Runtime Types

(for the outside world)

Patrik Keller @ LexiFi

25.10.2018

Reimplementing LexiFi Runtime Types as a syntactic preprocessor

It is often useful to get access to types at runtime in order to implement generic type-driven operations. A typical example is a generic pretty-printer. Unfortunately, the OCaml compiler does not keep type information at runtime. At LexiFi, we have extended OCaml to support runtime types. This extension has been in use for years and is now a key element in many of our interesting components, such as our automatic GUI framework (which derives GUIs from type definitions) or our high-level database layer (which derives SQL schema from type definitions, and exposes a well-typed interface for queries). This extension is tightly integrated with the OCaml typechecker, which allows the compiler to synthesize the runtime type representations with minimal input from the programmer.

First Contact

```
type foo = { bar: string }

let () = debug { bar= "Is this Python?" }
```

First Contact

```
let debug ~t x =
  let open Format in
  match stype_of_ttype t with
  | DT_int -> pp_print_int ppf (Obj.magic x)
  | DT_string -> pp_print_string ppf (Obj.magic x)
  .
  .
  .
```

Further Investigations

Further Investigations

Stype

- untyped representation of OCaml types
- serializable
- magic everywhere

Further Investigations

Stype

- untyped representation of OCaml types
- serializable
- magic everywhere

Ttype

- stype & OCaml type
- safe consumption of APIs
- inspection requires downgrade to stype $\longrightarrow$ magic

Further Investigations

Stype

- untyped representation of OCaml types
- serializable
- magic everywhere

Ttype

- stype & OCaml type
- safe consumption of APIs
- inspection requires downgrade to stype $\longrightarrow$ magic

Xtype

- typed representation of OCaml types
- safe implementation of APIs
- potential performance penalty

Agenda

Agenda

Create runtime types using PPX

Agenda

Create runtime types using PPX

New Xtype

Agenda

Create runtime types using PPX

New Xtype

Pattern matching on runtime types

Syntax Extension

Syntax Extension

Lexifi compiler produces ttypes at compile time.

A PPX should be able to do the same.

```
type foo = { bar: string }

let () = debug { bar = "Is this Python?" }
```

Syntax Extension

Lexifi compiler produces ttypes at compile time.

A PPX should be able to do the same.

```
type foo = { bar: string } [@@deriving t]

let () = debug ~t:[%t: foo] { bar = "No it's not!" }
```

Syntax Extension

```
type foo = { bar: string }

let (foo_t : foo ttype) =
  let foo_node = create_node "foo" [] in
  let () = set_fields foo_node [("bar", [], string_t)] in
  Obj.magic (DT_node foo_node)

let () = debug ~t:foo_t { bar = "No it's not!" }
```

Syntax Extension

```
type foo = { bar: string }
let _ = fun (_ : foo) -> ()
let (foo_t : foo Ttype.t) =
  let open! Dynt_ppx_runtime.Types in
    let foo_node = create_node "foo" [] in
    let rec (foo_t : foo Ttype.t lazy_t) =
      let _ = foo_t in lazy (ttype_of_stype (DT_node foo_no
    let () =
      let meta = [("bar", [])] in
      let args = [stype_of_ttype (string_t : string Ttype.t
      set_node_record foo_node
        ((rev_map2 (fun (n, p) -> fun a -> (n, p, a)) meta
          (record_representation args)) in
    force foo_t
let _ = foo_t
let () = debug ~t:foo_t { bar = "No it's not!" }
```

Syntax Extension

```
type 'a t =
  {
    id : int [@prop {json_field_name= "store_id"}];
    tbl: ((string, 'a value) Hashtbl.t [@patch hashtbl_t]);
  }
and 'a value =
  | Leave of 'a record
  | Store of 'a t
and 'a record = { field: 'a } [@@unboxed]
[@@deriving t]
```

```
val t : 'a ttype -> 'a t ttype = <fun>
val value_t : 'a ttype -> 'a value ttype = <fun>
val record_t : 'a ttype -> 'a record ttype = <fun>
```

Bonus Exercise: Paths

Syntax Extension - Path

```
type t =
 | A of {b: int array list * string}
 | C

let p1 : (t, string) Path.t = [%path? [ A b; (_, []) ]]

let lens = Path.lens p1
let get : t -> string option = lens.get
let set : t -> string -> t option = lens.set
```

Syntax Extension - Path

```
type t =
 | A of {b: int array list * string}
 | C

let p1 : (t, string) Path.t = [%path? [ A b; (_, []) ]]

let lens = Path.lens p1
let get : t -> string option = lens.get
let set : t -> string -> t option = lens.set


let string_t : string ttype = Xtype.project_path t p1
```

Syntax Extension - Path

```
type t =
 | A of {b: int array list * string}
 | C

let p1 : (t, string) Path.t = [%path? [ A b; (_, []) ]]

let lens = Path.lens p1
let get : t -> string option = lens.get
let set : t -> string -> t option = lens.set


let string_t : string ttype = Xtype.project_path t p1


let p2 = [%path? [ A b; ([], _); [0]; [|1|] ]]
```

