A Modal Calculus for Exception Handling

Aleksandar Nanevski [1]

Division of Engineering and Applied Science
Harvard University
Cambridge, Massachusetts

Abstract

The exception monad, while an adequate mechanism for providing the denotational semantics of exceptions, is somewhat awkward to program with. Just as any other monad, it forces a programming style in which exceptional computations are explicitly sequentialized in the program text. In addition, values of computation types must usually be tested before use, in order to determine if they correspond to a raised exception.

In this paper we propose a type system that rearranges the monadic formulation, so that the above shortcomings are avoided. Instead of the exception monad, we propose the operator □ from the modal logic S4 to encode exceptional computation. The way tracking of exceptions is organized in the modal system is exactly dual to the monadic case, reflecting the well-known property that □ is actually a comonad.

Key words: monads, exceptions, modal logic.

1 Introduction

Monads and the monadic λ-calculus [20,21,33,34,35] present a type theoretic method for grafting effectful features onto a purely functional language. Monads are type constructors (satisfying certain categorical properties) that internalize the notion of effectful computation. The idea is to limit the appearance of effects to terms of monadic type, thus separating the possibly impure subterms from the pure ones, and ensuring a disciplined propagation of effects. Furthermore, the monadic typing discipline forces monadic computations to be serialized; the reduction order for any single monadic term is apparent from the term itself, specifying in that way a total ordering on the effects that the evaluation of the term may bring about.

For example, in the literature today, the customary way of formalizing type-safe calculi of exceptions is via the exception monad [21,34]. This ap-

---

[1] Email: `aleks@eecs.harvard.edu`

proach defines the monadic type $\bigcirc A$ as $\bigcirc A = A + E$ where E is the type of the exception that can be raised. Then term constructors **comp** and **let comp** are introduced, using the following definitions that assume the standard formulation of disjoint sums.

$$\textbf{comp } e \stackrel{\text{def}}{=} \textbf{inl } e$$

$$\textbf{let comp } x = e_1 \textbf{ in } e_2 \stackrel{\text{def}}{=} \textbf{case } e_1 \textbf{ of inl } x \Rightarrow e_2 \mid \textbf{inr } y \Rightarrow \textbf{inr } y$$

The typing rules for these constructs are appropriately derived as:

$$\frac{\Delta \vdash e : A}{\Delta \vdash \textbf{comp } e : \bigcirc A} \qquad \frac{\Delta \vdash e_1 : \bigcirc A \qquad \Delta, x{:}A \vdash e_2 : \bigcirc B}{\Delta \vdash \textbf{let comp } x = e_1 \textbf{ in } e_2 : \bigcirc B}$$

There are also additional term constructors used to raise and handle the exception associated with the monad $\bigcirc$.

$$\textbf{raise} : E \Rightarrow \bigcirc A =$$
$$\lambda e.\ \textbf{inr } e$$
$$\textbf{handle} : \bigcirc A \Rightarrow (E \Rightarrow A) \Rightarrow A =$$
$$\lambda e.\ \lambda h.\ \textbf{case } e \textbf{ of inl } v \Rightarrow v \mid \textbf{inr } exn \Rightarrow h\ (exn)$$

The constructor **raise** takes the expression $e : E$ and coerces it into **inr** e. This way, it implements exception raising, passing the value of e along. The constructor **handle** takes the expression $e : \bigcirc A$ and a handler function h. If e evaluates to a value $v : A$, the result of handling is v. If e raises the exception with a value $exn : E$, then the result of handling is $h\ (exn)$.

The operational semantics follows the standard operational semantics associated with disjoint sums. Assume for example that $f : \textbf{int} \Rightarrow \bigcirc\textbf{int}$. The following program adds the results of $f(1)$ and $f(2)$; if the evaluation of any of the two function applications raises an exception, the overall computed result is zero.

```
handle (let comp x1 = f(1)
            comp x2 = f(2)
        in
            comp (x1 + x2)
        end) (λexn. 0)
```

There are several shortcomings with this approach, making it somewhat unnatural to program with exceptional computations [26,22].

First, the program forces a choice between the evaluation order of f(1) and f(2), even though the eventual effects of either computation do not influence

the other one. It would be very convenient to have a construct **uncomp** that we could use to rewrite the above program into the following.

```
handle (uncomp f(1) + uncomp f(2)) (λexn. 0)
```

In this version of the program, the evaluation order of the two computations `f(1)` and `f(2)` is left to the operational semantics of addition, rather than being specified by the program itself. In fact, as far as the type preservation is concerned, any evaluation order is sound.

The second problem with the monadic formulation of exceptions potentially concerns efficiency. Evaluation of an expression $e : \bigcirc A$ terminates either with a value, or with raising an exception. The outcome of the evaluation of e has to be tagged (with **inl** or **inr**) in order to distinguish between the two cases, and this tag has to be checked at run time whenever e is used (as apparent from the definition of **let comp**). Exceptions that have been raised but not yet handled are first-class objects in the language, with the same status as any other value.

However, the way exceptions are usually used in functional languages does not require this generality. Once an exception is raised, it must be handled (or the evaluation stops), and may not be passed as an argument to other functions. If raised unhandled exceptions were not values, there would be no need for tagging and, correspondingly, no need for tag checking.

The problem with excessive serialization of exceptional monadic programs has been addressed previously by means of monadic reflection and reification [8,9,10]. Reflection and reification are translations between an effectful source language (which provides the syntax for programming) and a monadic lambda calculus (which provides the semantics using the exception monad as defined above). Unfortunately, as concluded in [8], reflection and reification still incur the operational penalties of tagging and tag checking. More seriously, they do not admit a well-behaved formulation in the style of natural deduction, and are thus also not quite adequate to program with. Natural deduction usually organizes a calculus into mutually independent groups of rules, split according to the rule's main type operator. Reflection and reification must make it explicit that the exception monad is defined in terms of disjoint sums, and are thus presented as a pair of rules about disjoint sums. Obviously, these rules cannot be independent of the usual introduction and elimination rules for disjoint sum, and thus violate the basic principles of natural deduction.

In this paper, we propose a formulation of exceptions that avoids the described serialization and tagging. The approach respects natural deduction, and does not use any indirect translations. It also directly corresponds to a fragment of constructive modal logic S4 (CS4 in the future text).

Modal logic is a logic for reasoning about truth across various worlds. A proposition may be true in some worlds, but not true in some others. In CS4 we also have the propositional operators $\Box$ (also referred to as *necessity*), and $\Diamond$ (*possibility*). $\Box$ is a universal quantifier; $\Box A$ is true iff A is true in all worlds.

$\diamond$ is an existential quantifier; $\diamond A$ is true iff A is true in some world.

As described in [7,2,1], the fragment of CS4 including $\diamond$ but not $\square$, with some mild additional conditions, obtains the logic (called *lax* logic) which provides the foundation for monads and the monadic lambda calculus. A generic monad can thus be viewed as an existential quantifier over possible worlds, and the corresponding computational effect of the monad is a witness to the existence of an appropriate possible world. Computationally, $\diamond A$ shows that there exists a world (the one obtained after performing the monadic effect), in which A is true.

This view also logically explains the serialization property of the monadic lambda calculus. Composing effectful computations corresponds to threading of existentials, corresponds to stepping from one possible world to another. These steps do not necessarily commute, because once a step has been performed, all the further steps may depend on it. Thus, each step has global scope; it cannot be backtracked, and must persist till the end of the computation. This makes monads particularly convenient for representing persistent effects which engender a permanent change to the environment in which they execute (e.g. destructive state update).

