

## Catch me if you can

Towards type-safe, hierarchical, lightweight, polymorphic and efficient  
error management in OCaml  
in about 60 lines of code

David Teller, Arnaud Spiwack and Till Varoquaux

Lifo, Université d'Orléans + Ensi de Bourges / Lix, École Polytechnique

September 10, 2008

- 1 Introduction
- 2 The error monad
- 3 Representing errors
- 4 The need for speed
- 5 Conclusions

# Two awful truths

- 1 Even correct programs fail (think disk error, LHC...).
- 2 Handling failures can easily make your program incorrect (think missing failures, treating failures too early).

# The ideal

- 1 Ensure that all possible failures are managed.
- 2 Carry detailed information about error.
- 3 Only require dealing with errors which may actually happen.
- 4 All of this without getting in the way of the programmer.

# Exceptions

Mechanism introduced in PL/I, given semantics with ML. Present in most languages.

# Exceptions

Mechanism introduced in PL/I, given semantics with ML. Present in most languages.

- 1 Mark some expressions as failible (e.g. `try`).

# Exceptions

Mechanism introduced in PL/I, given semantics with ML. Present in most languages.

- 1 Mark some expressions as failible (e.g. `try`).
- 2 Install failure handlers to failible expressions (e.g. `with/catch`).

# Exceptions

Mechanism introduced in PL/I, given semantics with ML. Present in most languages.

- 1 Mark some expressions as failible (e.g. `try`).
- 2 Install failure handlers to failible expressions (e.g. `with/catch`).
- 3 In case of failure, stop the failible expression and try to find the right handler.

# Exceptions

Mechanism introduced in PL/I, given semantics with ML. Present in most languages.

- 1 Mark some expressions as failible (e.g. `try`).
- 2 Install failure handlers to failible expressions (e.g. `with/catch`).
- 3 In case of failure, stop the failible expression and try to find the right handler.
- 4 If there's no correct handler, propagate failure.

# Implementations

- 1 Ensure that all possible failures are managed.
- 2 Carry detailed information about error.
- 3 Only require dealing with errors which may actually happen.
- 4 All of this without getting in the way of the programmer.

# Implementations

- 1 Ensure that all possible failures are managed.
- 2 Carry detailed information about error.
- 3 Only require dealing with errors which may actually happen.
- 4 All of this without getting in the way of the programmer.

**C++**, **C#** Unchecked exceptions (-1, 2?, 3, 4).

**Java** Checked annotated exceptions (-1!, 2, -3, -4).

# Implementations

- 1 Ensure that all possible failures are managed.
- 2 Carry detailed information about error.
- 3 Only require dealing with errors which may actually happen.
- 4 All of this without getting in the way of the programmer.

**C++**, **C#** Unchecked exceptions ( $\neg 1$ , 2?, 3, 4).

**Java** Checked annotated exceptions ( $\neg 1!$ , 2,  $\neg 3$ ,  $\neg 4$ ).

**OCaml/SML** Unchecked exceptions ( $\neg 1$ ,  $\neg 2$ , 3, 4).

**Haskell** Error monads (1, 2,  $\neg 3$ ,  $\neg 4$ ).

# Implementations

- 1 Ensure that all possible failures are managed.
- 2 Carry detailed information about error.
- 3 Only require dealing with errors which may actually happen.
- 4 All if this without getting in the way of the programmer.

C++, C# Unchecked exceptions ( $\neg 1$ , 2?, 3, 4).

Java Checked annotated exceptions ( $\neg 1!$ , 2,  $\neg 3$ ,  $\neg 4$ ).

OCaml/SML Unchecked exceptions ( $\neg 1$ ,  $\neg 2$ , 3, 4).

Haskell Error monads (1, 2,  $\neg 3$ ,  $\neg 4$ ).

OCamlExc Checked exceptions (1,  $\neg 2$ , 3, 4).

Catch Refined error monads (1, 2, 3,  $\neg 4$ ).

Can we do better?

Can we do better? Preferably without developing a new programming language or a new type system.

