A Modal Calculus for Named Control Effects

Aleksandar Nanevski
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
aleks@cmu.edu

Abstract

The monadic formulation of exceptions forces a programming style in which the program itself must specify a total ordering on the evaluation of exceptional computations. Moreover, unless a call-by-name strategy is used, values of monadic types must be tested before they are used, in order to determine whether they correspond to a raised exception or not.

In this paper we propose a type system that internalizes the notion of exceptional computations, but avoids the above two properties. In our calculus, exceptional computations need not be ordered explicitly by the program, unless one of them actually depends on the other. Furthermore, the type system ensures that run-time tests are not needed to determine if an exceptional computation evaluates to a value or a raised exception.

To this end, we use the necessitation operator $\Box$ from a version of constructive modal logic. The idea is applicable to other control effects as well, and we use it to develop novel calculi for catch-and-throw and composable continuations.

1 Introduction

Monads and the monadic λ-calculus [19, 20, 29, 31, 32] present a type theoretic method for grafting effectful features onto a purely functional language. Monads are type constructors (satisfying certain categorical properties) which are used to internalize the notion of effectful computation. The idea is to limit the appearance of effects to terms of monadic type. Therefore, monadic types separate the possibly impure subterms from the pure ones, and ensure a disciplined propagation of effects. Furthermore, the monadic typing discipline forces monadic computations to 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 term may cause.

For example, in the literature today, the customary way of formalizing type-safe calculi of exceptions is via the exception monad [20, 31]. This approach defines the monadic type $\bigcirc A$ as $\bigcirc A = A+E$ where E is the type of the exception that can be raised. Then it introduces constructors **comp** and **let comp** using the following definitions that assume the standard formulation of disjoint sums.

$$\begin{array}{rcl} \textbf{comp}\ e & = & \textbf{inl}\ e \\ \textbf{let comp}\ x = e_1\ \textbf{in}\ e_2 & = & \textbf{case}\ \ e_1\ \textbf{of inl}\ x \Rightarrow e_2 \mid \textbf{inr}\ y \Rightarrow \textbf{inr}\ y \end{array}$$

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 $\bigcirc$.

$$\begin{array}{rcl} \textbf{raise} & : & E \Rightarrow \bigcirc A \\ \textbf{raise}\ e & = & \textbf{inr}\ e \\ \textbf{handle} & : & \bigcirc A \Rightarrow (E \Rightarrow A) \Rightarrow A \\ \textbf{handle}\ e\ h & = & \textbf{case}\ \ e\ \textbf{of inl}\ v \Rightarrow v \mid \textbf{inr}\ exn \Rightarrow h\ exn \end{array}$$

The constructor **raise** takes an 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 an 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. For example, let us assume that $\bigcirc A = A+E$ is an exception monad, and 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 problems with this approach, which complicate the programming and efficiency of exceptional computations [24, 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 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. Raised-but-unhandled exceptions are first-class objects in the language.

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 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 [9, 10, 11]. Reflection and reification are translations between an effectful source language (which provides the syntax for programming) and a monadic meta language (which provides the semantics). As concluded in [9], however, the translations still incur the operational penalties of tagging and tag checking.

In this paper, we propose a novel formulation of exceptions which avoids serialization and tagging. The approach uses only natural deduction, without any indirect translations. The idea is inspired by modal logic, which is a logic for reasoning about truth across various worlds. In modal logic, a proposition A may be true in some worlds, but not true in some others. Modal logic that we consider features a propositional operator $\Box$ (also referred to as *necessity*) which represents universal quantification over worlds: $\Box A$ is true if A is true in all worlds.

For our computational application, propositions correspond to types of values, and worlds from modal logic correspond to exception handlers. Then consider a computation of type A whose execution may raise the exceptions from the set C. This computation may evaluate – without getting stuck – in the scope of *all* handlers capable of handling the exceptions from the set C. Thus, it should be assigned a type $\Box_C A$ that corresponds to a *universal quantification* over exception handlers, *bounded* by C. The term assignment corresponding of a λ-calculus for modal logic will then give us a language adequate for representing exceptions. Furthermore, other control effects could be subjected to the same analysis and formulation.

To substantiate this claim, we present novel calculi for exceptions, catch-and-throw and composable continuations, which are based on the above intuition. The presented calculi follow the introduction/elimination pattern of natural deduction; effects are introduced by their introduction forms, and are eliminated by their handling forms. All presented languages are extensions of the λ-calculus; if the programmer is interested only in pure computations, the constructs related to modal logic need not be used.

The calculi from this paper are implemented, and the examples are tested. Sources for the interpreters are available at `http:/www.cs.cmu.edu/~aleks/papers/effects/nubox.tar.gz` An extended version of the paper, with detailed proofs is available at `http://www.cs.cmu.edu/~aleks/papers/effects/effects.ps`

2 Modal $\lambda^\Box$-calculus

The starting point for the development of our language for control effects is the $\lambda^\Box$-calculus of [25, 6]. The $\lambda^\Box$ is the proof-term system for the necessitation fragment of Constructive S4 (CS4) modal logic, and it was first considered in functional programming in the context of specialization for purposes of run-time code generation [6, 33, 34]. The syntax of $\lambda^\Box$ is summarized below, where we use b to stand for a predetermined set of base types.

