Pocaml: Poor Man’s OCaml

•••

Leo Qiao, Peter Choi, Yiming Fang, Yunlan Li
Advised by John Hui

Language Introduction & Demo

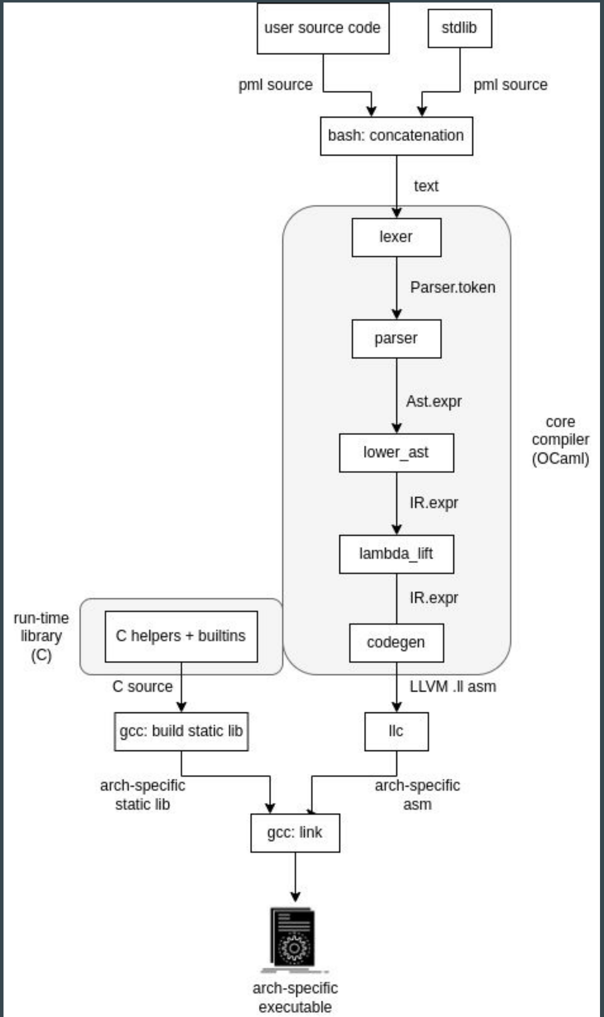
- "poor man's OCaml"

- Has main features of OCaml, such as higher-order functions, partial application, pattern matching, parametric polymorphism, and much of the same syntactic sugar.

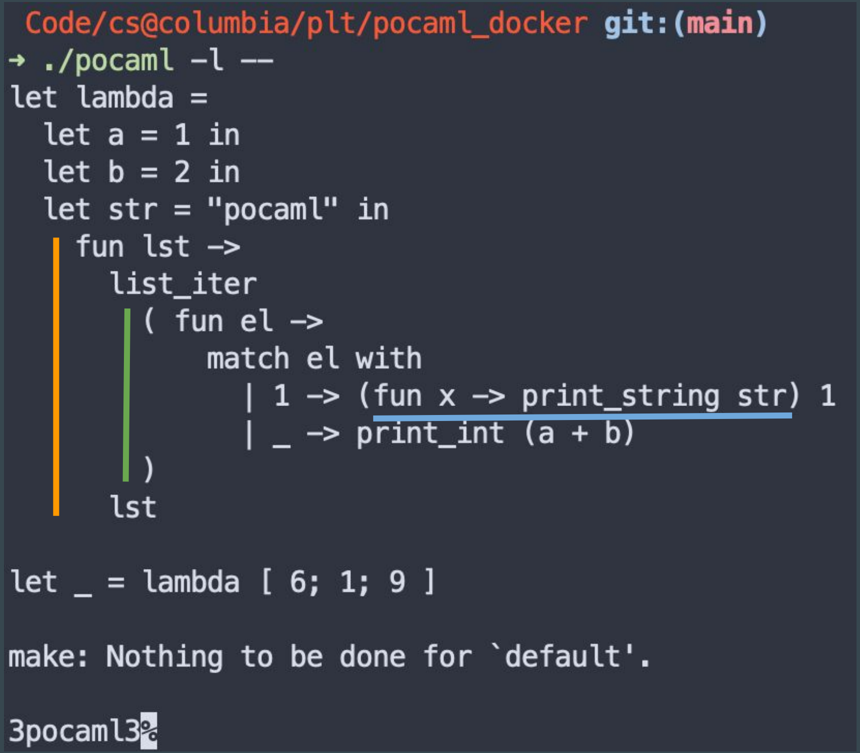
- Includes builtins and a standard library for common operations on lists and I/O.