But, exceptions are not like that. In fact, one of the defining characteristics of exceptions is the possibility of handling. A raised exception need not persist until the end of the execution because its scope can be delimited using an exception handler. For our computational application, we can view exception handlers as possible worlds. Then a computation of type A whose execution may raise the exceptions from the set C may evaluate in the scope of *all* handlers capable of handling the exceptions from the set C. Thus, it should be assigned a type $\square_C A$ that corresponds to a *universal quantification* over exception handlers, *bounded* by C. The corresponding λ-calculus for the fragment of modal logic with the $\square$ quantifier will then give us a language adequate for representing exceptions.

In the rest of the paper, we describe the necessitation fragment of CS4 and its lambda calculus (Section 2), and then show how the indexed variant of the $\square$ operator can be used for tracking effects (Sections 3 and 4). In Section 5 we specialize the development to exceptions, before discussing the related and future work in Section 7.

We note that the ideas presented here may be generalized to other kinds of named control effects, like catch-and-throw and composable continuations, as shown in the longer version of this paper [24] and the author's PhD dissertation [25]. An interpreter implementing the described type systems is available at

```
http:/www.cs.cmu.edu/~aleks/papers/effects/nubox.tar.gz.
```

2 Modal $\lambda^\square$-calculus

The starting point for the development of our language for effects in general, and exceptions in particular, is the $\lambda^\square$-calculus of [27,5]. The $\lambda^\square$ is the proof-

term system for the necessitation fragment of the CS4 modal logic, and it was first considered in functional programming in the context of specialization for purposes of run-time code generation [5,36,37]. The syntax of $\lambda^\square$ is summarized below, where we use b to stand for a predetermined set of base types.

$$\textit{Types} \qquad A ::= b \mid A_1 \to A_2 \mid \square A$$

$$\textit{Terms} \qquad e ::= x \mid u \mid \lambda x{:}A.\ e \mid e_1\ e_2 \mid \textbf{box}\ e \mid \textbf{let box}\ u = e_1\ \textbf{in}\ e_2$$

$$\textit{Contexts}\ \Gamma, \Delta ::= \cdot \mid \Gamma, x{:}A$$

The most important feature of the calculus is the type constructor $\square$ which is referred to as *modal necessity*, as in the CS4 modal logic it is a necessitation modifier on propositions [27]. For the purposes of this paper, a useful operational intuition is to consider the type $\square A$ as classifying *pure computations of type A*. In contrast, the non-modal type A contains only *values*. In functional programming, pure computations are usually identified with their values, but we separate the two here, as this would lead to easier development of the notion of *impure* (or *effectful*) computation in the subsequent sections.

The type system of $\lambda^\square$ is presented below.

$$\overline{\Delta; (\Gamma, x{:}A) \vdash x : A} \qquad \overline{(\Delta, u{:}A); \Gamma \vdash u : A}$$

$$\frac{\Delta; (\Gamma, x{:}A) \vdash e : B}{\Delta; \Gamma \vdash \lambda x{:}A.\ e : A \to B} \qquad \frac{\Delta; \Gamma \vdash e_1 : A \to B \quad \Delta; \Gamma \vdash e_2 : A}{\Delta; \Gamma \vdash e_1\ e_2 : B}$$

$$\frac{\Delta; \cdot \vdash e : A}{\Delta; \Gamma \vdash \textbf{box}\ e : \square A} \qquad \frac{\Delta; \Gamma \vdash e_1 : \square A \quad (\Delta, u{:}A); \Gamma \vdash e_2 : B}{\Delta; \Gamma \vdash \textbf{let box}\ u = e_1\ \textbf{in}\ e_2 : B}$$

It distinguishes between two variable contexts: Γ for variables bound to values, and Δ for variables bound to computations. The introduction and elimination forms of the type constructor $\square$ are the term constructors **box** and **let box**, respectively. Operationally, the term constructor **box** suspends the evaluation of its argument expression e, and wraps it into a thunk **box** e which can be then be further manipulated by the rest of the program. The expression **box** e is a value in this calculus. Note that the typing rule for **box** prohibits e to refer to variables from Γ; it is not possible to coerce values into computations. This is counter-intuitive to our interpretation of the modal calculus as a calculus of effects, and we will remedy it shortly. The elimination form **let box** $u = e_1$ **in** e_2 takes the computation boxed by e_1 and binds *the whole computation* to the variable u to be used in e_2. In other words, the

operational semantics for **let box** is given by a reduction rule

$$\textbf{let box } u = \textbf{box } e \textbf{ in } e_2 \longrightarrow [e/u]e_2$$

Observe that this is different from the monadic **let comp**. The evaluation rules for **let comp** $x = e_1$ **in** e_2 evaluates e_1 to a value which is then bound to x. In **let box** $u = \textbf{box } e$ **in** e_2, the computation e is *not* evaluated. The variable u binds a whole computation, rather than just a value.

Example 2.1 The function `exp2` below takes an integer argument n and builds a computation for 2^n.

```
fun exp2 (n : int) : □int =
    if n = 0 then box 1
    else
      let box u = exp2 (n - 1)
      in
          box (2 * u)
      end

- e5 = exp2 5;
val e5 = box (2 * (2 * (2 * (2 * (2 * 1))))) : □int
```

In the elimination form **let box** $u = e_1$ **in** e_2, the bound variable u belongs to the context Δ of expression variables, but it can be used in e_2 in both computation positions (i.e., under a box), and value positions. This way we can compose computations, but also explicitly force their evaluation. In the above example, we can force the evaluation of `e5` in the following way.

```
- let box u = e5 in u;
val it = 32 : int
```

Example 2.2 The operator $\Box$ satisfies the following characteristic axioms.

$$f_1 : \Box A \to A =$$
$$\quad \lambda x. \textbf{ let box } u = x \textbf{ in } u$$
$$f_2 : \Box A \to \Box\Box A =$$
$$\quad \lambda x. \textbf{ let box } u = x \textbf{ in box } (\textbf{box } u)$$
$$f_3 : \Box(A \to B) \to \Box A \to \Box B =$$
$$\quad \lambda x. \lambda y. \textbf{ let box } u = x \textbf{ in let box } v = y \textbf{ in box } (u \; v)$$

As already mentioned, the typing rule for **box** prohibits e to refer to variables from Γ; it is not possible to coerce values into computations. In order to provide for this, we redefine the notion of λ-abstraction, so that function arguments are kept in the context Δ as pure computations, rather than in

6

the context Γ of values. The context Γ may therefore be subsumed by Δ, to obtain the following system.

$$\frac{}{\Delta, x{:}A \vdash x : A}$$

$$\frac{(\Delta, x{:}A) \vdash e : B}{\Delta \vdash \lambda x{:}A.\ e : A \rightarrow B} \qquad \frac{\Delta \vdash e_1 : A \rightarrow B \quad \Delta \vdash e_2 : A}{\Delta \vdash e_1\ e_2 : B}$$

$$\frac{\Delta \vdash e : A}{\Delta \vdash \mathbf{box}\ e : \Box A} \qquad \frac{\Delta \vdash e_1 : \Box A \quad (\Delta, u{:}A) \vdash e_2 : B}{\Delta \vdash \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2 : B}$$

The above system annihilates the logical distinction between the propositions $\Box A$ and A, but we do retain their operational difference by which the type A classifies values, and the type $\Box A$ classifies pure computations. In the next section, we will introduce a whole family $\Box_C$ of necessitation operators indexed by a set of effect names, so that the type $\Box_C A$ will classify computations with effects C. The propositions A and $\Box_C A$ will be logically equivalent in case C is empty, but not otherwise.

3   Names as markers for effects

In this section, we extend the calculus presented above with the notion of *names*. Names are labels that provide a formal abstraction for tracking effects. Each effect will be assigned a name, and if an effect appears in a computation, then the corresponding $\Box$-type will be indexed by that name. For example, if we have an exception X, then a computation of type A that may raise this exception, will be given a type $\Box_X A$.

