The Trojan Poke

Embedding GNU poke in your own program

Mohammad-Reza Nabipoor
mnabipoor@gnu.org

GNU Project

This talk

- Introduction to `libpoke`

This talk

- Introduction to `libpoke`
- How to use `libpoke` in your programs

This talk

- Introduction to `libpoke`
- How to use `libpoke` in your programs
- Real-world examples

This talk

- Introduction to `libpoke`
- How to use `libpoke` in your programs
- Real-world examples
- Tips and best *known* practices of using `libpoke`

This talk

- Introduction to `libpoke`
- How to use `libpoke` in your programs
- Real-world examples
- Tips and best *known* practices of using `libpoke`
- Current known limitations and future work

What is GNU poke?

```
GNU poke = libpoke  #   Poke programming language
                    #     incremental compiler (PKL)
                    # &
                    #   Poke Virtual Machine (PVM)
                    # &
                    #   IO Space (IOS)
                    #
           + poke   #   CLI (Command-Line Interface)
                    #     based on libpoke
           ;
```

Poke programming language

- Procedural

Poke programming language

- Procedural
- Statically-typed

Poke programming language

- Procedural
- Statically-typed
- Interpreted

Poke programming language

- Procedural
- Statically-typed
- Interpreted
- Interactive (you can redeclare everything)

Poke programming language

- Procedural
- Statically-typed
- Interpreted
- Interactive (you can redeclare everything)
- Suitable for describing/poking binary formats

Poke Virtual Machine (PVM)

- Jitter-generated virtual machine

Poke Virtual Machine (PVM)

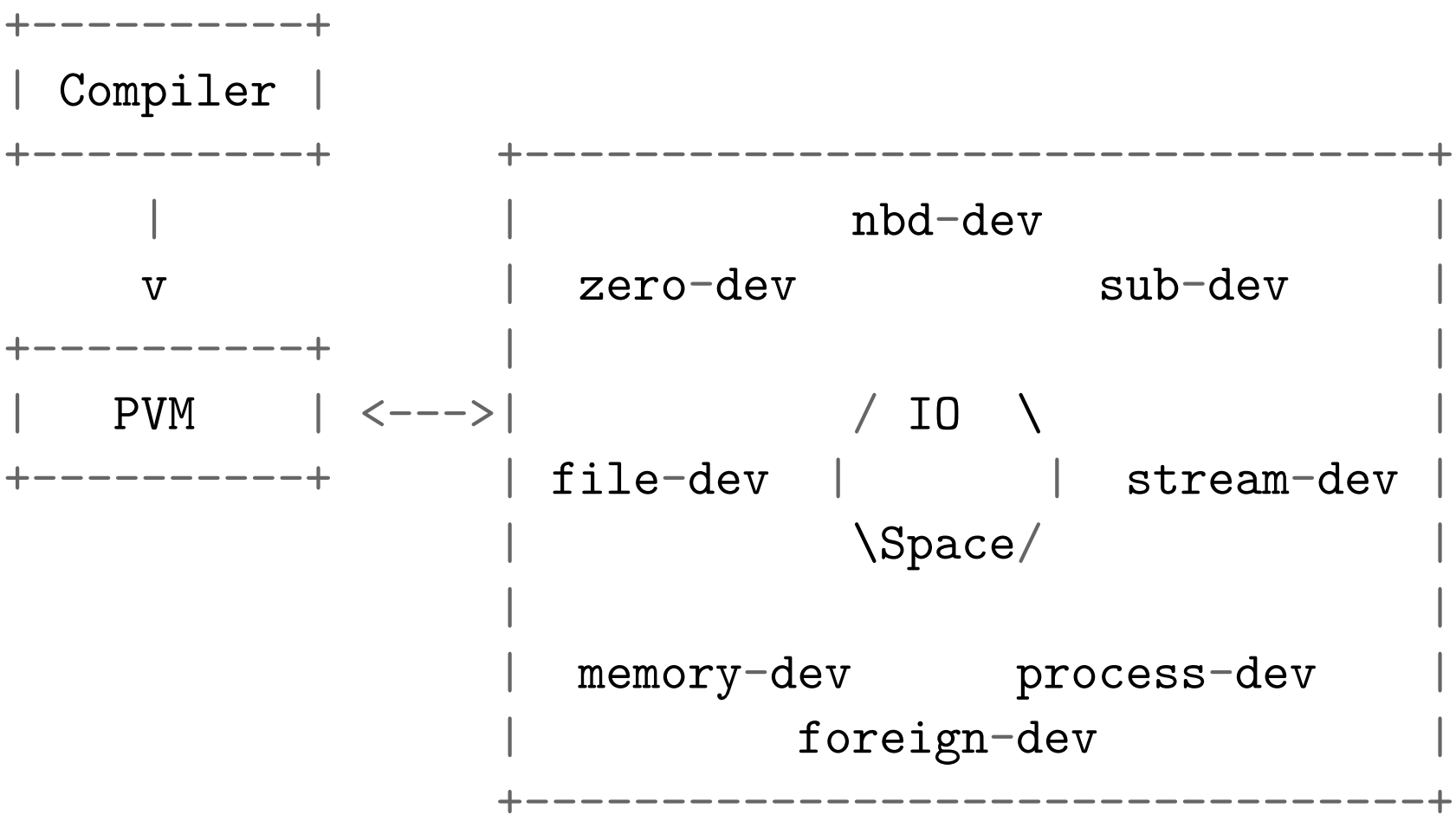
- Jitter-generated virtual machine
- Stack machine