PoCaml

Compiler Pipeline



Lambda Lifting Demo



```
 Code/cs@columbia/plt/pocaml_docker git:(main)
➜ ./pocaml -l --
let lambda =
  let a = 1 in
  let b = 2 in
  let str = "pocaml" in
    fun lst ->
      list_iter
        ( fun el ->
            match el with
              | 1 -> (fun x -> print_string str) 1
              | _ -> print_int (a + b)
        )
      lst

let _ = lambda [ 6; 1; 9 ]

make: Nothing to be done for `default'.

3pocaml3%
```

- produces the correct output “3pocaml3”
- demonstrates the correctness of lambda lifting in
  - let-in expression
  - applications
  - match arms
  - lambda

Lambda Lifting

- makes lambdas function properly in Pocaml
- happens after the lower_ast compiler pass
- rules: lift into top level functions all lambdas except
    - top-level lambdas:

      let a = fun b -> b

    - Immediately nested lambdas:

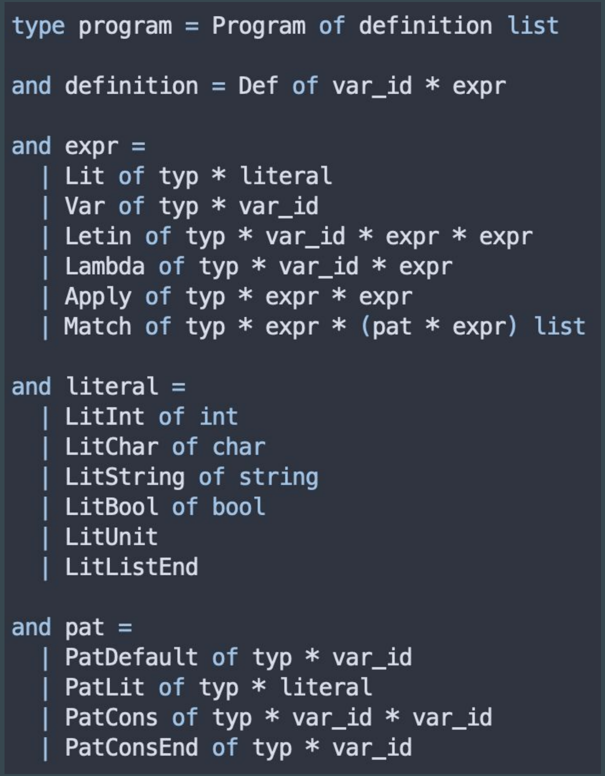
      let add3 = let a = 3 in fun x -> fun y -> x + y + a

- implementation
- example

```
let increment =
  let i = 1 in
  let j = 2 in
    fun x -> x + i * j
```

→

```
let lambda_1 = fun x -> ( fun i -> (  fun j -> x + i * j ) )

let increment =
  let i = 1 in
  let j = 2 in
    lambda_1  j  i
```