The described indexing of the modal operator with names is similar to the one found in the monadic lambda calculus given by Wadler [35], where labels are used to identify the effects that may occur under a monad. In our setup, however, we will also allow dynamic introduction of fresh names into the computation (and hence, generation of new effects), and establish a typing discipline for it. Having mentioned this idea to provide some intuition toward our overall goal, we proceed to introduce our calculus in stages. Rather than formally tying names to effects immediately, we now present a limited fragment that is intended only to account for dynamic introduction of names and for name propagation. The relationship between names and effects (and exceptions in particular), and how effects are raised and handled will be discussed in the subsequent sections.

We start by explaining the syntax and various syntactic conventions of our

language.

$$
\begin{array}{lll}
\textit{Names} & X \ \in \ \mathcal{N} \\[4pt]
\textit{Supports} & C, D ::= \cdot \mid C, X \\[4pt]
\textit{Types} & A ::= b \mid A_1 \to A_2 \mid A_1 \nrightarrow A_2 \mid \Box_C A \\[4pt]
\textit{Terms} & e ::= u \mid \lambda x{:}A.\ e \mid e_1\ e_2 \mid \\[4pt]
& \qquad \textbf{box}\ e \mid \textbf{let box}\ u = e_1\ \textbf{in}\ e_2 \mid \\[4pt]
& \qquad \nu X{:}A.\ e \mid \textbf{choose}\ e \\[4pt]
\textit{Variable contexts} & \Delta ::= \cdot \mid \Delta, u{:}A[C] \\[4pt]
\textit{Name contexts} & \Sigma ::= \cdot \mid \Sigma, X{:}A
\end{array}
$$

Just like $\lambda^{\Box}$, our calculus makes a distinction between values and computations. The two are separated by a modal type constructor $\Box$, except that now we have a whole family of modal type constructors – one for each *finite* sequence of names C, where the names are drawn from a countably infinite universe of names $\mathcal{N}$. As already hinted before, the type $\Box_C A$ classifies computations that may raise any of the effects whose names are in C. The sequence C is referred to as a *support* of such expressions. We will also consider a partial ordering $\sqsubseteq$ on supports. If a term has support C, than it can safely appear in the scope of a handler capable of dealing with the names in any $D \sqsupseteq C$. If a term is pure (i.e., it has empty support), it need not be restricted to any particular set of handlers. Therefore, we require that the empty support is the smallest element of $\sqsubseteq$.

Because now computations can contain effects, we extend the typing assignments in the context Δ to keep track not only of the typing, but also of the support of a variable. So, for example, the typing $u{:}A[C]$ declares a variable u which can be bound to an expression of type A and support C. We will frequently abbreviate $x{:}A[\,]$ as $x{:}A$.

A further change from $\lambda^{\Box}$ is an addition of the context Σ which declares the names (and their types) that are currently active in the program. Because the types of our calculus depend on names, we must impose some conditions on well-formedness of contexts. A context Σ is well-formed if every type in Σ uses only names declared to the left of it. The variable context Δ is well-formed with respect to Σ, if all the names that appear in the types of Δ are declared in Σ.

The types of the new calculus now include the family $A \nrightarrow B$ whose introduction and elimination forms are $\nu x{:}A.\ e$ and **choose** e. These constructs are used to dynamically introduce fresh names into the calculus. For example, the term $\nu X{:}A.\ e$ binds a name X of type A that can subsequently be used in e. Because names stand for effects, this construct really declares a new effect, and enables e to raise it and handle it. Whatever e does with X, though, we

will ensure through the type system that the result of the evaluation of e does not depend on X; we must prevent X to escape the scope of its introduction form. The ν-abstraction will be a value in our calculus. In particular, it will suspend the evaluation of e. If we want to evaluate it, we must **choose** it. The term constructor **choose** allocates a *fresh* name of type A, substitutes it for the name bound in the argument ν-abstraction of type $A \nrightarrow B$, and proceeds to evaluate the body of the abstraction.

Finally, enlarging an appropriate context by a new variable or a name is subject to the usual variable conventions: the new variables and names are assumed distinct, or are renamed in order not to clash with already existing ones. Terms that differ only in the syntactic representation of their bound variables and names are considered equal. The binding forms in the language are $\lambda x{:}A.\, e$, **let box** $u = e_1$ **in** e_2 and $\nu X{:}A.\, e$. Capture-avoiding substitution $[e_1/x]e_2$ of expression e_1 for the variable x in the expression e_2 is defined to rename bound variables and names when descending into their scope. Given a term e, we denote by $\mathbf{fv}(e)$ the set of free variables of e. The set of names appearing in the type A is denoted by $\mathbf{fn}(A)$.

The typing judgment of the core fragment is

$$\Sigma; \Delta \vdash e : A\,[C]$$

The judgment works with two contexts: context of names Σ and context of variables Δ. Given an expression e, the judgment checks whether e has type A, and whether its effects are in the support C. The core fragment of the typing rules is presented in Figure 1, and we explain it next.

A pervasive characteristic of the type system is the *support weakening principle*; that is

$$\text{if } \Sigma; \Delta \vdash e : A\,[C] \text{ and } C \sqsubseteq D, \text{ then } \Sigma; \Delta \vdash e : A\,[D]$$

Support of the expression e determines which effects e can raise, and therefore, which handlers can restore its purity. Consequently, the support weakening principle formally models a very intuitive property that if the effects of e can be handled by some handler, then they can be handled by a stronger handler as well. In particular, if e is effect-free, then it can be handled by any and all handlers; the empty support is the smallest element of the partial ordering $\sqsubseteq$.

A further property that we formally represent is that values of the language are effect free. Indeed, values obviously cannot raise any effects, simply because their evaluation is already finished. Therefore, the support of the values of our system will be empty, and according to the support weakening principle, it can then be weakened arbitrarily. This explains the explicit weakening in the hypothesis rule and the arbitrary support in the conclusions of the typing rules for λ- and ν-abstractions and for **box**.

λ-calculus fragment. The rule for λ-abstraction requires that the body e of the abstraction be pure; that is e has to match the empty support. This is not to

$$\frac{C \sqsubseteq D}{\Sigma; (\Delta, u{:}A[C]) \vdash u : A\,[D]} \qquad \frac{\Sigma; (\Delta, x{:}A) \vdash e : B\,[\;]}{\Sigma; \Delta \vdash \lambda x{:}A.\ e : A \to B\,[C]}$$

$$\frac{\Sigma; \Delta \vdash e_1 : A \to B\,[C] \quad \Sigma; \Delta \vdash e_2 : A\,[C]}{\Sigma; \Delta \vdash e_1\ e_2 : B\,[C]}$$

$$\frac{\Sigma; \Delta \vdash e : A\,[D]}{\Sigma; \Delta \vdash \mathbf{box}\ e : \Box_D A\,[C]}$$

$$\frac{\Sigma; \Delta \vdash e_1 : \Box_D A\,[C] \quad \Sigma; (\Delta, u{:}A[D]) \vdash e_2 : B\,[C]}{\Sigma; \Delta \vdash \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2 : B\,[C]}$$

$$\frac{(\Sigma, X{:}A); \Delta \vdash e : B\,[\;] \quad X \notin \mathbf{fn}(A, B, \Delta)}{\Sigma; \Delta \vdash \nu X{:}A.\ e : A \nrightarrow B\,[C]} \qquad \frac{\Sigma; \Delta \vdash e : A \nrightarrow B\,[C]}{\Sigma; \Delta \vdash \mathbf{choose}\ e : B\,[C]}$$

Fig. 1. Type system of the core fragment.

say that e cannot contain any effects; it can, but only if they are encapsulated under a **box** (and correspondingly accounted for in the type of e). This is similar to monadic type systems where function bodies must be pure, and effects can be raised only under a monad. On the other hand, because λ-terms are values, the support of the whole abstraction can be arbitrarily weakened, as explained before.