Poke Virtual Machine (PVM)

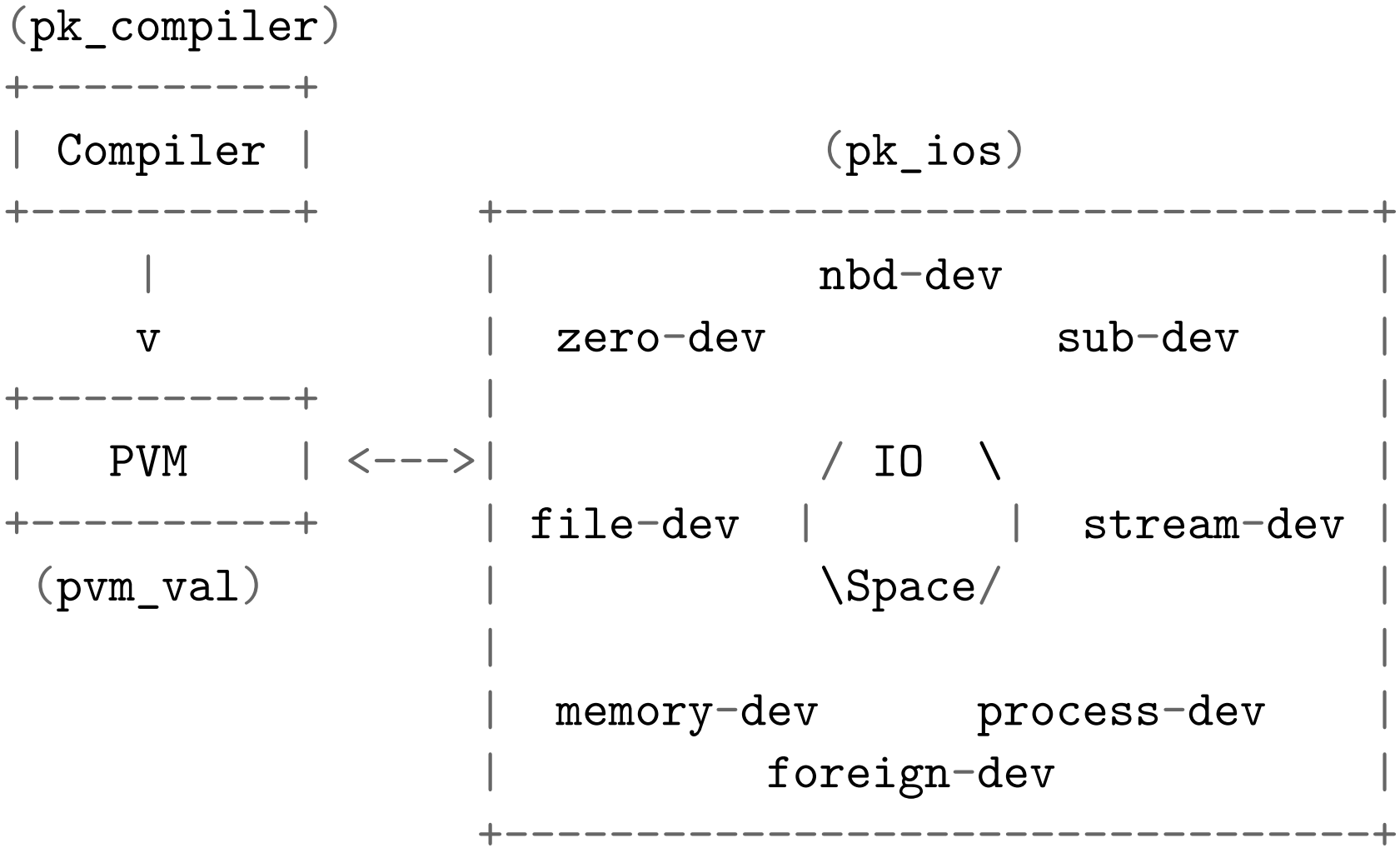
- Jitter-generated virtual machine
- Stack machine

