Atdgen tutorial

Martin Jambon

May 5, 2011

Contents

1 What is atdgen?

Atdgen is a tool that derives OCaml boilerplate code from type definitions. Currently it provides support for:

- JSON serialization and deserialization.
- Biniou serialization and deserialization. Biniou is a binary format extensible like JSON but more compact and faster to process.
- Convenience functions for creating and validating OCaml data.

2 Prerequisites

This tutorial assumes that you are using atdgen version 1.2.0 or above. The following command tells you which version you are using:

```
$ atdgen -version
1.2.0+dev
```

At the time of writing, atdgen 1.2.0 has not been officially released but the development version is available from Github. You can fetch it using the following command:

```
$ git clone git://github.com/MyLifeLabs/atdgen.git
```

A quick way of installing all the dependencies is via Godi. Run `godi_console` and install atdgen 1.1.1. You can then uninstall it but leave all its dependencies installed.

Now read the instructions in `atdgen/INSTALL` or just do:

```
$ cd atdgen
$ make
$ make install
```

3 Getting started

From now on we assume that atdgen 1.2.0 or above is installed properly. 1.2.0+dev is fine.

```
$ atdgen -version
1.2.0+dev
```

Type definitions are placed in a `.atd` file (`hello.atd`):

```
type date = {
  year : int;
  month : int;
  day : int;
}
```

Our handwritten OCaml program is `hello.ml`:

```
open Hello_t
let () =
  let date = { year = 1970; month = 1; day = 1 } in
  print_endline (Hello_j.string_of_date date)
```

We produce OCaml code from the type definitions using `atdgen`:

```
$ atdgen -t hello.atd     # produces OCaml type definitions
$ atdgen -j hello.atd     # produces OCaml code dealing with JSON
```

We now have `_t` and `_j` files produced by `atdgen -t` and `atdgen -j` respectively:

```
$ ls
hello.atd  hello.ml  hello_j.ml  hello_j.mli  hello_t.ml  hello_t.mli
```

We compile all `.mli` and `.ml` files:

```
$ ocamlfind ocamlc -c hello_t.mli -package atdgen
$ ocamlfind ocamlc -c hello_j.mli -package atdgen
$ ocamlfind ocamlopt -c hello_t.ml -package atdgen
$ ocamlfind ocamlopt -c hello_j.ml -package atdgen
$ ocamlfind ocamlopt -c hello.ml -package atdgen
$ ocamlfind ocamlopt -o hello hello_t.cmx hello_j.cmx hello.cmx \
    -package atdgen -linkpkg
```

And finally we run our `hello` program:

```
$ ./hello
{"year":1970,"month":1,"day":1}
```

4 Inspecting and pretty-printing JSON

Input JSON data:

```
$ cat single.json
[1234,"abcde",{"start_date":{"year":1970,"month":1,"day":1},
"end_date":{"year":1980,"month":1,"day":1}}]
```

Pretty-printed JSON can be produced with the `ydump` command:

```
$ ydump single.json
[
  1234,
  "abcde",
  {
    "start_date": { "year": 1970, "month": 1, "day": 1 },
    "end_date": { "year": 1980, "month": 1, "day": 1 }
  }
]
```

Multiple JSON objects separated by whitespace, typically one JSON object per line, can also be pretty-printed with `ydump`. Input:

```
$ cat stream.json
[1234,"abcde",{"start_date":{"year":1970,"month":1,"day":1},
"end_date":{"year":1980,"month":1,"day":1}}]
[1,"a",{}]
```

In this case the `-s` option is required:

```
$ ydump -s stream.json
[
  1234,
  "abcde",
  {
    "start_date": { "year": 1970, "month": 1, "day": 1 },
    "end_date": { "year": 1980, "month": 1, "day": 1 }
  }
]
[ 1, "a", {} ]
```

From an OCaml program, pretty-printing can be done with `Yojson.Safe.prettify` which has the following signature:

```
val prettify : string -> string
```

