BuckleScript User Manual

Hongbo Zhang – Version 1.2.1+Dev

Table of Contents

BuckleScript is a backend for the OCaml compiler which emits JavaScript. It works with both vanilla OCaml and Reason, the whole compiler is compiled into JS (and ASM) so that you can play it in the browser.

NOTE

PDF version is available.

Document under *bloomberg.github.io* matches with master branch.

They are also distributed (`docs/Manual.html` , `docs/Manual.pdf`) together with your Installation (with the exact version). If you find errors or omissions in this document, please don't hesitate to submit an issue, sources are here.

Why BuckleScript

Benefits of JavaScript platform

JavaScript is not just *the* browser language, it's also the *only* existing cross platform language. It is truly everywhere: users don't need to install binaries or use package managers to access software, just a link will work.

Another important factor is that the JavaScript VM is quite fast and keeps getting faster. The JavaScript platform is therefore increasingly capable of supporting large applications.

Problems of JavaScript && how BuckleScript solves it

BuckleScript is mainly designed to solve the problems of *large scale* JavaScript programming:

Type-safety

OCaml offers an industrial-strength state-of-the-art type system and provides very strong type inference (i.e. No verbose type annotation required compared with typescript), which proves invaluable in managing large projects. OCaml's type system is not just for tooling, it is a *sound* type system which means it is guaranteed that there will be no runtime type errors after type checking.

High quality dead code elimination

A large amount of web-development relies on inclusion of code dependencies by copying or referencing CDNs (the very thing that makes JavaScript highly accessible), but this also introduces a lot of dead code. This impacts performance adversely when the JavaScript VM has to interpret code that will never be invoked. BuckleScript provides powerful dead-code elimination at all levels:

- Function and module level elimination is facilitated by the sophistication of the type-system of OCaml and *purity analysis*.
- At the global level BuckleScript generates code ready for dead-code elimination done by bundling tools such as the Google closure-compiler.

Offline optimizations

JavaScript is a dynamic language, it takes a performance-hit for the VM to optimize code at runtime. While some JS engines circumvent the problem to some extent by caching, this is not available to all environments, and lack of a strong type system also limits the level of optimizations possible. Again, BuckleScript, using features of the OCaml type-system and compiler implementation is able to provide many optimizations during offline compilation, allowing the runtime code to be extremely fast.

JS platform and Native platform

Run your programs on all platforms, but run your system*faster* under specific platforms. Javascript is everywhere but it does not mean we have to run all apps in JS, under several platforms, for example, server side or iOS/Android native apps, when programs are written in OCaml, it can also be compiled to native code for *better and reliable performance*.

While a strong type-system helps in countering these problems, at the same time we hope to avoid some of the problems faced in using other offline transpilation systems:

Slow compilation

OCaml byte-code compilation is known to be fast (one or two orders of magnitude faster than other similar languages: Scala or Haskell), BuckleScript shares the same property and compiles even faster since it saves the link time. See the speeds at work in the playground, the native backend is one order faster than the JS backend.

Un-readable JS Code and hard to integrate with existing JS libraries

When compiling to JavaScript, many systems generate code, that while syntactically and semantically correct is not human-readable and very difficult to debug and profile. Our BuckleScript implementation and the multi-pass compilation strategy of OCaml, allows us to avoid name-mangling, and produce JavaScript code that is human-readable and easier to debug and maintain.

More importantly, this makes integration with existing JS libraries *much easier*.

Large JS output even for a simple program

In BuckleScript, a `Hello world` program generates *20 bytes* JS code instead of *50K bytes*. This is due to that BuckleScript has an excellent integration with JS libs that unlike most JS compilers, all BuckleScript's runtime is written in OCaml itself so that these runtime libraries are only needed when user actually call it.

Loss of code-structure

Many systems generate JavaScript code that is essentially a big ball of mud. We try to keep the original structure of the code by mapping one OCaml module to one JS module.

Installation

> NOTE
>
> *WINDOWS prerequisite*
>
> BuckleScript works natively on Windows, currently users have to install OCaml Cygwin first, make sure that `ocamlopt` is in `PATH`.
>
> ```
> npm install bs-platform
> ```
>
> After installation, BuckleScript does not rely on Cygwin anymore.

Install from NPM registries

Prerequisites

- Standard C toolchain
- `npm` (should be installed with Node)

The standard `npm` package management tool can be used to install BuckleScript. If you don't already have `npm` installed, follow the directions listed here. Once `npm` is installed, run the following command:

```
npm install --save bs-platform
```

or install it globally

```
npm install -g bs-platform
```

Install from source with npm package manager

Prerequisites:

1. Standard C toolchain
2. `npm` (should be installed with Node)

Instructions:

```
git clone https://github.com/bloomberg/bucklescript
cd bucklescript
npm install
```

Install with *minimal* dependencies

Prerequisites:

1. Standard C toolchain

BuckleScript has very few dependencies and building from source can easily be done.

Build OCaml compiler

```
git clone --recursive https://github.com/bloomberg/bucklescript
cd bucklescript/ocaml
./configure -prefix `pwd` # put your preferred directory
make world.opt
make install
```

The patched compiler is installed locally into your `$(pwd)/bin` directory. To start using it temporarily, check if `ocamlc.opt` and `ocamlopt.opt` exist in `$(pwd)/bin`, and temporarily add the location to your `$(PATH)` (e.g. `PATH=$(pwd)/bin:$PATH`).

Building BuckleScript

The following directions assume you already have the correct version of `ocamlopt.opt` in your `$PATH`, having followed the process described in the previous section.

```
export BS_RELEASE_BUILD=1
make world
```

At the end, you should have a binary called `bsc.exe` under `jscomp/bin` directory, which you can add to your `$PATH`. You could also set an environment variable pointing to the stdlib, e.g. `BSC_LIB=/path/to/jscomp/stdlib` for ease of use.

> **WARNING** The built compiler is not *relocatable* out of box, please don't move it around unless you know what you are doing

Introduction to OCaml ecosystem: OPAM

When working with OCaml we also recommend using opam package manager to install OCaml tools, available here. You will benefit from the existing OCaml ecosystem.

Once you have `opam` installed, ask `opam` to switch to using our version of the compiler:

```
opam update
opam switch 4.02.3
eval `opam config env`
```

Get Started

First example

- Create a directory called `hello` and create `package.json` as below:

  package.json

  ```
  {
      "dependencies": {
          "bs-platform": "1.0.1" (1)
      },
      "scripts" : {
          "build" : "bsc.exe -c  main_entry.ml"
      }
  }
  ```

  1. Version should be updated accordingly

- Create `main_entry.ml` as below:

main_entry.ml

```
let () =
  print_endline "hello world"
```

- Build the app

```
npm run build
```

Now you should see a file called `main_entry.js` generated as below:

main_entry.js

```
// GENERATED CODE BY BUCKLESCRIPT VERSION 1.0.1 , PLEASE EDIT WITH CARE
'use strict';


console.log("hello world");

/*  Not a pure module */ (1)
```

1. The compiler analyze this module is impure due to the side effect

> **TIP** The working code is available here:

An example with multiple modules

Now we want to create two modules, one file called `fib.ml` which exports `fib` function, the other module called `main_entry.ml` which will call `fib`.

- Create a directory `fib` and created a file `package.json`

  package.json

```
{
    "dependencies": {
        "bs-platform": "1.0.1"
    },
    "scripts" : {
        "build" : "bsc.exe -c -bs-main main_entry.ml" (1)
    }
}
```

  1. here `-bs-main` option tells the compiler compile `main_entry` module and its dependency accordingly

- Create file `fib.ml` and file `main_entry.ml`

fib.ml

```
let  fib n  =
  let rec aux n a b =
    if n = 0 then a
    else
      aux (n - 1) b (a+b)
  in aux n 1 1
```

main_entry.ml

```
let () =
  for i = 0 to 10 do
    Js.log (Fib.fib i) (1)
  done
```

1. `Js` module is a built-in module shipped with BuckleScript

- Build the app

```
npm install
npm run build
node main_entry.js
```

If everything goes well, you should see the output as below:

```
1
1
2
3
5
8
13
21
34
55
89
```

Built in npm support

Build OCaml library as a npm package

BuckleScript extends the OCaml compiler options with several flags to provide better experience for NPM users.