Types	A	$::=$	$b \mid A_1 \to A_2 \mid \Box A$
Terms	e	$::=$	$x \mid u \mid \lambda x{:}A.\ e \mid e_1\ e_2 \mid$ $\mathbf{box}\ e \mid \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2$
Value variable contexts	Γ	$::=$	$\cdot \mid \Gamma, x{:}A$
Expression variable contexts	Δ	$::=$	$\cdot \mid \Delta, u{:}A$

The most important feature of the calculus is the type constructor $\Box$ which is referred to as *modal necessity*, as in the CS4 modal logic it is a necessitation modifier on propositions [25]. For the purposes of this paper, a useful operational intuition is to consider the type $\Box A$ 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 *effectful* (or *impure*) computation in the subsequent section.

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

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

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

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

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

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

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

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 the context Γ of values. The context Γ may therefore be removed from the typing rules, to obtain the following system.

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

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

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

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 which 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 which 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 language from [32], 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. This fragment will be a common part of all the effect calculi we develop next. The relationship between names and effects, and how various 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.

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

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) which 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 $A \not\twoheadrightarrow B$, 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 is hypothetical and 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 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

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

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

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

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

$$\frac{(\Sigma, X{:}A);\Delta \vdash e : B\,[\,] \quad X \notin \mathbf{fn}(A,B,\Delta)}{\Sigma;\Delta \vdash \nu X{:}A.\ e : A \not\twoheadrightarrow B\,[C]} \qquad \frac{\Sigma;\Delta \vdash e : A \not\twoheadrightarrow B\,[C]}{\Sigma;\Delta \vdash \mathbf{choose}\ e : B\,[C]}$$

Figure 1. Type system of the core fragment.

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 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 at al. [1] and Bierman and de Paiva [2], for a detailed discussion.

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 $A \nrightarrow B$. It picks a fresh name and substitutes it for the bound name in the ν-abstraction.

Example 3 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.

1. $\vdash \lambda x.\ \mathbf{box}\ x : A \to \Box_D A$
2. $\vdash \lambda x.\ \mathbf{let\ box}\ u = x\ \mathbf{in}\ u : \Box_{C_1} A \to A\,[C]$
3. $\vdash \lambda x.\ \mathbf{let\ box}\ u = x\ \mathbf{in\ box}\ u : \Box_{C_1} A \to \Box_C A$
4. $\vdash \lambda x.\ \mathbf{let\ box}\ u = x\ \mathbf{in\ box\ box}\ u : \Box_{C_1} A \to \Box_D \Box_C A$
5. $\vdash \lambda x.\ \lambda y.\ \mathbf{let\ box}\ u = x\ \mathbf{in\ let\ box}\ v = y\ \mathbf{in\ box}\ u\ v :$ $\Box_{C_1}(A \to B) \to \Box_{C_2} A \to \Box_C B$

Example 4 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** $((\lambda x.\ \mathbf{box}\ e_2)\ e_1)$, 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

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

Example 5 Anticipating Section 5, suppose that our language contains the term constructor **raise**, such that $\mathbf{raise}_X\ e$ 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.

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

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

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

declares a new exception Y and then raises it within a computation $(\mathbf{box\ raise}_Y\ e_2){:}\Box_{X,Y} A$. 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 [35]. 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.

$$\begin{array}{rcl}
\Sigma, (\lambda x.\ e)\ v & \longrightarrow & \Sigma, [v/x]e \\
\Sigma, \mathbf{let\ box}\ u = \mathbf{box}\ e_1\ \mathbf{in}\ e_2 & \longrightarrow & \Sigma, [e_1/u]e_2 \\
\Sigma, \mathbf{choose}\ (\nu X{:}A.\ e) & \longrightarrow & (\Sigma, Y{:}A), [Y/X]e, \quad Y \notin \mathbf{dom}(\Sigma)
\end{array}$$

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

Example 6 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 5.

$(X{:}A), \mathbf{choose}\ (\nu Y{:}A.\ (\lambda x{:}\Box_{X,Y} A.\ e_1)\ (\mathbf{box\ raise}_Y\ e_2)) \longmapsto$
$(X{:}A, Z{:}A), (\lambda x{:}\Box_{X,Z} A.\ e_1)\ (\mathbf{box\ raise}_Z\ e_2) \longmapsto$
 where Z 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 1 (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 2 (REPLACEMENT). *If* $\Sigma; \Delta \vdash E[e] : A\,[C]$, *then there exist a type* B *such that*
1. $\Sigma; \Delta \vdash e : B\,[C]$, *and*
2. *if* Σ', Δ' *extend* Σ, Δ, *and* $\Sigma'; \Delta' \vdash e' : B\,[C]$, *then* $\Sigma'; \Delta' \vdash E[e'] : A\,[C]$

LEMMA 3 (CANONICAL FORMS). *Let* v *be a closed value such that* $\Sigma; \cdot \vdash v : A\,[C]$. *Then the following holds:*
1. *if* $A = A_1 \to A_2$, *then* $v = \lambda x{:}A_1.\ e$ *and* $\Sigma; x{:}A_1 \vdash e : A_2\,[\ ]$
2. *if* $A = \Box_D B$, *then* $v = \mathbf{box}\ e$ *and* $\Sigma; \cdot \vdash e : B\,[D]$

3. *if $A = A_1 \twoheadrightarrow A_2$, 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 (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 5 (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 6 (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 7 (UNIQUE DECOMPOSITION). *For every expression e, either:*

1. *e is a value, or*
2. *$e = E[r]$ for a unique evaluation context E and a redex r.*

THEOREM 8 (PROGRESS). *If $\Sigma;\cdot \vdash e : A\,[\,]$, then either*

1. *e is a value, or*
2. *there exists a term e' and a context Σ', such that $\Sigma, e \longmapsto \Sigma', e'$.*

PROPOSITION 9 (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.

