# FYP Final Presentation

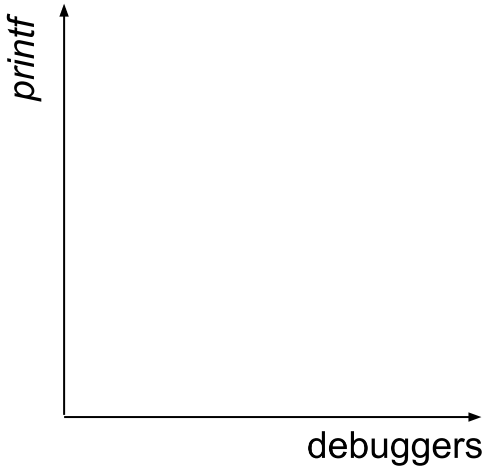
Higher-order Debugging and Logging in OCaml

# OCaml

- Multi-paradigm: functional, imperative, object-oriented
- Strong types, type inference
- Parametric polymorphism *and* (structural) subtyping
- Higher-order modules

# Debugging

- Inspect state at different points of a program
- Two main axes
  - *printf* debugging
    - Function tracing
    - Source-level
  - Use of tools
    - Debuggers, profilers, etc.
    - Bytecode



# Problem

- Debugging in OCaml is difficult!
- Strong types impose rigid structure
- Lack of accessible tooling for debugging

# *printf* debugging

- **Ad hoc polymorphism**
- “A polymorphic function can denote a number of distinct and potentially heterogeneous implementations depending on the type of argument(s) to which it is applied…”

- No function overloading, implicit coercions, typeclasses…
- No runtime type information/reflection by default
- Modular implicits (someday)

# Problem: no ad hoc polymorphism

- Leads to lots of boilerplate in practice

```
print_endline (string_of_int 1)
```

- *printf* debugging is cumbersome

# ocamldebug

- Great features!
  - Time-travel debugging
  - Events and time stamps
  - Module/function/event-level breakpoints
- Strict semantics