In general, you are expected to deploy two kinds of artifacts, the generated JS files

and meta data which your OCaml dependencies rely on.

Since CommonJS has no namespaces, to allow JS files live in different directories, we have a flag

```
bsc.exe -bs-package-name $npm_package_name -bs-package-output
modulesystem:path/to/your/js/dir -c a.ml
```

By passing this flag, `bsc.exe` will store your `package_name` and relative path to `package.json` in `.cmj` files. It will also generate JS files in the directory you specified. You can, and are encouraged to, store Javascript files in a hierarchical directory.

For the binary artifacts, (Note that this is not necessary if you only want your libraries to be consumed by JS developers, and it has benefit since end users don't need these binary data any more), the convention is to store all `*.cm` data in a *single* directory `package.json/lib/ocaml` and Javascript files in a *hierachical* directory like `package.json/lib/js`

To use OCaml library as a npm package

If you follow the layout convention above, using an OCaml package is pretty straightforward:

```
bsc.exe -bs-package-include ocaml-package-name -c a.ml
```

Together

Your command line would be like this:

```
bsc.exe -bs-package-include ocaml-package1 -bs-package-include ocaml-package2  -
bs-package-name $npm_package_name -bs-package-output commonjs:path/to/lib/js/ -c
a.ml
```

Examples

Please visit https://github.com/bloomberg/bucklescript-addons for more examples.

JS Calling OCaml

Since BuckleScript guarantees that all OCaml functions are exported as is, no extra work is required to expose OCaml function to JavaScript.

CAUTION

- `external` exports are not exported as JS functions, if you really want to export those external functions, please write `val` instead
- `operators` are escaped, since Javascript does not support user defined operators. For example, instead of calling `Pervasives.(^)`, you have to call `Pervasives.$caret` from your Javascript functions

If users want to consume some OCaml features only available in OCaml but not in JS, we recommend users to export it as functions.

For example, data constructors are not available in JS

```
type t =
  | Cons of int * t
  | Nil
```

Currently, we recommend user to expose the constructor as a function so that it can be constructed from the JS side.

```
let cons x y = Cons (x,y)
let nil = Nil
```

NOTE

In the future, we will derive these functions to automate such process

---

OCaml calling JS

To make OCaml work smoothly with Javascript, we introduced several extensions to the OCaml language. These BuckleScript extensions facilitate the integration of native JavaScript code and improve the generated code.

Like TypeScript, when building type-safe bindings from JS to OCaml, users have to

write type declarations. In OCaml, unlike TypeScript, users do not need to create a separate `.d.ts` file, since the type declarations is an integral part of OCaml.

The FFI is divided into several components:

- Binding to simple functions and values
- Binding to high-order functions
- Binding to object literals
- Binding to classes
- Extensions to the language for debugger, regex and embedding arbitrary JS code

Binding to simple JS functions values

This part is similar to [traditional FFI], with syntax as described below:

```
external value-name :  typexpr =  external-declaration  attributes
external-declaration :=  string-literal
```

Users need to declare types for foreign functions (JS functions for BuckleScript or C functions for native compiler) and provide customized `attributes`.

Binding to global value: bs.val

```
external imul : int -> int -> int = "Math.imul" [@@bs.val]
type dom
(* Abstract type for the DOM *)
external dom : dom = "document" [@@bs.val]
```

`bs.val` attribute is used to bind to a JavaScript value, it can be a function or plain value.

NOTE

- If `external-declaration` is the same as `value-name`, user can leave `external-declaration` empty, for example:

  ```
  external document : dom = "" [@@bs.val]
  ```

- If you want to make a single FFI for both C functions and JavaScript functions, you can give the JavaScript foreign

function a different name:

```
external imul : int -> int -> int =
  "c_imul" [@@bs.val "Math.imul"]
```

Binding to JavaScript constructor: bs.new

`bs.new` is used to create a JavaScript object.

```
external create_date : unit -> t = "Date" [@@bs.new]
let date = create_date ()
```

Output:

```
var date = new Date();
```

Binding to a value from a module: bs.module

Input:

```
external add : int -> int -> int = "add" [@@bs.module "x"]
external add2 : int -> int -> int = "add2"[@@bs.module "y", "U"] (1)
let f = add 3 4
let g = add2 3 4
```

1. "U" will hint the compiler to generate a better name for the module, see output

Output:

```
var U = require("y");
var X = require("x");
var f = X.add(3, 4);
var g = U.add2(3, 4);
```

NOTE

- if `external-declaration` is the same as value-name, it can be left empty, for example,

```
external add : int -> int -> int = "" [@@bs.module "x"]
```

Binding the whole module as a value or function

```
type http
external http : http = "http" [@@bs.module] (1)
```

1. `external-declaration` is the module name

NOTE

- if `external-declaration` is the same as value-name, it can be left empty, for example,

  ```
  external http : http = "" [@@bs.module]
  ```

Binding to method: bs.send, bs.send.pipe

`bs.send` helps the user send a message to a JS object

```
type id (** Abstract type for id object *)
external get_by_id : dom -> string -> id =
  "getElementById" [@@bs.send]
```

The object is always the first argument and actual arguments follow.

Input:

```
get_by_id dom "xx"
```

Output:

```
dom.getElementById("xx")
```

`bs.pipe.send` is similar to `bs.send` except that the first argument, i.e, the object, is put in the position of last argument to help user write in a *chaining style*:

```
external map : ('a -> 'b [@bs]) -> 'b array =
  "" [@@bs.send.pipe: 'a array] (1)
external forEach: ('a -> unit [@bs]) -> 'a array =
  "" [@@bs.send.pipe: 'a array]
let test arr =
    arr
    |> map (fun [@bs] x -> x + 1)
    |> forEach (fun [@bs] x -> Js.log x)
```

1. For the `[@bs]` attribute in the callback, see Binding to callbacks (high-order

1. For the `[@bs]` attribute in the callback, see Binding to callbacks (high-order function)

NOTE

- if `external-declaration` is the same as value-name, it can be left empty, for example,

```
external getElementById : dom -> string -> id =
  "" [@@bs.send]
```

Binding to dynamic key access/set: bs.set_index, bs.get_index

This attribute allows dynamic access to a JavaScript property

```
type t
external create : int -> t = "Int32Array" [@@bs.new]
external get : t -> int -> int = "" [@@bs.get_index]
external set : t -> int -> int -> unit = "" [@@bs.set_index]
```

Binding to Getter/Setter: bs.get, bs.set

This attribute helps get and set the property of a JavaScript object.

```
type textarea
external set_name : textarea -> string -> unit = "name" [@@bs.set]
external get_name : textarea -> string = "name" [@@bs.get]
```

Splice calling convention: bs.splice

In JS, it is quite common to have a function take variadic arguments, BuckleScript support typing homogeneous variadic arguments. For example,

```
external join : string array -> string = "" [@@bs.module "path"] [@@bs.splice]
let v = join [| "a" "b"|]
```

Output

```
var Path = require("path")
var v = Path.join("a","b")
```

NOTE

For the external call, if the `array` arguments is not a compile time array, the compiler will emit an error message

Special types on external declarations: bs.string, bs.int, bs.ignore

Using polymorphic variant to model enums and string types

There are several patterns heavily used in existing JavaScript codebase, for example, string type is used a lot. BuckleScript FFI allows to model string type in a safe way by using annotated polymorphic variant.

```
external readFileSync :
  name:string ->
  ([ `utf8
   | `my_name [@bs.as "ascii"] (1)
   ] [@bs.string]) ->
  string = ""
  [@@bs.module "fs"]

let _ =
  readFileSync ~name:"xx.txt" `my_name
```

