

# Typed Tagless Final Bioinformatics

Sebastien Mondet (@smondet)

Compose :: Conference, *Thursday, May 18, 2017.*



**Mount  
Sinai**

# Context

Seb: Software Engineering / Dev Ops at the **Hammer Lab**.



We're a [team](#) of software developers and data scientists [working](#) to understand and improve how the immune system battles cancer.



We occasionally [blog](#) about our work. Please [contact](#) us if you're interested in one of the [jobs](#) we have available!

We are grateful to the [Icahn School of Medicine at Mount Sinai](#), the [Parker Institute for Cancer Immunotherapy](#), and [Neon Therapeutics](#) for funding our work.

# Context

Was *here* 2 years ago to present:

- ▶ **Ketrew**: a workflow engine for complex computational pipelines.
  - ▶ EDSL/library to write programs that build workflows/pipelines
  - ▶ A separate application, The “Engine”, orchestrates those workflows
- ▶ **Biokepi**: a library of Ketrew “nodes” for **Bioinformatics**.

Now

- ▶ Used with GCloud/Kubernetes, AWS, YARN (incl. Spark).
- ▶ `Tyxml1_js` + react **WebUI**
- ▶ *Personalized Genomic Vaccine* clinical trial (NCT02721043) → `hammerlab/epidisco/`

# WebUI ⇒ 3.6 MB GIFs

Client XHR  About

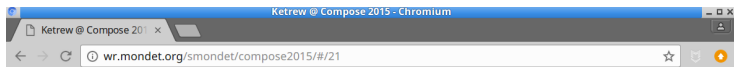
Status Target-table ([1, 10] of 40)

Showing 10 per page Show filters Columns Start [1, 10] Previous 10 Next 10 End [31, 40]

	Index	Name	Unique Id	Backend	Tags	Status
<input checked="" type="checkbox"/> 1	1	"du -sh /home/ubuntu" on /home/ubuntu/KT	ketrew_2015-08-31-24h18m42s307ms-UTC_089809344	daemonize		Finished
<input checked="" type="checkbox"/> 1	2	"sleep 5" on /home/ubuntu/KT	ketrew_2015-08-31-24h18m33s876ms-UTC_089809344	daemonize		Finished
<input checked="" type="checkbox"/> 1	3	build-all-docs	ketrew_2015-08-31-24h14m03s636ms-UTC_671426885	daemonize	build-all-docs	Finished
<input checked="" type="checkbox"/> 1	4	index-page	ketrew_2015-08-31-24h14m03s635ms-UTC_754955217	daemonize	build-all-docs	Finished
<input checked="" type="checkbox"/> 1	5	docof-trakeva	ketrew_2015-08-31-24h14m03s634ms-UTC_844678370	daemonize	build-all-docs	Finished
<input checked="" type="checkbox"/> 1	6	docof-oredoc	ketrew_2015-08-31-24h14m03s632ms-UTC_308219921	daemonize	build-all-docs	Finished
<input checked="" type="checkbox"/> 1	7	docof-ketrew	ketrew_2015-08-31-24h14m03s631ms-UTC_930807020	daemonize	build-all-docs	Finished
<input checked="" type="checkbox"/> 1	8	docof-pvem_lwt_unix	ketrew_2015-08-31-24h14m03s628ms-UTC_781944104	daemonize	build-all-docs	Finished
<input checked="" type="checkbox"/> 1	9	docof-pvem	ketrew_2015-08-31-24h14m03s626ms-UTC_641907500	daemonize	build-all-docs	Finished
<input checked="" type="checkbox"/> 1	10	docof-docout	ketrew_2015-08-31-24h14m03s625ms-UTC_332180439	daemonize	build-all-docs	Finished

# In Particular, We Presented:

Cool experiment: GADT-based, very high-level pipeline EDSL.



## GADTs, Hammers, and Nails

We have open-sourced [hammerlab/biokepi](https://github.com/hammerlab/biokepi).