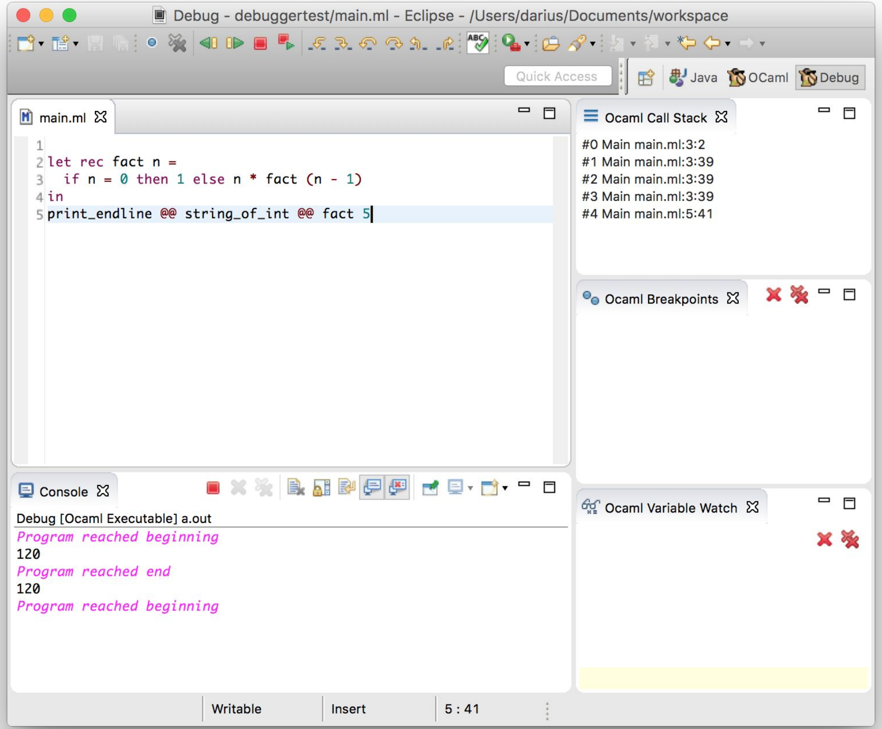
# Problem: lack of accessible tooling

- Command-line interface

```
→ ~/o/debug ocamldebug a.out
        OCaml Debugger version 4.02.3

(ocd) go 21
Loading program... done.
/Users/darius/ocaml/debug/a.out
Time: 21 - pc: 17004 - module Rd
53 Array.iter print_endline Sys.argv<|a|>;
(ocd) next
hello
Time: 26 - pc: 17028 - module Rd
58 print_endline "hello"<|a|>;
(ocd)
world
Time: 31 - pc: 17052 - module Rd
59 print_endline "world"<|a|>;
```