1. Here we intentionally made an example to show how to customize a name

Ouptut:

```
var Fs = require("fs");
Fs.readFileSync("xx.txt", "ascii");
```

Polymoprhic variants can also be used to model *enums*.

```
external test_int_type :
  ([ `on_closed (1)
   | `on_open [@bs.as 3]  (2)
   | `in_bin (3)
   ]
   [@bs.int])  -> int  =
  "" [@@bs.val]
```

1. `*on_closed* will be encoded as 0
2. `*on_open* will be 3 due to the attribute `bs.as`
3. `*in_bin* will be 4

Using polymorphic variant to model event listener

BuckleScript model this in a type-safe way by using annotated polymorphic variants

```
type readline
external on :
    (
    [ `close of unit -> unit
    | `line of string -> unit
    ] (1)
    [@bs.string])
    -> readline = "" [@@bs.send.pipe: readline]
let register rl =
  rl
  |> on (`close (fun event -> () ))
  |> on (`line (fun line -> print_endline line))
```

1. This is a very powerful typing: each event can have its *different types*

Output:

```
function register(rl) {
  return rl.on("close", function () {
                return /* () */0;
              })
            .on("line", function (line) {
              console.log(line);
              return /* () */0;
            });
}
```

WARNING

- These annotations will only have effect in `external` declarations.
- The runtime encoding of using polymorphic variant is internal to the compiler.
- With these annotations mentioned above, BuckleScript will automatically transform the internal encoding to the designated encoding for FFI. BuckleScript will try to do such conversion at compile time if it can, otherwise, it will do such conversion in the runtime, but it should be always correct.

Phantom Arguments and ad-hoc polyrmophism

`bs.ignore` allows arguments to be erased after passing to JS functional call, the side effect will still be recorded.

For example,

```
external add : (int [@bs.ignore]) -> int -> int = ""
[@@bs.val]
let v = add 0 1 2 (1)
```

1. the first argument will be erased

Output:

```
var v = add (1,2)
```

This is very useful to combine GADT:

```
type _ kind =
  | Float : float kind
  | String : string kind
external add : ('a kind [@bs.ignore]) -> 'a -> 'a -> 'a = "" [@@bs.val]

let () =
  Js.log (add Float 3.0 2.0);
  Js.log (add String "x" "y");
```

User can also have a payload for the GADT:

```
let string_of_kind (type t) (kind : t kind) =
  match kind with
  | Float -> "float"
  | String -> "string"

external add_dyn : ('a kind [@bs.ignore]) -> string ->  'a -> 'a -> 'a = ""
[@@bs.val]

let add2 k x y =
  add_dyn k (string_of_kind k) x y
```

Binding to NodeJS special variables: bs.node

NodeJS has several file local variables: `dirname`, `filename`, `module_`, and `require`, their semantics are more like macros instead of functions.

BuckleScript provides built-in macro support for these variables:

```
let dirname : string Js.undefined = [%bs.node __dirname]
let filename : string Js.undefined = [%bs.node __filename]
let module_ : Node.node_module Js.undefined = [%bs.node module_]
let require : Node.node_require Js.undefined = [%bs.node require]
```

Binding to callbacks (high-order function)

High order functions are functions where the callback can be another function. For example, suppose JS has a map function as below:

```
function map (a, b, f){
  var i = Math.min(a.length, b.length);
  var c = new Array(i);
  for(var j = 0; j < i; ++j){
    c[j] = f(a[i],b[i])
  }
  return c ;
}
```

A naive external type declaration would be as below:

```
external map : 'a array -> 'b array -> ('a -> 'b -> 'c) -> 'c array = ""
[@@bs.val]
```

Unfortunately, this is not completely correct. The issue is by reading the type `'a → 'b → 'c`, it can be in several cases:

```
let f x y = x + y
```

```
let g x  = let z  = x + 1 in fun y -> x + z
```

In OCaml, they all have the same type; however, `f` and `g` may be compiled into functions with different arities.

A naive compilation will compile `f` as below:

```
let f = fun x -> fun y -> x + y
```

```
function f(x){
  return function (y){
    return x + y;
  }
}
function g(x){
  var z = x + 1 ;
  return function (y){
    return x + z ;
  }
```

```
}
```

Its arity will be *consistent* but is *1* (returning another function); however, we expect *its arity to be 2*.

Bucklescript uses a more complex compilation strategy, compiling `f` as

```
function f(x,y){
  return x + y ;
}
```

No matter which strategy we use, existing typing rules **cannot guarantee a function of type `'a → 'b → 'c` will have arity 2.**

To solve this problem introduced by OCaml's curried calling convention, we support a special attribute `[@bs]` at the type level.

```
external map : 'a array -> 'b array -> ('a -> 'b -> 'c [@bs]) -> 'c array
= "map" [@@bs.val]
```

Here `('a → 'b → 'c [@bs])` will *always be of arity 2*, in general, `'a0 → 'a1 …'aN → 'b0 [@bs]` is the same as `'a0 → 'a1 …'aN → 'b0` except the former's arity is guaranteed to be `N` while the latter is unknown.

To produce a function of type `'a0 → .. 'aN → 'b0 [@bs]`, as follows:

```
let f : 'a0 -> 'a1 -> .. 'b0 [@bs] =
  fun [@bs] a0 a1 .. aN -> b0
let b : 'b0 = f a0 a1 a2 .. aN [@bs]
```

A special case for arity of 0:

```
let f : unit -> 'b0 [@bs] = fun [@bs] () -> b0
let b : 'b0 = f () [@bs]
```

Note that this extension to the OCaml language is *sound*. If you add an attribute in one place but miss it in other place, the type checker will complain.

Another more complex example:

```
type 'a return = int -> 'a [@bs]
type 'a u0 = int -> string -> 'a return  [@bs] (1)
type 'a u1 = int -> string -> int -> 'a [@bs] (2)
type 'a u2 = int -> string -> (int -> 'a [@bs]) [@bs] (3)
```

1. `u0` has arity of 2, return a function with arity 1
2. `u1` has arity of 3
3. `u2` has arity of 2, reutrn a function with arity 1

Uncurried calling convention as an optimization

Background:

As we discussed before, we can compile any OCaml function as arity 1 to support OCaml's curried calling convention.

This model is simple and easy to implement, but the native compilation is very slow and expensive for all functions.

```
let f x y z = x + y + z
let a = f 1 2 3
let b = f 1 2
```

can be compiled as

```
function f(x){
  return function (y){
    return function (z){
      return x + y + z
    }
  }
}
var a = f (1) (2) (3)
var b = f (1) (2)
```

But as you can see, this is *highly inefficient*, since the compiler already *saw the source definition* of `f`, it can be optimized as below:

```
function f(x,y,z) {return x + y + z}
var a = f(1,2,3)
var b = function(z){return f(1,2,z)}
```

BuckleScript does this optimization in the cross module level and tries to infer the arity as much as it can.

Callback optimization

However, such optimization will not work with *high-order* functions, i.e, callbacks.

For example,

```
let app f x = f x
```

Since the arity of `f` is unknown, the compiler can not do any optimization (unless `app` gets inlined), so we have to generate code as below:

```
function app(f,x){
  return Curry._1(f,x);
}
```

`Curry._1` is a function to dynamically support the curried calling convention.

Since we support the uncurried calling convention, you can write `app` as below

```
let app f x = f x [@bs]
```

Now the type system will infer `app` as type `('a →'b [@bs]) → 'a` and compile `app` as

```
function app(f,x){
  return f(x)
}
```

> **NOTE**
>
> In OCaml the compiler internally uncurries every function declared as `external` and guarantees that it is always fully applied. Therefore, for `external` first-order FFI, its outermost function does not need the `[@bs]` annotation.