- 1 Introduction
- 2 The error monad**
- 3 Representing errors
- 4 The need for speed
- 5 Conclusions

## Failures can happen

Let's toy with expressions:

```
type expr =  
  Value of float  
| Div   of expr * expr  
| Add   of expr * expr
```

## Failures can happen

Let's toy with expressions:

```
type expr =  
  Value of float  
| Div   of expr * expr  
| Add   of expr * expr
```

And let's evaluate them:

```
let rec eval = function  
| Value f      → f  
| Div (x, y)   → (eval x) /. (eval y)  
| Add (x, y)   → (eval x) +. (eval y)
```

## Failures can happen

Let's toy with expressions:

```
type expr =  
  Value of float  
| Div   of expr * expr  
| Add   of expr * expr
```

And let's evaluate them:

```
let rec eval = function  
| Value f      → f  
| Div (x, y)   → (eval x) /. (eval y)  
| Add (x, y)   → (eval x) +. (eval y)
```

**Possible failure** division by zero.

## Failures can be managed one-by-one...

```
let rec eval = function
| Value f      → Ok f
| Div (x, y) → (
  match eval x with
  | Error e → Error e
  | Ok x'  → match eval y with
    | Error e → Error e
    | Ok y' when y' = 0 →
      Error "Divison by 0"
    | Ok y'  → Ok (x' /. y')
)
| Add (x, y) → (
  match eval x with
  | Error e → Error e
  | Ok x'  → match eval y with
    | Error e → Error e
    | Ok y'  → Ok (x' +. y')
)
```

...or factorized...

```
| let bind m k = match m with  
|   Ok x      → k x  
|   Error _   → m
```

...or factorized...

```
let bind m k = match m with  
  Ok x      → k x  
  Error _   → m
```

```
let rec eval = function  
  Value f      → Ok f  
  Div (x, y)   →  
    bind (eval x) (fun x' →  
    bind (eval y) (fun y' →  
      if y' = 0. then Error "Division by 0"  
      else Ok (x' /. y'))))  
  Add (x, y)   →  
    bind (eval x) (fun x' →  
    bind (eval y) (fun y' →  
      Ok (x' /. y'))))
```

## ...into monads

```
let rec eval = function
| Value f          → return f
| Div (x, y)       → perform
    x' ← eval x;
    y' ← eval y;
    if y' = 0. then fail "Division by 0"
    else return (x' /. y')
| Add (x, y)       → perform
    x' ← eval x;
    y' ← eval y;
    return (x' +. y')
```

## Bottom line

Up Coverage checking.

Up Errors can carry any information.

## Bottom line

Up Coverage checking.

Up Errors can carry any information.

Down Slow.

Down Somewhat obtrusive.

## Bottom line

Up Coverage checking.

Up Errors can carry any information.

Down Slow.

Down Somewhat obtrusive.

? What information should I carry?

? How should I declare errors?

- 1 Introduction
- 2 The error monad
- 3 Representing errors**
- 4 The need for speed
- 5 Conclusions

## About polymorphic variants

Polymorphic variants = sum types + sharing of constructors -  
declarations.

## About polymorphic variants

Polymorphic variants = sum types + sharing of constructors -  
declarations.

```
# let l = ['A; 'B; 'C 5];;  
val l : [> 'A | 'B | 'C of int ] list =  
        ['A; 'B; 'C 5]
```

## About polymorphic variants

Polymorphic variants = sum types + sharing of constructors -  
declarations.

```
# let l = ['A; 'B; 'C 5];;  
val l : [> 'A | 'B | 'C of int ] list =  
        ['A; 'B; 'C 5]
```

```
# let m = ['B; 'E "jacques garrigue"];;  
val m : [> 'B | 'E of string ] list =  
        ['B; 'E "jacques garrigue"]
```