Exception handlers	Θ	$::=$	$\cdot \mid Xz \to e, \Theta$
Terms	e	$::=$	$\ldots \mid \mathbf{raise}_X\ e \mid e\ \mathbf{handle}\ \langle\Theta\rangle$

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 **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 $[\![\Theta]\!] : \mathcal{N} \to \mathit{Values} \to \mathit{Expressions}$ 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.

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

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

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

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

Figure 2. Typing rules for exceptions.

Example 7 Assuming X and Y are integer names, the following are well-formed expressions of the exception calculus.

1. $(1 - \mathbf{raise}_X\ \mathbf{raise}_Y\ 10)\ \mathbf{handle}\ \langle Xx \to x + 2, Yy \to y + 3\rangle$
2. $(1 - \mathbf{raise}_X\ 0)\ \mathbf{handle}\ \langle Xx \to (2 - \mathbf{raise}_Y\ x)\rangle\ \mathbf{handle}\ \langle Yy \to y\rangle$
3. $(1 - \mathbf{raise}_X\ 0)\ \mathbf{handle}\ \langle Yy \to (2 - \mathbf{raise}_X\ y)\rangle\ \mathbf{handle}\ \langle Xx \to x + 1\rangle$

The expression evaluate to 13, 0 and 1, respectively. Expression (1) raises the exception Y, passing 10 along. This is handled by the pattern $Yy \to y + 3$, to produce 13. Expression (2) 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 (3) 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 **raise** and **handle**. The specific of the calculus is that the support C represents sets, collecting the exceptions that e is *allowed* 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

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

and the handler Θ will be given the type $[C] \overset{A}{\Rightarrow} [D]$ 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 8 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 $\Box_{\mathtt{EMPTY}}$`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) : □EMPTYintlist =
      (case xs
         of nil => box (raiseEMPTY ())
          | 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 [18]. 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 $\Box$-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 $f : \mathbf{int} \Rightarrow \bigcirc\mathbf{int}$.

```
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 $f : \mathbf{int} \rightarrow \Box_X\mathbf{int}$. 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, because the computations obtained by f 1 and f 2 are independent of each other. 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 (Theorem8), 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.

Evaluation contexts	E	$::=$	$\ldots \mid \mathbf{raise}_X\ E \mid E\ \mathbf{handle}\ \langle\Theta\rangle$
Pure contexts	P	$::=$	$[\ ] \mid P\ e \mid v\ P \mid \mathbf{let\ box}\ u = P\ \mathbf{in}\ e \mid \mathbf{choose}\ P \mid \mathbf{raise}_X\ P$
Redexes	r	$::=$	$\ldots \mid v\ \mathbf{handle}\ \langle\Theta\rangle \mid P[\mathbf{raise}_X\ v]\ \mathbf{handle}\ \langle\Theta\rangle$

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.

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

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 7, 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.

LEMMA 10 (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 11 (UNIQUE DECOMPOSITION). *For every expression e, either:*

1. *e is a value, or*
2. $e = P[\textbf{raise}_X\ v]$, *for a unique pure context P, or*
3. $e = E[r]$ *for a unique evaluation context E and a redex r.*

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

6 Catch-and-throw calculus

The catch-and-throw calculus is a simplification of the calculus of exceptions. We consider it here in its own right in order to illustrate a different notion of handling. It will also provide some intuition for the calculus of composable continuation in Section 7. In the catch-and-throw calculus, names are associated with labels to which the program can jump. Informally, **catch** establishes a destination point for a jump and assigns a name to it, and **throw** jumps to the established point.

$$\textit{Terms}\quad e \quad ::= \quad \ldots \mid \textbf{throw}_X\ e \mid \textbf{catch}_X\ e$$

The **throw** and **catch** can be viewed as restrictions of **raise** and **handle**; **catch** handles a **throw** by immediately returning the value associated with the throw.

The typing judgment $\Sigma;\Delta \vdash e : A[C]$ establishes that e has type A and may throw to destination points whose names are listed in the support C. The supports are sets, rather than sequences, just like in the calculus of exceptions. The typing rules of the calculus are presented in Figure 3. A **throw** to a destination point is allowed

$$\frac{\Sigma;\Delta \vdash e : A[C] \quad X \in C \quad X{:}A \in \Sigma}{\Sigma;\Delta \vdash \textbf{throw}_X\ e : B[C]} \qquad \frac{\Sigma;\Delta \vdash e : A[C,X] \quad X{:}A \in \Sigma}{\Sigma;\Delta \vdash \textbf{catch}_X\ e : A[C]}$$

Figure 3. Typing rules for catch and throw.

only if the destination point is present in the support set. A **catch** establishes a destination point by placing it in the support set against which the argument expression is checked.

Example 9 The following terms (adapted from [14]) are well-typed in our catch-and-throw calculus.

