<'x> -> Alt<'x>`

Creates an alternative for reading the promise. If the promise was delayed, it is started as a separate job.

`module Now =`

Immediate or non-workflow operations on promises.

`val isFulfilled: Promise<'x> -> bool`

Returns true iff the given promise has already been fulfilled (either with a value or with a failure).

This operation is mainly provided for advanced uses of promises such as when creating more complex data structures that make internal use of promises. Using this to poll promises is not generally a good idea.

`val get: Promise<'x> -> 'x`

Returns the value or raises the failure exception that the promise has been fulfilled with. It is considered an error if the promise has not yet been fulfilled.

This operation is mainly provided for advanced uses of promises such as when creating more complex data structures that make internal use of promises. Using this to poll promises is not generally a good idea.

`type Ch<'x> =`

Represents a synchronous channel.

Channels provide a simple rendezvous mechanism for concurrent jobs and are designed to be used as the building blocks of selective synchronous abstractions.

Channels are lightweight objects and it is common to allocate fresh channels for short-term, possibly even one-shot, communications. When simple rendezvous is not needed in a one-shot communication, a write once variable, `IVar`, may offer slightly better performance.

Channels are optimized for synchronous message passing, which can often be done without buffering. Channels also provide an asynchronous `Ch.send` operation, but in situations where buffering is needed, some other message passing mechanism such as a bounded mailbox, `BoundedMb<_>`, or unbounded mailbox, `Mailbox<_>`, may be preferable.

`inherit Alt<'x>`

`Ch<'x>` is a subtype of `Alt<'x>` and `xCh :> Alt<'x>` is equivalent to `Ch.take xCh`.

`new: unit -> Ch<'x>`

Creates a new channel.

`module Ch =`

Operations on synchronous channels.

Creates an alternative that, at instantiation time, offers to give the given value on the given channel, and becomes available when another job offers to take the value. See also: `*<-`.

`val send: Ch<'x> -> 'x -> Job<unit>`

Creates a job that sends a value to another job on the given channel. A send operation is asynchronous. In other words, a send operation does not wait for another job to give the value to.

Note that channels have been optimized for synchronous operations; an occasional send can be efficient, but when sends are queued, performance maybe be significantly worse than with a `Mailbox` optimized for buffering. See also: `*<+`.

`val take: Ch<'x> -> Alt<'x>`

Creates an alternative that, at instantiation time, offers to take a value from another job on the given channel, and becomes available when another job offers to give a value.

`module Try =`

Polling, or non-blocking, operations on synchronous channels.

Note that polling operations only make sense when the other side of the communication is blocked waiting on the channel. If both a giver and a taker use polling operations on a channel, it is not guaranteed that communication will ever happen.

Also note that a job that performs arbitrarily many polling operations without blocking should not be used in a cooperative system, like Hopac, because such a job completely uses up a single core and prevents other ready jobs from being executed. Jobs that perform polling should be designed so that after a finitely many poll operations they will block waiting for communication.

`val give: Ch<'x> -> 'x -> Job<bool>`

Creates a job that attempts to give a value to another job waiting on the given channel. The result indicates whether a value was given or not. Note that the other side of the communication must be blocked on the channel for communication to happen.

`val take: Ch<'x> -> Job<option<'x>>`

Creates a job that attempts to take a value from another job waiting on the given channel. Note that the other side of the communication must be blocked on the channel for communication to happen.

`module Now =`

Immediate or non-workflow operations on synchronous channels.

`val send: Ch<'x> -> 'x -> unit`

Sends the given value to the specified channel. `Ch.Now.send xCh x` is equivalent to `Ch.send xCh x |> Hopac.start`.

Note that using this function in a job workflow is not optimal and you should use `Ch.send` instead.

`type IVar<'x> =`

Represents a write once variable.

Write once variables are designed for and most commonly used for getting replies from concurrent servers and asynchronous operations, but can also be useful for other purposes such as for one-shot events and for implementing incremental, but immutable, concurrent data structures.

Because it is common to need to be able to communicate either an expected successful result or an exceptional failure in typical use cases of write once variables, direct mechanisms are provided for both. The implementation is optimized in such a way that the ability to report an exceptional failure does not add overhead to the expected successful usage scenarios.

Write once variables are lightweight objects and it is typical to always just create a new write once variable when one is needed. In most cases, a write once variable will be slightly more lightweight than a channel. This is possible because write once variables do not support simple rendezvous like channels do. When simple rendezvous is necessary, a channel should be used instead.

`inherit Promise<'x>`

`IVar<'x>` is a subtype of `Promise<'x>` and `IVar.read xI` is equivalent to `xI :> Alt<'x>`.

`new: unit -> IVar<'x>`

Creates a new write once variable.

`new: 'x -> IVar<'x>`

Creates a new write once variable with the given value.

`new: exn -> IVar<'x>`

Creates a new write once variable with the given failure exception.

Operations on write once variables.

`val fill: IVar<'x> -> 'x -> Job<unit>`

Creates a job that writes the given value to the given write once variable. It is an error to write to a single write once variable more than once.

In most use cases of write once variables the write once assumption naturally follows from the property that there is only one concurrent job that may ever write to a particular write once variable. If that is not the case, then you should likely use some other communication primitive. See also: `*<=`, `tryFill`, `fillFailure`.

`val tryFill: IVar<'x> -> 'x -> Job<unit>`

Creates a job that tries to write the given value to the given write once variable. No operation takes place and no error is reported in case the write once variable has already been written to.

In most use cases of write once variables it should be clear that a particular variable is written to at most once, because there is only one specific concurrent job that may write to the variable, and `tryFill` should not be used as a substitute for not understanding how the program behaves. However, in some case it can be convenient to use a write once variable as a single shot event and there may be several concurrent jobs that initially trigger the event. In such cases, you may use `tryFill`.

`val fillFailure: IVar<'x> -> exn -> Job<unit>`

Creates a job that writes the given exception to the given write once variable. It is an error to write to a single `IVar` more than once. See also: `*<=!`, `fill`.

`val tryFillFailure: IVar<'x> -> exn -> Job<unit>`

Creates a job that tries to write the given exception to the given write once variable. No operation takes place and no error is reported in case the write once variable has already been written to.

`val read: IVar<'x> -> Alt<'x>`

Creates an alternative that becomes available after the write once variable has been written to.

`module Now =`

Immediate or non-workflow operations on write once variables.

`val isFull: IVar<'x> -> bool`

Returns true iff the given write once variable has already been filled (either with a value or with a failure).

This operation is mainly provided for advanced uses of write once variables such as when creating more complex data structures that make internal use of write once variables. Using this to poll write once variables is not generally a good idea.

`val get: IVar<'x> -> 'x`

Returns the value or raises the failure exception written to the write once variable. It is considered an error if the write once variable has not yet been written to.

This operation is mainly provided for advanced uses of write once variables such as when creating more complex data structures that make internal use of write once variables. Using this to poll write once variables is not generally a good idea.

`type Latch =`

Represents a dynamic latch. Latch is similar to the .Net `CountdownEvent`.

Latches are used for determining when a finite set of parallel jobs is done. If the size of the set is known a priori, then the latch can be initialized with the size as initial count and then each job just decrements the latch.

If the size is unknown (dynamic), then a latch is initialized with a count of one, the a priori known jobs are queued to the latch and then the latch is decremented. A queue operation increments the count immediately and decrements the count after the job is finished.

Both a first-order interface, with `create`, `increment` and `decrement` operations, and a higher-order interface, with `within`, `holding`, `queue` and `queueAsPromise` operations, are provided for programming with latches.

`inherit Alt<unit>`

`Latch` is a subtype of `Alt<unit>` and `Latch.await l` is equivalent to `l :> Alt<unit>`.

`new: int -> Latch`

Creates a new latch with the specified initial count.

`module Latch =`

Operations on latches.

Returns an alternative that becomes available once the latch opens.

```
val within: (Latch -> #Job<'x>) -> Job<'x>
```

Creates a job that creates a new latch, passes it to the given function to create a new job to run and then awaits for the latch to open.

```
val holding: Latch -> Job<'x> -> Job<'x>
```

Creates a job that runs the given job holding the specified latch. Note that the latch is only held while the given job is being run. See also `Latch.queue`.

```
val queue: Latch -> Job<unit> -> Job<unit>
```

Creates a job that queues the given job to run as a separate concurrent job and holds the latch until the queued job either returns or fails with an exception. See also `Latch.queueAsPromise`.

```
val queueAsPromise: Latch -> Job<'x> -> Job<Promise<'x>>
```

Creates a job that queues the given job to run as a separate concurrent job and holds the latch until the queued job either returns or fails with an exception. A promise is returned for observing the result or failure of the queued job.

```
val decrement: Latch -> Job<unit>
```

Returns a job that explicitly decrements the counter of the latch. When the counter reaches `0`, the latch becomes open and operations awaiting the latch are resumed.

```
module Now =
```

Immediate operations on latches.

```
val increment: Latch -> unit
```

Increments the counter of the latch.

```
type MVar<'x> =
```

Represents a serialized variable.

A serialized variable can be either empty or full. When a job makes an attempt to take the value of an empty variable, the job is suspended until some other job fills the variable with a value. At any one time there should only be at most one job that holds the state to be written to a serialized variable. If this cannot be guaranteed, in other words, there might be two or more jobs trying to fill a serialized variable, then you should not be using serialized variables. Indeed, the idea is that access to the state is serialized meaning that one and only one concurrent job has access to the state at any one time.

For example, one could declare a variable holding a shared map:

```
let sharedMap = MVar Map.empty
```

The map can then be accessed from multiple concurrent jobs, one job at a time, using the `MVar` operations:

```
MVar.mutateFun (Map.add key value) sharedMap
MVar.read sharedMap >>- Map.tryFind key
```

Another way to put the idea of serialized variables is that the variable acts as a mechanism for passing a permission token, the value contained by the variable, from one concurrent job to another. Only the concurrent job that holds the token is allowed to fill the variable. When used in this way, operations on the variable appear as atomic and access to the state will be serialized.

For example, here is an implementation of a synchronization object similar to the .Net `AutoResetEvent` using serialized variables:

```
type AutoResetEvent (init: bool) =
  let set = if init then MVar (()) else MVar ()
  let unset = if init then MVar () else MVar (())
  member this.Set = unset <|> set >>= MVar.fill set
  member this.Wait = set ^=> MVar.fill unset
```

The idea is to use two serialized variables to represent the state of the synchronization object. At most one of the variables, representing the state of the synchronization object, is full at any time.

In general, aside from a possible initial `fill` operation, an access to a serialized variable should be of the form `take >>= ... fill` or of the form `read`. Note that this follows naturally if you initialize a serialized variable with a value and then use the read, modify and mutate operations. On the other hand, accesses of the form `fill` and `read >>= ... fill` are unsafe. A `take` operation effectively grants permission to the job to access the shared state. The `fill` operation then gives that permission to the next job that wants to access the shared state.

WARNING: Unfortunately, `MVar`s are easy to use unsafely. Do not use an `MVar` to pass information from a client to a server, for example. Use a `Ch` or `Mailbox` for that. Note that if you are familiar with the `MVar` abstraction provided by Concurrent Haskell, then it is important to realize that the semantics and intended usage of Hopac's and Concurrent ML's `MVar` are quite different. The `MVar` of Concurrent Haskell is a bit like a simplified `Ch` with a buffer of one element and some additional operations. The `MVar` of Hopac and Concurrent ML does not al-

`inherit Alt<'x>`

`MVar<'x>` is a subtype of `Alt<'x>` and `xM :> Alt<'x>` is equivalent to `MVar.take xM`.

`new: unit -> MVar<'x>`

Creates a new serialized variable that is initially empty.

`new: 'x -> MVar<'x>`

Creates a new serialized variable that initially contains the given value.

`module MVar =`

Operations on serialized variables.

`val fill: MVar<'x> -> 'x -> Job<unit>`

Creates a job that writes the given value to the serialized variable. It is an error to write to a `MVar` that is full. See also: `*<<=`.

`val take: MVar<'x> -> Alt<'x>`

Creates an alternative that becomes available when the variable contains a value and, if committed to, takes the value from the variable.

`val read: MVar<'x> -> Alt<'x>`

Creates an alternative that becomes available when the variable contains a value and, if committed to, read the value from the variable.

Reference implementation:

```
let read xM = take xM ^=> fun x -> fill xM x >>-. x
```

`val mutateJob: ('x -> #Job<'x>) -> MVar<'x> -> Alt<unit>`

Creates an alternative that takes the value of the serialized variable and then fills the variable with the result of performing the given job.

`val tryMutateJob: ('x -> #Job<'x>) -> MVar<'x> -> Alt<unit>`

Creates an alternative that takes the value of the serialized variable and then fills the variable with the result of performing the given job. If the job raises an exception, the serialized variable is filled with its original value before propagating the exception.

`val mutateFun: ('x -> 'x) -> MVar<'x> -> Alt<unit>`

Creates an alternative that takes the value of the serialized variable and then fills the variable with the result of performing the given function.