It is implicitly assumed that the argument type A is well-formed with respect to the name context Σ before it is introduced into the variable context Δ. Note further that the bound variable x is introduced into Δ with *empty* support, according to our decision to allow coercion of values into pure computations. Thus, x must always be bound to an effect-free expression. This will force us to commit to call-by-value evaluation strategy for the calculus; we must reduce function arguments to values (which are effect-free) before passing them on.

Modal fragment. To type a computation **box** e, we must check if e is well-typed and matching the support that is supplied as an index to the $\Box$ constructor. Boxed expressions are values of computation type, so their support can be arbitrarily weakened to any well-formed support set C. The $\Box$-elimination rule is a straightforward extension of the corresponding $\lambda^\Box$ rule. The only difference is that the bound expression variable u from the context Δ now has to be stored with its support annotation.

It is interesting here to contrast the elimination construct **let box** with

10

the monadic elimination construct **let comp**, presented in the introduction. The construct **let comp** $x = e_1$ **in** e_2 evaluates e_1, to bind its *value* to x to be used in e_2. The type of e_2 *must* be monadic. On the other hand, the construct **let box** $u = e_1$ **in** e_2 evaluates e_1 to an *effectful computation* which is then bound to u to be used in e_2 (possible more than once). The type of e_2 need not specify any effects.

Thus, in the typing rule for **let comp**, effects are indicated by insisting on a monadic type of e_2, which appears to the *right* of the turnstile. In the typing rule for **let box**, effects are indicated by the support of the variable u, which appears to the *left* of the turnstile.

In this particular sense, the formulation of effect calculi using $\Box$ is *dual* to the monadic one. It is not surprising then that the $\Box$ operator of modal logic is usually categorically modeled by a *comonad*. We do not explore this distinction further in the paper, but refer the reader to the work of Kobayashi [17], Alechina et al. [1] and Bierman and de Paiva [3], for a detailed discussion of categorical models for $\Box$.

Names fragment. The rule for $\nu X{:}A.\ e$ must check e for well-typedness in a context Σ extended with the new name $X{:}A$. Similar to the λ rule, we require that e has empty support; all the eventual effects that e may raise must be boxed. The characteristics of the ν constructor, however, is the further requirement that X does not appear in the type B. This ensures that X remains local to e; it can never escape the scope of its introducing ν in any observable way. The effect corresponding to X will either never be raised in the course of evaluation of e (i.e., it never appears in e or it appears in some dead-code part of e), or all the occurrences of X are handled by some handler.

The term constructor **choose** is the elimination form for $A \twoheadrightarrow B$. It picks a fresh name and substitutes it for the bound name in the ν-abstraction.

Example 3.1 If C, C_1, C_2 and D are well-formed supports such that $C_1 \sqsubseteq C$ and $C_2 \sqsubseteq C$, then the following terms are well-typed.

(i) $\vdash \lambda x.\ \mathbf{box}\ x : A \to \Box_D A$

(ii) $\vdash \lambda x.\ \mathbf{let\ box}\ u = x\ \mathbf{in}\ u : \Box_{C_1} A \to A\,[C]$

(iii) $\vdash \lambda x.\ \mathbf{let\ box}\ u = x\ \mathbf{in}\ \mathbf{box}\ u : \Box_{C_1} A \to \Box_C A$

(iv) $\vdash \lambda x.\ \mathbf{let\ box}\ u = x\ \mathbf{in}\ \mathbf{box\ box}\ u : \Box_{C_1} A \to \Box_D \Box_C A$

(v) $\vdash \lambda x.\ \lambda y.\ \mathbf{let\ box}\ u = x\ \mathbf{in}\ \mathbf{let\ box}\ v = y\ \mathbf{in}\ \mathbf{box}\ u\ v :$
$$\Box_{C_1}(A \to B) \to \Box_{C_2} A \to \Box_C B$$

Example 3.2 To abbreviate notation and reduce clutter, we introduce into the calculus the term constructor **unbox** e as a syntactic abbreviation for **let box** $u = e$ **in** u. The new term constructor has the following derived typing rule

$$\frac{\Sigma; \Delta \vdash e : \Box_C A\,[D] \qquad C \sqsubseteq D}{\Sigma; \Delta \vdash \mathbf{unbox}\ e : A\,[D]}$$

We also define **let val** $x = e_1$ **in** e_2 to stand for **unbox** $((\lambda x.\ \textbf{box}\ e_2)\ e_1)$, rather than the usual $(\lambda x.\ e_2)\ e_1$. The additional complication arises because we have to box e_2 and make it pure before we can put it under a λ-abstraction. The derived typing rule for **let val** is

$$\frac{\Sigma; \Delta \vdash e_1 : A\,[C] \qquad \Sigma; (\Delta, x{:}A) \vdash e_2 : B\,[C]}{\Sigma; \Delta \vdash \textbf{let val}\ x = e_1\ \textbf{in}\ e_2 : B\,[C]}$$

Example 3.3 Anticipating Section 5, suppose that our language contains the term constructor **raise**, such that $\textbf{raise}_X\ e$ raises an exception X passing an argument e along (assuming that both X and e have the same type). If X is a name of type A, then the following term is well-typed.

$$\lambda x.\ \textbf{let box}\ u = x\ \textbf{in box}\ (\textbf{raise}_X\ u) : \Box A \to \Box_X A$$

Assume further that $e_1{:}B$ is a closed and exception-free term, and $e_2{:}A$ is a closed term which may raise the exception X. Then the expression

$$\textbf{choose}\ (\nu Y{:}A.\ (\lambda x{:}\Box_{X,Y} A.\ e_1)\ (\textbf{box raise}_Y\ e_2))$$

declares a new exception Y, which is then raised within the computation **box** $(\textbf{raise}_Y\ e_2)$ of type $\Box_{X,Y} A$. In fact, because neither x nor Y appear in e_1, the type of the application will not depend on Y either. Actually, even more is true: the argument computation will never even be forced; it is dead code. The ν-clause is well-typed, of type $A \nrightarrow B$, and the whole expression is of type B. In Section 5 where we introduce exception handling, we would be able to present a more meaningful use of **choose** and ν.

4 Operational semantics

The operational semantics of this basic fragment of our calculus is defined through the judgment

$$\Sigma, e \longmapsto \Sigma', e'$$

which relates an expression e with its one-step reduct e'. The relation is defined on expressions with no free variables. An expression e can contain effects, whose names must be declared in Σ, but it must have *empty support*. In other words, we only consider for evaluation those expressions whose effects are either boxed, or appear in a dead-code part, or are handled. The reduct e' can introduce new names into the computation, which will be accounted in the extended name context Σ'. However, the new names too, will mark effects which are either boxed, never raised or otherwise handled. We define the reduction judgment in the style of Wright and Felleisen [38]. The formalization is for a call-by-value strategy, and it relies on the definitions of redex and

evaluation contexts below.