We wrote a tiny program that simply calls the `prettify` function on some predefined JSON data (file `prettify.ml`):

```
let json =
"[1234,\"abcde\",{\"start_date\":{\"year\":1970,\"month\":1,\"day\":1},
\"end_date\":{\"year\":1980,\"month\":1,\"day\":1}}]"

let () = print_endline (Yojson.Safe.prettify json)
```

We now compile and run prettify.ml:

```
$ ocamlfind ocamlopt -o prettify prettify.ml -package atdgen -linkpkg
$ ./prettify
[
  1234,
  "abcde",
  {
    "start_date": { "year": 1970, "month": 1, "day": 1 },
    "end_date": { "year": 1980, "month": 1, "day": 1 }
  }
]
```

5 Inspecting biniou data

Biniou is a binary format that can be displayed as text using a generic command called `bdump`. The only practical difficulty is to recover the original field names and variant names which are stored as 31-bit hashes. Unhashing them is done by consulting a dictionary (list of words) maintained by the user.

Let's first produce a sample data file `tree.dat` containing the biniou representation of a binary tree. In the same program we will also demonstrate how to render biniou data into text from an OCaml program.

Here is the ATD file defining our tree type (file `tree.atd`):

```
type tree =
    [ Empty
    | Node of (tree * int * tree) ]
```

This is our OCaml program (file `tree.ml`):

```
open Printf
```

```
(* sample value *)
let tree : Tree_t.tree =
  `Node (
    `Node (`Empty, 1, `Empty),
    2,
    `Node (
      `Node (`Empty, 3, `Empty),
      4,
      `Node (`Empty, 5, `Empty)
    )
  )

let () =
  (* write sample value to file *)
  let fname = "tree.dat" in
  Ag_util.Biniou.to_file Tree_b.write_tree fname tree;

  (* write sample value to string *)
  let s = Tree_b.string_of_tree tree in
  printf "raw value (saved as %s):\n%S\n" fname s;
  printf "length: %i\n" (String.length s);

  printf "pretty-printed value (without dictionary):\n";
  print_endline (Bi_io.view s);

  printf "pretty-printed value (with dictionary):\n";
  let unhash = Bi_io.make_unhash ["Empty"; "Node"; "foo"; "bar" ] in
  print_endline (Bi_io.view ~unhash s)
```

Compilation:

```
$ atdgen -t tree.atd
$ atdgen -b tree.atd
$ ocamlfind ocamlopt -o tree \
    tree_t.mli tree_t.ml tree_b.mli tree_b.ml tree.ml \
    -package atdgen -linkpkg
```

Running the program:

```
$ ./tree
raw value (saved as tree.dat):
"\023\179\2276\"\020\003\023\179\2276\"\020\003\023\003\007\170m\017\002\023\003\007\170m\01
length: 75
pretty-printed value (without dictionary):
```

```
<#33e33622:
   (<#33e33622: (<#0307aa6d>, 1, <#0307aa6d>)>,
    2,
    <#33e33622:
       (<#33e33622: (<#0307aa6d>, 3, <#0307aa6d>)>,
        4,
        <#33e33622: (<#0307aa6d>, 5, <#0307aa6d>)>)>)>
pretty-printed value (with dictionary):
<"Node":
   (<"Node": (<"Empty">, 1, <"Empty">)>,
    2,
    <"Node":
       (<"Node": (<"Empty">, 3, <"Empty">)>,
        4,
        <"Node": (<"Empty">, 5, <"Empty">)>)>)>
```

Now let's see how to pretty-print any biniou data from the command line. Our sample data are now in file `tree.dat`:

```
$ ls -l tree.dat
-rw-r--r-- 1 martin martin 75 Apr 17 01:46 tree.dat
```

We use the command `bdump` to render our sample biniou data as text:

```
$ bdump tree.dat
<#33e33622:
   (<#33e33622: (<#0307aa6d>, 1, <#0307aa6d>)>,
    2,
    <#33e33622:
       (<#33e33622: (<#0307aa6d>, 3, <#0307aa6d>)>,
        4,
        <#33e33622: (<#0307aa6d>, 5, <#0307aa6d>)>)>)>
```

