Catch me if you can

Towards type-safe, hierarchical, lightweight, polymorphic and efficient error management in OCaml

David Teller
LIFO, Université d'Orléans
David.Teller@univ-orleans.fr

Arnaud Spiwack
LIX, École Polytechnique
Arnaud.Spiwack@lix.polytechnique.fr

Till Varoquaux
till@pps.jussieu.fr

Keywords exceptions, dynamic errors, coverage check, polymorphic variants, monads, typing, subtyping, syntactic sugar, code optimization, phantom types, code rewriting, code generation, ocaml, camlp4, tutorial

Abstract

This is the year 2008 and ML-style exceptions are everywhere. Most modern languages, whether academic or industrial, feature some variant of this mechanism. Languages such as Java even have a degree of out-of-the-box static coverage-checking for such exceptions, which is currently not available for ML languages, at least not without resorting to external tools.

In this document, we demonstrate a design principle and a tiny library for managing errors in a functional manner, with static coverage-checking, automatically-inferred, structurally typed and hierarchical exceptional cases, all of this for what we believe is a reasonable run-time cost. Our work is based on OCaml and features simple uses of higher-order programming, low-level exceptions, phantom types, polymorphic variants and compile-time code rewriting.

1. Introduction

Despite our best intentions and precautions, even correct programs may fail. The disk may be full, the password provided by the user may be wrong or the expression whose result the user wants plotted may end up in a division by zero. Indeed, management of dynamic errors and other exceptional circumstances inside programs is a problem nearly as old as programming. This management should be powerful enough to cover all possible situations and flexible enough to let the programmer concentrate on whichever cases are his responsibility while letting other modules handle other cases, it should be sufficiently noninvasive so as to let the programmer concentrate on the main path of execution while providing guarantees that exceptional circumstances will not remain unmanaged, all without compromising performance or violating the paradigm.

Nowadays, most programming languages feature a mechanism based on (or similar to) the notion of *exceptions*, pioneered by PL/I [14], usually with the semantics later introduced in ML [20].

The 2008 ACM SIGPLAN Workshop on ML 22-24 September 2008, Victoria, BC, Canada

A few languages, such as Haskell, define this as libraries [27], while most make this a language primitive, either because the language is not powerful enough, for the sake of performance, to add sensible debugging information, or as a manner of sharing a common mechanism for programmer errors and manageable errors.

As a support for our discussion on the management of errors and exceptional circumstances, let us introduce the following type for the representation of arithmetic expressions, written in OCaml:

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

The implementation of an evaluator for this type is a trivial task:

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

However, as such, the interpreter fails to take into account the possibility of division by zero. In order to manage this *exceptional circumstance* (or *error*), we promptly need to rewrite the code into something more complex:

Listing 1. Ad-hoc error management

```
type (α, β) result =
 | Ok    of α
 | Error of β

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')
 )
```

```
(*val eval: expr →
   (float, string) result*)
```

While this function succeeds in managing exceptional cases, the code is clumsy and possibly slow. An alternative is to use the built-in mechanism of exceptions – which we will refer to as "native exceptions"– as follows:

Listing 2. Error management with native exceptions

```
exception Error of string

let rec eval = function
 | Value f      → f
 | Div   (x, y) →
   let x' = eval x in
   let y' = eval y in
     if y' = 0. then
       raise (Error "division by zero")
     else x' /. y'
 | Add   (x, y) → eval x +. eval y
(*val eval: expr → float *)
```

This definition of `eval` is easier to write and read, closer to the mathematical definition of arithmetic operations and faster. While native exceptions appear to be a great win over explicitly returning an error value, they however are also arguably both less flexible and less safe. The lack of flexibility is due to the fact that the type associated with an exception constructor must be fully determined during the declaration of the constructor, i.e. without any polymorphic type parameter. As for the lack of safety, it is a consequence of `eval` containing no information which may allow us to determine that the function may fail and what kind of information may accompany the failure. Worse than that: the compiler itself does not have such information and cannot provide guarantees that every exceptional case will eventually be managed. Arguably, the possibility for a native exception to completely escape is comparable to the possibility for a pattern-matching to go wrong, which in turn is comparable to null-pointer exceptions in most modern industrial languages – while technically not a type error, this remains a source of unsafety which we will refer to as "incomplete coverage" in the rest of this document.

Now, determining statically the set of native exceptions which may be raised by an expression is a possibility, just as it is possible to use this information to guarantee complete coverage. Indeed, the Java compiler [1] uses exception-specific type annotations for this purpose, while OCaml has long benefited from OCamlExc [22], a variation on *types and effects* [25], dedicated to inferring and type-checking exceptions and their coverage. Unfortunately, this last tool, as many of its counterparts in other languages, is neither integrated in the toolchain, nor maintained. In addition, even when complete coverage has been proved, the lack of flexibility remains: ML languages do not allow any usable form of parametric polymorphism in native exceptions[1]. This is quite unfortunate, as numerous situations thus require the manual definition of many superfluous exceptions with identical semantics but different types, or lead to the overuse of magic constants to specify sub-exceptions, or require impure or unsafe hacks to implement simple features. While SML provides a way to regain some polymorphism in exceptions with *generative exceptions*, even this feature only manages to provide local polymorphism.

[1] For comparison, the Java type-checker rejects subtypes of `Exception` with parametric polymorphism, the C# parser rejects catch clauses with parametric polymorphism, while Scala accepts defining, throwing and catching exceptions with parametric polymorphism, but the semantics of the language ignores these type parameters both during compilation and during execution.

Another possible approach which may be used to obtain both the readability of exceptions, guarantee of complete coverage and parametric polymorphism, is to implement exceptions as monads [27], a path followed by Haskell. However, this approach often results in either not-quite-readable and possibly ambiguous type combinations consisting in large hierarchies of algebraic combinators, in the necessity of writing custom error monads or monad transformers, which need to be manually rewritten as often as the list of exceptional cases changes, or in the use of dynamic types. In addition, these monads are typically perceived as having a large computational cost, due to constant thunking and dethunking of continuations and to the lack of compiler-optimized stack unrolling.

In this document, we attempt to obtain the best of both worlds: polymorphism, type-safety, coverage-check, with the added benefits of automatic inference of error cases and the definition of classes and subclasses of exceptional cases, all of this without the need to modify the programming language. As this is an OCaml-based work, we also take into account the impact of this programming style in terms of both performances, syntactic sugar and possible compile-time optimizations. Despite the number of claims appearing in the previous sentences, our work is actually based on very simple concepts and does not have the ambition of introducing brand new programming methodologies, nor to revolutionize ML programming. Rather, our more limited objective is to present an interesting design principle and a tiny library for error management, in the spirit of Functional Pearls or of a tutorial, and based on ML-style exceptions, monads, phantom types, polymorphic variants and code rewriting. Some of our results are positive, some negative and, somewhere along the way, we revisit results discarded by previous works on hierarchical exceptions [19] and demonstrate that, when redefined with proper language support, they may be used to provide safer (and possibly faster) results.