Bindings to `this` based callbacks: bs.this

Many JS libraries have callbacks which rely on `this` (the source), for example:

```
x.onload = function(v){
  console.log(this.response + v )
}
```

Here, `this` would be the same as `x` (actually depends on how `onload` is called). It is clear that it is not correct to declare `x.onload` of type `unit -> unit [@bs]`. Instead, we introduced a special attribute `bs.this` allowing us to type `x` as below:

```
type x
external set_onload : x -> (x -> int -> unit [@bs.this]) -> unit = "onload"
[@@bs.set]
external resp : x -> int = "response" [@@bs.get]
set_onload x begin fun [@bs.this] o v ->
  Js.log(resp o + v )
end
```

Output:

```
x.onload = function(v){
  var o = this ; (1)
  console.log(o.response + v);
}
```

1. The first argument is automatically bound to `this`

`bs.this` is the same as `bs` : except that its first parameter is reserved for `this` and for arity of 0, there is no need for a redundant `unit` type:

```
let f : 'obj -> 'b [@bs.this] =
  fun [@bs.this] obj -> ....
let f1 : 'obj -> 'a0 -> 'b [@bs.this] =
  fun [@bs.this] obj a -> ...
```

NOTE

There is no way to consume a function of type `'obj -> 'a0 .. -> 'aN -> 'b0 [@bs.this]` on the OCaml side. This is an intentional design choice, we **don't encourage** people to write code in this style.

This was introduced mainly to be consumed by existing JS libraries. User can also type `x` as a JS class too (see later)

Binding to JS objects

Convention:

All JS objects of type `'a` are lifted to type `'a Js.t` to avoid conflict with OCaml's native object system (we support both OCaml's native object system and FFI to JS's objects), `##` is used in JS's object method dispatch and field access, while `#` is used in OCaml's object method dispatch.

Typing JavaScript objects:

OCaml supports object oriented style natively and provides structural type system. OCaml's object system has different runtime semantics from JS object, but they share the same type system, all JS objects of type `'a` is typed as `'a Js.t`

OCaml provide two kinds of syntaxes to mode structural typing: `< p1 : t1 >` style and `class type` style, they are mostly the same except that the latter is more feature rich (support inheritance) but more verbose.

Simple object type

Suppose we have a JS file `demo.js` which exports two properties: `height` and `width`:

demo.js

```
exports.height = 3
exports.width  = 3
```

There are different ways to writing binding to module `demo`, here we use OCaml objects to model module `demo`

```
external demo : < height : int ; width : int > Js.t = "" [@@bs.module]
```

There are too kinds of types on the method name:

- normal type

  ```
  < label : int >
  < label : int -> int >
  < label : int -> int [@bs]>
  < label : int -> int [@bs.this]>
  ```

- method

  ```
  < label : int -> int [@bs.meth] >
  ```

The difference is that for `method`, the type system will force users to full-fil its arguments all at the same time, since its semantics depends on `this` in JavaScript.

For example:

```
let test f =
  f##hi 1 (1)
let test2 f   =
  let u = f##hi in
  u 1
let test3 f =
  let u = f##hi in
  u 1 [@bs]
```

1. `##` is JS object property/method dispatch

The compiler would infer types differently

```
val test : < hi : int -> 'a [@bs.meth]; .. > -> 'a (1)
val test2 : < hi : int -> 'a ; .. > -> 'a
val test3 : < hi : int -> 'a [@bs]; .. >
```

1. `..` is a row variable, which means the object can contain more methods

Complex object type

Below is an example:

```
class type _rect = object
  method height : int
  method width : int
  method draw : unit -> unit
end [@bs] (1)
type rect = _rect Js.t
```

1. `class type` annotated with `[@bs]` is treated as a JS class type, it needs to be lifted to `Js.t` too

For JS classes, methods with arrow types are treated as real methods (automatically annotated with `[@bs.meth]`) while methods with non-arrow types are treated as properties.

So the type `rect` is the same as below:

```
type rect = < height : int ; wdith : int ; draw : unit -> unit [@bs.meth] > Js.t
```

How to consume JS property and methods

As we said: `##` is used in both object method dispatch and field access.

```
f##property (1)
f##property#= v
f##js_method args0 args1 args2 (2)
```

1. property get should not come with any argument as we discussed above, which will be checked by the compiler
2. Here `method` is of arity 3

> NOTE
>
> All JS method application is uncurried, JS's**method is not a function**, this invariant can be guaranteed by OCaml's type checker, a classic example shown below:
>
> ```
> console.log('fine')
> var log = console.log;
> log('fine') (1)
> ```
>
> 1. May cause exception, implementation dependent, `console.log` may depend on `this`

In BuckleScript

```
let fn = f0##f in
let a = fn 1 2
(* f##field a b would think `field` as a method *)
```

is different from

```
let b = f1##f 1 2
```

The compiler will infer as below:

```
val f0 : < f : int -> int -> int  > Js.t
val f1 : < f : int -> int -> int [@bs.meth] > Js.t
```

If we type `console` properly in OCaml, user could only write

```
console##log "fine"
let u = console##log
let () = u "fine" (1)
```

1. OCaml compiler will complain

> **NOTE** If a user were to make such a mistake, the type checker would complain by saying it expected `Js.method` but saw a function instead, so it is still sound and type safe.

getter/setter annotation to JS properties

Since OCaml's object system does not have getters/setters, we introduced two attributes `bs.get` and `bs.set` to help inform BuckleScript to compile them as property getters/setters.

```
type y  = <
 height : int [@@bs.set {no_get}] (1)
> Js.t
type y0 = <
 height : int [@@bs.set] [@@bs.get {null}] (2)
> Js.t
type y1 = <
  height : int [@@bs.set] [@@bs.get {undefined}] (3)
> Js.t
type y2 = <
  height : int [@@bs.set] [@@bs.get {undefined; null}] (4)
> Js.t
type y3 = <
  height : int  [@@bs.get {undefined ; null}] (5)
> Js.t
```

1. `height` is setter only
2. getter return `int Js.null`
3. getter return `int Js.undefined`
4. getter return `int Js.null_undefined`
5. getter only, return `int Js.null_undefined`

> **NOTE** Getter/Setter also applies to class type label

Create JS objects using bs.obj

Not only can we create bindings to JS objects, but also we can create JS objects in a type safe way in OCaml side:

```
let u = [%bs.obj { x = { y = { z = 3}}} ] (1)
```

1. `bs.obj` extension is used to mark `{}` as JS objects

Output:

```
var u = { x : { y : { z : 3 }}}
```

The compiler would infer `u` as type

```
val u : < x :  < y : < z : int > Js.t >  Js.t > Js.t
```

To make it more symmetric, extension `bs.obj` can also be applied into the type level, so you can write

```
val u : [%bs.obj: < x : < y < z : int > > > ]
```

Users can also write expression and types together as below:

```
let u = [%bs.obj ( { x = { y = { z = 3 }}} : < x : < y : < z : int > > > ]
```

Objects in a collection also works:

```
let xs = [%bs.obj [| { x = 3 } ; {x = 3 } |] : < x : int  > array  ]
let ys = [%bs.obj [| { x = 3} : { x = 4 } |] ]
```

Output:

```
var xs = [ { x : 3 } , { x : 3 }]
var ys = [ { x : 3 },  {x : 4 } ]
```

Create JS objects using external

`bs.obj` can also be used as an attribute in external declarations, as below:

```
external make_config : hi:int -> lo:int -> unit -> t = "" [@@bs.obj]
let v = make_config ~hi:2 ~lo:3
```

Output:

```
var v = { hi:2, lo:3}
```

Option argument is also supported:

```
external make_config : hi:int -> ?lo:int -> unit -> t = "" [@@bs.obj] (1)
let u = make_config ~hi:3 ()
let v = make_config ~lo:2 ~hi:3  ()
```

1. In OCaml, the order of label does not matter, and the evaluation order of arguments is undefined. Since the order does not matter, to make sure the compiler realize all the arguments are full-filled (including optional arguments), it is common to have a `unit` type before the result

Output:

```
var u = {hi : 3}
var v = {hi : 3 , lo: 2}
```

Now, we can write JS style code in OCaml too (in a type safe way):

```
let u = [%bs.obj {
  x = { y = { z = 3 } };
  fn = fun [@bs] u v -> u + v (1)
  } ]
let h = u##x##y##z
let a = h##fn
let b = a 1 2 [@bs]
```

1. `fn` property is not method, it does not rely on `this`, we will show how to create JS method in OCaml later.