Only PVM values are exposed to the user.

libpoke Components



libpoke Components



`libpoke` opaque data types

- `pk_compiler`
- `pk_val`
- `pk_ios`

Let's write some code!

We're going to write some code on slides!

Void!

```
int
main()
{
  return 0;
}
```

libpoke

```c
#include <libpoke.h>
int
main()
{
  return 0;
}
```

libpoke initialization

```
#include <libpoke.h>
int
main()
{
  struct pk_term_if term_if = {
    /*Function pointers for terminal output interface*/
  };
  pk_compiler pkc = pk_compiler_new (&term_if);
  pk_compiler_free (pkc);
  return 0;
}
```

libpoke initialization

```c
#include <err.h>
#include <libpoke.h>
int
main()
{
  struct pk_term_if term_if = { /* ... */ };
  pk_compiler pkc = pk_compiler_new (&term_if);
  if (pkc == NULL)
    err (1, "pk_compiler_new() failed");
  pk_compiler_free (pkc);
  return 0;
}
```

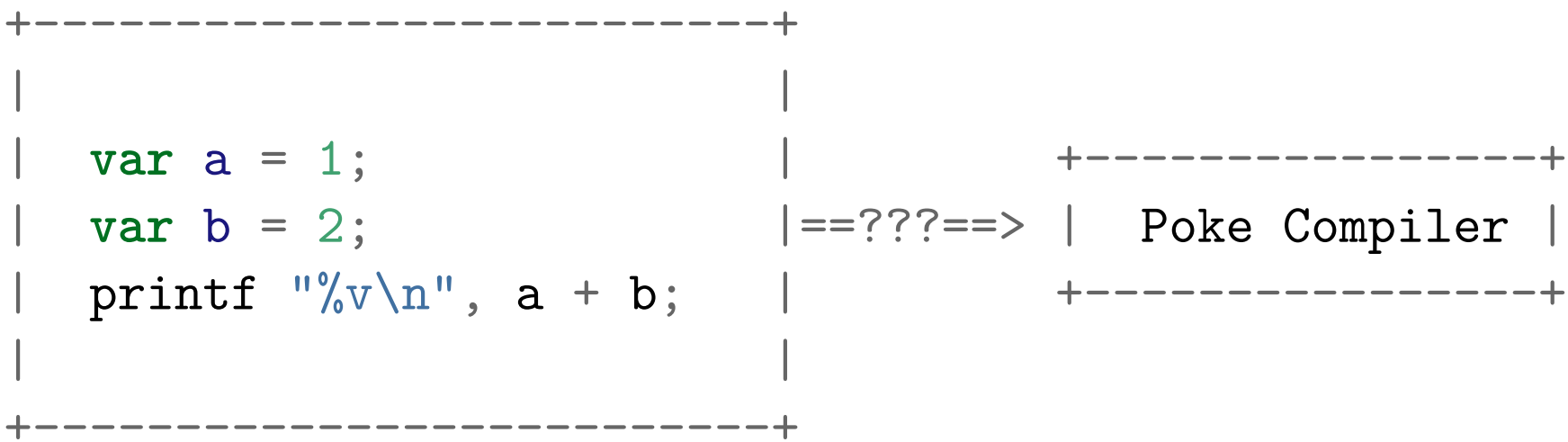
struct pk_term_if

```
struct pk_term_if {