In a first section we demonstrate briefly an error monad, well-known in the world of Haskell but perhaps less common in the ML family. We then proceed to complete this monad with the use of lightweight types to achieve automatic inference of error cases, before pushing farther these lightweight types to permit the representation of classes and subclasses of exceptional cases, while keeping guarantees of coverage. Once this is done, we study the performance of this monad, explore possible optimizations, some real and some only imaginary, and progressively move the code from the library to the compiler. Finally, we conclude by a discussion on usability, potential improvements, and comparison with other related works.

The source code for the library is available as a downloadable package [26].

2. The error monad

As we discussed already, listing 1 shows a rather clumsy manner of managing manually whichever errors may happen during the evaluation of an arithmetic exception. However, after a cursory examination of this extract, we may notice that much of the clumsiness may be factored away by adding an operation to check whether the result of an expression is `Ok` x, proceed with x if so and abort the operation otherwise. Indeed in the world of monads [27], this is the *binding* operation. In OCaml, this function is typically hidden behind syntactic sugar [15] `perform` and $\longleftarrow$, as follows

Listing 3. Towards monadic error management

```
let bind m k = match m with
   Ok x     → k x
 | Error _  → m
(*val bind: (α, β) result →
 (α → (γ, β) result) →
```

```
        (γ, β) result*)

let rec eval = function
 | Value f      → Ok f
 | Div   (x, y) → perform
    x' ⟵ eval x ;
    y' ⟵ eval y ;
    if y' = 0. then Error "Division by 0"
    else Ok (x' /. y')
 | Add   (x, y) → perform
    x' ⟵ eval x ;
    y' ⟵ eval y ;
    Ok (x' /. y')
(*val eval: expr →
   (float, string) result*)
```

For the sake of abstraction (and future changes of implementation), we also hide the implementation of type `('a, 'b) result` behind a `return` function (for successes) and a `throw` function (for failures):

Listing 4. Monadic error management

```
let return   x = Ok x
(*val return: α → (α, ε) result*)

let throw    x = Error x
(*val throw : ε → (α, ε) result*)

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

This new definition of `eval` is arguably as easy to read as the version of listing 2. As we have decided to abstract away type `result`, we need one last function to be able to run a computation and determine whether it succeeded or failed. As this function will be used essentially in place of `try...with` (or `try...catch` in many languages), we call it `attempt` and implement it as:

Listing 5. Entering/leaving the error monad

```
let attempt f arg ~catch =
 match f arg with
  | Ok    x → x
  | Error x → catch x
(*val attempt: (α → (β, γ) result) →
   α → catch:(γ → β) → β)*)
```

We may now group all the functions of the error monad as one module `Error_monad` with the following signature:

Listing 6. A module for the error monad

```
type (α, β) result
val return: α → (α, β) result
val throw : β → (α, β) result
val bind  : (α, β) result →
 (α → (γ, β) result) →
 (γ, β) result
val attempt: (α → (β, γ) result) →
   α → catch:(γ → β) → β
```

Before moving on to the next section, let us demonstrate the capability of the error monad to write functions with polymorphic exceptions, *i.e.* functions which can throw exceptions which depends on the type of their arguments. To that purpose we give a toy implementation of (persistent) association lists whose signature contains two functions: `find` to retrieve the value associated to a key, and `add` to add an association to the list. `find k l` fails when nothing is associated to `k` in `l`. `add k u l` fails when there is already a value `v` associated to `k` in `l`. In both case the key `k` is used as the error report.

We give the implementation of this module in listing 7, together with its signature in listing 8.

Listing 7. Association list with polymorphic exceptions

```
type (α,β) assoc = (α*β) list

let empty = []

let rec add k u = function
 | []            → return [k,u]
 | (x,v as a)::l →
   if k=x then throw k
   else        perform
    l' ⟵ add k u l ;
    return (a::l')

let rec find k = function
 | []       → throw k
 | (x,v)::l →
   if k=x then return v
   else        find k l
```

Listing 8. Association list signature

```
type (α,β) assoc
val empty: (α,β) assoc
val add  : α → β → (α,β) assoc →
          ((α,β) assoc, α) result
val find : α → (α,β) assoc →
           (β,α) result
```

In the rest of the paper, we will concentrate on the `eval` example, however, this example can be refined similarly.

3. Representing errors

While listing 4 presents a code much more usable than that of listing 1 and while this listing is type-safe, the awful truth is that this safety hides a fragility, due to the use of "magic" character strings to represent the nature of errors – here, `"Division by 0"`, a constant which the type-checker cannot take into account when attempting to guarantee coverage. Unfortunately, this fragility is shared by elements of both OCaml's, SML's or Haskell's standard libraries.

Now, of course, it may happen that we need to represent several possible errors cases, along with some context data. For instance, during the evaluation of simple arithmetic exceptions, in addition to divisions by zero, arithmetic overflow errors could arise. For debugging purposes, we may even decide that each error should be accompanied by the detail of the expression which caused the error and that overflows should be split between overflows during addition and overflows during division. To represent all this, numerous type-safe options are available.

3.1 Errors as heavy-weight sums

The first and most obvious choice is to represent errors as sum types. For our running example, we could write

Listing 9. Simple arithmetic errors

```
type cause_of_overflow =
| Addition
| Division

type eval_error =
| Division_by_zero of expr
| Overflow         of expr *
                      cause_of_overflow
```

Now, as our error monad lets us transmit polymorphic error information along with the error itself, we may rewrite `eval` so as to take advantage of `eval_error` instead of `string`, without having to declare a new exception constructor or to rewrite the interface or implementation of the error monad:

Listing 10. Monadic error management with sum types

