atd Documentation

Martin Jambon

May 12, 2023

Contents

1 The ATD Project **1**

2 The ATD Language **3**
- 2.1 ATD core syntax reference . . . 3
  - 2.1.1 Introduction . . . 3
  - 2.1.2 ATD language . . . 7
- 2.2 Interoperability with other tools . . . 11
  - 2.2.1 JSON Schema . . . 11

3 OCaml Support - atdgen **13**
- 3.1 Tutorial . . . 13
  - 3.1.1 What is atdgen? . . . 13
  - 3.1.2 What are the advantages of atdgen? . . . 13
  - 3.1.3 Prerequisites . . . 13
  - 3.1.4 Getting started . . . 14
  - 3.1.5 Inspecting and pretty-printing JSON . . . 15
  - 3.1.6 Inspecting biniou data . . . 16
  - 3.1.7 Optional fields and default values . . . 18
  - 3.1.8 Smooth protocol upgrades . . . 19
  - 3.1.9 Data validation . . . 21
  - 3.1.10 Modularity: referring to type definitions from another ATD file . . . 24
  - 3.1.11 Managing JSON configuration files . . . 25
  - 3.1.12 Integration with ocamldoc . . . 29
  - 3.1.13 Integration with build systems . . . 30
  - 3.1.14 Dealing with untypable JSON . . . 32
- 3.2 Atdgen reference . . . 34
  - 3.2.1 Description . . . 34
  - 3.2.2 Command-line usage . . . 34
  - 3.2.3 Default type mapping . . . 41
  - 3.2.4 ATD Annotations . . . 42
  - 3.2.5 Atdgen runtime library . . . 58

4 Java Support - atdj **61**
- 4.1 Installation . . . 61
- 4.2 Quick-start . . . 61
- 4.3 Generating the interface . . . 62
- 4.4 Generating Javadoc documentation . . . 62

4.5 Generating a class graph . . . 63
4.6 Translation reference . . . 63
4.6.1 Bools, ints, floats, string, lists . . . 63
4.6.2 Options . . . 63
4.6.3 Records . . . 64
4.6.4 Sums . . . 64
4.6.5 The Atdj and Visitor interfaces . . . 65

5 Scala Support - atds **67**

6 Python Support - atdpy **69**
6.1 Tutorials . . . 69
6.1.1 Hello World . . . 69
6.1.2 ATD Records, JSON objects, Python classes . . . 71
6.2 How-to guides . . . 72
6.2.1 Defining default field values . . . 72
6.2.2 Renaming field names . . . 72
6.3 Deep dives . . . 72
6.4 Reference . . . 72
6.4.1 Type mapping . . . 72
6.4.2 Supported ATD annotations . . . 72

7 TypeScript Support - atdts **75**
7.1 Tutorials . . . 75
7.1.1 Hello World . . . 75
7.1.2 ATD Records, JSON objects, TypeScript objects . . . 77
7.2 How-to guides . . . 78
7.2.1 Defining default field values . . . 78
7.2.2 Renaming field names . . . 78
7.3 Deep dives . . . 79
7.4 Reference . . . 79
7.4.1 Type mapping . . . 79
7.4.2 Supported ATD annotations . . . 79

CHAPTER 1

The ATD Project

The ATD project aims to facilitate the design and the implementation of APIs, and in particular JSON APIs. It offers type safety and automatic data validation by deriving boilerplate code from type definitions. The data ends up being represented with idiomatic data structures in the target programming language, removing the hassle of manually converting from/to the JSON representation.

Currently, the supported target languages are OCaml, Java, Scala, and Python. The project is run by volunteers and users from various organizations. Check out the ATD project on GitHub for any bug report, feature request, or question.

Some properties of interest of ATD schemas include:

- support for optional fields and default field values
- support for sum types aka algebraic data types or tagged unions
- options to select alternate representations than the default, e.g. use a JSON object rather than an array of pairs

CHAPTER 2

The ATD Language

2.1 ATD core syntax reference

2.1.1 Introduction

ATD stands for Adjustable Type Definitions.

```
(* This is a sample ATD file *)

type profile = {
  id : string;
  email : string;
  ~email_validated : bool;
  name : string;
  ?real_name : string option;
  ~about_me : string list;
  ?gender : gender option;
  ?date_of_birth : date option;
}

type gender = [ Female | Male ]

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

ATD is a language for defining data types across multiple programming languages and multiple data formats. That's it.

We provide an OCaml library that provides a parser and a collection of tools that make it easy to write data validators and code generators based on ATD definitions.

Unlike big frameworks that provide everything in one monolithic package, we split the problem of data exchange into logical modules and ATD is one of them. In particular, we acknowledge that the following pieces have little in common and should be defined and implemented separately:

- data type specifications
- transport protocols
- serialization formats

Ideally we want just one single language for defining data types and it should accomodate all programming languages and data formats. ATD can play this role, but its OCaml implementation makes it particularly easy to translate ATD specifications into other interface definition languages if needed.

It is however much harder to imagine that a single transport protocol and a single serialization format would ever become the only ones used. A reader from the future might wonder why we are even considering defining a transport protocol and a serialization format together. This has been a widespread practice at least until the beginning of the 21st century (ONC RPC, ICE, Thrift, etc.). For mysterious reasons, people somehow became convinced that calls to remote services should be made to mimic internal function calls, pretending that nothing really bad could happen on the way between the caller and the remote service. Well, I don't let my 3-old daughter go to school by herself because the definition of the external world is precisely that it is unsafe.

Data input is by definition unsafe. A program whose internal data is corrupted should abort but a failed attempt to read external data should not cause a program to abort. On the contrary, a program should be very resistent to all forms of data corruption and attacks and provide the best diagnosis possible when problems with external data occur.

Because data exchange is critical and involves multiple partners, we depart from magic programming language-centric or company-centric approaches. We define ATD, a data type definition language designed for maximum expressivity, compatibility across languages and static type checking of programs using such data.

Scope

ATD offers a core syntax for type definitions, i.e. an idealized view of the structure of data. Types are mapped to each programming language or data format using language-specific conventions. Annotations can complete the type definitions in order to specify options for a particular language. Annotations are placed in angle brackets after the element they refer to:

```
type profile = {
  id : int <ocaml repr="int64">;
    (*
      An int here will map to an OCaml int64 instead of
      OCaml's default int type.
      Other languages than OCaml will use their default int type.
    *)

  age : int;
    (* No annotation here, the default int type will be used. *)
}
```

ATD supports:

- the following atomic types: bool, int, float, string and unit;
- built-in list and option types;
- records aka structs with a syntax for optional fields with or without default;
- tuples;
- sum types aka variant types, algebraic data types or tagged unions;

- parametrized types;
- inheritance for both records and sum types;
- abstract types;
- arbitrary annotations.

ATD by design does not support:

- function types, function signatures or method signatures;
- a syntax to represent values;
- a syntax for submodules.

Language overview

ATD was strongly inspired by the type system of ML and OCaml. Such a type system allows static type checking and type inference, properties which contribute to the safety and conciseness of the language.

Unlike mainstream languages like Java, C++, C# or Python to name a few, languages such as Haskell or OCaml offer sum types, also known as algebraic data types or variant types. These allow to specify that an object is of one kind or another without ever performing dynamic casts.

```
(* Example of a sum type in ATD. The vertical bar reads `or'. *)
type shape = [
    Square of float                (* argument: side length *)
  | Rectangle of (float * float)   (* argument: width and height *)
  | Circle of float                (* argument: radius *)
  | Dot                            (* no argument *)
]
```

A notable example of sum types is the predefined option type. An object of an option type contains either one value of a given type or nothing. We could define our own *int_option* type as follows:

```
type int_option = [ None | Some of int ]
```

ATD supports parametrized types also known as generics in Java or templates in C++. We could define our own generic option type as follows:

```
type 'a opt = [ None | Some of 'a ]
  (* 'a denotes a type parameter. *)

type opt_int = int opt
  (* equivalent to int_option defined in the previous example *)

type opt_string = string opt
  (* same with string instead of int *)
```

In practice we shall use the predefined option type. The option type is fundamentally different from nullable objects since the latter don't allow values that would have type *'a option option*.

ATD also support product types. They come in two forms: tuples and records:

```
type tuple_example = (string * int)

type record_example = {
  name : string;
```

(continues on next page)

(continued from previous page)

```
  age : int;
}
```

Although tuples in theory are not more expressive than records, they are much more concise and languages that support them natively usually do not require type definitions.

Finally, ATD supports multiple inheritance which is a simple mechanism for adding fields to records or variants to sum types:

```
type builtin_color = [
    Red | Green | Blue | Yellow
  | Purple | Black | White
]

type rgb = (float * float * float)
type cmyk = (float * float * float * float)

(* Inheritance of variants *)
type color = [
    inherit builtin_color
  | Rgb of rgb
  | Cmyk of cmyk
]
```

```
type basic_profile = {
  id : string;
  name : string;
}

(* Inheritance of record fields *)
type full_profile = {
  inherit basic_profile;
  date_of_birth : (int * int * int) option;
  street_address1 : string option;
  street_address2 : string option;
  city : string option;
  zip_code : string option;
  state : string option;
}
```

Editing and validating ATD files

The extension for ATD files is `.atd`. Editing ATD files is best achieved using an OCaml-friendly editor since the ATD syntax is vastly compatible with OCaml and uses a subset of OCaml's keywords.

Emacs users can use caml-mode or tuareg-mode to edit ATD files. Adding the following line to the `~/.emacs` file will automatically use tuareg-mode when opening a file with a `.atd` extension:

```
(add-to-list 'auto-mode-alist '("\\.atd\\'" . tuareg-mode))
```

The syntax of an ATD file can be checked with the program `atdcat` provided with the OCaml library `atd`. `atdcat` pretty-prints its input data, optionally after some transformations such as monomorphization or inheritance. Here is the output of `atdcat -help`:

```
Usage: _build/install/default/bin/atdcat FILE
  -o <path>
         write to this file instead of stdout
  -x
         make type expressions monomorphic
  -xk
         keep parametrized type definitions and imply -x.
         Default is to return only monomorphic type definitions
  -xd
         debug mode implying -x
  -i
         expand all 'inherit' statements
  -if
         expand 'inherit' statements in records
  -iv
         expand 'inherit' statements in sum types
  -jsonschema <root type name>
         translate the ATD file to JSON Schema.
  -jsonschema-no-additional-properties
         emit a JSON Schema that doesn't tolerate extra fields on JSON
         objects.
  -jsonschema-version { draft-2019-09 | draft-2020-12 }
         specify which version of the JSON Schema standard to target.
         Default: latest supported version, which is currently
         'draft-2020-12  '.
  -ml <name>
         output the ocaml code of the ATD abstract syntax tree
  -html-doc
         replace directly <doc html="..."> by (*html ... *)
         or replace <doc text="..."> by (*html ... *)
         where the contents are formatted as HTML
         using <p>, <code> and <pre>.
         This is suitable input for "caml2html -ext html:cat"
         which converts ATD files into HTML.
  -strip NAME1[,NAME2,...]
         remove all annotations of the form <NAME1 ...>,
         <NAME2 ...>, etc.
  -strip-all
         remove all annotations
  -version
         print the version of atd and exit
  -help  Display this list of options
  --help  Display this list of options
```

2.1.2 ATD language

This is a precise description of the syntax of the ATD language, not a tutorial.

Notations

Lexical and grammatical rules are expressed using a BNF-like syntax. Graphical terminal symbols use *unquoted strings in typewriter font*. Non-graphical characters use their official uppercase ASCII name such as LF for the newline character or SPACE for the space character. Non-terminal symbols use the regular font and link to their definition. Parentheses are used for grouping.

The following postfix operators are used to specify repeats:

x*	0, 1 or more occurrences of x
x?	0 or 1 occurrence of x
x+	1 or more occurrences of x

Lexical rules

ATD does not enforce a particular character encoding other than ASCII compatibility. Non-ASCII text and data found in annotations and in comments may contain arbitrary bytes in the non-ASCII range 128-255 without escaping. The UTF-8 encoding is however strongly recommended for all text. The use of hexadecimal or decimal escape sequences is recommended for binary data.

An ATD lexer splits its input into a stream of tokens, discarding whitespace and comments.

token ::=	keyword	
|	lident	
|	uident	
|	tident	
|	string	
ignorable ::=	space	discarded
|	comment	
space ::=	SPACE | TAB | CR | LF	
blank ::=	SPACE | TAB	
comment ::=	`(*` (comment | string | byte)* `*)`	
lident ::=	(lower | _ identchar) identchar*	lowercase identifier
uident ::=	upper identchar*	uppercase identifier
tident ::=	`'` lident	type parameter
lower ::=	`a`...``z``	
upper ::=	`A`...``Z``	
identchar ::=	upper | lower | digit | _ | `'`	
string ::=	`"` (substring | `'`)* `"`	double-quoted string literal, used in annotations
|	`'` (substring | `"`)* `'`	single-quoted string literal, used in annotations
substring ::=	`\\`	single backslash
|	`\"`	double quote
|	`\'`	single quote
|	`\x` hex hex	single byte in hexadecimal notation
|	\ digit digit digit	single byte in decimal notation
|	`\n`	LF
|	`\r`	CR
|	`\t`	TAB
|	`\b`	BS
|	\ CR? LF blank*	discarded
|	not-backslash	any byte except \ or `"` or `'`
digit ::=	`0`...`9`	
hex ::=	`0`...`9`	
|	`a`...`f`	
|	`A`...`F`	
keyword ::=	`(` | `)` | `[`	all keywords
|	`]` | `{` | `}`	
|	`<` | `>`	

Continued on next page

Table 1 – continued from previous page

|	`;` | `,` | `:` | `*`	
|	`|` | `=` | `?` | `~`	
|	`type` | `of` | `inherit`	

Grammar

module ::=	annot* typedef*	entry point
annot ::=	< lident annot-field* >	annotation
annot-field ::=	(lident (= string)?)	
typedef ::=	`type` params? lident annot = expr	type definition
params ::=	tident	one parameter
|	`(` tident (`,` tident)+ `)`	two or more parameters
expr ::=	expr-body annot*	type expression
|	tident	
expr-body ::=	args? lident	
|	`(` (cell (`*` cell)*)? `)`	tuple type
|	`{` ((field (`;` field)*) `;`?)? `}`	record type
|	`[` (`|`? variant (`|` variant)*)? `]`	sum type
args ::=	expr	one argument
|	`(` expr (`,` expr)+ `)`	two or more arguments
cell ::=	(annot+ `:`)? expr	
field ::=	(`?` | `~`)? lident = expr	
|	`inherit` expr	
variant ::=	uident annot* `of` expr	
|	uident annot*	
|	`inherit` expr	

Predefined type names

The following types are considered predefined and may not be redefined.

Type name	Intended use
`unit`	Type of just one value, useful with parametrized types
`bool`	Boolean
`int`	Integer
`float`	Floating-point number
`string`	Sequence of bytes or characters
`'a option`	Container of zero or one element of type *'a*. See also *'a nullable*.
`'a list`	Collection or sequence of elements of type *'a*
`'a nullable`	Extend type *'a* with an extra conventional value, typically called "null". The operation is idempotent, i.e. *'a nullable* is equivalent to *'a nullable nullable*.
`'a shared`	Values of type *'a* for which sharing must be preserved
`'a wrap`	Values on which a custom, reversible transformation may be applied, as specified by language-specific annotations.
`abstract`	Unspecified type. By default, this is meant to accept any data that is syntactically valid, such as any JSON data that could be parsed successfully. With the help of ATD annotations, this can be used to express types not supported by the ATD language such as "either a boolean or a string".

Shared values (deprecated)

ATD supports a special type `x shared` where `x` can be any monomorphic type expression. It allows notably to represent cyclic values and to enforce that cycles are preserved during transformations such as serialization.

```
(* Example of a simple graph type *)
type shared_node = node shared (* sharing point *)
type graph = shared_node list
type node = {
  label : string;
  neighbors : shared_node list;
}
```

Two shared values that are physically identical must remain physically identical after any translation from one data format to another.

Each occurrence of a `shared` type expression in the ATD source definition defines its own sharing point. Therefore the following attempt at defining a graph type will not preserve cycles because two sharing points are defined:

```
(* Incorrect definition of a graph type *)
type node = {
  label : string;
  neighbors : node shared (* sharing point 1 *) list;
}

(* Second occurrence of "shared", won't preserve cycles! *)
type graph = node shared (* sharing point 2 *) list
```

There is actually a way of having multiple `shared` type expressions using the same sharing point but this feature is designed for code generators and should not be used in handwritten ATD definitions. The technique consists in providing an annotation of the form `<share id=x>` where `x` is any string identifying the sharing point. The graph example can be rewritten correctly as:

```
type node = {
  label : string;
  neighbors : node shared <share id="1"> list;
}

type graph = node shared <share id="1"> list
```

2.2 Interoperability with other tools

2.2.1 JSON Schema

ATD type definitions can be translated to JSON Schema with `atdcat`. The user must specify the main type on the command line since ATD doesn't have a notion of main type or root type. This can be useful for target languages that are not yet supported by ATD or for educational purposes.

Example

Input: ATD file `message.atd`:

```
type msg = {
  subject: string;
  ?body: string option;
  ~attachments: attachment list;
}

type attachment = [
  | Image of string
  | Virus
]
```

Conversion to JSON Schema:

```
$ atdcat -jsonschema msg message.atd -o message.schema.json
```

Output: JSON Schema file `message.schema.json`:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "description": "Translated by atdcat from 'message.atd'",
  "type": "object",
  "required": [ "subject" ],
  "properties": {
    "subject": { "type": "string" },
    "body": { "type": "string" },
    "attachments": {
      "type": "array",
      "items": { "$ref": "#/definitions/attachment" }
    }
  },
  "definitions": {
    "attachment": {
      "oneOf": [
        {
```

(continues on next page)

(continued from previous page)

```
          "type": "array",
          "minItems": 2,
          "items": false,
          "prefixItems": [ { "const": "Image" }, { "type": "string" } ]
        },
        { "const": "Virus" }
      ]
    }
  }
}
```

The `jsonschema` tool (Python implementation) can validate JSON data using the JSON Schema file that we generated. For example, passing an empty object `{}` correctly results in an error telling us the `subject` field is missing:

```
$ jsonschema message.json -i <(echo '{}')
{}: 'subject' is a required property
```

With valid JSON input such as `{"subject": "hello", "attachments": ["Virus"]}`, the command exits successfully and silently:

```
$ jsonschema message.json -i <(echo '{"subject": "hello", "attachments": ["Virus"]}')
```

CHAPTER 3

OCaml Support - atdgen

3.1 Tutorial

3.1.1 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.

3.1.2 What are the advantages of atdgen?

Atdgen has a number of advantages over its predecessor json-static which was based on Camlp4:

- produces explicit interfaces which describe what is available to the user (*.mli* files).
- produces readable OCaml code that can be easily reviewed (*.ml* files).
- produces fast code, 3x faster than json-static.
- runs fast, keeping build times low.
- same ATD definitions can be used to generate code other than OCaml. See for instance atdj which generates Java classes for JSON IO. Auto-generating GUI widgets from type definitions is another popular use of annotated type definitions. The implementation of such code generators is facilitated by the atd library.

3.1.3 Prerequisites

This tutorial assumes that you are using atdgen version 1.5.0 or above. The following command tells you which version you are using:

```
$ atdgen -version
1.5.0
```

The recommended way of installing atdgen and all its dependencies is with opam:

```
$ opam install atdgen
```

3.1.4 Getting started

From now on we assume that atdgen 1.5.0 or above is installed properly.

```
$ atdgen -version
1.5.0
```

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}
```

Source code for this section

3.1.5 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
```

(continues on next page)

(continued from previous page)

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

Source code for this section

3.1.6 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
  Atdgen_runtime.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);
```

(continues on next page)

(continued from previous page)

```
  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\017\004\023\179\2276\
↪"\020\003\023\179\2276\
↪"\020\003\023\003\007\170m\017\006\023\003\007\170m\017\b\023\179\2276\
↪"\020\003\023\003\007\170m\017\n\023\003\007\170m"
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">)>)>)>)>
```

Source code for this section

3.1.7 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 }
```

By default, JSON fields whose value is `null` are treated as missing fields. The following two JSON objects are therefore equivalent:

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

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 }
```

3.1.8 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.

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

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

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)

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

Removing a variant case

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

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.

3.1.9 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 valid="fun s -> String.length s >= 8"> option
```

As we can see from this example, the validation function is specified using the annotation `<ocaml valid="p">`, where `p` is a predicate `p : t -> bool`, returning `true` when the value of type `t` is valid and `false` otherwise.

Calling `atdgen -v` on a file containing this specification will produce a validation function equivalent to the following implementation:

```
let validate_t path x =
  match x with
  | None -> None
  | Some x ->
      let msg = "Failed check by fun s -> String.length s >= 8" in
      if (fun s -> String.length s >= 8) x
      then None
      else Some {error_path = path; error_msg = msg}
```

Let's consider this particular example as an illustration of the general shape of generated validation functions.

The function takes two arguments: the first, `path`, is a list indicating where the second, `x`, was encountered. As specified by our example `.atd` code above, `x` has type `t option`.

The body of the validation function does two things:

1. it checks the value of `x` against the validation function specified in our `.atd` file, namely, checking whether there is `Some s`, and verifying that `s` is at least 8 characters long if so 2. in the event that the validation check fails, it constructs an appropriate error record.

In general, generated validation functions for a type `t` have a type equivalent to `validate_t : path -> t -> error option`, where the `path` gives the current location in a data structure and the `error` is a record of the location of, and reason for, validation failure.

A return value of `None` indicates successful validation, while `Some {error_path; error_msg}` tells us where and why validation failed.

Let's now consider a more realistic example with complex validators defined in a separate `.ml` file. We will define a data structure representing a section of a resume recording work experience. We will also define validation functions that can enforce certain properties to protect against errors and junk data.

In the course of this example, we will manually create 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 checks it using our generated validation functions.

After generating additional code with `atdgen`, we will end up with the following OCaml modules:

- `Resume_t`: generated into `resume_t.ml` by `atdgen -t resume.atd`, this provides our OCaml type definitions
- `Resume_util`: written manually in `resume_util.ml`, this depends on `Resume_t` and provides validators we will use in `resume.atd`
- `Resume_v`: generated into `resume_v.ml` by `atdgen -v resume.atd`, this depends on `Resume_util` and `Resume_t` and provides a validation function for each type

- `Resume_j`: generated into `resume_j.ml` by `atdgen -j resume.atd`, this provides functions to serialize and deserialize data in and out of JSON.
- `Resume`: written manually in `resume.ml`, this depends on `Resume_v`, and `Resume_t`, and makes use of the generated types and validation functions.

To begin, we specify type definitions for a data structure representing a resume in `resume.atd`:

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

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

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

type work_experience = job list
```

We can now call `atdgen -t resume.atd` to generate our `Resume_t` module in `resume_t.ml`, providing our data types. Using these data types, we'll define the following handwritten validators in `resume_util.ml` (note that we've already referred to these validators in `resume.atd`):

```
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
```

(continues on next page)

(continued from previous page)

```
        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
```

After we call `atdgen -v resume.atd`, the module `Resume_v` will be generated in `resume_v.ml`, providing the function `validate_work_experience`. We can then use this function, along with the generated `Resume_j` in the following program written in `resume.ml`:

```
let check_experience x =
  let is_valid = match Resume_v.validate_work_experience [] x with
    | None -> false
    | _ -> true
  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
```

(continues on next page)

(continued from previous page)

```
  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 }
  }
```

Source code for this section

3.1.10 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 () =
  Atdgen_runtime.Util.Json.to_channel Part3_j.write_t3 stdout v;
  print_newline ()
```

Output:

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

Source code for this section

3.1.11 Managing JSON configuration files

JSON makes a good format for configuration files because it is human-readable, easy to modify programmatically and widespread. Here is an example of how to use atdgen to manage config files.