```
choose (νX:int.
  (λf:int->□Xint.
            let box u = f 0
            in
               catchX (1 + u)
            end) (λy:int. box (throwX y)))

choose (νX:int.
  (λf:int->□Xint.
                let box u = f 0
                in
                    1 + catchX u
                end) (λy:int. box (throwX y)))
```

The first term evaluates to 0, because the addition with 1 is skipped over by a **throw**. In the second term, the **catch** is pushed further inside, to preserve this addition, and so the term evaluates to 1.

Example 10 The program segment below defines a recursive function for multiplying elements of an integer list. If an element is found to be equal to 0, then the whole product will be 0, so rather than uselessly performing the remaining computation, we terminate by an explicit **throw** outside of the recursive function.

```
- choose (νEXIT:int.
    let fun mult (xs : intlist) : □EXITint =
            case xs
             of nil => box 1
            | x::xs =>
                if x = 0 then box (throwEXIT 0)
                else
                  let box u = mult xs in box(x * u)
    in
        catchEXIT (unbox (mult [2, 1, 0, 3]))
    end);

val it = 0 : int
```

The evaluation judgment of the catch-and-throw calculus is again a straightforward extension of the evaluation judgment $\Sigma, e \longmapsto \Sigma', e'$ of the core fragment from Section 3. We first need to define the new redexes, corresponding to the new **catch** and **throw** constructs, and extend the syntactic category of evaluation contexts.

$$\begin{array}{lcll} \textit{Redexes} & r & ::= & \ldots \mid \textbf{catch}_X\ v \mid \textbf{catch}_X\ E[\textbf{throw}_X\ v] \\ \textit{Evaluation contexts} & E & ::= & \ldots \mid \textbf{catch}_X\ E \mid \textbf{throw}_X\ E \end{array}$$

In the redex $\textbf{catch}_X\ E[\textbf{throw}_X\ v]$ it is assumed that the context E does not contain a $\textbf{catch}_X$ phrase acting on the hole of E. The primitive reductions on the new redexes are defined as follows.

$$\begin{array}{rcl} \Sigma, \textbf{catch}_X\ v & \longrightarrow & \Sigma, v \\ \Sigma, (\textbf{catch}_X\ E[\textbf{throw}_X\ v]) & \longrightarrow & \Sigma, v \end{array}$$

Similar to the exception calculus, values simply fall through the **catch**, and every **throw** is caught by the closes surrounding **catch** with the appropriate name. The operational semantics of catch-and-throw requires that only values be passed along a **throw**. Thus, of possibly nested throws, only the last one will actually be subject to catching.

The structural properties lemmas of the core fragment only require a minor modification for Unique decomposition.

LEMMA 12 (UNIQUE DECOMPOSITION). *For every expression e, either:*

1. *e is a value, or*
2. *$e = E[\mathbf{throw}_X\ v]$, for a unique context E which does not catch X, or*
3. *$e = E[r]$ for a unique evaluation context E and a redex r.*

The calculus satisfies the same preservation and progress theorems (Theorems 5 and 8).

7 Composable continuations

Similar to the catch-and-throw calculus, composable continuations use names to label destination points to which a program can jump. A destination point for a jump is established with the construct **mark** which also assigns a name to it; thus, it is similar to **catch** from the previous section. The jump itself is performed by **recall**, which corresponds to **throw** from the catch-and-throw calculus. The exact syntax of the calculus is defined as follows.

$$\textit{Terms}\quad e \quad ::= \quad \ldots \mid \mathbf{recall}_X\ k.\ e \mid \mathbf{mark}_X\ e$$

The differences from the catch-and-throw calculus, however, arise from the following property, which is characteristic for continuation calculi: unlike **throw**, when the construct $\mathbf{recall}_X\ k.\ e$ is evaluated, it captures into the variable k the part of the surrounding term between this **recall** and corresponding **mark** which precedes it; k may then be used to compute the value of e that is passed along with the jump. It is important that the evaluation of e is undertaken in the changed environment from which the part captured in k has been *removed*. More specifically, e itself will not be able to recall to mark points which were defined in the captured and removed part.

The above operational intuition is formalized by the following definitions of evaluation contexts, redexes and primitive reductions.

$$\begin{array}{lrcl}
\textit{Evaluation contexts} & E & ::= & \ldots \mid \mathbf{mark}_X\ E \\
\textit{Pure contexts} & P & ::= & [\,] \mid P\ e_1 \mid v_1\ P \mid \mathbf{let\ box}\ u = P\ \mathbf{in}\ e \mid \mathbf{choose}\ P \\
\textit{Redexes} & r & ::= & \ldots \mid \mathbf{mark}_X\ v \mid \mathbf{mark}_X\ P[\mathbf{recall}_Y\ k.\ e]
\end{array}$$

$$\begin{array}{rcl}
\Sigma, \mathbf{mark}_X\ v & \longrightarrow & \Sigma, v \\
\Sigma, (\mathbf{mark}_X\ P[\mathbf{recall}_X\ k.\ e]) & \longrightarrow & \Sigma, [K/k]e, \\
& & \text{where } K = \lambda x.\ \mathbf{let\ box}\ u = x\ \mathbf{in\ box}\ P[u]
\end{array}$$

Example 11 The following expressions (adapted from [3, 30]), are well-formed examples in our calculus of composable continuations.