Values	$v ::= x \mid \lambda x{:}A.\ e \mid \mathbf{box}\ e \mid \nu X{:}A.\ e$
Redexes	$r ::= v_1\ v_2 \mid \mathbf{let\ box}\ u = v\ \mathbf{in}\ e \mid \mathbf{choose}\ v$
Evaluation contexts	$E ::= [\ ] \mid E\ e_1 \mid v_1\ E \mid \mathbf{let\ box}\ u = E\ \mathbf{in}\ e \mid \mathbf{choose}\ E$

Each expression e can be decomposed uniquely as $e = E[r]$ where E is an evaluation context and r is a redex. To define a small-step operational semantics of the calculus, it is enough to define primitive reduction relation for redexes (which we denote by $\longrightarrow$), and let the evaluation of expressions always first reduce the redex identified by the unique decomposition.

$$\Sigma, (\lambda x.\ e)\ v \longrightarrow \Sigma, [v/x]e$$

$$\Sigma, \mathbf{let\ box}\ u = \mathbf{box}\ e_1\ \mathbf{in}\ e_2 \longrightarrow \Sigma, [e_1/u]e_2$$

$$\Sigma, \mathbf{choose}\ (\nu X{:}A.\ e) \longrightarrow (\Sigma, Y{:}A), [Y/X]e, \quad Y \notin \mathbf{dom}(\Sigma)$$

$$\frac{\Sigma, r \longrightarrow \Sigma', e'}{\Sigma, E[r] \longmapsto \Sigma', E[e']}$$

Example 4.1 As an illustration of the operational semantics of the calculus, we present the first couple of steps from the evaluation of the term from Example 3.3.

$$(X{:}A), \mathbf{choose}\ (\nu Y{:}A.\ (\lambda x{:}\Box_{X,Y}A.\ e_1)\ (\mathbf{box\ raise}_Y\ e_2)) \longmapsto$$

$$(X{:}A, Z{:}A), (\lambda x{:}\Box_{X,Z}A.\ e_1)\ (\mathbf{box\ raise}_Z\ e_2) \longmapsto$$

$$\text{where } Z \text{ is a fresh name}$$

$$(X{:}A, Z{:}A), e_1 \longmapsto$$

$$\cdots$$

The rest of this section develops the basic properties of the calculus. We present them here, because the future extensions will all rely on the basic structure of these results.

Proposition 4.2 (Expression substitution principle) *If* $\Sigma; \Delta \vdash e_1 : A\,[C]$ *and* $\Sigma; (\Delta, u{:}A[C]) \vdash e_2 : B\,[D]$, *then* $\Sigma; \Delta \vdash [e_1/u]e_2 : B\,[D]$.

Lemma 4.3 (Replacement) *If* $\Sigma; \Delta \vdash E[e] : A\,[C]$, *then there exist a type* B *such that*

(i) $\Sigma; \Delta \vdash e : B\,[C]$, *and*

(ii) *if* Σ', Δ' *extend* Σ, Δ, *and* $\Sigma'; \Delta' \vdash e' : B\,[C]$, *then* $\Sigma'; \Delta' \vdash E[e'] : A\,[C]$

Lemma 4.4 (Canonical forms) *Let v be a closed value such that $\Sigma; \cdot \vdash v : A\,[C]$. Then the following holds:*

  (i) *if $A = A_1 \to A_2$, then $v = \lambda x{:}A_1.\,e$ and $\Sigma; x{:}A_1 \vdash e : A_2\,[\,]$*

  (ii) *if $A = \square_D B$, then $v = \mathbf{box}\ e$ and $\Sigma; \cdot \vdash e : B\,[D]$*

  (iii) *if $A = A_1 \nrightarrow A_2$, then $v = \nu X{:}A_1.\,e$ and $(\Sigma, X{:}A_1); \cdot \vdash e : A_2\,[\,]$*

As a consequence, the support of v can be arbitrarily weakened, i.e. $\Sigma; \cdot \vdash v : A\,[D]$, for any support D.

Lemma 4.5 (Subject reduction) *If $\Sigma; \cdot \vdash e : A\,[C]$ and $\Sigma, e \longrightarrow \Sigma', e'$, then Σ' extends Σ and $\Sigma'; \cdot \vdash e' : A\,[C]$.*

Theorem 4.6 (Preservation) *If $\Sigma; \cdot \vdash e : A\,[C]$ and $\Sigma, e \longmapsto \Sigma', e'$, then Σ' extends Σ, and $\Sigma'; \cdot \vdash e' : A\,[C]$.*

Lemma 4.7 (Progress for $\longrightarrow$) *If $\Sigma; \cdot \vdash r : A\,[C]$, then there exists a term e' and a context Σ', such that $\Sigma, r \longrightarrow \Sigma', e'$.*

Lemma 4.8 (Unique decomposition) *For every expression e, either:*

  (i) *e is a value, or*

  (ii) *$e = E[r]$ for a unique evaluation context E and a redex r.*

Theorem 4.9 (Progress) *If $\Sigma; \cdot \vdash e : A\,[\,]$, then either*

  (i) *e is a value, or*

  (ii) *there exists a term e' and a context Σ', such that $\Sigma, e \longmapsto \Sigma', e'$.*

Proposition 4.10 (Determinacy) *If $\Sigma, e \longmapsto^n \Sigma_1, e_1$ and $\Sigma, e \longmapsto^n \Sigma_2, e_2$, then there exists a permutation of names $\pi : \mathcal{N} \to \mathcal{N}$, fixing the domain of Σ, such that $\Sigma_2 = \pi(\Sigma_1)$ and $e_2 = \pi(e_1)$.*

5 Exceptions

The calculus presented thus far did not involve any concrete notions of effects. It was only capable of dynamic introduction and of propagation of effects, but not, in fact, of raising or handling them. In this section we extend our code fragment into a calculus of exceptions. The idea is to assign a name to each exception, which could then be propagated and tracked by means of the core fragment. To be able to raise and handle exceptions, we need further constructs specific only to exceptions. Thus, we extend the syntax of our language in the following way.

$$\textit{Exception handlers}\ \Theta ::= \cdot \mid X z \to e, \Theta$$

$$\textit{Terms} \qquad\qquad e ::= \dots \mid \mathbf{raise}_X\ e \mid e\ \mathbf{handle}\ \langle\Theta\rangle$$

Informally, the role of $\mathbf{raise}_X\ e$ is to evaluate e and than raise an exception X, passing the value of e along. On the other hand, $e\ \mathbf{handle}\ \langle\Theta\rangle$ evaluates

e (which may raise exceptions), and all the raised exceptions are handled by the exception handler Θ.

An exception handler is defined as a finite set of *exception patterns*. A pattern $Xz \to e$ associates the exception X with the expression e. Whenever X is raised with some value v, it will be handled by evaluating the expression $[v/z]e$. Given a handler Θ, its domain $\mathbf{dom}(\Theta)$ is defined as the set

$$\mathbf{dom}(\Theta) = \{X \in \mathcal{N} \mid Xz \to e \in \Theta\}$$

Every exception $X \in \mathbf{dom}(\Theta)$ must be associated with a unique pattern of Θ.

An exception handler Θ defines a unique map $[\![\Theta]\!] : \mathcal{N} \to \text{Values} \to \text{Expressions}$ as follows.

$$[\![\Theta]\!](X)(v) = \begin{cases} [v/z]e & \text{if } Xz \to e \in \Theta \\ \mathbf{raise}_X \ v & \text{otherwise} \end{cases}$$

We will frequently identify the handler Θ with the function $[\![\Theta]\!]$, and write $\Theta(X)(v)$ instead of $[\![\Theta]\!](X)(v)$. According to the above definition, if X is an exception such that $X \notin \mathbf{dom}(\Theta)$, then Θ handles X simply by propagating it further.

Example 5.1 Assuming X and Y are integer names, the following are well-formed expressions of the exception calculus.

(i) $(1 - \mathbf{raise}_X \ \mathbf{raise}_Y \ 10) \ \mathbf{handle} \ \langle Xx \to x + 2, Yy \to y + 3 \rangle$

(ii) $(1 - \mathbf{raise}_X \ 0) \ \mathbf{handle} \ \langle Xx \to (2 - \mathbf{raise}_Y \ x) \rangle \ \mathbf{handle} \ \langle Yy \to y \rangle$

(iii) $(1 - \mathbf{raise}_X \ 0) \ \mathbf{handle} \ \langle Yy \to (2 - \mathbf{raise}_X \ y) \rangle \ \mathbf{handle} \ \langle Xx \to x + 1 \rangle$

The expression evaluate to 13, 0 and 1, respectively. Expression (i) raises the exception Y, passing 10 along. This is handled by the pattern $Yy \to y + 3$, to produce 13. Expression (ii) raises X with value 0, but while handling X it raises Y with value 0, which is finally handled by the outside handler $\langle Yy \to y \rangle$, to produce 0. Expression (iii) raises X with 0, which is propagated by the inside handler, and then handled by the outside handler $\langle Xx \to x + 1 \rangle$, to return 1.