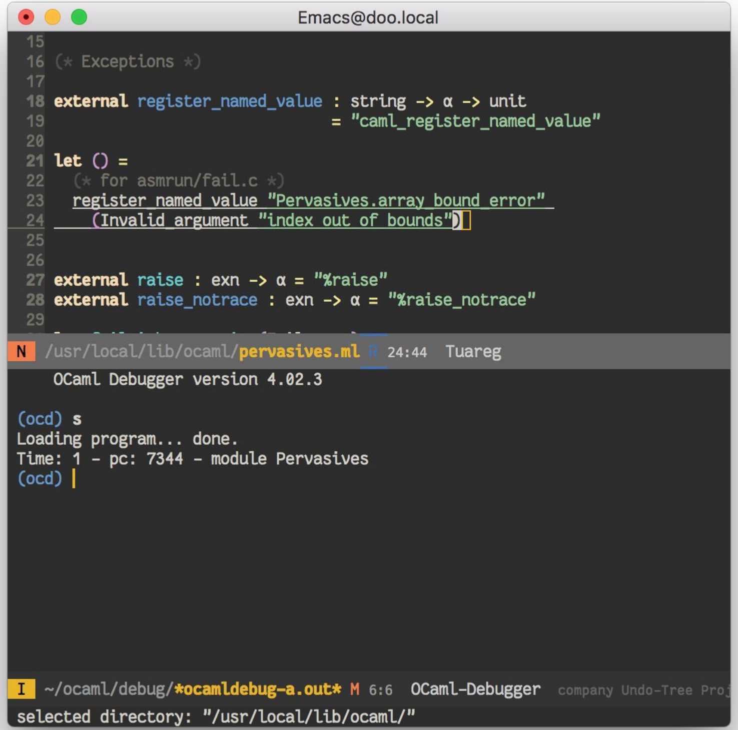
# Problem: lack of accessible tooling

- Eclipse frontend



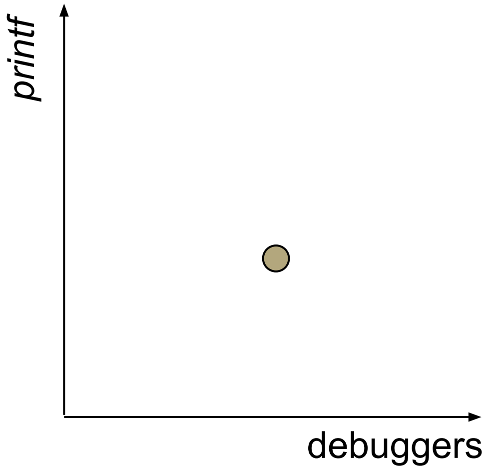
# Problem: lack of accessible tooling

- Emacs frontend



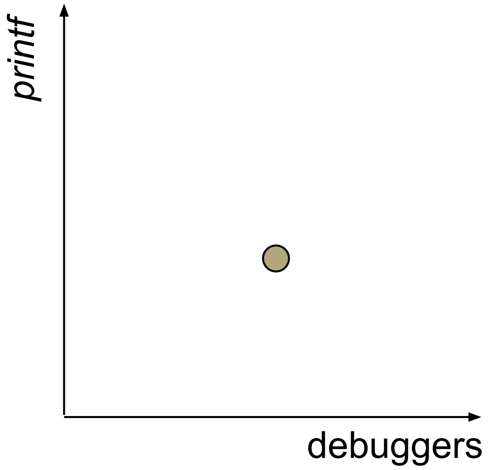
# Solutions

- Improve ocamldebug
  - Support tracing-friendly workflow
    - Interface: program's traces
    - Main means of navigation, rather than single-stepping and breakpoints
  - Takes advantage of time-travel
  - Superset of existing functionality
  - Focus on Emacs frontend



# Solutions

- Generate code
  - Implement ad hoc polymorphism
  - Generate tracing boilerplate

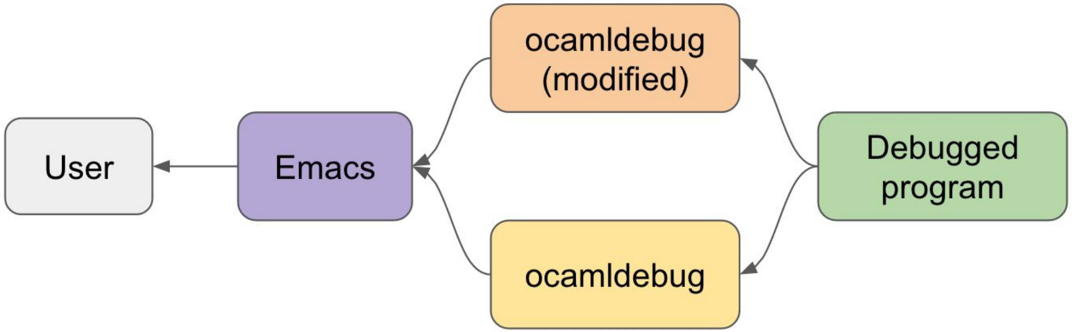


# ocamldebug

```
➜ ~/o/debug ocamldebug a.out
        OCaml Debugger version 4.02.3

(ocd) go 21
Loading program... done.
/Users/darius/ocaml/debug/a.out
Time: 21 - pc: 17004 - module Rd
53 Array.iter print_endline Sys.argv<|a|>;
(ocd) next
hello
Time: 26 - pc: 17028 - module Rd
58 print_endline "hello"<|a|>;
(ocd)
world
Time: 31 - pc: 17052 - module Rd
59 print_endline "world"<|a|>;
```

# Initial Architecture

- Separate instance of `ocamldebug`
  - Modified to single-step continuously
- Use info to control first instance of `ocamldebug`

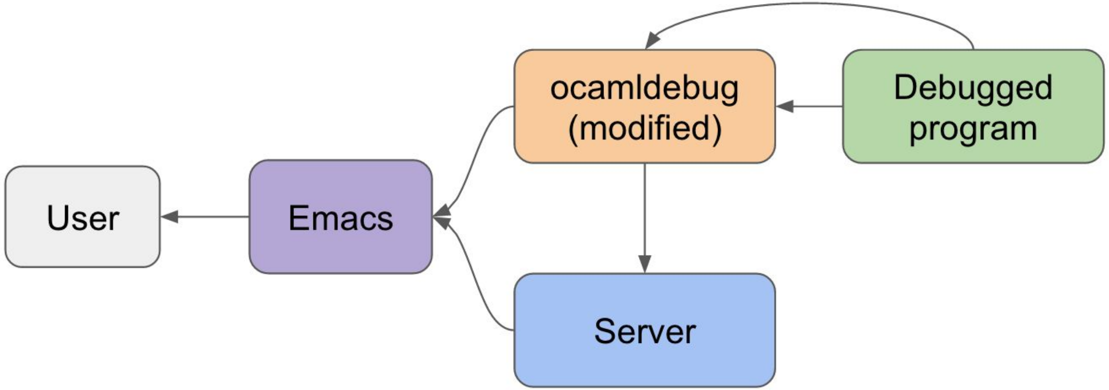


# Initial Architecture

- Worked as a first cut, but...
  - Inefficient: two instances of `ocamldebug`
  - Second instance only used to generate log
  - Interface wasn't much improved
  - User input and debugger output interleaved
  - All modifications were compiled directly into debugger

