

*Recalled: An EDSL for persistent, incremental, parallel computations*

Vesa Karvonen

## About me

- Been programming since 1990
    - Real-time Graphics (demoscene) Atari ST, Amiga, PC
      - Assembly (68k, x86, SH4), SW/HW interface, pointers, bit manipulation, instruction scheduling, caches, DMA, blitter, graphics chips, interrupts, ...
  - Professionally since 1996 (mostly tech for console games)
    - C++
      - Got me interested in languages
        - Any higher level language is “easy” (*good thing!*) if you can master C++
      - Metaprogramming for fun
        - Wrote a full Scheme/ML like language interpreter with C preprocessor (~2004)
        - Wrote first non-strict C++ template metaprogramming library (~2004)
  - Functional programming since 2001
    - First project was a very simple compiler using OCaml
    - Scheme (~2002-2007), Standard ML (~2005-2008), F# (~2008-)
  - Long time interest in language oriented programming & reusable code
- 
-

## *Recalled*

- A library for persistent, incremental, parallel computations
    - “*Such as build systems*”
  - *Loosely* inspired by
    - Self-Adjusting Computation (Umut Acar)
      - Semantics: values not side-effects
    - Shake (Neil Mitchell)
      - Convenience & problem domain
  - Goals
    - Make it ***easy to define*** such computations and
    - Scale such computations by
      - running computations in *parallel* and
      - *distributing results* over a network of workstations
- 
-

## *Everyone's favourite function*

```
let rec fib n =  
  logAs ("fib " + n.ToString ()) {  
    if n < 2I then  
      return n  
    else  
      let! xL = fib (n-2I)  
      let! yL = fib (n-1I)  
  
      let! x = read xL  
      let! y = read yL  
  
      return x + y  
  }
```

---

---

## *Everyone's favourite function with monad ops*

```
let rec fib n =  
  logAs ("fib " + n.ToString ()) << delay <| fun () ->  
    if n < 2I then  
      result n  
    else  
      fib (n-2I) >>= fun xL ->  
      fib (n-1I) >>= fun yL ->  
  
      read xL >>= fun x ->  
      read yL >>= fun y ->  
  
      result (x + y)
```

---

---

## *Everyone's favourite function with some types*

```
let rec fib (n: BigInteger) : LogAs<Logged<BigInteger>> =  
  logAs ("fib " + n.ToString ()) {  
    if n < 2I then  
      return n  
    else  
      let! (xL : Logged<BigInteger>) = fib (n-1I)  
      let! (yL : Logged<BigInteger>) = fib (n-2I)  
  
      let! (x : BigInteger) = read xL  
      let! (y : BigInteger) = read yL  
  
      return x + y  
  }
```

---

---

Let's run it!



## *What is in a name?*

- “Recall”
  - Bring back from memory
- “Recall”
  - To call previous number



## *“Such as build systems”*

- Recalled
    - doesn't use time stamps
    - doesn't check file dates
    - doesn't run external programs
  - Recalled is
    - based on the idea of *logged* computations that
      - return **values** and
      - may **depend** on other computations.
    - Recompute when dependencies or their result values change
    - You can write your own build system like primitive computations
    - And *straightforwardly* write *forward build system* code.
- 
-

## *Primitive Computations*

- If computation does not depend on anything
  - it is considered *primitive* and
  - always run to completion

```
let written (path: string) = logAs (sprintf "written \"%s\"" path) {  
    return File.GetLastWriteTimeUtc path  
}
```

## Dependent Computations

- Other computations are *dependent* and run to completion
  - on the first time
  - if some known dependency has changed
  - otherwise the remainder of the computation is skipped and old result reused

```
let md5 (path: string) = logAs (sprintf "md5 \"%s\"" path) {  
    do! depAs (written path)  
    -----  
    let result =  
        let md5 = MD5.Create ()  
        use stream =  
            new FileStream (path, FileMode.Open, FileAccess.Read)  
        md5.ComputeHash stream  
    return result  
}
```