The type system of the calculus of exceptions consists of two judgments: one for typing expressions, and another one for typing exception handlers. The judgment for expressions has the form

$$\Sigma; \Delta \vdash e : A \, [C]$$

and it simply extends the judgment from the core fragment presented in Section 3 with the new rules for $\mathbf{raise}$ and $\mathbf{handle}$. The specific of the calculus is that the support C represents sets, collecting the exceptions that e is *allowed*

$$\frac{C \sqsubseteq D}{\Sigma; \Delta \vdash \langle \; \rangle : [C] \stackrel{A}{\Rightarrow} [D]}$$

$$\frac{\Sigma; (\Delta, z{:}A) \vdash e : B\,[D] \quad \Sigma; \Delta \vdash \langle \Theta \rangle : [C \setminus X] \stackrel{B}{\Rightarrow} [D] \quad X{:}A \in \Sigma}{\Sigma; \Delta \vdash \langle Xz \to e, \Theta \rangle : [C] \stackrel{B}{\Rightarrow} [D]}$$

$$\frac{\Sigma; \Delta \vdash e : A\,[C] \quad X \in C \quad X{:}A \in \Sigma}{\Sigma; \Delta \vdash \mathbf{raise}_X\ e : B\,[C]}$$

$$\frac{\Sigma; \Delta \vdash e : A\,[C] \quad \Sigma; \Delta \vdash \langle \Theta \rangle : [C] \stackrel{A}{\Rightarrow} [D]}{\Sigma; \Delta \vdash e\ \mathbf{handle}\ \langle \Theta \rangle : A\,[D]}$$

Fig. 2. Typing rules for exceptions.

to raise. Thus, $C \sqsubseteq D$ is defined as $C \subseteq D$ when C and D are viewed as sets (i.e., when the ordering and repetition of elements are ignored). By support weakening, e need not raise all the exceptions from its support C, but if an exception can be raised, then it must be in C. The judgment for exception handlers has the form

$$\Sigma; \Delta \vdash \langle \Theta \rangle : [C] \stackrel{A}{\Rightarrow} [D]$$

and the handler Θ will be given the type $[C] \stackrel{A}{\Rightarrow} [D]$ if: (1) Θ can handle exceptions from the support set C arising in a term of type A, and (2) during the handling, Θ is allowed to itself raise exceptions only from the support set D. The typing rules of both judgments are presented in Figure 2, and we briefly comment on them below.

An exception X can be raised only if it is accounted for in the support. Thus the rule for **raise** requires $X \in C$. The term $\mathbf{raise}_X\ e$ changes the flow of control, by passing e to the nearest handler. Because of that, the environment in which this term is encountered does not matter; we can type $\mathbf{raise}_X\ e$ by any arbitrary type B. In the rule for **handle**, the type and the support of the expression e must match the type and the domain support of the handler Θ. The exception handler $\langle \; \rangle$ only propagates whichever exceptions it encounters. Thus, if it is supplied an expression of support C it will produce an expression of the same support. To maintain the support weakening property, we allow the range support D of an empty handler to be a superset of C. Notice that the empty support handler may be assigned an arbitrary type A. The rule for nonempty exception handlers simply prescribes inductively checking each of the exception patterns in the handler. The type of each pattern variable z must match the type of the corresponding exception; this is the type of the value that the exception will be raised with. The handling terms e must all have the same type B, which would also be the type assigned to the handler

itself.

Example 5.2 The function `tail` below computes a tail of the argument integer list, raising an exception `EMPTY:unit` if the argument list is empty. The function `length` uses `tail` to compute the length of a list. Note that the range type of `tail` is $\square_{\text{EMPTY}}$`intlist`. This is required because the body of `tail` may raise an exception, and, as explained in the previous section, all the effects in function bodies must be boxed.

```
- choose (νEMPTY: unit.
    let fun tail (xs : intlist) : □_EMPTY intlist =
        (case xs
            of nil => box (raise_EMPTY ())
             | x::xs => box xs)
        fun length (xs : intlist) : int =
            (1 + length (unbox (tail xs)))
            handle <EMPTY z -> 0>
    in
      length [1,2,3,4]
    end);
  val it = 4;
```

Before we proceed to describe the operational semantics of the exception calculus, let us outline some of its properties and how they relate to other treatments of exceptions in functional languages.

First of all, exceptions in our calculus are second class. They are not values and cannot be bound to variables. Correspondingly, exceptions must be explicitly raised; raising a variable exception is not possible. Aside from this fact, when *local* exceptions are concerned (i.e., exceptions which do not originate from a function call, but are raised and handled in the body of the one and the same function), our calculus very much resembles Standard ML [19]. In particular, exceptions can be raised, and then handled, without forcing any changes to the type of the function. It is only when we want the function to propagate an exception so that it is handled by the caller, that we need to specifically mark the range type of that function with a $\square$-type.

It is also instructive to compare our calculus with the monadic formulation of exceptions. To that end, we recall the monadic program presented in the introduction section, where we assume that $f : \textbf{int} \Rightarrow \bigcirc\textbf{int}$.

```
handle (let comp x1 = f(1)
             comp x2 = f(2)
         in
             comp (x1 + x2)
         end) (λv. v) (λexn. 0)
```

The program adds the results of $f(1)$ and $f(2)$. If the evaluation of any of the

two function applications raises an exception, the overall computed result is zero.

In our calculus of exceptions, the equivalent of the above program may be written in several ways, depending on the evaluation order that the programmer may wish to specify. For example, let us assume that $X{:}E$ is an exception name, and that $f : \textbf{int} \to \Box_X \textbf{int}$. Then the behavior of the previous monadic program is exhibited by the following program in the calculus of exceptions.

```
(let val x1 = unbox f(1)
     val x2 = unbox f(2)
 in
     x1 + x2
end) handle <X exn -> 0>
```

However, the computations obtained by $f(1)$ and $f(2)$ are independent of each other, so there is no need to first evaluate and unbox $f(1)$ and then evaluate and unbox $f(2)$. For example, we could write the following program that computes the same results.

```
let box u1 = f 1
    box u2 = f 2
in
    (u1 + u2) handle <X exn -> 0>
end
```

The first two **let box** branches of this program evaluate the expressions $f(1)$ and $f(2)$ in that order to obtain boxed computations **box** e_1 and **box** e_2, but they *do not evaluate* e_1 and e_2. The computations e_1 and e_2 are substituted for u_1 and u_2, and only then is the execution of $(e_1 + e_2)$ attempted, in the order specified by the operational semantics of addition. Following a similar idea, an even more compact way to compute the sum of $f(1)$ and $f(2)$ is given simply as

```
(unbox f(1) + unbox f(2)) handle <X exn -> 0>
```

As a conclusion, our calculus of exceptions allows programs that are uncommitted about the evaluation order of its expressions, when these expressions do not depend on each other. The evaluation order is eventually determined by the operational semantics, but it is not necessary to make this order explicit in the program.

Note that the modal formulation of exceptions may also benefit efficiency. Because we only consider for evaluation those expressions with empty support, the exceptional computation boxed in the expression $e : \Box_X A$ will only be evaluated within the scope of some handler for X. As a consequence of the progress theorem (Theorem 4.9), this evaluation can only terminate with a *value*, and cannot result with an unhandled exception. This contrasts the monadic calculus of exceptions where unhandled exceptions are given the status of values (as explained in the introduction), and this incurred the need for

tagging and tag checking. In the modal case, raised unhandled exceptions are not values of the modal type, so there is no need for tagging.