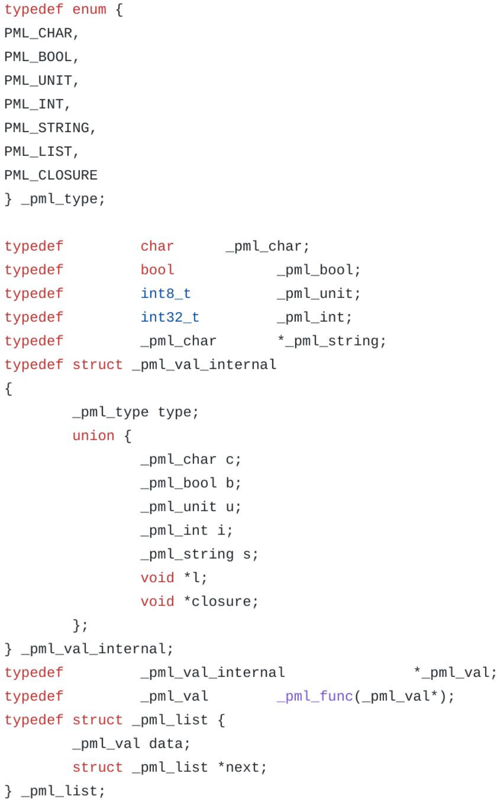


```
type program = Program of definition list

and definition = Def of var_id * expr

and expr =
    | Lit of typ * literal
    | Var of typ * var_id
    | Letin of typ * var_id * expr * expr
    | Lambda of typ * var_id * expr
    | Apply of typ * expr * expr
    | Match of typ * expr * (pat * expr) list

and literal =
    | LitInt of int
    | LitChar of char
    | LitString of string
    | LitBool of bool
    | LitUnit
    | LitListEnd

and pat =
    | PatDefault of typ * var_id
    | PatLit of typ * literal
    | PatCons of typ * var_id * var_id
    | PatConsEnd of typ * var_id
```

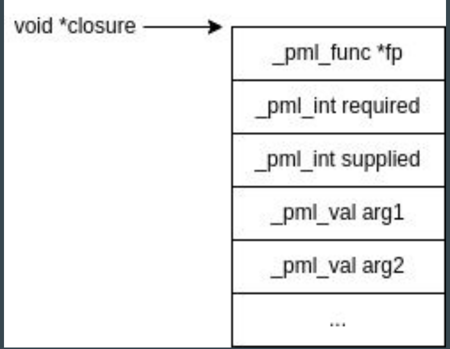
Fig: Reduced Abstract Syntax Tree after lower_ast

Codegen: run-time value representation

- _pml_val
  - pointer to _pml_val_internal
- _pml_val_internal
  - type information
    - added to support operator overloading
  - union of all Pocaml data types
- closure:
  - representation for lambda
  - run-time support for partial application
  - lambda creation and application are done with C run-time library:
    - `_pml_val _make_closure(_pml_func *fp, _pml_int num_args);`
    - `_pml_val _apply_closure(_pml_val closure, _pml_val arg);`
- uniform representation
  - parametric polymorphism



```
typedef enum {
PML_CHAR,
PML_BOOL,
PML_UNIT,
PML_INT,
PML_STRING,
PML_LIST,
PML_CLOSURE
} _pml_type;

typedef         char      _pml_char;
typedef         bool            _pml_bool;
typedef         int8_t          _pml_unit;
typedef         int32_t         _pml_int;
typedef         _pml_char       *_pml_string;
typedef struct _pml_val_internal
{
        _pml_type type;
        union {
                _pml_char c;
                _pml_bool b;
                _pml_unit u;
                _pml_int i;
                _pml_string s;
                void *l;
                void *closure;
        };
} _pml_val_internal;
typedef         _pml_val_internal               *_pml_val;
typedef         _pml_val        _pml_func(_pml_val*);
typedef struct _pml_list {
        _pml_val data;
        struct _pml_list *next;
} _pml_list;
```