`val tryMutateFun: ('x -> 'x) -> MVar<'x> -> Alt<unit>`

Creates an alternative that takes the value of the serialized variable and then fills the variable with the result of performing the given function. If the function raises an exception, the serialized variable is filled with its original value before propagating the exception.

`val modifyJob: ('x -> #Job<'x * 'y>) -> MVar<'x> -> Alt<'y>`

Creates an alternative that takes the value of the serialized variable and then fills the variable with the result of performing the given job.

Reference implementation:

```
let modifyJob (x2xyJ: 'x -> Job<'x * 'y>) (xM: MVar<'x>) =
  xM ^=> (x2xyJ >=> fun (x, y) -> fill xM x >>-. y)
```

`val tryModifyJob: ('x -> #Job<'x * 'y>) -> MVar<'x> -> Alt<'y>`

Creates an alternative that takes the value of the serialized variable and then fills the variable with the result of performing the given job. If the job raises an exception, the serialized variable is filled with its original value before propagating the exception.

`val modifyFun: ('x -> 'x * 'y) -> MVar<'x> -> Alt<'y>`

Creates an alternative that takes the value of the serialized variable and then fills the variable with the result of performing the given function.

Reference implementation:

```
        xM ^=> (x2xy >> fun (x, y) -> fill xM x >>-. y)
```

```
val tryModifyFun: ('x -> 'x * 'y) -> MVar<'x> -> Alt<'y>
```

Creates an alternative that takes the value of the serialized variable and then fills the variable with the result of performing the given function. If the function raises an exception, the serialized variable is filled with its original value before propagating the exception.

```
type BoundedMb<'x> =
```

Represents a bounded synchronous mailbox for many to many communication.

Bounded synchronous mailboxes are a useful tool for coordinating work among co-operating jobs. They provide slack in the form of buffering between producers and consumers allowing them to proceed in parallel. They also provide back-pressure in the form of blocking producers when consumers cannot keep up.

In cases where buffering is not necessary, the basic channel primitive, `Ch<_>`, should be preferred. In cases where unbounded buffering is not a problem, the basic mailbox primitive, `Mailbox<_>`, should be preferred.

At the time of writing, `BoundedMb<_>` is not implemented as a primitive, but is implemented using other primitives of Hopac, and it is likely that performance can be improved significantly. If you run into a case where the performance of `BoundedMb<_>` becomes problematic, please submit an issue.

```
new: int -> BoundedMb<'x>
```

Creates a new bounded mailbox.

```
module BoundedMb =
```

Operations on bounded synchronous mailboxes.

```
val put: BoundedMb<'x> -> 'x -> Alt<unit>
```

Selective synchronous operation to put a message to a bounded mailbox. `put` operations are processed in FIFO order and become enabled as soon as there is room in the bounded buffer. If the buffer capacity is `0`, `put` behaves exactly like `Ch.give`.

```
val take: BoundedMb<'x> -> Alt<'x>
```

Selective synchronous operation to take a message from a bounded mailbox. `take` operations are processed in FIFO order and become enabled as soon as there are messages in the bounded buffer. If the buffer capacity is `0`, `take` behaves exactly like `Ch.take`.

```
type Mailbox<'x> =
```

Represents an asynchronous, unbounded buffered mailbox.

Compared to channels, mailboxes take more memory when empty, but offer space efficient buffering of messages. In situations where buffering must be bounded, a bounded mailbox, `BoundedMb<_>`, should be preferred. In situations where buffering is not needed, a channel, `Ch<_>`, should be preferred.

```
inherit Alt<'x>
```

`Mailbox<'x>` is a subtype of `Alt<'x>` and `xMb :> Alt<'x>` is equivalent to `Mailbox.take xMb`.

```
new: unit -> Mailbox<'x>
```

Creates a new mailbox.

```
module Mailbox =
```

Operations on buffered mailboxes.

```
val send: Mailbox<'x> -> 'x -> Job<unit>
```

Creates a job that sends the given value to the specified mailbox. This operation never blocks. See also: `*<<+`.

```
val take: Mailbox<'x> -> Alt<'x>
```

Creates an alternative that becomes available when the mailbox contains at least one value and, if committed to, takes a value from the mailbox.

```
module Now =
```

Immediate or non-workflow operations on buffered mailboxes.

```
val send: Mailbox<'x> -> 'x -> unit
```

Sends the given value to the specified mailbox. `Mailbox.Now.send xMb x` is equivalent to `Mailbox.send xMb x |> Hopac.start`.

`type Lock =`

A non-recursive mutual exclusion lock for jobs.

In most cases you should use higher-level message passing primitives such as `Ch`, `Mailbox`, `MVar` or `IVar`, but in some cases a simple lock might be more natural to use.

Note that this lock is for synchronizing at the level of jobs. A job may even block while holding the lock. For short non-blocking critical sections, native locks (e.g. `Monitor` and `SpinLock`), concurrent data structures or interlocked operations should be faster. On the other hand, suspending and resuming a job is several orders of magnitude faster than suspending and resuming a native thread.

`new: unit -> Lock`

Creates a new lock.

`module Lock =`

Operations on mutual exclusion locks.

`val duringJob: Lock -> Job<'x> -> Job<'x>`

Creates a job that runs the given job so that the lock is held during the execution of the given job.

`val duringFun: Lock -> (unit -> 'x) -> Job<'x>`

Creates a job that calls the given function so that the lock is held during the execution of the function.

`module Extensions =`

Extensions to various system modules and types for programming with jobs. You can open this module to use the extensions much like as if they were part of the existing modules and types.

`module Array =`

Operations for processing arrays with jobs.

`val mapJob: ('x -> #Job<'y>) -> array<'x> -> Job<array<'y>>`

Sequentially maps the given job constructor to the elements of the array and returns an array of the results. `Array.mapJob x2yJ xs` is an optimized version of `Seq.mapJob x2yJ xs >>- fun ys -> ys.ToArray ()`.

`val iterJob: ('x -> #Job<unit>) -> array<'x> -> Job<unit>`

Sequentially iterates the given job constructor over the given array. `Array.iterJob x2uJ xs` is an optimized version of `Seq.iterJob x2uJ xs`.

`val iterJobIgnore: ('x -> #Job<_>) -> array<'x> -> Job<unit>`

`Array.iterJobIgnore x2yJ xs` is equivalent to `Array.iterJob (x2yJ >> Job.Ignore) xs`.

`module Seq =`

Operations for processing sequences with jobs.

`val foldJob: ('x -> 'y -> #Job<'x>) -> 'x -> seq<'y> -> Job<'x>`

Sequentially folds the job constructor over the given sequence and returns the result of the fold.

Reference implementation:

```
let foldJob xy2xJ x (ys: seq<'y>) = Job.delay <| fun () ->
  Job.using <| ys.GetEnumerator () <| fun ys ->
  let rec loop x =
    if ys.MoveNext () then
      xy2xJ x ys.Current >>= loop
    else
      Job.result x
  loop x
```

`val foldFromJob: 'x -> ('x -> 'y -> #Job<'x>) -> seq<'y> -> Job<'x>`

`foldFromJob x x2y2xJ ys` is equivalent to `foldJob x2y2xJ x ys` and is often syntactically more convenient.

`val iterJob: ('x -> #Job<unit>) -> seq<'x> -> Job<unit>`

Sequentially iterates the given job constructor over the given sequence. See also: `Seq.iterJobIgnore`, `Seq.Con.iterJob`,

Reference implementation:

```
let iterJob x2uJ (xs: seq<'x>) = Job.delay <| fun () ->
  Job.using <| xs.GetEnumerator () <| fun xs ->
  Job.whileDoDelay xs.MoveNext <| fun () ->
    x2uJ xs.Current
```

```
val iterJobIgnore: ('x -> #Job<_>) -> seq<'x> -> Job<unit>
```

`Seq.iterJobIgnore x2yJ xs` is equivalent to `Seq.iterJob (x2yJ >> Job.Ignore) xs`.

```
val mapJob: ('x -> #Job<'y>) -> seq<'x> -> Job<ResizeArray<'y>>
```

Sequentially maps the given job constructor to the elements of the sequence and returns a list of the results. See also: `seqCollect`, `Seq.Con.mapJob`, `Array.mapJob`.

Reference implementation:

```
let mapJob x2yJ (xs: seq<'x>) = Job.delay <| fun () ->
  let ys = ResizeArray<_>()
  Job.using <| xs.GetEnumerator () <| fun xs ->
  Job.whileDoDelay xs.MoveNext <| fun () ->
        x2yJ xs.Current >>- ys.Add
  >>-. ys
```

```
module Con =
```

Operations for processing sequences using concurrent jobs.

```
val iterJob: ('x -> #Job<unit>) -> seq<'x> -> Job<unit>
```

Iterates the given job constructor over the given sequence, runs the constructed jobs as separate concurrent jobs and waits until all of the jobs have finished. See also: `Con.iterJobIgnore`, `conIgnore`.

Note that this is not optimal for fine-grained parallel execution.

```
val iterJobIgnore: ('x -> #Job<_>) -> seq<'x> -> Job<unit>
```

`Con.iterJobIgnore x2yJ xs` is equivalent to `Con.iterJob (x2yJ >> Job.Ignore) xs`.

Note that this is not optimal for fine-grained parallel execution.

```
val mapJob: ('x -> #Job<'y>) -> seq<'x> -> Job<ResizeArray<'y>>
```

Iterates the given job constructor over the given sequence, runs the constructed jobs as separate concurrent jobs and waits until all of the jobs have finished collecting the results into a list. See also: `Seq.mapJob`, `conCollect`.

Note that this is not optimal for fine-grained parallel execution.

```
module Async =
```

Operations for interfacing F# async operations with jobs.

Note that these operations are provided for interfacing with existing APIs that work with async operations. Running async operations within jobs and vice versa incurs potentially significant overheads.

Note that there is almost a one-to-one mapping between async operations and jobs. The main semantic difference between async operations and Hopac jobs is the threads and schedulers they are being executed on.

```
[<AbstractClass>]
type OnWithSchedulerBuilder =
```

Builder for async workflows. The methods in this builder delegate to the default `async` builder.

```
new: unit -> OnWithSchedulerBuilder
```

```
abstract Scheduler: Scheduler
```

```
abstract Context: SynchronizationContext
```

```
member Bind: Task<'x> * ('x -> Async<'y>) -> Async<'y>
```

```
member Bind: Job<'x> * ('x -> Async<'y>) -> Async<'y>
```

```
member Bind: Async<'x> * ('x -> Async<'y>) -> Async<'y>
```

```
        member Delay: (unit -> Async<'x>) -> Async<'x>

        member For: seq<'x> * ('x -> Async<unit>) -> Async<unit>

        member Return: 'x -> Async<'x>

        member ReturnFrom: Task<'x> -> Async<'x>

        member ReturnFrom: Job<'x> -> Async<'x>

        member ReturnFrom: Async<'x> -> Async<'x>

        member TryFinally: Async<'x> * (unit -> unit) -> Async<'x>

        member TryWith: Async<'x> * (exn -> Async<'x>) -> Async<'x>

        member Using: 'x * ('x -> Async<'y>) -> Async<'y> when 'x :> IDisposable

        member While: (unit -> bool) * Async<unit> -> Async<unit>

        member Zero: unit -> Async<unit>

        member Run: Async<'x> -> Job<'x>

    module Global =
```

Operations on the global scheduler.

```
        val onMain: unit -> OnWithSchedulerBuilder
```

Creates a builder for running an async workflow on the main synchronization context and interoperating with the Hopac global scheduler. The application must call `Hopac.Extensions.Async.setMain` to configure Hopac with the main synchronization context.

```
    val setMain: SynchronizationContext -> unit
```

Sets the main synchronization context. This must be called by application code in order to use operations such as `onceAltOnMain` and `Hopac.onMain`

```
    val getMain: unit -> SynchronizationContext
```

Gets the main synchronization context. The main synchronization context must be set by application code using `setMain` before calling this function.

```
val asyncOn: SynchronizationContext -> Scheduler -> Async.OnWithSchedulerBuilder
```

Builder for an async operation started on the given synchronization context with jobs on the specified scheduler wrapped as a job.

```
type Task with
```

Operations for interfacing tasks with jobs.

Note that these operations are provided for interfacing with existing APIs that work with tasks. Starting a job as a task and then awaiting for its result has much higher overhead than simply starting the job as a `Promise`, for example.

Note that starting tasks correctly can be tricky. Hopac jobs are designed to be executed by Hopac worker threads, which have the default `null` synchronization context like the .Net thread pool, but Hopac jobs can also be started on other threads, which may live in non-default synchronization contexts. Tasks that have been written using the C# async-await mechanism may capture the current synchronization context. This means that when you call a function to start a task within a Hopac job, you may need to explicitly post that function call to a specific synchronization context.