  void (*flush_fn) (void);
  void (*puts_fn) (const char *str);
  void (*printf_fn) (const char *format, ...);
  void (*indent_fn) (unsigned int lvl, unsigned int step);
  void (*class_fn) (const char *aclass);
  int (*end_class_fn) (const char *aclass);
  void (*hyperlink_fn) (const char *url, const char *id);
  int (*end_hyperlink_fn) (void);
  struct pk_color (*get_color_fn) (void);
  struct pk_color (*get_bgcolor_fn) (void);
  void (*set_color_fn) (struct pk_color color);
  void (*set_bgcolor_fn) (struct pk_color color);
};
```

pkl-rt.pk and std.pk

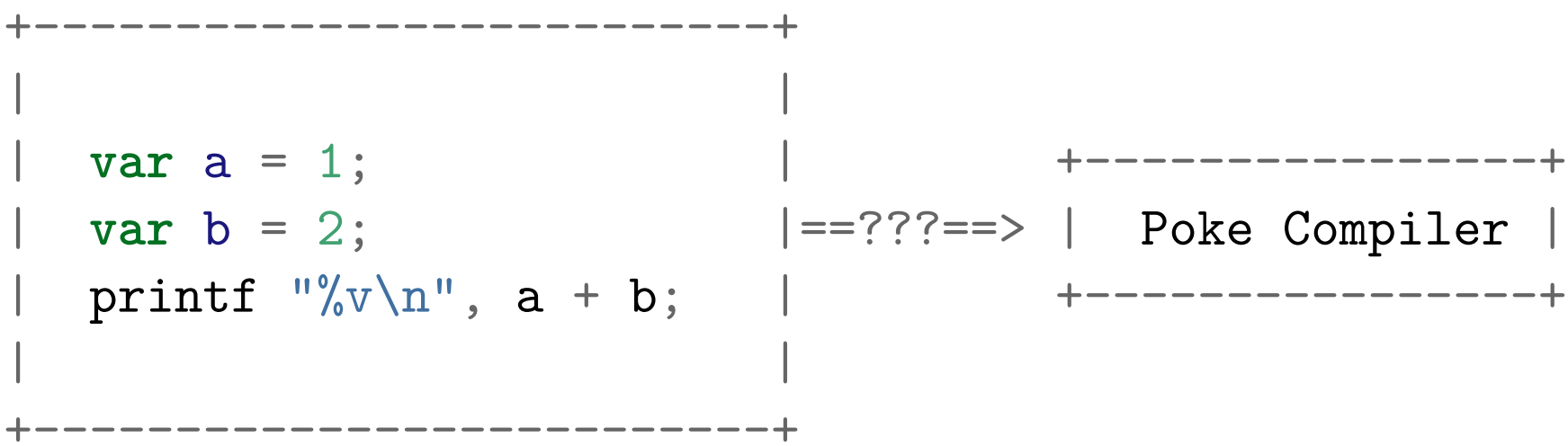
```c
#include <err.h>
#include <libpoke.h>
int
main()
{
  struct pk_term_if term_if = { /* ... */ };
  pk_compiler pkc = pk_compiler_new (&term_if);
  if (pkc == NULL)
    err (1, "pk_compiler_new() failed");
  /* `pkl-rt.pk` and `std.pk` have been loaded. */
  pk_compiler_free (pkc);
  return 0;
}
```

How to feed the Poke compiler?