# Final Architecture

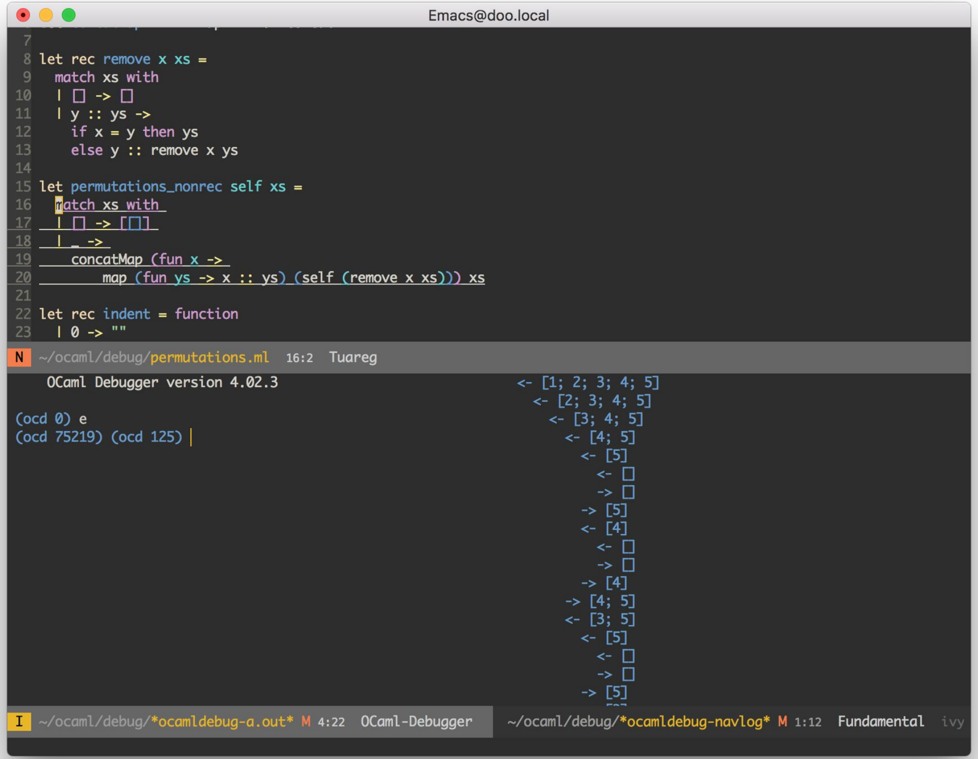
- Intermediate server between `ocamldebug` and Emacs
  - Buffers, formats, and controls output to Emacs



# Final Architecture

- Higher potential for interactivity
- Debugger only modified to send events to server
- Written in OCaml instead of Emacs Lisp

# Result


Emacs@doo.local
let rec remove x xs =
match xs with
| [] -> []
| y :: ys ->
if x = y then ys
else y :: remove x ys
let permutations_nonrec self xs =
match xs with
| [] -> [[]]
| _ ->
concatMap (fun x ->
map (fun ys -> x :: ys) (self (remove x xs))) xs
let rec indent = function
| 0 -> ""
N ~/ocaml/debug/permutations.ml 16:2 Tuareg
OCaml Debugger version 4.02.3
(ocd 0) e
(ocd 75219) (ocd 125)
<- [1; 2; 3; 4; 5]
<- [2; 3; 4; 5]
<- [3; 4; 5]
<- [4; 5]
<- [5]
<- []
-> []
-> [5]
<- [4]
<- []
-> []
-> [4]
-> [4; 5]
<- [3; 5]
<- [5]
<- []
-> []
-> [5]
I ~/ocaml/debug/*ocamldebug-a.out* M 4:22 OCaml-Debugger
~/ocaml/debug/*ocamldebug-navlog* M 1:12 Fundamental ivy