Note that tasks and jobs are quite different in nature as tasks are comonadic while jobs are monadic.

```
    static member startJob: Job<'x> -> Job<Task<'x>>
```

Creates a job that starts the given job as a separate concurrent job, whose result can be obtained from the returned task.

```
exception OnCompleted
```

Raised by `onceAltOn` when the associated observable signals the `OnCompleted` event.

Operations for interfacing Hopac with observables.

```
member onceAltOn: SynchronizationContext -> Alt<'x>
```

Creates an alternative that, when instantiated, subscribes to the observable on the specified synchronization context for at most one event. Passing `null` as the synchronization context means that the subscribe and unsubscribe actions are performed on an unspecified thread.

After an `OnNext` event, the alternative returns the value given by the observable. After an `OnError` event, the alternative raises the exception given by the observable. After an `OnCompleted` event, the alternative raises the `OnCompleted` exception.

The alternative becomes available as soon as the observable signals any event after which the alternative unsubscribes from the observable. If some other alternative is committed to before the observable signals any event, the alternative unsubscribes from the observable. Note, however, that if the current job explicitly aborts while instantiating some other alternative involved in the same synchronous operation, there is no guarantee that the observable would be unsubscribed from.

Note that, as usual, the alternative can be used many times and even concurrently.

```
member onceAltOnMain: Alt<'x>
```

This is equivalent to calling `onceAltOn` with the main synchronization context. The application must call `Hopac.Extensions.Async.setMain` to configure Hopac with the main synchronization context.

```
member onceAlt: Alt<'x>
```

`xO.onceAlt` is equivalent to `xO.oneAltOn null`. Note that it is often necessary to specify the synchronization context to subscribe on. See also: `Observable.SubscribeOn`.

```
module Infixes =
```

Infix operators for concise expression of key Hopac idioms. You can open this module to bring all of the infix operators into scope.

The operator symbols have been designed to allow the most common expressions to be written without parentheses:

- Message passing operators start with `*` and have the highest precedence.

- After action operators start with `^` and have the second highest precedence.

- Choice and pairing operators start with `<` and have the lowest precedence.

- Sequencing operators start with `>` and also have the lowest precendence.

Fortunately, sequencing usually follows after choice or pairing, or is inside a function expression, so parentheses are not required in that case.

As an example of operator usage, the `swap` operation of a swap channel combines message passing, after actions, sequencing and choice:

```
let swap swapCh outMsg =
      swapCh ^=> fun (inMsg, outIv) -> outIv *<= outMsg >>-. inMsg
   <|> swapCh *<-=>- fun inIv -> (outMsg, inIv)
    :> Alt<_>
```

The type ascription above is unnecessary and is there only as an example of how it can be placed.

```
val ( *<+->= ): Ch<'q> -> (Ch<'r> -> Promise<unit> -> #Job<'q>) -> Alt<'r>
```

Creates an alternative that, using the given job constructor, constructs a query with a reply channel and a nack, sends it to the query channel and commits on taking the reply from the reply channel. See also: `*<+->-`.

Reference implementation:

```
let ( *<+->= ) qCh rCh2n2qJ = Alt.withNackJob <| fun nack ->
  let rCh = Ch<_> ()
  rCh2n2qJ rCh nack >>= fun q ->
  qCh *<+ q >>-.
  rCh
```

```
val ( *<+->- ): Ch<'q> -> (Ch<'r> -> Promise<unit> -> 'q) -> Alt<'r>
```

Creates an alternative that, using the given function, constructs a query with a reply channel and a nack, sends it to the query channel and commits on taking the reply from the reply channel. `*<+->-` captures the most common use case of `Alt.withNackJob` and is a slightly less expressive form of `*<+->=`. See also: `*<-=>-`.

Here is the `incrementBy` function from the example in `Alt.withNackJob` expressed using `*<+->-`:

```
let incrementBy n =
  counterServer *<+->- fun replyCh nack -> (n, nack, replyCh)
```

```
 let ( *<+->- ) qCh rCh2n2q =
   qCh *<+->= fun rCh n -> rCh2n2q rCh n |> result
```

```
val ( *<-=>= ): Ch<'q> -> (IVar<'r> -> #Job<'q>) -> Alt<'r>
```

Creates an alternative that, using the given job constructor, constructs a query with a reply variable, commits on giving the query and reads the reply variable. See also: `*<-=>-`.

Reference implementation:

```
 let ( *<-=>= ) qCh rI2qJ = Alt.prepareJob <| fun () ->
   let rI = IVar<_> ()
   rI2qJ rI >>- fun q ->
   qCh *<- q ^=>.
   rI
```

```
val ( *<-=>- ): Ch<'q> -> (IVar<'r> -> 'q) -> Alt<'r>
```

Creates an alternative that, using the given function, constructs a query with a reply variable, commits on giving the query and reads the reply variable. `*<-=>-` captures the most common use case of `Alt.prepareFun` and is a slighly less expressive form of `*<-=>=`. See also: `*<+->-`.

Reference implementation:

```
 let ( *<-=>- ) qCh rI2q = qCh *<-=>= (rI2q >> result)
```

```
val ( *<+=>= ): Ch<'q> -> (IVar<'r> -> #Job<'q>) -> Alt<'r>
```

Creates an alternative that, using the given job constructor, constructs a query with a reply variable, sends the query and reads the reply. In order for the alternative to make sense, the operation must not require exclusive choice. If this not the case, then the resulting value should only be used as a job.

Reference implementation:

```
 let ( *<+=>= ) qCh rI2qJ = Alt.prepareJob <| fun () ->
   let rI = IVar ()
   rI2qJ rI >>= fun q ->
   qCh *<+ q >>-.
   rI
```

```
val ( *<+=>- ): Ch<'q> -> (IVar<'r> -> 'q) -> Alt<'r>
```

Creates an alternative that, using the given function, constructs a query with a reply variable, sends the query and reads the reply. In order for the alternative to make sense, the operation must not require exclusive choice. If this is not the case, then the resulting value should only be used as a job.

Reference implementation:

```
 let ( *<+=>- ) qCh rI2q = qCh *<+=>= (rI2q >> result)
```

```
val ( *<- ): Ch<'x> -> 'x -> Alt<unit>
```

Creates an alternative that, at instantiation time, offers to give the given value on the given channel, and becomes available when another job offers to take the value. `xCh *<- x` is equivalent to `Ch.give xCh x`.

```
val ( *<+ ): Ch<'x> -> 'x -> Job<unit>
```

Creates a job that sends a value to another job on the given channel. A send operation is asynchronous. In other words, a send operation does not wait for another job to give the value to. `xCh *<+ x` is equivalent to `Ch.send xCh x`.

Note that channels have been optimized for synchronous operations; an occasional send can be efficient, but when sends are queued, performance maybe be significantly worse than with a `Mailbox` optimized for buffering.

```
val ( *<= ): IVar<'x> -> 'x -> Job<unit>
```

Creates a job that writes to the given write once variable. It is an error to write to a single `IVar` more than once. `xI *<= x` is equivalent to `IVar.fill xI x`.

```
val ( *<=! ): IVar<'x> -> exn -> Job<unit>
```

Creates a job that writes the given exception to the given write once variable. It is an error to write to a single `IVar` more than once. `xI *<=! e` is equivalent to `IVar.fillFailure xI e`.

```
val ( *<<= ): MVar<'x> -> 'x -> Job<unit>
```

Creates a job that writes the given value to the serialized variable. It is an error to write to a `MVar` that is full. `xM *<<= x` is equivalent to `val ( *<<+ ): Mailbox<'x> -> 'x -> Job<unit>`

Creates a job that sends the given value to the specified mailbox. This operation never blocks. `xMb *<<+ x` is equivalent to `Mailbox.send xMb x`.

`val ( ^=> ): Alt<'x> -> ('x -> #Job<'y>) -> Alt<'y>`

Creates an alternative whose result is passed to the given job constructor and processed with the resulting job after the given alternative has been committed to. This is the same as `afterJob` with the arguments flipped.

`val ( ^-> ): Alt<'x> -> ('x -> 'y) -> Alt<'y>`

`xA ^-> x2y` is equivalent to `xA ^=> (x2y >> result)`. This is the same as `afterFun` with the arguments flipped.

`val ( ^=>. ): Alt<_> -> Job<'y> -> Alt<'y>`

`xA ^=>. yJ` is equivalent to `xA ^=> fun _ -> yJ`.

`val ( ^->. ): Alt<_> -> 'y -> Alt<'y>`

`xA ^->. y` is equivalent to `xA ^-> fun _ -> y`.

`val ( ^->! ): Alt<_> -> exn -> Alt<_>`

`xA ^->! e` is equivalent to `xA ^-> fun _ -> raise e`.

`val ( <|> ): Alt<'x> -> Alt<'x> -> Alt<'x>`

Creates an alternative that is available when either of the given alternatives is available. `xA1 <|> xA2` is an optimized version of `choose [xA1; xA2]`. See also: `choosy`.

The given alternatives are processed in a left-to-right order with short-cut evaluation. In other words, given an alternative of the form `first <|> second`, the `first` alternative is first instantiated and, if it is available, is committed to and the `second` alternative will not be instantiated at all.

`val ( <|>* ): Alt<'x> -> Alt<'x> -> Promise<'x>`

A memoizing version of `<|>`.

`val ( <~> ): Alt<'x> -> Alt<'x> -> Alt<'x>`

`xA1 <~> xA2` is like `xA1 <|> xA2` except that the order in which `xA1` and `xA2` are instantiated is determined at random every time the alternative is used. See also: `chooser`.

Note that randomization only applies to the instantiation order. It makes no difference after instantiation.

WARNING: Chained uses of `<~>` do not lead to uniform distributions. Consider the expression `xA1 <~> xA2 <~> xA3`. It parenhesizes as `(xA1 <~> xA2) <~> xA3`. This means that `xA3` has a 50% and both `xA1` and `xA2` have 25% probability of being considered first.

`val ( <~>* ): Alt<'x> -> Alt<'x> -> Promise<'x>`

A memoizing version of `<~>`.

`val ( >>= ): Job<'x> -> ('x -> #Job<'y>) -> Job<'y>`

Creates a job that first runs the given job and then passes the result of that job to the given function to build another job which will then be run. This is the same as `bind` with the arguments flipped.

`val ( >>=* ): Job<'x> -> ('x -> #Job<'y>) -> Promise<'y>`

A memoizing version of `>>=`.

`val ( >>- ): Job<'x> -> ('x -> 'y) -> Job<'y>`

Creates a job that runs the given job and maps the result of the job with the given function. `xJ >>- x2y` is an optimized version of `xJ >>= (x2y >> result)`. This is the same as `map` with the arguments flipped.

`val ( >>-* ): Job<'x> -> ('x -> 'y) -> Promise<'y>`

A memoizing version of `>>-`.

`val ( >>=. ): Job<_> -> Job<'y> -> Job<'y>`

Creates a job that runs the given two jobs and returns the result of the second job. `xJ >>=. yJ` is equivalent to `xJ >>= fun _ -> yJ`.

A memoizing version of `>>=.`.

`val ( >>-. ): Job<_> -> 'y -> Job<'y>`

Creates a job that runs the given job and then returns the given value. `xJ >>-. y` is an optimized version of `xJ >>= fun _ -> result y`.

`val ( >>-*. ): Job<_> -> 'y -> Promise<'y>`

A memoizing version of `>>-.`.

`val ( >>-! ): Job<_> -> exn -> Job<_>`

Creates a job that runs the given job and then raises the given exception. `xJ >>-! e` is equivalent to `xJ >>= fun _ -> raise e`.

`val ( >>-*! ): Job<_> -> exn -> Promise<_>`

A memoizing version of `>>-!`.

`val ( >=> ): ('x -> #Job<'y>) -> ('y -> #Job<'z>) -> 'x -> Job<'z>`

Creates a job that is the composition of the given two job constructors. `(x2yJ >=> y2zJ) x` is equivalent to `x2yJ x >>= y2zJ` and is much like the `>>` operator on ordinary functions.

`val ( >=>* ): ('x -> #Job<'y>) -> ('y -> #Job<'z>) -> 'x -> Promise<'z>`

A memoizing version of `>=>`.

`val ( >-> ): ('x -> #Job<'y>) -> ('y -> 'z) -> 'x -> Job<'z>`

Creates a job that is the composition of the given job constructor and function. `(x2yJ >-> y2z) x` is equivalent to `x2yJ x >>- y2z` and is much like the `>>` operator on ordinary functions.

`val ( >->* ): ('x -> #Job<'y>) -> ('y -> 'z) -> 'x -> Promise<'z>`

A memoizing version of `>->`.

`val ( >=>. ): ('x -> #Job<_>) -> Job<'z> -> 'x -> Job<'z>`

`(x2yJ >=>. zJ) x` is equivalent to `x2yJ x >>=. zJ`.

`val ( >=>*. ): ('x -> #Job<_>) -> Job<'z> -> 'x -> Promise<'z>`

A memoizing version of `>=>.`.