Output:

```
var u = { x : { y : {z : 3}}, fn : function (u,v) {return u + v}}
var h = u.x.y.z
var a = h.fn
var b = a(1,2)
```

NOTE

When the field is an uncurried function, a short-hand syntax `#@` is available:

```
let b x y h = h#@fn x y
```

```
function b (x,y,h){
  return h.fn(x,y)
}
```

The compiler will infer the type of `b` as

```
val b : 'a -> 'b ->  < fn :  'a -> 'b -> 'c [@bs] > Js.t -> 'c
```

Create JS objects with `this` semantics

The objects created above can not use `this` in the method, this is supported in BuckleScript too.

```
let v2  =
  let x = 3. in
  object (self) (1)
    method hi x  y = self##say x +. y
    method say x =  x *. self## x ()
    method x () = x
  end [@bs] (2)
```

1. `self` is bound to `this` in generated JS code
2. `[@bs]` marks `object .. end` as a JS object

Output:

```
var v2 = {
  hi: function (x, y) {
    var self = this ;
    return self.say(x) + y;
  },
  say: function (x) {
    var self = this ;
    return x * self.x();
  },
  x: function () {
    return 3;
```

```
  }
};
```

Compiler infer the type of `v2` as below:

```
val v2 : object
  method hi : float -> float -> float
  method say : float -> float
  method x : unit -> float
end [@bs]
```

Below is another example to consume JS object :

```
let f (u : rect) =
  (* the type annotation is un-necessary,
     but it gives better error message
  *)
   Js.log u##height ;
   Js.log u##width ;
   u##width #= 30;
   u##height #= 30;
   u##draw ()
```

Output:

```
function f(u){
  console.log(u.height);
  console.log(u.width);
  u.width = 30;
  u.height = 30;
  return u.draw()
}
```

Method chaining

```
f
##(meth0 ())
##(meth1 a)
##(meth2 a b)
```

Object label translation convention

In JS, it is quite common to have several types for a single method, to model this ad-hoc polymorphism, we introduced a small convention when translating object labels, this is useful as below

Ad-hoc polymorphism

```
f##draw_cat (x,y)
f##draw_dog (x,y)
```

OUTPUT:

```
f.draw(x,y) // f.draw in JS can accept different types
f.draw(x,y)
```

NOTE

Rules

1. If `_` apperas in the first char,
   - The first char `_` will be discarded
   - If there is `_` in the rest chars, chars after last `_` will be discarded
2. Else if there is `_` in the rest chars, chars after last `_` will be discarded

Embedding raw Javascript code

WARNING

This is not encouraged. The user should minimize and localize use cases of embedding raw Javascript code, however, sometimes it's necessary to get the job done.

Embedding raw JS code as an expression

```
let keys : t -> string array [@bs] = [%bs.raw "Object.keys" ]
let unsafe_lt : 'a -> 'a -> Js.boolean [@bs] = [%bs.raw{|function(x,y){return x <
y}|}]
```

We highly recommend writing type annotations for such unsafe code. It is unsafe to refer to external OCaml symbols in raw JS code.

Embedding raw JS code as statements

```
[%%bs.raw{|
  console.log ("hey");
|}]
```

Other examples:

```
let x  : string = [%bs.raw{|"\x01\x02"|}]
```

It will be compiled into:

```
var x = "\x01\x02"
```

Polyfill of `Math.imul`

```
  [%%bs.raw{|
  // Math.imul polyfill
  if (!Math.imul){
      Math.imul = function (..) {..}
   }
  |}]
```

WARNING

- So far we don't perform any sanity checks in the quoted text (syntax checking is a long-term goal).
- Users should not refer to symbols in OCaml code. It is not guaranteed that the order is correct.

Debugger support

We introduced the extension `bs.debugger`, for example:

```
  let f x y =
    [%bs.debugger];
    x + y
```

which will be compiled into:

```
  function f (x,y) {
     debugger; // JavaScript developer tools will set an breakpoint and stop here
     x + y;
  }
```

Regex support

We introduced `bs.re` for Javascript regex expression:

```
let f  = [%bs.re "/b/g"]
```

The compiler will infer `f` has type `Js.Re.t` and generate code as below

```
var f = /b/g
```

> NOTE
>
> `Js.Re.t` is an abstract type, we are working on providing bindings for it.

Examples

Below is a simple example for mocha library. For more examples, please visit https://github.com/bloomberg/bucklescript-addons

A simple example: binding to mocha unit test library

This is an example showing how to provide bindings to themochajs unit test framework.

```
external describe : string -> (unit -> unit [@bs]) -> unit = "" [@@bs.val]
external it : string -> (unit -> unit [@bs]) -> unit = "" [@@bs.val]
```

Since, `mochajs` is a test framework, we also need some assertion tests. We can also describe the bindings to `assert.deepEqual` from nodejs `assert` library:

```
external eq : 'a -> 'a -> unit = "deepEqual"  [@@bs.module "assert"]
```

On top of this we can write normal OCaml functions, for example:

```
let assert_equal = eq
let from_suites name suite  =
    describe name (fun [@bs] () ->
         List.iter (fun (name, code) -> it name code) suite
         )
```

The compiler would generate code as below:

```
 var Assert = require("assert");
 var List = require("bs-platform/lib/js/list");

function assert_equal(prim, prim$1) {
 return Assert.deepEqual(prim, prim$1);
 }
```

```
function from_suites(name, suite) {
 return describe(name, function () {
   return List.iter(function (param) {
    return it(param[0], param[1]);
      }, suite);
  });
 }
```

Js module

Js module is shipped with BuckleScript, both the namespace `Js` and `Node` are preserved.

Js Public types

```
type +'a t
(** Js object type *)
type + 'a null
(** nullable, value of this type can be either [null] or ['a]
    this type is the same as {!Js.Null.t}  *)
type + 'a undefined
(** value of this type can be either [undefined] or ['a]
    this type is the same as {!Js.Undefined.t}  *)
type + 'a null_undefined
(** value of this type can be [undefined], [null] or ['a]
    this type is the same as {!Js.Null_undefined.t}*)
type boolean
```

Js Nested modules

```
(** {3 nested modules}*)
module Null = Js_null
module Undefined = Js_undefined
module Null_undefined = Js_null_undefined
```

Note that `Null`, `Undefined` and `Null_undefined` have similar interfaces, for example:

Js.Null module

```
type + 'a t = 'a Js.null
external to_opt : 'a t -> 'a option = "js_from_nullable"
external return : 'a -> 'a t  = "%identity"
external test : 'a t -> bool = "js_is_nil"
external empty : 'a t = "null" [@@bs.val]
```

Js Utility functions

```
external to_bool : boolean -> bool = "js_boolean_to_bool"
(** convert Js boolean to OCaml bool *)
external typeof : 'a -> string = "js_typeof"
(** [typeof x] will be compiled as [typeof x] in JS *)
external log : 'a -> unit = "js_dump"
(** A convenience function to log *)

(** {4 operators }*)
external unsafe_lt : 'a -> 'a -> boolean = "js_unsafe_lt"
(**  [unsafe_lt a b] will be compiled as [a < b] *)
external unsafe_le : 'a -> 'a -> boolean = "js_unsafe_le"
(**  [unsafe_le a b] will be compiled as [a <= b] *)
external unsafe_gt : 'a -> 'a -> boolean = "js_unsafe_gt"
(**  [unsafe_gt a b] will be compiled as [a > b] *)
external unsafe_ge : 'a -> 'a -> boolean = "js_unsafe_ge"
(**  [unsafe_ge a b] will be compiled as [a >= b] *)
```

Js Predefined JS values

```
external true_ : boolean = "true" [@@bs.val]
external false_ : boolean = "false" [@@bs.val]
external null : 'a null = ""
[@@bs.val] (* The same as {!Js.Null.empty} will be compiled as [null]*)
external undefined : 'a undefined = ""
[@@bs.val] (* The same as  {!Js.Undefined.empty} will be compiled as [undefined]*)
```

Extended compiler options

BuckleScript inherits the command line arguments of the[OCaml compiler](). It also adds several flags:

-bs-main (single directory build)

```
bsc.exe -bs-main main.ml
```

`bsc.exe` will build module `Main` and all its dependencies, when it finishes, it will run `node main.js`.

```
bsc.exe -c -bs-main main.ml
```

The same as above, but will not run `node`.