```
let ensure_finite f e message =
  if f = infinity ∨ f = neg_infinity then
    throw (Overflow(e, message))
  else return f

let rec eval e = match e with
 | Value f       → return f
 | Div   (x, y)  → perform
    x' ⟵ eval x ;
    y' ⟵ eval y ;
    if y' = 0. then
         throw (Division_by_zero e)
    else
      ensure_finite (x' /. y') e Division
 | Add   (x, y)   → perform
    x' ⟵ eval x ;
    y' ⟵ eval y ;
      ensure_finite (x' +. y') e Addition
(*val eval: expr →
   (float, eval_error) result*)
```

While this solution improves on the original situation, it is not fully satisfying. Indeed, it is quite common to have several functions share some error cases but not all. For instance, let us assume the development of both a basic visual 10-digits calculator and a scientific plotter, based on a common arithmetic library. Both evaluators use division and may suffer from divisions by zero. However, only the scientific plotter defines logarithm and may thus suffer from logarithm-related errors.

Should the error-reporting mechanism of the library be defined as one heavy-weight sum type, the visual calculator will need to be able to handle all the same error cases as the scientific plotter. OCaml's built-in pattern-matching coverage test will therefore require all error cases to be managed, even though the functions which may trigger these error cases are never invoked by the visual calculator.

The alternative is to use disjoint possible errors for distinct functions. However, this choice quickly leads to composability nightmares. Since a division by zero and a logarithm-error are members of two disjoint types, they need to be injected manually into a type `division_by_zero_or_log_error`, defined as a sum type, for use by the scientific plotter. While possible, this solution is cumbersome to generalize and tends to scale very poorly for large projects, especially during a prototyping phase. This composability nightmare also appears as soon as two different libraries use disjoint types to represent errors: arithmetic errors, disk errors or interface toolkit errors, for instance, must then be injected into an awkward common type of errors, and projected back towards smaller types of errors as need arises.

3.2 Errors as lightweight composition of sums

Another approach, commonly seen in the Haskell world, and actually not very different from the second choice just mentioned, is to define a more general type along the lines of

Listing 11. Haskell-style `either` type

```
type (α, β) either =
  Left  of α
| Right of β
```

With such a data structure, building lightweight compositions of error cases becomes a trivial task. However, these lightweight compositions are also an easy recipe for obtaining unreadable constructions consisting in trees of `either` and tuples. That is, attempting to convert `eval` to use only such lightweight types quickly results in the following expression, with its somewhat caricatural type:

Listing 12. Monadic error management with lightweight `either`

```
let ensure_finite f message =
  if f = infinity ∨ f = neg_infinity then
    throw message
  else return f

let rec eval e = match e with
 | Value f       → return f
 | Div   (x, y)  → perform
    x' ⟵ eval x ;
    y' ⟵ eval y ;
    if y' = 0. then throw (Right e)
    else
      ensure_finite (x' /. y')
         (Left (Left e))
 | Add    (x, y)  → perform
    x' ⟵ eval x ;
    y' ⟵ eval y ;
      ensure_finite (x' +. y')
         (Left (Right e))
(* val eval : expr →
 ( float,
   ( (expr, expr) either,
      expr
   ) either
 ) result *)
```

While it is possible to avoid such chains of `either` by combining this approach with the manual definition of new types – perhaps abstracted away behind modules – the result remains unsatisfactory in terms of comprehension and falls far from solving the composability nightmare.

3.3 Errors as extensible types

Another alternative would be the use of *extensible types*, as featured in Alice ML [23]. More generally, one such type is available in languages of the ML family: native exceptions. Instead of our current type `eval_error`, and with the same code of `eval`, we could therefore define two native exceptions

```
exception Division_by_zero of expr
exception Overflow         of expr *
   cause_of_overflow
```

If, at a later stage, the set of exceptions needs to be extended to take into account, say, logarithm errors, a one-liner suffices to extend the definition of errors:

```
exception Logarithm_error  of expr
```

Better even, this solution proves compatible with the existing native exception system and permits trivial conversion of native exceptions for use with the error monad:

```
let attempt_legacy f arg ~catch =
 try   f arg
 with  e → catch e
(*val attempt_legacy: (α → β) →
   α → catch:(exn → β) → β *)
```

At this point, a first weakness appears: while the addition of brand new error cases such as Logarithm_error is a trivial task, extending cause_of_overflow is impossible unless we find a way to define cause_of_overflow as an extensible type. Assuming that we have a way to express several distinct extensible types, perhaps by using an hypothetical encoding with phantom types, we are still faced with a dilemma: should all errors be represented by items of the same type exn or should we use several disjoint extendable types? The question may sound familiar, as we have already been faced with the same alternative in the case of heavy-weight sum types. As it turns out, and for the same reasons, neither choice is acceptable: sharing one type gets into the way of coverage guarantees, while splitting into several types leads, again, to composability nightmares.

Or does it? After all, OCaml does contain an additional kind of types, close cousin to extensible sum types, but with much better flexibility: Polymorphic Variants [10].

3.4 Errors as polymorphic variant

Polymorphic variants represent a kind of lightweight sum types designed to maximize flexibility. Indeed, the main traits of polymorphic variants are that

- they do not need to be declared before being used – rather, their definition is inferred from their usage
- they may be declared *a posteriori* if necessary, for specification and documentation purposes
- unless some constructor is used in two contradictory fashions, two open polymorphic variants may be composed, without any declaration, into in a larger polymorphic variant containing all the constructors of each of the smaller variants
- the same constructor may appear in several otherwise unrelated polymorphic variants.

When used to represent errors, this first trait will let us concentrate on the task of building the algorithm, without having to write down the exact set of errors before the prototyping phase is over. The second trait will prove useful at the end of the prototyping phase, to improve error-checking of client code and documentation. The third trait will let OCaml infer automatically the set of errors which may be triggered by an expression – and check completeness of the error coverage, just as it would do for heavy-weight sum types. Finally, the last trait will let us define functions algorithms which may share some – but not necessarily all – error cases.

Before rewriting the full implementation of eval, let us build a smaller example. The following extract defines an expression e with type expr and another expression div_by_zero which may throw a division by zero with information e:

```
let e = Value 0.  (*For testing*)
let div_by_zero =
    throw (`Division_by_zero e)
(* val div_by_zero : ('_a,
    _[> `Division_by_zero of expr ]
) result *)
```

The type of div_by_zero mentions that it may have any result _'a, much like raising ML-exceptions produce results of type 'a, and that it may throw an error consisting in an *open* variant, marked by constructor `Division_by_zero, and containing an expr. Note that neither type _'a nor type _[> `Division_by_zero of expr] is generalisable, which is consistent with the fact that div_by_zero is a constant.

Similarly, we may define an expression overflow_div, with the ability to cause an overflow during division, much as we could with the heavy-weight sum type:

```
let overflow_div =
    throw (`Overflow (`Division e))
(* val overflow_div : ('_a,
    _[> `Overflow of
      _[> `Division of expr ] ]
  ) result *)
```

Finally, the division by zero and the overflow during division may be composed, resulting in:

```
if true then div_by_zero
else         overflow_div
(* ('_a,
     _[> `Division_by_zero of expr
       |  `Overflow of
           _[> `Division of expr ] ]
  ) result *)
```

As we see from the inferred type of this expression, the result of the composition may produce either results of any type or errors marked either by `Division_by_zero (and accompanied by an expr) or by `Overflow (and accompanied by another tag `Division, itself accompanied by an expr). This error signature remains *open*, which allows us to add further error cases.

As we may see on the following listing, conversion to polymorphic variants is a straightforward task:

Listing 13. Monadic error management with polymorphic variants