```
e₁ =  1 + mark_X (10 + recall_X f:□_X int->□_X int.
             let box u = f (f (box 100))
             in
                 mark_X u
             end)

e₂ = 1 + mark_X (10 + recall_X f. 100)

e₃ = 1 + mark_X (10 + recall_X f.
             let box u1 = f (box 100)
                 box u2 = f (box 1000)
             in
                 mark_X (u1 + u2)
             end)
```

The expressions evaluate to 121, 101 and 1121, respectively. In each of these examples, the continuation variable $f : \Box_X\mathbf{int} \to \Box_X\mathbf{int}$ is bound to $\lambda x.\ \mathbf{let\ box}\ v = x\ \mathbf{in\ box}\ (10+v)$. It captures and internalizes the evaluation environment $(10+-)$, which is enclosed between **mark** and **recall**. Notice that upon capturing of the environment into f, the delimiting **mark** is *removed* from the reduct.

It is the expressions bound to k that is actually referred to as a *composable continuation* (and other names in use are: partial continuation, delimited continuation and subcontinuation). Ordinary calculus of continuations can be viewed as a calculus of composable continuations in which all the jumps have a unique destination point, predefined to be at the beginning of the program. In both calculi, continuations are functions whose range type is equal to the type of the destination point. But, in the special case of ordinary continuations, this type is necessarily $\bot$, and that is why ordinary continuations cannot be composed in any non-trivial way.

The typing judgment of the calculus for composable continuations is again $\Sigma;\Delta \vdash e : A\,[C]$. It establishes that the expression e has type A and may recall the destination points whose names are listed in the support C. The typing rules for composable continuations are presented in Figure 4.

In the case of composable continuations, it is a **recall** to a name that is the notion of effect, and **mark**-ing a name as a destination point is the notion of handling. Therefore, the type system should enable a **recall** to X only if X appears at the support C, placed there by a corresponding **mark**. The situation, however, is a bit more involved. As already mentioned, $\mathbf{recall}_X\ k.\ e$ evaluates e in a changed environment from which the part enclosed between $\mathbf{mark}_X$ and $\mathbf{recall}_X$ has been removed. Correspondingly, e has to be checked against a support from which X has been removed.

The above argument indicates that in the calculus of composable continuations, the ordering of names in the support of a term is important. Unlike in the previous calculi where supports were simply sets, here we actually must endow supports with a list-like structure. For example, we allow a **recall** to a certain name only if that name is at the end of the support. This is illustrated in the typing rule for $\mathbf{recall}_X\ k.\ e$, where we demand that X is the *rightmost* name in the support (C,X). If a recall is required to a name which is deeper to the left in C, it can still be done by performing a sequence of nested recalls in a last-in-first-out manner to all the names in between. In this sense, the supports of the calculus of composable continuations may be seen as *stacks*, where the top of the stack is at the rightmost end of the support.

There are yet further important aspects of the typing rule for **recall** that need to be explained. The expression e computes the value to be passed along with the jump, so it must have the same type as the destination point X. Because the jump changes the flow of control, the immediate environment of the **recall** does not matter; we can type **recall** by an arbitrary type B. The domain and the range of the continuation k must match the source and the destination points of the jump, which in this rule have types B and A, respectively. The **recall** appears in the context of a support (C,X) and that is why the

$$\frac{\Sigma;(\Delta, k{:}\Box_{C,X}B \to \Box_{C,X}A) \vdash e : A[C] \quad X{:}A \in \Sigma}{\Sigma;\Delta \vdash \mathbf{recall}_X\ k.\ e : B[C,X]}$$

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

Figure 4. Typing rules for composable continuations.

domain type of k is $\Box_{C,X}B$. The range type of k is $\Box_{C,X}A$, meaning that the environment captured in k *will not include* the delimiting $\mathbf{mark}_X$.

The typing rule for **mark** is much simpler. The construct $\mathbf{mark}_X\ e$ establishes a destination point X and allows the expression e to recall to X by placing X in the support. If e is a value, it immediately falls through to the destination point X, and thus e and X must have same types. We further allow an arbitrary weakening of supports in the conclusion of this rule, in order to satisfy the support weakening principle.

The partial ordering imposed on the family of supports is the trivial partial ordering with the empty stack as the smallest element: $C \sqsubseteq D$ holds iff $C = (\cdot)$ or $C = D$ as sequences.

Example 12 The program below is a particularly convoluted way of reversing a list, adopted from [3].

```
fun reverse (l : intlist) : intlist =
 choose (νX: intlist.
  let fun rev′ (l : intlist) : □Xintlist =
      case l
        of nil => box nil
      | (x::xs) =>
          let val y = rev′ xs
          in
              box (recallX c:□Xintlint -> □Xintlist.
                    markX x :: unbox (c y))
          end
      box v = rev′ l
  in
     markX v
  end)
```

To understand `reverse`, it is instructive to view a particular evaluation of the helper function `rev′`. For example, `rev′ [2, 1, 0]` produces