- **Specifying defaults** is done in the .atd file. See section [Optional fields and default values] for details on how to do that.
- **Auto-generating a template config file with default values**: a sample value in the OCaml world needs to be created but only fields without default need to be specified.
- **Describing the format** is achieved by embedding the .atd type definitions in the OCaml program and printing it out on request.
- **Loading a config file and reporting illegal fields** is achieved using the JSON deserializers produced by `atdgen -j`. Option `-j-strict-fields` ensures the misspelled field names are not ignored but reported as errors.
- **Reindenting a config file** is achieved by the pretty-printing function `Yojson.Safe.prettify` that takes a JSON string and returns an equivalent JSON string.
- **Showing implicit (default) settings** is achieved by passing the `-j-defaults` option to `atdgen`. The OCaml config data is then serialized into JSON containing all fields, including those whose value is the default.

The example uses the following type definitions:

```
type config = {
  title : string;
  ?description : string option;
```

(continues on next page)

(continued from previous page)

```
  ~timeout <ocaml default="10"> : int;
  ~credentials : param list
    <ocaml valid="fun l ->
                  l <> [] || failwith \"missing credentials\"">;
}

type param = {
  name : string
    <ocaml valid="fun s -> s <> \"\"">;
  key : string
    <ocaml valid="fun s -> String.length s = 16">;
}
```

Our program will perform the following actions:

```
$ ./config -template
{
  "title": "",
  "timeout": 10,
  "credentials": [ { "name": "foo", "key": "0123456789abcdef" } ]
}

$ ./config -format
type config = {
  title : string;
  ?description : string option;
  ~timeout <ocaml default="10"> : int;
  ~credentials : param list
    <ocaml valid="fun l ->
                  l <> [] || failwith \"missing credentials\"">;
}

type param = {
  name : string
    <ocaml valid="fun s -> s <> \"\"">;
  key : string
    <ocaml valid="fun s -> String.length s = 16">;
}

$ cat sample-config.json
{
  "title": "Example",
  "credentials": [
    {
      "name": "joeuser",
      "key": "db7c0877bdef3016"
    },
    {
      "name": "tester",
      "key": "09871ff387ac2b10"
    }
  ]
}

$ ./config -validate sample-config.json
{
  "title": "Example",
```

(continues on next page)

(continued from previous page)

```
  "timeout": 10,
  "credentials": [
    { "name": "joeuser", "key": "db7c0877bdef3016" },
    { "name": "tester", "key": "09871ff387ac2b10" }
  ]
}
```

This is our *demo.sh* script that builds and runs our example program called *config*:

```
#! /bin/sh -e

set -x

# Embed the contents of the .atd file into our OCaml program
echo 'let contents = "\' > config_atd.ml
sed -e 's/\([\\"]\)/\\\1/g' config.atd >> config_atd.ml
echo '"' >> config_atd.ml

# Derive OCaml type definitions from .atd file
atdgen -t config.atd

# Derive JSON-related functions from .atd file
atdgen -j -j-defaults -j-strict-fields config.atd

# Derive validator from .atd file
atdgen -v config.atd

# Compile the OCaml program
ocamlfind ocamlopt -o config \
  config_t.mli config_t.ml config_j.mli config_j.ml config_v.mli config_v.ml \
  config_atd.ml config.ml -package atdgen -linkpkg

# Output a sample config
./config -template

# Print the original type definitions
./config -format

# Fail to validate an invalid config file
./config -validate bad-config1.json || :

# Fail to validate another invalid config file (using custom validators)
./config -validate bad-config3.json || :

# Validate, inject missing defaults and pretty-print
./config -validate sample-config.json

This is the hand-written OCaml program. It can be used as a start
point for a real-world program using a JSON config file:
```

```
open Printf

let param_template =
  (* Sample item used to populate the template config file *)
  {
    Config_v.name = "foo";
```

(continues on next page)

(continued from previous page)

```
    key = "0123456789abcdef"
  }

let config_template =
  (*
    Records can be conveniently created using functions generated by
    "atdgen -v".
    Here we use Config_v.create_config to create a record of type
    Config_t.config. The big advantage over creating the record
    directly using the record notation {...} is that we don't have to
    specify default values (such as timeout in this example).
  *)
  Config_v.create_config ~title:"" ~credentials: [param_template] ()

let make_json_template () =
  (* Thanks to the -j-defaults flag passed to atdgen, even default
    fields will be printed out *)
  let compact_json = Config_j.string_of_config config_template in
  Yojson.Safe.prettify compact_json

let print_template () =
  print_endline (make_json_template ())

let print_format () =
  print_string Config_atd.contents

let validate fname =
  let x =
    try
      (* Read config data structure from JSON file *)
      let x = Atdgen_runtime.Util.Json.from_file Config_j.read_config fname in
      (* Call the validators specified by <ocaml valid=...> *)
      if not (Config_v.validate_config x) then
        failwith "Some fields are invalid"
      else
        x
    with e ->
      (* Print decent error message and exit *)
      let msg =
        match e with
            Failure s
          | Yojson.Json_error s -> s
          | e -> Printexc.to_string e
      in
      eprintf "Error: %s\n%!" msg;
      exit 1
  in
  (* Convert config to compact JSON and pretty-print it.
    ~std:true means that the output will not use extended syntax for
    variants and tuples but only standard JSON. *)
  let json = Yojson.Safe.prettify ~std:true (Config_j.string_of_config x) in
  print_endline json

type action = Template | Format | Validate of string

let main () =
  let action = ref Template in
```

(continues on next page)

(continued from previous page)

```
  let options = [
    "-template", Arg.Unit (fun () -> action := Template),
    "
          prints a sample configuration file";

    "-format", Arg.Unit (fun () -> action := Format),
    "
          prints the format specification of the config files (atd format)";

    "-validate", Arg.String (fun s -> action := Validate s),
    "<CONFIG FILE>
          reads a config file, validates it, adds default values
          and prints the config nicely to stdout";
  ]
  in
  let usage_msg = sprintf "\
Usage: %s [-template|-format|-validate ...]
Demonstration of how to manage JSON configuration files with atdgen.
"
    Sys.argv.(0)
  in
  let anon_fun s = eprintf "Invalid command parameter %S\n%!" s; exit 1 in
  Arg.parse options anon_fun usage_msg;

  match !action with
      Template -> print_template ()
    | Format -> print_format ()
    | Validate s -> validate s

let () = main ()
```

The full source code for this section with examples can be inspected and downloaded here.

3.1.12 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. }
```

(continues on next page)

(continued from previous page)

```
}}}
">
```

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.

3.1.13 Integration with build systems

OMake

We provide an Atdgen plugin for OMake. It simplifies the compilation rules to a minimum.

The plugin consists of a self-documented file to copy into a project's root. The following is a sample `OMakefile` for a project using JSON and five source files (`foo.atd`, `foo.ml`, `bar.atd`, `bar.ml` and `main.ml`):

```
# require file Atdgen.om
include Atdgen

# OCaml modules we want to build
OCAMLFILES = foo_t foo_j foo bar_t bar_j bar main

Atdgen(foo bar, -j-std)
OCamlProgram(foobar, $(OCAMLFILES))

.DEFAULT: foobar.opt

.PHONY: clean
clean:
  rm -f *.cm[ioxa] *.cmx[as] *.[oa] *.opt *.run *~
  rm -f $(ATDGEN_OUTFILES)
```

Running `omake` builds the native code executable `foobar.opt`.

`omake clean` removes all the products of compilation including the `.mli` and `.ml` produced by `atdgen`.

GNU Make

We provide Atdgen.mk, a generic makefile that defines the dependencies and rules for generating OCaml `.mli` and `.ml` files from `.atd` files containing type definitions. The `Atdgen.mk` file contains its own documentation.

Here is a sample *Makefile* that takes advantage of OCamlMakefile:

```
.PHONY: default
default: opt

ATDGEN_SOURCES = foo.atd bar.atd
ATDGEN_FLAGS = -j-std
include Atdgen.mk

SOURCES = \
  foo_t.mli foo_t.ml foo_j.mli foo_j.ml \
  bar_t.mli bar_t.ml bar_j.mli bar_j.ml \
  hello.ml
RESULT = hello
PACKS = atdgen
# "include OCamlMakefile" must come after defs for SOURCES, RESULT, PACKS, etc.
include OCamlMakefile

.PHONY: sources opt all
sources: $(SOURCES)
opt: sources
        $(MAKE) native-code
all: sources
        $(MAKE) byte-code
```

`make` alone builds a native code executable from source files `foo.atd`, `bar.atd` and `hello.ml`. `make clean` removes generated files. `make all` builds a bytecode executable.

In addition to `native-code`, `byte-code` and `clean`, `OCamlMakefile` provides a number of other targets and options which are documented in `OCamlMakefile`'s README.

Ocamlbuild

There is an atdgen plugin for ocamlbuild.

Dune (formerly jbuilder)

Dune currently needs atdgen build rules specified manually. Given an `example.atd`, this will usually look like:

```
(rule
 (targets example_j.ml
          example_j.mli)
 (deps    example.atd)
 (action  (run atdgen -j -j-std %{deps})))

(rule
 (targets example_t.ml
          example_t.mli)
 (deps    example.atd)
 (action  (run atdgen -t %{deps})))
```

You can refer to `example_t.ml` and `example_j.ml` as usual (by default, they will be automatically linked into the library being built in the same directory). You will need to write rules for each .atd file individually until Dune supports wildcard rules.

Note that any options `atdgen` supports can be included in the `run atdgen` section (`-open`, `-deriving-conv`, etc.).

3.1.14 Dealing with untypable JSON

Sometimes we have to deal with JSON data that cannot be described using type definitions. In such case, we can represent the data as its JSON abstract syntax tree (AST), which lets the user inspect it at runtime.

Let's consider a list of JSON objects for which we don't know the type definitions, but somehow some other system knows how to deal with such data. Here is such data:

```
[
  {
    "label": "flower",
    "value": {
      "petals": [12, 45, 83.5555],
      "water": "a340bcf02e"
    }
  },
  {
    "label": "flower",
    "value": {
      "petals": "undefined",
      "fold": null,
      "water": 0
    }
  },
  { "labels": ["fork", "scissors"],
    "value": [ 8, 8 ]
  }
]
```

Hopefully this means something for someone. We are going to assume that each object has a `value` field of an unknown type, and may have a field `label` or a field `labels` of type `string`:

```
(* File untypable.atd *)

type obj_list = obj list

type obj = {
  ?label: string option;
  ?labels: string list option;
  value: abstract  (* requires ATD >= 2.6.0 *)
}
```

Until ATD 2.5, `abstract` could not be used as freely and would not stand for raw JSON by default. One had to write a dedicated type definition as shown below:

```
(* File untypable.atd *)

(* deprecated since ATD 2.6 *)
type json <ocaml module="Yojson.Safe"> = abstract
  (* uses type Yojson.Safe.t,
```

(continues on next page)

(continued from previous page)

```
     with the functions Yojson.Safe.write_json
     and Yojson.Safe.read_json *)

type obj_list = obj list

type obj = {
  ?label: string option;
  ?labels: string list option;
  value: json
}
```

It is possible to give a different name than `json` to the type of the JSON AST, but then the name of the type used in the original module must be provided in the annotation, i.e.:

```
(* deprecated since ATD 2.6 *)
type raw_json <ocaml module="Yojson.Safe" t="json"> = abstract
  (* uses type Yojson.Safe.t,
     with the functions Yojson.Safe.write_json
     and Yojson.Safe.read_json *)

type obj_list = obj list

type obj = {
  ?label: string option;
  ?labels: string list option;
  value: raw_json
}
```

Compile either example with:

```
$ atdgen -t untypable.atd
$ atdgen -j -j-std untypable.atd
$ ocamlfind ocamlc -a -o untypable.cma -package atdgen \
    untypable_t.mli untypable_t.ml untypable_j.mli untypable_j.ml
```

Test the example with your favorite OCaml toplevel (`ocaml` or `utop`):

```
# #use "topfind";;
# #require "atdgen";;
# #load "untypable.cma";;
# Atdgen_runtime.Util.Json.from_channel Untypable_j.read_obj_list stdin;;
[
  {
    "label": "flower",
    "value": {
      "petals": [12, 45, 83.5555],
      "water": "a340bcf02e"
    }
  },
  {
    "label": "flower",
    "value": {
      "petals": "undefined",
      "fold": null,
      "water": 0
    }
```

(continues on next page)

(continued from previous page)

```
  },
  { "labels": ["fork", "scissors"],
    "value": [ 8, 8 ]
  }
]
- : Untypable_t.obj_list =
[{Untypable_t.label = Some "flower"; labels = None;
  value =
  `Assoc
    [("petals", `List [`Int 12; `Int 45; `Float 83.5555]);
      ("water", `String "a340bcf02e")]};
{Untypable_t.label = Some "flower"; labels = None;
  value =
  `Assoc [("petals", `String "undefined");
          ("fold", `Null);
          ("water", `Int 0)]};
{Untypable_t.label = None; labels = Some ["fork"; "scissors"];
  value = `List [`Int 8; `Int 8]}]
```

3.2 Atdgen reference

3.2.1 Description

Atdgen is a command-line program that takes as input type definitions in the ATD syntax and produces OCaml code suitable for data serialization and deserialization.

Two data formats are currently supported, these are JSON and biniou, a binary format with extensibility properties similar to JSON. Atdgen-json and Atdgen-biniou will refer to Atdgen used in one context or the other.

Atdgen was designed with efficiency and durability in mind. Software authors are encouraged to use Atdgen directly and to write tools that may reuse part of Atdgen's source code.

Atdgen uses the following packages that were developed in conjunction with Atdgen:

- `atd`: parser for the syntax of type definitions
- `biniou`: parser and printer for biniou, a binary extensible data format
- `` `yojson <https://github.com/ocaml-community/yojson>`__ ``: parser and printer for JSON, a widespread text-based data format

3.2.2 Command-line usage

Command-line help

Call `atdgen -help` for the full list of available options.

Atdgen-json example

```
$ atdgen -t example.atd
$ atdgen -j -j-std example.atd
```

Input file `example.atd`:

```
type profile = {
  id : string;
  email : string;
  ~email_validated : bool;
  name : string;
  ?real_name : string option;
  ~about_me : string list;
  ?gender : gender option;
  ?date_of_birth : date option;
}

type gender = [ Female | Male ]

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

is used to produce files `example_t.mli`, `example_t.ml`, `example_j.mli` and `example_j.ml`. This is `example_j.mli`:

```
(* Auto-generated from "example.atd" *)


type gender = Example_t.gender

type date = Example_t.date = { year: int; month: int; day: int }

type profile = Example_t.profile = {
  id: string;
  email: string;
  email_validated: bool;
  name: string;
  real_name: string option;
  about_me: string list;
  gender: gender option;
  date_of_birth: date option
}

val write_gender :
  Bi_outbuf.t -> gender -> unit
  (** Output a JSON value of type {!gender}. *)

val string_of_gender :
  ?len:int -> gender -> string
  (** Serialize a value of type {!gender}
      into a JSON string.
      @param len specifies the initial length
                 of the buffer used internally.
                 Default: 1024. *)

val read_gender :
  Yojson.Safe.lexer_state -> Lexing.lexbuf -> gender
  (** Input JSON data of type {!gender}. *)

val gender_of_string :
```

(continues on next page)

(continued from previous page)

```
  string -> gender
  (** Deserialize JSON data of type {!gender}. *)

val write_date :
  Bi_outbuf.t -> date -> unit
  (** Output a JSON value of type {!date}. *)

val string_of_date :
  ?len:int -> date -> string
  (** Serialize a value of type {!date}
      into a JSON string.
      @param len specifies the initial length
                 of the buffer used internally.
                 Default: 1024. *)

val read_date :
  Yojson.Safe.lexer_state -> Lexing.lexbuf -> date
  (** Input JSON data of type {!date}. *)

val date_of_string :
  string -> date
  (** Deserialize JSON data of type {!date}. *)

val write_profile :
  Bi_outbuf.t -> profile -> unit
  (** Output a JSON value of type {!profile}. *)

val string_of_profile :
  ?len:int -> profile -> string
  (** Serialize a value of type {!profile}
      into a JSON string.
      @param len specifies the initial length
                 of the buffer used internally.
                 Default: 1024. *)

val read_profile :
  Yojson.Safe.lexer_state -> Lexing.lexbuf -> profile
  (** Input JSON data of type {!profile}. *)

val profile_of_string :
  string -> profile
  (** Deserialize JSON data of type {!profile}. *)
```

Module `Example_t` (files `example_t.mli` and `example_t.ml`) contains all OCaml type definitions that can be used independently from Biniou or JSON.

For convenience, these definitions are also made available from the `Example_j` module whose interface is shown above. Any type name, record field name or variant constructor can be referred to using either module. For example, the OCaml expressions `((x : Example_t.date) : Example_j.date)` and `x.Example_t.year = x.Example_j.year` are both valid.

Atdgen-biniou example

```
$ atdgen -t example.atd
$ atdgen -b example.atd
```

Input file `example.atd`:

```
type profile = {
  id : string;
  email : string;
  ~email_validated : bool;
  name : string;
  ?real_name : string option;
  ~about_me : string list;
  ?gender : gender option;
  ?date_of_birth : date option;
}

type gender = [ Female | Male ]

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

is used to produce files `example_t.mli`, `example_t.ml`, `example_b.mli` and `example_b.ml`.

This is `example_b.mli`:

```
(* Auto-generated from "example.atd" *)


type gender = Example_t.gender

type date = Example_t.date = { year: int; month: int; day: int }

type profile = Example_t.profile = {
  id: string;
  email: string;
  email_validated: bool;
  name: string;
  real_name: string option;
  about_me: string list;
  gender: gender option;
  date_of_birth: date option
}

(* Writers for type gender *)

val gender_tag : Bi_io.node_tag
  (** Tag used by the writers for type {!gender}.
      Readers may support more than just this tag. *)

val write_untagged_gender :
  Bi_outbuf.t -> gender -> unit
  (** Output an untagged biniou value of type {!gender}. *)

val write_gender :
  Bi_outbuf.t -> gender -> unit
  (** Output a biniou value of type {!gender}. *)

val string_of_gender :
```

(continues on next page)

(continued from previous page)

```
  ?len:int -> gender -> string
  (** Serialize a value of type {!gender} into
      a biniou string. *)

(* Readers for type gender *)

val get_gender_reader :
  Bi_io.node_tag -> (Bi_inbuf.t -> gender)
  (** Return a function that reads an untagged
      biniou value of type {!gender}. *)

val read_gender :
  Bi_inbuf.t -> gender
  (** Input a tagged biniou value of type {!gender}. *)

val gender_of_string :
  ?pos:int -> string -> gender
  (** Deserialize a biniou value of type {!gender}.
      @param pos specifies the position where
                reading starts. Default: 0. *)

(* Writers for type date *)

val date_tag : Bi_io.node_tag
  (** Tag used by the writers for type {!date}.
      Readers may support more than just this tag. *)

val write_untagged_date :
  Bi_outbuf.t -> date -> unit
  (** Output an untagged biniou value of type {!date}. *)

val write_date :
  Bi_outbuf.t -> date -> unit
  (** Output a biniou value of type {!date}. *)

val string_of_date :
  ?len:int -> date -> string
  (** Serialize a value of type {!date} into
      a biniou string. *)

(* Readers for type date *)

val get_date_reader :
  Bi_io.node_tag -> (Bi_inbuf.t -> date)
  (** Return a function that reads an untagged
      biniou value of type {!date}. *)

val read_date :
  Bi_inbuf.t -> date
  (** Input a tagged biniou value of type {!date}. *)

val date_of_string :
  ?pos:int -> string -> date
  (** Deserialize a biniou value of type {!date}.
      @param pos specifies the position where
                reading starts. Default: 0. *)
```

(continues on next page)

(continued from previous page)

```
(* Writers for type profile *)

val profile_tag : Bi_io.node_tag
  (** Tag used by the writers for type {!profile}.
      Readers may support more than just this tag. *)

val write_untagged_profile :
  Bi_outbuf.t -> profile -> unit
  (** Output an untagged biniou value of type {!profile}. *)

val write_profile :
  Bi_outbuf.t -> profile -> unit
  (** Output a biniou value of type {!profile}. *)

val string_of_profile :
  ?len:int -> profile -> string
  (** Serialize a value of type {!profile} into
      a biniou string. *)

(* Readers for type profile *)

val get_profile_reader :
  Bi_io.node_tag -> (Bi_inbuf.t -> profile)
  (** Return a function that reads an untagged
      biniou value of type {!profile}. *)

val read_profile :
  Bi_inbuf.t -> profile
  (** Input a tagged biniou value of type {!profile}. *)

val profile_of_string :
  ?pos:int -> string -> profile
  (** Deserialize a biniou value of type {!profile}.
      @param pos specifies the position where
                 reading starts. Default: 0. *)
```

Module `Example_t` (files `example_t.mli` and `example_t.ml`) contains all OCaml type definitions that can be used independently from Biniou or JSON.

For convenience, these definitions are also made available from the `Example_b` module whose interface is shown above. Any type name, record field name or variant constructor can be referred to using either module. For example, the OCaml expressions `((x : Example_t.date) : Example_b.date)` and `x.Example_t.year = x.Example_b.year` are both valid.

Validator example

```
$ atdgen -t example.atd
$ atdgen -v example.atd
```

Input file `example.atd`:

```
type month = int <ocaml valid="fun x -> x >= 1 && x <= 12">
type day = int <ocaml valid="fun x -> x >= 1 && x <= 31">

type date = {
```

(continues on next page)

(continued from previous page)

```
  year : int;
  month : month;
  day : day;
}
  <ocaml validator="Date_util.validate_date">
```

is used to produce files `example_t.mli`, `example_t.ml`, `example_v.mli` and `example_v.ml`. This is `example_v.ml`, showing how the user-specified validators are used:

```
(* Auto-generated from "example.atd" *)


type gender = Example_t.gender

type date = Example_t.date = { year: int; month: int; day: int }

type profile = Example_t.profile = {
  id: string;
  email: string;
  email_validated: bool;
  name: string;
  real_name: string option;
  about_me: string list;
  gender: gender option;
  date_of_birth: date option
}

val validate_gender :
  Atdgen_runtime.Util.Validation.path -> gender -> Atdgen_runtime.Util.Validation.
↪error option
  (** Validate a value of type {!gender}. *)

val create_date :
  year: int ->
  month: int ->
  day: int ->
  unit -> date
  (** Create a record of type {!date}. *)

val validate_date :
  Atdgen_runtime.Util.Validation.path -> date -> Atdgen_runtime.Util.Validation.error␣
↪option
  (** Validate a value of type {!date}. *)

val create_profile :
  id: string ->
  email: string ->
  ?email_validated: bool ->
  name: string ->
  ?real_name: string ->
  ?about_me: string list ->
  ?gender: gender ->
  ?date_of_birth: date ->
  unit -> profile
  (** Create a record of type {!profile}. *)
```

(continues on next page)

(continued from previous page)