How to feed the Poke compiler?

The assumption here is you have the Poke code as a string or file



Most useful functions!

```
int pk_compile_file (
  pk_compiler, const char *filename, pk_val *exc);

int pk_compile_buffer (
  pk_compiler, const char *code, const char **end,
  pk_val *exc);

int pk_compile_statement (pk_compiler, const char *code,
  const char **end, pk_val *val, pk_val *exc);

int pk_compile_expression (pk_compiler, const char *code,
  const char **end, pk_val *val, pk_val *exc);
```

Poke support code for your app

```
/* app.pk */

var some_int_setting = 0;
var some_str_setting = "no";

fun do_something = void:
  {
    /* ... */
  }
```

libpoke: Loading code from file

```
/* ... */
int main() {
  /* ... */
  pk_val exception;

  pk_compile_file (pkc, "app.pk", &exception);
  /* ... */
}
```

libpoke: Loading code from file

```
/* ... */
int main() {
  /* ... */
  pk_val exception;

  if (pk_compile_file (pkc, "app.pk", &exception)
      != PK_OK)
    /* Compile-time problems, like syntax error, etc. */
    errx (1, "Compilation of 'app.pk' failed");
  /* ... */
}
```

libpoke: Loading code from file

```
/* ... */
int main() {
  /* ... */
  pk_val exception;

  if (pk_compile_file (pkc, "app.pk", &exception)
      != PK_OK)
    errx (1, "Compilation of 'app.pk' failed");
  if (exception != PK_NULL) {
    /* The code raised an unhandled exception. */
    handle_exception(exception);
    goto somewhere;
  }
  /* ... */
}
```

`libpoke`: Loading code from file

Yay!

Successfully loaded `app.pk` :)

Let's go to the main loop!

To interact with the incremental compiler :)

Interaction strategies

- Put all logic in `app.pk` and only call Poke functions (`pk_call`)

Interaction strategies

- Put all logic in `app.pk` and only call Poke functions (`pk_call`)
- Generate Poke code (as string) and use `pk_compile_{buffer,statement,expression}`

Interaction strategies: Example I

```
/* app.pk */
var a = -1;
var b = 1;
fun reset_a_and_b = void: { a = -1; b = 1; }
```

```
/* app.c */
pk_val reset_a_and_b_val
  = pk_decl_val (pkc, "reset_a_and_b");

assert (reset_a_and_b_val != PK_NULL);
if (pk_call (
      pkc, reset_a_and_b_val, /*ret*/NULL, /*narg*/0)
    != PK_OK)
  errx (1, "Poke function invocation failed");
```

Interaction strategies: Example II

```
/* app.pk */
var a = -1;
var b = 1;

/* app.c */
pk_val exception;

if (pk_compile_buffer (
       pkc, "a = -1; b = 1;", /*end*/NULL, &exception)
    != PK_OK)
  errx (1, "pk_compile_buffer() failed");
if (exception != PK_NULL) {
  handle_exception (exception);
  goto somewhere_else;
}
```

Which approch is better?

It depends!

Which approch is better?

It depends!

But, I prefer the first approach (unless it doesn't make sense)

Isn't the first approch too slow?

No!