Syntax Extension - Q&A

Xtype

Xtype

Before

- everything is a record
- modules by kind of type: Record, Constructor, Sum

Xtype

Before

- everything is a record
- modules by kind of type: Record, Constructor, Sum

Now

- disambiguation between tuple / record / inline record
- single mutually recursive type
- modules by use case: Read, Build, Step

Xtype - JSON Demo + Q&A

Benchmark

Benchmark

```
type rec1 =
  { rec1_f1: string
  ; rec1_f2: int
  ; rec1_f3: int * string
  ; rec1_f4: bool
  ; rec1_f5: float list }

and rec2 = {rec2_f1: float; rec2_f2: float; rec2_i1: int}

and variant =
  | R1 of rec1
  | R2 of rec2
  | V1 of bool option array
  | V2 of currency list
  | E1

and t = (variant * variant)
```

Benchmark

- list `l` with 100 000 entries (≈ 65M OCaml syntax)
- `of_json (to_json l)`
- 10 times

My Json

```
[ 24.69G cycles in 10 calls ]  - 82.17% : test
[ 16.35G cycles in 10 calls ]  |   - 66.24% : to_json
[  8.33G cycles in 10 calls ]  |   - 33.76% : of_json
```

Mlfi_json

```
[ 22.23G cycles in 10 calls ]  - 83.15% : test
[ 13.05G cycles in 10 calls ]  |   - 58.74% : to_json
[  9.17G cycles in 10 calls ]  |   - 41.26% : of_json
```

Pattern Matching

Pattern Matching - Pattern

```
type pattern =
 | App of s_id * pattern list
 | Var of v_id
```

- `s_id` uniquely identifies a symbol
- `v_id` uniquely identifies a variable
- invariant: fixed arity per `s_id`

Pattern Matching - Pattern

- symbols arity 0: $a, b, \ldots$
- symbols arity 1: $f, g, \ldots$
- symbols arity 2: $m, n, \ldots$
- variables: $x, y, \ldots$

Pattern Matching - Pattern

(In)Equality

$$m(f(b), g(a)) \neq m(f(b), g(b)) \neq n(f(b), g(b))$$

Pattern Matching - Pattern

(In)Equality

$m(f(b), g(a)) \neq m(f(b), g(b)) \neq n(f(b), g(b))$

Unification

$m(x, g(b)) \circ m(f(b), g(b)) \longrightarrow [x = f(b)]$

Pattern Matching - Pattern

(In)Equality

$m(f(b), g(a)) \neq m(f(b), g(b)) \neq n(f(b), g(b))$

Unification

$m(x, g(b)) \circ m(f(b), g(b)) \longrightarrow [x = f(b)]$

Normalization

$m(y, x) \longrightarrow m(x, y)$

Pattern Matching - Pattern

(In)Equality

$m(f(b), g(a)) \neq m(f(b), g(b)) \neq n(f(b), g(b))$

Unification

$m(x, g(b)) \circ m(f(b), g(b)) \longrightarrow [x = f(b)]$

Normalization

$m(y, x) \longrightarrow m(x, y)$

Precedence

$m(x, y) < m(x, x)$

Pattern Matching - Result

```
match int_of_string_opt x with
| Some i -> i
| None -> 42
```

- fixed type on the right
- values on the right may depend on variables on the left

Pattern Matching - Result

match $m(f(b), g(b))$ with

$$\begin{aligned} m(f(b), g(a)) &\longrightarrow \\ m(x, g(a)) &\longrightarrow \\ m(x, y) &\longrightarrow \\ x &\longrightarrow \end{aligned}$$

Pattern Matching - Result

match $m(f(b), g(b))$ with

$$
\begin{aligned}
m(f(b), g(a)) &\longrightarrow \\
m(x, g(a)) &\longrightarrow \\
m(x, y) &\longrightarrow \\
x &\longrightarrow res_3[x = m(f(b), g(b))]
\end{aligned}
$$

Pattern Matching - Result

match $m(f(b), g(b))$ with

$$\begin{aligned} m(f(b), g(a)) &\longrightarrow \\ m(x, g(a)) &\longrightarrow \\ m(x, y) &\longrightarrow res_2[x = f(b),\ y = g(b)] \\ x &\longrightarrow res_3[x = m(f(b), g(b))] \end{aligned}$$

Pattern Matching - Result

match $m(f(b), g(b))$ with

$$\begin{aligned} m(f(b), g(a)) &\longrightarrow \\ m(x, g(a)) &\longrightarrow \text{no match} \\ m(x, y) &\longrightarrow res_2[x = f(b),\ y = g(b)] \\ x &\longrightarrow res_3[x = m(f(b), g(b))] \end{aligned}$$

Pattern Matching - Result

match $m(f(b), g(b))$ with

$$\begin{aligned} m(f(b), g(a)) &\longrightarrow \text{no match} \\ m(x, g(a)) &\longrightarrow \text{no match} \\ m(x, y) &\longrightarrow res_2[x = f(b),\ y = g(b)] \\ x &\longrightarrow res_3[x = m(f(b), g(b))] \end{aligned}$$