-bs-files

So that you can do

```
bsc.exe -c -bs-files *.ml *.mli
```

The compiler will sort the order of input files before starting compilation.

BuckleScript supports two compilation mode, script mode and package mode, in package mode, you have to provide `package.json` on top and set the options `-bs-package-name`, `-bs-package-output`. In script mode, such flags are not needed

-bs-package-name

The project name of your project, user is suggested to make it consistent with the `name` field in `package.json`

-bs-packge-output

The format is `module_system:oupt/path/relative/to/package.json` Currently supported module systesms are: `commonjs`, `amdjs` and `goog:<namespace>`

For example, when you want to use the `goog` module system, you can do things like this:

```
bsc.exe -bs-package-name your_package -bs-package-output goog:lib/goog -c xx.ml
```

> **NOTE** User can supply multiple `-bs-package-output` at the same time.

For example:

```
bsc.exe -bs-package-name name -bs-package-output commonjs:lib/js -bs-package-
output goog:lib/goog -bs-package-output amdjs:lib/amdjs -c x.ml
```

It will generate `x.js` in `lib/js` as commonjs module, `lib/goog` as google module and `lib/amdjs` as amdjs module at the same time.

You would then need a bundler for the different module systems: `webpack` supports `commonjs` and `amdjs` while `google closure compiler` supports all.

-bs-gen-tds

Trigger the generation of TypeScript `.d.ts` files. `bsc.exe` has the ability to also emits `.d.ts` for better interaction with typescript. This is still experimental.

For more options, please see the documentation of `bsc.exe -help`.

-bs-no-warn-ffi-type

Turn off warnings on FFI type declarations

-bs-eval

Example

```
bsc.exe -dparsetree -drawlambda -bs-eval 'Js.log "hello"'
```

```
[ (1)
  structure_item (//toplevel//[1,0+0]..[1,0+14])
    Pstr_eval
    expression (//toplevel//[1,0+0]..[1,0+14])
      Pexp_apply
      expression (//toplevel//[1,0+0]..[1,0+6])
        Pexp_ident "Js.log" (//toplevel//[1,0+0]..[1,0+6])
      [
        <label> ""
          expression (//toplevel//[1,0+7]..[1,0+14])
            Pexp_constant Const_string("hello",None)
      ]
]
(2)
(setglobal Bs_internal_eval! (seq (js_dump "hello") (makeblock 0)))
// Generated by BUCKLESCRIPT VERSION 1.0.2 , PLEASE EDIT WITH CARE
'use strict';


console.log("hello");

/*  Not a pure module */
```

1. Output by flag -dparsetree
2. Output by flag -drawlambda

For this flag, it will not create any intermediate file, which is useful for learning or troubleshooting.

-bs-no-builtin-ppx-ml, -bs-no-builtin-ppx-mli

If users don't use any bs specific annotaions, user can explicitly turn it off. Another use case is that users can use `-ppx` explicitly as below:

```
bsc.exe -c -ppx bsppx.exe -bs-no-builtin-ppx-ml c.ml
```

Semantics difference from other backends

This is particularly important when porting an existing OCaml application to JavaScript.

Custom data type

In OCaml, the C FFI allows the user to define a custom data type and customize `caml_compare`, `caml_hash` behavior, etc. This is not available in our backend (since we have no C FFI).

Physical (in)equality

In general, Users should only use physical equality as an optimization technique, but not rely on its correctness, since it is tightly coupled with the runtime.

String char range

Currently, BuckleScript assumes that the char range is `0-255`. The user should be careful when they pass a JavaScript string to OCaml side. Note that we are working on a solution for this problem.

Weak map

OCaml's weak map is not available in BuckleScript. The weak pointer is replaced by a strict pointer.

Integers

OCaml has `int`, `int32`, `nativeint` and `int64` types. - Both `int32` and `int64` in BuckleScript have the exact same semantics as OCaml. - `int` in BuckleScript is the same as `int32` while in OCaml it's platform dependent. - `nativeint` is treated as JavaScript float, except for division. For example, `Nativeint.div a b` will be translated into `a /b | 0`.

> **WARNING** `Nativeint.shift_right_logical x 0` is different from `Int32.shift_right_local x 0`. The former is literally translated into `x >>> 0` (translated into an unsigned int), while

> the latter is `x | 0`.

Printf.printf

The `Printf.print` implementation in BuckleScript requires a newline (`\n`) to trigger the printing. This behavior is not consistent with the buffered behavior of native OCaml. The only potential problem we foresee is that if the program terminates with no newline character, the text will never be printed. # Obj module

We do our best to mimic the native compiler, but we have no guarantee and there are differences.

Hashtbl hash algorithm

BuckleScript uses the same algorithm as native OCaml but the output is different due to runtime representation of int/int64/int32 and float.

Marshall

Marshall module is not supported yet.

Sys.argv, Sys.max_array_length, Sys.max_string_length

Command line arguments are always empty, might be fixed in the near future. `Sys.max_array_length` and `Sys.max_string_length` will be the same as `max_int`, but it might be respected.

Unsupported IO primitives

Because of the JavaScript environment limitation, `Pervasives.stdin` is not supported but both `Pervasives.stdout` and `Pervasives.stderr` are.

Conditional compilation support - static if

It is quite common that people want to write code works with different versions of compilers and libraries.

People used to use preprocessors like C preprocessor for C family languages. In OCaml community there are several preprocessors: cppo, ocp-pp, camlp4 IFDEF macros, optcomp and ppx optcomp.

Instead of using a preprocessor, BuckleScript adds language level static if compilation to the language. It is less powerful than other preprocessors since it only support static if, no `#define`, `#undefine`, `#include`, but there are several advantages.

- It's very small (only around 500 LOC) and highly efficient. There is no added pass, everything is done in a **single pass**. It is easy to rebuild the pre-processor in a stand alone file, with no dependencies on compiler libs to back-port it to old OCaml compilers
- It's purely functional and type safe, easy to work with IDEs like merlin

Concrete syntax

```
static-if
| HASH-IF-BOL conditional-expression THEN (1)
   tokens
(HASH-ELIF-BOL conditional-expression THEN) *
(ELSE-BOL tokens)?
HASH-END-BOL

conditional-expression
| conditional-expression && conditional-expression
| conditional-expression || conditional-expression
| atom-predicate

atom-predicate
| atom operator atom
| defined UIDENT
| undefined UIDENT

operator
| (= | < | > | <= | >= | =~ )

atom
| UIDENT | INT | STRING | FLOAT
```

1. IF-BOL means `#IF` should be in the beginning of a line

Typing rules

- type of INT is `int`
- type of STRING is `string`
- type of FLOAT is `float`
- value of UIDENT comes from either built-in values (with documented types) or environment variable, if it is literally `true`, `false` then it is `bool`, else if it is

parsable by `int_of_string` then it is of type int, else if it is parsable by `float_of_string` then it is float, otherwise it would be string

- In `lhs operator rhs` , `lhs` and `rhs` are always the same type and return boolean. `=~` is a semantics version operator which requires both sides to be string

Evaluation rules are obvious. `=~` respect semantics version, for example, the underlying engine

```
semver Location.none "1.2.3" "~1.3.0" = false;;
semver Location.none "1.2.3" "^1.3.0" = true ;;
semver Location.none "1.2.3" ">1.3.0" = false ;;
semver Location.none "1.2.3" ">=1.3.0" = false ;;
semver Location.none "1.2.3" "<1.3.0" = true ;;
semver Location.none "1.2.3" "<=1.3.0" = true ;;
semver Location.none "1.2.3" "1.2.3" = true;;
```