But, I didn't measure it :)

Poke keywords

Be careful!

```
static const char* keywords[] = {
  "pinned", "struct", "union", "else", "while", "until",
  "for", "in", "where", "if", "sizeof", "fun", "method",
  "type", "var", "unit", "break", "continue", "return",
  "string", "as", "try", "catch", "raise", "void", "any",
  "print", "printf", "isa", "unmap", "big", "little",
  "load", "lambda", "assert",
};
```

Compile-time environment & run-time environment

```
/* Poke */
var a = 1;
a *= 2;
```

Compile-time environment & run-time environment

```
/* Poke */
var a = 1;
a *= 2;
```

```
/* PVM asm */
push    0x1
regvar
pushvar 0x0, 0x0
push    0x2
muli
nip2
popvar  0x0, 0x0
```

Check if a variable is in compile-time env

```
/* libpoke.h */

#define PK_DECL_KIND_VAR 0
#define PK_DECL_KIND_FUNC 1
#define PK_DECL_KIND_TYPE 2

int pk_decl_p (pk_compiler pkc,
               const char *name,
               int kind);

// pk_decl_p (pkc, "a", PK_DECL_KIND_VAR);
```

PVM values

```
/* app.pk */

fun double = (int x) long:
  {
    return (x as long) * 2L;
  }
```

PVM values

```
/* app.pk */
fun double = (int x) long: { return (x as long) * 2L; }
```

```
/* app.c */
#define CHK(...) /* do error handling */
#define CHK_EXC(...) /* do Poke exception handling */

pk_val func_double = pk_decl_val (pkc, "double");
pk_val x = pk_make_int (/*value*/ 10, /*size*/ 32);
pk_val ret, exception;

CHK (pk_call (pkc, func_double, &ret, /*narg*/ 1, x));
pk_print_val (pkc, ret, &exception); /* 0x14L */
CHK_EXC (exception);
```

PVM values: Integers

```
/* libpoke.h */

pk_val pk_make_int (int64_t value, int size);
pk_val pk_make_uint (uint64_t value, int size);

int64_t pk_int_value (pk_val val);
uint64_t pk_uint_value (pk_val val);

int pk_int_size (pk_val val);
int pk_uint_size (pk_val val);
```

PVM values: Strings

```
/* libpoke.h */

pk_val pk_make_string (const char *str);

const char *pk_string_str (pk_val val);
```

PVM values: Offsets

```
/* libpoke.h */

pk_val pk_make_offset (pk_val magnitude, pk_val unit);

pk_val pk_offset_magnitude (pk_val val);
pk_val pk_offset_unit (pk_val val);
```

PVM values: ...

Please see `libpoke.h` for more info :)

Dealing with compile-time environment

```
/* libpoke.h */

int pk_defvar (pk_compiler pkc,
               const char *varname, pk_val val);

void pk_decl_set_val (pk_compiler pkc,
                      const char *name, pk_val val);
```

Alien Tokens

```
/* Lexer */
L [a-zA-Z_]
D [0-9]
A $
S ::

{A}({L}|{D})({L}|{D}|({S}({L}|{D})))* {
  pkl_alien_token_handler_fn cb
    = pkl_alien_token_fn (yyextra->compiler);

  if (pkl_lexical_cuckolding_p (yyextra->compiler)
      && cb != NULL) /* insert stuff into AST */
  else REJECT;
}
```

Alien Tokens

```
#define PK_ALIEN_TOKEN_IDENTIFIER 0
#define PK_ALIEN_TOKEN_INTEGER    1
#define PK_ALIEN_TOKEN_OFFSET     2
#define PK_ALIEN_TOKEN_STRING     3

/* Let's look at the actual code
 *    - Lexer code
 *    - GDB code
 */
```

IO Space

```
/* Poke */

var id_file = open ("/somewhere/something", IOS_M_RDONLY);
var id_mem = open ("*memory_1*");
var id_proc_mem = open ("pid://1234");
var id_nbd = open ("nbd+unix://socket");
// ...
```

IO Space

```
int pk_ios_open (pk_compiler pkc, const char *handler,
                 uint64_t flags, int set_cur_p);
void pk_ios_close (pk_compiler pkc, pk_ios ios);

pk_ios pk_ios_search (pk_compiler pkc,
                      const char *handler);
pk_ios pk_ios_search_by_id (pk_compiler pkc, int id);

pk_ios pk_ios_cur (pk_compiler pkc);
void pk_ios_set_cur (pk_compiler pkc, pk_ios ios);

uint64_t pk_ios_size (pk_ios ios);
uint64_t pk_ios_flags (pk_ios ios);
// ...
```

What if you need more IO spaces?