Pattern Matching - Result

match $m(f(b), g(b))$ with

$$
\begin{aligned}
m(f(b), g(a)) &\longrightarrow \text{no match} \\
m(x, g(a)) &\longrightarrow \text{no match} \\
m(x, y) &\longrightarrow res_2[x = f(b),\ y = g(b)] \\
x &\longrightarrow res_3[x = m(f(b), g(b))]
\end{aligned}
$$

- store res_i indexed by pattern
- unify during lookup
- substitute variables in res_i on return

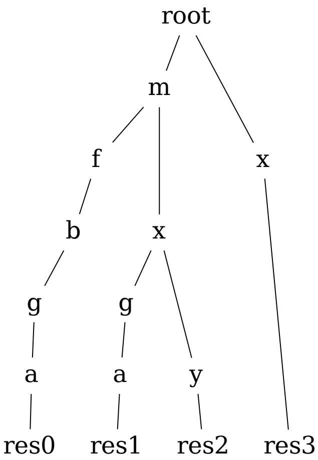
Pattern Matching - Index

Pattern Matching - Index

$$
\begin{aligned}
m(f(b), g(a)) &\longrightarrow mfbga \\
m(x, g(a)) &\longrightarrow mxga \\
m(x, y) &\longrightarrow mxy \\
x &\longrightarrow x
\end{aligned}
$$

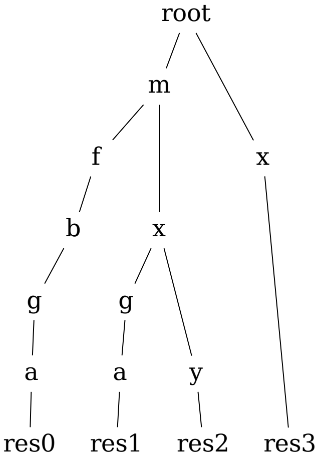
Pattern Matching - Index

$$m(f(b), g(a)) \longrightarrow mfbga$$
$$m(x, g(a)) \longrightarrow mxga$$
$$m(x, y) \longrightarrow mxy$$
$$x \longrightarrow x$$



Pattern Matching - Index

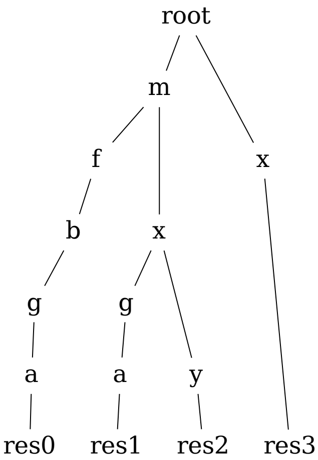
$$
\begin{aligned}
m(f(b), g(a)) &\longrightarrow mfbga \\
m(x, g(a)) &\longrightarrow mxga \\
m(x, y) &\longrightarrow mxy \\
x &\longrightarrow x
\end{aligned}
$$



$m(f(b), g(b))$

Pattern Matching - Index

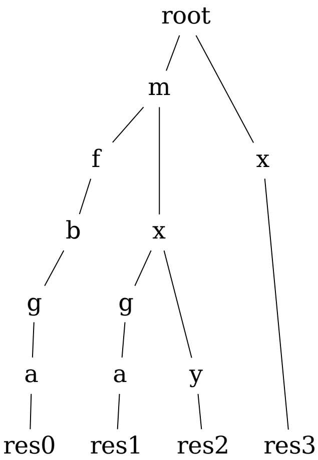
$$m(f(b), g(a)) \longrightarrow mfbga$$
$$m(x, g(a)) \longrightarrow mxga$$
$$m(x, y) \longrightarrow mxy$$
$$x \longrightarrow x$$



$$m(f(b), g(b)) \longrightarrow mfbgb$$

Pattern Matching - Index

$$m(f(b), g(a)) \longrightarrow mfbga$$
$$m(x, g(a)) \longrightarrow mxga$$
$$m(x, y) \longrightarrow mxy$$
$$x \longrightarrow x$$



$$m(f(b), g(b)) \longrightarrow mfbgb \longrightarrow res_2[x = f(b); y = g(b)]$$

Pattern Matching - Implementation

```
type pattern =
 | App of s_id * pattern list
 | Var of v_id
```

How do stypes fit into this structure?

Pattern Matching - Implementation

```
type pattern =
 | App of s_id * pattern list
 | Var of v_id
```

How do stypes fit into this structure?

- base types and tuples: negative integers
- sum, record and abstract: name mapped to positive integer

Towards v0.1

Towards v0.1

- documentation / clean-up

Towards v0.1

- documentation / clean-up
- wording: dynamic types vs. runtime types

Towards v0.1

- documentation / clean-up
- wording: dynamic types vs. runtime types
- name: `dynamic_types` / `runtime_types` / `dynt` / `runt` / ?

Towards v0.1

- documentation / clean-up
- wording: dynamic types vs. runtime types
- name: `dynamic_types` / `runtime_types` / `dynt` / `runt` / ?
- code review, please

Thanks!