`val ( >->. ): ('x -> #Job<_>) -> 'z -> 'x -> Job<'z>`

`(x2yJ >->. z) x` is equivalent to `x2yJ x >>-. z`.

`val ( >->*. ): ('x -> #Job<_>) -> 'z -> 'x -> Promise<'z>`

A memoizing version of `>->.`.

`val ( >->! ): ('x -> #Job<_>) -> exn -> 'x -> Job<_>`

`(x2yJ >->! e) x` is equivalent to `x2yJ x >>-! e`.

`val ( >->*! ): ('x -> #Job<_>) -> exn -> 'x -> Promise<_>`

A memoizing version of `>->!`.

`val ( <&> ): Job<'x> -> Job<'y> -> Job<'x * 'y>`

Creates a job that runs the given two jobs and then returns a pair of their results. `xJ <&> yJ` is equivalent to `xJ >>= fun x -> yJ >>= fun y -> result (x, y)`.

`val ( <*> ): Job<'x> -> Job<'y> -> Job<'x * 'y>`

Creates a job that either runs the given jobs sequentially, like `<&>`, or as two separate parallel jobs and returns a pair of their results.

Note that when the jobs are run in parallel and both of them raise an exception then the created job raises an `AggregateException`.

Note that, because it is not guaranteed that the jobs would always be run as separate parallel jobs, a job such as

```
        let c = Ch ()
        Ch.give c () <*> Ch.take c
```

may deadlock. If two jobs need to communicate with each other they need to be started as two separate jobs.

```
val ( <+> ): Alt<'x> -> Alt<'y> -> Alt<'x * 'y>
```

An alternative that is equivalent to first committing to either one of the given alternatives and then committing to the other alternative. Note that this is not the same as committing to both of the alternatives in a single transaction. Such an operation would require a more complex synchronization protocol like with the so called Transactional Events.

```
type Proc =
```

Represents a handle to a (started, running or terminated) job.

A handle makes it possible to determine when a job is known to have been terminated. An example use for handles would be a system where critical resources are managed by a server job and those critical resources need to be released even in case a client job suffers from a fault and is terminated before properly releasing resources.

For performance reasons, Hopac creates handles lazily for simple jobs, because for many uses of lightweight threads such a capability is simply not necessary. However, when handles are known to be needed, it is better to allocate them eagerly by directly starting jobs using `Proc.start` or `Proc.queue`.

```
inherit Alt<unit>
```

`Proc` is a subtype of `Alt<unit>` and `p :> Alt<unit>` is equivalent to `Proc.join p`.

```
module Proc =
```

Operations on handles.

```
val queue: Job<unit> -> Job<Proc>
```

Creates a job that queues a new job with a handle. See also: `start`, `Job.queue`.

```
val queueIgnore: Job<_> -> Job<Proc>
```

Creates a job that queues a new job with a handle. `queueIgnore xJ` is equivalent to `Job.Ignore xJ |> queue`.

```
val start: Job<unit> -> Job<Proc>
```

Creates a job that starts a new job with a handle. See also: `queue`, `Job.start`.

```
val startIgnore: Job<_> -> Job<Proc>
```

Creates a job that starts a new job with a handle. `startIgnore xJ` is equivalent to `Job.Ignore xJ |> start`.

```
val bind: (Proc -> #Job<'x>) -> Job<'x>
```

Creates a job that calls the given job contructor with the handle of the current job.

Note that this is an `O(n)` operation where `n` is the number of continuation or stack frames of the current job. In most cases this should not be an issue, but if you need to repeatedly access the proc handle of the current job it may be advantageous to cache it in a local variable.

```
val map: (Proc -> 'x) -> Job<'x>
```

`map p2x` is equivalent to `bind (p2x >> result)`.

```
val self: unit -> Job<Proc>
```

Returns a job that returns the handle of the current job. `self ()` is equivalent to `bind result`.

```
val join: Proc -> Alt<unit>
```

Returns an alternative that becomes available once the job corresponding to the handle is known to have been terminated for any reason.

```
type JobBuilder =
```

Expression builder type for jobs.

The following expression constructs are supported:

```
... ; ...
do ...
do! ... | async | task | obs
```

```
for ... in ... do ...
if ... then ...
if ... then ... else ...
let ... = ... in ...
let! ... = ... | async | task | obs in ...
match ... with ...
return ...
return! ... | async | task
try ... finally ...
try ... with ...
use ... = ... in ...
use! ... = ... in ...
while ... do ...
```

In the above, an ellipsis denotes either a job, an ordinary expression or a pattern. A job workflow can also directly bind and return from async operations, which will be started on a Hopac worker thread (see `Job.fromAsync`), tasks (see `Job.awaitTask`) and observables (see `IObservable<'x>.onceAlt`).

Note that the `Job` module provides more combinators for constructing jobs. For example, the F# workflow notation does not support `Job.tryFinallyJob` and `Job.tryIn` is easier to use correctly than `try ... with ...` expressions. Operators such as `>>-` and `>>-.` and operations such as `Job.iterate` and `Job.forever` are frequently useful and may improve performance.

```
new: unit -> JobBuilder
```

```
member Bind: IObservable<'x> * ('x -> Job<'y>) -> Job<'y>
```

```
member Bind: Async<'x> * ('x -> Job<'y>) -> Job<'y>
```

```
member Bind: Task<'x> * ('x -> Job<'y>) -> Job<'y>
```

```
member Bind: Job<'x> * ('x -> Job<'y>) -> Job<'y>
```

```
member Combine: Job<unit> * (unit -> Job<'x>) -> Job<'x>
```

```
member Delay: (unit -> Job<'x>) -> (unit -> Job<'x>)
```

```
member For: seq<'x> * ('x -> Job<unit>) -> Job<unit>
```

```
member Return: 'x -> Job<'x>
```

```
member ReturnFrom: IObservable<'x> -> Job<'x>
```

```
member ReturnFrom: Async<'x> -> Job<'x>
```

```
member ReturnFrom: Task<'x> -> Job<'x>
```

```
member ReturnFrom: Job<'x> -> Job<'x>
```

```
member Run: (unit -> Job<'x>) -> Job<'x>
```

```
member TryFinally: (unit -> Job<'x>) * (unit -> unit) -> Job<'x>
```

```
member TryWith: (unit -> Job<'x>) * (exn -> Job<'x>) -> Job<'x>
```

```
member Using: 'x * ('x -> Job<'y>) -> Job<'y> when 'x :> IDisposable
```

```
member While: (unit -> bool) * (unit -> Job<unit>) -> Job<unit>
```

```
member Zero: unit -> Job<unit>
```

```
type EmbeddedJob<'x> = struct
```

Represents a job to be embedded within a computation built upon jobs.

Embedded jobs can be useful when defining computations built upon jobs. Having to encode lightweight threads using the job monad is somewhat unfortunate, because it is such a fundamental abstraction. One sometimes, perhaps even often, wants to define more interesting computations upon jobs, but the traditional way of doing that requires adding yet another costly layer of abstraction on top of jobs. Another possibility is to expose the `Job<'x>` type constructor as shown in the following example:

```
type Monad<'x>
```

```
    member Delay: unit -> Job<Monad<'x>>
    member Return: 'x -> Job<Monad<'x>>
    member Bind:   Job<Monad<'x>> * ('x -> Job<Monad<'y>>) -> Job<Monad<'y>>
    member Bind: EmbeddedJob<'x>  * ('x -> Job<Monad<'y>>) -> Job<Monad<'y>>
```

The `Monad<'x>` type constructor and the `MonadBuilder` defines the new computation mechanism on top of jobs. The `Bind` operation taking an `EmbeddedJob<'x>` allows one to conveniently embed arbitrary jobs within the computations without introducing nasty overload resolution problems.

Consider what would happen if one would instead define `MonadBuilder` as follows:

```
  type MonadBuilder =
    member Delay: unit -> Job<Monad<'x>>
    member Return: 'x -> Job<Monad<'x>>
    member Bind: Job<Monad<'x>> * ('x -> Job<Monad<'y>>) -> Job<Monad<'y>>
    member Bind: Job<      'x > * ('x -> Job<Monad<'y>>) -> Job<Monad<'y>>
```

A `Bind` operation is now almost always ambiguous and one would have to annotate bind expressions to resolve the ambiguity.

The types of the operations in the `MonadBuilder` may, at first glance, seem complicated. Essentially the covariant positions in the signature are wrapped with the `Job<_>` type constructor to make it possible to use lightweight threads. In a language with built-in lightweight threads this would be unnecessary. Reading the signature by mentally replacing every `Job<'x>` with just `'x`, the signature should become clear.

`type EmbeddedJobBuilder =`

A builder for embedded jobs.

`type Scheduler`

Represents a scheduler that manages a number of worker threads.

`module Scheduler =`

Operations on schedulers. Use of this module requires more intimate knowledge of Hopac, but may allow adapting Hopac to special application requirements.

`type Create =`

A record of scheduler configuration options.

`Foreground: option<bool>`

Specifies whether worker threads are run as background threads or as foreground threads. The default is to run workers as background threads. If you want to run worker threads as foreground threads, then you will have to explicitly kill the worker threads. Using foreground threads is probably preferable if your application dynamically creates and kills local schedulers to make sure the worker threads are properly killed.

`IdleHandler: option<Job<int>>`

Specifies the idle handler for workers. The worker idle handler is run whenever an individual worker runs out of work. The idle handler must return an integer value that specifies how many milliseconds the worker is allowed to sleep. `Timeout.Infinite` puts the worker into sleep until the scheduler explicitly wakes it up. `0` means that the idle handler found some new work and the worker should immediately look for it.

`MaxStackSize: option<int>`

Specifies the maximum stack size for worker threads. The default is to use the default maximum stack size of the `Thread` class.

`NumWorkers: option<int>`

Number of worker threads. Using more than `Environment.ProcessorCount` is not optimal and may, in some cases, significantly reduce performance. The default is `Environment.ProcessorCount.`

`TopLevelHandler: option<exn -> Job<unit>>`

Specifies the top level exception handler job constructor of the scheduler. When a job fails with an otherwise unhandled exception, the job is killed and a new job is constructed with the top level handler constructor and then started. To avoid infinite loops, in case the top level handler job raises exceptions, it is simply killed after printing a message to the console. The default top level handler simply prints out a message to the console.

`static member Def: Create`

Default options.

`module Global =`

Operations on the global scheduler.

Sets options for creating the global scheduler. This must be called before invoking any Hopac functionality that implicitly creates the global scheduler.

```
val create: Create -> Scheduler
```

Creates a new local scheduler.

Note that a local scheduler does not automatically implement services such as the global wall-clock timer.

```
val queue: Scheduler -> Job<unit> -> unit
```

Queues the given job for execution on the scheduler.

Note that using this function in a job workflow is not optimal and you should use `Job.queue` instead.

```
val queueIgnore: Scheduler -> Job<_> -> unit
```

`queueIgnore xJ` is equivalent to `Job.Ignore xJ |> queue`.

```
val server: Scheduler -> Job<Void> -> unit
```

Like `Scheduler.start`, but the given job is known never to return normally, so the job can be spawned in an even more lightweight manner.

```
val start: Scheduler -> Job<unit> -> unit
```

Starts running the given job, but does not wait for the job to finish.

Note that using this function in a job workflow is not optimal and you should use `Job.start` instead.

```
val startIgnore: Scheduler -> Job<_> -> unit
```

`startIgnore xJ` is equivalent to `Job.Ignore xJ |> start`.

```
val startWithActions: Scheduler -> (exn -> unit) -> ('x -> unit) -> Job<'x> -> unit
```

Starts running the given job, but does not wait for the job to finish. Upon the failure or success of the job, one of the given actions is called once. See also: `abort`.

Note that using this function in a job workflow is not optimal and you should instead use `Job.start` with desired Job exception handling construct (e.g. `Job.tryIn` or `Job.catch`).

```
val run: Scheduler -> Job<'x> -> 'x
```

Starts running the given job on the specified scheduler and then blocks the current thread waiting for the job to either return successfully or fail.

WARNING: Use of `run` should be considered carefully, because calling `run` from an arbitrary thread can cause deadlock.

A call of `run xJ` is safe when the call is not made from within a Hopac worker thread and the job `xJ` does not perform operations that might block or that might directly, or indirectly, need to communicate with the thread from which `run` is being called.

```
val wait: Scheduler -> unit
```

Waits until the scheduler becomes completely idle.

Note that for this to make sense, the scheduler should be a local scheduler that your program manages explicitly.

```
val kill: Scheduler -> unit
```

Kills the worker threads of the scheduler one-by-one. This should only be used with a local scheduler that is known to be idle.

```
module Stream =
```

Operations on choice streams.

```
type Cons<'x> =
```

Represents a point in a non-deterministic stream of values.

```
| Cons of Value: 'x * Next: Promise<Cons<'x>>
```

Communicates a value and the remainder of the stream.

```
| Nil
```

```
type Stream<'x> = Promise<Cons<'x>>
```

Represents a non-deterministic stream of values called a choice stream.