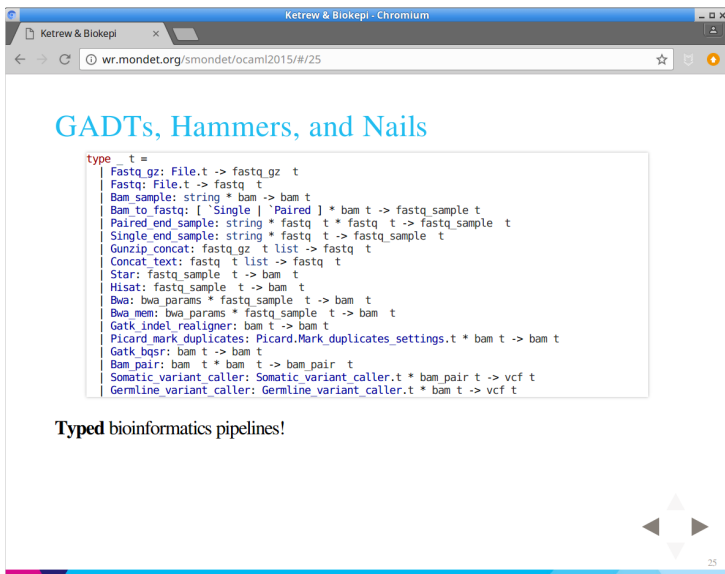
```
type _t =
| Fastq_gz: File.t -> fastq_gz t
| Fastq: File.t -> fastq t
| Paired_end_sample: string * fastq t * fastq t -> fastq_sample t
| Single_end_sample: string * fastq t -> fastq_sample t
| Gunzip_concat: fastq_gz t list -> fastq t
| Concat_text: fastq t list -> fastq t
| Bwa: bwa_params * fastq_sample t -> bam t
| Gatk_indel_realigner: bam t -> bam t
| Picard_mark_duplicates: bam t -> bam t
| Gatk_bqsr: bam t -> bam t
| Bam_pair: bam t * bam t -> bam_pair t
| Mutect: bam_pair t -> vcf t
| Somaticsniper: [ `S of float ] * [ `T of float ] * bam_pair t -> vcf t
| Varscan: [ Adjust_mapq of int option ] * bam_pair t -> vcf t
```

**Typed** bioinformatics pipelines!



# Then, At OCaml / ICFP 2015

Cool experiment: add tools / *tool-kinds*:




The screenshot shows a Chromium browser window with the address bar displaying `wr.mondet.org/smondet/ocaml2015/#/25`. The slide content is as follows:

## GADTs, Hammers, and Nails

```
type _ t =
| Fastq_gz: File.t -> fastq_gz t
| Fastq: File.t -> fastq t
| Bam_sample: string * bam -> bam t
| Bam_to_fastq: [ `Single | `Paired ] * bam t -> fastq_sample t
| Paired_end_sample: string * fastq t * fastq t -> fastq_sample t
| Single_end_sample: string * fastq t -> fastq_sample t
| Gunzip_concat: fastq_gz t list -> fastq t
| Concat_text: fastq t list -> fastq t
| Star: fastq_sample t -> bam t
| Hisat: fastq_sample t -> bam t
| Bwa: bwa_params * fastq_sample t -> bam t
| Bwa_mem: bwa_params * fastq_sample t -> bam t
| Gatk_indel_realigner: bam t -> bam t
| Picard_mark_duplicates: Picard.Mark_duplicates_settings.t * bam t -> bam t
| Gatk_bqsr: bam t -> bam t
| Bam_pair: bam t * bam t -> bam_pair t
| Somatic_variant_caller: Somatic_variant_caller.t * bam_pair t -> vcf t
| Germline_variant_caller: Germline_variant_caller.t * bam t -> vcf t
```