```
let rec eval e = match e with
 | Value f      → return f
 | Div   (x, y) → perform
    x' ←— eval x ;
    y' ←— eval y ;
    if y' = 0. then
      throw (`Division_by_zero e)
    else
      ensure_finite (x' /. y')
       (`Overflow (`Division e))
 | Add    (x, y) → perform
    x' ←— eval x ;
    y' ←— eval y ;
      ensure_finite (x' +. y')
       (`Overflow (`Addition e))
(*val eval :
 expr →
 ( float,
   [> `Division_by_zero of expr
   |  `Overflow of
      [> `Addition of expr
      |  `Division of expr ] ]
 ) result *)
```

As we hoped, with polymorphic variants, we do not have to manually label error cases. Rather, the compiler may infer error cases from the source code.

Further, as this inferred information appears in the type of our function, coverage may be proved by the type-checker. Therefore, we may write:

```
let test1 e = attempt eval e ~catch:(
 function ‘Division_by_zero _ →
      print_string "Division by 0"; 0.
 |        ‘Overflow _          →
      print_string "Overflow"; 0. )
(*val test : expr → float*)
```

On the other hand, the following extract fails to compile:

```
# let test2 e = attempt eval e ~catch:(
  function ‘Overflow _ →
      print_string "Overflow"; 0. )

  function ‘Overflow _ →
           ^^^^^^^^^^^^
This pattern matches values of type
  [< ‘Overflow of α ]
but is here used to match values of type
  [> ‘Division_by_zero of expr
  |  ‘Overflow of [> ‘Addition of expr
                   |  ‘Division of expr ] ]
```

In addition, the composability of polymorphic variants, which we have demonstrated, means that we do not have to decide whether to put all the error cases defined by a library in one common type or to split them among several disjoint types: barring any conflicting name or any specification which we may decide to add to prevent composition, there is no difference between one large open polymorphic variant type and the automatically inferred union of several smaller ones.

As we will see, polymorphic variants, along with a little syntactic sugar, may carry us farther than usual ML-style exceptions.

3.5 From polymorphic variants to exception hierarchies

We have just demonstrated how polymorphic variants solve the problem of composing error cases. Truth be told, our example shows a little bit more: we have not only defined two kinds of errors (divisions by zero and overflows), we have also defined two sub-cases of errors (overflow due to addition and overflow due to division).

Passing the right parameters to function `attempt`, we may choose to consider all overflows at once, as we have done in our latest examples, or we may prefer to differentiate subcases of overflows:

Listing 14. Matching cases and subcases

```
let test3 e = attempt eval e ~catch:(
 function ‘Division_by_zero _    →
  print_string "Division by 0"; 0.
 | ‘Overflow ‘Addition _         →
  print_string "Overflowing addition"; 0.
 | ‘Overflow _                   →
  print_string "Other overflow"; 0.
)
```

In other words, while we have chosen to present sub-cases as additional information carried by the error, we could just as well have decided to consider them elements of a small hierarchy:

- division by zero is a class of errors ;
- overflow is a class of errors ;
- overflow through addition is a class of overflows ;
- overflow through division is a class of overflows.

From this observation, let us try and derive a general notion of classes of errors – all of this without compromising the composability and coverage checking allowed by polymorphic variants.

Before we proceed, we need to decide exactly what an exception class should be. If it is to have any use at all, it should be possible to determine if an exception belongs to a given class by simple pattern-matching. In order to preserve our results and the features used up to this point, an exception class should be a data structure, defined by one or more polymorphic variant constructors and their associated signature, as well as some error content. In addition, for exception classes to be useful, it must be possible to specify a subtyping relation between classes. We also need to ensure consistency between the error content of classes related by subtyping. Finally, we should be able to define new classes and subclasses without having to modify the definition of existing code.

To achieve all this, we encode classes using a method comparable to tail polymorphism [3] with polymorphic variants[2]. Where classical uses of tail polymorphism take advantage of either algebraic data-types or records, though, the use of polymorphic variants preserves extensibility.

We first introduce a chain-link record, whose sole use is to provide human-readable field names `sub` and `content`. Field `sub` is used to link a class to its super-class, while field `content` serves to record the class-specific additional error information which the programmer wishes to return along with the error:

```
type (α, β) ex = {
  content: α;
  sub:     β option;
} constraint β = [> ]
```

Once we have a chain-link container, and assuming for the course of this example that division by zero is a top-level class of exceptions, we may produce the following constructor:

```
let division_by_zero_exc ?sub content =
 ‘Division_by_zero {
    content = content;
    sub     = sub;
 }
(*val ?sub:([>  ] as α) → β →
   [> ‘Division_by_zero of (β, α) ex]*)
```

Argument `content` is self-documented, while argument `sub` will serve for subclasses to register the link. Similarly, we may now define overflow:

```
let overflow_exc ?sub content =
 ‘Overflow {
    content = content;
    sub     = sub;
 }
```

Since we decided to describe overflow during addition as a subclass of overflow, we may define its constructor by chaining a call to `overflow_exc`, passing the new chain-link as argument.

```
let overflow_addition ?sub content =
 overflow_exc ~sub:(‘Addition {
    content = ();
    sub     = sub;
 }) content
```

Or, equivalently, with a small piece of syntactic sugar introduced for this purpose:

[2] A similar idea has been suggested in the context of Haskell [19] but discarded as a “very interesting, but academic” and a “failed alternative”.

```
let exception overflow_division content =
    Overflow content; Division ()
```

The changes to the library are complete. Indeed, one simple record type is sufficient to move from polymorphic variants to polymorphic variants with hierarchies. To confirm our claim that we preserve composability and coverage guarantees, let us revisit `eval` and our test cases.

Adapting `eval` to our hierarchy is just the matter of replacing concrete type constructors with abstract constructors:

Listing 15. Eval with hierarchies

```
let rec eval e = match e with
 | Value f       → return f
 | Div   (x, y) → perform
    x' ⟵ eval x ;
    y' ⟵ eval y ;
    if y' = 0. then
      throw (division_by_zero e)
    else
      ensure_finite (x' /. y')
       (overflow_division e)
 | Add   (x, y) → perform
    x' ⟵ eval x ;
    y' ⟵ eval y ;
    ensure_finite (x' +. y')
      (overflow_addition e)
(*
val eval : expr →
  (float,
   [> ‘Division_by_zero of (expr, α) ex
   |  ‘Overflow of
     (expr,
       [> ‘Addition of (unit, β) ex
       |   ‘Division of (unit, γ) ex ])
     ex ])
  result
*)
```

While the type information is painful to read – and could benefit from some syntactic sugar – it accurately reflects the possible result of `eval`, the nature of exceptions and subexceptions and their contents.

Adapting the test of listing 14 to our more general framework, we obtain the following extract, slightly more complex:

```
let test4 e = attempt eval e ~catch:(
 function ‘Division_by_zero _    →
  print_string "Division by 0"; 0.
 | ‘Overflow {sub = Some (‘Addition _)} →
  print_string "Overflowing addition"; 0.
 | ‘Overflow _               →
  print_string "Other overflow"; 0. )
```

As a demonstration of coverage guarantees, let us write the same example, with the omission of the case of overflow division:

```
let test5 e = attempt eval e ~catch:(
 function ‘Division_by_zero _    →
  print_string "Division by 0"; 0.
 | ‘Overflow {sub = Some (‘Addition _)} →
  print_string "Overflowing addition"; 0.
) (*Oops, forgot division by zero.*)
```

The following error message demonstrates that the type-checker has correctly detected the missing case. The solution is suggested at the end of the message:

Listing 16. Missing subcase (error message)

```
   function ‘Division_by_zero _    →
            ^^^^^^^^^^^^^^^^^^
This pattern matches values of type
  [< ‘Division_by_zero of α
  |  ‘Overflow of (expr,
              [< ‘Addition of β ]) ex ]
but is here used to match values of type
 [> ‘Division_by_zero of (expr, _) ex
 |  ‘Overflow of
   (expr,
     [> ‘Addition of (unit, γ) ex
     |   ‘Division of (unit, δ) ex ])
   ex ]
The first variant type does not allow
   tag(s) ‘Division
```

Similarly, this encoding lets the type-checker spot type or tag errors in exception-matching, as well as provide warnings in case of some useless catch clauses. We do not detail the error messages, which are not any more readable than the one figuring in listing 16, and which could just as well benefit from some customized pretty-printing.

As a last step, for convenience, we introduce another layer of syntactic sugar, marked by a new keyword `attempt`, and which provides a simpler notation for exception patterns, adds an optional post-treatment for successes, introduced by `val`, and an optional post-treatment for both successes and failures, introduced by `finally`:

```
let test6 e = attempt eval e with
 | Division_by_zero _        →
   print_string "Division by zero"
 | Overflow _ ; Addition _ →
   print_string "Overflow while adding"
 | Overflow _                →
   print_string "Other overflow"
 | val f                      →
   print_float f
 | finally _                  → ()
```

3.6 Bottom line

In this section, we have examined a number of possible designs for error reports within the error monad. Some were totally unapplicable, some others were impractical. As it turns out, by using polymorphic variants, we may achieve both inference of error cases, composability of error cases and simple definition of hierarchies of error classes, while retaining the ability of the type-checker to guarantee coverage of all possible cases. All of this is implemented in a meager 27 lines of code, including the module signature.

At this point, we have obtained all the features we intended to implement. Our next step is to study the performance cost – and to minimize it, if possible.

4. The hunt for performance

According to our experience, when hearing about monads, typical users of OCaml tend to shrug and mutter something about breaking performances too much to be as useful as built-in exceptions. Is that true?

Figure 1 presents the result of a benchmark on the performance of error-management schemes in OCaml. To obtain this benchmark, we measure the execution time of three different implementations of

Ad-hoc error management (demonstrated in listing 1)			
	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 (demonstrated in listing 2)			
	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 (demonstrated in listing 4)			
	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

Figure 1. Testing the performance of the error monad

- an arithmetic evaluator – errors are rather uncommon, being raised only in case of division by zero (300 samples)
- the n queens problem – only one pseudo-error is raised, when a solution has been found (5 samples)
- union of applicative sets of integers – pseudo-errors are raised very often, as an optimization, to mark the fact that no change is necessary to a given set (300 samples).

Every test has been performed with OCaml 3.10.1, under Linux, on native code compiled for the 32bit x86 platform, with maximal inlining, executed 15 times, after a major cycle of garbage-collection, with the best and the worst result discarded. The results are presented as a percentage of the number of samples in which the execution time falls within given bounds:

Very good Execution time of the sample is within 5% of the execution time of the fastest implementation for this test (the "shortest execution time")

Good Within 5-15% of the shortest execution time.

Acceptable Within 15-30% of the shortest execution time.

Slow Within 30-50% of the shortest execution time.

Bad At least 50% longer than the shortest execution time.

For information, we also provide

Average Average of ratio $\frac{\text{duration of test}}{\text{shortest execution time}}$.

Deviation Standard deviation of $\frac{\text{duration of test}}{\text{shortest execution time}}$.

From these results, we may already draw the conclusion that, while using the error monad causes slowdowns in the program, these slowdowns remain reasonable whenever errors are used only exceptionally. On the other hand, with the error monad, using error cases as optimizations doesn't work.

While this last point is hardly a surprise, it may come as a disappointment for OCaml programmers, many of which have come to use exceptions as an elaborate and mostly-safe `goto`. Perhaps we can try and improve performance. The source provides two hints regarding slowdown:

1. the permanent thunkification and de-thunkification at each monadic binding, hidden behind each occurrence of the syntactic sugar $\longleftarrow$ – indeed, listing 4 contains four such local function definitions;
2. the manual unrolling of the stack hidden inside the definition of function `bind` – indeed, a call to `throw` from a depth of n inside the evaluation of an expression may require $\mathcal{O}(n)$ returns from interleaved calls of functions `bind` and `eval`.

Let us try and tackle down the second issue first.

4.1 Take 1: Going against the flow

Manual stack unrolling is slow? Perhaps we might achieve some faster result by adopting OCaml's built-in mechanism for fast stack unrolling: native exceptions. Let's see how we may re-implement the error monad with native exceptions:

Listing 17. Towards a monad with native exceptions?
```
exception Raised of string

type (α, β) result = α
let throw  (x:β) : (α, β) result =
  raise (Raised x)
let return (x:α) : (α, β) result = x
let bind m k = k m
let attempt f arg ~catch =
 try  f arg
 with Raised s → catch s
```

Of course, such an implementation has the major drawback of restricting the kind of exceptions raised to character strings. This is not only ugly, it's in complete contradiction with our objective. Unfortunately, OCaml's type system disallows the definition of exceptions with polymorphic type variables. While we might be able to obtain similar features by using functors (much as Haskell's error monads use typeclasses for the same purposes), this would be quite awkward, plus this would require defining exception modules – manually, in OCaml – for each and every possible type of information carried by the exception.

Fortunately, we can do better.

Indeed, rather than using the exception as a channel for transmitting information, we may decide to use the exception as a simple mean of unrolling the stack – and propagate the value through a different channel. For this purpose, we revert the visible flow of information. Rather than transporting the error information through layers of stack, we first decide of a convenient recipient for this error information, then start the evaluation, propagate the recipient towards the leaves of the evaluation, and check at the end of computation whether this recipient has been filled. For this purpose, we will use a reference, in a manner reminiscent of the implementation of SML-style local globally-quantified exceptions [17] in OCaml. This reference is created and collected by `attempt`:

Listing 18. Implementing the error monad with native exceptions
```
exception Raised

type (α, β) result = β option ref → α

let attempt f arg ~catch =
 let  result = ref None in
```

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

Figure 2. Testing the performance of a monad based on native exceptions and references