## About polymorphic variants

Polymorphic variants = sum types + sharing of constructors - declarations.

```
# let l = ['A; 'B; 'C 5];;  
val l : [> 'A | 'B | 'C of int ] list =  
        ['A; 'B; 'C 5]
```

```
# let m = ['B; 'E "jacques garrigue"];;  
val m : [> 'B | 'E of string ] list =  
        ['B; 'E "jacques garrigue"]
```

```
# l @ m ;;  
- : [> 'A | 'B | 'C of int | 'E of string ] list =  
        ['A; 'B; 'C 5; 'B; 'E "jacques garrigue"]
```

## About polymorphic variants

Polymorphic variants = sum types + sharing of constructors - declarations.

```
# let l = ['A; 'B; 'C 5];;  
val l : [> 'A | 'B | 'C of int ] list =  
        ['A; 'B; 'C 5]
```

```
# let m = ['B; 'E "jacques garrigue"];;  
val m : [> 'B | 'E of string ] list =  
        ['B; 'E "jacques garrigue"]
```

```
# l @ m ;;  
- : [> 'A | 'B | 'C of int | 'E of string ] list =  
        ['A; 'B; 'C 5; 'B; 'E "jacques garrigue"]
```

Symbol > states that the variant is open: new constructors may be added.

## With polymorphic variants, we can...

### Represent errors

```
# fun x → x +.. x ;;  
- : float →  
  (float, [> 'AddOverflow      ]) result = <fun>
```

```
# fun x → x /.. x ;;  
- : float →  
  (float, [> 'DivisionByZero  
          | 'DivOverflow      ]) result = <fun>
```

## With polymorphic variants, we can...

### Represent errors

```
# fun x → x +.. x ;;
- : float →
  (float, [> 'AddOverflow      ]) result = <fun>
```

```
# fun x → x /.. x ;;
- : float →
  (float, [> 'DivisionByZero
          | 'DivOverflow      ]) result = <fun>
```

### Combine errors

```
# fun x → x /.. x +.. x;;
- : float →
  (float, [> 'AddOverflow
          | 'DivisionByZero
          | 'DivOverflow      ]) result = <fun>
```

## What about hierarchies?

Arithmetic exception is just an exception

## What about hierarchies?

Arithmetic exception is just an exception

Arithmetic exception can be a Division\_by\_zero

Arithmetic exception can be a Overflow

## What about hierarchies?

Arithmetic exception is just an exception

Arithmetic exception can be a Division\_by\_zero

Arithmetic exception can be a Overflow

Arithmetic exception + Overflow can be an Addition\_overflow

Arithmetic exception + Overflow can be a Division\_overflow

## What about hierarchies?

Arithmetic exception is just an exception

Arithmetic exception can be a Division\_by\_zero

Arithmetic exception can be a Overflow

Arithmetic exception + Overflow can be an Addition\_overflow

Arithmetic exception + Overflow can be a Division\_overflow

...

## What about hierarchies?

Arithmetic exception is just an exception

Arithmetic exception can be a Division\_by\_zero

Arithmetic exception can be a Overflow

Arithmetic exception + Overflow can be an Addition\_overflow

Arithmetic exception + Overflow can be a Division\_overflow

...

Sounds like object-orientation, just reversed.

## What about hierarchies?

Arithmetic exception is just an exception

Arithmetic exception can be a Division\_by\_zero

Arithmetic exception can be a Overflow

Arithmetic exception + Overflow can be an Addition\_overflow

Arithmetic exception + Overflow can be a Division\_overflow

...

Sounds like object-orientation, just reversed.

Polymorphic variants let us encode an open hierarchy.

## What about hierarchies?

Arithmetic exception is just an exception

Arithmetic exception can be a Division\_by\_zero

Arithmetic exception can be a Overflow

Arithmetic exception + Overflow can be an Addition\_overflow

Arithmetic exception + Overflow can be a Division\_overflow

...

Sounds like object-orientation, just reversed.

Polymorphic variants let us encode an open hierarchy.

```
type ( $\alpha$ ,  $\beta$ ) ex = {  
  information:  $\alpha$ ;  
  can_be_a:  $\beta$  option;  
} constraint  $\beta$  = [ $>$  ]
```

...