void *closure

_pml_func *fp

_pml_int required

_pml_int supplied

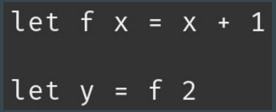
_pml_val arg1

_pml_val arg2

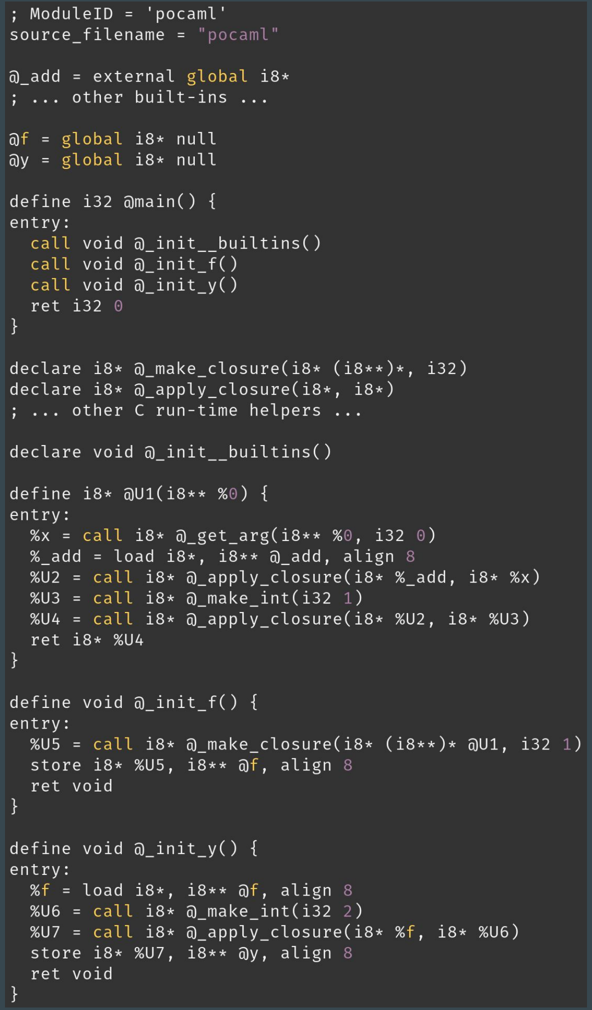
...

Codegen: program representation

- Pocaml: sequential evaluation of top-level definitions
- LLVM: evaluation of an entry main function
- solution:
  - top-level variable -> global variable
  - value evaluation -> _init_ functions
  - sequential evaluation -> call _init_ functions in main
- example:
  - generated LLVM with parts omitted
  - notice
    - lambda =\= function
    - lambda == closure
    - _init_f() stores the closure in @f