```
try  f arg result
with Raised →
  match !result with
    None   → assert false  (*Unused*)
  | Some e → catch e
```

The reference is then propagated by `bind`, to make sure that uses of `throw` within the same domain will share the same reference:

```
let bind m k r = k (m r) r
```

Finally, throwing an error consists in constructing the error information, placing it in the reference and then unrolling the stack:

```
let throw  x b =
  b := Some x;
  raise Raised

let return x _ = x
```

With this rewriting of the error monad, we have introduced native exceptions inside a monad with a purely functional interface. From the point of view of optimization, we have altered the implementation of the monad to have it take advantage of the built-in feature of stack unrolling. From the point of view of functional vs. imperative programming, we have domesticated the *extra-functional effect* of native exception-raising into a functional monad.

Now, how good is that optimization? Not good, as we may see on figure 2. Indeed, the performances of the resulting algorithms are uniformly worse than those of the basic error monad, whether error cases are common or rare. So perhaps the biggest problem is not with the unrolling of the stack but with the constant repetition of thunking and dethunking, actually made worse in this implementation. To put this to the test, let us try a variant strategy – one which still requires thunking and dethunking, but in a fashion which may be easier to optimize by the compiler.

4.2 Take 2: Meaningful units

In our latest implementation of `bind`, we wrote

```
let bind m k r = k (m r) r
```

Equivalently, we could have written

```
let bind m k = fun r → k (m r) r
```

As we may see, this implementation dethunkifies monad `m` and rethunkifies the result. From this implementation, the ideal optimization would be removing the need for `r`. In the absence of dynamic scoping in OCaml[3], however, this is not directly possible. At this point, it may be tempting to abandon monads and turn to arrows. Without entering the details – such a discussion would go largely beyond the scope of this paper – our experiments hint that arrow-based error-management libraries fall in two categories: those which share the same bottlenecks as the error monad, if not worse, and those which can't provide coverage check with ML-style type systems.

Now, if we can't completely remove `r`, perhaps we can make `r` so simple that the compiler will be able to optimize it away. For the sake of an experiment, let us replace occurrences of `r` with the simplest possible form of data: the unit.

Listing 19. Towards a monad with exceptions and unit ?

```
exception Raised

type (α, β) result = unit → α
        constraint β = [> ]

let throw  x () = assert false
let return x () = x
let bind m k () = k ( m () ) ()

let attempt f arg ~catch =
 try  f arg ()
 with Raised → assert false
```

While this implementation of the error monad is clearly incomplete, it is sufficient to demonstrate that the type of error cases is actually totally independent from the reference argument we just removed. Rather, the signature of `throw` and `bind`, as defined in the interface of our module, are sufficient for the type system to infer the type of error cases, regardless of whether the error information is actually transmitted – indeed, in this implementation, type parameter β of `result` behaves as a *phantom type* [6].

In other words, the implementation defined in listing 19 actually demonstrates two channels for information propagation. The first one is dynamic but is limited by the type system: exception `Raised`, in addition to unrolling the stack, may be used to carry information at run-time but can't accept polymorphic type arguments. The second one is static, has all the power of OCaml's type system but cannot convey run-time information: the type of `throw`, `return` and `bind`. Additionally, the unit argument of `throw` prevents any premature control flow and enforces an order of evaluation. Last but not least, at this stage, we are certain that the type of the error case is actually a polymorphic variant – here, materialized by `constraint` β `= [> ]`. All the ingredients are gathered for the safe projection of the error type onto some generic variant type and back onto the phantom type obtained through the static channel.

To obtain this, we do not need to change the definition of `return`, `bind` or `result`. As generic variant type, we may use the universal existential type `Obj.t`, along with the safe projection `Obj.repr :` α `-> Obj.t` and the unsafe projection `Obj.obj : Obj.t ->` α. We may now use exception `Raised` to convey the content of the error, minus its type:

```
exception Raised of Obj.t

let throw  x () =
  raise (Raised (Obj.repr x))
```

Finally, `attempt` receives both the type-less information from `raised` and the type information from the type of its arguments. We only have to put these informations together, as follows:

[3] Truth be told, an extension of OCaml exists, which provides dynamic scoping [18], either as a library or as a compiler patch. However, the implementation of the library introduces the exact same bottleneck we attempt to avoid, while the compiler patch applies only to bytecode OCaml, which makes either implementation unusable for this work.

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

Figure 3. Testing the performance of a monad based on exceptions and phantom types

```
let attempt (f:_ → (_, β) result) arg
  ~catch =
 try  f arg result ()
 with Raised r → catch (Obj.obj r : β)
```

With the exact same signature as the other implementations of the error monad and the certainty that a variant type is only projected onto a variant type, this implementation actually achieves a small type-safe extension of OCaml's type system.

Does this optimization fare better than our first attempt? The answer lies on figure 3: yes, but the difference is so minimal that it is in fact meaningless. Fortunately for us, our work is not lost. Indeed, moving a little of the code to the compiler works wonders.

4.3 Take 3: Playing with the compiler

Up to this point, all the work we have demonstrated – with the exception of the thin syntactic sugar used for simplifying the work with exception hierarchies – aimed at developing a type-safe library with an exception control flow for managing errors. Now that we seem to have hit a performance dead-end, it is time to take a step back and try and determine which parts of the code are actually meaningful and which parts are just type annotations under the guise of expressions.

Let us start with the simple functions. Can we get rid of the code of `throw`? No, we can't. In addition to its type information, `throw` performs the essential task of unrolling the stack. We may also not remove the `()`, as it is necessary to force the order of evaluation – without this unit argument, stack unrolling could take place at an inconvenient time[4]. Can we get rid of the code of `return`, then? It seems unlikely that we could get rid of the return value. The situation of `bind`, however, is different. Every occurrence of `bind` is meant to be used as

```
p ⟵ m; e
```

or equivalently

```
bind m (fun p →e)
```

where m and e are some expressions and p is a pattern. In either case, in addition to type information, this code serves to implement

```
let p = m () in e
```

In other words, the *semantics* of this section's `bind` is the same as the semantics of `let`. However, at the moment, for typing reasons, while `let` is a primitive of the language, monadic binding is more costly, requiring two thunkifications, one dethunkification and two function calls. As it turns out, assuming that we have a projection function `proj : ('a, 'b) result -> (unit -> 'a)`, we may rewrite