Choice streams are essentially lazy lists of promises. Anything that can be done using ordinary lazy lists can also be done using choice streams. In addition, choice streams allow asynchronicity at any point and have a concept of time making it possible to support various operations, such as a non-deterministic `merge`, that are not supported by ordinary lazy lists.

Thanks to the additional power compared to ordinary lazy lists, choice streams can be used to solve similar problems as Rx observable sequences. However, the underlying implementations of choice streams and observable sequences are almost polar opposites: choice streams are pull based immutable (or write once) chains, while observable sequences are push based imperative pipes. Many things that are difficult with observables can be easy with choice streams and vice versa.

Probably the most notable advantage of observable sequences over choice streams is that observables support disposables via their all-or-nothing subscription protocol. Choice streams cannot support disposables in the exact same manner, because elements are requested asynchronously one at a time and choice streams do not have a subscription protocol. However, `onCloseJob` and `doFinalizeJob` provide similar functionality.

On the other hand, choice streams offer several advantages over observable sequences:

- Choice streams are simple and allow consumers and producers to be written using simple programming techniques such as lexical binding, recursion and immutable data structures. You can see many examples of this in the reference implementations of various stream combinators. Observable sequences can only be subscribed to by imperative callbacks. The implementation of choice streams is two orders of magnitude shorter than the implementation of .Net Rx.

- Basically all operations on ordinary lazy streams can be implemented on and are meaningful on choice streams. The same is not true of observable sequences, because they do not compose the same way. Many trivial choice stream combinators, such as `foldBack` and `tails`, can be either impossible or very challenging to specify and implement meaningfully for observable sequences.

- Choice streams allow for the use of asynchronous programming at any point. Most higher-order choice stream combinators have both an asynchronous `Job` and a synchronous `Fun` form. For example, `iterJob` waits for the asynchronous job to finish before consuming the next value from the stream. The `Subscribe` operation of observables cannot support such behavior, because `OnNext` calls are synchronous.

- Choice streams are consistent in that every consumer of a stream gets the exact same sequence of values unlike with observable sequences. In other words, choice streams are immutable and elements are not discarded implicitly. There is no need for `Connect` and `Publish` or `Replay` like with observable sequences.

- The asynchronous one element at a time model of choice streams allows for a basic form of backpressure or the flow of synchronization from consumers to producers. This allows for many new operations to be expressed. For example, `keepPreceding1` and `keepFollowing1` cannot be implemented for observable sequences. Operations such as `afterEach` and `beforeEach` have semantics that are pull based and lazy and cannot be implemented for observable sequences. Operations such as `pullOn`, or `zip` in disguise, have new uses.

All of the above advantages are strongly related and result from the pull based nature of choice streams. Pull semantics puts the consumer in control.

While the most common operations are very easy to implement on choice streams, some operations perhaps require more intricate programming than with push based models. For example, `groupByFun` and `shift`, which corresponds to `Delay` in Rx, are non-trivial, although both implementations are actually much shorter than their .Net Rx counterparts.

```
type Src<'x>
```

Represents an imperative source of a stream of values called a stream source.

A basic use for a stream source would be to produce a stream in response to events from a GUI. For example, given a GUI button, one could write

```
let buttonClickSrc = Stream.Src.create ()
button.Click.Add (ignore >> Stream.Src.value buttonClickSrc >> start)
```

to produce a stream of button clicks. The `Stream.Src.tap` function returns the generated stream, which can then be manipulated using stream combinators. See also: `ofObservableOnMain`.

Here is a silly example. We could write a stream combinator that counts the number of events within a given timeout period:

```
let eventsWithin timeout xs =
  let inc = xs |> Stream.mapConst +1
  let dec = xs |> Stream.mapConst -1 |> Stream.shift timeout
  Stream.merge inc dec
  |> Stream.scanFromFun 0 (+)
```

Given two stream sources for buttons, the following program would then print "That was fast!" whenever both buttons are clicked within 500ms.

```
let t500ms = timeOutMillis 500
let n1s = Stream.Src.tap button1ClickSrc |> eventsWithin t500ms
let n2s = Stream.Src.tap button2ClickSrc |> eventsWithin t500ms
Stream.combineLatest n1s n2s
|> Stream.chooseFun (fun (n1, n2) ->
   if n1 > 0 && n2 > 0 then Some () else None)
|> Stream.consumeFun (fun () -> printfn "That was fast!")
```

imperative mechanisms outside of the stream library.

`module Src =`

Operations on stream sources.

`val create: unit -> Src<'x>`

Creates a new stream source.

`val value: Src<'x> -> 'x -> Job<unit>`

Appends a new value to the end of the generated stream. This operation is atomic and non-blocking and can be safely used from multiple parallel jobs.

`val error: Src<'x> -> exn -> Job<unit>`

Terminates the stream with an error. The given exception is raised in the consumers of the stream if and when they reach the end of the stream.

`val close: Src<'x> -> Job<unit>`

Terminates the stream.

`val tap: Src<'x> -> Stream<'x>`

Returns the remainder of the generated stream after the point in time when `tap` is called.

`type Var<'x>`

Represents a mutable variable, called a stream variable, that generates a stream of values as a side-effect. See also: `MVar<'x>`.

The difference between a stream variable and a stream source is that a stream variable cannot be closed and always has a value. Stream variables are one way to represent state, or the model, manipulated by a program using streams.

Note that there are no special hidden mechanisms involved in the implementation of stream variables. You can easily implement similar imperative mechanisms outside of the stream library.

`module Var =`

Operations on stream variables.

`val create: 'x -> Var<'x>`

Creates a new stream variable.

`val get: Var<'x> -> 'x`

Gets the value of the variable.

`val set: Var<'x> -> 'x -> Job<unit>`

Sets the value of the variable and appends the value to the end of the generated stream. Note that while this operation is atomic, and can be safely used from multiple parallel jobs, a combination of `get` and `set` is not atomic. See also: `MVar<'x>`.

`val tap: Var<'x> -> Stream<'x>`

Returns the generated stream, including the current value of the variable, from the point in time when `tap` is called.

`type MVar<'x>`

Represents a serialized mutable stream variable that generates a stream of values as a side-effect. The difference between `MVar<'x>` and `Var<'x>` is that read-modify-write operations, such as `MVar.updateJob`, are serialized, so they effectively appear as atomic, like with the ordinary `MVar<'x>`.

`module MVar =`

Operations on serialized stream variables.

`val create: 'x -> MVar<'x>`

Creates a new serialized stream variable.

`val get: MVar<'x> -> Job<'x>`

Returns a job that gets the value of the variable.

Creates a job that sets the value of the variable. Note that a combination of `get` and `set` is not serialized. See also: `updateFun`.

```
val updateFun: MVar<'x> -> ('x -> 'x) -> Job<unit>
```

Creates a job that updates the value of the variable with the given function in a serialized fashion. If the function raises an exception, the variable will not be modified. See also: `updateJob`.

```
val updateJob: MVar<'x> -> ('x -> #Job<'x>) -> Job<unit>
```

Creates a job that updates the value of the variable with the given job in a serialized fashion. If the job raises an exception, the variable will not be modified. See also: `updateFun`.

```
val tap: MVar<'x> -> Stream<'x>
```

Returns the generated stream, including the current value of the variable, from the point in time when `tap` is called.

```
type Property<'x> =
```

Represents a mutable property, much like a stream variable, that generates a stream of values and property change notifications as a side-effect.

```
new: 'x -> Property<'x>
```

Creates a new property with the specified initial value.

```
member Value: 'x with get, set
```

Allows to get and set the value of a `Property<'x>`.

```
member Tap: unit -> Stream<'x>
```

Returns the generated stream, including the current value of the property, from the point in time when `Tap` is called.

```
val nil<'x> : Stream<'x>
```

An empty or closed choice stream. `nil` is also the identity element for the `merge` and `append` combinators. `nil` is equivalent to `Promise Nil`.

```
val never<'x> : Stream<'x>
```

A choice stream that never produces any values and never closes. While perhaps rarely used, this is theoretically important as the identity element for the `switch` and `amb` combinators.

```
val cons: 'x -> Stream<'x> -> Stream<'x>
```

`cons x xs` constructs a choice stream whose first value is `x` and the rest of the stream is computed using `xs`. For example, `cons 1 << cons 2 << cons 3 <| cons 4 nil` is a stream producing the sequence `1 2 3 4`. See also: `delay`.

`cons x xs` is equivalent to `Promise (Cons (x, xs))`.

Note that `cons` and `nil` directly correspond to the ordinary list constructors `::` and `[]` and, aside from the obvious notational differences, you can construct choice streams just like you would create ordinary lists.

```
val delay: (unit -> #Job<Cons<'x>>) -> Stream<'x>
```

`delay` creates a stream that is constructed lazily. Use `delay` to make lazy streams and to avoid unbounded eager recursion.

`delay u2xs` is equivalent to `Promise (Job.delay u2xs)`.

Note that with `delay`, `cons` and `nil`, you can express arbitrary lazy streams. For example,

```
let fibs: Stream<BigInteger> =
  let rec lp f0 f1 = cons f0 << delay <| fun () -> lp f1 (f0 + f1)
  lp 0I 1I
```

is the stream of all fibonacci numbers.

The above `fibs` streams produces results lazily, but can do so at a relatively fast rate when it is being pulled eagerly. The following

```
let slowFibs =
  fibs
  |> afterEach (timeOutMillis 1000)
```

stream would produce the fibonacci sequence with at most one element per second.

Constructs a choice stream that is closed with an error.

```
val one: 'x -> Stream<'x>
```

Returns a stream of length one containing the given value. `one x` is equivalent to `cons x nil`.

```
val repeat: 'x -> Stream<'x>
```

Creates an infinite stream of the given value. `repeat x` is equivalent to `cycle <| one x`.

```
val ofSeq: seq<'x> -> Stream<'x>
```

Converts the given sequence to a lazy stream.

```
val once: Job<'x> -> Stream<'x>
```

Returns a lazy stream that, when pulled, runs the given job, produces the result of the job and closes. `once xJ` is equivalent to `indefinitely xJ |> take 1`.

```
val indefinitely: Job<'x> -> Stream<'x>
```

Returns a lazy stream whose elements are generated by running the given job. For example, given a channel, `xCh`, a stream can be created, `indefinitely xCh`, through which all the values given on the channel can be observed. See also: `values`.

Reference implementation:

```
let rec indefinitely xJ = xJ >>=* fun x -> cons x <| indefinitely xJ
```

```
val unfoldJob: ('s -> #Job<option<'x * 's>>) -> ('s -> Stream<'x>)
```

Returns a lazy stream that contains the elements generated by the given job. See also: `foldBack`.

For example, `mapJob` could be defined via `unfoldJob` as follows:

```
let mapJob x2yJ xs =
  xs
  |> unfoldJob (fun xs ->
     xs >>= function Nil -> Job.result None
                   | Cons (x, xs) ->
                     x2yJ x >>- fun y -> Some (y, xs))
```

Reference implementation:

```
let rec unfoldJob f =
  f >=>* function None -> nil
                | Some (x, s) -> cons x <| unfoldJob f s
```

```
val unfoldFun: ('s -> option<'x * 's>) -> ('s -> Stream<'x>)
```

Returns a lazy stream that contains the elements generated by the given function.

```
[<AbstractClass>]
type GenerateFuns<'s, 'x> =
```

Generator functions for `generateFuns`.

```
new: unit -> GenerateFuns<'s, 'x>
```

Default constructor.

```
abstract While: 's -> bool
```

Called to determine whether to generate more.

```
abstract Next: 's -> 's
```

Called to compute next state.

```
abstract Select: 's -> 'x
```

Called to extract value from state.

```
val generateFuns: 's -> GenerateFuns<'s, 'x> -> Stream<'x>
val generateFun: initial: 's
              -> doWhile: ('s -> bool)
              -> doNext: ('s -> 's)
              -> doSelect: ('s -> 'x)
              -> Stream<'x>
```

Generates a stream.

```
val iterateJob: ('x -> #Job<'x>) -> 'x -> Stream<'x>
```

Returns an infinite stream of repeated applications of the given job to the given initial value.

Reference implementation:

```
let rec iterateJob f x = f x >>=* iterateJob f |> cons x
```

```
val iterateFun: ('x -> 'x) -> 'x -> Stream<'x>
```

Returns an infinite stream of repeated applications of the given function to the given initial value.

```
val afterDateTimeOffsets: Stream<DateTimeOffset> -> Stream<DateTimeOffset>
```

Given a stream of dates, returns a stream that produces the dates after the dates.

```
val afterDateTimeOffset: DateTimeOffset -> Stream<DateTimeOffset>
```

Returns a stream that produces the given date after the given date.

```
val afterTimeSpan: TimeSpan -> Stream<unit>
```

Returns a stream that produces an element after the given time span. Note that streams are memoized.

```
val shift: timeout: Job<_> -> Stream<'x> -> Stream<'x>
```

Returns a stream that produces the same sequence of elements as the given stream, but shifted in time by the given timeout.

```
  input: 1        2 3   4        5
timeout: +---x    +---x +---x
                    +---x        +---x
 output:     1        2 3   4        5
```

The `shift` operation pulls the input while the stream returned by `shift` is being pulled. If the stream produced by `shift` is not pulled, `shift` will stop pulling the input. This basically means that the timing of the output can be determined by an eager producer of the input.