For sure `libpoke` cannot support **ALL** useful IO devices!

Foreign IO devices interface

```
struct pk_iod_if {
  const char *(*get_if_name) ();
  char * (*handler_normalize) (
    const char *handler, uint64_t flags, int* error);
  void * (*open) (
    const char *handler, uint64_t f, int *err, void *data);
  int (*close) (void *);
  int (*pread) (
    void *, void *buf, size_t n, pk_iod_off);
  int (*pwrite) (
    void *, const void *buf, size_t n, pk_iod_off);
  uint64_t (*get_flags) (void *);
  pk_iod_off (*size) (void *);
  int (*flush) (void *, pk_iod_off);
  void *data;
```

`libpoke` API Categories

- Terminal output interface callbacks

`libpoke` API Categories

- Terminal output interface callbacks
- Library intialization/finalization

`libpoke` API Categories

- Terminal output interface callbacks
- Library intialization/finalization
- Compilation and execution of Poke code

`libpoke` API Categories

- Terminal output interface callbacks
- Library intialization/finalization
- Compilation and execution of Poke code
- Alien token handling

`libpoke` API Categories

- Terminal output interface callbacks
- Library intialization/finalization
- Compilation and execution of Poke code
- Alien token handling
- PVM code debugging/profiling facilities

`libpoke` API Categories

- Terminal output interface callbacks
- Library intialization/finalization
- Compilation and execution of Poke code
- Alien token handling
- PVM code debugging/profiling facilities
- Declaration of types and variables

`libpoke` API Categories

- Terminal output interface callbacks
- Library intialization/finalization
- Compilation and execution of Poke code
- Alien token handling
- PVM code debugging/profiling facilities
- Declaration of types and variables
- Text completion

`libpoke` API Categories

- Terminal output interface callbacks
- Library intialization/finalization
- Compilation and execution of Poke code
- Alien token handling
- PVM code debugging/profiling facilities
- Declaration of types and variables
- Text completion
- IO space

`libpoke` API Categories

- Terminal output interface callbacks
- Library intialization/finalization
- Compilation and execution of Poke code
- Alien token handling
- PVM code debugging/profiling facilities
- Declaration of types and variables
- Text completion
- IO space
- Compiler/PVM settings

`libpoke` API Categories

- Terminal output interface callbacks
- Library intialization/finalization
- Compilation and execution of Poke code
- Alien token handling
- PVM code debugging/profiling facilities
- Declaration of types and variables
- Text completion
- IO space
- Compiler/PVM settings
- PVM values: Declaration, use and modification

Current Limitations and Future Work

Geting rid of global states in `libpoke`

The, you can

- run more than one `libpoke` instance in a process

Current Limitations and Future Work

Geting rid of global states in `libpoke`

The, you can

- run more than one `libpoke` instance in a process
- use a library which is also using `libpoke` itself

Current Limitations and Future Work

Geting rid of global states in `libpoke`

The, you can

- run more than one `libpoke` instance in a process
- use a library which is also using `libpoke` itself
- safely use `libpoke` with threads

Current Limitations and Future Work

Make Poke and C better friends

- `Poke code -> C code -> Poke code`
- Foreign functions

```
/* Poke */
fun do_something = (string x, int y) long:
  clibrary ("my_c_library"); /* my suggested syntax */

/* C */
pk_val do_something (pk_val x, pk_val y)
{ /* ... */ }
```