# Issues

- Concurrency
  - Debugger: system threads
  - Server: cooperative multitasking; scales transparently
- Synchronisation
  - Mapping output lines to time stamps
- Communication
  - S-expressions
- Augmented client
  - end

# Code generation

# Problem: lack of ad hoc polymorphism

```
let f : int list -> string option =
  fun n -> ...
```

# Problem: lack of ad hoc polymorphism

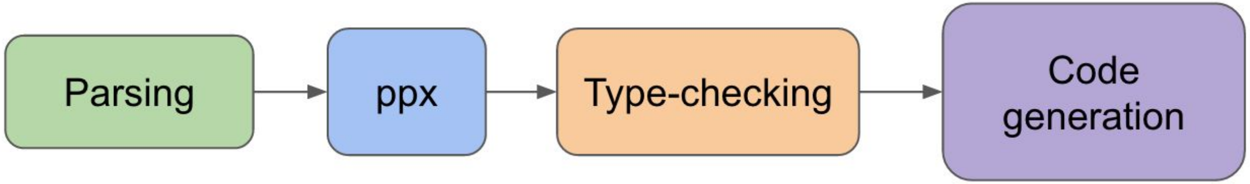
```
let f : int list -> string option =
  let pr_arg = string_of_list string_of_int in
  let pr_result = string_of_option id in
  Debug.no_1 "f" pr_arg pr_result f a
```

# Boilerplate

- Potential for copy-paste bugs and inconsistencies
- Tedious and boring to write; breaks flow
- Clutters code

# Solution: syntactic metaprogramming

- Code generation with ppx
  - Compile-time preprocessing of the AST
  - Enables us to transform code in various ways
  - Untyped by default



# Goals

Figure out parameter types

```
let fact : int list -> string option =
  let pr_arg = string_of_list string_of_int in
  let pr_result = string_of_option id in
  Debug.no_1 "fact" pr_arg pr_result fact a
```

Provide a means of customisation

Generate function boilerplate

# Goal: figuring out parameter types

- Ad hoc polymorphism (i.e. overloading)

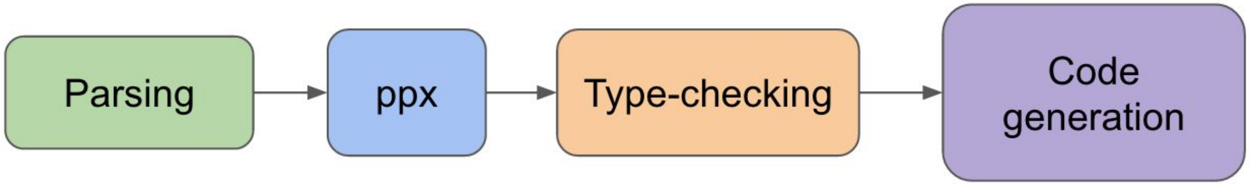
```
show 1
=> 1

show (true, "true")
=> (true, "true")
```

# Goal: ad hoc polymorphism

- Different methods
  - Overloading
    - Closed (SML)
    - Open (F#)
  - Typeclasses (Haskell)
- Our method: SML overloading → typeclasses
- Extending OCaml
  - Compiler extension
  - Typed ppx transformation