...

Code-rewriting involved.

...

Code-rewriting involved.  
Generation of code-rewriters involved.

. . .

Code-rewriting involved.  
Generation of code-rewriters involved.  
(Musical Cue: *Team America, World Police*, “Montage”)

## Declaring exception types

Declaration is simple

```
exception arithmetics = Arithmetic  
exception overflow   = Overflow  
  of arithmetics
```

## Declaring exception types

Declaration is simple

```
exception arithmetics = Arithmetic  
exception overflow   = Overflow  
of arithmetics
```

And it hides something not too complex

```
let arithmetic ?sub content =  
  'Arithmetic { content = content; sub = sub }  
  
let overflow ?sub content =  
  arithmetic  
  ~sub:( 'Overflow {content = content; sub = sub } )  
  content
```

## Catching errors

Exception-matching is simple

```
attempt eval e with
| div_by_zero _          →
  print_string "Division by zero"
| add_overflow _ _      →
  print_string "Overflow while adding"
| overflow _            →
  print_string "Other overflow"
| val f                 →
  print_float f
| finally _             → ()
```

## Catching errors

Exception-matching is simple

```
attempt eval e with
| div_by_zero _          →
  print_string "Division by zero"
| add_overflow _ _      →
  print_string "Overflow while adding"
| overflow _            →
  print_string "Other overflow"
| val f                  →
  print_float f
| finally _              → ()
```

The generated code is relatively simple, too.

- 1 Introduction
- 2 The error monad
- 3 Representing errors
- 4 The need for speed**
- 5 Conclusions

# Benchmarks

Let's test the error monad against native exceptions:

# Benchmarks

Let's test the error monad against native exceptions:

- Simple interpreter

- $n$ -Queens problem

- Simple sets

# Benchmarks

Let's test the error monad against native exceptions:

Simple interpreter    execution time  $\times 1.12$

$n$ -Queens problem    execution time  $\times 1.24$

Simple sets            execution time  $\times 1.48$

Can we improve on this?

# Take 1: Reintroducing the wildlife

## The idea

- Keep the rich monadic interface.
- Accelerate implementation with native exceptions.
- Use reference cells to carry the rich information.

# Take 1: Reintroducing the wildlife

## The idea

- Keep the rich monadic interface.
- Accelerate implementation with native exceptions.
- Use reference cells to carry the rich information.

The *width* About 14 lines of code.

# Take 1: Reintroducing the wildlife

## The idea

- Keep the rich monadic interface.
- Accelerate implementation with native exceptions.
- Use reference cells to carry the rich information.

**The width** About 14 lines of code.

**The result** (against native exceptions)

|                     |                              |
|---------------------|------------------------------|
| Simple interpreter  | execution time $\times 1.35$ |
| $n$ -Queens problem | execution time $\times 1.75$ |
| Simple sets         | execution time $\times 2.26$ |

# Take 1: Reintroducing the wildlife

## The idea

- Keep the rich monadic interface.
- Accelerate implementation with native exceptions.
- Use reference cells to carry the rich information.

**The width** About 14 lines of code.

**The result** (against native exceptions)

|                     |                              |
|---------------------|------------------------------|
| Simple interpreter  | execution time $\times 1.35$ |
| $n$ -Queens problem | execution time $\times 1.75$ |
| Simple sets         | execution time $\times 2.26$ |

Disappointing.

## Take 2: Necromantics

### The idea

- Keep the rich monadic interface.
- Accelerate implementation with native exceptions.
- Carry the information in the native exception.
- Shortcut the OCaml type system using Phantom Types and black magic.

## Take 2: Necromantics

### The idea

- Keep the rich monadic interface.
- Accelerate implementation with native exceptions.
- Carry the information in the native exception.
- Shortcut the OCaml type system using Phantom Types and black magic. Let's say it's legal.

## Take 2: Necromantics

### The idea

- Keep the rich monadic interface.
- Accelerate implementation with native exceptions.
- Carry the information in the native exception.
- Shortcut the OCaml type system using Phantom Types and black magic. Let's say it's legal.