Note that this operation has a fairly complex implementation. Unless you absolutely want this behavior, you might prefer a combinator such as `delayEach`.

```
val combineLatest: Stream<'x> -> Stream<'y> -> Stream<'x * 'y>
```

Returns a stream that produces a new pair of elements whenever either one of the given pair of streams produces an element. If one of the streams produces multiple elements before any elements are produced by the other stream, then those elements are skipped. See also: `zip`.

```
 xs: 1 2                  3     4
 ys:     a     b     c
xys:   (2,a) (2,b) (2,c)(3,c) (4,c)
```

```
val debounce: timeout: Alt<_> -> Stream<'x> -> Stream<'x>
```

`debounce timeout elements` returns a stream so that after each element a timeout is started and the element is produced if no other elements is received before the timeout is signaled. Note that if the given stream produces elements more frequently than the timeout, the returned stream never produces any elements.

```
elements: 1        2 3  4     5 6 7 8 9 ...
 timeout: +---x    +-+--+---x +-+-+-+-+-...
  output:     1             4
```

```
val ignoreUntil: timeout: Job<_> -> Stream<'x> -> Stream<'x>
```

`ignoreUntil timeout elements` returns a stream that, after getting an element, starts a timeout and produces the last element received when the timeout is signaled.

```
elements: 1   2      3       4 5 6
timeouts: +-----x    +-----x +-----x
```

```
val ignoreWhile: timeout: Job<_> -> Stream<'x> -> Stream<'x>
```

`ignoreWhile timeout elements` returns a stream that, after getting an element, starts a timeout, produces the element and ignores other elements until the timeout is signaled.

```
elements: 1   2      3       4 5 6
timeouts: +-----x    +-----x +-----x
  output: 1          3       4
```

```
val samplesBefore: ticks: Stream<_> -> Stream<'x> -> Stream<'x>
```

`samplesBefore ticks elements` returns a stream that consumes both ticks and elements and produces each element that precedes a tick. Excess elements from both streams are skipped.

```
elements: 1  2  3        4 5 6  7
   ticks:     x    x    x     x   x
  output:     2    3          6   7
```

```
val samplesAfter: ticks: Stream<_> -> Stream<'x> -> Stream<'x>
```

`samplesAfter ticks elements` returns a stream that consumes both ticks and elements and produces each element that follows a tick. Excess elements from both streams are skipped.

```
elements: 1  2  3        4 5 6  7
   ticks:     x    x    x     x   x
  output:       3        4      7
```

```
val skipUntil: Alt<_> -> Stream<'x> -> Stream<'x>
```

Returns a stream that discards elements from the given stream until the given alternative is committed to after which the remainder of the given stream is produced. Note that `append <| takeUntil evt xs <| skipUntil evt xs` may not be equivalent to `xs`, because there is an inherent race-condition. See also: `takeAndSkipUntil`.

```
val takeUntil: Alt<_> -> Stream<'x> -> Stream<'x>
```

Returns a stream that produces elements from the given stream until the given alternative is committed to after which the returned stream is closed. Note that `append <| takeUntil evt xs <| skipUntil evt xs` may not be equivalent to `xs`, because there is an inherent race-condition. See also: `takeAndSkipUntil`.

```
val takeAndSkipUntil: Alt<_> -> Stream<'x> -> Stream<'x> * Stream<'x>
```

Returns a pair of streams of which the first one takes elements from the given stream and the second skips elements from the given stream until the given alternative is committed to. It is guaranteed that `takeAndSkipUntil evt xs |> fun (hs, ts) -> append hs ts` is equivalent to `xs`. See also: `skipUntil`, `takeUntil`.

```
[<AbstractClass>]
type KeepPrecedingFuns<'x, 'y> =
```

Functions for collecting elements from a live stream to be lazified.

```
new: unit -> KeepPrecedingFuns<'x, 'y>
```

Default constructor.

```
abstract First: 'x -> 'y
```

Called to begin the next batch of elements.

```
abstract Next: 'y * 'x -> 'y
```

Called to add an element to the current batch.

```
val keepPrecedingFuns: KeepPrecedingFuns<'x, 'y> -> Stream<'x> -> Stream<'y>
```

Converts a given imperative live stream into a lazy stream by spawning a job to eagerly consume and collect elements from the live stream using the given `KeepPrecedingFuns<_, _>` object.

```
val keepPreceding: maxCount: int -> Stream<'x> -> Stream<Queue<'x>>
```

Converts a given imperative live stream into a lazy stream of queued elements by spawning a job to eagerly consume and queue elements from the live stream. At most `maxCount` most recent elements are kept in a queue and after that the oldest elements are thrown away. See also: `keepPreceding1`.

```
val keepPreceding1: Stream<'x> -> Stream<'x>
```

stream and to produce only one most recent element each time when requested. See also: `pullOn`, `keepPreceding`, `keepFollowing1`.

Basically,

```
live |> keepPreceding1 |> afterEach timeout
```

is similar to

```
live |> ignoreWhile timeout
```

and

```
live |> keepPreceding1 |> beforeEach timeout
```

is similar to

```
live |> ignoreUntil timeout
```

However, a lazified stream, assuming there is enough CPU time to consume elements from the live stream, never internally holds to an arbitrary number of stream conses.

`val keepFollowing1: Stream<'x> -> Stream<'x>`

Converts an imperative live stream into a lazy stream by spawning a job to eagerly consume (and throw away) elements from the live stream and to produce only one element after each pull request. See also: `pullOn`, `keepPreceding1`.

`val afterEach: timeout: Job<_> -> Stream<'x> -> Stream<'x>`

Returns a stream that produces the same elements as the given stream, but after each element, the given job is used as a delay before a request is made to the given stream for the next element. If the given job fails, the returned stream also fails. See also: `duringEach`.

Suppose that an application needs to poll for some information, e.g. by making a http request, using a job named `poll`. Using `indefinitely` and `afterEach` we can specify a stream for polling:

```
indefinitely poll
|> afterEach (timeOutMillis 10000)
```

The above stream ensures that there is at least 10 seconds of idle time after the previous poll has finished and before a new poll is started. When polls are requested less frequently, there is no delay before a poll.

`val beforeEach: timeout: Job<_> -> Stream<'x> -> Stream<'x>`

Returns a stream that runs the given job each time a value is requested before requesting the next value from the given stream. If the given job fails, the returned stream also fails. `beforeEach yJ xs` is equivalent to `pullOn (indefinitely yJ) xs`.

`val delayEach: timeout: Job<_> -> Stream<'x> -> Stream<'x>`

Returns a stream that produces the same elements as the given stream, but delays each pulled element using the given job. If the given job fails, the returned stream also fails. `delayEach yJ xs` is equivalent to `zipWithFun (fun x _ -> x) xs (indefinitely yJ)`. See also: `shift`.

```
  input: 1        2 3   4        5
timeout: +---x    +---x---x---x  +---x
 output:     1        2   3   4      5
```

In the above, the `input` is considered to be independent of the pull operations performed by `delayEach`. For streams that produce output infrequently in relation to the timeout, `delayEach` behaves similarly to `shift`.

`val duringEach: timeout: Job<_> -> Stream<'x> -> Stream<'x>`

Returns a stream that produces the same elements as the given stream, but runs the given job, which is typically a timeout, in parallel each time a value is requested from the stream and waits for the job to complete before next request. The elements from the underlying stream are not otherwise delayed. If the given job fails, the returned stream also fails when the next element is requested. This means that when the underlying streams ends, failure from the last started job will be ignored. See also: `afterEach`.

Suppose that an application needs to poll for some information, e.g. by making a http request, using a job named `poll`. Using `indefinitely` and `duringEach` we can specify a stream for polling:

```
indefinitely poll
|> duringEach (timeOutMillis 10000)
```

The above stream ensures that polls are started at least 10 seconds apart. When polls are requested less frequently, there is no delay before a poll.

`val pullOn: ticks: Stream<_> -> Stream<'x> -> Stream<'x>`

Given a stream of ticks and a lazy stream of elements returns a stream of elements pulled from the lazy stream based on the ticks.

For example,

```
live |> keepPreceding1 |> pullOn ticks
```

is similar to

```
live |> samplesBefore ticks
```

and

```
live |> keepFollowing1 |> pullOn ticks
```

is similar to

```
live |> samplesAfter ticks
```

However, a lazified stream, assuming there is enough CPU time to consume elements from the live stream, never internally holds to an arbitrary number of stream conses.

`pullOn ts xs` is equivalent to `zipWithFun (fun _ x -> x) ts xs`.

`val zip: Stream<'x> -> Stream<'y> -> Stream<'x * 'y>`

Returns a stream of pairs of elements from the given pair of streams. No elements from either stream are skipped and each element is used only once. In `zip xs ys`, the `xs` stream is examined first. See also: `pullOn`, `combineLatest`.

For example,

```
zip xs <| tail xs
```

is a stream of consecutive pairs from the stream `xs`.

Note that `zip` consumes the same number of elements from both given streams. If one of the streams accumulates elements faster than the other stream, there will be an effective space leak.

`val zipWithFun: ('x -> 'y -> 'z) -> Stream<'x> -> Stream<'y> -> Stream<'z>`

`zipWithFun f xs ys` is equivalent to `zip xs ys |> mapFun (fun (x, y) -> f x y)`.

`val ofObservableOn: subscribeOn: SynchronizationContext -> IObservable<'x> -> Stream<'x>`

Subscribes to the given observable on the specified synchronization context and returns the events pushed by the observable as a stream. A finalizer is used to automatically unsubscribe from the observable after the stream is no longer reachable.

`val ofObservableOnMain: IObservable<'x> -> Stream<'x>`

`ofObservableOnMain x0` is equivalent to `ofObservable main x0`, where `main` is the main synchronization context as set by application code using `setMain`.

`val ofObservable: IObservable<'x> -> Stream<'x>`

`ofObservable x0` is equivalent to `ofObservableOn null x0`. Note that it is often necessary to specify the synchronization context to subscribe on. See also: `Observable.SubscribeOn`.

`val toObservable: Stream<'x> -> IObservable<'x>`

Returns an observable that eagerly consumes the given stream.

`val mapJob: ('x -> #Job<'y>) -> Stream<'x> -> Stream<'y>`

Returns a stream that produces elements passed through the given job whenever the given streams produces elements.

Reference implementation:

```
let rec mapJob f xs =
  xs >>=* function Nil -> nil
                 | Cons (x, xs) ->
                   f x >>=* fun y -> cons y <| mapJob f xs
```

`val mapFun: ('x -> 'y) -> Stream<'x> -> Stream<'y>`

Returns a stream that produces elements passed through the given function whenever the given streams produces elements. `mapFun x2y` is equivalent to `mapJob (Job.lift x2y)`.

`val mapPipelinedJob: degree: int -> ('x -> #Job<'y>) -> Stream<'x> -> Stream<'y>`

parallel.

Note that due to the sequential nature of streams, this simply isn't an effective solution for fine grained parallel processing.

```
val mapPipelinedFun: degree: int -> ('x -> 'y) -> Stream<'x> -> Stream<'y>
```

`mapPipelinedFun degree x2y xs` is like `mapFun x2y xs` except that up to `degree` number of invocations of `x2y` may be run in parallel.

Note that due to the sequential nature of streams, this simply isn't an effective solution for fine grained parallel processing.

```
val mapConst: 'y -> Stream<'x> -> Stream<'y>
```

Returns a stream that produces the given element each time the given stream produces an element.

```
val mapIgnore: Stream<'x> -> Stream<unit>
```

`xs |> mapIgnore` is equivalent to `xs |> mapConst ()`.

```
val chooseJob: ('x -> #Job<option<'y>>) -> Stream<'x> -> Stream<'y>
```

Returns a stream that produces results whenever the given stream produces an element and the given job returns `Some` result from that element.

Reference implementation:

```
let rec chooseJob f xs =
  xs >>=* function Nil -> nil
                 | Cons (x, xs) ->
                   f x >>=* function None -> chooseJob f xs
                                   | Some y -> cons y <| chooseJob f xs
```

```
val chooseFun: ('x -> option<'y>) -> Stream<'x> -> Stream<'y>
```

Returns a stream that produces results whenever the given stream produces an element and the given function returns `Some` result from that element.

```
val choose: Stream<option<'x>> -> Stream<'x>
```

`xs |> choose` is equivalent to `xs |> chooseFun id`.

```
val filterJob: ('x -> #Job<bool>) -> Stream<'x> -> Stream<'x>
```

Returns a stream that contains the elements from the given stream for which the given job returns `true`.

```
val filterFun: ('x -> bool) -> Stream<'x> -> Stream<'x>
```

Returns a stream that contains the elements from the given stream for which the given function returns `true`.

```
val groupByJob: ('k -> Job<unit> -> Stream<'x> -> #Job<'y>) -> ('x -> #Job<'k>) -> Stream<'x> -> Stream<'y> when 'k: equality
```

`groupByJob newGroup keyOf elems` splits the given source stream into substreams or groups based on the keys extracted from the elements by `keyOf` and formed using `newGroup`. See also: `groupByFun`.