```
val validate_profile :
  Atdgen_runtime.Util.Validation.path -> profile -> Atdgen_runtime.Util.Validation.
↪error option
  (** Validate a value of type {!profile}. *)
```

3.2.3 Default type mapping

The following table summarizes the default mapping between ATD types and OCaml, biniou and JSON data types. For each language more representations are available and are detailed in the next section of this manual.

ATD	OCaml	JSON	Biniou
`unit`	`unit`	null	unit
`bool`	`bool`	boolean	bool
`int`	`int`	-?(0|[1-9][0-9]*)	svint
`float`	`float`	number	float64
`string`	`string`	string	string
`'a option`	`'a option`	`"None"` or `["Some", ...]`	numeric variants (tag 0)
`'a nullable`	`'a option`	`null` or representation of `'a`	numeric variants (tag 0)
`'a list`	`'a list`	array	array
`'a shared`	no wrapping	not implemented	no longer supported
`'a wrap`	defined by annotation, converted from `'a`	representation of `'a`	representation of `'a`
variants	polymorphic variants	variants	regular variants
record	record	object	record
`('a * 'b)`	`('a * 'b)`	array	tuple
`('a)`	`'a`	array	tuple

Notes:

- Null JSON fields by default are treated as if the field was missing. They can be made meaningful with the `keep_nulls` flag.
- JSON nulls are used to represent the unit value and is useful for instanciating parametrized types with "nothing".
- OCaml floats are written to JSON numbers with either a decimal point or an exponent such that they are distinguishable from ints, even though the JSON standard does not require a distinction between the two.
- The optional values of record fields denoted in ATD by a question mark are unwrapped or omitted in both biniou and JSON.
- JSON option values and JSON variants are represented in standard JSON (`atdgen -j -j-std`) by a single string e.g. `"None"` or a pair in which the first element is the name (constructor) e.g. `["Some", 1234]`. Yojson also provides a specific syntax for variants using edgy brackets: `<"None">`, `<"Some": 1234>`.
- Biniou field names and variant names other than the option types use the hash of the ATD field or variant name and cannot currently be overridden by annotations.
- JSON tuples in standard JSON (`atdgen -j -j-std`) use the array notation e.g. `["ABC", 123]`. Yojson also provides a specific syntax for tuples using parentheses, e.g. `("ABC", 123)`.
- Types defined as abstract are defined in another module.

3.2.4 ATD Annotations

Section `json`

Field `keep_nulls`

Position: after record

Values: none, `true` or `false`

Semantics: this flag, if present or set to true, indicates that fields whose JSON value is `null` should not be treated as if they were missing. In this case, `null` is parsed as a normal value, possibly of a `nullable` type.

Example: patch semantics

```
(* Type of the objects stored in our database *)
type t = {
  ?x : int option;
  ?y : int option;
  ?z : int option;
}
```

```
(* Type of the requests to modify some of the fields of an object. *)
type t_patch = {
  ?x : int nullable option; (* OCaml type: int option option *)
  ?y : int nullable option;
  ?z : int nullable option;
} <ocaml field_prefix="patch_"> <json keep_nulls>
```

Let's consider the following json patch that means "set `x` to 1, clear `y` and keep `z` as it is":

```
{
  "x": 1,
  "y": null
}
```

It will be parsed by the generated function `t_patch_of_string` into the following OCaml value:

```
{
  patch_x = Some (Some 1);
  patch_y = Some None;
  patch_z = None;
}
```

Then presumably some code would be written to apply the patch to an object of type `t`. Such code is not generated by atdgen at this time.

Available: from atd 1.12

Field `name`

Position: after field name or variant name

Values: any string making a valid JSON string value

Semantics: specifies an alternate object field name or variant name to be used by the JSON representation.

Example:

```
type color = [
    Black <json name="black">
  | White <json name="white">
  | Grey <json name="grey">
]

type profile = {
  id <json name="ID"> : int;
  username : string;
  background_color : color;
}
```

A valid JSON object of the `profile` type above is:

```
{
  "ID": 12345678,
  "username": "kimforever",
  "background_color": "black"
}
```

Field `repr`

Association lists

Position: after `(string * _) list` type

Values: `object`

Semantics: uses JSON's object notation to represent association lists.

Example:

```
type counts = (string * int) list <json repr="object">
```

A valid JSON object of the `counts` type above is:

```
{
  "bob": 3,
  "john": 1408,
  "mary": 450987,
  "peter": 93087
}
```

Without the annotation `<json repr="object">`, the data above would be represented as:

```
[
  [ "bob", 3 ],
  [ "john", 1408 ],
  [ "mary", 450987 ],
  [ "peter", 93087 ]
]
```

Floats

Position: after `float` type

Values: `int`

Semantics: specifies a float value that must be rounded to the nearest integer and represented in JSON without a decimal point nor an exponent.

Example:

```
type unixtime = float <json repr="int">
```

Ints

Position: after `int` type

Values: `string`

Semantics: specifies a int value that must be represented in JSON as a string.

Example:

```
type int64 = int <ocaml repr="int64"> <json repr="string">
```

Field `tag_field`

Superseded by `<json adapter.ocaml="...">`. Available since atdgen 1.5.0 and yojson 1.2.0 until atdgen 1.13.

This feature makes it possible to read JSON objects representing variants that use one field for the tag and another field for the untagged value of the specific type associated with that tag.

Position: on a record field name, for a field holding a variant type.

Value: name of another JSON field which holds the string representing the constructor for the variant.

Semantics: The type definition

```
type t = {
  value <json tag_field="kind">: [ A | B <json name="b"> of int ];
}
```

covers JSON objects that have an extra field `kind` which holds either `"A"` or `"b"`. Valid JSON values of type `t` include `{ "kind": "A" }` and `{ "kind": "b", "value": 123 }`.

Field `untyped`

Superseded by `<json open_enum>` and `<json adapter.ocaml="...">`. Available since atdgen 1.10.0 and atd 1.2.0 until atdgen 1.13.

This flag enables parsing of arbitrary variants without prior knowledge of their type. It is useful for constructing flexible parsers for extensible serializations. `json untyped` is compatible with regular variants, `json tag_field` variants, default values, and implicit `tag_field` constructors.

Position: on a variant constructor with argument type `string * json option` (at most one per variant type)

Value: none, `true` or `false`

Semantics: The type definition

```
type v = [
  | A
  | B <json name="b"> of int
  | Unknown <json untyped> of (string * json option)
]
```

will parse and print `"A"`, `["b", 0]`, `"foo"`, and `["bar", [null]]` in a regular variant context. In the `tag_field` type `t` context in the previous section, `v` will parse and print `{ "kind": "foo" }` and `{ "kind": "bar", "value": [null] }` as well as the examples previously given.

Field `open_enum`

Where an enum (finite set of strings) is expected, this flag allows unexpected strings to be kept under a catch-all constructor rather than producing an error.

Position: on a variant type comprising exactly one constructor with an argument. The type of that argument must be `string`. All other constructors must have no arguments.

Value: none

For example:

```
type language = [
  | English
  | Chinese
  | Other of string
] <json open_enum>
```

maps the json string `"Chinese"` to the OCaml value `` `Chinese `` and maps `"French"` to `` `Other "French" ``.

Available since atdgen 2.0.

Field `adapter.ocaml`

Json adapters are a mechanism for rearranging json data on-the-fly, so as to make them compatible with ATD. The programmer must provide an OCaml module that provides converters between the original json representation and the ATD-compatible representation. The signature of the user-provided module must be equal to `Atdgen_runtime.Json_adapter.S`, which is:

```
sig
  (** Convert from original json to ATD-compatible json *)
  val normalize : Yojson.Safe.t -> Yojson.Safe.t

  (** Convert from ATD-compatible json to original json *)
  val restore : Yojson.Safe.t -> Yojson.Safe.t
end
```

The type `Yojson.Safe.t` is the type of parsed JSON as provided by the yojson library.

Position: on a variant type or on a record type.

Value: an OCaml module identifier. Note that `Atdgen_runtime.Json_adapter` provides a few modules and functors that are ready to use. Users are however encouraged to write their own to suit their needs.

Sample ATD definitions:

```
type document = [
  | Image of image
  | Text of text
] <json adapter.ocaml="Atdgen_runtime.Json_adapter.Type_field">

type image = {
  url: string;
}

type text = {
  title: string;
  body: string;
}
```

ATD-compliant json values:

- `["Image", {"url": "https://example.com/ocean123.jpg"}]`
- `["Text", {"title": "Cheeses Around the World", "body": "..."}]`

Corresponding json values given by some API:

- `{"type": "Image", "url": "https://example.com/ocean123.jpg"}`
- `{"type": "Text", "title": "Cheeses Around the World", "body": "..."}`

The json adapter `Type_field` that ships with the atdgen runtime takes care of converting between these two forms. For information on how to write your own adapter, please consult the documentation for the yojson library.

Fields `adapter.to_ocaml` and `adapter.from_ocaml`

This is an alternative form of specifying `adapter.ocaml`. It permits to specify arbitrary code and doesn't require the "adapter" module to be defined in advance.

For example, the above usage of `adapter.ocaml` can be rewritten as following:

```
type document = [
  | Image of image
  | Text of text
]
<json
  adapter.to_ocaml="Atdgen_runtime.Json_adapter.normalize_type_field \"type\""
  adapter.from_ocaml="Atdgen_runtime.Json_adapter.restore_type_field \"type\""
>

type image = {
  url: string;
}

type text = {
  title: string;
  body: string;
}
```

Section `biniou`

Field `repr`

Integers

Position: after `int` type

Values: `svint` (default), `uvint`, `int8`, `int16`, `int32`, `int64`

Semantics: specifies an alternate type for representing integers. The default type is `svint`. The other integers types provided by biniou are supported by Atdgen-biniou. They have to map to the corresponding OCaml types in accordance with the following table:

Biniou type	Supported OCaml type	OCaml value range
`svint`	`int`	`min_int` ... `max_int`
`uvint`	`int`	0 ... `max_int`, `min_int` ... -1
`int8`	`char`	`'\000` ... `'\255`
`int16`	`int`	0 ... 65535
`int32`	`int32`	`Int32.min_int` ... `Int32.max_int`
`int64`	`int64`	`Int64.min_int` ... `Int64.max_int`

In addition to the mapping above, if the OCaml type is `int`, any biniou integer type can be read into OCaml data regardless of the declared biniou type.

Example:

```
type t = {
  id : int
    <ocaml repr="int64">
    <biniou repr="int64">;
  data : string list;
}
```

Floating-point numbers

Position: after `float` type

Values: `float64` (default), `float32`

Semantics: `float32` allows for a shorter serialized representation of floats, using 4 bytes instead of 8, with reduced precision. OCaml floats always use 8 bytes, though.

Example:

```
type t = {
  lat : float <biniou repr="float32">;
  lon : float <biniou repr="float32">;
}
```

Arrays and tables

Position: applies to lists of records

Values: `array` (default), `table`

Semantics: `table` uses biniou's table format instead of a regular array for serializing OCaml data into biniou. Both formats are supported for reading into OCaml data regardless of the annotation. The table format allows

Example:

```
type item = {
  id : int;
  data : string list;
}

type items = item list <biniou repr="table">
```

Section `ocaml`

Field `attr`

Position: on a type definition, i.e. on the left-handside just before the equal sign =

Semantics: specifies custom ppx attributes for the type definition. Overrides any default attributes set globally via the command line option `-type-attr`.

Values: the contents of a ppx annotation without the enclosing `[@@` and `]`

Example:

```
type foo <ocaml attr="deriving show,eq"> = int list
```

translates to

```
type foo = int list [@@deriving show,eq]
```

Field `predef`

Position: left-hand side of a type definition, after the type name

Values: none, `true` or `false`

Semantics: this flag indicates that the corresponding OCaml type definition must be omitted.

Example:

```
(* Some third-party OCaml code *)
type message = {
  from : string;
  subject : string;
  body : string;
}
```

```
(*
   Our own ATD file used for making message_of_string and
   string_of_message functions.
*)
type message <ocaml predef> = {
  from : string;
  subject : string;
  body : string;
}
```

Field `mutable`

Position: after a record field name

Values: none, `true` or `false`

Semantics: this flag indicates that the corresponding OCaml record field is mutable.

Example:

```
type counter = {
  total <ocaml mutable> : int;
  errors <ocaml mutable> : int;
}
```

translates to the following OCaml definition:

```
type counter = {
  mutable total : int;
  mutable errors : int;
}
```

Field `default`

Position: after a record field name marked with a `\~{}` symbol or at the beginning of a tuple field.

Values: any valid OCaml expression

Semantics: specifies an explicit default value for a field of an OCaml record or tuple, allowing that field to be omitted. Default strings must be escaped.

Example:

```
type color = [ Black | White | Rgb of (int * int * int) ]

type ford_t = {
  year : int;
  ~color <ocaml default="`Black"> : color;
  ~name <ocaml default="\"Ford Model T\""> : string;
}

type point = (int * int * <ocaml default="0"> : int)
```

Field `from`

Position: left-hand side of a type definition, after the type name

Values: OCaml module name without the `_t`, `_b`, `_j` or `_v` suffix. This can be also seen as the name of the original ATD file, without the `.atd` extension and capitalized like an OCaml module name.

Semantics: specifies the base name of the OCaml modules where the type and values coming with that type are defined.

It is useful for ATD types defined as `abstract` and for types annotated as predefined using the annotation `<ocaml predef>`. In both cases, the missing definitions must be provided by modules composed of the base name and the standard suffix assumed by Atdgen which is `_t`, `_b`, `_j` or `_v`.

Example: First input file `part1.atd`:

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

Second input file `part2.atd` depending on the first one:

```
type point <ocaml from="Part1"> = abstract
type points = point list
```

To use a different type name than defined in the `Part1` module, add a `t` field declaration to the annotation which refers to the original type name:

```
type point_xy <ocaml from="Part1" t="point"> = abstract
type points = point_xy list
```

Field `module`

Using a custom wrapper

Using the built-in `wrap` constructor, it is possible to add a layer of abstraction on top of the concrete structure used for serialization.

Position: after a `wrap` type constructor

Values: OCaml module name

A common use case is to parse strings used as unique identifiers and wrap the result into an abstract type. Our OCaml module `Uid` needs to provide a type `t`, and two functions `wrap` and `unwrap` as follows:

```
type t
val wrap : string -> t
val unwrap : t -> string
```

Given that `Uid` OCaml module, we can write the following ATD definition:

```
type uid = string wrap <ocaml module="Uid">
```

Other languages than OCaml using the same ATD type definitions may or may not add their own abstract layer. Without an annotation, the `wrap` construct has no effect on the value being wrapped, i.e. `wrap` and `unwrap` default to the identity function.

It is also possible to define `t`, `wrap`, and `unwrap` inline:

```
type uid = string wrap <ocaml t="Uid.t"
                              wrap="Uid.wrap"
                              unwrap="Uid.unwrap">
```

This can be useful for very simple validation:

```
type uid = string wrap
  <ocaml wrap="fun s ->
                 if String.length s <> 16 then
                   failwith \"Invalid user ID\";
                 s"
  >
```

Importing an external type definition

In most cases since Atdgen 1.2.0 `module` annotations are deprecated in favor of `from` annotations previously described.

Position: left-hand side of a type definition, after the type name

Values: OCaml module name

Semantics: specifies the OCaml module where the type and values coming with that type are defined. It is useful for ATD types defined as `abstract` and for types annotated as predefined using the annotation `<ocaml predef>`. In both cases, the missing definitions can be provided either by globally opening an OCaml module with an OCaml directive or by specifying locally the name of the module to use.

The latter approach is recommended because it allows to create type and value aliases in the OCaml module being generated. It results in a complete module signature regardless of the external nature of some items.

Example: Input file `example.atd`:

```
type document <ocaml module="Doc"> = abstract

type color <ocaml predef module="Color"> =
  [ Black | White ] <ocaml repr="classic">

type point <ocaml predef module="Point"> = {
  x : float;
  y : float;
}
```

gives the following OCaml type definitions (file `example.mli`):

```
type document = Doc.document

type color = Color.color =  Black | White

type point = Point.point = { x: float; y: float }
```

Now for instance `Example.Black` and `Color.Black` can be used interchangeably in other modules.

Field `t`

Using a custom wrapper

Specifies the OCaml type of an abstract `wrap` construct, possibly overriding the default *M*`.t` if *M* is the module where the `wrap` and `unwrap` functions are found.

Position: after a `wrap` type constructor

Values: OCaml type name

Example:

```
type uid = string wrap <ocaml module="Uid" t="Uid.uid">
```

is equivalent to:

```
type uid = string wrap <ocaml t="Uid.uid" wrap="Uid.wrap" unwrap="Uid.unwrap">
```

Importing an external type definition

Position: left-hand side of a type definition, after the type name. Must be used in conjunction with a `module` field.

Values: OCaml type name as found in an external module.

Semantics: This option allows to specify the name of an OCaml type defined in an external module.

It is useful when the type needs to be renamed because its original name is already in use or not enough informative. Typically we may want to give the name `foo` to a type originally defined in OCaml as `Foo.t`.

Example:

```
type foo <ocaml_biniou module="Foo" t="t"> = abstract
type bar <ocaml_biniou module="Bar" t="t"> = abstract
type t <ocaml_biniou module="Baz"> = abstract
```

allows local type names to be unique and gives the following OCaml type definitions:

```
type foo = Foo.t
type bar = Bar.t
type t = Baz.t
```

Fields `wrap` and `unwrap`

See "Using a custom wrapper" under section `ocaml`, fields `module` and `t`.

Field `field_prefix`

Position: record type expression

Values: any string making a valid prefix for OCaml record field names

Semantics: specifies a prefix to be prepended to each field of the OCaml definition of the record. Overridden by alternate field names defined on a per-field basis.

Example:

```
type point2 = {
  x : int;
  y : int;
} <ocaml field_prefix="p2_">
```

gives the following OCaml type definition:

```
type point2 = {
  p2_x : int;
  p2_y : int;
}
```

Field `name`

Position: after record field name or variant name

Values: any string making a valid OCaml record field name or variant name

Semantics: specifies an alternate record field name or variant names to be used in OCaml.

Example:

```
type color = [
    Black <ocaml name="Grey0">
  | White <ocaml name="Grey100">
  | Grey <ocaml name="Grey50">
]

type profile = {
  id <ocaml name="profile_id"> : int;
  username : string;
}
```

gives the following OCaml type definitions:

```
type color = [
    `Grey0
  | `Grey100
  | `Grey50
]

type profile = {
  profile_id : int;
  username : string;
}
```

Field `repr`

Integers

Position: after `int` type

Values: `char`, `int32`, `int64`, `float`

Semantics: specifies an alternate type for representing integers. The default type is `int`, but `char`, `int32`, `int64` or `float` can be used instead.

The three types `char`, `int32` and `int64` are supported by both Atdgen-biniou and Atdgen-json but Atdgen-biniou currently requires that they map to the corresponding fixed-width types provided by the biniou format.

The type `float` is only supported in conjunction with JSON and is useful when an OCaml float is used to represent an integral value, such as a time in seconds returned by `Unix.time()`. When converted into JSON, floats are rounded to the nearest integer.

Example:

```
type t = {
  id : int
    <ocaml repr="int64">
    <biniou repr="int64">;
  data : string list;
}
```

Lists and arrays

Position: after a `list` type

Values: `array`

Semantics: maps to OCaml's `array` type instead of `list`.

Example:

```
type t = {
  id : int;
  data : string list
    <ocaml repr="array">;
}
```

Sum types

Position: after a sum type (denoted by square brackets)

Values: `classic`

Semantics: maps to OCaml's classic variants instead of polymorphic variants.

Example:

```
type fruit = [ Apple | Orange ] <ocaml repr="classic">
```

translates to the following OCaml type definition:

```
type fruit = Apple | Orange
```

Shared values (obsolete)

Position: after a `shared` type

This feature is obsolete and was last supported by atdgen 1.3.1.

Field `valid`

Since atdgen 1.6.0.

Position: after any type expression except type variables

Values: OCaml function that takes one argument of the given type and returns a bool

Semantics: `atdgen -v` produces for each type named *t* a function `validate_t`:

```
val validate_t : Atdgen_runtime.Util.Validation.path -> t -> Atdgen_runtime.Util.
↪Validation.error option
```

Such a function returns `None` if and only if the value and all of its subnodes pass all the validators specified by annotations of the form `<ocaml validator="...">` or `<ocaml valid="...">` (at most one per node).

Example:

```
type positive = int <ocaml validator="fun x -> x > 0">

type point = {
  x : positive;
  y : positive;
  z : int;
}
  <ocaml valid="Point.validate">
  (* Some validating function from a user-defined module Point *)
```

The generated `validate_point` function calls the validator for the containing object first (`Point.validate`) and continues on its fields `x` then `y` until an error is returned.

```
match validate_point [] { x = 1; y = 0; z = 1 } with
| None -> ()
| Some e ->
    Printf.eprintf "Error: %s\n%!"
      (Atdgen_runtime.Util.Validation.string_of_error e)
```

The above code prints the following error message:

```
Error: Validation error; path = <root>.y
```

In order to customize the error message and print the faulty value, use `validator` instead of `valid`, as described next.

Field `validator`

This is a variant of the `valid` annotation that allows full control over the error message that gets generated in case of an error.

Position: after any type expression except type variables

Values: OCaml function that takes the path in current JSON structure and the object to validate, and returns an optional error.

Semantics: `atdgen -v` produces for each type named *t* a function `validate_t`:

```
val validate_t : Atdgen_runtime.Util.Validation.path -> t -> Atdgen_runtime.Util.
↪Validation.error option
```

Such a function returns `None` if and only if the value and all of its subnodes pass all the validators specified by annotations of the form `<ocaml validator="...">` or `<ocaml valid="...">` (at most one per node).

Example:

```
type positive = int <ocaml validator="
  fun path x ->
    if x > 0 then None
    else
      Some (
        Atdgen_runtime.Util.Validation.error
          ~msg: (\"Not a positive integer: \" ^ string_of_int x)
          path
      )
">
```

(continues on next page)

(continued from previous page)

```
type point = {
  x : positive;
  y : positive;
  z : int;
}
  <ocaml validator="Point.validate">
  (* Some validating function from a user-defined module Point *)
```

The following user code

```
match Toto_v.validate_point [] { x = 1; y = 0; z = 1 } with
| None -> ()
| Some e ->
    Printf.eprintf "Error: %s\n%!"
      (Atdgen_runtime.Util.Validation.string_of_error e)
```

results in printing:

```
Error: Validation error: Not a positive integer: 0; path = <root>.y
```

Section `ocaml_biniou`

Section `ocaml_biniou` takes precedence over section `ocaml` in Biniou mode (`-b`) for the following fields:

- `predef` (see section `ocaml`, field `predef`)
- `module` (see section `ocaml`, field `module`)
- `t` (see section `ocaml.t`)

Section `ocaml_json` (obsolete)

Section `ocaml_json` takes precedence over section `ocaml` in JSON mode (`-json` or `-j`) for the following fields:

- `predef` (see section `ocaml`, field `predef`)
- `module` (see section `ocaml`, field `module`)
- `t` (see section `ocaml`, field `t`)

Please note that `atdgen -json` is now deprecated in favor of `atdgen -j` (json) and `atdgen -t` (types). The latter is in charge of producing type definitions independently from JSON and will ignore `<ocaml_json ...>` annotations, making them almost useless. The equivalent `<ocaml ...>` annotations are almost always preferable.

Example:

This example shows how to parse a field into a generic tree of type `Yojson.Safe.t` rather than a value of a specialized OCaml type.