Examples

lwt_unix.mli

```
type open_flag =
    Unix.open_flag =
  | O_RDONLY
  | O_WRONLY
  | O_RDWR
  | O_NONBLOCK
  | O_APPEND
  | O_CREAT
  | O_TRUNC
  | O_EXCL
  | O_NOCTTY
  | O_DSYNC
  | O_SYNC
  | O_RSYNC
#if OCAML_VERSION =~ ">=3.13" then
  | O_SHARE_DELETE
#end
#if OCAML_VERSION =~ ">=4.01" then
  | O_CLOEXEC
#end
```

Built in variables and custom variables

```
ocamlscript>bsc.exe -bs-D CUSTOM_A="ghsigh" -bs-list-conditionals
OCAML_PATCH "BS"
BS_VERSION "1.2.1"
OS_TYPE "Unix"
```

```
BS true
CUSTOM_A "ghsigh"
WORD_SIZE 64
OCAML_VERSION "4.02.3+BS"
BIG_ENDIAN false
```

Changes to command line options

For BuckleScript users, nothing needs to be done (it is baked in the language level), for non BuckleScript users, we provide an external pre-processor, so it will work with other OCaml compilers too. Note that the bspp.ml is a stand alone file, so that it even works without compilation.

Example

```
bsc.exe -c lwt_unix.mli
ocamlc -pp 'bspp.exe' -c lwt_unix.mli
ocamlc -pp 'ocaml -w -a bspp.ml' -c lwt_unix.mli
```

WARNING

This is a very small extension to the OCaml language, it is backward compatible with OCaml language with such exception.

```
let f x =
  x
#elif (1)
```

1. `#elif` at the beginning of a line is interpreted as static if, there is no issue with `#if` or `#end`, since they are already keywords

---

Build system support

The BuckleScript compilation model is similar to OCaml native compiler. If `b.ml` depends on `a.ml`, you have to compile `a.ml` **and** `a.mli` first.

NOTE

The technical reason is that BuckleScript will generate intermediate files with the extension `.cmj` which are later used for cross module inlining, arity inference and other information.

BuckleScript distribution has `bsdep.exe` which has the same interface as

`ocamldep`

Here is a simple Makefile to get started:

Makefile

```
OCAMLC=bsc.exe (1)
OCAMLDEP=bsdep.exe (2)
SOURCE_LIST := src_a src_b
SOURCE_MLI  = $(addsuffix .mli, $(SOURCE_LIST))
SOURCE_ML   = $(addsuffix .ml, $(SOURCE_LIST))
TARGETS := $(addsuffix .cmj, $(SOURCE_LIST))
INCLUDES=
all: $(TARGETS)
.mli:.cmi
        $(OCAMLC) $(INCLUDES) $(COMPFLAGS)  -c $<
.ml:.cmj:
        $(OCAMLC) $(INCLUDES) $(COMPFLAGS)  -c $<
-include .depend
depend:
        $(OCAMLDEP) $(INCLUDES) $(SOURCE_ML) $(SOURCE_MLI) > .depend
```

1. bsc.exe is the BuckleScript compiler
2. ocamldep executable is part of the OCaml compiler installation

In theory, people need run `make depend && make all`, `make depend` will calculate dependency while `make all` will do the job.

However, in practice, people used to use a file watch service, takewatchman for example, you need json configure

build.json

```
[
    "trigger", ".", {
        "name": "build",
        "expression": ["pcre", "(\\.(ml|mll|mly|mli|sh|sh)$|Makefile)"], (1)
        "command": ["./build.sh"],
        "append_files" : true
    }
]
```

1. whenever such files changed, it will trigger `command` field to be run

build.sh

```
make -r -j8 all (1)
make depend (2)
```

1. build
2. update the dependency

Now in your working directory, type `watchman -j < build.json` and enjoy the lightning build speed.

FAQ

1. *How does IO work in browser?*

   In general, it is very hard to simulate IO in browser, we recommend users to write bindings to NodeJS directly for server side, or use `Js.log` in client side, see disucssions in #748

2. *The compiler does not build?*

   In production mode, the compiler is a single file in `jscomp/bin/compiler.ml`. If it is not compiling, make sure you have the right OCaml compiler version. Currently the OCaml compiler is a submodule of BuckleScript. Make sure the exact commit hash matches (we only update the compiler occasionally).

3. *Which version of JavaScript syntax does BuckleScript target?*

   BuckleScript targets **ES5**.

4. *Does BuckleScript work with merlin?*

   Yes, you need edit your `.merlin` file:

   ```
   B node_modules/bs-platform/lib/ocaml
   S node_modules/bs-platform/lib/ocaml
   FLG -ppx node_modules/bs-platform/bin/bsppx.exe
   ```

   Note there is a upstream fix in Merlin, make sure your merlin is updated

5. *What polyfills does BuckleScript need?*

   - *Math.imul*: This polyfill is needed for `int32` multiplication. BuckleScript provides this by default(when feature detection returns false), no action is required from the user.
   - *TypedArray*: The TypedArray polyfill is not provided by BuckleScript and it's the responsibility of the user to bundle the desired polyfill implementation with the BuckleScript generated code.

```
The following functions from OCaml stdlib
require the TypedArray polyfill:
```

- Int64.float_of_bits
- Int64.bits_of_float
- Int32.float_of_bits
- Int32.bits_of_float

> **WARNING**
>
> For the current BuckleScript version, if the user does not bundle the TypedArray polyfill, the JavaScript engine does not support it and user used functions mentioned above, the code will fail at runtime.

High Level compiler workflow

The high level architecture is illustrated below:

```
Source Language
  |
  | (Reuse OCaml Parser)
  v
Surface Syntax Tree
  |
  | (built in Syntax tree transformation)
  v
Surface Syntax Tree
  |
  | (Reuse OCaml Type checker)
  v
Typedtree
  |
  | (Reuse OCaml pattern match compiler and erase types)
  |
  |
OCaml Lambda IR
  |
  |
  v
Buckle Lambda IR ------------------+
  |   ^                            |
  |   |                     Lambda Passes (lam_* files)
  |   |             Optimization/inlining/dead code elimination
  |   \                            |
  |    \ --------------------------+
```

```
  |
  |  Self tail call elimination
  |  Constant folding + propagation
  V
JS IR (J.ml)  ---------------------+
  |   ^                            |
  |   |                      JS Passes (js_* files)
  |   |            Optimization/inlining/dead code elimination
  |   \                            |
  |    \  --------------------------+
  |
  |  Smart printer includes scope analysis
  |
  V
Javascript Code
```

Design Principles

The current design of BuckleScript follows several high level principles. While those principles might change in the future, there are enforced today and can explain certain technical limitations BuckleScript has.

Lambda Representation

As pictured in the diagram above, BuckleScript is primarily based on the Lambda representation of the OCaml compiler. While this representation is quite rich, some information is lost from the upstream representation. The patch to the OCaml compiler tries to enrich this representation in a non-intrusive way (see next section).

Minimal Patch to the OCaml compiler

BuckleScript requires patches to the OCaml compiler. One of the main reasons is to enrich the Lambda representation so that the generated code is as nice as possible. A design goal is to keep those patches minimal and useful for the OCaml compiler in general so that they can later be integrated.

NOTE

> A common question is to wonder why BuckleScript transpiles an OCaml record value to a JavaScript array while a more intuitive representation would be a JavaScript object. This technical decision is a direct consequence of the above 2 design principles: the Lambda layer assumes in a lot of places that a record value is an array and such modification would be too large of a change to OCaml compiler.

Soundness

BuckleScript preserves the soundness of the OCaml language. Assuming the FFI is correctly implemented, the type safety is preserved.

Minimal new symbol creation

In order to make the JavaScript generated code as close as possible to the original OCaml core we thrive to introduce as few new symbols as possible.

Runtime representation

Below is a description of how OCaml values are encoded in JavaScript, the **internal** description means **users should not rely on its actual encoding (and it is subject to change)**. We recommend that you write setter/getter functions to manipulate safely OCaml values from JavaScript.