We got hashes for the variant names `Empty` and `Node`. Let's add them to the dictionary:

```
$ bdump -w Empty,Node tree.dat
<"Node":
   (<"Node": (<"Empty">, 1, <"Empty">)>,
    2,
    <"Node":
       (<"Node": (<"Empty">, 3, <"Empty">)>,
        4,
        <"Node": (<"Empty">, 5, <"Empty">)>)>)>
```

`bdump` remembers the dictionary so we don't have to pass the `-w` option anymore (for this user on this machine). The following now works:

```
$ bdump tree.dat
<"Node":
   (<"Node": (<"Empty">, 1, <"Empty">)>,
    2,
    <"Node":
       (<"Node": (<"Empty">, 3, <"Empty">)>,
        4,
        <"Node": (<"Empty">, 5, <"Empty">)>)>)>
```

6 Optional fields and default values

Although OCaml records do not support optional fields, both the JSON and biniou formats make it possible to omit certain fields on a per-record basis.

For example the JSON record `{ "x": 0, "y": 0 }` can be more compactly written as `{}` if the reader knows the default values for the missing fields `x` and `y`. Here is the corresponding type definition:

```
type vector_v1 = { ~x: int; ~y: int }
```

`x` means that field `x` supports a default value. Since we do not specify the default value ourselves, the built-in default is used, which is 0.

If we want the default to be something else than 0, we just have to specify it as follows:

```
type vector_v2 = {
  ~x <ocaml default="1">: int; (* default x is 1 *)
  ~y: int;                     (* default y is 0 *)
}
```

It is also possible to specify optional fields without a default value. For example, let's add an optional `z` field:

```
type vector_v3 = {
  ~x: int;
  ~y: int;
  ?z: int option;
}
```

The following two examples are valid JSON representations of data of type `vector_v3`:

```
{ "x": 2, "y": 2, "z": 3 }  // OCaml: { x = 2; y = 2; z = Some 3 }
```

```
{ "x": 2, "y": 2 }          // OCaml: { x = 2; y = 2; z = None }
```

For a variety of good reasons JSON's `null` value may not be used to indicate that a field is undefined. Therefore the following JSON data cannot be read as a record of type `vector_v3`:

```
{ "x": 2, "y": 2, "z": null }  // invalid value for field z
```

Note also the difference between `?z: int option` and `z: int option`:

```
type vector_v4 = {
  ~x: int;
  ~y: int;
  ~z: int option;  (* no unwrapping of the JSON field value! *)
}
```

Here are valid values of type `vector_v4`, showing that it is usually not what is intended:

```
{ "x": 2, "y": 2, "z": [ "Some", 3 ] }
```

```
{ "x": 2, "y": 2, "z": "None" }
```

```
{ "x": 2, "y": 2 }
```

7 Smooth protocol upgrades

Problem: you have a production system that uses a specific JSON or biniou format. It may be data files or a client-server pair. You now want to add a field to a record type or to add a case to a variant type.

Both JSON and biniou allow extra record fields. If the consumer does not know how to deal with the extra field, the default behavior is to happily ignore it.

7.1 Adding or removing an optional record field

```
type t = {
  x: int;
  y: int;
}
```

Same `.atd` source file, edited:

```
type t = {
  x: int;
  y: int;
  ~z: int; (* new field *)
}
```

- Upgrade producers and consumers in any order
- Converting old data is not required nor useful

7.2 Adding a required record field

```
type t = {
  x: int;
  y: int;
}
```

Same `.atd` source file, edited:

```
type t = {
  x: int;
  y: int;
  z: int; (* new field *)
}
```

- Upgrade all producers before the consumers
- Converting old data requires special-purpose hand-written code

7.3 Removing a required record field

- Upgrade all consumers before the producers
- Converting old data is not required but may save some storage space (just read and re-write each record using the new type)

7.4 Adding a variant case

```
type t = [ A | B ]
```

Same `.atd` source file, edited:

```
type t = [ A | B | C ]
```

- Upgrade all consumers before the producers
- Converting old data is not required and would have no effect

7.5 Removing a variant case

- Upgrade all producers before the consumers
- Converting old data requires special-purpose hand-written code

7.6 Avoiding future problems

- In doubt, use records rather than tuples because it makes it possible to add or remove any field or to reorder them.
- Do not hesitate to create variant types with only one case or records with only one field if you think they might be extended later.