```
box (recallX c3.
      markX 2 :: unbox c3 (box recallX c2.
        markX 1 :: unbox c2 (box recallX c1.
          markX 0 :: unbox c1 (box nil))))
```

When prepended by a `markX`, unboxed and evaluated, this code uses the continuations c_i to accumulate the reversed prefix of the list. For example, c_3 is bound to $\lambda x.$ **let box** $u = x$ **in box** u corresponding to the initial empty prefix; c_2 is bound to $\lambda x.$ **let box** $u = x$ **in box** $(2 :: u)$; c_1 is bound to $\lambda x.$ **let box** $u = x$ **in box** $(1 :: 2 :: u)$, until finally the reversed list `[0,1,2]` is produced.

There is actually a bit of a leeway in defining the static and dynamic semantics for composable continuations, which has to do with whether the continuation captured by **recall** should include the delimiting **mark** and/or remove it from the environment. The reduction rule we have used in our formulation is

$$\Sigma, (\mathbf{mark}_X\ P[\mathbf{recall}_X\ k.\ e]) \longrightarrow \Sigma, [K/k]e,$$
$$\text{where } K = \lambda x.\ \mathbf{let\ box}\ u = x\ \mathbf{in\ box}\ P[u]$$

As can be seen, this reduction removes **mark** both from the captured continuation K, and from the evaluation context of the reduced term. But either of the following rules is a possible choice, and we discuss them below.

$$\Sigma, (\mathbf{mark}_X\ P[\mathbf{recall}_X\ k.\ e]) \longrightarrow \Sigma, [K/k]e, \qquad (1)$$
$$\text{where } K = \lambda x.\ \mathbf{let\ box}\ u = x\ \mathbf{in\ box}\ (\mathbf{mark}_X\ P[u])$$
$$\Sigma, (\mathbf{mark}_X\ P[\mathbf{recall}_X\ k.\ e]) \longrightarrow \Sigma, \mathbf{mark}_X\ [K/k]e, \qquad (2)$$
$$\text{where } K = \lambda x.\ \mathbf{let\ box}\ u = x\ \mathbf{in\ box}\ P[u]$$
$$\Sigma, (\mathbf{mark}_X\ P[\mathbf{recall}_X\ k.\ e]) \longrightarrow \Sigma, \mathbf{mark}_X\ [K/k]e, \qquad (3)$$
$$\text{where } K = \lambda x.\ \mathbf{let\ box}\ u = x\ \mathbf{in\ box}\ (\mathbf{mark}_X\ P[u])$$

The rule (1) captures $\mathbf{mark}_X$ into K, but removes it from the evaluation environment of e. The typing rule matching this operational semantics is

$$\frac{\Sigma;(\Delta, k{:}\Box_{C,X}B \to \Box_{C}A) \vdash e : A[C] \quad X{:}A \in \Sigma}{\Sigma;\Delta \vdash \mathbf{recall}_X\ k.\ e : B[C,X]}$$

Because the mark X is removed from the environment, it becomes impossible for e to recall to X. This is why X does not appear in the support of the premise of this typing rule. Because the mark X is captured into the continuation, the result of applying the continuation does not require a mark for X in its evaluation environment, and so X is also dropped from the range type of k.

The rule (2) omits the mark from the continuation K, but leaves it in the evaluation environment of e. The corresponding typing rule leaves X in the support of the premise and in the range type of k.

$$\frac{\Sigma;(\Delta, k{:}\Box_{C,X}B \to \Box_{C,X}A) \vdash e : A[C,X] \quad X{:}A \in \Sigma}{\Sigma;\Delta \vdash \mathbf{recall}_X\ k.\ e : B[C,X]}$$

Because the mark is left in the evaluation environment, it becomes impossible to jump in sequence to names that are further down in the support stack. In this setting, it becomes necessary to consider semantics that allow jumps arbitrarily deep into the support stack. This is very related to the behavior of Felleisen's $\mathcal{F}$ operator [8]. If we label by D the top of the support stack, up to but not including the target mark, then a recall which would jump over the names in D will be typed as follows.

$$\frac{\Sigma;(\Delta, k{:}\Box_{C,X,D}B \to \Box_{C,X}A) \vdash e : A[C,X] \quad X{:}A \in \Sigma \quad X \notin D}{\Sigma;\Delta \vdash \mathbf{recall}_X\ k.\ e : B[C,X,D]}$$

Indeed, because the names from D are captured into the continuation, they must be removed from the range type of k. Support D is also removed from the evaluation environment, and hence must be omitted from the support of the premise.

The rule (3) leaves the mark into both the continuation K and the evaluation environment of e, and the typing rule for it is thus

$$\frac{\Sigma;(\Delta, k{:}\Box_{C,X}B \to \Box_{C}A) \vdash e : A[C,X] \quad X{:}A \in \Sigma}{\Sigma;\Delta \vdash \mathbf{recall}_X\ k.\ e : B[C,X]}$$

This choice of semantics corresponds to Danvy and Filinski's **shift** operator [3, 4, 5]. If only jumps to the last established mark are allowed (as is the case in [3, 4]), then it may be possible to simplify

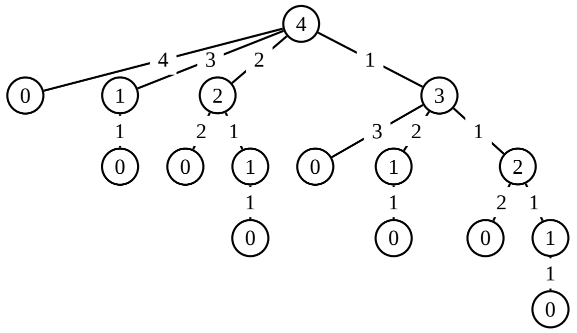

Figure 5. Partition tree for $n = 4$.

the typing rules so that the modal types only record the top-most name in the support.

Our choice of operational semantics for composable continuations is similar to the one for the **set**/**cupto** operators of Gunter, Rémy and Riecke [12]. We have decided on this choice of operational semantics for composable continuations because all the other choices can be encoded within it. Obviously, if the mark is discarded during reduction, it can always be placed back; if it is retained, it can never be eliminated. We do not know if the other operational semantics can match this expressiveness.

Example 13 Composable continuations have been used to conveniently express "nondeterministic computation"; that is, computation which can return many results [3, 4]. The following example is a program for finding all the partitions of a natural number *n*, i.e. all the lists of natural numbers that add up to *n*. The main function `partition` is very effectively phrased in terms of a primitive function `choice`. The idea is to use `choice` to non-deterministically pick a number between 1 and *n*, and not worry about backtracking and exploring other options. Backtracking is automatically handled by `choice`.