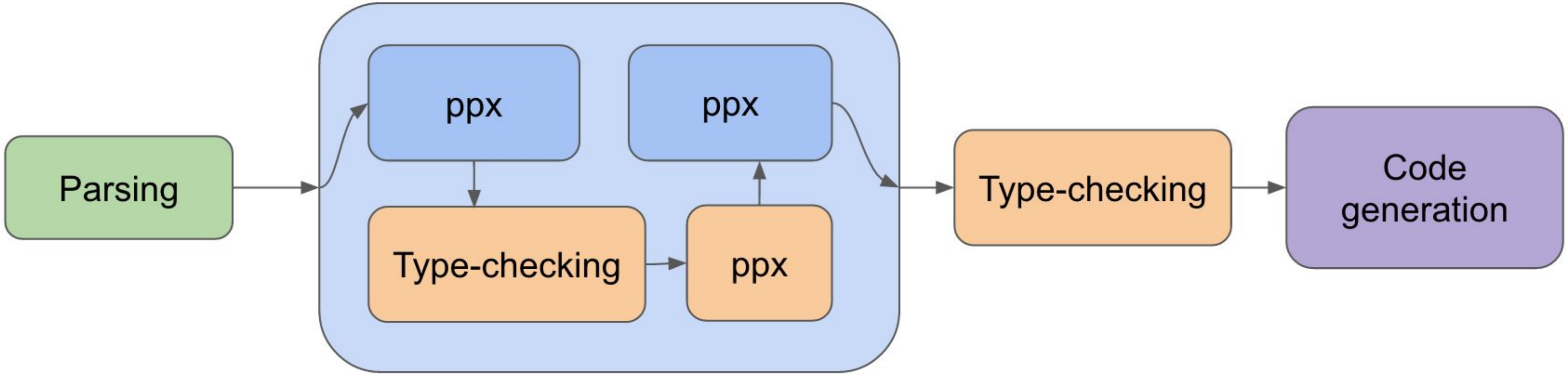
# Goal: ad hoc polymorphism

- Traditional (untyped) ppx process



# Goal: ad hoc polymorphism

- Augmented (typed) ppx process



# Goal: ad hoc polymorphism

- Monomorphic versions of functions are inserted

```
show 1
```

```
Format.sprintf "%d" 1
```

# Goal: ad hoc polymorphism

- Monomorphic versions of functions are inserted

```
show (true, "true")
```

```
Format.asprintf "%a"
  (PolyPrint.Printers.pp_tuple Format.pp_print_bool
    Format.pp_print_string) (true, "true")
```

# Goal: ad hoc polymorphism

- Functions in higher-order contexts?

```
show : 'a -> string

let apply printer value = printer value

apply show 1
```

show is an identifier, and its inferred type is polymorphic:
`'a -> string`

# Goal: ad hoc polymorphism

- Solution: eta abstraction

```
show : 'a -> string

let apply printer value = printer value

apply (fun x -> show x) 1
```

show is applied to a monomorphic type variable, and its inferred type is now: `int -> string`

# Goal: ad hoc polymorphism

- Abstract data types?

```
(* User-defined *)
module M : sig
  type t
  val thing : t
  val pp_t : Formatter -> t -> string
end = ...
```

# Goal: ad hoc polymorphism

- Solution: follow conventions

```
show M.thing
```

```
Format.asprintf "%a" M.pp_t M.thing
```

# Limitations: ad hoc polymorphism

- We can’t implement `show : 'a. 'a -> string` meaningfully without constraints on `'a`

```
let stringify x =
  show x        Has type 'a

stringify 1
=> <polymorphic>
```

# Limitations: ad hoc polymorphism

- We can’t implement `show : 'a. 'a -> string` meaningfully without constraints on `'a`
- Can access compiler internals, but cannot modify behaviour
- Shortcoming of our implementation
- Hypothetical type-checker extension

# Proposal: ad hoc polymorphism

- *Lifting* transform
  - Abstract over function implementations
  - Conceptually, propagate information from call sites
  - A variety of use cases
    - Abstracting over recursive calls (more on this later)
    - Generalising implementations
    - Polymorphism

