ATD
Adjustable Type Definitions
release 1.0.1

Martin Jambon

February 8, 2011

Contents

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 the 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.

1.1 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 with 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.

1.2 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;
  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;
}
```

1.3 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: atdcat FILE
  -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
  -ml <name>
          output the ocaml code of the ATD abstract syntax tree
  -version
          print the version of atd and exit
  -help  Display this list of options
  --help  Display this list of options
```

2 ATD language

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

2.1 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

2.2 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 | comment	*discarded*
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* `"`	*string literal, used in annotations*
substring	::=	`\\`	*single backslash*
	|	`\"`	*double 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* `"`
digit	::=	`0`...`9`	
hex	::=	`0`...`9` | `a`...`f` | `A`...`F`	
keyword	::=	`(` | `)` | `[` | `]` | `{` | `}` | `<` | `>`	
		| `;` | `,` | `:` | `*` | `|` | `=` | `?` | `~`	
		| `type` | `of` | `inherit`	*all keywords*

2.3 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
```

2.4 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`	Zero or one element
`'a list`	Collection or sequence of elements
`'a shared`	Values for which sharing must be preserved
`abstract`	Type defined elsewhere

2.5 Shared values

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
```

3 OCaml atd library

The documentation for the `atd` library is available in HTML form at `http://oss.wink.com/atd/atd-1.0.1/odoc/index.html`.