## *Partial logged computations*

- Sometimes you don't want to log the result of the computation, because the result can be quickly computed anyway

```
let allLines (path: string) = update {  
  do! depAs (md5 path)  
  return File.ReadAllLines path  
}
```

## *Dynamic dependencies*

- Computations can log dependencies after examining their inputs

```
let md5s (listPath: string) = logAs (sprintf "md5s \"%s\"" listPath) {  
    let! paths = allLines listPath  
    let! md5Ls = paths |> Seq.mapLogAs md5  
    let! md5s = md5Ls |> Seq.mapUpdate read  
    return md5s.ToArray ()  
}
```

## *A couple of things to note*

- Computations are always reconstructed (*called again*)
    - No need to serialize closures!
  - Primitive computations (leafs) are always run to completion
    - Traditional build system would also need to do that
  - Dependent computations are run until dependencies are checked
    - Most traditional systems do that too  $\rightarrow O(n+d)$
  - Dependencies can be dynamic
    - Computation can examine input and log as many dependencies as needed
- 
-

## *Slightly trickier things*

- Recalled code can be evolved, but needs a bit of care
  - Must make sure at least one previously known dependency changes
  - Conversely, to fool the mechanism, add new dependency after previous ones
- Recalled works with *typed values*, but *side-effects* are possible too
  - Always need side-effect → Wrap as a side-effecting dependency

Let's see..

---

---

## *Revisioning Computations: The easy case*

```
type TexInfo = {  
    w: int  
    h: int  
  
    bpp: int  
}  
  
let texInfo path =  
    logAs (sprintf "texInfo \"%s\"" path) {  
        do! depAs (md5 path)  
        ...  
        return {w = ...; h = ...; bpp = ...}  
    }
```

```
type TexInfo = {  
    w: int  
    h: int  
    hasAlpha: bool  
    bpp: int  
}  
  
let texInfo path =  
    logAs (sprintf "texInfo2 \"%s\"" path) {  
        do! depAs (md5 path)  
        ...  
        return {w = ...  
                h = ...  
                hasAlpha = ...  
                bpp = ...}  
    }
```



## *Revisioning Computations: The tricky case*

```
let sorted path =  
  logAs (sprintf "sorted \"%s\"" path) {  
    let! lines = allLines path  
    return lines  
      |> Array.sort  
  }
```

```
let sorted path =  
  logAs (sprintf "sorted \"%s\"" path) {  
    do! watch 1  
    let! lines = allLines path  
    return lines  
      |> Array.sort  
      |> Array.distinct  
  }
```

- We don't want to rerun dependents of sorted unless output changes

## Tricky side-effects

```
let copy (src: string) (dst: string) =  
  logAs (sprintf "copy \"%s\" \"%s\"" src dst) {  
    let! srcInfo = written src |> waitAs  
    do File.Copy (src, dst, true)  
    return srcInfo  
  }
```

```
let copy (src: string) (dst: string) =  
  logAs (sprintf "copy \"%s\" \"%s\"" src dst) {  
    let! srcInfo = written src |> waitAs  
    do! wait <| log {  
      if not (File.Exists dst) ||  
        File.GetLastWriteTimeUtc dst <> srcInfo  
      then File.Copy (src, dst, true)  
    }  
    return srcInfo  
  }
```

- Left copies when src changes
- Right copies when src or dst changes
- Will likely design special primitive for this in future
- But not very important for main problem domain...