The *width* About 12 lines of code.

## Take 2: Necromantics

### The idea

- Keep the rich monadic interface.
- Accelerate implementation with native exceptions.
- Carry the information in the native exception.
- Shortcut the OCaml type system using Phantom Types and black magic. Let's say it's legal.

**The width** About 12 lines of code.

**The result** (against native exceptions)

|                     |                              |
|---------------------|------------------------------|
| Simple interpreter  | execution time $\times 1.35$ |
| $n$ -Queens problem | execution time $\times 1.73$ |
| Simple sets         | execution time $\times 2.22$ |

## Take 2: Necromantics

### The idea

- Keep the rich monadic interface.
- Accelerate implementation with native exceptions.
- Carry the information in the native exception.
- Shortcut the OCaml type system using Phantom Types and black magic. Let's say it's legal.

**The width** About 12 lines of code.

**The result** (against native exceptions)

|                     |                              |
|---------------------|------------------------------|
| Simple interpreter  | execution time $\times 1.35$ |
| $n$ -Queens problem | execution time $\times 1.73$ |
| Simple sets         | execution time $\times 2.22$ |

Not better.

## Take 3: Higher Black Magic

### The idea

- Keep the phantom black magic.
- Lift the monad onto the compiler.

## Take 3: Higher Black Magic

### The idea

- Keep the phantom black magic.
- Lift the monad onto the compiler.

**The width** About 22 specific + 50 generic lines of code.

## Take 3: Higher Black Magic

### The idea

- Keep the phantom black magic.
- Lift the monad onto the compiler.

**The width** About 22 specific + 50 generic lines of code.

**The result** (against native exceptions)

|                     |                              |
|---------------------|------------------------------|
| Simple interpreter  | execution time $\times 1.13$ |
| $n$ -Queens problem | execution time $\times 1.18$ |
| Simple sets         | execution time $\times 1.34$ |

## Take 3: Higher Black Magic

### The idea

- Keep the phantom black magic.
- Lift the monad onto the compiler.

**The width** About 22 specific + 50 generic lines of code.

**The result** (against native exceptions)

|                     |                              |
|---------------------|------------------------------|
| Simple interpreter  | execution time $\times 1.13$ |
| $n$ -Queens problem | execution time $\times 1.18$ |
| Simple sets         | execution time $\times 1.34$ |

Now, that's better.

## Take 4: Compiler monads

### The idea

- Return to the original error monad.
- Lift the monad onto the compiler.

## Take 4: Compiler monads

### The idea

- Return to the original error monad.
- Lift the monad onto the compiler.

**The width** About 24 specific + the same 50 lines.

## Take 4: Compiler monads

### The idea

- Return to the original error monad.
- Lift the monad onto the compiler.

**The width** About 24 specific + the same 50 lines.

**The result** (against native exceptions)

|                     |                              |
|---------------------|------------------------------|
| Simple interpreter  | execution time $\times 1.07$ |
| $n$ -Queens problem | execution time $\times 1.07$ |
| Simple sets         | execution time $\times 1.48$ |

## Take 4: Compiler monads

### The idea

- Return to the original error monad.
- Lift the monad onto the compiler.

**The width** About 24 specific + the same 50 lines.

**The result** (against native exceptions)

|                     |                              |
|---------------------|------------------------------|
| Simple interpreter  | execution time $\times 1.07$ |
| $n$ -Queens problem | execution time $\times 1.07$ |
| Simple sets         | execution time $\times 1.48$ |

Yeah! (mostly)

## Bottom line

Native exceptions

Black magic

Staged computation

## Bottom line

Native exceptions Remain the fastest goto.

Black magic

Staged computation

## Bottom line

Native exceptions Remain the fastest goto.

Black magic Provides an interesting exercise but didn't work.

Staged computation

## Bottom line

Native exceptions Remain the fastest goto.

Black magic Provides an interesting exercise but didn't work.