**Typed** bioinformatics pipelines!



# And Soon After

Kept growing, became the default...

```
type _ t =  
| Fastq_gz: File.t -> fastq_gz t  
| Fastq: File.t -> fastq t  
| Bam_sample: string * bam -> bam t  
| Bam_to_fastq: [ `Single | `Paired ] * bam t -> fastq_sample t  
| Paired_end_sample: fastq_sample_info * fastq t * fastq t -> fastq_sample t  
| Single_end_sample: fastq_sample_info * fastq t -> fastq_sample t  
| Gunzip_concat: fastq_gz t list -> fastq t  
| Concat_text: fastq t list -> fastq t  
| Star: Star.Configuration.Align.t * fastq_sample t -> bam t  
| Hisat: Hisat.Configuration.t * fastq_sample t -> bam t  
| Stringtie: Stringtie.Configuration.t * bam t -> gtf t  
| Bwa: Bwa.Configuration.Aln.t * fastq_sample t -> bam t  
| Bwa_mem: Bwa.Configuration.Mem.t * fastq_sample t -> bam t  
| Mosaik: fastq_sample t -> bam t  
| Gatk_indel_realigner: Gatk.Configuration.indel_realigner * bam t -> bam t  
| Picard_mark_duplicates: Picard.Mark_duplicates_settings.t * bam t -> bam t  
| Gatk_bqsr: (Gatk.Configuration.bqsr * bam t) -> bam t  
| Bam_pair: bam t * bam t -> bam_pair t  
| Somatic_variant_caller: somatic Variant_caller.t * bam_pair t -> vcf t  
| Germline_variant_caller: germline Variant_caller.t * bam t -> vcf t  
| Seq2HLA: fastq_sample t -> seq2hla_hla_types t  
| Optitype: ([`DNA | `RNA] * fastq_sample t) -> optitype_hla_types t  
| With_metadata: metadata_spec * 'a t -> 'a t
```

# Very Concise Pipelines

```
let crazy_example ~normal_fastqs ~tumor_fastqs ~dataset =
  let open Pipeline.Construct in
  let normal = input_fastq ~dataset normal_fastqs in
  let tumor = input_fastq ~dataset tumor_fastqs in
  let bam_pair ?gap_open_penalty ?gap_extension_penalty () =
    let normal =
      bwa ?gap_open_penalty ?gap_extension_penalty normal
      |> gatk_indel_realigner |> picard_mark_duplicates |> gatk_bqsr in
    let tumor =
      bwa ?gap_open_penalty ?gap_extension_penalty tumor
      |> gatk_indel_realigner |> picard_mark_duplicates in
    pair ~normal ~tumor in
  let bam_pairs = [
    bam_pair ();
    bam_pair ~gap_open_penalty:10 ~gap_extension_penalty:7 ();
  ] in
  let vcfs =
    List.concat_map bam_pairs ~f:(fun bam_pair ->
      [
        mutect bam_pair;
        somaticsniper bam_pair;
        somaticsniper ~prior_probability:0.001 ~theta:0.95 bam_pair;
        varscan_somatic bam_pair;
        strelka ~configuration:Strelka.Configuration.exome_default bam_pair;
      ])
  in
  vcfs
```



# Type Information

```
14 |> gatk_indel_realigner
13 |> picard_mark_duplicates
12 |> gatk_bqsr
11 |   in
10 | pair ~normal ~tumor in
9 | let bam_pairs = [
8 |   bam_pair ();
7 |   bam_pair ~gap_open_penalty:10 ~gap_extension_penalty:7 ();
6 | ] in
5 | let vcfs =
4 |   List.concat_map bam_pairs ~f:(fun bam_pair ->
3 |     [
2 |       mutect bam_pair;
1 |       somaticsniper bam_pair;
56 |       somaticsniper ~prior_probability:0.001 ~theta:0.95 bam_pair;
1 |       varscan_somatic bam_pair;
2 |       strelka ~configuration:Strelka.Configuration.exome_default bam_pair;
3 |     ])
4 |   in
5 |   vcfs
6 |
NORMAL > master > src/lib/common_pipelines.ml ocaml < utf-8[unix] < 75% : 56: 15 <
?prior_probability:float ->
?theta:float -> Pipeline.bam_pair Pipeline.t -> Pipeline.vcf Pipeline.t
Press ENTER or type command to continue
```

Merlin + Biokepi

# There's a "But"

Fancy but not that practical:

- ▶ `Pipeline.t` is getting too big
  - ▶ Just `compile_aligner_step` is about 170 lines of pattern-matching
  - ▶ Still missing proper `lambda/apply`, list functions, etc.
- ▶ Not Extensible
  - ▶ Adding new types is pretty annoying.
  - ▶ Optimization passes need to deal with whole language at once, always.
  - ▶ Optimization are not proper language transformations.