```
fun partition n =
  if n = 0 then box (nil)
  else
      box (let val i = unbox (choice n)
               box l = partition (n - i)
           in
              (i::l)
           end)
```

The important point is that `choice` itself can be implemented using composable continuations. The way `choice` is implemented will determine the ordering in which `partition` considers the candidate lists for partitioning *n*.

The process of generating partitions for *n* may be seen as a traversal of a tree with labeled nodes and edges – a partition tree. Paths in the partition tree emanating from a node labeled by *n* represent the partitions of *n*. An inductive definition of the partition tree for *n* is given as follows:

(i) if $n = 0$, then the tree consists of a single node labeled 0.

(ii) if $n > 0$, then the root of the tree is labeled with *n*, and edges labeled with $n, n-1, \dots, 1$ connect the root to partition trees for $0, 1, \dots, n-1$, respectively.

An example partition tree for $n = 4$ is presented in Figure 13. Of course, just as with any tree, various traversal strategies may be employed to generate the partitions for *n*. For example, a *depth-first strategy* may employ a *stack* *k* to store the nodes that remain to be traversed. After putting the root node on the stack, the depth-first strategy repeats the following algorithm: remove the top node *t* from *k*, and *expand* it, i.e. determine all the children of *t* (if any), and put them onto the *top* of *k*; if *k* is empty, then exit.

On the other hand, a *breadth-first strategy* may employ a *queue* *k* to store the nodes that remain to be traversed. After putting the root node on the queue, the breadth-first strategy repeats the following: remove the top node *t* from *k*, and *expand* it, i.e. determine all the children of *t* (if any), and put them at the *bottom* of *k*; if *k* is empty, then exit.

In our implementation of the partition algorithm, the partition tree for *n* is never explicitly built, but is implicitly described by the execution of the `partition` function. For example, we present below a version of `choice` which facilitates a depth-first traversal of the tree. It assumes a name *X* of unit type.

```
(* choice : int -> □_X int *)
fun choice n =
    box (recall_X t : □_X int -> □_X unit.
       let fun loop (s:int):unit =
           if s = 0 then ()
           else
              let box u = t (box s)
              in
                  (mark_X u); loop (s - 1)
              end
       in
           loop (n)
       end)
```

The program works by viewing the current global program continuation as an implicit stack *k* of nodes to be expanded in order. Each node has its own composable continuation, all of which *compose* to create *k*. The function `choice` simply captures into *t* the composable continuation for the first node in the sequence. The captured node is *removed*, and *t* is applied to generate all of its children – one child for each possible value of the variable *s*. The children nodes are added in place of the parent node at the top of the global program continuation *k*. Because the new nodes are added to the beginning, they will be the the first to expand in the subsequent execution. As a consequence, this implementation of `choice` uses *depth-first traversal strategy*.

With this version of `choice`, `partition` has the type `int -> □`$_X$`intlist`. To compute the partitions for 4, we run `mark`$_X$ `print (unbox partition 4)`. The result consists of the lists [4], [3, 1], [2, 2], [2, 1, 1], [1, 3], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]. Because depth-first traversal is employed, the lists are sorted in lexicographical order.

In our calculus, it is also possible to implement `choice` so that it facilitates breadth-first strategy. When generating the children of some node, we only need to attach them at the end, rather than at the beginning of the queue *k* that the global continuation represents. One possible breadth-first implementation of `choice` is given below.