```
type dyn <ocaml_json module="Yojson.Safe" t="json"> = abstract

type t = { foo: int; bar: dyn }
```

translates to the following OCaml type definitions:

```
type dyn = Yojson.Safe.t

type t = { foo : int; bar : dyn }
```

Sample OCaml value of type `t`:

```
{
  foo = 12345;
  bar =
    `List [
      `Int 12;
      `String "abc";
      `Assoc [
        "x", `Float 3.14;
        "y", `Float 0.0;
        "color", `List [ `Float 0.3; `Float 0.0; `Float 1.0 ]
      ]
    ]
}
```

Corresponding JSON data as obtained with `string_of_t`:

```
{"foo":12345,"bar":[12,"abc",{"x":3.14,"y":0.0,"color":[0.3,0.0,1.0]}]}
```

Section `doc`

Unlike comments, `doc` annotations are meant to be propagated into the generated source code. This is useful for making generated interface files readable without having to consult the original ATD file.

Generated source code comments can comply to a standard format and take advantage of documentation generators such as javadoc or ocamldoc.

Field `text`

Position:

- after the type name on the left-hand side of a type definition
- after the type expression on the right hand of a type definition (but not after any type expression)
- after record field names
- after variant names

Values: UTF-8-encoded text using a minimalistic markup language

Semantics: The markup language is defined as follows:

- Blank lines separate paragraphs.
- `{{ }}` can be used to enclose inline verbatim text.
- `{{{ }}}` can be used to enclose verbatim text where whitespace is preserved.
- The backslash character is used to escape special character sequences. In regular paragraph mode the special sequences are `\`, `{{` and `{{{`. In inline verbatim text, special sequences are `\` and `}}`. In verbatim text, special sequences are `\` and `}}}`.

Example: The following is an example demonstrating the use of `doc` annotations generated using:

```
$ atdgen -t ocamldoc_example.atd
```

Input file `ocamldoc_example.atd`:

```
<doc text="This is the title">

type point = {
  x <doc text="The first coordinate">: float;
  y <doc text="The second coordinate">: float;
}
  <doc text="
The type of a point. A value {{p}} can be created as follows:
{{{
let p = { x = 1.2; y = 5.0 }
}}}
">

type color = [
 | Black <doc text="Same as {{RGB (0,0,0)}}">
 | White <doc text="Same as {{RGB (255, 255, 255)}}">
 | RGB
     <doc text="Red, green, blue components">
     of (int * int * int)
]
```

translates using `atdgen -t ocamldoc_example.atd` into the following OCaml interface file `ocamldoc_example_t.mli` with ocamldoc-compliant comments:

```
(* Auto-generated from "ocamldoc_example.atd" *)


(** This is the title *)

(**
  The type of a point. A value [p] can be created as follows:

{v
let p = \{ x = 1.2; y = 5.0 \}
v}
*)
type point = {
  x: float (** The first coordinate *);
  y: float (** The second coordinate *)
}

type color = [
    `Black (** Same as [RGB (0,0,0)] *)
  | `White (** Same as [RGB (255, 255, 255)] *)
  | `RGB of (int * int * int) (** Red, green, blue components *)
]
```

3.2.5 Atdgen runtime library

A library named atdgen-runtime is installed by the standard installation process. Only a fraction of it is officially supported and documented.

Modules intended for all users are:

- `Util`
- `Json_adapter`

The other modules exported by the library are used directly by generated code. Tool developers may use them but we don't guarantee strong compatibility across releases.

CHAPTER 4

Java Support - atdj

The ATDJ tool generates a Java interface from an ATD interface. In particular, given a set of ATD types, this tool generates a set of Java classes representing those types. These classes may then be instantiated from JSON representations of those same ATD types.

The primary benefits of using the generated interface, over manually manipulating JSON strings from within Java, are safety and ease of use. Specifically, the generated interface offers the following features:

- JSON strings are automatically checked for correctness with respect to the ATD specificion.
- Details such as optional fields and their associated default values are automatically handled.
- Several utility methods are included "for free". These support equality testing, the visitor pattern and conversion back to JSON.

4.1 Installation

Build and install the `atdj` command with opam:

```
opam install atdj
```

4.2 Quick-start

In this section we briefly describe how to to generate a Java interface from an example ATD file `test.atd`. We then show how to build and run an example application `AtdjTest` that uses the generated interface.

1. Generate and compile the interface:

```
atdj -graph -package com.mylife.test test.atd
export CLASSPATH='.:json.jar'
javac com/mylife/test/*.java
```

2. Compile and run the example, saving the output for later inspection:

```
javac AtdjTest.java
java AtdjTest >test.out
```

3. Optionally, generate Javadoc documentation:

```
javadoc -d doc -public com.mylife.test
```

   The resulting documentation is located in the directory `doc`.

4. Optionally, generate a class graph of the generated interface:

```
dot -Tpdf test.dot >test.pdf
```

The output file `test.pdf` contains a class graph of the generated Java interface. The required `dot` program is part of the Graphviz graph visualisation package, and may be downloaded from http://www.graphviz.org/.

In the following sections we discuss the individual steps in more detail, using the example from above.

4.3 Generating the interface

In this section we describe the process of generating a Java interface from an ATD specification.

A Java interface is generated from an ATD file as

```
atdj -package <package> <atd_file>
```

This outputs a set of Java source files. The `-package` option causes the resulting classes to be members of the specified package, and also to be located in the corresponding output directory. If no package is specified, then the default package of `out` is used.

For example, the command

```
atdj -graph -package com.mylife.test test.atd
```

causes the generated files to be members of the package `com.mylife.test` and to be located in the directory `com/mylife/test`.

The generated source files reference various members of the included org.json package. Therefore, in order to compile the generated files, the `org.json` package must be located within the Java classpath. Supposing that the `org.json` package is located within the archive `json.jar` within the current directory, it is sufficient to set the classpath as follows:

```
export CLASSPATH='json.jar'
```

Returning to our example, the generated source files may then be compiled as:

```
javac com/mylife/test/*.java
```

4.4 Generating Javadoc documentation

The generated Java code contains embedded Javadoc comments. These may be extracted to produce Javadoc documentation. In the case of our example, it is sufficient to run the following command:

```
javadoc -d doc/example -public com.mylife.test
```

4.5 Generating a class graph

We now discuss the `-graph` option of ATDJ. When enabled, this causes ATDJ to output a graph of the class hierarchy of the generated code. The output is intended to document the generated code, helping users to avoid consulting the source code.

Continuing with our example, the use of this option results in the generation of an additional output file named `test.dot`. Assuming that the `dot` program is installed, a PDF class graph named `test.pdf` can then created by running the command

```
dot -Tpdf test.dot >test.pdf
```

In the generated class graph, rectangular and oval nodes correspond to classes and interfaces, respectively. Field names are specified in the second line of retangular (class) nodes. Solid arcs denote subtyping (`implements`/`extends`), whilst dashed arcs link fields to their types.

4.6 Translation reference

In this section we informally define how Java types are generated from ATD types.

4.6.1 Bools, ints, floats, string, lists

ATD type, t	Java type, <t>
bool	boolean
int	int
float	double
string	String
t list	<t>[]

4.6.2 Options

Suppose that we have ATD type `t option`. Then this is translated into the following Java reference type:

```
public class CNAME implements Atdj {
  // Constructor
  public CNAME(String s) throws JSONException { ... }

// Get the optional value, if present
public CNAME get() throws JSONException     { ... }

// Comparison and equality
public int     compareTo(CNAME that)        { ... }
public boolean equals(CNAME that)           { ... }

public <t> value;           // The value
public boolean is_set;      // Whether the value is set
}
```

4.6.3 Records

Suppose that we have the ATD record type

```
{ f_1: t_1
;  ...
; f_n: t_n
}
```

Then this is translated into the following Java reference type:

```
public class CNAME implements Atdj {
  // Constructor
  public CNAME(String s) throws JSONException { ... }

// Comparison and equality
public int     compareTo(CNAME that)        { ... }
public boolean equals(CNAME that)           { ... }

// The individual fields
public <t_1> f_1;
...
public <t_n> f_n;
}
```

An optional field `~f_i: t_i` causes the class field `f_i` to be given a default value of type `<t_i>` if the field is absent from the JSON string used to instantiate the class. The default values are as follows:

ATD type	Default Java value
bool	false
int	0
float	0.0
string	“”
t list	Empty array
t option	Optional value with is_set = false

Default values cannot be defined for record and sum types.

An optional field `?f_i: t_i option` has the same default behaviour as above, with the additional behaviour that if the field is present in the JSON string then the value must be of type <t> (not <t> option); the value is then automatically lifted into a <t> option, with is_set = true.

4.6.4 Sums

Suppose that we have the ATD sum type

```
[ C_1 of t_1
| ...
| C_n of t_n
]
```

Then this is translated into the following Java reference types:

```
public interface IFCNAME extends Atdj {
  public int     compareTo(IFCNAME that);
  public boolean equals(IFCNAME that);
  ...
}
```

```
public class CNAME_i implements IFCNAME, Atdj {
  // Comparison and equality
  public int     compareTo(CNAME that)        { ... }
  public boolean equals(CNAME that)           { ... }

public <t_i> value;
}
```

The value field is absent if the constructor C_i has no argument.

4.6.5 The Atdj and Visitor interfaces

All generated reference types additionally implement the interface

```
interface Atdj {
  String toString();
  String toString(int indent);
  int hashCode();
  Visitor accept(Visitor v);
}
```

where the Visitor interface is defined as

```
public interface Visitor {
  public void visit(CNAME_1 value);
  ...
  public void visit(CNAME_n value);
}
```

for generated reference types `CNAME_i`. Visit methods for primitive and optional primitive types are omitted.

CHAPTER 5

Scala Support - atds

CHAPTER 6

Python Support - atdpy

This documentation is incomplete. Your help would be appreciated! In particular, some how-to guides would be great.

6.1 Tutorials

6.1.1 Hello World

Install `atdpy` with `opam`:

```
opam install atdpy
```

Create a file `hello.atd` containing this:

```
type message = {
  subject: string;
  body: string;
}
```

Call `atdpy` to produce `hello.py`:

```
$ atdpy hello.atd
```

There's now a file `hello.py` that contains a class looking like this:

```
...

@dataclass
class Message:
  """Original type: message = { ... }"""

  subject: str
  body: str
```

(continues on next page)

(continued from previous page)

```
  @classmethod
  def from_json(cls, x: Any) -> 'Message':
      ...

  def to_json(self) -> Any:
      ...

  @classmethod
  def from_json_string(cls, x: str) -> 'Message':
      ...

  def to_json_string(self, **kw: Any) -> str:
      ...
```

Let's write a Python program `say_hello.py` that uses this code:

```
import hello

msg = hello.Message("Hello", "Dear friend, I hope you are well.")
print(msg.to_json_string())
```

Running it will print the JSON message:

```
$ python3 say_hello.py
{"subject": "Hello", "body": "Dear friend, I hope you are well."}
```

Such JSON data can be parsed. Let's write a program `read_message.py` that consumes JSON data from standard input:

```
import hello, sys, json

data = json.load(sys.stdin)
msg = hello.Message.from_json(data)
print(f"subject: {msg.subject}")
```

Output:

```
$ echo '{"subject": "big news", "body": ""}' | python3 read_message.py
subject: big news
```

It works! But what happens if the JSON data lacks a `"subject"` field? Let's see:

```
$ echo '{"subj": "big news", "body": ""}' | python3 read_message.py
Traceback (most recent call last):
...
ValueError: missing field 'subject' in JSON object of type 'Message'
```

And what if our program also thought that the correct field name was `subj` rather than subject? Here's `read_message_wrong.py` which tries to access a `subj` field:

```
import hello, sys, json

data = json.load(sys.stdin)
msg = hello.Message.from_json(data)
print(f"subject: {msg.subj}")
```

Let's run the program through mypy:

```
$ mypy read_message_wrong.py
read_message_wrong.py:5: error: "Message" has no attribute "subj"
Found 1 error in 1 file (checked 1 source file)
```

Mypy detected that our program makes incorrect assumptions about the message format without running it. On the correct program `read_message.py`, we get a reassuring message:

```
$ mypy read_message.py
Success: no issues found in 1 source file
```

6.1.2 ATD Records, JSON objects, Python classes

An ATD file contains types that describe the structure of JSON data. JSON objects map to Python classes and objects. They're called records in the ATD language. Let's define a simple record type in the file `hello_plus.atd`:

```
type message = {
  subject: string;
  ~body: string;
}
```

Note the ~ in front of the `body` field. It means that this field has a default value. Whenever the JSON field is missing from a JSON object, a default value is assumed. The implicit default value for a string is `""`.

Let's add a `signature` field whose default value isn't the empty string:

```
type message = {
  subject: string;
  ~body: string;
  ~signature <python default="'anonymous'">: string;
}
```

Finally, we'll add an optional `url` field that doesn't take a default value at all:

```
type message = {
  subject: string;
  ~body: string;
  ~signature <python default="'anonymous'">: string;
  ?url: string option;
}
```

Let's generate the Python code for this.