For example, users should not rely on how OCaml `list` is encoded in JavaScript; instead, the OCaml stdlib provides three functions: `List.cons`, `List.hd` and `List.tl`. JavaScript code should only rely on those three functions.

Simple OCaml type

ocaml type	**JavaScript type**
int	number
nativeint	number
int32	number
float	number
bool	number • true → 1 • false → 0
int64	Array of size two numbers `[hi,lo]`. `hi`

ocaml type	JavaScript type
char	number for example: • 'a' → `97`
string	string
bytes	number array NOTE We might encode it as buffer in NodeJS.
'a array	Array

record	Array **internal**
	
	For instance:
	
	`type t = { x : int; y : int }`
	`let v = {x = 1;  y = 2}`
	
	Output:
	
	`var v = [1,2]`

tuple	Array For example: • (3,4) → [3,4]
`'a option`	**internal** For example:

is signed while `lo` is unsigned

ocaml type	JavaScript type
	• None → 0 • `Some a` → `[a]`
list	**internal** For example: • `[]` → `0` • `x::y` → `[x,y]` • `1::2::[3]` → `[ 1, [ 2, [ 3, 0 ] ] ]`
Variant	**internal**
Polymorphic variant	**internal**
exception	**internal**
extension	**internal**
object	**internal**
`Js.boolean`	boolean For example: • Js.true_ → true • Js.false_ → false *Js module* `val Js.to_bool: Js.boolean -> bool`

`'a Js.Null.t`	either `'a` or `null`
	
	Js.Null module
	
	`val to_opt : 'a t -> 'a option`
	
	`val return : 'a -> 'a t`

ocaml type	
	`val test : 'a t -> bool`

`'a Js.Undefined.t`	either `'a` or `undefined`
	
	Js.Undefined
	
	`val to_opt : 'a t -> 'a option`
	`val return : 'a -> 'a t`
	`val test : 'a t -> bool`

`'a Js.Null_undefined.t`	either `'a`, `null` or `undef`

NOTE `Js.to_opt` is optimized when the `option` is not escaped

NOTE In the future, we will have a *debug* mode, in which the corresponding js encoding will be instrumented with more information

As we clarified before, the internal representation should not be relied upon. We are working to provide a ppx extension as below:

```
type t =
  | A
  | B of int * int
  | C of int * int
  | D [@@bs.deriving{export}]
```

So that it will a automatically provide `constructing` and `destructing` functions:

```
val a : t
val b : int -> int -> t
val c : int -> int -> t
val d : int

val a_of_t : t -> bool
val d_of_t : t -> bool
val b_of_t : t -> (int * int ) Js.Null.t
val c_of_t : t -> (int * int ) Js.Null.t
```

Integration with Reason

You can play with Reason using the playground [Facebook Reason]

> **NOTE** *The playgrounds are only for demos and might not be the latest*
> You should always use the command line as your production tool.

There is a stand alone example [here].

How to contribute

Build the compiler

The development of BuckleScript compiler relies on 2 tools which are readily available in `opam` and work with our patched OCaml compiler:

- [ocamlbuild]: Default build tool for OCaml project
- [camlp4]: Tool used to generate OCaml code for processing large AST. (j.ml file).

After having installed the above dependencies from opam you can run the following:

```
cd jscomp/
./build.sh
```

Build the runtime

```
cd ./runtime; make all
```

Build the stdlib

```
cd ./stdlib; make all
```

Help rewrite the whole runtime in OCaml

BuckleScript runtime implementation is currently a mix of OCaml and JavaScript. (`jscomp/runtime` directory). The JavaScript code is defined in the `.ml` file using

the `bs.raw` syntax extension.

The goal is to implement the runtime**purely in OCaml** and you can help contribute.

Each new PR should include appropriate testing.

Currently all tests are in `jscomp/test` directory and you should either add a new test file or modify an existing test which covers the part of the compiler you modified.

- Add the filename in `jscomp/test/test.mllib`
- Add a suite test

The specification is in `jscomp/test/mt.ml`

For example some simple tests would be like:

```
let suites : _ Mt.pair_suites =
   ["hey", (fun _ -> Eq(true, 3 > 2));
       "hi", (fun _ ->  Neq(2,3));
       "hello", (fun _ -> Approx(3.0, 3.0));
       "throw", (fun _ -> ThrowAny(fun _ -> raise 3))
       ]
let () = Mt.from_pair_suites __FILE__ suites
```

- Run the tests

Suppose you have mocha installed, if not, try `npm install mocha`

`mocha -R list jscomp/test/your_test_file.js`

- See the coverage

`npm run cover`

Comparisons

Difference from js_of_ocaml

Js_of_ocaml is a popular compiler which compiles OCaml's bytecode into

JavaScript. It is the inspiration for this project, and has already been under development for several years and is ready for production. In comparison, BuckleScript, while moving fast, is still a very young project. BuckleScript's motivation, like `js_of_ocaml`, is to unify the ubiquity of the JavaScript platform and the truly sophisticated type system of OCaml, however, there are some areas where we view things differently from `js_of_ocaml`. We describe below, some of these differences, and also refer readers to some of the original informal discussions.

- Js_of_ocaml takes lowlevel bytecode from OCaml compiler, BuckleScript takes the highlevel rawlambda representation from OCaml compiler
- Js_of_ocaml focuses more on existing OCaml eco-system(opam) while BuckleScript's major goal is to target npm
- Js_of_ocaml and BuckleScript have slightly different runtime encoding in several places, for example, BuckleScript encodes OCaml Array as JS Array while js_of_ocaml requires its index 0 to be of value 0.

Both projects are improving quickly, so this can change in the future!

Appendix A: CHANGES

1.2.1 + dev

- Features
  - add `-bs-D` `-bs-list-conditionals` flags #851
  - add `-bs-syntax-only`
  - add `-bs-binary-ast` #854

1.1.2

- Fixes
  - Bug fix with opam issues #831
- Features
  - Provide bspp.exe for official compiler

1.1.1

- Features
  - Add bsdep.exe #822
  - Conditional compilation support #820
  - Relax syntactic restrictions for all extension point #793 so that `bs.obj`, `obj`, `bs.raw`, `raw`, etc will both work. Note that all attributes will still be qualified
  - Suport bs.splice in bs.new #793
  - Complete `bs.splice ` support and documentation #798

1.03

- Features
  - Add an option `-bs-no-warn-unused-bs-attribute` #787
- Incompatible changes (due to proper Windows support):
  - `bsc`, `bspack` and `bsppx` are renamed into `bsc.exe`, `bspack.exe` and `bsppx.exe`
  - no symlink from .bin any more.

  Old symlinks

  ```
    tmp>ls -al node_modules/.bin/
    total 96
    drwxr-xr-x  14 hzhang295  staff  476 Sep 20 17:26 .
    drwxr-xr-x   4 hzhang295  staff  136 Sep 20 17:27 ..
    lrwxr-xr-x   1 hzhang295  staff   22 Sep 20 17:26 bsc -> ../bs-
  platform/bin/bsc
    lrwxr-xr-x   1 hzhang295  staff   25 Sep 20 17:26 bspack -> ../bs-
  platform/bin/bspack
    lrwxr-xr-x   1 hzhang295  staff   24 Sep 20 17:26 bsppx -> ../bs-
  platform/bin/bsppx
  ```

  Now these symlinks are removed, you have to refer to `bs-platform/bin/bsc.exe`

1.02

- Bug fixes and enhancement
  - Fix Bytes.blit when src==dst #743

- Features
  - Add an option `-bs-no-warn-ffi-type` #783 By default, `bsc.exe` will warn when it detect some ocaml datatype is passed from/to external FFi
  - Add an option `-bs-eval` 784

1.01

- FFI
  - support fields and mutable fields in JS object creation and private method #694
  - Introduce phantom arguments (`bs.ignore`) for ad-hoc polymorphism #686
- Bug fixes and enhancement
  - Enforce `#=` always return unit #718 for better error messages

1.0

Initial release

Version 1.2.1+dev