```
p ⟵ m; e
```

as

Listing 20. Inlined bind

```
let (p:α) = proj
  (m: (α, β) result) () in
  (e: (γ, β) result)
```

For this example, we have assumed that names `'a`, `'b` and `'c` are free in p, m and e.

Writing a function `proj` is hardly a difficult task, as the identity would fit nicely for that job:

```
external proj: (α, β) result →
  (unit → α) = "%identity"
```

However, defining this new function `proj` is probably a bad idea: in our case, the very act of breaking this abstraction is type-unsafe, because as this also violates the constraints on the phantom type. As the very act of making a function `proj` available for use at compile-time also makes that function available for invocation by the user, we prefer avoiding the issue and making use of the equally unsafe `Obj.magic`.

The only remaining question is how we may transform the ⟵ notation into the code of listing 20. For this purpose, we first extend our module with a compile-time function

```
val rewrite_bind:
 m:Ast.expr → p:Ast.patt →
 e:Ast.expr → Ast.loc  →
 Ast.expr
```

This function uses the standard library/tool Camlp4 – given a different setting, we could just as well have used MetaOCaml. The role of this function is to generate the rewritten code, as follows:

```
let fresh_type _loc =
 <:ctyp< '$fresh_name ()$>>
let result _loc res err =
 <:ctyp< ($res$, $err$) result>>

let rewrite_bind ~m ~p ~e _loc =
 let _α    = fresh_type _loc
 and _β    = fresh_type _loc
 and _β    = fresh_type _loc in
 let type_of_m = result _loc _α _β
 and type_of_e = result _loc _γ _β
 and abst  = <:ctyp<unit → $_α$ >> in
  <:expr< let $p$ = (Obj.magic
            ($m$:$type_of_m$) :
             $abst$) () in
            ($e$:$type_of_e$) >>
```

Once this is done, remains the task of adapting syntactic sugar `perform`/⟵ to take advantage of `rewrite_bind` when appropriate – that is, in our current implementation, whenever a compile-time rewriter has been registered. This is the matter of a few dozen lines of code which we will not detail in this document.

Once the transformation is complete, we obtain the results shown in figure 4. As we may see, these results are much better than those of our two previous attempts – in particular, this implementation slightly outscores our first implementation of the error monad in the first benchmark and beats it hands down in the two other benchmarks. Again, results seem to indicate that the slowdown induced by our monad is reasonable when the library is used for actual error reporting but that our work is ill-adapted to replace exceptions when these are used as optimizations.

Before calling our latest optimization a moderate victory, let us pursue our hunt for performances one step further.

[4] According to our experiments, replacing this abstraction by a lazy expression actually incurs an additional slowdown

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

Figure 4. Testing the performance of the phantom type monad lifted to the pre-processor

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

Figure 5. Testing the performance of the pure implementation lifted to the pre-processor

4.4 Take 4: Back to basics

After three attempts to improve performances using increasingly complex techniques, we have achieved a moderate speed-up with respect to our first implementation. As it turns out, our latest technique may apply just as well to the first implementation.

Indeed, once again, despite its implementation as composition of function, this version of monadic binding only represents a simple pattern-matching. In other words,

$p \longleftarrow m;\ e$

actually stands for the following expression

```
match proj m with
| Ok    p    → e
| Error err  → Error err
```

Again, we may define `proj` as the identity – or prefer to take advantage of a private sum type. Once this is done, we may implement bind rewriting as