```
$ atdpy hello_plus.atd
```

Let's update our reader program `read_message_plus.py` to this:

```
import hello_plus, sys, json

data = json.load(sys.stdin)
msg = hello_plus.Message.from_json(data)
print(msg)
```

We can test it, showing us the final value of each field:

```
$ echo '{"subject":"hi"}' | python3 read_message_plus.py
Message(subject='hi', body='', signature='anonymous', url=None)
```

6.2 How-to guides

6.2.1 Defining default field values

[missing]

6.2.2 Renaming field names

[missing]

6.3 Deep dives

[missing]

6.4 Reference

6.4.1 Type mapping

ATD type	Python type	JSON example
`unit`	`None`	`null`
`bool`	`bool`	`True`
`int`	`int`	`42`
`float`	`float`	`6.28`
`string`	`str`	`"Hello"`
`int list`	`List[int]`	`[1, 2, 3]`
`(int * int)`	`Tuple[int, int]`	`[-1, 1]`
`int nullable`	`Optional[int]`	`42` or `null`
`int option`	`Optional[int]`	`["Some", 42]` or `"None"`
`abstract`	`Any`	anything
record type	class	`{"id": 17}`
`[A | B of int]`	`Union[A, B]`	`"A"` or `["B", 5]`
`foo_bar`	`FooBar`	

6.4.2 Supported ATD annotations

Default field values

Record fields following a ~ assume a default value. The default value can be implicit as mandated by the ATD language specification (false for `bool`, zero for `int`, etc.) or it can be a user-provided value.

A user-provided default uses an annotation of the form `<python default="VALUE">` where `VALUE` evaluates to a Python expression e.g.

```
type foo = {
  ~answer <python default="42">: int;
}
```

Default values are always honored when reading JSON data from Python. However, the implementation of `dataclass` via the `@dataclass` decorator prevents the use of mutable values for defaults. This causes class constructors to not have default fields that are mutable such as `[]`. For example:

```
type bar = {
  ~items: int list;
}
```

will translate to a class constructor that requires one argument of type list. For example, `Bar([1, 2, 3])` would be legal but `Bar()` would be illegal. Reading from the JSON object `{}` would however succeed. Therefore, the following two Python expressions would be valid and equivalent:

```
Bar([])
Bar.from_json_string('{}')
```

Field and constructor renaming

Alternate JSON object field names can be specified using an annotation of the form `<json name="NAME">` where `NAME` is the desired field name to be used in the JSON representation. For example, the following specifies the JSON name of the `id` field is `ID`:

```
type foo = {
  id <json name="ID">: string
}
```

Similarly, the constructor names of sum types can also be given alternate names in the JSON representation. Here's an example:

```
type bar = [
| Alpha <json name="alpha">
| Beta <json name="beta"> of int
]
```

Note that field names and constructor names in the generated Python code are assigned automatically so as to avoid conflicts with Python keywords or reserved identifiers.

Alternate representations for association lists

List of pairs can be represented by JSON objects or by Python dictionaries if the correct annotations are provided:

- `(string * bar) list <json repr="object">` will use JSON objects to represent a list of pairs of Python type `List[str, Bar]`. Using the annotation `<json repr="array">` is equivalent to the default.
- `(foo * bar) list <python repr="dict">` will use a Python dictionary of type `Dict[Foo, Bar]` to represent the association list. Using the annotation `<python repr="list">` is equivalent to the default.

Additional imports

At the beginning of the ATD file, placing annotations like this one allow inserting arbitrary Python code or comments:

```
<python text="import deco">
```

This is the recommended mechanism for inserting imports. In contrast, it should be used only as last resort for inserting functions or classes.

In the future, atdpy may generate more than one kind of files. An annotation of the form `<python text="...">` will insert that text into all the generated files. In order to insert code only in the `.py` file that handles JSON, it is recommended to use a more specific annotation of the form `<python json_py.text="...">`:

```
<python json_py.text="import deco">
```

Custom class decorators

Extra class decorators can be specified in addition to `@dataclass`. The following ATD definition will add 3 decorators:

```
type thing <python decorator="deco.deco1"
                   decorator="deco.deco2(42)"
                   decorator="dataclass(order=True)"> = {
  foo: int;
  bar: string;
}
```

The generated Python class will start like this:

```
@deco.deco1
@deco.deco2(42)
@dataclass(order=True)
@dataclass
class Thing:
    ...
```

If extra class decorators are specifed on a sum type, the python classes generated for the constructors of the sum type will also have the extra class decorators.

CHAPTER 7

TypeScript Support - atdts

This documentation is incomplete. Your help would be appreciated! In particular, some how-to guides would be great.

7.1 Tutorials

7.1.1 Hello World

Install `atdts` with `opam`:

```
opam install atdts
```

Create a file `hello.atd` containing this:

```
type message = {
  subject: string;
  body: string;
}
```

Call `atdts` to produce `hello.ts`:

```
$ atdts hello.atd
```

There's now a file `hello.ts` that contains a class looking like this:

```
...

export type Message = {
  subject: string;
  body: string;
}

export function writeMessage(x: Message, context: any = x): any {
```

(continues on next page)

(continued from previous page)

```
  ...
}

export function readMessage(x: any, context: any = x): Message {
  ...
}

...
```

Let's write a TypeScript program `say_hello.ts` that uses this code:

```
import * as hello from "./hello"

const msg: hello.Message = {
  subject: "Hello",
  body: "Dear friend, I hope you are well."
}

console.log(JSON.stringify(hello.writeMessage(msg)))
```

Running it will print the JSON message:

```
$ tsc --lib es2017,dom say_hello.ts
{"subject":"Hello","body":"Dear friend, I hope you are well."}
```

Such JSON data can be parsed. Let's write a program `read_message.ts` that consumes JSON data from standard input:

```
import * as hello from "./hello"
import * as readline from "readline"

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
})

rl.question('', (data: string) => {
  const msg = hello.readMessage(JSON.parse(data))
  console.log("subject: " + msg.subject)
})
```

Output:

```
# Install dependencies
$ npm install --save-dev @types/node
$ npm install readline

# Compile
$ tsc --lib es2017,dom read_message.ts

# Run
$ echo '{"subject": "big news", "body": ""}' | js read_message.js
subject: big news
```

It works! But what happens if the JSON data lacks a `"subject"` field? Let's see:

```
$ echo '{"body": ""}' | js read_message.js
{"body": ""}
readline.js:1086
            throw err;
            ^

Error: missing field 'subject' in JSON object of type 'Message'
...
```

And what if our program also thought that the correct field name was `subj` rather than subject? Here's `read_message_wrong.ts` which tries to access a `subj` field:

```
import * as hello from "./hello"
import * as readline from "readline"

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
})

rl.question('', (data: string) => {
  const msg = hello.readMessage(JSON.parse(data))
  console.log("subject: " + msg.subj)
})
```

Let's compile our program:

```
$ tsc --lib es2017,dom read_message_wrong.ts
read_message_wrong.ts:11:33 - error TS2339: Property 'subj' does not exist on type
↪'Message'.

11   console.log("subject: " + msg.subj)
                                   ~~~~


Found 1 error in read_message_wrong.ts:11
```

The typechecker detected that our program makes incorrect assumptions about the message format without running it.

7.1.2 ATD Records, JSON objects, TypeScript objects

An ATD file contains types that describe the structure of JSON data. JSON objects map to TypeScript types and objects. They're called records in the ATD language. Let's define a simple record type in the file `hello_plus.atd`:

```
type message = {
  subject: string;
  ~body: string;
}
```

Note the ~ in front of the `body` field. It means that this field has a default value. Whenever the JSON field is missing from a JSON object, a default value is assumed. The implicit default value for a string is `""`.

Let's add a `signature` field whose default value isn't the empty string:

```
type message = {
  subject: string;
  ~body: string;
  ~signature <ts default="'anonymous'">: string;
}
```

Finally, we'll add an optional `url` field that doesn't take a default value at all:

```
type message = {
  subject: string;
  ~body: string;
  ~signature <ts default="'anonymous'">: string;
  ?url: string option;
}
```

Let's generate the TypeScript code for this.

```
$ atdts hello_plus.atd
```

Let's update our reader program `read_message_plus.ts` to this:

```
import * as hello_plus from "./hello_plus"
import * as readline from "readline"

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
})

rl.question('', (data: string) => {
  const msg = hello_plus.readMessage(JSON.parse(data))
  console.log(msg)
})
```

We can test it, showing us the final value of each field:

```
$ tsc --lib es2017,dom read_message_plus.ts
$ echo '{"subject":"hi"}' | js read_message_plus.js
{"subject":"hi"}
{ subject: 'hi',
  body: '',
  signature: 'anonymous',
  url: undefined }
```

7.2 How-to guides

7.2.1 Defining default field values

[missing]

7.2.2 Renaming field names

[missing]

7.3 Deep dives

[missing]

7.4 Reference

7.4.1 Type mapping

ATD type	TypeScript type	JSON example
`unit`	`null`	`null`
`bool`	`bool`	`True`
`int`	`Int*`	`42` or `42.0`
`float`	`number`	`6.28`
`string`	`string`	`"Hello"`
`string list`	`string[]`	`["a", "b", "c!"]`
`(bool * float)`	`[boolean, number]`	`[-1, 1]`
`int nullable`	`Int | null`	`42` or `null`
`abstract`	`any`	anything
`{ id: string }`	`{ id: string }`	`{"id": "3hj8d"}`
`[A | B of int]`	`{kind: 'A'} | {kind: 'B', value: Int}`	`"A"` or `["B", 5]`
`foo_bar`	`FooBar`	

*the `Int` type is an alias for `number` but additionally, the read and write functions generated by atdts check that the number is a whole number.

7.4.2 Supported ATD annotations

Default field values

Record fields following a ~ assume a default value. The default value can be implicit as mandated by the ATD language specification (false for `bool`, zero for `int`, etc.) or it can be a user-provided value.

A user-provided default uses an annotation of the form `<ts default="VALUE">` where `VALUE` evaluates to a TypeScript expression e.g.

```
type foo = {
  ~answer <ts default="42">: int;
}
```

For example, the JSON value `{}` will be read as `{answer: 42}`.

Field and constructor renaming

Alternate JSON object field names can be specified using an annotation of the form `<json name="NAME">` where `NAME` is the desired field name to be used in the JSON representation. For example, the following specifies the JSON name of the `id` field is `ID`:

```
type foo = {
  id <json name="ID">: string
}
```

Similarly, the constructor names of sum types can also be given alternate names in the JSON representation. Here's an example:

```
type bar = [
| Alpha <json name="alpha">
| Beta <json name="beta"> of int
]
```

Alternate representations for association lists

List of pairs can be represented by JSON objects or by TypeScript maps if the correct annotations are provided:

- `(string * bar) list <json repr="object">` will use JSON objects to represent a list of pairs of TypeScript type `[string, Bar][]`. Using the annotation `<json repr="array">` is equivalent to the default.
- `(foo * bar) list <ts repr="map">` will use a TypeScript map of type `Map<Foo, Bar>` to represent the association list. Using the annotation `<ts repr="array">` is equivalent to the default.