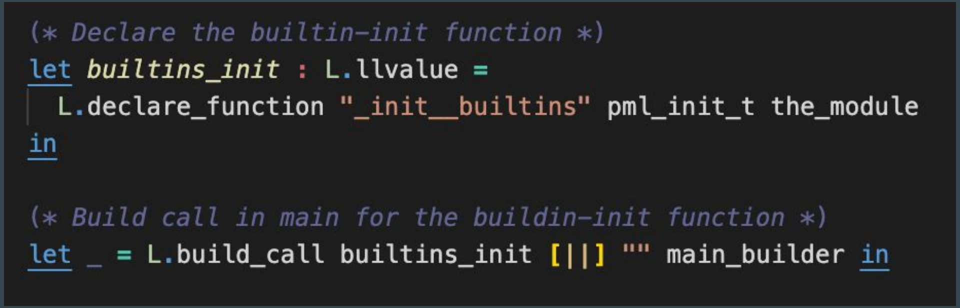
```
let f x = x + 1

let y = f 2
```



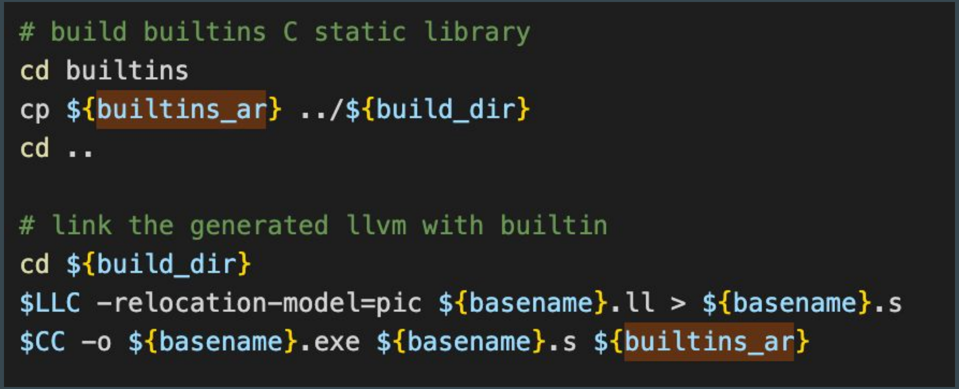
```
; ModuleID = 'pocaml'
source_filename = "pocaml"

@_add = external global i8*
; ... other built-ins ...

@f = global i8* null
@y = global i8* null

define i32 @main() {
entry:
  call void @_init__builtins()
  call void @_init_f()
  call void @_init_y()
  ret i32 0
}

declare i8* @_make_closure(i8* (i8**)*, i32)
declare i8* @_apply_closure(i8*, i8*)
; ... other C run-time helpers ...

declare void @_init__builtins()

define i8* @U1(i8** %0) {
entry:
  %x = call i8* @_get_arg(i8** %0, i32 0)
  %_add = load i8*, i8** @_add, align 8
  %U2 = call i8* @_apply_closure(i8* %_add, i8* %x)
  %U3 = call i8* @_make_int(i32 1)
  %U4 = call i8* @_apply_closure(i8* %U2, i8* %U3)
  ret i8* %U4
}

define void @_init_f() {
entry:
  %U5 = call i8* @_make_closure(i8* (i8**)* @U1, i32 1)
  store i8* %U5, i8** @f, align 8
  ret void
}

define void @_init_y() {
entry:
  %f = load i8*, i8** @f, align 8
  %U6 = call i8* @_make_int(i32 2)
  %U7 = call i8* @_apply_closure(i8* %f, i8* %U6)
  store i8* %U7, i8** @y, align 8
  ret void
}
```

C built-ins

- Built-ins functions exist in the form of closure.
- During codegen, the built-ins initializer, `_init__builtins`, is declared.
- The C code for built-in operators and functions is linked to the rest of the LLVM code so that it can be accessed.



```
(* Declare the builtin-init function *)
let builtins_init : L.llvalue =
  L.declare_function "_init__builtins" pml_init_t the_module
in

(* Build call in main for the buildin-init function *)
let _ = L.build_call builtins_init [||] "" main_builder in
```



```
# build builtins C static library
cd builtins
cp ${builtins_ar} ../${build_dir}
cd ..

# link the generated llvm with builtin
cd ${build_dir}
$LLC -relocation-model=pic ${basename}.ll > ${basename}.s
$CC -o ${basename}.exe ${basename}.s ${builtins_ar}
```

C built-ins

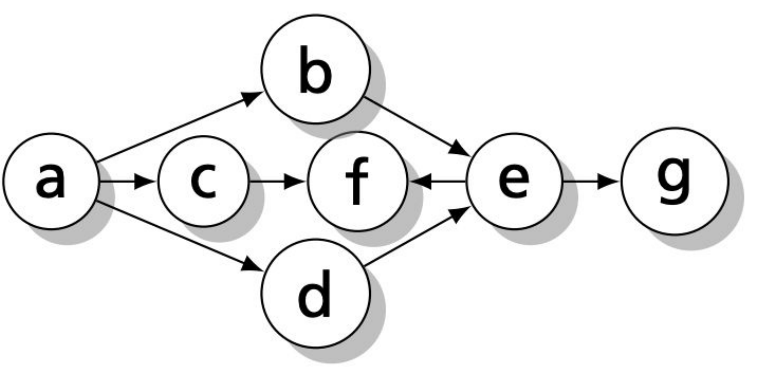
- One example of a function called by `_init__builtins` is `_init__add`.