# Try Again

We want what we already have + *users* of the library to be able to:

- ▶ Extend the language to their needs
- ▶ Re-use default compilers when implementing theirs
- ▶ Write future-proof optimizations
- ▶ Do transformations “by hand” if easier than an optimization pass

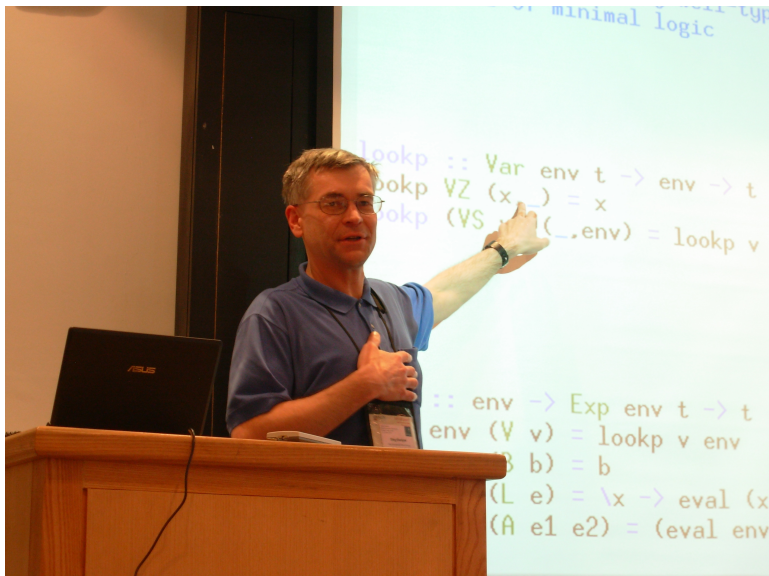
# Not-Really Extensible Hacks

Tried a few experiments:

- ▶ extensible types
  - ▶ lose a lot of the type-strength benefits
  - ▶ are not *that* extensible
- ▶ basic “language” based-on GADTs and extensible bioinformatics atoms
  - ▶ could have worked further but not really extensible either

# Oleg

“We trivially and elegantly solved that problem 20 years ago!”



# Que $\Lambda$ and The Course Notes

First:

- ▶ Oleg emailed the OCaml mailing-list on 2015-07-15
- ▶ Presenting “Que $\Lambda$ ”, first just some `.tar.gz` and draft paper; then it got to PEPM’16  $\rightarrow$  DOI:2847538.2847542).
- ▶ Asked the author for an actual repo and licence  
 $\rightarrow$  `bitbucket.org/knih/que1`.
- ▶ It uses modules *and* the EDSL is well typed.

@**pveber** pointed us to Oleg’s course:

- ▶ In Haskell (very concise code, very *un-modular*).
- ▶ Well explained and progressive.

$\Rightarrow$  Follow the course; with Que $\Lambda$ ’s help; in a Biokepi-like setting.

# And We Did It

Up Next

## Module Biokepi.EDSL (.ml)

```
module EDSL: sig .. end
```

The Embedded Bioinformatics Domain Specific Language

This Embedded DSL is implemented following the "Typed Tagless Final Interpreter" method.

It's usage is as follows:

- Write EDSL expressions inside a functor taking the module type `Biokepi.EDSL.Semantics` (i.e. the definition of the EDSL) as argument. Export some of them with the `observe` function.
- Apply the functor the desired "compilers/interpreters." The interpreter can themselves be functors.

Example:

```
module Pipeline_1 (Bfx : Biokepi.EDSL.Semantics) = struct

  (* Reusable function withing the EDSL: *)
  let align_list_of_single_end_fastqs (l : string list) : [ `Bam ] Bfx.repr =
    let list_expression : [ `Fastq ] list Bfx.repr =
      list.map l ~f:(fun path ->
        (* create [ `Fastq ] repr term: *)
        Bfx.fastq ~sample_name:"Test" ~r1:path ())
    in
    > Bfx.list (* Assemble OCaml list into an EDSL list *)
  in
  let aligner : ([ `Fastq ] -> [ `Bam ] Bfx.repr =
    (* create an EDSL-level function with `lambda': *)
    Bfx.lambda (fun fq -> Bfx.bwa_aln ~reference_build:"hg19" fq)
  in
  (* Call the aligner on all fastq-terms and then merge the result
  into a single bam: *)
  Bfx.list_map list_expression ~f:aligner > Bfx.merge_bams
```

# We TTFI-ed Everything

And it's more powerful:

- ▶ More constructs: `lambda/apply`, `list` and `pair` functions, ...
- ▶ Easier to document.
- ▶ Easier to maintain.
- ▶ Extensible by the users.