Staged computation Works better.

- 1 Introduction
- 2 The error monad
- 3 Representing errors
- 4 The need for speed
- 5 Conclusions**

# Summary

## Monads

- + Polymorphic variants
  - + Syntactic sugar
  - + Staged computation
-

# Summary

## Monads

- + Polymorphic variants
  - + Syntactic sugar
  - + Staged computation
- 
- Failure management

# Summary

## Monads

- + Polymorphic variants
- + Syntactic sugar
- + Staged computation

---

- + Failure management
- + Rich informations

# Summary

## Monads

- + Polymorphic variants
- + Syntactic sugar
- + Staged computation

---

- + Failure management
- + Rich informations
- + Coverage check

# Summary

## Monads

- + Polymorphic variants
- + Syntactic sugar
- + Staged computation

---

- + Failure management
- + Rich informations
- + Coverage check
- + Transparent union of failures

# Summary

## Monads

- + Polymorphic variants
- + Syntactic sugar
- + Staged computation

---

- + Failure management
- + Rich informations
- + Coverage check
- + Transparent union of failures
- + Exception hierarchies

# Summary

## Monads

- + Polymorphic variants
- + Syntactic sugar
- + Staged computation

---

- + Failure management
- + Rich informations
- + Coverage check
- + Transparent union of failures
- + Exception hierarchies
- + Local exceptions

# Summary

## Monads

- + Polymorphic variants
- + Syntactic sugar
- + Staged computation

---

- + Failure management
- + Rich informations
- + Coverage check
- + Transparent union of failures
- + Exception hierarchies
- + Local exceptions
- + Readable syntax

# Summary

## Monads

- + Polymorphic variants
- + Syntactic sugar
- + Staged computation

---

- + Failure management
- + Rich informations
- + Coverage check
- + Transparent union of failures
- + Exception hierarchies
- + Local exceptions
- + Readable syntax
- + Acceptably fast

# Summary

## Monads

- + Polymorphic variants
- + Syntactic sugar
- + Staged computation

---

- + Failure management
- + Rich informations
- + Coverage check
- + Transparent union of failures
- + Exception hierarchies
- + Local exceptions
- + Readable syntax
- + Acceptably fast
- ⊖ No nice partial catch

## Future work

**Staged computation** Application to other monads and monad transformers.

**Type system** How can we remove cases from a polymorphic variant type?

## Future work

**Staged computation** Application to other monads and monad transformers.

**Type system** How can we remove cases from a polymorphic variant type?

**Effects** Hey, we have (hierarchical) effects for free, don't we?

Thank you

Questions?

# Benchmarks (1)

| Ad-hoc error management |           |        |       |
|-------------------------|-----------|--------|-------|
|                         | Evaluator | Queens | Union |
| Very good               | 56%       | 40 %   | 18%   |
| Good                    | 26%       | 60 %   | 43%   |
| Acceptable              | 12%       | 0 %    | 35%   |
| Slow                    | 3%        | 0 %    | 4%    |
| Bad                     | 3%        | 0 %    | 0%    |
| Average                 | 1.06      | 1.05   | 1.13  |
| Deviation               | 0.12      | 0.04   | 0.10  |
| Native exceptions       |           |        |       |
|                         | Evaluator | Queens | Union |
| Very good               | 70%       | 100%   | 100%  |
| Good                    | 16%       | 0 %    | 0%    |
| Acceptable              | 12%       | 0 %    | 0%    |
| Slow                    | 2%        | 0 %    | 0%    |
| Bad                     | 0%        | 0 %    | 0%    |
| Average                 | 1.06      | 1.00   | 1.00  |
| Deviation               | 0.13      | 0.00   | 0.00  |
| Error monad             |           |        |       |
|                         | Evaluator | Queens | Union |
| Very good               | 37%       | 0 %    | 0%    |
| Good                    | 35%       | 20 %   | 0%    |
| Acceptable              | 18%       | 60 %   | 14%   |
| Slow                    | 7%        | 20 %   | 71%   |
| Bad                     | 3%        | 0 %    | 15%   |
| Average                 | 1.12      | 1.24   | 1.48  |
| Deviation               | 0.14      | 0.02   | 0.14  |