# Proposal: ad hoc polymorphism

- *Lifting* transform
    - Predicate to figure out which functions are eligible
        - Inductively defined
    - Annotated function arrow

$$\frac{\text{let}\, f = \text{fun } x \to y : T_x \to T_y, p(y)}{\text{let}\, f = \text{fun } s\ x \to y[a \rightarrowtail s] : T_a \to^a T_x \to T_y}\ [def]$$

```
let stringify x =
  show x
```

⇒

```
let stringify s x =
  s x
```

# Proposal: ad hoc polymorphism

- *Lifting* transform
  - Function of call site environment
  - Insert missing parameter
  - Remove annotated arrow

$$\frac{(a : T_a \rightarrow^a T_x \rightarrow T_y)(x : T_x), f(\Gamma) : T_a}{(a\ f(\Gamma) : T_x \rightarrow T_y)(x : T_x) : T_y}\ [use]$$

```
stringify x
```

$\Longrightarrow$

```
stringify show x
```

# Goals

Figure out parameter types

```
let fact : int list -> string option =
  let pr_arg = string_of_list string_of_int in
  let pr_result = string_of_option id in
  Debug.no_1 "fact" pr_arg pr_result fact a
```

Generate function boilerplate

# Goal: generating function boilerplate

- ppx annotations

```
let plus x y = x + y
  [@@trace]
```

```
(* Conceptual implementation *)
let print x = show x |> print_endline
```

# Goal: generating function boilerplate

- Tracing
  - Non-recursively (topmost call only) `[@@trace]`
  - Recursively (all calls) `[@@tracerec]`

# Goal: generating function boilerplate

- Tracing non-recursively

Print parameters

```
let plus x y =
  print x; print y;
  let result = x + y in
  print result;
  result
```

Print result

# Goal: generating function boilerplate

- Tracing recursively

```
let rec fact n =
  if n = 0 then 1 else n * fact (n - 1)
  [@@tracerec]
```

# Goal: generating function boilerplate

- Tracing recursively
  - Make function non-recursive by abstracting over recursive call
  - Take the fixed point of the function (‘tying the knot’)
  - Add instrumentation

# Goal: generating function boilerplate

- Tracing recursively

Abstract over recursive call
(function is no longer recursive)

Preserve original function

```
let fact n =
  let fact_original self n =
    if n = 0 then 1 else n * self (n - 1) in
  let rec aux n =
    print n;
    let result = fact_original aux in
    print result;
    result in
  aux n
```

Take the fixed point of `fact_original` (‘tie the knot’)

Print parameters and result

# Goals

Figure out parameter types

```
let fact : int list -> string option =
  let pr_arg = string_of_list string_of_int in
  let pr_result = string_of_option id in
  Debug.no_1 "fact" pr_arg pr_result fact a
```

Provide a means of customisation

Generate function boilerplate

# Goal: customisable tracing

- Generalising…

```
let plus x y = x + y
  [@@trace Config]
```

Allow customisation

# Goal: customisable tracing

- Generalising...

Wrap function call

```
let plus x y =
  let plus_original x y = x + y in
  Config.run2 "plus"
    ("x", show x, x)
    ("y", show y, y)
    show plus_original
```

Augment information available

show called in higher-order context

# Goal: customisable tracing

- Configuration module

```
[@@trace Default]
```

```
fact <- n = 3
fact -> 6
```

# Goal: customisable tracing

- Configuration module

```
[@@tracerec Default]
```

```
fact <- n = 3
fact <- n = 2
fact <- n = 1
fact <- n = 0
fact -> 1
fact -> 1
fact -> 2
fact -> 6
```

# Goal: customisable tracing

- Configuration module

```
[@@tracerec Recursive]
```

```
fact <- n = 3
  fact <- n = 2
    fact <- n = 1
      fact <- n = 0
      fact -> 1
    fact -> 1
  fact -> 2
fact -> 6
```

# Goal: customisable tracing

- Configuration module

```
[@@tracerec Minimal]
```

```
fact 3
  fact 2
    fact 1
      fact 0
      = 1
    = 1
  = 2
= 6
```