And keeps growing:

```
$ grep 'val ' src/pipeline_edsl/semantics.ml | wc -l
```

56



# How Does It Work?

Now tutorial mode:

- ▶ GADT dumb example.
- ▶ Translation to TTFI.
- ▶ Show how to manipulate the *pseudo-AST*.
- ▶ Show how to extend the EDSL.

# First, Quickly, GADTs

## Type Constraints + Existential Types:

```
type _ t =  
| Int: int -> int t  
| True: bool t  
| False: bool t  
| Equal: 'a t * 'a t -> bool t  
  
let rec eval: type v. v t -> v =  
  function  
  | Int i -> i  
  | True -> true  
  | False -> false  
  | Equal (a, b) -> (=) (eval a) (eval b)  
  
let () = assert (eval (Int 42) = 42)  
let () = assert (eval (Equal (True, (Equal (Int 42, Int 42)))) = true)
```

# GADT Usages

- ▶ Existentials to “pack” types applied to different type parameters:
  - ▶ `type pack = Deal_with_it: 'a * 'a how_to -> pack`
- ▶ EDSLs ☺
  - ▶ Generate POSIX-shell scripts & one-liners: `hammerlab/genspio`
  - ▶ Tezos: 250-LoC mutually recursive GADT definition of a smart-contract language:  
`/src/proto/alpha/script_typed_ir.ml`
- ▶ Difference-lists:
  - ▶ Cf. `Printf.printf.`
  - ▶ Or `Eliom_service.create.`
- ▶ Session types.

# TTFI

Type Constraints + Existential Types, using module types and functors:

```
module type Symantics = sig
  type 'a repr
  val int: int -> int repr
  val t: bool repr
  val f: bool repr
  val equal: 'a repr -> 'a repr -> bool repr
end

module Eval_ocaml : Symantics with type 'a repr = 'a = struct
  type 'a repr = 'a
  let int i = i
  let t = true
  let f = false
  let equal a b = (a = b)  (* Cheating a bit *)
end

module Examples (EDSL: Symantics) = struct
  let ex1 = EDSL.int 42
  let ex2 = EDSL.(equal t (equal (int 42) (int 42)))
end

let () =
  let module Compiled_examples = Examples(Eval_ocaml) in
  assert (Compiled_examples.ex1 = 42);
  assert (Compiled_examples.ex2 = true);
  ()
```

# TTFI in Bullet Points

In OCaml:

- ▶ **defintion of the language:** module type Semantics
- ▶ **program:** functor: Semantics -> whatever
- ▶ **compiler:** module implementing Semantics
- ▶ **optimization/transformation:** functor: Semantics -> Semantics
- ▶ **optimization framework:** functor + GADT that implements “default behavior”

# Mysteriously Useful Bit

More jargon: “*observations*” are useful artifacts of optimization passes:

```
module type Symantics = sig
  type 'a repr
  val int: int -> int repr
  val t: bool repr
  val f: bool repr
  val equal: 'a repr -> 'a repr -> bool repr
  type 'a observation
  val observe: (unit -> 'a repr) -> 'a observation
end
```

# To-String Compiler

```
module Eval_string
  : Symantics with type 'a repr = string and type 'a observation = string
  = struct
  type 'a repr = string
  let int = string_of_int
  let t = "True"
  let f = "False"
  let equal a b = Printf.sprintf "(%s = %s)" a b
  type 'a observation = string
  let observe f = f ()
end
module More_examples (EDSL: Symantics) = struct
  let ex1 =
    let open EDSL in
    observe (fun () -> int 42)
  let ex2 =
    let open EDSL in
    observe (fun () ->
      equal (equal t t) (equal (int 42) (int 43))
    )
  end
  let () =
    let module Compiled_examples = More_examples(Eval_string) in
    Printf.printf "Ex1: %s\nEx2: %s\n%"
      Compiled_examples.ex1 Compiled_examples.ex2;
    ()
  end
end
```

