Typed Tagless Final Bioinformatics

Sebastien Mondet (@smondet)


Context

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

WebUI ⇒ 3.6 MB GIFs

In Particular, We Presented:

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

Then, At OCaml / ICFP 2015

Cool experiment: add tools / tool-kinds:

Context

Was here 2 years ago to present:

• Kettle: 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 Kettle “nodes” for Bioinformatics.

Now

• Used with GCloud/Kubernetes, AWS, YARN (incl. Spark).
• Tyxml.js + react WebUI
• Personalized Genomic Vaccine clinical trial (NCT02721043) → hammer-lab/epidisco/
And Soon After

Kept growing, became the default…

```ml
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 vcf =
    List.concat_map bam_pairs ~f:fun bam_pair ->
    [ mutect 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
  vcf
```

Type Information

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
  - loose 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!”
QueΛ and The Course Notes

First:
- Oleg emailed the OCaml mailing-list on 2015-07-15
- Presenting "QueΛ", first just some .tar.gz and draft paper; then it got to PEPM'16 → DOI:2847538.2847542
- Asked the author for an actual repo and licence → bitbucket.org/knih/quel.
- 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.
⇒ Follow the course; with QueΛ’s help; in a Biokepi-like setting.

And We Did It

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 =
| Int i -> i
| True -> true
| False -> false
| Equal (a, b) -> (eval a) (eval b)

let () = assert (eval (Int 42) = 42)

GADT Usages
- Existentials to "pack" types applied to different type parameters:
- EDSLs ☺
- Difference-lists:
- Session types.

TTFI

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

```
module type Symantics = sig
  val int: int -> int repr
  val bool: 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 bool = 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 (Int 42) (Int 42))
end

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

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
We can do some rewriting with functors:

```
let module Simple Optimization Example end
```

The functors allow us to define new types and operations that can be used to rewrite terms:

```
module More jargon: observations are useful artifacts of optimization passes:

val (type 'a repr) and type 'a observation = string
```

The `More jargon` section explains the definition of the `Observations` type, which is used to represent the results of optimization passes.

To-String Compiler

```
module Eval_string :
| 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 t = t
| end
```

The `Eval_string` module defines a type `Observations` and provides functions to convert between `Observations` and strings.

Simple Optimization Example

We can do some rewriting with functors:

```
module True_equal_true_true (Input : Syntactics) :
| type 'a observation = 'a Input.observaton
| struct
| include Input
| let t = Input.(equal t t)
| end
```

The `Simple Optimization Example` section shows how to rewrite terms using functors.

Not Enough

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

```
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, False))
```

The `Not Enough` section shows an example of a more complex transformation.

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
| let bwd = Input.fwd (Input
| val bwd : 'a term -> 'a from
| let observe f t = f t
| end
```

The `Optimization Framework` section explains the definition of the `Transformation_base` type.

Using The Optimization Framework

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

```
module True_equal_true_true (Input : Syntactics) = struct
| module Transformation = struct
| type 'a from = 'a Input.observaton
| 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.observaton (bwd a) (bwd b)
| | True -> Input.t
| end
```

The `Using The Optimization Framework` section shows how to use the `Transformation_base` type to define new transformations.

Using The Optimization Framework

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

```
module True_equal_true_true (Input : Syntactics) = struct
| module Transformation = struct
| type 'a from = 'a Input.observaton
| 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.observaton (bwd a) (bwd b)
| | True -> Input.t
| end
```

The `Using The Optimization Framework` section shows how to use the `Transformation_base` type to define new transformations.
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
Optimized: %s
%!

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 "x%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
  Print.printf "Ex1: %s
Ex2: %s
%!

Compiled: ex1 Compiled: ex2
Ex1: (λ x370 → (x370 = True))
Ex2: ((λ x921 → (x921 = True)) True)

Extend The Optimization Pass

True_true does not touch the new stuff:

module True_true_with_lambdas : Symantics_with_lambdas = struct
  let module Compiled = Example_with_lambdas(True_true(Eval_string_with_lambdas)) in
  Printf.printf "Ex2 normal: %s
Ex2 optimized: %s
%!

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.

Deal With Insanity

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?