8 Data validation

Atdgen can be used to produce data validators for all types defined in an ATD file, based on user-given validators specified only for certain types. A simple example is:

```
type t = string <ocaml validator="fun s -> String.length s >= 8"> option
```

`atdgen -v` will produce something equivalent to the following implementation:

```
let validate_t x =
  match x with
      None -> true
    | Some x -> (fun s -> String.length s >= 8) x
```

Let's now consider a more realistic example with complex validators defined in a separate .ml file. We created the following 3 source files:

- `resume.atd`: contains the type definitions with annotations
- `resume_util.ml`: contains our handwritten validators
- `resume.ml`: is our main program that creates data and calls the validators

In terms of OCaml modules we have:

- `Resume_t`: produced by `atdgen -t resume.atd`, provides OCaml type definitions
- `Resume_util`: depends on `Resume_t`, provides validators mentioned in `resume.atd`
- `Resume_v`: produced by `atdgen -v resume.atd`, depends on `Resume_util`, provides a validator for each type
- `Resume`: depends on `Resume_v`, uses the validators

Type definitions are placed in `resume.atd`:

```
type text = string <ocaml validator="Resume_util.validate_some_text">

type date = {
  year : int;
  month : int;
  day : int;
} <ocaml validator="Resume_util.validate_date">

type job = {
  company : text;
  title : text;
  start_date : date;
  ?end_date : date option;
} <ocaml validator="Resume_util.validate_job">

type work_experience = job list
```

`resume_util.ml` contains our handwritten validators:

```
open Resume_t

let ascii_printable c =
  let n = Char.code c in
  n >= 32 && n <= 127

(*
  Check that string is not empty and contains only ASCII printable
  characters (for the sake of the example; we use UTF-8 these days)
*)
let validate_some_text s =
  s <> "" &&
```

```
    try
      String.iter (fun c -> if not (ascii_printable c) then raise Exit) s;
      true
    with Exit ->
      false

(*
  Check that the combination of year, month and day exists in the
  Gregorian calendar.
*)
let validate_date x =
  let y = x.year in
  let m = x.month in
  let d = x.day in
  m >= 1 && m <= 12 && d >= 1 &&
  (let dmax =
     match m with
         2 ->
           if y mod 4 = 0 && not (y mod 100 = 0) || y mod 400 = 0 then 29
           else 28
       | 1 | 3 | 5 | 7 | 8 | 10 | 12 -> 31
       | _ -> 30
   in
   d <= dmax)

(* Compare dates chronologically *)
let compare_date a b =
  let c = compare a.year b.year in
  if c <> 0 then c
  else
    let c = compare a.month b.month in
    if c <> 0 then c
    else compare a.day b.day

(* Check that the end_date, when defined, is not earlier than the start_date *)
let validate_job x =
  match x.end_date with
      None -> true
    | Some end_date ->
        compare_date x.start_date end_date <= 0
```