The operational semantics of the exception calculus is a simple extension of the semantics of the core fragment. The evaluation judgment has the same form

$$\Sigma, e \longmapsto \Sigma', e'$$

We only need to extend the syntactic categories of evaluation contexts and redexes, and define primitive reductions for the new redexes.

$$\textit{Evaluation contexts } E ::= \dots \mid \textbf{raise}_X\ E \mid E\ \textbf{handle}\ \langle\Theta\rangle$$

$$\textit{Pure contexts} \qquad P ::= [\ ] \mid P\ e \mid v\ P \mid \textbf{let box}\ u = P\ \textbf{in}\ e \mid$$

$$\textbf{choose}\ P \mid \textbf{raise}_X\ P$$

$$\textit{Redexes} \qquad\qquad r ::= \dots \mid v\ \textbf{handle}\ \langle\Theta\rangle \mid P[\textbf{raise}_X\ v]\ \textbf{handle}\ \langle\Theta\rangle$$

We have already explained that each exception handler can handle all exceptions. It is only that some exceptions are handled in a specified way, while others are handled by simple propagation. This will simplify the operational semantics somewhat, because in order to find the handler capable of handling a particular **raise** we only need to find the nearest handler preceding this **raise**. For that purpose, we select a special subclass of *pure evaluation contexts*, which are pure in the sense that they do not contain any exception handlers acting on the hole of the context. It can easily be shown that each evaluation context E is either pure, or there exist unique evaluation context E' and pure context P', such that $E = E'[P'\ \textbf{handle}\ \langle\Theta\rangle]$.

The primitive reduction on the new redexes follows.

$$\Sigma, v\ \textbf{handle}\ \langle\Theta\rangle \longrightarrow \Sigma, v$$

$$\Sigma, P[\textbf{raise}_X\ v]\ \textbf{handle}\ \langle\Theta\rangle \longrightarrow \Sigma, \Theta(X)(v)$$

The first reduction exploits the fact that values are exception free, and therefore simply fall through any handler. The second reduction chooses the closest handler for any particular raise. It also requires that only values be passed along with the exceptions; the operational semantics demands that before an exception is raised, its argument must be evaluated. If it so happens that the evaluation of the argument raises another exception, this later one will take precedence and actually be raised. This is already illustrated in the first term from Example 5.1, where it is the exception Y which is raised and eventually handled.

The structural properties and the type soundness of the core fragment readily extend to the exception calculus. Here we only list some specific additional lemmas.

19

Lemma 5.3 (Handler substitution principle) *If $\Sigma; \Delta \vdash e_1 : A\,[C]$ and $\Sigma; (\Delta, u{:}A[C]) \vdash \langle \Theta \rangle : [D'] \overset{B}{\Rightarrow} [D]$, then $\Sigma; \Delta \vdash \langle [e_1/u]\Theta \rangle : [D'] \overset{B}{\Rightarrow} [D]$*

Lemma 5.4 (Unique decomposition) *For every expression e, either:*

(i) *e is a value, or*

(ii) *$e = P[\mathbf{raise}_X\ v]$, for a unique pure context P, or*

(iii) *$e = E[r]$ for a unique evaluation context E and a redex r.*

The calculus satisfies the same preservation and progress theorems of the core fragment.

6 Conclusions, related and future work

In this paper, we have used the necessitation type operator $\Box$ from the modal logic CS4 to internalize exceptional computations. In modal logic, the operator $\Box$ corresponds to universal quantification, so that $\Box A$ is true if and only if A is true at all possible worlds. The application to exceptions is based on the following observation: a computation of type A that may raise exceptions from the set C, can be viewed as executing – without getting stuck – under *all* possible handlers for the effects in C. This statement specifies universal quantification over handlers, bounded by the support C. We adopt that handlers correspond to worlds in modal logic, and thus a described effectful computation can be typed with a bounded universal type $\Box_C A$. We also point out that $\Box$ is usually categorically modeled by a comonad, rather than a monad, as explored in [17,1,3].

Our formulation of exceptions have certain advantages over the monadic representation of exceptions [21,34]. As a first distinction, the modal calculus allow programs that are uncommitted about the evaluation order of their subexpressions, when these subexpression do not really depend on each other. The evaluation order is eventually determined by the operational semantics, but it is not necessary to make this order explicit in the program. As a second distinction, monadic representation of exceptions treats raised but not handled exceptions as values, and thus forces tagging and run-time tag checking. This is avoided in the modal setting, and may improve the efficiency of exceptional computations.

There is also an extensive literature on the non-monadic treatment of exceptions and effects in general in the style of *type-and-effect systems*. We just list some of the representative papers here [11,18,14,13,30,31,32]. The approach usually taken by type-and-effect systems is to extend the language with a type of *effectful functions* $A \overset{C}{\to} B$. Here, C is a set of effects that the evaluation of the function body may cause. The characteristic typing rules

are usually a variation on the following.

$$\frac{\Sigma; (\Delta, x{:}A) \vdash e : B\,[C]}{\Sigma; \Delta \vdash \lambda x{:}A.\ e : A \overset{C}{\to} B\,[\,]}\ (*)$$

$$\frac{\Sigma; \Delta \vdash e_1 : A \overset{C}{\to} B\,[D_1] \qquad \Sigma; \Delta \vdash e_2 : A\,[D_2]}{\Sigma; \Delta \vdash e_1\ e_2 : B\,[C, D_1, D_2]}\ (**)$$

It is interesting here to draw a parallel between the operational behavior of the modal constructors in our language, and that of λ-abstraction in type-and-effect systems. Both constructs suspend the evaluation of their bodies, so one may view **box** e in the modal system as somehow corresponding to $\lambda x{:}1.\ e$ (where $x \notin \mathbf{fv}(e)$) in the type-and-effect systems. Does this similarity indicate that modal constructs are perhaps superfluous and may be removed in favor of functional abstraction?

The answer to the above question is negative, as the import of the modal constructors in our language is not solely operational. Their main role is *not* to suspend the evaluation of expressions, but to *internalize* the notion of effectful computation. For example, note that the rules (*) and (**) are not locally complete, and therefore are not logically justified. The local expansion of $e : A \overset{C}{\to} B\,[D]$ is given as

$$e : A \overset{C}{\to} B\,[D] \quad \Longrightarrow_E \quad \lambda x.\ e\ x : A \overset{C,D}{\to} B$$

and the expression e has a *different* type and support from its expansion. To contrast this, local expansion in the modal calculus of exceptions preserves types and supports, as can easily be checked from the equation below.

$$e : \Box_C A\,[D] \quad \Longrightarrow_E \quad \mathbf{let\ box}\ u = e\ \mathbf{in\ box}\ u$$

In fact, when effectful computations are internalized as a separate semantic category which is different from functions, then functions and function types are freed from the responsibility to track effects. Moreover, in such situations functions are usually required to be *pure*. This is the case in our modal calculus of exceptions, but is also true of the monadic λ-calculus [21,33]. In both calculi, a function body may contain an effect only if the effect is encapsulated by a computation-forming construct. And in both calculi, the range type of such a function will be a computation type (monadic type in the monadic calculus, and a modal type $\Box_C A$ in the modal calculus).

A treatment of exceptions in Haskell is considered by Peyton Jones et al. in [26]. It is interesting that this paper does not use the exception monad in order to extend the underlying language, but rather implements imprecise exceptions. With imprecise exceptions, the program is not guaranteed to always

report the same exception that would be encountered by a straightforward sequential execution. In this calculus, an exceptional expression evaluates to an *exceptional value*, which has a whole set of possible exceptions associated with it. The associated exceptions are the ones that the expression may have potentially raised. Informally, this associated exception set compares to our notion of support.