Ex1: 42

Ex2: ((True = True) = (42 = 43))

# Simple Optimization Example

We can do some rewriting with functors:

```
module True_equal_true_true (Input: Symantics)
  : Symantics with type 'a observation = 'a Input.observation
  = struct
    include Input
    let t = Input.(equal t t)
end

let () =
  let module Compiled_examples = More_examples(True_equal_true_true(Eval_string)) in
  Printf.printf "Ex1: %s\nEx2: %s\n%!";
  Compiled_examples.ex1 Compiled_examples.ex2;
  ()
```

EX1: 42

EX2: (((True = True) = (True = True)) = (42 = 43))

(this works without the 'a observation thing ...)



# Not Enough

For more complex/interesting transformations,  
what we really want is to “match term with”:

```
type _ t =
| Int: int -> int t
| True: bool t
| False: bool t
| Equal: 'a t * 'a t -> bool t

let rec transform_equal_true_true : type v. v t -> v t =
  function
  | Int i -> Int i
  | True -> True
  | False -> False
  | Equal (True, True) -> True (* Optimization Pass ! *)
  | Equal (a, b) ->
    Equal (transform_equal_true_true a, transform_equal_true_true b)

let () =
  assert (
    transform_equal_true_true (Equal (False, (Equal (True, True))))
    =
    (Equal (False, True))
  )
```

# Optimization Framework

Some type-hackery later ... *A Generic Extensible Optimization Pass Generator.*

```
module type Transformation_base = sig
  type 'a from
  type 'a term
  val fwd : 'a from -> 'a term (* reflection *)
  val bwd : 'a term -> 'a from (* reification *)
end
module Generic_optimizer
(X: Transformation_base)
(Input: Symantics with type 'a repr = 'a X.from)
: Symantics
  with type 'a repr = 'a X.term
  and type 'a observation = 'a Input.observation
= struct
  open X
  type 'a repr = 'a term
  let int i = fwd (Input.int i)
  let t = fwd Input.t
  let f = fwd Input.f
  let equal a b =
    fwd (Input.equal (bwd a) (bwd b))
  type 'a observation = 'a Input.observation (* Here we "get out" ! *)
  let observe f =
    Input.observe (fun () -> bwd (f ()))
end
```

# Using The Optimization Framework

So we want to do | Equal (True, True) -> True:

```
module True_true (Input: Symantics) = struct
  module Transformation = struct
    type 'a from = 'a Input.repr
    type 'a term =
      | Unknown: 'a from -> 'a term
      | Equal: 'a term * 'a term -> bool term
      | True: bool term
    let fwd x = Unknown x
    let rec bwd : type a. a term -> a from = function
      | Unknown x -> x
      | Equal (True, True) -> Input.t
      | Equal (a, b) -> Input.equal (bwd a) (bwd b)
      | True -> Input.t
  end
  module Language_delta = struct
    let equal a b = Transformation.Equal (a, b)
    let t = Transformation.True
  end
  include Generic_optimizer(Transformation)(Input)
  include Language_delta
end
```

# Using the Optimization Pass

Still just a functor to apply “*in the chain:*”

```
let () =  
  let module Compiled = More_examples(Eval_string) in  
  let module Optimized = More_examples(True_true(Eval_string)) in  
  Printf.printf "Compiled: %s\nOptimized: %s\n%!"  
    Compiled.ex2 Optimized.ex2
```

Success!

Compiled: ((True = True) = (42 = 43))

Optimized: (True = (42 = 43))

# Extensions

Some `include`, and module *sub-typing* magic:

```
module type Symantics_with_lambdas = sig
  include Symantics
  (** Lambda abstraction *)
  val lambda : ('a repr -> 'b repr) -> ('a -> 'b) repr
  (** Application *)
  val apply : ('a -> 'b) repr -> 'a repr -> 'b repr
end

module Eval_string_with_lambdas
  : Symantics_with_lambdas
  with type 'a repr = string and type 'a observation = string
  = struct
  include Eval_string
  open Printf
  let lambda f =
    let var = sprintf "%d" (Random.int 1000) in
    sprintf "(λ %s → %s)" var (f var)
  let apply f x =
    sprintf "(%s %s)" f x
end
```