```
let rewrite_bind m p e _loc =
  <:expr< match ErrorMonad.proj $m$ with
    | Ok $p$     → $e$
    | Error err  → Error err>>
```

Finally, we achieve the results presented on figure 5. Perhaps disappointingly, after all this effort, in two out of three benchmarks, our new purely functional implementation of the error monad is much faster than the impure implementation based on phantom types and exceptions – and actually reveals itself reasonably fast for the n queens. On the third benchmark, unsurprisingly, the monad confirms itself as unsuited for an optimization.

4.5 Bottom line

This section started by a claim that monads are a slow technique for managing errors. As it turns out, while our pure error monad proves unsurprisingly inappropriate as a mechanism for optimising returns, according to our experiments, the speed of the pure monad is actually quite reasonable when it is used to deal with errors – even without taking into account compile-time optimizations. By opposition, our attempts to domesticate native exceptions into a construction which could be checked for complete coverage incurred an impressive slowdown which made them useless.

Interestingly, it turns out that a little compiler support – or, in our case, simple compile-time support provided by the library – goes a long way towards improving the speed of our monads. Further experiments with improvements, which go beyond the scope of this document, hint that slightly more complex rewriting rules can go even further – and not just for error monads.

At this point, our library consists in the pure implementation of the error monad (29 lines), compile-time optimizations (49 lines), in addition to some (larger) syntactic sugar.

5. Conclusion

We have demonstrated how to design an error-reporting mechanism for OCaml extending the exception semantics of ML, without altering the language. With respect to OCaml's built-in exception mechanisms, our work adds static checks, polymorphic error reports, hierarchies, support for locally-defined exceptions, and relaxes the need of declaring error cases, while retaining a readable syntax and acceptable performances.

To obtain this richer mechanism, we make use of monads, polymorphic variants and code rewriting and demonstrate the folk theorem of the OCaml community that polymorphic variants are a more generic kind of exceptions. We have also attempted to optimize code through type-unsafe conversions, applied in type-safe manners thanks to the use of type constraints and phantom types, and succeeded in optimizing results through the use of compile-time domain-specific code generators. While we demonstrated five complete implementations of our work, we actually built several dozens, based on either monads, arrows or families of arrows indexed by their effect. We chose not to present these implementations either because they were type-unsafe, too similar to presented implementations or sometimes because their performance was much too poor to make them worthy candidates.

Related works Other works have been undertaken with the objective of making exceptions safer or more flexible. Some of these approaches take the form of compile-time checkers like OCamlExc [22] or similar works for SML [30]. These tools perform program analysis and thus need to evolve whenever the language's semantic does; their maintenance can be quite involved. Similarly, the Catch tool for Haskell [21] uses abstract interpretation to provide guarantees that pattern matches of a program (including pattern-matching upon errors) suffice to cover all possible cases, even when individual pattern-matches are not exhaustive. All these tools retain the exception mechanism of the underlying language and therefore add no new feature, in particular no hierarchies of error classes.

Other efforts are closer to our approach. In addition to the very notion of monads [27], the Haskell community has seen numerous implementations of extendable sets of exceptions, either through monad transformers or dynamic type reflection. Hierarchical exceptions [19] through typeclass hierarchies and dynamic type reflection have also been implemented for Haskell. These choices could have been transposed and sometimes improved into OCaml. We decided to avoid monad transformers in the simple case of error reporting, as these too often require manual definition and manual composition of program-specific or library-specific error cases. Similarly, several variants on run-time type information are possible in OCaml: Deriving's `Typeable` [28] or DynaML [8] provide dynamic type reflection comparable to Haskell's `Data.Typeable`, a combination of Patterns [29] and Coca-ml [5] may be used to provide F#-style class-based pattern-matching downcast [24], while a combination of Patterns and Polymap [11] might be used for the same purpose, with lightweight extendable records. However, we preferred avoiding these dynamic typing solutions which, as their Haskell counterpart, forbid any automatic coverage-check. Yet another encoding of hierarchies has been demonstrated for ML languages, through the use of phantom types [9]. While this work is very interesting, our experiments seem to show that the use of this encoding for exceptions leads to a much less flexible and composable library, in which new sub-kinds of errors cannot be added post-facto to an existing design.

Numerous other works focus on performances in ML languages and their kin. In particular, the Glasgow Haskell Compiler is usually able to efficiently inline simple functions [16] – something which, to our surprise, the OCaml compiler didn't manage in our case. This automatic inlining is, in practice, what we implement manually to optimize away needless abstractions. As for the technique we employ for performing this inlining, it is essentially a specialized form of multi-stage compilation, as available in MetaOCaml [7] or outside the ML community [12]. In particular, our use of specialized code rewriters to avoid the cost of abstraction is an idea also found in MetaOCaml-based works [4].

Future works From this point, our next step will be an official release, consisting in a tiny run-time library, a slightly larger compile-time library defining syntactic sugar and code transformations and a small patch for the `pa_monad` syntactic sugar to drive the code-rewriting functions.

While we have no further plan pertaining to exceptions in the close future, we intend to pursue our work on efficient implementation of monads and arrows in OCaml and perhaps in MetaOCaml, by studying how to best generate run-time and compile-time code from common source code and how to lift monad/arrow transformers from functors to compile-time code transformations. Ideally, this may lead to efficient and readable monadic code, which may find its applications in exceptions, but also for lightweight threading [2].

Finally, some of our work has yielded ideas which may have applications for ongoing efforts toward the design of features similar to Haskell's typeclasses [13] for OCaml [28].

Acknowledgements

We wish to thank Gabriel Scherer for his help with the elaboration and implementation of the syntactic sugar.

References

[1] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1998.

[2] Vincent Balat. Ocsigen: typing web interaction with objective caml. In *ML '06: Proceedings of the 2006 workshop on ML*, pages 84–94, New York, NY, USA, 2006. ACM.

[3] F. Warren Burton. Type extension through polymorphism. *ACM Trans. Program. Lang. Syst.*, 12(1):135–138, 1990.

[4] Jacques Carette and Oleg Kiselyov. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. In *GPCE*, pages 256–274, 2005.

[5] Emmanuel Chailloux. Dynamic object typing in objective caml. In *International LISP Conference 2002*, Oct 2002.

[6] James Cheney and Ralf Hinze. Phantom types, 2003.

[7] Krzysztof Czarnecki, John T. O'Donnell, Jörg Striegnitz, and Walid Taha. Dsl implementation in metaocaml, template haskell, and c++. In *Domain-Specific Program Generation*, pages 51–72, 2003.

[8] Jim Farrand. Dynaml: O'caml dynamic types extensions, 2005. Software package available at `http://farrand.net/dynaml.shtml`.

[9] Matthew Fluet and Riccardo Pucella. Phantom types and subtyping. In *TCS '02: Proceedings of the IFIP 17th World Computer Congress - TC1 Stream / 2nd IFIP International Conference on Theoretical Computer Science*, pages 448–460, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[10] Jacques Garrigue. Programming with polymorphic variants. In *ML Workshop*, 1998.

[11] Jacques Garrigue. Polymorphic mappings for ocaml, 2007. Software package available at `http://www.math.nagoya-u.ac.jp/~garrigue/code/ocaml.html`.

[12] Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *DSL*, pages 39–52, 1999.

[13] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996.

[14] Richard C. Holt and David B. Wortman. A sequence of structured subsets of pl/i. *SIGCSE Bull.*, 6(1):129–132, 1974.

[15] Lydia E. van Dijk Jacques Carette and Oleg Kiselyov. Syntax extension for monads in ocaml. Software package available at `http://www.cas.mcmaster.ca/~carette/pa_monad/`.

[16] Simon Peyton Jones and Simon Marlow. Secrets of the glasgow haskell compiler inliner. *J. Funct. Program.*, 12(5):393–434, 2002.

[17] Oleg Kiselyov. Local globally-quantified exceptions. Software package available at `http://okmij.org/ftp/ML/#poly-exn`.

[18] Oleg Kiselyov, Chung chieh Shan, and Amr Sabry. Delimited dynamic binding. *SIGPLAN Not.*, 41(9):26–37, 2006.

[19] Simon Marlow. An extensible dynamically-typed hierarchy of exceptions. In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*. ACM Press, September 2006.

[20] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[21] Neil Mitchell and Colin Runciman. A static checker for safe pattern matching in Haskell. In *Trends in Functional Programming*, volume 6. Intellect, February 2007.

[22] François Pessaux. *Détection statique d'exceptions non rattrapées en Objective Caml*. PhD thesis, Université Pierre & Marie Curie - Paris 6, 2000.

[23] Andreas Rossberg, Didier Le Botlan, Guido Tack, Thorsten Brunklaus, and Gert Smolka. Alice through the looking glass. *Trends in Functional Programming*, 5:77–96, 2006.

[24] Don Syme. Leveraging .net meta-programming components from f#: integrated queries and interoperable heterogeneous execution. In *ML '06: Proceedings of the 2006 workshop on ML*, pages 43–54, New York, NY, USA, 2006. ACM.

[25] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Inf. Comput.*, 111(2):245–296, 1994.

[26] David Teller, Arnaud Spiwack, and Till Varoquaux. Catch me if you can. Software package available at `http://www.univ-orleans.fr/lifo/Members/David.Teller/software/catch_0_2.tgz`.

[27] Philip Wadler. The essence of functional programming. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, New York, NY, USA, 1992. ACM.

[28] Jeremy Yallop. Practical generic programming in ocaml. In *ML '07: Proceedings of the 2007 workshop on Workshop on ML*, pages 83–94, New York, NY, USA, 2007. ACM.

[29] Jeremy Yallop. Ocaml patterns: General-purpose extension to ocaml pattern-matching facilities, 2008. Software package available at `http://code.google.com/p/ocaml-patterns/`.

[30] Kwangkeun Yi and Sukyoung Ryu. A cost-effective estimation of uncaught exceptions in standard ml programs. *Theor. Comput. Sci.*, 277(1-2):185–217, 2002.