# Goal: customisable tracing

- DSL for annotations

```
[@@trace Config]

[@@trace Config; x, y]

[@@trace Config; x { printer = string_of_int }, y]
```

# Additional features

- Call site wrapping
  - Make call site information available to traced functions
  - If a function is traced, corresponding function calls should be wrapped

```
let rec fact n = ...
  [@@tracerec]

fact 5
```

⇨

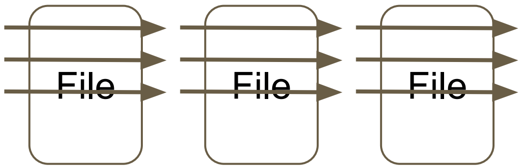
```
(PolyPrint.DefaultTraceConfig.call1
  "fact" (__FILE__, __LINE__) fact 5)
```

# Difficulties: wrapping of call sites

- How to identify call sites?
    - Higher-order language (aliasing; name not sufficient)
    - Partial application
    - Across modules
- Idea: propagate information via types
    - Can only access, not modify compiler internals
- Our implementation
    - Use (local function name, arity, type) triple
    - Assume uniqueness
    - Use first application
- Hypothetical compiler extension

# Maintaining state across ppx invocations

- ppx invocations are independent by design
  - Compilation units are processed modularly and independently




Multiple partial passes over one input file at a time

# Difficulties: persisting ppx environment

- Function definitions seen previously must be remembered!
    - Call sites come first in a topological ordering
- Idea: use interface (mli/cmi) file
    - Limited potential to change compiler with ppx
- Our implementation: persist ppx environment to a file
    - Parallel builds (races vs reduced parallelism)
    - Incremental compilation (staleness, no way to invalidate state)
- Hypothetical extension...

# Proposal: wrapping of call sites

- *Wrapping* transform
  - Carry function's traced status in types (persisted to mli/cmi)
  - Starred function arrow

$$\frac{\text{let } f = \text{fun } x \rightarrow y : T_x \rightarrow T_y \ [\text{@@trace}]}{\text{let } f = \text{fun } x \rightarrow y : T_x \rightarrow^* T_y} \ [trace]$$

# Proposal: wrapping of call sites

- *Wrapping* transform
  - Traced functions need to be wrapped to lose their starred arrow
  - Distinguish partial and saturated application (control call site used)

$$\frac{(a : T_1 \rightarrow^* T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_r)(b_1 : T_1)\ldots(b_n : T_n)}{call_n\ a\ b_1\ldots b_n : T_r}\ [call_n(saturated)]$$

$$\frac{(a : T_1 \rightarrow^* T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_r)(b_1 : T_1)\ldots(b_m : T_m), m < n}{\text{fun}\ b_{m+1}\ldots b_n \rightarrow call_n\ a\ b_1\ldots b_n : T_r}\ [call_n(partial)]$$

# Proposal: wrapping of call sites

- *Wrapping* transform
  - Solves both problems
    - Types are persisted in cmi/mli
    - Types allow us to identify traced functions
  - Starred arrow can be extended with additional information

# Issues

- Integration with wider ecosystem
  - Bugs discovered and reported in compiler, Merlin, ocamlbuild, ocamlfind, libraries, preprocessors, testing framework...
- Limitations of typed ppx transformations
  - Fixed position in the pipeline
  - Part of the build system

# Summary

- Typed ppx transformations are interesting
  - Compiler extensions as libraries
  - Lots of power
- … and a bit idiosyncratic
  - Limitations due to design and being part of build system
  - Don’t work in the top level
  - Application order of build tools matters
  - Dependent on compiler version

# Summary

- Improvements to ocaml debugging
  - Debugger extensions
  - Ad hoc polymorphism
  - Boilerplate generation

# Integration and future work

- Code generation and online debugging do not interoperate seamlessly
  - Generated ASTs don’t exist in source code
- Solutions
  - Investigate bytecode format
  - Restore debugging information

# Thank you!

Q&A