New groups are formed by calling the given function with a key, a job for closing the substream and the substream. Unless explicitly closed, substreams remain alive as long as the source stream. When closing substreams, it is important to understand that streams operate concurrently. This means that one should always consume the substream until it ends after closing it. If, after closing a substream, the given stream produces more elements with the same key, a new substream with the key will be opened.

```
val groupByFun: ('k -> Job<unit> -> Stream<'x> -> 'y) -> ('x -> 'k) -> Stream<'x> -> Stream<'y> when 'k: equality
```

`groupByJob newGroup keyOf elems` splits the given source stream into substreams or groups based on the keys extracted from the elements by `keyOf` and formed using `newGroup`. See `groupByJob` for further details.

```
val buffer: int -> Stream<'x> -> Stream<array<'x>>
```

Converts a stream of elements into a stream of non-overlapping buffers of at most given number of elements.

```
val scanJob: ('s -> 'x -> #Job<'s>) -> 's -> Stream<'x> -> Stream<'s>
```

Returns a stream whose elements are computed using the given job and initial state as with `foldJob`.

```
let rec scanJob f s xs =
  xs >>=* function Nil -> nil
                 | Cons (x, xs) ->
                   f s x >>=* fun s -> scanJob f s xs
  |> cons s
```

Returns a stream whose elements are computed using the given function and initial state as with `foldFun`.

`val scanFromJob: 's -> ('s -> 'x -> #Job<'s>) -> Stream<'x> -> Stream<'s>`

`scanFromJob s sx2sJ xs` is equivalent to `scanJob sx2sJ s xs` and is often syntactically more convenient to use.

`val scanFromFun: 's -> ('s -> 'x -> 's) -> Stream<'x> -> Stream<'s>`

`scanFromFun s sx2sJ xs` is equivalent to `scanFun sx2sJ s xs` and is often syntactically more convenient to use.

`val distinctByJob: ('x -> #Job<'k>) -> Stream<'x> -> Stream<'x> when 'k: equality`

Returns a stream that contains no duplicate entries based on the keys returned by the given job.

`val distinctByFun: ('x -> 'k) -> Stream<'x> -> Stream<'x> when 'k: equality`

Returns a stream that contains no duplicate entries based on the keys returned by the given function.

`val distinctUntilChangedWithJob: ('x -> 'x -> #Job<bool>) -> Stream<'x> -> Stream<'x>`

Returns a stream that contains no successive duplicate elements according to the given comparison job.

`val distinctUntilChangedWithFun: ('x -> 'x -> bool) -> Stream<'x> -> Stream<'x>`

Returns a stream that contains no successive duplicate elements according to the given comparison function.

`val distinctUntilChangedByJob: ('x -> #Job<'k>) -> Stream<'x> -> Stream<'x> when 'k: equality`

Returns a stream that contains no successive duplicate elements based on the keys returned by the given job.

`val distinctUntilChangedByFun: ('x -> 'k) -> Stream<'x> -> Stream<'x> when 'k: equality`

Returns a stream that contains no successive duplicate elements based on the keys returned by the given function.

`val distinctUntilChanged: Stream<'x> -> Stream<'x> when 'x: equality`

Returns a stream that contains no successive duplicate elements.

Reference implementation:

```
let distinctUntilChanged xs =
  append (head xs)
   (zip xs (tail xs)
    |> chooseFun (fun (x0, x1) ->
       if x0 <> x1 then Some x1 else None))
```

`val skip: int64 -> Stream<'x> -> Stream<'x>`

`skip n xs` returns a stream without the first `n` elements of the given stream. If the given stream is shorter than `n`, then the returned stream will be empty. Note that if `n` is non-negative, then `append <| take n xs <| skip n xs` is equivalent to `xs`.

`val take: int64 -> Stream<'x> -> Stream<'x>`

`take n` returns a stream that has the first `n` elements of the given stream. If the given stream is shorter than `n`, then `take n` is the identity function. Note that if `n` is non-negative, then `append <| take n xs <| skip n xs` is equivalent to `xs`.

`val skipWhileJob: ('x -> #Job<bool>) -> Stream<'x> -> Stream<'x>`

Returns the stream without the maximal prefix of elements that satisfy the given predicate given as a job.

`val skipWhileFun: ('x -> bool) -> Stream<'x> -> Stream<'x>`

Returns the stream without the maximal prefix of elements that satisfy the given predicate given as a function.

`val takeWhileJob: ('x -> #Job<bool>) -> Stream<'x> -> Stream<'x>`

Returns the maximal prefix of the given stream of elements that satisfy the given predicate given as a job.

`val takeWhileFun: ('x -> bool) -> Stream<'x> -> Stream<'x>`

Returns the maximal prefix of the given stream of elements that satisfy the given predicate given as a function.

`val catch: (exn -> #Stream<'x>) -> Stream<'x> -> Stream<'x>`

structed by calling the given function and that stream becomes the remainder of the stream.

`val onCloseJob: Job<unit> -> Stream<'x> -> Stream<'x>`

Returns a stream that is just like the given stream except that just before the returned stream is closed, due to the given stream being closed, whether with an error or without, the given job is executed. In case the job raises an exception, that exception closes the returned stream. See also: `onCloseFun`, `doFinalizeJob`.

`val onCloseFun: (unit -> unit) -> Stream<'x> -> Stream<'x>`

Returns a stream that is just like the given stream except that just before the returned stream is closed, due to the given stream being closed, whether with an error or without, the given function is called. In case the function raises an exception, that exception closes the returned stream. See also: `onCloseJob`, `doFinalizeFun`.

`val doFinalizeJob: Job<unit> -> Stream<'x> -> Stream<'x>`

Returns a stream that is just like the given stream except that after the returned stream is closed or becomes garbage, the given job is started as a separate concurrent job. See also: `doFinalizeFun`, `onCloseJob`.

`val doFinalizeFun: (unit -> unit) -> Stream<'x> -> Stream<'x>`

Returns a stream that is just like the given stream except that after the returned stream is closed or becomes garbage, a separate job is started that calls the given function. See also: `doFinalizeJob`, `onCloseFun`.

`val count: Stream<'x> -> Job<int64>`

Returns a job that computes the length of the given stream.

`val foldJob: ('s -> 'x -> #Job<'s>) -> 's -> Stream<'x> -> Job<'s>`

Eagerly reduces the given stream using the given job. See also: `foldBack`.

`val foldFun: ('s -> 'x -> 's) -> 's -> Stream<'x> -> Job<'s>`

Eagerly reduces the given stream using the given function.

`val foldFromJob: 's -> ('s -> 'x -> #Job<'s>) -> Stream<'x> -> Job<'s>`

`foldFromJob s sx2sJ xs` is equivalent to `foldJob sx2sJ s xs` and is often syntactically more convenient to use.

`val foldFromFun: 's -> ('s -> 'x -> 's) -> Stream<'x> -> Job<'s>`

`foldFromFun s sx2s xs` is equivalent to `foldFun sx2s s xs` and is often syntactically more convenient to use.

`val foldBack: ('x -> Promise<'s> -> 'sJ) -> Stream<'x> -> 'sJ -> Promise<'s> when 'sJ :> Job<'s>`

Performs a lazy backwards fold over the stream. See also: `foldJob`, `unfoldJob`, `foldFromBack`.

`foldBack` is a fundamental function on streams. Consider that `foldBack cons xs nil` is equivalent to `xs`. Many other stream functions can be implemented using `foldBack`. For example, `mapJob` could be defined using `foldBack` as follows:

```
let mapJob x2yJ xs =
  foldBack (fun x s -> x2yJ x >>=* fun y -> cons y s) xs nil
```

Note that `foldFromBack` allows the same to written more concisely.

Reference implementation:

```
let rec foldBack x2s2sJ xs s =
  xs >>=* function Nil -> s
                 | Cons (x, xs) -> x2s2sJ x <| foldBack x2s2sJ xs s
```

`val foldFromBack: 'sJ -> (Promise<'s> -> 'x -> 'sJ) -> Stream<'x> -> Promise<'s> when 'sJ :> Job<'s>`

`foldFromBack s s2x2sJ xs` is equivalent to `foldBack (flip s2x2sJ) xs s` and is often syntactically more convenient to use.

For example, here is how one could write `mapJob` using `foldFromBack`:

```
let mapJob x2yJ =
  foldFromBack nil <| fun s -> x2yJ >=>* flip cons s
```

Contrast the above to how the same can be written using `foldBack`.

`val tryPickJob: ('x -> #Job<option<'y>>) -> Stream<'x> -> Job<option<'y>>`

`tryPickJob x2yOJ xs` returns a job that returns the first `Some y` result produced by the job `x2yOJ x` for an element `x` of the `xs` stream.

`tryPickFun x2yO xs` is equivalent to `tryPickJob (x2yO >> result) xs`.

`val iterJob: ('x -> #Job<unit>) -> Stream<'x> -> Job<unit>`

Returns a job that iterates the given job constructor over the given stream. See also: `consumeJob`.

Reference implementation:

```
let rec iterJob x2uJ xs =
  xs >>= function Nil -> Job.unit ()
                | Cons (x, xs) -> x2uJ x >>=. iterJob x2uJ xs
```

`val iterFun: ('x -> unit) -> Stream<'x> -> Job<unit>`

Returns a job that iterates the given function over the given stream. See also: `iterJob`, `consumeFun`.

`val iter: Stream<'x> -> Job<unit>`

Returns a job that iterates over all the elements of the given stream. `iter xs` is equivalent to `iterFun ignore xs`. See also: `consume`.

`val consumeJob: ('x -> #Job<unit>) -> Stream<'x> -> unit`

`xs |> consumeJob x2uJ` is equivalent to `xs |> iterJob x2uJ |> queue`.

`val consumeFun: ('x -> unit) -> Stream<'x> -> unit`

`xs |> consumeFun x2u` is equivalent to `xs |> iterFun x2u |> queue`.

`val consume: Stream<'x> -> unit`

`xs |> consume` is equivalent to `xs |> iter |> queue`.

`val head: Stream<'x> -> Stream<'x>`

Returns a stream containing only the first element of the given stream. If the given stream is closed, the result stream will also be closed. Note that `append <| head xs <| tail xs` is equivalent to `xs`.

`val tail: Stream<'x> -> Stream<'x>`

Returns a stream just like the given stream except without the first element. If the given stream is closed, the result stream will also be closed. Note that `append <| head xs <| tail xs` is equivalent to `xs`.

`val init: Stream<'x> -> Stream<'x>`

Returns a stream with all the elements of the given stream except the last element. If the stream is closed, a closed stream is returned. Note that `append <| init xs <| last xs` is equivalent to `xs`.

`val last: Stream<'x> -> Stream<'x>`

Returns a stream containing the last element of the given stream. If the given stream is closed, a closed stream is returned. Note that `append <| init xs <| last xs` is equivalent to `xs`.

`val inits: Stream<'x> -> Stream<Stream<'x>>`

Returns a stream of all initial segments of the given stream from shortest to longest.

`val tails: Stream<'x> -> Stream<Stream<'x>>`

Returns a stream of all final segments of the given stream from longest to shortest.

Reference implementation:

```
let rec tails xs =
  xs >>=* function Nil -> nil
                 | Cons (_, xs) -> tails xs
  |> cons xs
```

`val initsMapFun: (Stream<'x> -> 'y) -> Stream<'x> -> Stream<'y>`

`initsMapFun xs2y xs` is equivalent to `inits xs |> mapFun xs2y`.

`val tailsMapFun: (Stream<'x> -> 'y) -> Stream<'x> -> Stream<'y>`

`tailsMapFun xs2y xs` is equivalent to `tails xs |> mapFun xs2y`.

Concatenates the given two streams. In other words, returns a stream that first produces all the elements from first stream and then all the elements from the second stream. If the first stream is infinite, `append` should not be used, because no elements would be produced from the second stream. See also: `appendMap`, `nil`.

```
val appendAll: Stream<#Stream<'x>> -> Stream<'x>
```

Joins all the streams together with `append`.

```
val appendMap: ('x -> #Stream<'y>) -> Stream<'x> -> Stream<'y>
```

Maps and joins all the streams together with `append`. This is roughly the same function as `Seq.collect`, but is probably less frequently used with choice streams.

```
val amb: Stream<'x> -> Stream<'x> -> Stream<'x>
```

Of the two given streams, returns the stream that first produces an element or is closed. See also: `ambMap`, `never`.