resume.ml uses the validate_work_experience function provided by the Resume_v module:

```
let check_experience x =
  let is_valid = Resume_v.validate_work_experience x in
```

```
  Printf.printf "%s:\n%s\n"
    (if is_valid then "VALID" else "INVALID")
    (Yojson.Safe.prettify (Resume_j.string_of_work_experience x))

let () =
  (* one valid date *)
  let valid = { Resume_t.year = 2000; month = 2; day = 29 } in
  (* one invalid date *)
  let invalid = { Resume_t.year = 1900; month = 0; day = 0 } in
  (* two more valid dates, created with Resume_v.create_date *)
  let date1 = { Resume_t.year = 2005; month = 8; day = 1 } in
  let date2 = { Resume_t.year = 2006; month = 3; day = 22 } in

  let job = {
    Resume_t.company = "Acme Corp.";
    title = "Tester";
    start_date = date1;
    end_date = Some date2;
  }
  in
  let valid_job = { job with Resume_t.start_date = valid } in
  let invalid_job = { job with Resume_t.end_date = Some invalid } in
  let valid_experience = [ job; valid_job ] in
  let invalid_experience = [ job; invalid_job ] in
  check_experience valid_experience;
  check_experience invalid_experience
```

Output:

```
VALID:
[
  {
    "company": "Acme Corp.",
    "title": "Tester",
    "start_date": { "year": 2005, "month": 8, "day": 1 },
    "end_date": { "year": 2006, "month": 3, "day": 22 }
  },
  {
    "company": "Acme Corp.",
    "title": "Tester",
    "start_date": { "year": 2000, "month": 2, "day": 29 },
    "end_date": { "year": 2006, "month": 3, "day": 22 }
  }
]
INVALID:
[
```

```
  {
    "company": "Acme Corp.",
    "title": "Tester",
    "start_date": { "year": 2005, "month": 8, "day": 1 },
    "end_date": { "year": 2006, "month": 3, "day": 22 }
  },
  {
    "company": "Acme Corp.",
    "title": "Tester",
    "start_date": { "year": 2005, "month": 8, "day": 1 },
    "end_date": { "year": 1900, "month": 0, "day": 0 }
  }
]
```

9 Modularity: referring to type definitions from another ATD file

It is possible to define types that depend on types defined in other `.atd` files. The example below is self-explanatory.

`part1.atd`:

```
type t = { x : int; y : int }
```

`part2.atd`:

```
type t1 <ocaml from="Part1" t="t"> = abstract
    (*
      Imports type t defined in file part1.atd.
      The local name is t1. Because the local name (t1) is different from the
      original name (t), we must specify the original name using t=.
    *)

type t2 = t1 list
```

`part3.atd`:

```
type t2 <ocaml from="Part2"> = abstract

type t3 = {
  name : string;
  ?data : t2 option;
}
```

`main.ml`:

```
let v = {
  Part3_t.name = "foo";
  data = Some [
    { Part1_t.x = 1; y = 2 };
    { Part1_t.x = 3; y = 4 };
  ]
}

let () =
  Ag_util.Json.to_channel Part3_j.write_t3 stdout v;
  print_newline ()
```

Output:

```
{"name":"foo","data":[{"x":1,"y":2},{"x":3,"y":4}]}
```

10 Integration with ocamldoc

Ocamldoc is a tool that comes with the core OCaml distribution. It uses comments within `(**` and `*)` to produce hyperlinked documentation (HTML) of module signatures.

Atdgen can produce `.mli` files with comments in the syntax supported by ocamldoc but regular ATD comments within `(*` and `*)` are always discarded by atdgen. Instead, `<doc text="...">` must be used and placed after the element they describe. The contents of the text field must be UTF8-encoded.

```
type point = {
  x : float;
  y : float;
  ~z
    <doc text="Optional depth, its default value is {{0.0}}.">
    : float;
}
  <doc text="Point with optional 3rd dimension.

OCaml example:
{{{
let p =
  { x = 0.5; y = 1.0; z = 0. }
}}}
">
```

is converted into the following `.mli` file with ocamldoc-compatible comments:

```
(**
  Point with optional 3rd dimension.

  OCaml example:

{v
let p =
  \{ x = 0.5; y = 1.0; z = 0. \}
v}
*)
type point = {
  x: float;
  y: float;
  z: float (** Optional depth, its default value is [0.0]. *)
}
```

The only two forms of markup supported by `<doc text="...">` are `{{ ... }}` for inline code and `{{{ ... }}}` for a block of preformatted code.