## *Original Problem Domain*

- Preprocess resources for console games
  - Textures, meshes, anims, scripts, localization, game specific ad hoc data, ...
  - Produce a “stream file”
    - Think: zip file optimized for streaming from optical media and with dependencies
  - Why?
    - Avoid complex & expensive data manipulation at game run-time
    - Reduce load times

## *Workflow*

- Developers (artists, designers, programmers)
    - Repeatedly
      - Get latest sources including graphics assets
      - Edit what they are working on
      - Export
      - Build
      - Test
      - Commit changes
  - Every single item has been built once at the point the source committed
  - Makes sense to distribute results of expensive computations
- 
-

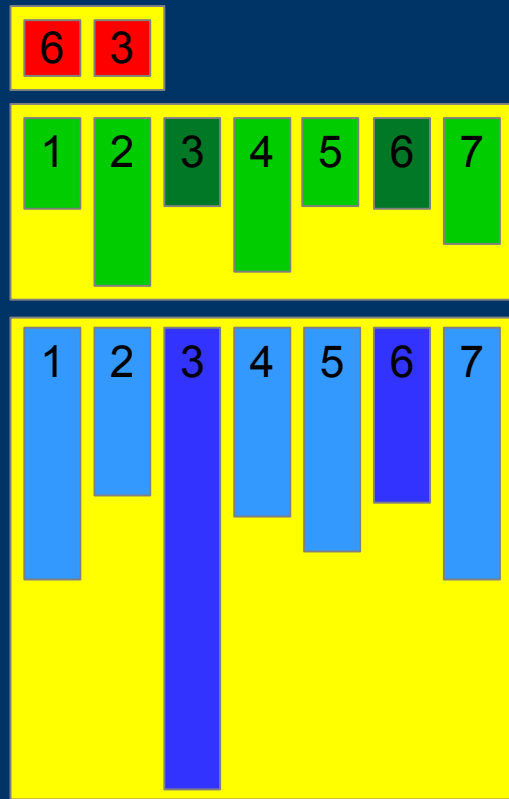
## Challenges

- More heterogeneous than typical software build
    - Often just want some custom data → Language based approach
  - Preprocess with
    - own libraries
    - (buggy) external programs and libs
      - Shader compiler uses dll with global variables → Crashes if you run it in parallel!
  - Very large number of items to build (100k-500k items)
  - Must run with ordinary workstations
    - Hard limits on memory use
      - Parallel alg → #cores x memory usage
  - Gigabytes of source data
  - Takes tens of minutes to build from scratch in parallel
  - Every second wasted is wasted for N people M times a day
- 
-

## *Persistence at IO bandwidths*

- Recalled crucially depends on persistence solution performance
    - Every logged computation must be read
  - Design
    - Store just enough data to match computations and check dependencies
      - 128-bit digests using MurmurHash3
        - Identities
        - Results
      - Combined digest of identity and dependencies identifies result → Key for distribution
    - Log structured storage: new items added to the end of files
    - Separate *add*, *remove* and *bob* (binary object) logs
    - Memory mapped buffers
      - Serialization with direct aligned reads and writes
      - Approach memory bandwidth for some ops
      - Read and write in parallel
  - Less than 0.04 seconds on this laptop to read 32k *add* log entries
- 
-

## Log Architecture



- Top is removed entries
- Middle is add entries
- Bottom is bob entries
  - One bob per add
- Two entries removed
- Add entries processed linearly
  - Rem entries sorted in-place first
- Lookups satisfied as items added
  - So, computations can proceed before all add entries have been read

## Infrastructure

- **Infers**

- A library for *type directed programming* in F#
- **Makes light work of datatype generic functions**
- View member functions as **Horn clauses**
  - *prove value of desired type can be created by invoking the member functions*

- **Hopac**

- Concurrent ML + Cilk for F#
  - **Practical to spawn every logged computation as a separate I-w thread**
  - “Async +”
    - parallel work distributing scheduler
    - negative acknowledgements
    - synchronous channels with simple rendezvous
  - Scales to multiple cores
  - Allows asynchronous and distributed operations without blocking native threads
- 
-



## *Serialization aka Pickling*

```
type [<AbstractClass>] PU<'x> =  
  abstract Size: 'x -> int  
  abstract Dopickle: 'x * nativeptr<byte> -> unit  
  abstract Unpickle: nativeptr<byte> -> 'x
```