# Use The Extension

```
module Example_with_lambdas (EDSL : Symantics_with_lambdas) = struct
  open EDSL
  let l1 =
    lambda (fun x -> equal x t)
  let ex1 =
    observe (fun () -> l1)
  let ex2 =
    observe (fun () -> apply l1 (equal t t))
  (* Of course still type checked:
  let ex2 =
    observe (fun () -> apply l1 (int 42))
  Error: This expression has type int repr
       but an expression was expected of type bool repr
       Type int is not compatible with type bool
  *)
end

let () =
  let module Compiled = Example_with_lambdas(Eval_string_with_lambdas) in
  Printf.printf "Ex1: %s\nEx2: %s\n%"
    Compiled.ex1 Compiled.ex2
```

Ex1: ( $\lambda x370 \rightarrow (x370 = \text{True})$ )

Ex2: ( $(\lambda x370 \rightarrow (x370 = \text{True})) (\text{True} = \text{True})$ )

# Extend The Generic Optimization Thing

Soooo *meta*:

```
module Generic_optimizer_with_lambdas
  (X: Transformation_base)
  (Input: Symantics_with_lambdas with type 'a repr = 'a X.from)
  : Symantics_with_lambdas
    with type 'a repr = 'a X.term
    and type 'a observation = 'a Input.observation
= struct
  open X
  include Generic_optimizer(X)(Input)
  let lambda f = fwd (Input.lambda (fun x -> bwd (f (fwd x))))
  let apply e1 e2 = fwd (Input.apply (bwd e1) (bwd e2))
end
```

# Extend The Optimization Pass

True\_true does not touch the new stuff:

```
module True_true_with_lambdas (Input: Symantics_with_lambdas) = struct
  module Previous_true_true = True_true(Input)
  include Generic_optimizer_with_lambdas(Previous_true_true.Transformation)(Input)
  include Previous_true_true.Language_delta
end

let () =
  let module Compiled = Example_with_lambdas(Eval_string_with_lambdas) in
  let module Optimized =
    Example_with_lambdas(True_true_with_lambdas(Eval_string_with_lambdas)) in
  Printf.printf "Ex2 normal: %s\nEx2 optimized: %s\n!"
    Compiled.ex2 Optimized.ex2
```

Ex2 normal: ((λ x20 → (x20 = True)) (True = True))

Ex2 optimized: ((λ x921 → (x921 = True)) True)



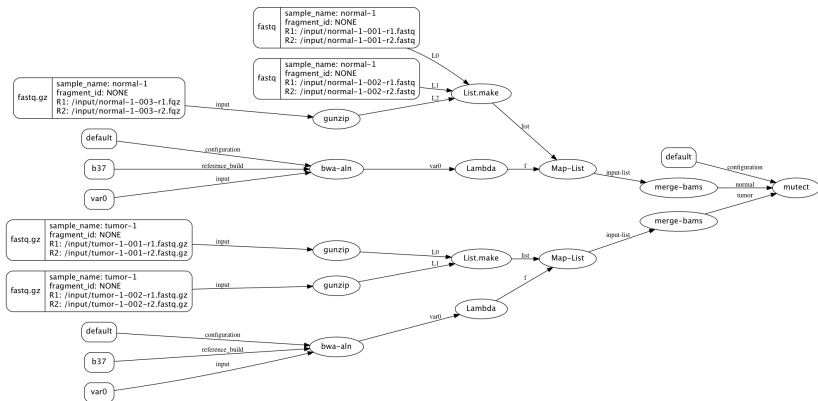
# Back To Biokepi

Fully replaced the GADT-based EDSL:

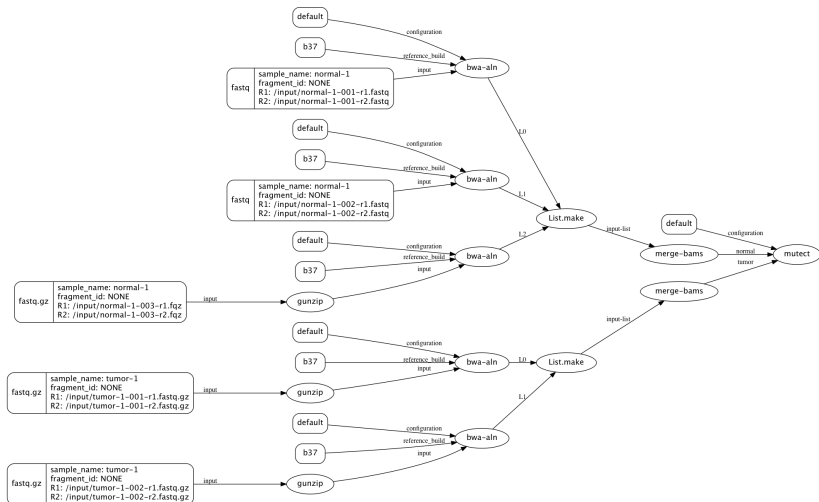
- ▶ Compiles to:
  - ▶ Ketrew workflows.
  - ▶ JSON “provenance proofs.”
  - ▶ Display-friendly, high-level, Dot-graphs.
- ▶ Optimizations *not that* useful:
  - ▶ In our application, it’s mostly for display/readability purposes.

# Apply Lambdas

From PR #236:



# For A Nice Display



# Epidisco

Big (family of) pipeline(s) that drive a clinical trial and other people's analyses:

`hammerlab/epidisco/`

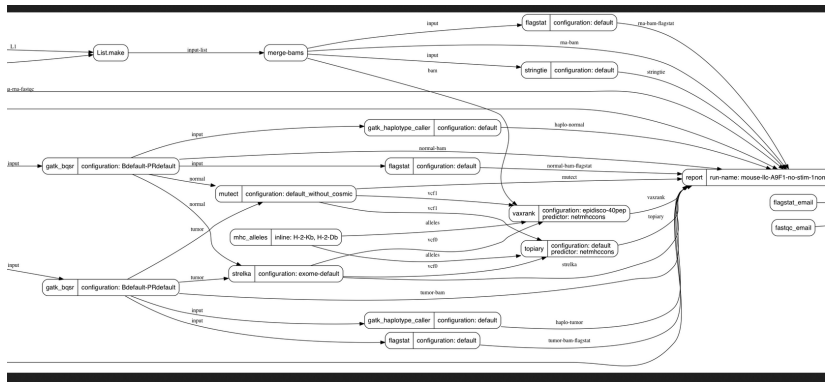
Cf. output to dot-graphs:



We actually do extend the EDSL:

- ▶ Custom HTML “report.”
- ▶ Custom “saving” of important artifacts.

# Zoom



# Deal With Insanity

```
~
~
~configuration:(mark_dups_config parameters.Parameters.picard_java_max_heap)
254   in
255 + let bam = List.map samples ~f:sample_to_bam
256 +     |> Bfx.list
257 +     |> Bfx.merge_bams in
258 + (* We split out the spliced and non-spliced reads so that we can run indel
259 +   realignment on all reads that don't span a splice junction (and thus
260 +   cause the GATK IndelRealigner we're using to crash.) We then merge the
261 +   spliced reads back in. *)
262 + let spliced_bam =
263 +   let filter = Biokepi.Tools.Sambamba.Filter.Defaults.only_split_reads in
264 +   Bfx.sambamba_filter ~filter bam in
265 + let indel_realigned_bam =
266 +   let filter = Biokepi.Tools.Sambamba.Filter.Defaults.drop_split_reads in
267 +   Bfx.sambamba_filter ~filter bam
268 +   |> Bfx.gatk_indel_realigner
269 +   ~configuration:indel_realigner_config
270 + in
271 + Bfx.merge_bams @@ Bfx.list [spliced_bam; indel_realigned_bam]
```

# Limitations

## *Minor issues:*

- ▶ Applying functors, while conceptually simple, scares beginners.
  - ▶ Though they *can* → PR #429.
- ▶ Losing type variance because of the *optimization framework*.
  - ▶ And in our case optimization framework is useful only for display.
- ▶ Cannot always use sub-modules because of `include`.
  - ▶ Hence the *flat/tagged* API with `list_map`, `pair_first`, `pair_second`, ...

The End

Questions?