- A closure containing the execution instructions is made public, created in the same as for a lambda expression and used in the same way during codegen.

```c
_pml_val _add;

_pml_val _builtin__add(_pml_val *args)
{
  _pml_val left, right;
  left = (_pml_val)args[0];
  right = (_pml_val)args[1];
  _pml_int res = _pml_get_int(left) + _pml_get_int(right);
  return _make_int(res);
}

void _init__add()
{
  _add = _make_closure(_builtin__add, 2);
}
```

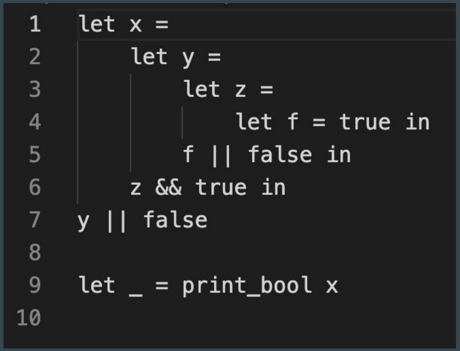
Standard Library

- List
  - length, hd, tl, append
  - iter, filter, map, mem
  - fold_left, fold_right
- I/O
  - print functions for all types
  - print functions for printing lists
  - to_string functions for all types
- example:
  - Implementing graph algorithms with stdlib
  - Demo

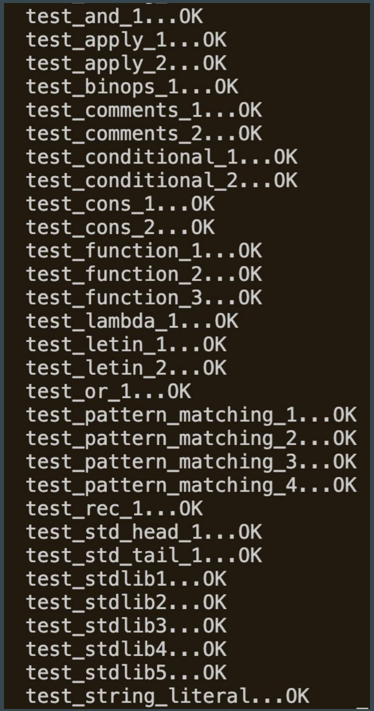


Automated Testing

- Unit Testing
  - Used during active development
  - Pretty printing for AST and IR
  - Utilized OCaml’s ppx_expect functionality to auto-generate expected value
- Integration Testing
  - Automatic shell script
  - MicroC-style
  - Checks the output/error against reference
  - Saves the execution details to log
- Test suites:
  - More than 50 test cases for integration test
  - Include both tests that should pass and should fail



```
let x =
    let y =
        let z =
            let f = true in
        f || false in
    z && true in
y || false

let _ = print_bool x

```



```
test_and_1...OK
test_apply_1...OK
test_apply_2...OK
test_binops_1...OK
test_comments_1...OK
test_comments_2...OK
test_conditional_1...OK
test_conditional_2...OK
test_cons_1...OK
test_cons_2...OK
test_function_1...OK
test_function_2...OK
test_function_3...OK
test_lambda_1...OK
test_letin_1...OK
test_letin_2...OK
test_or_1...OK
test_pattern_matching_1...OK
test_pattern_matching_2...OK
test_pattern_matching_3...OK
test_pattern_matching_4...OK
test_rec_1...OK
test_std_head_1...OK
test_std_tail_1...OK
test_stdlib1...OK
test_stdlib2...OK
test_stdlib3...OK
test_stdlib4...OK
test_stdlib5...OK
test_string_literal...OK
```

Conclusions and Lessons Learned

- "Be the compiler"
- The power of using the team to solve tough problems, rather than fighting alone
- Viewing programming languages from a more critical lens
- Clean code can be easily explained to others