# Benchmarks (2)

| Reference and native exceptions         |           |        |       |
|---|-----------|--------|-------|
|   | Evaluator | Queens | Union |
| Very good                               | 0%        | 0 %    | 0%    |
| Good                                    | 7%        | 0 %    | 0%    |
| Acceptable                              | 33%       | 0 %    | 0%    |
| Slow                                    | 41%       | 0 %    | 0%    |
| Bad                                     | 19%       | 100%   | 100%  |
| Average                                 | 1.35      | 1.75   | 2.26  |
| Deviation                               | 0.20      | 0.06   | 0.23  |
| Phantom types and native exceptions     |           |        |       |
|   | Evaluator | Queens | Union |
| Very good                               | 1%        | 0 %    | 0%    |
| Good                                    | 8%        | 0 %    | 0%    |
| Acceptable                              | 39%       | 0 %    | 0%    |
| Slow                                    | 35%       | 0 %    | 0%    |
| Bad                                     | 17%       | 100%   | 100%  |
| Average                                 | 1.35      | 1.73   | 2.22  |
| Deviation                               | 0.22      | 0.06   | 0.22  |
| Phantom types, exceptions and rewriting |           |        |       |
|   | Evaluator | Queens | Union |
| Very good                               | 40%       | 0 %    | 0%    |
| Good                                    | 34%       | 20 %   | 3%    |
| Acceptable                              | 17%       | 80 %   | 36%   |
| Slow                                    | 7%        | 0 %    | 52%   |
| Bad                                     | 3%        | 0 %    | 9%    |
| Average                                 | 1.13      | 1.18   | 1.34  |
| Deviation                               | 0.17      | 0.03   | 0.14  |

## Benchmarks (3)

| Original error monad and rewriting |           |        |       |
|------------------------------------|-----------|--------|-------|
|                                    | Evaluator | Queens | Union |
| Very good                          | 54%       | 0%     | 0%    |
| Good                               | 28%       | 100%   | 0%    |
| Acceptable                         | 12%       | %      | 5%    |
| Slow                               | 5%        | 0%     | 56%   |
| Bad                                | 1%        | 0%     | 38%   |
| Average                            | 1.07      | 1.07   | 1.48  |
| Deviation                          | 0.15      | 0.01   | 0.14  |

# Final monad

```
type ( $\alpha$ ,  $\beta$ ) exc = {
  content :  $\alpha$ ;
  sub     :  $\beta$  option ;
} constraint  $\beta$  = [> ]

type ( $\alpha$ ,  $\beta$ ) result =
| Ok      of ( $\alpha$ )
| Error  of ( $\beta$ )

let return x = Ok x
let throw f = Error f
let bind (m:( $\alpha$ ,  $\beta$ ) result) (k: $\alpha$   $\rightarrow$  ( $\gamma$ ,  $\beta$ ) result) :
  ( $\gamma$ ,  $\beta$ ) result = match m with
| Ok    x  $\rightarrow$  k x
| Error _ as w  $\rightarrow$  w

let attempt m ~catch =
  match m with
| Error e  $\rightarrow$  catch e
| Ok    x  $\rightarrow$  x
```

## Rewriter

```
value rewrite_bind _loc ~eval:m p ~fail ~cnt:e =
  let _'err      = fresh_type _loc
  let type_of_m = result_type _loc <:ctyp<_>> _'err
  and type_of_e = result_type _loc <:ctyp<_>> _'err in
  match fail with
  [ None →
    <:expr<match ($m:$type_of_m) with
    [ (Error _) as err → (err : $type_of_e)
    | Ok    $p$ → $e$      ]>>
  | Some f →
    <:expr<match ($m:$type_of_m) with
    [ Ok    $p$      → $e$
    | (Error _) as err → (err : $type_of_e)
    | _          → $f$ ]>>];
```