- Pickle directly to memory mapped buffer
  - Unpickle directly from memory mapped buffer
- 
-

## Inference Rules

```
type [<InferenceRules>] PU =  
  static member Get: unit -> PU<'t>  
  
  member toPU: OpenPU<'t> -> PU<'t>  
  
  member byte: OpenPU<byte>  
  member DateTime: OpenPU<DateTime>  
  member BigInteger: OpenPU<BigInteger>  
  
  member bytes: OpenPU<array<byte>>  
  member array: OpenPU<'t> -> OpenPU<array<'t>>
```

```
  member elem: Elem<'e, 'es, 't>  
    * OpenPU<'e>  
    -> ProductPU<'e, 'es, 't>  
  member times: ProductPU<'e, 'es, 't>  
    * ProductPU<'e, 'es, 't>  
    -> ProductPU<And<'e, 'es>, And<'e, 'es>, 't>  
  member product: Rep  
    * Product<'t>  
    * AsProduct<'es, 't>  
    * ProductPU<'es, 'es, 't>  
    -> OpenPU<'t>
```

- Rules for union types and all primitives not shown
- Note specialization: bytes
- Datatype generics using Rep module via sums and products
- Rules invoked (via Reflection) by Infers to *build value of desired type*

## *Infers Invocation*

```
static member makePU () : PU<'x> =  
  lock typeof<PU> <| fun () ->  
    match Engine.TryGenerate (PU ()) with  
    | None -> failwithf "PU: %A" typeof<'x>  
    | Some pu -> pu
```

```
static member Get () = memoize PU.makePU
```

- The resolution engine matches rule result types to desired type
  - Prefers specific rules to less specific rules (more general unifier)
  - Recursively builds arguments of rules
- Invokes rule functions
- Returns result as proof

# Hopac

```
type Job<'x>
val (>>=): Job<'x> -> ('x -> Job<'y>) -> Job<'y>
val result: 'x -> Job<'x>
val queue: Job<_> -> Job<unit>

type Alt<'x> :> Job<'x>
val (>>=?): Alt<'x> -> ('x -> Job<'y>) -> Alt<'y>
val choose: seq<Alt<'x>> -> Alt<'x>
val guard: Job<Alt<'x>> -> Alt<'x>
val withNack: (Alt<unit> -> Job<Alt<'x>>) -> Alt<'x>

type Ch<'x>
val give: Ch<'x> -> 'x -> Alt<unit>
val take: Ch<'x> -> Alt<'x>

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

- Poor man's threads + CML + Parallel Scheduler
- Alternatives
  - Selective
  - Higher-order
  - Negative acknowledgement
- Synchronous channels
  - Simple rendezvous
- Plus many more primitives
  - IVar, MVar, Mailbox, ...

## *On Hopac*

- Carefully tweaked implementation
  - ~50 million msgs/s on this laptop (with power connected)
  - ~20 million spawns/s on this laptop (with power connected)
- Scheduler designed to scale
  - Run lightweight threads in a co-operative fashion
  - One native thread per hardware thread
  - Each thread has its own stack of work
  - Share work when runs out
  - Some work remains to make it scale as well as .Net allows
- In Recalled
  - Cheap lightweight threads allow every computation to be a separate lightweight thread
  - Some concurrent modules (MemMapBuf, LoggedMap, ...)
  - Will use more when get to work with the planned distribution mechanism
  - User code can make use of arbitrary parallel, async, concurrent code within computations
    - E.g. Make sure that shader compiler is not run in parallel

## *Recalled Future Work*

- 1.0.0
    - Implement log clean up (1 day of work)
    - Finalize interface
  - 2.0.0
    - Result distribution system
    - Design for change propagation
      - To support “build agent” programming
        - Agent runs in background
        - Builds after changes to files
      - Concept of primitive computation probably needs to be extended
- 
-

Questions?