```
(* choice : int -> $\Box_{Y,X}$int *)
fun choice n =
  box (recall$_X$ t : $\Box_{Y,X}$int -> $\Box_{Y,X}$unit.
           recall$_Y$ k : $\Box_Y$unit -> $\Box_Y$unit.
       mark$_Y$
          let fun loop (s : int) : $\Box_Y$unit =
               if s = 0 then box ()
               else
                  let box u = t(box s)
                      box u' = loop(s-1)
                  in
                      box (mark$_X$ u; u')
                  end
              box v = k (box mark$_X$())
              box v' = loop n
          in
              v; v'
          end)
```

How does this function work? First, we must assume that the queue is marked by a new name Y of unit type, so that it can be captured into a continuation itself. The function `choice` captures the topmost node into t, and then captures the rest of the queue into k. It is important that the continuation k will not contain the delimiting $\mathbf{mark}_Y$. Then `choice` expands the topmost node t, adds the obtained children nodes to the bottom of k, and puts $\mathbf{mark}_Y$ back, so that its scope includes the children nodes. Again, it is crucial for this application that the captured continuations omit the target mark (unlike, for example, in the calculi from [3, 4]), as this mark will get in the way of adding new nodes at the bottom of k.

With this implementation of `choice`, the appropriate type for the function `partition` is `int->`$\Box_{Y,X}$`intlist`. To compute the partitions for 4, we run `mark`$_Y$ `mark`$_X$ `print (unbox partition 4)` to obtain the lists [4], [3, 1], [2, 2], [1, 3], [2, 1, 1], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]. Because we used breadth-first traversal strategy, we first explored all the partitions of size 1, then all the partitions of size 2, etc. Thus, the lists will be sorted first by size, rather than lexicographically, as was the case with depth-first traversal.

The structural properties from the core fragment readily extend with the new cases characteristic to the calculus of composable continuations.

LEMMA 13 (UNIQUE DECOMPOSITION). *For every expression e, either:*

1. *e is a value, or*
2. *$e = P[\mathbf{recall}_X\ k.\ e']$, for a unique pure context P, or*
3. *$e = E[r]$ for a unique evaluation context E and a redex r.*

The calculus satisfies the same preservation and progress theorems as before.

8 Conclusions, related and future work

In this paper, we have used the necessitation type operator $\Box$ from the modal logic CS4 to internalize computations with control-flow effects like exceptions, catch-and-throw and composable continuations. 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 approach of our calculi is based on the following observation. A computations of type A that may cause control effects from the list C, may 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. In our application, we adopt that handlers correspond to worlds in modal logic, and thus, a described effectful computation can be typed with a bounded universal type $\Box_C A$.

Our calculus of exceptions have certain advantages over the monadic representation of exceptions [20, 31] (and similar statements may be formulated about our catch-and-throw calculus). As a first distinction, modal calculi 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.

Similar consideration lead to our modal formulation of the calculus for composable continuations. It is particularly interesting that the calculi we presented here share one and the same core fragment, and only differ in the notion of support. Supports in the case of exceptions and catch-and-throw are simply sets of names, while in the case of composable continuations supports are stacks of names. It remains future work to determine how to combine effects which are heterogeneous in their notion of support.

There is extensive literature on the non-monadic treatment of exceptions [7, 12, 24, 16, 22]. In general, however, the considered calculi typically do not internalize the notion of exceptional computation. As a consequence, they usually weaken the type safety, either by imposing restrictions on the scope of exceptions and/or on the form of values, or by allowing well-typed terms to get stuck because of unhandled effects.

In Section 7 we already informally compared our calculus of composable continuation with the proposals of Felleisen, and Danvy and Filinski. There are also further proposals, like [13, 14, 15, 28], and especially [12] by Gunther, Rémi and Riecke to which our calculus is very similar. Again, as in the case of exceptions, these systems typically do not internalize effectful computations, and hence have to impose restrictions on expressiveness and type safety. A monadic approach to composable continuations have been attempted by Murthy in [21], but this calculus restricts the destination marks to only implication-free types in order to preserve soundness. Wadler in [30] further analyses and compares the above type systems using indexed monads.

In our paper, the various notions of control effects are uniformly represented by names. Names are labels which 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 systems we presented here. It is motivated by the work of Pitts and Gabbay on Nominal Logic and FreshML [27, 26] which introduce names as urelements of Fraenkel-Mostowski set theory.

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.

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

[3] O. Danvy and A. Filinski. A functional abstraction of typed contexts. Technical Report 89/12, DIKU - Computer Science Department, University of Copenhagen, 1989.

[4] O. Danvy and A. Filinski. Abstracting control. In *Conference on LISP and Functional Programming*, pages 151–160, Nice, France, 1990.

[5] O. Danvy and A. Filinski. Representing Control: a Study of the CPS Transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

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

[7] 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.

[8] M. Felleisen. The theory and practice of first-class prompts. In *Symposium on Principles of Programming Languages, POPL'88*, pages 180–190, San Diego, California, 1988.

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

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

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

[12] C. A. Gunter, D. R´emy, and J. G. Riecke. A generalization of exceptions and control in ML-like languages. In *International Conference on Functional Programming Languages and Computer Architecture, FPCA'95*, pages 12–23, La Jolla, California, 1995.

[13] R. Hieb, K. Dybvig, and C. W. Anderson, III. Subcontinuations. *Lisp and Symbolic Computation*, 7(1):83–110, 1994.

[14] Y. Kameyama. Towards logical understanding of delimited continuations. In A. Sabry, editor, *Proceedings of the Third ACM SIGPLAN Workshop on Continuations, CW'01*, pages 27–33, 2000. Technical Report No. 545, Computer Science Department, Indiana University.

[15] Y. Kameyama. A type-theoretic study on partial continuations. In J. van Leeuwen, O. Watanabe, M. Hagiya, P. D. Mosses, and T. Ito, editors, *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics*, volume 1872 of *Lecture Notes in Computer Science*, pages 489–504. Springer, 2000.

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

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

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

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

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

[21] C. R. Murthy. Control operators, hierarchies, and pseudo-classical type systems: A-translation at work. In O. Danvy and C. Talcott, editors, *Proceedings of the ACM SIGPLAN Workshop on Continuations, CW'92*, pages 49–71, 1992. Technical Report STAN-CS-92-1426, Stanford University.

[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] 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.

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

[26] 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.

[27] 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.

[28] D. Sitaram. Handling control. In *Conference on Programming Language Design and Implementation, PLDI'93*, pages 147–155, 1993.

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

[30] P. Wadler. Monads and composable continuations. *Lisp and Symbolic Computation*, 7(1):39–56, 1994.

[31] 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.

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

[33] 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.

[34] 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.

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