At run time, of course, it is not a whole set of exceptions that an evaluation of an expression returns. What is returned is the first expression out of this set, that got raised. It is important that the returned exception may change in different compilations and runs, because the optimizations performed at different compilations may result with different order of evaluation. Obviously, the semantics of the calculus cannot depend on optimizations, so it assumes that the returned exception is chosen *non-deterministically* out of the possible set. This also has a parallel in our modal calculus, which would have been non-deterministic except for the operational semantics from Sections 4 and 5. Our operational semantics only resolves the non-determinism in choosing which exception from a set of possibilities (bounded by the support) should be raised, and thus resolves which value is eventually returned by the computation. But, the operational semantics is not needed for serializing a pair of exceptional computations, because exceptional computations cannot interfere to change each others results.

Another exception calculus is presented by de Groote in [6]. It is a call-by-value calculus which uses separate binding mechanisms to introduce exceptions into the computation. However, because of the lack of modal or monadic types, it cannot enforce that values of the language are effect-free. In the specific cases when values turn out to be effect-free, the calculus implements the Standard ML exception mechanism. This paper also discusses the logical content of exceptions, and relationship with classical logic. The exception mechanism of Java relates to our calculus as well, as Java methods must be labeled by the exceptions they can raise [12]. We also refer to several works on the catch-and-throw calculi, which can be viewed as a specific simplification of exceptions [22,15]. These calculi also do not consider a type constructor for exceptional computations, and thus must restrict the way exceptions are introduced, propagated and handled. In particular, the handler for any particular exception must be determined statically; it is the nearest handler whose scope encloses the point of raising at the time of the definition of the program, rather than at run time.

We represent exceptions in the calculus by names. Names are labels that can be dynamically introduced into the calculus, and are subject to a typing discipline which ensures that no name escapes the scope of its introducing binder. That modal necessity can be very naturally extended with the notion of names was argued in [23]. The calculus from that paper is a direct precursor to the effect system we presented here. It is motivated by the work of Pitts and Gabbay on Nominal Logic and FreshML [29,28] which introduce names

as urelements of Fraenkel-Mostowski set theory.

Comonads have also been considered previously for purposes of modeling intensional computations [4], and for representing effectful computations that may depend on the run-time environment, but do not change it [16]. The cited papers, however, do not make the connection with modal logic and exception handling.

Finally, practical programming with our system may be hampered by the verbosity of support annotations on types and the fact that the effect instances are second-class object in the calculus. Therefore, we will need to investigate the questions of type and support inference and develop new abstraction mechanism which can hide the unwanted support and result in a more practical language (perhaps using support polymorphism [23]). The system presented here is logically motivated and fully explicit about the support of terms, and is therefore a solid theoretical basis for such investigations.

References

[1] N. Alechina, M. Mendler, V. de Paiva, and E. Ritter. Categorical and Kripke semantics for Constructive S4 modal logic. In L. Fribourg, editor, *International Workshop on Computer Science Logic, CSL'01*, volume 2142 of *Lecture Notes in Computer Science*, pages 292–307, Paris, 2001. Springer.

[2] P. N. Benton, G. M. Bierman, and V. de Paiva. Computational types from a logical perspective. *Journal of Functional Programming*, 8(2):177–193, March 1998.

[3] G. M. Bierman and V. C. V. de Paiva. On an intuitionistic modal logic. *Studia Logica*, 65(3):383–416, 2000.

[4] S. Brookes and S. Geva. Computational comonads and intensional semantics. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, *Application of Categories in Computer Science*, volume 177 of *London Mathematical Society Lecture Notes*, pages 1–44. Cambridge University Press, Cambridge, 1992.

[5] R. Davies and F. Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, 2001.

[6] P. de Groote. A simple calculus of exception handling. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 201–215. Springer, 1995.

[7] M. Fairtlough and M. Mendler. Propositional lax logic. *Information and Computation*, 137(1):1–33, 1997.

[8] A. Filinski. Representing monads. In *Symposium on Principles of Programming Languages, POPL'94*, pages 446–457, Portland, Oregon, 1994.

[9] A. Filinski. *Controlling Effects*. PhD thesis, Carnegie Mellon University, 1996.

[10] A. Filinski. Representing layered monads. In *Symposium on Principles of Programming Languages, POPL'99*, pages 175–188, San Antonio, Texas, 1999.

[11] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *Conference on LISP and Functional Programming*, pages 28–38, Cambridge, Massachusetts, 1986.

[12] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1997.

[13] P. Jouvelot and D. Gifford. Algebraic reconstruction of types and effects. In *Symposium on Principles of Programming Languages, POPL'91*, pages 303–310, Orlando, Florida, 1991.

[14] P. Jouvelot and D. K. Gifford. Reasoning about continuations with control effects. In *Conference on Programming Language Design and Implementation, PLDI'89*, pages 218–226, Portland, Oregon, 1989.

[15] Y. Kameyama and M. Sato. Strong normalizability of the non-deterministic catch/throw calculi. *Theoretical Computer Science*, 272(1–2):223–245, 2002.

[16] R. B. Kieburtz. Codata and comonads in Haskell. Unpublished. Available from `http://www.cse.ogi.edu/~dick`, 1999.

[17] S. Kobayashi. Monad as modality. *Theoretical Computer Science*, 175(1):29–74, 1997.

[18] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Symposium on Principles of Programming Languages, POPL'88*, pages 47–57, San Diego, California, 1988.

[19] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[20] E. Moggi. Computational lambda-calculus and monads. In *Symposium on Logic in Computer Science, LICS'89*, pages 14–23, Asilomar, California, 1989.

[21] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[22] H. Nakano. A constructive formalization of the catch and throw mechanism. In *Symposium on Logic in Computer Science, LICS'92*, pages 82–89, Santa Cruz, California, 1992.

[23] A. Nanevski. Meta-programming with names and necessity. In *International Conference on Functional Programming, ICFP'02*, pages 206–217, Pittsburgh, Pennsylvania, 2002. A significant revision is available as a technical report CMU-CS-02-123R, Computer Science Department, Carnegie Mellon University.

[24] A. Nanevski. A modal calculus for effect handling. Technical Report CMU-CS-03-149, Computer Science Department, Carnegie Mellon University, June 2003.

[25] A. Nanevski. *Functional Programming with Names and Necessity*. PhD thesis, Computer Science Department, Carnegie Mellon University, Aug. 2004.

[26] S. Peyton Jones, A. Reid, T. Hoare, S. Marlow, and F. Henderson. A semantics for imprecise exceptions. In *Conference on Programming Language Design and Implementation, PLDI'99*, pages 25–36, Atlanta, Georgia, 1999.

[27] F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001.

[28] A. M. Pitts. Nominal logic: A first order theory of names and binding. In N. Kobayashi and B. C. Pierce, editors, *Theoretical Aspects of Computer Software*, volume 2215 of *Lecture Notes in Computer Science*, pages 219–242. Springer, 2001.

[29] A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer, 2000.

[30] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

[31] J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, 1994.

[32] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[33] P. Wadler. The essence of functional programming. In *Symposium on Principles of Programming Languages, POPL'92*, pages 1–14, Albequerque, New Mexico, 1992.

[34] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995.

[35] P. Wadler. The marriage of effects and monads. In *International Conference on Functional Programming, ICFP'98*, pages 63–74, Baltimore, Maryland, 1998.

[36] P. Wickline, P. Lee, and F. Pfenning. Run-time code generation and Modal-ML. In *Conference on Programming Language Design and Implementation, PLDI'98*, pages 224–235, Montreal, Canada, 1998.

[37] P. Wickline, P. Lee, F. Pfenning, and R. Davies. Modal types as staging specifications for run-time code generation. *ACM Computing Surveys*, 30(3es), 1998.

[38] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.