Reference implementation:

```
let amb ls rs = ls <|>* rs
```

```
val merge: Stream<'x> -> Stream<'x> -> Stream<'x>
```

Returns a stream that produces elements from both of the given streams so that elements from the streams are interleaved non-deterministically in the returned stream. See also: `mergeMap`, `nil`.

Reference implementation:

```
let rec mergeSwap ls rs =
  ls ^=> function Nil -> rs
                | Cons (l, ls) -> cons l <| merge rs ls
and merge ls rs = mergeSwap ls rs <|>* mergeSwap rs ls
```

```
val switch: Stream<'x> -> Stream<'x> -> Stream<'x>
```

Returns a stream that produces elements from the first stream as long as the second stream produces no elements. As soon as the second stream produces an element, the returned stream only produces elements from the second stream. See also: `switchTo`, `switchMap`, `never`.

```
 first: a b    c   d
second:      1  2 3  4 ...
output: a b  1  2 3  4 ...
```

Reference implementation:

```
let rec switch ls rs =
       rs
  <|>* ls ^=> function Nil -> rs
                     | Cons (l, ls) -> cons l <| switch ls rs
```

```
val switchTo: Stream<'x> -> Stream<'x> -> Stream<'x>
```

`switchTo lhs rhs` is equivalent to `switch rhs lhs`.

`switchTo` is designed to be used in pipelines:

```
firstStream
|> switchTo otherStream
```

```
val ambAll: Stream<#Stream<'x>> -> Stream<'x>
```

Joins all the streams together with `amb`.

```
val mergeAll: Stream<#Stream<'x>> -> Stream<'x>
```

Joins all the streams together with `merge`.

```
val switchAll: Stream<#Stream<'x>> -> Stream<'x>
```

Joins all the streams together with `switch`.

```
val ambMap: ('x -> #Stream<'y>) -> Stream<'x> -> Stream<'y>
```

Maps and joins all the streams together with `amb`. This corresponds to the idea of starting several alternative streams in parallel and

```
val mergeMap: ('x -> #Stream<'y>) -> Stream<'x> -> Stream<'y>
```

Maps and joins all the streams together with `merge`. This corresponds to interleaving results based on all sources of information. While this is a theoretically important combinator, `mergeMap` is probably not the most useful binding form on choice streams.

```
val switchMap: ('x -> #Stream<'y>) -> Stream<'x> -> Stream<'y>
```

Maps and joins all the streams together with `switch`. This is perhaps the most useful binding form with choice streams as this correspond to the idea of producing results based only on the latest source of information.

```
[<AbstractClass>]
type Builder =
```

A generic builder for streams. The abstract `Combine` and `Zero` operations need to be implemented in a derived class. The operations are then used to implement `Bind`, `For` and `While` to get a builder with consistent semantics.

```
new: unit -> Builder
```

Default constructor.

```
abstract Combine': Alt<Cons<'x>> * Alt<Cons<'x>> -> Alt<Cons<'x>>
```

Called to combine two streams.

```
abstract Zero: unit -> Stream<'x>
```

Called to obtain the zero or unit stream corresponding to the `Combine` method.

```
member Combine: Stream<'x> * Stream<'x> -> Stream<'x>
```

`this.Combine (xs, ys)` is equivalent to `this.Combine' (xs, ys) |> memo`.

```
member Bind: Stream<'x> * ('x -> Stream<'y>) -> Stream<'y>
```

`this.Bind (xs, x2ys)` is equivalent to `mapJoin (fun x y -> this.Combine (x, y)) x2ys xs`.

```
member Delay: (unit -> Stream<'x>) -> Stream<'x>
```

`this.Delay u2xs` is equivalent to `delay u2xs`.

```
member For: seq<'x> * ('x -> Stream<'y>) -> Stream<'y>
```

`this.For (xs, x2ys)` is equivalent to `this.Bind (ofSeq xs, x2ys)`

```
member TryWith: Stream<'x> * (exn -> Stream<'x>) -> Stream<'x>
```

`this.TryWith (xs, e2xs)` is equivalent to `catch e2xs xs`.

```
member While: (unit -> bool) * Stream<'x> -> Stream<'x>
```

`this.While (u2b, xs)` is equivalent to
`delay <| fun () -> if u2b () then this.Combine (xs, this.While (u2b, xs)) else this.Zero ()`

```
member Yield: 'x -> Stream<'x>
```

`this.Yield x` is equivalent to `one x`.

```
member YieldFrom: Stream<'x> -> Stream<'x>
```

`this.YieldFrom xs` is equivalent to `xs`.

```
val ambed: Builder
```

This builder joins substreams with `amb'` to produce a stream with the first results.

```
val appended: Builder
```

This builder joins substreams with `append'` to produce a stream with all results in sequential order.

```
val merged: Builder
```

This builder joins substreams with `merge'` to produce a stream with all results in completion order.

This builder joins substreams with `switch'` to produce a stream with the latest results.

```
val cycle: Stream<'x> -> Stream<'x>
```

Creates an infinite repetition of the given stream. For infinite streams `cycle` is the identity function.

```
val values: Stream<'x> -> Alt<'x>
```

Creates an alternative through which all the values of the stream generated after the point at which the alternative has been created can be read. See also: `indefinitely`.

```
val amb': Alt<Cons<'x>> -> Alt<Cons<'x>> -> Alt<Cons<'x>>
```

Primitive version of `amb`.

```
val append': Alt<Cons<'x>> -> Alt<Cons<'x>> -> Alt<Cons<'x>>
```

Primitive version of `append`.

```
val merge': Alt<Cons<'x>> -> Alt<Cons<'x>> -> Alt<Cons<'x>>
```

Primitive version of `merge`.

```
val switch': Alt<Cons<'x>> -> Alt<Cons<'x>> -> Alt<Cons<'x>>
```

Primitive version of `switch`.

```
val joinWith: ('y -> Alt<Cons<'z>> -> #Job<Cons<'z>>) -> Stream<'y> -> Stream<'z>
```

Joins all the streams in the given stream of streams together with the given binary join combinator primitive.

```
val mapJoin: ('y -> Alt<Cons<'z>> -> #Job<Cons<'z>>) -> ('x -> 'y) -> Stream<'x> -> Stream<'z>
```

`mapJoin j f xs` is equivalent to `joinWith j <| mapFun f xs`.

```
val toSeq: Stream<'x> -> Job<ResizeArray<'x>>
```

Returns a job that collects all the elements from the stream. This function is provided for testing purposes.

```
[<AutoOpen>]
module Hopac =
```

Convenience bindings for programming with Hopac.

```
type Stream<'x> = Stream.Stream<'x>
```

Represents a non-deterministic stream of values called a choice stream.

```
val job: JobBuilder
```

Default expression builder for jobs.

```
val onMain: Extensions.Async.OnWithSchedulerBuilder
```

Builder for running an async workflow on the main synchronization context and interoperating with Hopac. The application must call `Hopac.Extensions.Async.setMain` to configure Hopac with the main synchronization context.

The builder has been constructed by calling `Hopac.Extensions.Async.Global.onMain ()`.

```
val queue: Job<unit> -> unit
```

Queues the given job for execution. See also: `start`, `server`.

Note that using this function in a job workflow is not optimal and you should use `Job.queue` instead.

```
val queueIgnore: Job<_> -> unit
```

Queues the given job for execution. `queueIgnore xJ` is equivalent to `Job.Ignore xJ |> queue`.

```
val queueDelay: (unit -> #Job<_>) -> unit
```

Queues the given delayed job for execution. `queueDelay u2xJ` is equivalent to `queueIgnore <| Job.delay u2xJ`.

```
val server: Job<Void> -> unit
```

more lightweight manner.

Note that using this function in a job workflow is not optimal and you should use `Job.server` instead.

```
val start: Job<unit> -> unit
```

Starts running the given job, but does not wait for the job to finish. See also: `queue`, `server`.

Note that using this function in a job workflow is not optimal and you should use `Job.start` instead.

```
val startIgnore: Job<_> -> unit
```

Starts running the given job, but does not wait for the job to finish. `startIgnore xJ` is equivalent to `Job.Ignore xJ |> start`.

```
val startDelay: (unit -> #Job<_>) -> unit
```

Starts running the given delayed job, but does not wait for the job to finish. `startDelay u2xJ` is equivalent to `startIgnore <| Job.delay u2xJ`.

```
val run: Job<'x> -> 'x
```

Starts running the given job and then blocks the current thread waiting for the job to either return successfully or fail. See also: `start`.

WARNING: Use of `run` should be considered carefully, because calling `run` from an arbitrary thread can cause deadlock.

`run` is mainly provided for conveniently running Hopac code from F# Interactive and can also be used as an entry point to the Hopac runtime in console applications. In Windows applications, for example, `run` should not be called from the GUI thread.

A call of `run xJ` is safe when the call is not made from within a Hopac worker thread and the job `xJ` does not perform operations that might block or that might directly, or indirectly, need to communicate with the thread from which `run` is being called.

Note that using this function from within a job workflow should never be needed, because within a workflow the result of a job can be obtained by binding.

```
val runDelay: (unit -> #Job<'x>) -> 'x
```

`runDelay u2xJ` is equivalent to `run <| Job.delay u2xJ`.

```
val queueAsTask: Job<'x> -> Task<'x>
```

Queues the given job for execution. The result can be obtained from the returned task.

```
val startAsTask: Job<'x> -> Task<'x>
```

Starts running the given job. The result can be obtained from the returned task.

```
val startWithActions: (exn -> unit) -> ('x -> unit) -> Job<'x> -> unit
```

Starts running the given job, but does not wait for the job to finish. Upon the failure or success of the job, one of the given actions is called once.

Note that using this function in a job workflow is not optimal and you should instead use `Job.start` with the desired exception handling construct (e.g. `Job.tryIn` or `Job.catch`).

```
val timeOut: TimeSpan -> Alt<unit>
```

Creates an alternative that, after instantiation, becomes available after the specified time span.

Note that the timer mechanism is simply not intended for high precision timing and the resolution of the underlying mechanism is very coarse (Windows system ticks).

Also note that you do not need to create a new timeout alternative every time you need a timeout with a specific time span. For example, you can create a timeout for one second

```
let after1s = timeOut <| TimeSpan.FromSeconds 1.0
```

and then use that timeout many times

```
    makeRequest ^=> fun rp -> ...
<|> after1s     ^=> fun () -> ...
```

Timeouts, like other alternatives, can also directly be used as job level operations. For example, using the above definition of `after1s`

```
after1s >>= fun () -> ...
```

has the effect of sleeping for one second.

It is an idiomatic approach with Hopac to rely on garbage collection to clean up concurrent jobs than can no longer make progress. It is therefore important to note that a server loop

```
      ...
   <|> timeOut ... ^=> ... serverLoop ...
   <|> ...
```

that always waits for a timeout is held live by the timeout. Such servers need to support an explicit kill protocol.

When a timeout is used as a part of a non-deterministic choice, e.g. `timeOut span <|> somethingElse`, and some other alternative is committed to before the timeout expires, the memory held by the timeout can be released by the timer mechanism. However, when a timeout is not part of a non-deterministic choice, e.g.

```
 timeOut span >>=. gotTimeout *<= () |> start
```

no such clean up can be performed. If there is a possibility that such timeouts are kept alive beyond their usefulness, it may be possible to arrange for the timeouts to be released by making them part of a non-deterministic choice:

```
     timeOut span ^=> IVar.tryFill gotTimeoutOrDoneOtherwise
 <|> gotTimeoutOrDoneOtherwise
  |> start
```

The idea is that the `gotTimeoutOrDoneOtherwise` is filled, using `IVar.tryFill` as soon as the timeout is no longer useful. This allows the timer mechanism to release the memory held by the timeout.

A timeout of zero is optimized to an alternative that is immediately available, while a negative timeout results in an `Alt` that is never available. See `idle` for an alternative that yields the thread of execution to any ready work before becoming available.

`val timeOutMillis: int -> Alt<unit>`

`timeOutMillis n` is equivalent to `timeOut << TimeSpan.FromMilliseconds <| float n`.

`val idle: Alt<unit>`

Creates an alternative that yields the thread of execution to any ready work and then becomes available.

This is similar to `timeOutMillis 0` except that it yields the current thread of execution to any other ready work before attempting to become available.

When executing multiple concurrent jobs that do not have natural points where they yield execution, `idle` allows progress to be made by other ready work.

`val memo: Job<'x> -> Promise<'x>`

Creates a promise whose value is computed lazily with the given job when an attempt is made to read the promise.

`val asAlt: Alt<'x> -> Alt<'x>`

Use object as alternative. This function is a NOP and is provided as a kind of syntactic alternative to using a type ascription or an `upcast`.

`val asJob: Job<'x> -> Job<'x>`

Use object as job. This function is a NOP and is provided as a kind of syntactic alternative to using a type ascription or an `upcast`.